©Copyright 1999-2008 Telekinesys Research Ltd. (t/a Havok). All rights reserved. [1]

[1] Havok.com and the Havok buzzsaw logo are trademarks of Havok. All other trademarks contained herein are the properties of their respective owners.

This document is protected under copyright law. The contents of this document may not be reproduced or transmitted in any form, in whole or in part, or by any means, mechanical or electronic, without the express written consent of Havok. This document is supplied as a manual for the Havok game dynamics software development kit. Reasonable care has been taken in preparing the information it contains. However, this document may contain omissions, technical inaccuracies, or typographical errors. Havok does not accept any responsibility of any kind for losses due to the use of this document. The information in this document is subject to change without notice.

Contents

1 Introduction **7**

2 Common Havok Components **9**
2.1 Havok Base Library . . . 10
2.1.1 Introduction . . . 10
2.1.2 The Base System . . . 10
2.1.3 Container Classes . . . 15
2.1.4 Havok Singletons . . . 17
2.1.5 Memory Management . . . 19
2.1.6 Error Reporting . . . 39
2.1.7 Monitoring and Timing . . . 43
2.1.8 Working with Streams . . . 47
2.2 Havok Math Library . . . 53
2.2.1 Introduction . . . 53
2.2.2 Math Types . . . 53
2.2.3 Using the Maths Library . . . 54
2.2.4 Common Conventions . . . 57
2.2.5 SIMD and Non SIMD support . . . 57
2.2.6 Optimisation . . . 58
2.3 Serialization . . . 60
2.3.1 Introduction to Serialization . . . 60
2.3.2 Features . . . 60
2.3.3 Packfiles . . . 61
2.3.4 Loading Game Data . . . 62
2.3.5 Saving Game Data . . . 65
2.3.6 Type Registration . . . 66
2.3.7 Memory Considerations . . . 67
2.3.8 Havok Tools and Data . . . 70
2.3.9 Serializing Havok Physics Objects . . . 71
2.3.10 Snapshot Utility . . . 75
2.3.11 Using the Infrastructure . . . 77

3 Havok Physics **83**
3.1 Introduction . . . 84
3.1.1 Introduction . . . 84
3.1.2 What is Havok? . . . 84
3.1.3 How it works . . . 86
3.1.4 About the documentation . . . 89
3.1.5 About the demos . . . 90
3.2 Havok Dynamics . . . 92
3.2.1 Introduction . . . 92
3.2.2 The Havok World . . . 92
3.2.3 Rigid Bodies . . . 95

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.2.4 Stepping the simulation forward . . . 108
3.2.5 Phantoms . . . 115
3.2.6 Scale . . . 122
3.2.7 Constraints . . . 123
3.2.8 Constraint Chains . . . 156
3.2.9 Actions . . . 161
3.2.10 Listeners . . . 170
3.2.11 Postponing of Operations . . . 173
3.2.12 Simluation Determinism . . . 174
3.2.13 Memory Issues . . . 175
3.2.14 Saving Contact Points . . . 178
3.3 Collision Detection . . . 181
3.3.1 Introduction . . . 181
3.3.2 Overview . . . 181
3.3.3 Collidables . . . 184
3.3.4 Creating Shapes . . . 185
3.3.5 Broadphase Collision Detection . . . 209
3.3.6 Narrowphase Collision Detection . . . 211
3.3.7 Collision filtering . . . 214
3.3.8 Removing child shapes from compound shapes . . . 219
3.3.9 Welding . . . 219
3.3.10 Optimizing and Tuning Collision Detection . . . 221
3.3.11 Offline generation of data . . . 225
3.3.12 Getting Collision Information . . . 229
3.3.13 Contact modifiers . . . 240
3.3.14 Raycasting . . . 242
3.3.15 Linear Casting . . . 244
3.3.16 Introduction to Customization . . . 245
3.3.17 Summary . . . 247
3.4 Continuous Physics . . . 248
3.4.1 Introduction . . . 248
3.4.2 Fundamental Concepts . . . 248
3.4.3 Priorities and Quality Types . . . 256
3.4.4 Performance Optimizations . . . 261
3.4.5 Asynchronous Simulation . . . 265
3.4.6 Advantages of Continuous Simluation . . . 266
3.4.7 Limitations of Continuous Simulation . . . 267
3.4.8 Overview - Basic Parameters . . . 271
3.4.9 Overview - Advanced Parameters . . . 272
3.4.10 Troubleshooting . . . 272
3.4.11 Demo Walkthroughs . . . 273
3.5 Multithreading . . . 277
3.5.1 Introduction . . . 277
3.5.2 Traditional Single Threaded Approach . . . 277
3.5.3 Running graphics in a separate thread from the main game loop . . . 278
3.5.4 Running physics in a separate thread from the main game loop . . . 278
3.5.5 Splitting Physics . . . 280
3.5.6 Running Physics Multithreaded . . . 281
3.5.7 Memory Manager . . . 287
3.5.8 Synchronization Issues . . . 288
3.6 Character Control . . . 291
3.6.1 Introduction . . . 291
3.6.2 Overview . . . 291
3.6.3 The State Machine . . . 293

3.6.4 The Character Controller Behavior . . . 296
3.6.5 The Character Rigid Body . . . 302
3.6.6 The Character Proxy . . . 304
3.6.7 The Simplex Solver . . . 312
3.6.8 The Character Controller Examples and Use Cases . . . 314
3.7 Vehicle Physics . . . 316
3.7.1 The Havok Vehicle Kit . . . 316
3.7.2 Vehicle Programming Guide . . . 336
3.8 Using the Visual Debugger . . . 340
3.8.1 Introduction . . . 340
3.8.2 Quick Start Guide . . . 341
3.8.3 The Visual Debugger Game Side . . . 341
3.8.4 The Visual Debugger Application . . . 355
3.8.5 Visual Debugger Menu Options . . . 365
3.8.6 Preparing a visual debugger movie for a support query . . . 367
3.8.7 Platform-specific issues . . . 367
3.9 Miscellaneous . . . 368
3.9.1 Performance Tips and Common Errors . . . 368
3.10 Physics Articles . . . 372
3.10.1 Physics Primer . . . 372
3.10.2 Rotations, Handedness, and all that. . . . 393

4 Havok Animation 425
4.1 Introduction . . . 426
4.2 Architecture Overview . . . 427
4.2.1 Object Overview / Glossary . . . 427
4.2.2 Export Path . . . 430
4.3 Animation Runtime . . . 432
4.3.1 Animation Features . . . 432
4.3.2 Animation Runtime Modules . . . 433
4.3.3 Dataflow Example . . . 434
4.3.4 World, Model and Local Spaces . . . 436
4.3.5 Using hkQsTransform . . . 438
4.3.6 Animation Representation . . . 445
4.3.7 Rig Information in Havok Animation . . . 448
4.3.8 Playback . . . 449
4.3.9 Motion Extraction / Locomotion . . . 454
4.3.10 Mirroring Animations . . . 466
4.3.11 Skeleton Mappers . . . 476
4.3.12 Animation Compression . . . 486
4.3.13 Caching Decompressed Animation Data (Wavelet and Delta Compression) . . . 495
4.3.14 Deformation support . . . 500
4.3.15 Inverse Kinematics . . . 502
4.3.16 Using hkaPose . . . 519
4.4 Animation Examples and Use Cases . . . 531
4.4.1 GDC 2005 demo . . . 531

5 Physics and Animation Integration 537
5.1 Integrating Havok Animation with Havok Ragdolls . . . 538
5.1.1 Introduction . . . 538
5.1.2 A Typical Scenario . . . 538
5.1.3 Ragdoll Instances . . . 542
5.1.4 Mappers . . . 544
5.1.5 Ragdoll Controllers . . . 545
5.1.6 The Detect Ragdoll Penetration Utility . . . 547

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6 Havok Content Tools **553**
6.1 Introduction and Architecture . . . 554
6.1.1 Introduction . . . 554
6.1.2 Architecture . . . 555
6.2 The Havok Filter Pipeline . . . 558
6.2.1 The Filter Manager . . . 558
6.2.2 Core Filters . . . 564
6.2.3 Physics Filters . . . 570
6.2.4 Animation Filters . . . 577
6.2.5 Graphics Filters . . . 594
6.3 3ds Max Tools . . . 602
6.3.1 Introduction . . . 602
6.3.2 3ds Max: Scene Exporter . . . 602
6.3.3 3ds Max: Physics Tools . . . 605
6.3.4 3ds Max: Animation Tools . . . 625
6.3.5 Tutorial: Export and Animation Basics . . . 625
6.3.6 Tutorial: Physics Basics . . . 639
6.3.7 Tutorial: More on Rigid Bodies . . . 655
6.3.8 Tutorial: Rag Doll Toolbox . . . 670
6.4 Maya Tools . . . 687
6.4.1 Introduction . . . 687
6.4.2 Maya: Scene Exporter . . . 689
6.4.3 Maya: Physics Tools . . . 692
6.4.4 Maya: Animation Tools . . . 711
6.4.5 Tutorial: Export and Animation Basics . . . 714
6.4.6 Tutorial: Physics Basics . . . 728
6.4.7 Tutorial: More on Rigid Bodies . . . 744
6.4.8 Tutorial: Rag Doll Setup . . . 754
6.5 XSI Tools . . . 769
6.5.1 Introduction . . . 769
6.5.2 XSI: Scene Exporter . . . 769
6.5.3 XSI: Physics Tools . . . 778
6.5.4 XSI: Animation Tools . . . 796
6.5.5 Tutorial: Export and Animation Basics . . . 799
6.5.6 Tutorial: Physics Basics . . . 813
6.5.7 Tutorial: More on Rigid Bodies . . . 827
6.5.8 Tutorial: Rag Doll Toolbox . . . 837
6.6 Common Concepts . . . 858
6.6.1 Introduction . . . 858
6.6.2 Physics: Rigid Body Concepts . . . 858
6.6.3 Physics: Constraint Concepts . . . 864
6.6.4 Animation: Annotations . . . 883
6.6.5 Animation: Controlling Compression . . . 884
6.7 Integrating with the Havok Content Tools . . . 889
6.7.1 Introduction . . . 889
6.7.2 Offline Processing . . . 889
6.7.3 The Havok Scene Data Format . . . 892
6.7.4 Extending the 3ds Max Tools . . . 902
6.7.5 Extending the Maya Tools . . . 910
6.7.6 Extending the XSI Tools . . . 914
6.7.7 Writing your own Filters . . . 923
6.7.8 Customizing your Installation Method . . . 931
6.7.9 Tutorial : Extending the Toolchain . . . 933
6.8 Troubleshooting . . . 978

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.8.1 Troubleshooting: Processing Assets 978
6.8.2 Troubleshooting: 3ds Max Tools 984
6.8.3 Troubleshooting: Maya Tools 987
6.8.4 Troubleshooting: XSI Tools 989
6.8.5 Troubleshooting: Toolchain Integration 989

Chapter 1

Introduction

Welcome to the Havok Game Dynamics SDK Manual! The Havok Game Dynamics SDK is the fastest, most flexible, cross-platform game dynamics solution available on the market today.

The Havok team has one focus - building game-specific solutions that work. That focus has made Havok the No.1 provider of middleware physics to the games industry today. Havok delivers great physics: from spectacular ragdoll effects to large scale interactive environments. Not only that, but Havok also provides you with a flexible, high-performance core collision detection engine that can be used to dramatically enhance the performance of your game.

This manual will guide you through using the Havok Game Dynamics SDK and answer many of the questions you have about using the SDK. The manual is divided into several sections:

Common Havok Components

The Havok infrastructure contains elements common to both the Havok Physics and Havok Animation products. This section covers the Havok base and maths libraries in detail, and explains the fundamentals of Havok Serialization.

Havok Physics

The Havok Physics Manual describes in detail the architecture and concepts used in Havok. Some parts of this are required reading for understanding how to work with the Havok Physics SDK - in particular the Dynamics and Collision Detection chapters.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Havok Animation

The Havok Animation Manual explains everything you'll need to know about using Havok Animation in your games, such as the animation runtime.

Physics and Animation Integration

This section describes how to integrate Havok Physics with Havok Animation. Particular focus is given to integrating ragdoll physics with character animation.

Havok Content Tools

This section describes how to use the Havok Content Tools (a set of tools for content creation, export and processing) to create and process assets for use in the Havok SDK

Reference Manual

The reference manual documents all the modules in the Havok system, on a module by module basis. For each module, a list of classes is presented, together with a hierarchical list of the classes in that module. For each class, an inheritance diagram is provided in the class description.

You should start with the QuickStart guide - when you get the demos up and running it's easier to start working through the rest of the manual. We hope you enjoy working with the Havok Game Dynamics SDK as much as we have enjoyed developing it!

Throughout the course of this manual references to Havok's source code may be made. Havok CompleteXs users should note that their product includes source code for Havok's demo suite, but not for other areas of the sdk.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Chapter 2

Common Havok Components

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

2.1 Havok Base Library

2.1.1 Introduction

The Havok base library - hkBase - provides functionality common to all Havok libraries. This includes

- A platform-independent interface to system services, such memory management and timing
- Basic data types and container classes

Havok has been specifically designed to allow you to customize its default behavior so that it matches how your existing environment functions. Here, we take a detailed and practical look at how to hook in to all of the integration points in the Havok Base Library. We will also show how default behavior can be extended or replaced for many of these system functions.

2.1.2 The Base System

2.1.2.1 Configuring and customizing hkBase

All platforms use common hkBase header files. The platform-specific code is out of line so that you can change the implementation without affecting the rest of Havok. hkBase is the only Havok library that includes non-Havok headers.

The default hkBase implementations are fully usable without modification. Alternatively, you can make your own changes or additions to the provided source code. Havok provides a simple mechanism for updating and replacing many of the hkBase classes, which you can find out more about in the Singletons section of this chapter.

Base root directory

This contains `hkBase.h`, which includes all the other hkBase headers.

Non-replaceable code

Some parts of hkBase are compiled into the Havok libraries, and are not replaceable without a complete rebuild of all libraries. These include

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- the container classes, which have inline methods
- the build configuration options in the `config` directory

In general, none of the hkBase header files should be modified.

Compiler Warnings

An important function of hkBase.h is to enable and disable compiler warnings. Havok has determined that some level 4 compiler warnings in the Visual Studio compilers (.NET and Visual C++ 6) can be disabled without affecting performance. Disabling these warnings results in better code readability and class design, without compromising performance. The warnings are disabled in hkbase/hkBase.h. Please note that as almost all Havok headers include hkbase/hkBase.h, if any Havok header is included in a source file you will be disabling certain warnings for that source file. If you require these warnings in non Havok code it is recommended that you #pragma reset or re enable the warnings as necessary.

Visual Studio Warnings

The following level 4 compiler warnings are disabled in hkBase.h under the Visual Studio Compilers. These compilers support the push/pop pragma control. Before any of the following pragma warnings are disabled, a pragma push is issued. The push stores the current warning state so that it can be reverted to easily using a pragma pop anytime after these warnings have been enabled .

- C4100 'identifier' : unreferenced formal parameter.
  A function need not use all its arguments
- C4127 conditional expression is constant.
  Constant conditionals are often used inside asserts
- C4505 'function' : unreferenced local function has been removed.
  Lots of inline functions are not used in a compilation unit
- C4510 'class' : default constructor could not be generated
- C4511 'class' : copy constructor could not be generated
- C4512 'class' : assignment operator could not be generated.
  Many classes are not designed with value semantics
- C4514 unreferenced inline/local function has been removed.
  Lots of inline functions are not used in a compilation unit
- C4714 force inlined function not inlined.
  This warning is only disabled in debug modes.

Metrowerks Warnings

Havok does not explicitly disable or enable any warnings for Metrowerks Codewarrior. Instead Havok code is manually tested for certain Codewarrior warnings prior to each release. This test includes most warnings in the project settings panel with the exception of the warning for unused arguments which is ignored. Due to Havok's class design it is typical for some methods to have parameters but no implementation. Havok recommends that this warning is disabled when including Havok headers. A

warning that Havok does monitor closely is the warning for implicit conversion for arithmetic operators. Prior to each release Havok attempts to minimize these warnings in its codebase.

Compile-time configuration

The `config` directory contains the compile time settings used when building all the Havok libraries - these include global debug output options, and whether to use the memory manager.

Note that any changes you make to this information will require a *full rebuild* of Havok. Changing configuration values with compiled libraries will not work.

2.1.2.2 Havok Types

The hkBaseTypes class provides you with some useful macros and basic types.

Platform information

hkBaseTypes defines a number of macros based on the platform information obtained from the compiler. These are

- HK_COMPILER_*: the compiler
- HK_COMPILER_*_VERSION: the compiler version
- HK_ARCH_*: the CPU
- HK_PLATFORM_*: the operating system

These definitions, in turn, are used to define some further platform-specific macros in hkBaseTypes.

Basic types

hkBaseTypes provides some Havok-specific basic types, including floating-point types and unsigned and signed integers. The default Havok floating-point type is hkReal.

Aligned variables

hkBaseTypes defines macros that allow you to specify an alignment in memory when declaring a variable, as in the following example:

```
HK_ALIGN16( hkReal myVar );
```

These macros are platform-dependent, and are defined based on the platform information macros.

2.1.2.3 Havok Initialization

The hkBaseSystem is responsible for creating all of Havok's subsystems. Many of these are singletons and have to be initialized in a specific order, for example: memory management, error handling, stream

handling. You can find out more about how hkSingletons are created in the next section.

The primary interface to this class is through an `init` and `quit` method. `init` *must* be called to initialize Havok's subsystems. The method takes a number of parameters:

```
static hkResult HK_CALL init( hkMemory* memoryManager,
                              hkThreadMemory* threadMemory,
                              hkErrorReportFunction errorReportFunction
                              void* errorReportObject = HK_NULL );
```

The *memoryManager* is the memory management implementation that Havok will use internally. This interface allows you to specify the memory manager during initialization and no rebuild of hkBase is necessary. Havok provides a number of default memory management implementations that you can use. Although no default parameter is provided, unless you have your own memory manager, Havok recommends using hkPoolMemory. Implementations for this memory manager and the others available can be found in `<hkbase/memory/impl>`. For more information on Havok's memory management system please see the Memory Management section.

The `threadMemory` is a memory manager local to the containing thread. It may be used to optimize frequent memory allocations and deallocations. You can pass an HK_NULL for this parameter and the `hkBaseSystem::init()` call will create a `hkThreadMemory` instance for you.

The *errorReportFunction* and *errorReportObject* are used by Havok's error handler. The error handler is responsible for dealing with any asserts, errors, warnings or reports that occur in the engine. The default error handler is `<hkbase/hkerror/hkDefaultError.h>` it uses *errorReportFunction* to print its messages. A typedef is defined in `hkBaseSystem.h` that allows a basic *errorReportFunction* such as *printf* to be used. It may be neccessary to wrap printf with a stub function in case a '%s', for example, is embedded in the error string. Another alternative to this would be to use std::puts(), if your platform supports it. An example of passing printf wrapped in a small stub function to hkDefaultError is provided below. For details on how to further customise the error reporting methods and use the *errorReportObject* parameter please see the Error Reporting section.

To initialize hkBaseSystem with hkPoolMemory use the following syntax:

```
#include <Common/Base/System/hkBaseSystem.h> // include for hkBaseSystem
#include <Common/Base/Memory/Memory/Pool/hkPoolMemory.h> // hkPoolMemory
extern "C" int printf(const char* fmt, ...); // For printf, used by the error handler

// Stub function to print any error report functionality to stdout
// std::puts(...) could be used here alternatively
static void errorReportFunction(const char* str, void* errorOutputObject)
{
    printf("%s", str);
}

{
...
    hkBaseSystem::init( new hkPoolMemory(), HK_NULL, errorReportFunction );
...
}
```

hkBaseSystem also has a `quit()` function that quits the memory system and destroys all the hkSingleton instances.

If you are using the Havok `hkPoolMemory` memory system, the Havok memory pool is released on `quit()`. Thus it is reasonable to call `init()` and `quit()` several times during a program's execution.

Warning:
Note that as the memory system is not initialized until after `main()` starts, any statically initialized classes that need to allocate memory will fatally fail. In practice, this means that most non-trivial Havok objects cannot be global or static.

2.1.2.4 Initialization in a Multithreaded Mode

Each thread used by Havok has its own hkThreadMemory manager which must be initialized at the begining and destroyed at the end of a thread's life. `hkThreadMemory` may be set up to cache memory blocks upon deallocation and reallocation.

The code below initializes a caching hkThreadMemory instance and uses it for Havok's initialization in its main thread.

```
hkMemory* memoryManager = new hkPoolMemory();
hkThreadMemory threadMemory(memoryManager, 16);
hkBaseSystem::init( memoryManager, &threadMemory, errorReportFunction );
memoryManager->removeReference();
```

Each newly created Havok thread must also create its own hkThreadMemory instance and pass it to the function that initialized the thread's resources. At the end of the thread's life the resources must be freed.

```
DWORD WINAPI MyThread(void *v)
{
  hkThreadMemory threadMemory(&hkMemory::getInstance(), 16);
  hkBaseSystem::initThread( &threadMemory );

  ....

  hkBaseSystem::clearThreadResources();
}
```

2.1.2.5 Managing Your Objects

hkBaseObject

This is the base class for all Havok classes that have virtual functions.

hkReferencedObject and reference counting

hkReferencedObject is a subclass of hkBaseObject. It's used as the base class for Havok objects that can have multiple owners, and facilitates Havok's object management mechanism. Rigid bodies, constraints, and actions are all hkReferencedObjects.

All hkReferencedObjects store size information about themselves for use by the memory manager. You can find out more about how the memory manager works in the Memory section.

A hkReferencedObject has a reference counter that is incremented each time a reference is added to the object, and decremented when the reference is removed. When a hkReferencedObject is created, it automatically has a reference count of 1. The object is deleted from memory when its reference count reaches zero. You can increment and decrement the reference counter using the `addReference()` and `removeReference()` functions.

To see how this works, let's say you create a Havok rigid body in your game code and then add it to the physics engine using the hkpWorld's `addEntity()` method. When the object is created, it has a reference count of 1. This is incremented to 2 by the hkpWorld when the object is added to it. When Havok is finished with the object, the reference count is decremented back to 1. As the reference count is not zero, the object is not deleted from memory and can be reused in your code, thus saving the overhead of creating a new object. So, for example, if you create a piece of debris that falls through and out of the physical scene, you can keep removing and then re-adding the same object to the scene to create more debris.

Alternatively, if you don't want to reuse a hkReferencedObject, you can relinquish ownership of it by calling `removeReference()`, as in the example below.

```
m_world->addEntity(rigidBody);
rigidBody->removeReference();
```

As you know, this decrements the reference count. Because of this, the rigid body's reference count will be reduced to zero when the physics engine is finished with it, and the object will be deleted automatically. You don't have to worry about deleting the object yourself.

You can quickly find out which Havok classes are hkReferencedObjects by checking the hkReferencedObject class hierarchy diagram in the Reference Manual.

Note:
Some classes use reference counting but are not in fact hkReferencedObjects. These are: hkpMoppCode, which you use like any other referenced object; and hkMemory, about which you will find out more later this chapter.

2.1.3 Container Classes

2.1.3.1 Arrays

hkArray is the default Havok array class. It is very similar to the standard STL array, with a number of important differences:

Element types

The hkArray class can be used only with plain old data types since constructors and destructors are not called. You can only use hkArray with objects if they have a trivial constructor, trivial destructor, a copy constructor and an assignment operator that are equivalent to memcpy. Generally, if you need to create an array of objects, you should use an hkObjectArray.

Naming conventions

The hkArray functions use a slightly different naming convention to the STL functions. For example, the corresponding hkArray function to the STL `push_back()` function is called `pushBack()`

Preserving order

The hkArray `removeAt()` function does not preserve the order of the array elements - instead, the last array element is used to replace the removed element, to save the overhead of copying all the subsequent elements back one place in the array. To get the STL behavior when removing an element, you use the `removeAtAndCopy()` function.

Unchecked functions

hkArray has special unchecked versions of `pushBack()`, `setSize()` and `expandBy()`. These are called `pushBackUnchecked()`, `setSizeUnchecked()`, and `expandByUnchecked()`, respectively. With these functions, the hkArray does not check if the array capacity needs to be increased before performing the task. These functions are useful if you have already reserved an appropriate amount of memory using `reserve()`.

Using existing arrays

If you have an existing array that you want to use in Havok, you can use your array to create a new hkArray with the following constructor. The *ptr* parameter is a pointer to your array data.

```
inline hkArray(T* ptr, int size, int capacity);
```

Note that the original array is not copied, just used in place. You should ensure that the data that *ptr* points to is valid for the scope of the hkArray.

Inplace arrays

hkInplaceArray is a subclass of hkArray. An hkInplaceArray has internal storage capacity for its elements within the object itself. The creation of an inplace array is much faster than the creation of an ordinary hkArray since no dynamic allocations occur. They can be very useful in cases where the size of an array is likely to be less than a small fixed size.

When you create an inplace array, you specify an initial size for it. If you subsequently make the array too large for all the elements to be stored within the object, Havok converts your inplace array into an ordinary hkArray.

There is also a hkInplaceArrayAligned16 class. Arrays of this class have a small internal storage capacity, aligned to 16 bytes within the object

Deallocate mask

Each hkArray has an `m_capacityAndDeallocateFlag` member that stores the array's capacity in the lower 31 bits. The highest bit is used to indicate whether the memory storing the original array elements should be deallocated if the array is resized or destroyed. If the highest bit is set, the memory will not be freed. This flag is set, for instance, if you create a hkArray from an existing C style array or hkInplaceArray.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Object arrays

You can use the hkObjectArray class to create arrays of objects which have nontrivial constructors, destructors or assignment methods. The following must be publicly accessible for each element of the array:

- Default constructor, copy constructor, assignment operator, destructor (either hand or compiler generated)
- Placement new, needed to construct objects in place. You may need to use HK_DECLARE_CLASS_ALLOCATOR or HK_DECLARE_NONVIRTUAL_CLASS_ALLOCATOR for structs, nested classes, etc.

hkObjectArray provides similar functions to hkArray, with the exception of the unchecked functions, and the array can not be used for serialization.

Sorting

hkBase provides you with two different classes that you can use to sort the elements of an array - hkHeapSort and hkQuickSort. Each element of the array to be sorted must have the < operator implemented.

The macro hkSort is defined per platform to point to either hkHeapSort or hkQuickSort depending on performance characteristics.

2.1.3.2 Maps

hkPointerMap and hkStringMap

hkPointerMap is a hashing map between pointer sized integers. The hashing function is chosen to perform well for keys which are addresses. hkStringMap is similar, except that the hashing function is suitable for strings. Note that in both cases, the maps hold only the pure key and value pairs - the pointed to objects and strings are not copied.

2.1.4 Havok Singletons

hkSingleton is a utility class used in Havok to define global objects. Singletons are created when `hkBaseSystem::init()` is called, and are deleted automatically when `hkBaseSystem::quit()` is called. You access a singleton using its `getInstance()` method.

Many of the hkBase classes described in this chapter, including hkError and hkStreamBufFactory, are hkSingletons, which facilitates managing and replacing them.

2.1.4.1 Replacing singletons

You can create your own version of any of the hkBase singletons - for example, your own error handling class - by subclassing the appropriate class and implementing the virtual functions. Your new subclass will also be an hkSingleton.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

To use your own custom singleton, you use the hkSingleton's `replaceInstance()` function. This allows you to quickly replace any singleton without recompiling Havok. First you call `init()`, which instantiates all the default singletons. Then you get the instance of the singleton you want to replace, and call its `replaceInstance()` function, passing it your custom singleton. The following snippet shows the default error handler being replaced:

```
hkBaseSystem::init();
hkError::replaceInstance( new myErrorHandler() );
```

2.1.4.2 Instantiating singletons

You can specify that you want an hkSingleton to be instantiated in one of two ways, using macros defined in `hkSingleton.h`. The default is `HK_SINGLETON_IMPLEMENTATION(CLASS)`, which means that the singleton `CLASS` is automatically created when `init()` is called. For example `HK_SINGLETON_IMPLEMENTATION(hkMonitorBank)`. When you create a singleton which implements an interface, use `HK_SINGLETON_CUSTOM_IMPLEMENTATION`.

In some cases you may need to create singletons in a particular order. Specifically when the functionality of one singleton depends on another, e.g. hkError depends on hkMemory. In this case, the singleton classes use `HK_SINGLETON_MANUAL_IMPLEMENTATION`, which specifies that you want to explicitly create them in `init()`. The following snippet shows `init()` function and the order in which the singletons are created.

```
hkResult HK_CALL hkBaseSystem::init(hkMemory* memoryManager, hkErrorReportFunction errorReportFunction,
  void* errorReportObject)
{
        extern hkBool hkBaseSystemIsInitialized;
        if(hkBaseSystemIsInitialized==false)
        {
#if HK_CONFIG_MEMORY_CLASS == HK_CONFIG_MEMORY_CLASS_ENABLED
                if (memoryManager)
                {
                        hkMemory::replaceInstance( memoryManager );
                }
                else
                {
                        HK_BREAKPOINT();

                        // No memory manager system specified.
                        // You must specify a memory manager when initializing hkBaseSystem.
                        // Use hkPoolMemory if no memory manager is present.

                        return HK_FAILURE;
                }
#endif
                hkStreambufFactory::replaceInstance( new hkDefaultStreambufFactory );
                hkError::replaceInstance( new hkDefaultError(errorReportFunction, errorReportObject) );

                initSingletons();
                hkDummySingleton::getInstance().forceLinkage();

                hkBaseSystemIsInitialized = true;
                showHavokBuild();
        }
        return HK_SUCCESS;
}
```

2.1.5 Memory Management

The memory management module in the Havok base library is known as hkMemory. The following sections explain how the memory system works, and how to configure, extend, and profile it.

2.1.5.1 System Heap Allocation

All Havok allocations are routed through an instance of `hkMemory`. Most supplied implementations of this interface in turn interface to the system heap through the `hkSystemMalloc()` and `hkSystemFree()` function pointers defined in memory/hkMemory.cpp. This provides an easy way to reroute all allocations by assigning these pointers before `hkBaseSystem::init` is called. Note as an exception, instances of hkMemory use hkSystemMalloc directly for their instantiation.

2.1.5.2 Allocation Types

The memory system supports two types of allocation: one which is similar to malloc / free where the manager handles the size of memory blocks transparently (hkAllocate / hkDeallocate), and one where the size must explicitly be specified when deallocating (hkAllocateChunk / hkDeallocateChunk). Because

the chunk versions are more prone to error, you are also provided with a memory class parameter for debugging checks. The following snippet shows both types of allocation:

```
char* alphabet = hkAllocate<char>(24, HK_MEMORY_CLASS_STRING);
hkDeallocate(alphabet);

int* tenInts = hkAllocateChunk<int>(10, HK_MEMORY_CLASS_ARRAY);
hkDeallocateChunk(tenInts, 10, HK_MEMORY_CLASS_ARRAY);
```

You will not normally need to use these functions directly - class new and delete are overloaded for all Havok classes, allowing you to use new and delete as usual. However, you may need to use the Havok allocation functions when allocating plain old data types (char, int, float) from within Havok, as we cannot override new and delete for them. See `hkThreadMemory` for more information.

2.1.5.3 Alignment

By default, Havok expects allocated memory to be 16-byte aligned. You will need to take this into account if you override the memory allocation methods.

2.1.5.4 Optimising for space

Often memory managers store "magic" information, such as size, just in front of the allocated memory. On modern machines the overhead for this may be up to 64 bytes because of alignment issues.

The Havok memory manager tries to save this space using two techniques:

- Many simple classes (e.g. `hkAabb`) know their size, and so the memory allocator doesn't need to explicitly store it - the class knows how much space to free when it is deleted.
- For classes derived from hkBaseObject, the allocated size is stored directly in the instance.

Note, however, that some cases cannot be optimised in this way (for instance, C style arrays) and for these the memory manager falls back on storing the magic header information.

2.1.5.5 Overriding class new and delete

Havok uses macros to declare class local memory management.

The `HK_DECLARE_CLASS_ALLOCATOR` macro overrides class new and delete so that classes derived from `hkBaseObject` are automatically handled correctly by the memory system. Note that the memory manager expects to be able to find a `hkBaseObject` at the object address, so when using multiple inheritance the `hkBaseObject` should come first in the inheritance list. The MEMORY_CLASS macro argument is for gathering memory statistics. A derived class may wish to declare its memory as belonging to a different class. See `memory/hkMemoryClasses.h` for a list of possible values.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Unfortunately, non virtual classes cannot inherit from a memory management class, since some compilers do not implement the empty object optimization. For each of these classes we require them to use the HK_DECLARE_NONVIRTUAL_CLASS_ALLOCATOR macro. The memory class parameter is the same as in the virtual case. The name of the class must also be passed to get its size information.

2.1.5.6 Memory implementations

The 'FreeList' memory system

`hkFreeListMemory` is designed to avoid memory fragmentation and is fairly complex, composed of multiple layers. The first layer is the class hkFreeListMemory itself and derives from `hkMemory`. Requests reaching hkFreeListMemory check an internal structure of free lists, and potentially make an allocation from `hkLargeBlockAllocator`, the next layer. Finally, hkLargeBlockAllocator in turn gets it memory from an instance of `hkMemoryBlockServer`. Each of these layers are discussed in detail below.

For small allocations (less than 512 bytes), hkFreeListMemory uses a 'freelist' allocation strategy from which it is named. For each block size, there is a freelist containing blocks "owned" by Havok but available for use. If an allocation is requested, and a suitable free element is available on the appropriate freelist, the free block is returned to the caller. If there isn't an available block, a new allocation is made to the underlying memory system for a large block of memory which can hold many smaller blocks; these blocks are then added to the freelist and available for use. When an allocation is freed, the block is added back to the freelist, allowing again for a quick allocation (of similar size) at a later time.

For large allocations (equal to or greater than 512 bytes) the hkFreeListMemory uses a contained hkLargeBlockAllocator instance. This provides facilities for fast (although not as fast as a freelist allocation) allocation of large blocks. The overhead of an allocation from the large block allocator is 16 bytes per allocation; however, with this informational overhead the large block allocator can fit allocations tightly together and fuse neighboring free blocks, limiting wasted space otherwise caused by fragmentation. On a side note, the free lists discussed above actually operate on blocks of memory allocated from the hkLargeBlockAllocator.

In order to make the memory system flexible, the hkLargeBlockAllocator receives its memory supply from an instance of `hkMemoryBlockServer`. Havok provides you with two implementations of hkMemoryBlockServer which you can use during the construction of your hkLargeBlockAllocator. These are `hkFixedMemoryBlockServer` and `hkSystemMemoryBlockServer`.

The `hkFixedMemoryBlockServer` provides memory in a single large continuous block. By specifying a block of memory for hkFixedMemoryBlockServer, all allocations made through hkFreeListMemory will be restricted to that block. Note that a result of this approach is that calls to hkSystemAlloc/hkSystemFree will no longer occur (aside from the allocation of the hkFreeListMemory instance, etc).

The second implementation of hkMemoryBlockServer is `hkSystemMemoryBlockServer`. This implementation allocates memory using hkSystemAlloc, always allocates memory of the same size, and is ideally configured to perform fairly large allocations. To configure the size of the allocations it makes simply pass a size (in bytes) in the constructor.

The following two examples show how to create an hkFreeListMemory using hkFixedMemoryBlockServer and hkSystemMemoryBlockServer respectively.

Using hkFreeListMemory with a pre-allocated block of memory:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
// Our pre-allocated block of memory.
static char memoryBlock[1024*1024];

// Our interface to this block of memory.
hkFixedMemoryBlockServer* blockserver = new hkFixedMemoryBlockServer(memoryBlock, sizeof(memoryBlock));

// Our instance of hkMemory to be used by Havok.
hkFreeListMemory* memory = new hkFreeListMemory(&server);
```

Using hkFreeListMemory with hkSystemMalloc/hkSystemFree:

```
// Our block server that uses hkSystemMalloc with 1Mb allocations at a time.
hkSystemMemoryBlockServer* server = new hkSystemMemoryBlockServer(1*1024*1024);

// Our instance of hkMemory to be used by Havok.
hkFreeListMemory* memory = new hkFreeListMemory(&server);
```

The 'Pool' memory system

`hkPoolMemory` uses the system allocator to get 8k 'pages' and then chops them up into different sized blocks depending on the size of allocation. For sizes up to 512 Havok will allocate and use an 8K page. For sizes greater than 512 (1K, 2K, 4K and 8K) Havok will allocate a dedicated page for allocations of that size. In both cases these blocks are then kept on free lists for each size. Large blocks (>8192 bytes) are handed directly to the system allocator.

As mentioned before, you can change where the pool gets pages from by setting the `hkSystemMalloc()` and `hkSystemFree()` function pointers.

With the pool memory system, there is always an inherent 'wasted space' as it is unlikely your memory allocation will exactly fill all of the allocated pages. It is quite common that you will want to tailor the memory allocation strategy to your specific needs. You can find out how to do this in the section Replacing the memory manager

The 'Debug' memory system

There is a special implementation of the memory manager `hkDebugMemory` that provides much more strict error checking than the default `hkPoolMemory`. In addition, on `quit()` it prints the context of each memory leak. Memory leak tracking is dependent on stack tracing facilities being available. See the stack tracing section. To use this:

- Ensure you are compiling in *full debug* configuration. This corresponds to the *full debug* configuration in Havok project files. Please note that *debug* configuration is an optimized build with line numbers enabled. hkDebugMemory will not work in *debug* configuration as defined by Havok.
- Pass `an instance of hkDebugMemory` into `hkBaseSyste::init(...)`.
- Ensure that frame pointer optimisations are turned off (they can mangle the call stack)

If there are NO memory leaks, you should see:

```
************************************************************
-------------- NO HAVOK MEMORY LEAKS FOUND -----------------
************************************************************
```

In the case of memory leaks the report will look like this:

```
************************************************************
                BEGIN MEMORY LEAK REPORT
************************************************************
80 bytes leaked. (chunk==1)
../source2\hkcollide/shape/hkpShape.h:24:hkpShape::operator new
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi?.cpp:54:TriangleShapeApi::TriangleShapeApi
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi?.cpp:109:TriangleShapeApi::TriangleShapeApi
G:\demos\common\menu\MenuGame?.cpp:438:MenuGame::startCurrentGame
G:\demos\common\menu\MenuGame?.cpp:599:MenuGame::stepMenuGame
G:\demos\common\menu\MenuGame?.cpp:142:MenuGame::stepGame
G:\framework\hkdemoframework\hkDemoFramework.cpp:475:hkFrameworkMain
G:\demos\main.cpp:46:main
crt0.c:206:mainCRTStartup
(null):0:ProcessIdToSessionId
------------------------------------------------------------------
132 bytes leaked. (chunk==1)
../source2\hkdynamics/entity/hkpEntity.h:43:hkpEntity::operator new
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi?.cpp:93:TriangleShapeApi::TriangleShapeApi
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi?.cpp:109:TriangleShapeApi::TriangleShapeApi
G:\demos\common\menu\MenuGame?.cpp:438:MenuGame::startCurrentGame
G:\demos\common\menu\MenuGame?.cpp:599:MenuGame::stepMenuGame
G:\demos\common\menu\MenuGame?.cpp:142:MenuGame::stepGame
G:\framework\hkdemoframework\hkDemoFramework.cpp:475:hkFrameworkMain
G:\demos\main.cpp:46:main
crt0.c:206:mainCRTStartup
(null):0:ProcessIdToSessionId
------------------------------------------------------------------
240 bytes leaked. (chunk==1)
../source\hkbase/baseobject/hkBaseObject.h:17:hkBaseObject::operator new
G:\source2\hkdynamics\entity\hkpRigidBody.cpp:48:hkpRigidBody::hkpRigidBody
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi?.cpp:93:TriangleShapeApi::TriangleShapeApi
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi?.cpp:109:TriangleShapeApi::TriangleShapeApi
G:\demos\common\menu\MenuGame?.cpp:438:MenuGame::startCurrentGame
G:\demos\common\menu\MenuGame?.cpp:599:MenuGame::stepMenuGame
G:\demos\common\menu\MenuGame?.cpp:142:MenuGame::stepGame
G:\framework\hkdemoframework\hkDemoFramework.cpp:475:hkFrameworkMain
G:\demos\main.cpp:46:main
crt0.c:206:mainCRTStartup
(null):0:ProcessIdToSessionId
------------------------------------------------------------------
96 bytes leaked. (chunk==1)
..\hkdynamics/entity/hkpEntityDeactivator.h:22:hkpEntityDeactivator::operator new
G:\source2\hkdynamics\entity\hkpRigidBody.cpp:101:hkpRigidBody::hkpRigidBody
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi?.cpp:93:TriangleShapeApi::TriangleShapeApi
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi?.cpp:109:TriangleShapeApi::TriangleShapeApi
G:\demos\common\menu\MenuGame?.cpp:438:MenuGame::startCurrentGame
G:\demos\common\menu\MenuGame?.cpp:599:MenuGame::stepMenuGame
G:\demos\common\menu\MenuGame?.cpp:142:MenuGame::stepGame
G:\framework\hkdemoframework\hkDemoFramework.cpp:475:hkFrameworkMain
G:\demos\main.cpp:46:main
crt0.c:206:mainCRTStartup
(null):0:ProcessIdToSessionId
------------------------------------------------------------------
************************************************************
                  END MEMORY LEAK REPORT
************************************************************
```

The report here contains 4 leaks, with a call stack of each, detailing all calls resulting in the allocation causing the leak. In this demo, a reference was not removed from a rigid body, so it was never deleted. In addition, the shape and entity deactivator owned by the rigid body were also not deleted.

You can see the type of object which leaked by looking at the first line of each leak block. The interesting bit is the following line which gives the file and line number of the call to new the object, which should give you an indication of what class was supposed to "own" the object, and hence forgot to either delete it later, or remove a reference to it.

Note that some leaks obscure others, or are really caused by the same leak. It's up to you to find out the real cause, which is generally the "largest" object not deleted. In this case, one line (rb->removeReference()) fixes 5 leaks, but this may not be obvious from the debug output. You'll need to fix, re-run, fix, re-run etc. until you get the "NO HAVOK MEMORY LEAKS FOUND" all-clear.

You can see an example of the debug memory manager usage running demos with command line option '-c'. Please see quickguide for details.

The Debug Memory Manager also allows you to get information about its current allocations. See the section on Memory Snapshots for more information.

Note: Memset free memory. Marking unused memory.

Some memory implementations overwrite freed/uninitialized memory with known values to aid in debugging. `hkPoolMemory` overwrites freed and newly allocated memory when `HK_DEBUG` is defined. `hkDebugMemory` always overwrites its memory. See `hkMemory::MemoryFill` for the values used.

2.1.5.7 Memory Snapshots

The Debug Memory Manager can be used to gather data about the memory that is currently allocated by the Debug Memory Manager via the `hkDebugMemory::getSnapshot()` method. The snapshots generated by this method are different than those of the `hkpHavokSnapshot` class. These snapshots contains the following information about each allocation:

- A pointer to the allocation.
- The size (in bytes) of the allocation.
- Allocation flags, which indicate how the allocation was made (i.e., as a chunk allocation and/or a byte-alligned allocation).
- The number of locks on that object, as made by the `hkMemory::lockBlock()` method.
- A trace of the call stack when the allocation was made.
- A user-specified mark which can be placed on the allocation, as set by `hkDebugMemory::setAllocationMark()` .

Of particular interest is the user-specified mark, which you can use to tag certain allocations. This mark can be used in the `hkDebugMemory::getSnapshot()` method to filter out certain allocations. For instance, you could give a certain mark to all allocations made by one part of the code, and then get a snapshot of just those allocations made by that particular code. The `getSnapshot()` method also allows for two different modes of allocation filtering, which are governed by a parameter that determines if the

mark argument is to be a treated as a bit mask, or an absolute value. If the mask argument is treated as an absolute value, the method will gather only those allocations with that given value. However, if the mark is to be treated as a bit mask, then an allocation will be added to the snapshot only if its mark has any bits in common with the given mark.

2.1.5.8 Optimizing memory usage

Getting maximum performance out of a memory system can be a difficult as at different sections of your program your memory needs can change dramatically. The havok memory system provides two basic operations which can be used to improve memory performance - one for time and the other for space.

Most fast memory managers use 'pools' in order to manage small memory allocations quickly. The memory system does not need to go to the effort of doing a more complicated memory allocation process if it has a free block of memory available in the pool. Both the freelist memory and the pool memory use similar mechanisms in this respect. Its a form of caching mechanism, if a block of such a size has been allocated before its probably going to happen again in the future. The downside of this is that once a memory block has been added to a pool it does not become available for another size of allocation. So if you allocated all of your memory in 128 byte chunks, and then freed them all, and then tried to allocate a 256 byte chunk the allocation would fail - as all of the memory has been chunked up into 128 byte chunks.

It is possible to write an allocator to check during each free if the pool that contained it is now free, but such mechanisms, generally have an overhead in time and space which is undesirable. To solve this the havok memory interface provides methods where the application can tell the memory allocator that it should optimize memory, for time or space. The most straight forward method is 'garbageCollect'. This method will cause the allocator to optimize memory in space - potentially making more, and larger chunks of memory available. Doing a garbage collection takes some work, so it is not recommended that it is called every frame. Better would be to call it in between levels, or at a point where you know your memory usage is going to change significantly.

Note that doing a garbage collection cannot make available all 'free' memory available - it only makes a pools block of memory free if all of the contained blocks are free. Therefore if you allocated all of the memory with 128 byte chunks and then freed every other chunk - performing a 256 byte allocation will still fail, as no pool will have been freed, and moreover memory is now fragmented such that there is no contiguous block of 256 bytes.

Doing a garbage collection will cause the memory allocator to free up memory currently free in pools. This may slow down subsequent allocations initially as memory will have to be reallocated from the underlying memory system. If you know your memory pattern of usage will be similar but want to improve memory allocation performance you can use the method 'optimize'. This may free up some of your garbage, but mainly it will reorganise subseqent memory allocations so they can be performed more quickly. Also doing an optimize or a garbage collect may significantly improve the performance of the actual use of the memory returned. Implementations can reorder how memory will be returned from allocations to make them more optimal. For example the hkFreeListMemory will reorder memory such that allocations of the same size will tend to be returned contiguously, improving cache usage.

Note that the problem of running out of memory due a pool containing all free memory is not an issue with the hkFreeListMemory. When this allocator cannot allocate it will automatically cause a garbage collection. It is still recommended that you perform a garbage collect independently of this mechanism, as the garbage collection could happen at a time critical section of your application.

On modern computer systems memory can be key in achieving good performance for your application.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

In particular the speed of memory can hugely effect performance, easily overtaking the amount of cycles an algoritm might consume executing instructions. Both the implementation of hkPoolMemory and hkpFreeList memory are designed to be very efficient. Keep in mind that they will return chunks of memory in a completely different pattern for the same allocations. Whether one pattern is better or worse for your application - is tightly coupled to your application. Therefore if you are trying to optimize memory performance we would recommend at least testing both allocators with your application to see which provides the best overall results.

As a rough explanation of the allocation differences, with pool memory for allocations less than 8k, allocations will be generally contiguous even if the allocations are of different sizes. With the freelist allocator, allocations less than or equal to 512 bytes will be generally contiguous by size, not by the order they are allocated.

2.1.5.9 Low memory environments

On many platforms and applications memory is at a premium. Havok provides a selection of facilities which aim to help an application use memory efficiently.

Detecting low memory situations

There are many ways that you could organise a system to expose the current memory situation. For example you could have a method that returns how much memory has been used. Then in the application you could check to see if it had gone above some threshold and if so do something about it.

This mechanism is undesirable though for a variety of reasons. For one it means that the memory limit has to be distributed around the application, which could be messy and open to inconsistencies, and secondly working out how much memory has been used may be relatively costly. Having to work out how much memory is used exactly every allocation can slow and and significantly complicate an allocator - when you only rarely want to test such a value.

In most applications you are not interested on how much memory is used, what you are interested in is 'is there enough memory available?'. The difference is subtle but important because having a method that doesn't return the amount of memory thats used, but will answer if there is enough available allows an implementation to perform optimizations in order to work out the result - most simply it can lazily evaluate the result. Thus the main mechanism for determining the memory situation is the method 'hasMemoryAvailable' on the hkMemory interface. The method takes a size in bytes and returns true if that amount of memory is available. Note that this does not mean that a subsequent allocation of that size will succeed - because it does not mean that there is a contiguous chunk of that size available, just that if all the free blocks are added up they would be greater than that amount.

Thus instead of writing

```
hk_size_t memoryUsed = ... // somehow get the memory used
hk_size_t memoryNeeded = ... // the memory I need to do the operation
hk_size_t memoryLimit = ... // the maximum amount of memory I want to use
if (memoryUsed + memoryNeeded <= memoryLimit)
{
    /// Do something that presumably needs at most memoryNeeded bytes
}
```

You now write

```
hk_size_t memoryNeeded = ... // the memory I need to do the operation
if (hkMemory::getInstance().hasMemoryAvailable(memoryNeeded))
{
    /// Do something that presumably needs at most memoryNeeded bytes
}
```

This has the nice properties of requiring less code, and not having to distribute 'memoryLimit' around your application. The memory limit here is equivalent to the 'soft' memory limit described in the section on memory limiting.

If you just want to know if memory is constrained you can call hasMemoryAvailable with 0, this will return true if the allocator is not in a serious memory limited situation. The amount of memory which constitutes a serious memory limit is implementation specific, but generally amounts to a <4k.

Memory limiting

There are two mechanisms for handling a limit on memory usage - 'hard' and 'soft' memory limits.

The hard memory limit is hit when memory is exhausted as far as the allocator is concerned. At this point the allocator will call a method 'cannotAllocate' on the hkLimitedMemoryListener if one is set on the hkMemory instance. A call to this method means that the allocator couldn't allocate, and indicates you probably need to release some memory. On returning from the call the memory system will garbage collect - so there is no need for you to garbage collect in this method. If the memory allocator is still unable to service the requested allocation it will call allocationFailure on the hkLimitedMemoryListener. At this point the allocator has given up. Most implementations after returning from hkLimitedMemoryListener will return HK_NULL but it is possible the allocator may just halt (such is the case with hkPoolMemory).

You can set a hard memory limit in a couple of ways - if you are using a hkFreeListMemory, you can set the hard limit on the hkMemoryBlockServer. You can also set the hard limit via the setMemoryHardLimit method on hkMemory. It is possible to have an implementation that does not implement this facility - but all of the havok provided allocators will do. Note that with an hkFreeListMemory any hard memory limit setting will be passed straight through to the hkMemoryBlockServer. If you try and set a limit to an amount less than is already allocated the limit cannot be set. Setting the hard limit to 0 means setting the hard limit to the maximum amount of memory. If virtual memory is providing the underlying memory (say with the hkSystemMemoryBlockServer) this means memory can be allocated until all virtual memory is exhausted.

The soft memory limit is the limit that alters the behaviour of the hasMemoryAvailable method. If the amount of memory used plus the amount that is asked for is greater than the soft memory limit, hasMemoryAvailable will return false. Just as with the hard memory limit you can indicate you want no limit by setting the soft memory limit to 0. It is necessarily the case that the soft memory limit must be smaller than the hard memory limit.

You can think of the difference between the soft and the hard memory limit as being the 'wiggle room' to make things run smoothly. If you just had a hard memory limit - each time the limit is hit a garbage collection is performed, as well as perhaps the application releasing some havok objects. This can lead to hard limits being hit repeatidly as memory is requested, some is freed and so on. Also if the hard memory limit fails the result is the return of HK_NULL which can often lead to a crash, unless specifically guarded against.

Having a soft memory limit enables the amount of memory to be restricted without having to hit the hard limit. So what should you set the hard and the soft memory limits to? The hard memory limit

should be set to the total amount of memory havok is going to use. The soft memory limit should be less than the hard limit, but working out what the value should be can be more difficult. One way of working it out would be to set the soft limit 5% lower than the hard memory limit, and run your application with a breakpoint on the 'cannotAllocate' method of the hkLimitedMemoryListener. You want to set the soft limit such that cannotAllocate is never called during normal operation.

If you have no hard limit, then the soft limit setting is fairly easy - its roughly the maximum amount of memory you want havok to use.

Note its not possible to give very concrete values for soft limits compared to hard limits because if your application asks for a large memory block, even if that memory is available it may not be available in a continuous chunk. So another way of thinking of the difference between the soft and hard limit should be it is larger than the largest contiguous block you want allocated at runtime.

Altering memory usage during application execution

Using the hkFreeListMemory and an implementation of hkMemoryBlockServer it is possible to change your memory usage over the execution of your application.

Most trivially after a section where havok memory is greatly used (say in game as opposed to in the menu system), you can call a garbage collect, and then change your soft and hard limits. Doing a garbage collect will cause memory to be handed back to the hkMemoryBlockServer if it is no longer being used. Before entering the more memory intensive section of code again, it would be wise to do a garbage collect again and set your soft and hard memory limits once again.

There are special caveats to this when using the hkFixedMemoryBlockServer or any 'singleBlockServer' type. When you do a garbage collect the server will resize the allocated block such that it is as large enough to hold all of the allocations. At this point you can set the hard limit to being close to this value if needs be, and the memory above the limit is now available to your application.

2.1.5.10 Profiling the Memory Manager

Let's take a look at exactly how much memory Havok is taking up and which elements are using the most memory. Calling the following method will produce memory profiling info:

```
hkMemory::getInstance().printStatistics( &hkcout );
```

Note that not all memory managers support printStatistics. As it stands the hkFreeListMemory does not support the printing of statistics. The hkPoolMemory and hkDebugMemory do have support and produce output of the style below.

The following output is produced this data was taken from the multi-pendulum example in the Havok demo framework. There are 1000 rigid bodies and 1000 ball and socket constraints in this demo:

```
Memory allocated by the Operating System for the Pool
*****************************************************
page size = 8256
number of pages = 694

Memory allocated by the Operating System outside the Pool
*********************************************************
num allocs, total size, high mark
        23,    3472664,   3669272

Details by type
***************

type                , block bytes,     blocks,     allocs,   OS bytes,       news, high mark (block bytes)
ROOT                ,    3710848,      20684,     132280,    3472664,         23,    5649712
--BASE              ,     182568,       6498,     115361,    3177592,         11,    2083160
----BASE_CLASS      ,         64,          3,          3,          0,          0,         64
----STRING          ,      65048,        931,      19178,          0,          0,      67704
----ARRAY           ,     116632,       5544,      49068,     112568,          8,     275880
----SINGLETON       ,        768,         16,         17,      16416,          1,        816
----STREAM          ,         16,          2,         16,          0,          0,       8288
----MONITOR         ,         40,          2,      47083,    2000016,          1,    1730408
--COLLIDE           ,      28952,         34,       1388,          0,          0,      66584
----AGENT           ,       1096,          4,          4,          0,          0,       1096
----BROAD_PHASE     ,        160,          4,          4,          0,          0,        160
----CONTACT         ,       1008,          9,        186,          0,          0,       9968
----CDINFO          ,      10304,         13,       1190,          0,          0,      38976
--DYNAMICS          ,     867728,       6010,       6186,          0,          0,     868240
----ACTION          ,         96,          1,          1,          0,          0,         96
----DEACTIVATOR     ,     128000,       1000,       1000,          0,          0,     128000
----CONSTRAINT      ,      95904,       1998,       1998,          0,          0,      95904
----ENTITY          ,     257536,       1006,       1006,          0,          0,     257536
----MOTION          ,     257536,       1006,       1006,          0,          0,     257536
----WORLD           ,       1024,          1,          1,          0,          0,       1024
----SIMISLAND       ,     127616,        997,       1173,          0,          0,     128128
--DISPLAY           ,    2627776,       8068,       9271,     295072,         12,    2627904
--UTILITIES         ,        320,          7,          7,          0,          0,        320
----VDB             ,        320,          7,          7,          0,          0,        320
--DEMO_FRAMEWORK    ,         48,          1,          1,          0,          0,         48
--DEMO              ,       3456,         66,         66,          0,          0,       3456

Current Memory details by size
******************************
block size, blocks in use, free blocks, size in use, size not in use
         8,          4570,        7975,       36560,       63800
        16,            23,        4131,         368,       66096
        32,          2924,         279,       93568,        8928
        48,          3076,          15,      147648,         720
        64,          1805,       27041,      115520,     1730624
        96,          2173,           4,      208608,         384
       128,          2033,           3,      260224,         384
       160,            21,          57,        3360,        9120
       192,          1006,           2,      193152,         384
       256,          2016,           2,      516096,         512
       320,             7,           1,        2240,         320
       512,             9,          56,        4608,       28672
      1024,             3,           5,        3072,        5120
      2048,          1006,           2,     2060288,        4096
      4096,             8,           6,       32768,       24576
      8192,             4,           1,       32768,        8192

Pre-allocate source code
```

. . . continued from above:

```
**********************

{
        int sizes[] = { 8, 16, 32, 48, 64, 96, 128, 160, 192, 256, 320, 512, 1024, 2048, 4096, 8192, -1 };
        int blocks[] = { 12545, 4154, 3203, 3091, 28846, 2177, 2036, 78, 1008, 2018, 8, 65, 8, 1008, 14, 5
          , -1 };
        hkArray<void*> ptrs;
        int j = 0;
        while (sizes[j] != -1 )
        {
            int i;
            for (i = 0; i < blocks[j]; ++i )
                ptrs.pushBack( hkAllocateChunk<char>(sizes[j], HK_MEMORY_CLASS_PRE_ALLOCATED ) );
            for (i = 0; i < blocks[j]; ++i )
                hkDeallocateChunk<char>(ptrs[i], sizes[j], HK_MEMORY_CLASS_PRE_ALLOCATED );
            ptrs.clear();
            j++;
        }
    }

Current Memory Usage Summary

***************************
Total memory in pool = 5729664
Unused memory in pool = 1974400 (= 34.728 percent of total)
Memory System Overhead = 44416
Total outside Pool   = 3472664

**********************
Total memory (pool and system) = 9202328
```

First of all, we are currently using the pool allocation system. This has a pool of memory pages, each page can accept many allocation requests without the request resulting in an operating system call. This is efficient as a memory allocation can be costly in terms of CPU cycles. The pool accepts allocations of up to 8192 bytes (8Kb) and passes all bigger requests on to the operating system.

There are four sections in the memory summary:

- Breakdown of in-pool and out-of-pool memory allocs
- Details by classification
- Details by size
- Pre-allocate source code
- Summary

Looking at the summary above, we can see that the total memory used is 9.2MB, of which 3.4MB was allocated in requests over 8192 bytes, and 5729664 bytes was allocated as part of the pool (i.e. in allocations less than 8K bytes).

The breakdown section tells us how many system allocations were made, how many pool pages were required and what the overhead per page was. Also we can see the size of a single page:

```
Memory allocated by the Operating System for the Pool
*****************************************************
page size = 8256
number of pages = 694

Memory allocated by the Operating System outside the Pool
*********************************************************
num allocs, total size, high mark
        23,    3472664,   3669272
```

Looking at the details by type, there are considerable allocations in three areas: display, dynamics, base and outside of the pool. Firstly the out of pool allocations are primarily the result of a 2MB allocation for the Monitor Stream in hkBase plus a number of allocation in display. Both these allocations can be ignored when analysing the physics allocations. The large number of rigid bodies and constraints constitute the dynamics allocations. We have only one type of object and it's very simple (box) so there is very little overhead for agents, geometry or bounding volume info. The display section is mainly display bodies being used for rendering and can usually be ignored when analysing the physics memory usage.

Finally, the base category on the whole contains allocations for base classes and arrays. The array section usually means internal lists and can become large in cases of extreme activity. In this situation, there can be a large number of lists of listeners, agents, contact points, islands, etc. The base_class section is a catch-all type for objects which inherit from hkBaseObject but do not have their own memory category.

The pre-allocate source code section provides sample code, which when run prior to making any in-game Havok allocations, should remove all heap allocations. I.e. the preallocation sets up all the pages required by the pool memory system for the game in advance.

2.1.5.11 Memory Statistics

In order to find out how memory is being used by and object you can use the 'calcStatistics' method that can be found on hkReferencedObject derived classes. The calcStatistics method will tell a hkReportStasticsCollector interface how much memory is used by the object it is called on, it will then go on to inform the collector about the memory that is used by contained objects.

The classes that are used with the calcStatistics method all derive from the hkReportStatisticsCollector interface. The most simple class is hkReportStatisticsCollector which will dump out to a hkOstream memory that is reached.

```
hkRigidBody* bd = ...;

// Set up stream to write the results to
hkOfstream statOutFile("stat.txt");
// Set up the collector
hkReportStatisticsCollector collector(statOutFile);
// We need to call start before we do the calcStatistics call
collector.start();
// The collector will be informed of memory reached from db, and write to the stream
db->calcStatistics( &collector);
// Must be called when all the stastics you want to collet is complete
collector.end();
```

If you want your own classes to make use of the calcStatistics system, then it needs to either derive from

hkReferencedObject and implement calcStatistics and/or calcContentStatistics, or the class is reflected. When a class is reflected an automatic implementation of calcStatics will be used that will traverse the memory of the contained members. NOTE that if the base class implements calcStatistics, you will need to implement it in in derived classes. There is a toolkit which has the functionality for the automatic traversal based on reflection - its called hkStatisticsCollector util.

The reason you will get a default implementation of calcStatistics/calcContentStatistics is that hkReferencedObject implements the methods. This looks like

```
void hkReferencedObject::calcStatistics( hkStatisticsCollector* collector ) const
{
    const hkClass* cls = hkStatisticsCollectorUtil::defaultBeginObject((void*)this,collector);
    if (cls)
    {
        calcContentStatistics(collector,cls);
        collector->endObject();
    }
}
void
hkReferencedObject::calcContentStatistics( hkStatisticsCollector* collector,const hkClass* cls ) const
{
    HK_ASSERT(0x423423,cls != HK_NULL&&"For the default implementation to work the class must be passed in
      ");
    hkStatisticsCollectorUtil::addClassContentsAll((void*)this, *cls,collector);
}
```

Read the documentation on hkStatisticsCollectorUtil for more information on the methods. There are two methods on hkReferencedObject to support derivation, correctly and being able to chain to parent classes implementations. The calcContentStatistics method can always be chained by calling the parent. The calcStatistics method should not call its parent, as doing so will the derived class will called 'start/endObject' eroneously on the collector.

If you have a hkReferencedObject then you can override these methods and provide an implementation of how memory is used by your class. If the class isn't derived from hkReferencedObject, it can still be traversed by hkStaticsCollectorUtil if the class is reflected. An example of writing the methods is given below. Note that when calling calcContentStatistics you can pass cls as HK_NULL if you are not internally using the hkStatisticsCollectorUtil.

```
void hkSomething::calcStatistics( hkStatisticsCollector* collector ) const
{
    collector->beginObject("hkSomething", collector->MEMORY_INSTANCE, this);
    calcContentStatistics(collector,&hkSomethingClass);
    collector->endObject();
}

void hkSomething::calcContentStatistics( hkStatisticsCollector* collector,const hkClass* cls ) const
{
    // Call the parents calcContentStatistics
    hkSomethingParent::calcContentStatistics(collector,&hkSomethingParentClass);

    // Add contained memory
    collector->addArray("AgentPtr", collector->MEMORY_RUNTIME, m_collisionDetails);
    for ( int i = 0; i < m_collisionDetails.getSize(); i++ )
    {
        collector->addChildObject( "Agent", collector->MEMORY_RUNTIME, m_collisionDetails[i].m_agent );
    }
}
```

Finally one of the more powerful abilities is to match up memory that was reached via doing a calcStatistics traversal and what memory is allocated. This can be achieved using hkMatchSnapshotStatisticsCollector. Code using the matcher is already in the the demo framework. The matcher requires hkDebugMemory being used as it uses the hkMemorySnapshot functionality to work out what memory was allocated before and after the demo. To enable the matching in the demos, just use the -c command line switch, which will enable the use of hkDebugMemory and the matching. The contents of the matching is written out to a file in the demo directory called 'captureMemoryDump.txt'.

2.1.5.12 Replacing the memory manager

The memory manager class used by Havok has to be specified when you initialize hkBaseSystem. See the Base System section for details on how to do this. When getting started with Havok it is common to use one of Havok's own memory managers such as `hkPoolMemory`. `hkPoolMemory` is a subclass of hkMemory and implements its virtual methods. When integrating Havok fully into your game engine it is likely you will want to create your own memory manager by creating your own hkMemory subclass. This new subclassed memory manager can now be passed to `hkBaseSystem::init(...)`, after which Havok will now direct all its allocations through your memory manager.

When writing your own memory manager it is worth bearing in mind the following:

- By default, Havok expects all allocated memory to be 16-byte aligned. Make sure that you have considered this in your custom allocation implementation.

2.1.5.13 Example: Custom Memory Manager

We have implemented the following memory manager, which aligns allocations to 16 bytes by default, and ignores the diagnostic methods. These are the pure virtual methods in the hkMemory class which must be implemented in your own memory manager. Note we will define the two utility functions from hkMemory.

```
// given an unaligned pointer, round it up to align and
// store the offset just before the returned pointer.
static inline void* hkMemoryRoundUp(void* pvoid, int align=16)
{
  char* p = reinterpret_cast<char*>(pvoid);
  char* aligned = reinterpret_cast<char*>(
    HK_NEXT_MULTIPLE_OF( align, reinterpret_cast<int>(p+1)) );
  reinterpret_cast<int*>(aligned)[-1] = (int)(aligned - p);
  return aligned;
}

// given a pointer from hkMemoryRoundUp, recover the original pointer.
static inline void* hkMemoryRoundDown(void* p)
{
  int offset = reinterpret_cast<int*>(p)[-1];
  return static_cast<char*>(p) - offset;
}

class CustomMemory : public hkMemory
{
  public:
    CustomMemory() { }

    void lock(); // platform specific thread lock
    void unlock(); // platform specific thread unlock

    inline void* _allocateChunk(int blockSize)
    {
      return hkMemoryRoundUp(::new char[blockSize+16]);
    }

    inline void* _deallocateChunk(void* p, int blockSize)
    {
      ::delete [] static_cast<char*>( hkMemoryRoundDown(p) );
    }

    virtual void* allocateChunkBatch(void** ptrs, int numBlocks, int blockSize)
    {
      lock();
      for( int i = 0; i < numBlocks; ++i )
      {
        ptrs[i] = _allocateChunk(blockSize);
      }
      unlock();
    }

    virtual void deallocateChunkBatch(void** ptrs, int numBlocks, int blockSize)
    {
      lock();
      for( int i = 0; i < numBlocks; ++i )
      {
        ptrs[i] = _deallocateChunk(p,blockSize);
      }
      unlock();
    }

    virtual void* allocateChunk(int nbytes, HK_MEMORY_CLASS cl)
    {
      lock();
      void* p = _allocateChunk(nbytes);
      unlock();
      return p;
```

...continued from above:

```
    }

    virtual void deallocateChunk(void* p, int nbytes, HK_MEMORY_CLASS )
    {
      if(p)
      {
        lock();
        _deallocateChunk(p, nbytes);
        unlock();
      }
    }

    virtual void printStatistics(hkOstream* c) { }
};
```

To make this the memory manager used by Havok simply pass it to hkBaseSystem::init(...) ensuring that hkBaseSystem::quit was called previously. No recompile of hkBase is needed.

Now we have created our custom memory manager, used it to create the other Havok singletons and ensured that it is going to be used for the instantiation of all Havok objects.

2.1.5.14 Multithreaded Memory Management

Havok's memory managers are thread-safe. For this reason they might pose a serious performance bottleneck when used intensively by multiple threads. To overcome this, each Havok thread uses its own instance of `hkThreadMemory`.

`hkThreadMemory` has the ability to cache memory blocks freed by a thread and use them for subsequent memory allocations without the need to reference the global `hkMemory` manager.

Note that:

- All memory allocations per thread are handled by the `hkThreadMemory` class.
- All memory allocations are shared between the threads. This means a block which is allocated by one `hkThreadMemory` can be freed by another.
- Using `hkThreadMemory` is optional. You can also use `hkMemory::getInstance()` directly, though it might be slower.

`hkThreadMemory` works similarly to `hkPoolMemory`. In addition:

- It is highly optimized to handle memory for a single thread. It does so by caching a limited number of blocks in a thread-local free list. There are free lists for each different size of block.
- Each free list (for each size) can hold only a maximum number of items (`m_maxNumElemsOnFreeList`)
- If `maxNumElemsOnFreeList == 0` then the free list is not checked at all, and the `hkThreadMemory` calls the appropiate `hkMemory::getInstance()` functions.

- If, for a given size, there is no free block on the free list, a block is fetched by calling `hkMemory::getInstance().allocateChunkBatch()`.
- If a block is freed, it is assumed that this block was allocated by this memory manager. Its corresponding free list is found (by rounding up the size of the block to the next suitable block size) and the block added to the free list.

  If the number of elements on a particular free list exceeds `m_maxNumElemsOnFreeList`, some blocks are forwarded to `hkMemory::getInstance().deallocateChunkBatch()`.

Notes:

- All allocations greater than 8 bytes should be 16 byte aligned.
- Allocations of 8 bytes or less should be aligned to 8 bytes.
- You are only allowed to set `maxNumElemsOnFreeList != 0` if you use this class with the pool memory.
- The `hkThreadMemory` class cannot be replaced like `hkMemory`, but it can be disabled.

2.1.5.15 Fast Runtime Memory Management for Multithreaded Environments

To further improve the performance of memory allocation and deallocation Havok's memory managers provide an interface to so-called fast runtime memory blocks. These blocks can be preallocated once by the user before simulation is started and are then cached. Every allocation on these blocks will return the smallest available memory block or allocate a matching new one through the regular `hkThreadMemory` class functions (for smaller blocks) and `hkMemory` class functions (for larger blocks). Every deallocation simply flags that block of memory as 'available' and returns it to the fast runtime's internal pool of memory blocks for future re-use. This implementation is thread-safe and can be seen as some kind of replacement for a regular singlethreaded stack-based memory system which wouldn't work properly with multiple threads.

Fast runtime blocks are currently only used inside Havok's actual simulation calls (i.e. `stepDeltaTime()` or `stepProcessMt()`) and can freely be used on the outside by the user.

Note that:

- All memory allocations through the `allocateRuntimeBlock()` function must be matched by a deallocation through `deallocateRuntimeBlock()`.
- All user-provided memory through provideRuntimeBlock() must be freed by the user himself.

2.1.5.16 Stack based allocation - Reducing heap allocations

Most of Havok's allocations are less than 512 bytes and so fit into the pool in `hkPoolMemory`. Many of the heap allocations outside of the pool are for purely temporary arrays and we have provided hooks to make these as efficient as possible both in terms of speed and not fragmenting the heap. The methods are `void hkThreadMemory::allocateStack(int nbytes)` and `void hkThreadMemory::deallocateStack(void* ptr, int nbytes)`.

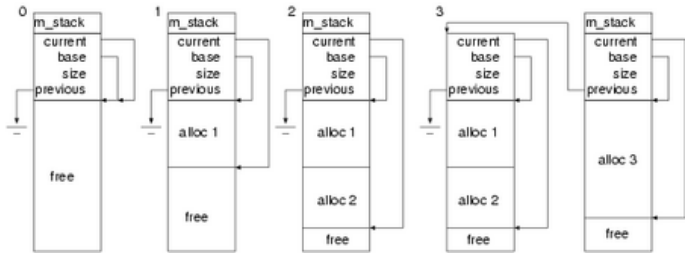
These temporary allocations follow a *stackframe-based* pattern. That is, deallocations always occur in reverse order to allocations. Furthermore the allocations are scoped like local variables. For example

```
void f()
{
  void* mem0 = hkThreadMemory::allocateStack(1000);
  //...
  {
    void* mem1 = hkThreadMemory::allocateStack(2000);
    g(mem0, 1000, mem1, 2000);
    // mem1 must be freed before the end of this block
    hkThreadMemory::deallocateStack(mem1, 2000);
  }
  //...
  // mem0 must be freed before f() returns.
  hkThreadMemory::deallocateStack(mem0, 1000);
}
```

Usually, these functions will be called from a `hkLocalArray<>` which will take care of these requirements. See the API reference for details on `hkLocalArray<>`.

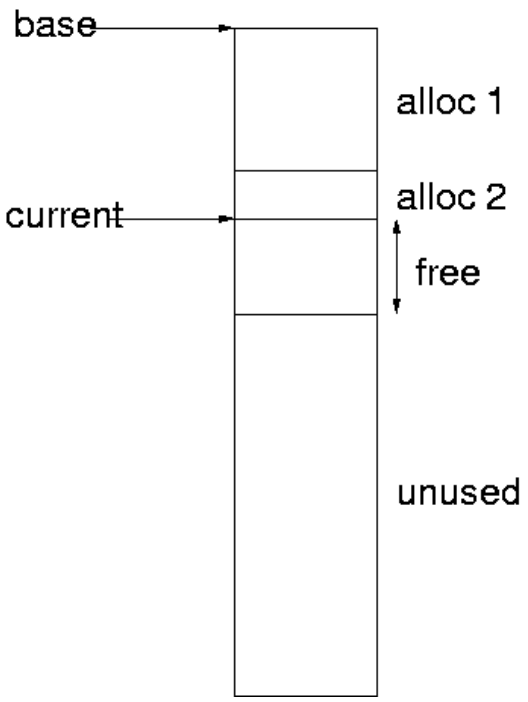
You will notice that the `allocateStack` function is inline. The user replaceable part is `onStackOverflow` which is called from the inline function when we there is not enough memory in the current allocation stack. Similarly onStackUnderflow is called when the current allocation stack is empty. See `hkbase/memory/hkThreadMemory.cpp` for the default implementations of these functions.

The following diagram shows three allocations. The first two fit into the current stack. The third allocation calls onStackOverflow to create a new block which it allocates from. When allocation 3 is released, onStackUnderflow is called which will free the newest block and restore the system to state 2.



Note that the blocks allocated by `onStackOverflow()` need not be of identical size. Also, if `onStackOverflow()` is allocating from a physically contiguous pool of memory, then an implementation may want to remove dead space at the end of the current block before allocating the new block.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



You can disable the frame based allocation if needed. You can disable it by not updating the `m_stack` variable in the virtual function `onStackOverflow`. This means that `allocateStack` will always forward to `onStackOverflow`. Similarly for `deallocateStack` and `onStackUnderflow`. `hkThreadMemory::Stack` is not thread-safe and for that reason its a part of thread-local `hkThreadMemory` instance.

setStackArea()

You can set up a small initial area for the stack allocator to use using `setStackArea()`, to prevent lots of calls to `onStackOverflow` and `onStackUnderflow`. You now must explicitly set this memory with a block yourself, after you initialize `hkBaseSystem`. For example, to allocate 256k using the Havok memory manager, you would add the following code:

```
int stackSize = 0x40000; // enusre 16 byte aligned
char* stackBuffer = hkAllocate<char>( stackSize, HK_MEMORY_CLASS_BASE);
hkThreadMemory::getInstance().setStackArea( stackBuffer, stackSize);
```

You can call `setStackArea()` any time the stack is empty. The stack will be empty outside of `hkpWorld::stepDeltaTime` and all callbacks.

Asserts and warnings have been added which will fire if you do not set the stack area, or if you set it too small a value. It is safe to disable these, however you may experience a large performance hit if you do not set the stack area. As a fallback solution for situations where the stack actually does overflow, a heap allocation is made, using `hkAllocate()`.

Choosing an appropriate stack size

The optimum stack size can vary quite a bit between games, depending on the quantity of physics objects present in a level and the manner of their interactions. Since `onStackOverflow()` will assert when memory runs out, a good way to choose a suitable stack size is to start with a higher value than

necessary and tweak it until it's down to an appropriate level.

Scratch Space

The stack is intended to be used with a block of "scratch space" memory, which many games already have. Since you can set the amount of scratch space allocated to the stack whenever it is empty (outside of calls to Havok and your own usage of hkLocalArray, hkLocalBuffer etc.), you may wish to temporarily switch to a larger stack for larger operations, such as a world step or perhaps large broadphase computations. This kind of scratch space usage is shown here:

```
// The game arranges that nothing else is using
// bigScratchMem at this point.
setStackArea( bigScratchMem, bigScratchMemSize );
world->step();
...
// At this point some other subsystem may be using bigScratchMem
// so we use a smaller private stack so that stack allocations
// (e.g. from broadphase queries) do not trash bigScratchMem.
extern char havokStack[16000];
setStackArea( havokStack, sizeof(havokStack) );
```

2.1.6 Error Reporting

The Havok libraries have embedded error checking which can be used to help debug problems. There are three types of error supported by Havok: warnings, asserts and errors. Macros are provided for convenience to raise these errors. This also means that they can be easily compiled out so that they use no CPU cycles.

Note:
Remember, asserts and warnings are not raised in release builds, so make sure you are linking against the debug versions of the libraries.

2.1.6.1 hkError

The hkError class is an hkSingleton that provides a number of error reporting functions to direct error, warning, and assert information from Havok to an appropriate location.

The default implementation of hkError is hkDefaultError, which prints out a message for each error, warning, or assert to the standard error output, e.g. STDERR. During the initialization of hkBaseSystem a `hkErrorReportFunction` must be specified. Typically this just forwards to `printf` but it is also possible to invoke a method on a custom error handler object.

The four levels of error have the following behaviour:

Error Level	Execution Impact	Disabling	Release / Debug
HK_ERROR	Break Execution	No	Release, Debug
HK_ASSERT	Break Execution	Yes	Debug
HK_WARN	Continue Execution	No	Debug
HK_REPORT	Continue Execution	No	Debug

HK_ERROR	Header files and libraries have different build numbers
HK_ASSERT	Length of ray cast must be non-zero
HK_WARN	You are setting the collision filter after adding entities. Collisions between these entities will not have been filtered correctly.
HK_REPORT	Visual debugger server initialized, my IP address is 10.55.1.10

Table 2.1: Example errors raised from within Havok

The file and line number are output in addition to the type of message (warning, assert, error) and a descriptive string that indicates the reason for the error.

2.1.6.2 Raising an Error

The following are examples of how you can raise an error, assert or warning with or without the convenience macros:

```
#include <Common/Base/System/Error/hkerror.h>

HK_WARN( "You haven't initialised the memory manager!" );
HK_ERROR( "Null object passed as parameter." );

// Asserts evaluate an expression and print out the expression and description
HK_ASSERT2(0x473a8c3a, (pRigidBody != HK_NULL), "Null body passed as parameter" );

// You can use the error handler without using the MACROS
hkError::getInstance().message( hkError::MESSAGE_ERROR, 0x10101010, "Directly accessing the error handler.
  ", __FILE__, __LINE__);
```

2.1.6.3 Creating a Custom Error Handler

The error handling system provides a means to redirect the default error output to fit into your existing framework.

There are two main ways of redirecting error output, the first approach is to use Havok's `hkDefaultError` handler in conjunction with user defined print functions or print objects. The simplest approach is to pass a `hkErrorReportFunction` to `hkBaseSystem::init` when the Havok subsystem's (including `hkDefaultError`) are being constructed. You can do this as follows:

```
#include <Common/Base/System/hkBaseSystem.h> // include for hkBaseSystem
#include <Common/Base/Memory/Memory/Pool/hkPoolMemory.h> // hkPoolMemory
extern "C" int printf(const char* fmt, ...); // For printf, used by the error handler

// Stub function to print any error report functionality to stdout
// std::puts(...) could be used here alternatively
static void errorReportFunction(const char* str, void* errorOutputObject)
{
    printf("%s", str);
}

{
...
    hkBaseSystem::init( new hkPoolMemory(), errorReportFunction );
...
}
```

Note:
`printf` is not useful for error reporting on consoles, since there is no console window. Instead, the error should be sent to some platform-specific or custom debug handler.

It is also possible to invoke a method on a custom error handler object. Use the *errorReportObject* parameter interface to specify the custom error handler object. A custom *errorReportFunction* can be used to invoke the method on the custom error handler object. For example consider the following pseudo code:

```
class MyErrorHandler
{
    public:
        virtual void printError(const char* error);
};

MyErrorHandler g_gameError; // The instance of your error handler

void errorReport(const char* s, void* errorReportObject)
{
    static_cast<MyErrorHandler>(errorReportObject)->printError(s);
}

{
...
    hkBaseSystem::init( new hkPoolMemory(), errorReport, &g_gameError );
...
}
```

The second approach is to create your own error handler. In this case simply create your own `hkError` subclass, implementing the virtual functions. You can then replace the default error handler using the method described in the hkSingletons section. In this example, we want to redirect the output from the standard error channel to a custom dialog box.

Here's the derived custom error class:

```
// Our custom handler needs to show all warnings, asserts and errors using a dialog box.
class hkCustomError: public hkDefaultError
{
  public:
    virtual int message(Message m, int id, const char* description, const char* file, int line)
    {
      if (m == hkError::MESSAGE_ERROR)
      {
          // My own error output mechanism, in this case a custom dialog class.
          gDialog->showError( description );
      }
    }
};
```

It derives from the default error class so that it can redirect errors to the dialog class. We now have an error handler, but it needs to be registered as the handler that Havok will use. When the base system is initialised the default memory object is created. We must replace this with an instance of our own handler using the *hkError::replaceInstance()* method.

```
{
  ...
  // Initialise memory, constructing a default error handler
  hkBaseSystem::init();

  // Replace default error handler
  hkError::replaceInstance( new hkCustomError() );
  ...
}
```

2.1.6.4 Disabling errors, warnings and asserts

Be careful! If Havok raises an assert then it usually means that something is fundamentally wrong. Disabling an assert may well let you continue development but should only be done as a short term measure. You should ensure that you address the underlying cause of the assert.

In Havok all warnings and asserts have unique ids which are persistent across releases. Occasionally you may not want to be notified by Havok with an assert or warning every time a certain event happens. To disable warnings, errors or asserts for a particular line of code, you can use the `setEnabled( int id, hkBool enabled )` method, passing it the appropriate assert ID. You can find the ID by examining the output after a particular assert etc. is triggered.

```
#include <Common/Base/System/Error/hkError.h>

// Disable the assert by its id (a hexadecimal number).
hkError::getInstance().setEnabled( 0x44444444, false );
```

2.1.6.5 Stack Trace

`hkStackTracer` provides stack tracing facilities for several platforms.The win32 implementation can resolve addresses to file:line pairs and function names. For some platforms, external programs may be needed.

`hkStackTracer` is used primarily by the debug memory manager and by the default assert handler.

2.1.7 Monitoring and Timing

In certain situations, you may need to extract and use statistics from Havok. The Havok libs have embedded profiling code that is easy to output and examine. The monitor classes and macros can be found in the monitor directory. All the timer macros are described in hkMonitorStream.h and hkMonitorStream.inl. A utility class is available for manipulating and viewing the monitor data, called hkMonitorStreamAnalyzer.

The Havok monitor macros are very simple and are compiled into the Havok release build. They simply check whether there is any space available in a thread local buffer, which we call a monitor stream, and if there is they write a string pointer and a timer value together with a command identifier into the stream. If the stream of size 0 then the compiled in monitors do not degrade performance. You can use the macros in your own code too, and if you do your own timers and monitors will show up when you output a monitor analysis.

2.1.7.1 Creating a Timer

You can use a the HK_TIMER macros to time sections of code. You create and use a timer as in the following example:

```
HK_TIMER_BEGIN("Sweep", HK_NULL);

//your code to be timed here

HK_TIMER_END();
```

The first parameter is a string identifying the monitor. The second parameter will allow you to specify a particular object to be timed - however, this functionality has not yet been implemented, so for now you should use `HK_NULL`. You finish timing by calling `HK_TIMER_END`.

2.1.7.2 Creating a Monitor

You can also insert the value of any arbitrary variable into the monitor stream using the HK_MONITOR_ADD_VALUE macro. An example of this is shown below:

```
hkArray<Agent> myArray;
//...

HK_MONITOR_ADD_VALUE("num agents", hkReal(myArray.getSize()),
                     hkMonitor::HK_MONITOR_TYPE_INT);
```

As with the HK_TIMER macros the first value is the string identifier. The send value is the variable you wan to capture. The final parameter is currently unused, and should be set to HK_MONITOR_TYPE_INT. Variables captured in this way will show up in the timer dump files.

2.1.7.3 Setting Up and Printing Monitor Results

In order to view and use monitor results, you can print them as they are updated to the screen or a file. To do this you use the hkMonitorStreamAnalyser class. The code below is from the demo CommonApi/DetailedTimers/DetailedTimersDemo, and shows how to use the hkMonitorStream and the hkMonitorStreamAnalyser.

The hkMonitorStream is a singleton that wraps a thread local buffer into which timers are written during simulation. By default this stream is zero size, so no timers will be captured. To enable timers you have to resize this stream. The following code shows how to do this, and should go in your world setup code:

```
/*
 * At startup / construction time e.g. when you create the hkpWorld
 */
{
        // Allocate space for monitor results
        hkMonitorStream::getInstance().resize( 2 * 1024 * 1024  );// 2 meg for timer info per frame
}
```

This prepares a buffer for the profiling results. Here we allocate 2 Mb, which is certainly enough for capturing the timers from one frame. If you do not allocate enough the timers simply stop being written to the stream when it gets full, so you will have a stream with inconsistent timer results (i.e. missing HK_TIMER_END tokens). Next you need to create a hkMonitorStreamAnalyzer class. You use this class to store multiple frames of data, and to print the data out to file when your timing session is finished.

```
// Create the monitor stream analyzer
  hkMonitorStreamAnalyzer analyzer( 10000000 ); // 10 Mb buffer for multiple frames capture
```

This is a very large buffer that will capture a lot of data. If you do not have this space in memory you can create a smaller buffer.

Before each step you reset the hkMonitorStream buffer. After each step you copy the data from the hkMonitorStream into the analyzer buffer. The code for this is:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
/*
 * In your main loop
 */
{
  // Reset monitor stream
  hkMonitorStream::getInstance().reset();




  // Step the world
  m_world->stepDeltaTime( deltaTime );


#ifdef HK_PS2
  // Stop timers. This is necessary as a timer overflow on PlayStation(R)2 causes an exception
  scePcStop() ;
#endif

    hkMonitorStreamFrameInfo frameInfo;


        frameInfo.m_indexOfTimer0 = 0;
        frameInfo.m_indexOfTimer1 = -1;
        frameInfo.m_heading = "usecs";
        frameInfo.m_timerFactor0 = 1e6f / hkReal(hkStopwatch::getTicksPerSecond());

  analyzer.captureFrameDetails(hkMonitorStream::getStart(), hkMonitorStream::getEnd(), frameInfo );
```

When you complete your timer trace, you can output the timers in the hkMonitorAnalyzer to a text file, as folows:

```
// record stats to file
  hkOstream ostr (fileName);
  ostr << TEST_DEMO_NAME "   Timers: \n";
  streamUtility.writeStatistics( ostr );
}
```

If you are running multithreaded, you need to do all this on a per thread basis. Each thread must have its own hkMonitorStream and hkMonitorStreamAnalyzer buffers initialized. Each step each thread must reset the hkMonitorStream, and after each step each thread must call captureFrameDetails to copy the info into its hkMonitorStreamAnalyzer. The theads created by the hkpMultithreadingUtil class in the hctUtilities library create their own buffers. When you wish to output the data to a text file, you need to call hkMonitorStreamAnalyzer::writeMultiThreadedStatistics and pass it the list of streams. See the CommonApi/DetailedTimers/DetailedTimersDemo for the code to do this.

2.1.7.4 Stopwatches

The hkStopwatch class implements a timer that stores time internally as a 64 bit integer.

hkStopwatches are used internally by hkMonitors, and the most straightforward approach to timing sections of your code is to use the hkMonitorBank macros described above. However, you can also use hkStopwatches directly if you like. hkStopwatch has functions for starting, stopping, and resetting a stopwatch, and allows you to get the total running time or the "split" time (the time since the last split

time was requested), in either ticks or seconds.

The following snippet illustrates getting both the split time and the total running time from a hkStopwatch:

```
hkStopwatch sw;
sw.start();
function();
sw.stop();
hkcout << "Iter:" << sw.getSplitTicks() << "\tTotal:" << sw.getElapsedTicks();
```

2.1.7.5 System clock

hkSystemClock is an hkSingleton used by hkStopwatch to get its tick counter and ticks per second values. As it is an hkSingleton, it can be easily replaced by your own clock implementation.

2.1.7.6 Memory Statistics

The main Havok classes, aswell as having internal timers, also provide a way to query how much memory they are using in a more meaningful and per object way than is available from the memory system itself. All hkReferencedObjects (most user level objects in the Havok products) have the following virtual call:

```
virtual void calcStatistics( hkStatisticsCollector* collector ) const ;
```

It is up to the individual Havok object to fill out the 'collector' with the current statistics for the object and all its children (owned objects). Currently the stats collected are all to do with memory usage. The collectors work by using the memory size member (m_memSizeAndFlags) of all hkReferencedObjects so reflect the true mem usage of object itself (less any pointed to objects or storage) and it is up to the object to give all its children (arrays and objects owned) to the collector so that it can recurse. The collectors add both the memory used (eg: hkArray::getSize()) and memory allocated (eg: hkArray::getCapacity()) as the two data values in each Node.

There are two implimentations of hkStatisticsCollector in hkBase/debugutil, and you can of course make your own to collect the stats directly into your own format. The hkSimpleStatisticsCollector creates hkMonitorAnalyzer::Nodes so that the tree structure produced can be inspected by the same code that inspects the monitor timer node trees. Currently the PhysicsApi\Dynamics\MemoryIssues\LimitingMemory and the PhysicsApi\Dynamics\MemoryIssues\SuspendInactiveAgents demos both use the hkSimpleStatisticsCollector to report on the mem usage in the respective demos.

The hkStreamStatisticsCollector does not create Nodes straight away but stores the data in a continous data stream so that it can be copied easily say to another computer to be inspected. To inspect the stream after collection, you simply pass the data block into a hkStreamStatisticsCollector object by using the following two methods in turn:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
void setDataByCopying(const char* buffer, int length, hkBool endianSwap);
void makeTree(hkArray<Node*>& m_nodesOut, hkObjectArray<hkString>* stringTable );
```

You only need to give a string table if you want the Node tree produced to exist after you have deleted the collector that produced it (so the Node name strings will point into the string table rather than the data stream where they came from). The Havok Visual Debugger has a hkpWorldMemoryViewer (hkutilities/visualdebugger/viewer/hkpWorldMemoryViewer.cpp) that takes the current hkpWorld in the Havok Physics simulation and uses the hkStreamStatisticsCollector to create a data block to send across the network for the VDB to interpret. You could use the make a similar simple viewer to stream whatever other memory stats you like.

Currently only the Physics classes have implimentations of calcStatistics() and thus support the statistics collection fully. Both of the collector types support writing the data to a text stream too (writeStatistics(hkOstream& outstream, int reportLevel);)

2.1.8 Working with Streams

2.1.8.1 Low level I/O

The low level input and output of raw bytes is done through the `hkStreamReader` and `hkStreamWriter` interfaces. Havok provides implementations of these classes for

- Memory blocks
- Expandable buffer (hkArray)
- Buffering on top of an unbuffered stream
- Platform specific I/O devices (hard disk, DVD emulator etc)

These low level interfaces are discussed later in this document.

2.1.8.2 hkOstream and hkIstream

The `hkOstream` class is a simple ASCII formatted output stream modelled after `std::ostream`. It contains the usual << operators for basic types, but only the hkflush and hkendl manipulators are supported. A `hkOstream` simply formats its arguments and forwards them to an `hkStreamWriter` object for writing as a stream of characters.

The following snippet shows two examples of using a `hkOstream`.

```
hkcout << "Hello world" << hkendl;
hkcout.printf("Hello world\n"); hkcout.flush(); // equivalent to above.
```

By default, the hkcout `hkOstream` writes to a console window.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Similarly, `hkIstream` is a simple input stream that provides comparable functionality to `std::istream` and requires an `hkStreamReader` to do the actual raw reading of characters.

2.1.8.3 hkOArchive and hkIArchive

Whereas the "Stream" classes operate on text, the "Archive" classes provide I/O for binary data in an endian safe manner. The "Archive" classes perform endian conversion as necessary and again forward to an `hkStreamReader` or an `hkStreamWriter` for the raw byte I/O.

The following snippet shows an example of using a `hkOArchive.`

```
hkOArchive oa("file.dat");
struct {
  int id;
  float scale;
  char data[10];
} foo;
oa.write32( foo.id );
oa.writeFloat32( foo.scale );
oa.writeArray8( foo.data, 10 );
```

2.1.8.4 hkStreamWriter

As you know, Havok streams use an `hkStreamWriter` to write their data.

The `hkStreamWriter` class has two virtual functions that must be implemented. These are:

- `int write(const void* buf, int nbytes) // write the buffer buf`
- `bool isOk() const // has the stream encountered an error?`

For example an implementation which used stdio could be as follows

```
class hkStdioStreamWriter : public hkStreamWriter
{
  public:

    hkStdioStreamWriter(const char* fname)
      :  m_owned(true)
    {
      m_fhandle = fopen(fname, "rb");
    }

    virtual $\sim$hkStdioStreamWriter()
    {
      if(m_fhandle) { fclose(m_fhandle); }
    }

    virtual int write( const void* buf, int nbytes)
    {
      HK_ASSERT(m_fhandle != HK_NULL);
      return fwrite(buf, 1, nbytes, m_fhandle );
    }

    virtual bool isOk() const
    {
      return m_fhandle != HK_NULL && !feof(m_fhandle);
    }

    //
    // This simple example does not implement seek/tell
    //
  protected:

    FILE* m_fhandle;
};
```

Additionally if the stream supports random access, it should override `seekTellSupported()` to return true and must then implement `seek()` and `tell()`.

Prior to version 2.3, buffering was built into every stream interface. From 2.3 onwards buffering is done in a proxy class which can buffer any underlying writer. This class (`hkBufferedStreamWriter`) can handle alignment and block size issues allowing the underlying classes to be considerably simplified.

In the above example, stdio handles buffering. We can easily add buffering if required by wrapping a writer in a `hkBufferedStreamWriter`. For example:

```
hkStreamWriter* openWriter(const char* filename)
{
    hkStreamWriter* custom = new MyCustomWriter(fileName);
    hkStreamWriter* buffered = new hkBufferedStreamWriter(custom);
    custom->removeReference(); // buffered now owns the underlying custom writer
    return buffered;
}
```

2.1.8.5 hkStreamReader

The `hkStreamReader` interface parallels the writer interface with the mandatory methods

- `int read(void* buf, int nbytes) // read into buffer buf`
- `bool isOk() const // has the stream encountered an error?`

As before, random access may be implemented if desired.

The `hkStreamReader` interface has one additional aspect worth mentioning, namely the mark/rewind interface. When parsing, it is common to read a few characters and then put some back. This can be done very efficiently through the mark/rewind interface. Normally users will not need to implement this interface - it is much simpler to wrap the reader in an `hkBufferedStreamReader` instead. This class also handles block alignment and size issues.

For example a minimal reader:

```
class MyCustomReader : public hkStreamReader
{
    public:

        MyCustomReader(const char* fname)
        {
            m_fd = myOpen(fname, MY_RDONLY);
        }

        virtual $\sim$MyCustomReader()
        {
            if (m_fd >= 0) { myClose(m_fd); }
        }

        virtual int read( void* buf, int nbytes )
        {
            if( m_fd != -1 && nbytes > 0 )
            {
                int nread = myRead(m_fd, buf, nbytes);
                if( nread > 0 )
                {
                    return nread;
                }
                myClose(m_fd);
                m_fd = -1;
            }
            return 0;
        }

        virtual hkBool isOk() const
        {
            return m_fd >= 0;
        }

        int m_fd;
};
```

This simple reader does not handle buffering nor the mark/rewind interface which is used by xml parsing for instance. We can support the mark/rewind interface and likely improve read performance by wrapping it in a buffered stream thus:

```
hkStreamReader* openReader(const char* filename)
{
    hkStreamReader* custom = new MyCustomReader(fileName);
    hkStreamReader* buffered = new hkBufferedStreamReader(custom);
    custom->removeReference(); // buffered now owns the underlying custom reader
    return buffered; // buffered now handles all buffering and mark/rewind calls
                     // and calls custom->read as necessary
}
```

2.1.8.6 Stream buffer factories

Havok uses an `hkStreambufFactory` to create `hkStreamReaders` and `hkStreamWriters`. The default `hkStreambufFactory` is `hkDefaultStreambufFactory`, which chooses an appropriate stream buffer type based on your platform.

Usually you will not need to use the factory directly as the stream and archive classes have constructors for most common cases (filename, memory buffer and explicitly created readers/writers). However, some low level interface require raw streams. You can use the high level formatted stream and archive classes as a shortcut for creating the most common types of raw stream readers and writers. For example:

```
hkOstream ostreamToFile("filename.txt");
hkStreamWriter* writer = ostreamToFile.getStreamWriter();
// writer is owned by ostreamToFile
// or we can take ownership by adding a reference.

char myMemBuffer[1000];
hkIstream istreamFromMemory(myMemBuffer, sizeof(myMemBuffer));
hkStreamReader* reader = istreamFromMemory.getStreamReader();
// again reader is owned by the hkIstream and we can
// take ownership by adding a reference.
```

`hkStreambufFactory` is an `hkSingleton`, so you can create your own `hkStreambufFactory` class and replace the default using the method described in the Singletons section. This is useful if you want to specify that your own custom stream buffer should be used instead of the default for your platform. This is described in the following section.

Note:
The default factory wraps a buffer around streams to improve performance and also to ensure that the mark/rewind interface is always available. See the examples in `hkStreamReader` and `hkStreamWriter` sections.

2.1.8.7 Integrating Custom Streams

If you do not want to use one of the provided implementations, you can create your own custom stream readers and/or writers. To ensure that your custom class is instantiated and used by the Havok libraries you must do the following:

- create a factory to instantiate this type of stream objects instead of the default and
- register this factory with the base system, overwriting the default factory

The factory will look like this:

```
class MyCustomStreambufFactory : public hkStreambufFactory
{
  public:

    virtual hkStreamWriter* openWriter(const char* name)
    {
      return new MyCustomStreamWriter(name);
    }

    virtual hkStreamReader* openReader(const char* name)
    {
      return new MyCustomStreamReader(name);
    }

    // openConsole omitted - see next section
};
```

Once we register this factory, all calls to open a named stream will result in a custom stream being used. Here's how we register the factory just after the base system init() call:

```
// create the custom streambuf factory
hkStreambufFactory::getInstance().replaceInstance( new MyCustomStreambufFactory() );
```

Now all of Havok's streaming calls are passed through our custom implementations.

2.2 Havok Math Library

2.2.1 Introduction

The hkMath library provides a platform independent interface to simple linear algebra primitives such as vectors, matrices and quaternions. The interface is designed to be able to take advantage of platform specific optimizations, especially SIMD (vector) instructions, even if compiler support is lacking.

This chapter introduces you to the hkMath classes, highlights some common usage problems and some of the conventions used in hkMath, as well as some optimization tips for working with the library. For more detailed API information for each hkMath class, see the Reference Manual.

2.2.2 Math Types

2.2.2.1 Basic

Depending on your compiler/platform/simd configuration, sev1

- **hkReal**

  The default floating point type, defined as `float`.

- **hkQuadReal**

  A type representing a SIMD register defined as four `hkReals`. This will generally be a typedef to a native type if the compiler has such support. e.g. `vector float` for alitvec platforms.

- **hkSimdReal**

  When SIMD is enabled, a single `hkReal` in a SIMD register, i.e in the `x` component of an `hkQuadReal`. When SIMD is disabled, an `hkSimdReal` is a simple `hkReal`. Using this type instead of `hkReal` can allow the compiler more optimization opportunities.

- **Other hkMath functions**

  Functions such as `hkMath::sqrt()`, `hkMath::sin()`, and so on.

2.2.2.2 Compound Types

This directory contains the main hkMath classes, and a subdirectory containing platform-specific implementations of some of their methods.

- **hkVector4**

  `hkVector4` is Havok's general purpose vector class. Most of the platform specific optimization is done here, and the other math classes are mostly compositions of `hkVector4s`.

  Each vector has four `hkReal` elements, for ease of use with vector processors. However, it can be used to represent either a point or vector (x,y,z) in 3-space, i.e. a 3-vector. The final, or w, value of the `hkVector4` defaults to zero.

- **hkQuaternion**

  Class for quaternions, which you can use to represent rotations. *Note that hkQuaternion transformation methods assume a normalized (unit) quaternion, which is the most common case.*

- **hkMatrix3**

  Class for a 3X3 matrix of `hkReals`.

- **hkRotation**

  Class to store an orthonormal rotation matrix. In Havok there are two ways to store rotations: `hkQuaternion` and `hkRotation`. If you are operating on `hkVector4s` e.g. rotating `hkVector4s` then using `hkRotation` will be faster than an `hkQuaternion`. If you need to store many rotations, `hkQuaternion` is more space efficient. Also, quaternions are faster to renormalize.

- **hkTransform**

  This class represents a rotation and a translation. It is a composition of an `hkRotation` and an `hkVector4`.

- **hkVector4Util**

  Special purpose routines for working with vectors that do not fit into hkVector4, but deserve platform specific optimizations. For example hkVector4Util::rotatePoints(...)

- **hkQsTransform**

  A decomposed (translation vector + quaternion + scale vector) transform. Used by the Havok Animation system; check the Havok Animation chapter in this manual for more details.

2.2.3 Using the Maths Library

This section aims to highlight some of the common issues you may run into when using the hkMath library.

2.2.3.1 Stack Allocations and 16-byte alignment issues

Havok expects math objects memory to be 16-byte aligned. This is because unaligned loads and stores from the SIMD unit are typically very expensive. On platforms which pass SIMD parameters on the stack (e.g. IA32), this means that you cannot reliably pass hkMath objects by value, but should pass them by reference instead. The most common cause of misaligning a `hkVector4` is through passing a stack allocated `hkVector4` by value.

The following examples shows common cases where a `hkVector4` is stack allocated. Note that these cases will not always cause a crash as occasionally they will be correctly aligned. It is this fact that make it difficult to link a runtime crash to an incorrectly allocated `hkVector4`, hence they should be caught as early as possible.

Example 1:

Here a `hkVector4` will be stack allocated before it is passed to the user method `animateToPosition(...)`.

```
// incorrect - compiler bugs mean the temporary may not be correctly aligned
animateToPosition( hkVector4( 10.0f, 0.0f, 0.0f) );

// correct - works on all compilers
hkVector4 pos( 10.0f, 10.0f, 10.0f );
animateToPosition( pos );
```

Example 2:

Here a stack allocated `hkVector4` is returned by value from a conversion method. This is a common case as many users have an in-house Maths library and need to convert `hkVector4s` to and from their own proprietary format.

```
// incorrect: creates and returns a stack allocated hkVector4
hkVector4 convertMyVec2HKVec( MyVec in )
{
    return hkVector4( (float)(in[0]), (float)(in[1]), (float)(in[2]) );
}

or

// incorrect: creates and returns a stack allocated hkVector4
hkVector4 convertMyVec2HKVec( MyVec in )
{
    hkVector4 temp;
    temp.set( (float)(in[0]), (float)(in[1]), (float)(in[2]) );
    return temp;
}

// correct: passes the hkVector4 to be set by reference
void convertMyVec2HKVec( const MyVec& in, hkVector4& out )
{
    out.set( in[0], in[1], in[2] );
}
```

2.2.3.2 Correctly constructing hkMath objects

All hkMath default constructors are lightweight and do not initialize their data members. This means that code such as:

Example 1:

```
hkRotation rotation;
hkQuaternion quat(rotation)
```

Will cause an assert the following assert in the hkQuaternion constructor:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
HK_ASSERT2(isOk(), "hkRotation used for hkQuaternion construction is invalid. hkQuaternion is not
  normalized/invalid!");
```

To prevent this class of assert ensure you construct hkMath objects with valid data.

2.2.3.3 Aliasing of arguments

hkVector4 methods are explicitly safe against argument aliasing. All other math classes are not. For instance:

```
a.setAdd4(b,c) -> a = b+c
a.add4(b) -> a += b
```

This is done in order to avoid creating temporaries for the common case of no aliasing.

2.2.3.4 Viewing asserts in hkMath

The assert mechanism in the Havok library is designed to, amongst other things, catch bad or out-of-range data in the engine. A common place that bad data is trapped is in the hkMath library. It is essential that you are able to view asserts that the hkMath library generates. For details on how to view asserts or override the assert mechanism please consult the Base System Integration guide. Note that asserts will only fire in the Debug Havok libraries, in the release libraries bad data would generally cause a crash or machine exception.

2.2.3.5 Nonstandard operator=

Normally, the member method `operator=` returns a reference to itself so that assignments can be chained.

```
object a,b,c; // fictitious type 'object'
a = b = c;
```

Even when the return value is unused, some compilers are unable to optimise the return value away and generate extra redundant code. Since the Havok math classes are used so pervasively and are so performance critical, objects do not return a reference to themselves.

2.2.3.6 No overloaded operators +,-,*

The definitions of operator such as +, - and * are required to return their result by value. Unfortunately, because of compiler issues, the generated code is often either extremely inefficient or causes a crash as a result of alignment issues. (See the previous section)

2.2.4 Common Conventions

2.2.4.1 Location of methods

Methods are located in the class to which the result is assigned. For example, the operations to transform a `hkVector4` by rotations and transformations are part of `hkVector4` because the result is a `hkVector4`.

2.2.4.2 Naming of methods

Methods that begin with `set`, such as `hkVector4::setAdd4()`, set the value as a function of its input arguments only.

Methods that do not begin with `set` mutate or use the existing value of the object. For instance `hkVector4::add4()` adds to the existing value and `hkVector4::dot3()` uses the existing value.

```
a.setAdd4(b,c) -> a = b+c
a.add4(b) -> a += b
```

2.2.4.3 Method suffixes

Some methods in `hkVector4` have the suffix "3" or "4". As you know, in Havok, all vectors actually store four elements. We use the suffixes to make explicit which elements are being operated on - the first three, or all four. For example, there is both a `normalize3()`and a `normalize4()` method in this class.

However, many `hkVector4` methods have a "4" variant but not a "3". This is because there is no generic hardware support for the "3" version of the operation. These include `add4()` and `setAdd4()`

Other hkMath classes do not have these suffixes because there is no similar ambiguity.

2.2.5 SIMD and Non SIMD support

2.2.5.1 hkSimdReal and hkReal

The Havok math library is carefully constructed so that there is no penalty for platforms that do not have SIMD units. This is done by never passing hkQuadReals by value which results in a register move on SIMD machines but possibly a stack copy on those without.

Havok currently provides SIMD support for version SSE1. For compilers that have SIMD support, we define `hkQuadReal` to be the compiler's native four element SIMD type (for example __m128 or __v4sf). `hkSimdReal` is an `hkQuadReal` for which only the x element contains a valid value. Implicit conversions are provided between `hkSimdReal` and `hkReal` for ease of use.

For compilers that do not have SIMD support, `hkQuadReal` is a structure with four `hkReals` and 16 byte alignment, and `hkSimdReal` is a typedef for `hkReal`.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

2.2.5.2 Ambiguous overloads

Overloaded operators are provided for `hkSimdReal` so that it behaves much like an `hkReal`. Thus:

```
hkSimdReal s0, s1;
hkReal r0, r1;

s0 * s1; // Ok, result is an hkSimdReal.
r0 * r1; // Ok, result is an hkReal.

s0 * r0; // Error ambiguous overload
r0 * s0; // Error ambiguous overload
```

Mixing types is an error because the operation could be performed on either the FPU or SIMD unit. Use an explicit cast in this case.

```
hkSimdReal s0, s1;
hkReal r0, r1;

hkReal(s0) * r0; // Ok, calculation on FPU, result is an hkReal
s0 * hkSimdReal(r0); // Ok, calculation on SIMD unit, result is an hkSimdReal
```

2.2.6 Optimisation

2.2.6.1 FPU / SIMD conversions

Most compilers find it difficult to optimise conversion between SIMD types and FPU types. Where possible use the SIMD types and operators when working with hkMath.

The following examples illustrate this:

```
hkVector4 v0, v1;
hkReal r = v0.dot3(v1);
hkVector4 v2;
v2.addMul4(r, v0);
```

gives the compiler more optimization opportunities if written as

```
hkVector4 v0, v1;
hkSimdReal r = v0.dot3(v1);
hkVector4 v2;
v2.addMul4(r, v0);
```

Similarly

```
// ok
hkVector4 v0;
hkReal r = v0(0)
hkVector4 v2;
v2.addMul4(r, v0);

// better
hkVector4 v0;
hkSimdReal r = v0.getSimdAt(0);
hkVector4 v2;
v2.addMul4(r, v0);
```

2.2.6.2 Load / Store Hazards

Some platforms incur a penalty for overlapping loads and stores of different sizes .

```
hkVector4 v0, v1;
v0(3) = 0;   // store 4 byte float
v1.add4(v0); // read 16 byte simd value
```

Try to avoid mixing element wise write access with whole vector access.

You can safely and quickly get read only access to a vector element using the `getSimdAt()`method of hkVector4. Similarly the `setXYZ0` and `setXYZW` can be used for write access.

2.2.6.3 Multiple dot products with the same vector

Dot products are generally quite slow because of the need to perform a horizontal add an operation not directly supported by most SIMD units. For example, this is the dot product for Pentium III and IV.

```
hkQuadReal x2 = _mm_mul_ps(m_quad,a.m_quad);
hkQuadReal broadcastY = _mm_shuffle_ps(x2,x2,_MM_SHUFFLE(1,1,1,1))
hkQuadReal xySum = _mm_add_ss(broadcastY, x2);
hkQuadReal broadcastZ = _mm_shuffle_ps(x2,x2,_MM_SHUFFLE(2,2,2,2))
hkQuadReal xyzSum = _mm_add_ss(broadcastZ, xySum);
```

If you have (for instance) three dot products with the same vector, you can put the three vectors into the rows of a matrix3 and get three results for only one horizontal add.

2.3 Serialization

2.3.1 Introduction to Serialization

Serialization is the process of converting data to a writable format. Once serialized, the original data can be accurately reconstructed later on. This means that serialized data can safely be written to disk or transmitted across a network. In a game development context, serialization is used to transfer information to and from the game world. It is generally used when saving and loading data, or in networked multiplayer games. This chapter explains how serialization works in Havok, and how best to use it in your games.

2.3.2 Features

- High performance binary format.
  - Binary file format is a memory image.
  - Loading consists of single raw read and very fast pointer patching.
  - Loading is done without any memory allocation.
- Flexible XML format.
  - Human readable XML format.
  - Cross platform load and save.
  - Can process with standard XML tools.
  - Can losslessly convert between XML and binary formats.
- Asset pipeline support.
  - Cross platform export. Generate console assets directly from a PC.
  - Files are introspectable. Files optionally contain the metadata for its contents.
  - Metadata is reusable by tools outside the SDK.
- Forward and backward compatibility
  - Don't need to re-export your assets when your objects change.
  - Objects can be optionally converted at load time.
  - Versioning info may be stripped for smaller assets.
  - Can add custom versioning code when automatic versioning is not enough.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

  - Versioning code can be stripped for final assets.

- Small runtime footprint.
  - Configurable size/features tradeoff.
  - Binary loading code is very small.

- Extensible
  - File format is flexible.
  - Tools supplied to serialize user classes.
  - Building blocks supplied so custom file format/streams are possible.
  - Supports generic tweakers/editors through object reflection.

2.3.3 Packfiles

Any data object can be serialized using the Havok serialization tools. When an object is serialized, all the information needed to accurately capture its state is saved. Serialized objects can then be loaded into your game as desired.

Serialized data is stored in structures called packfiles, which can be saved in XML or binary form. Data from Maya and 3D Studio Max can be saved in packfiles using the Havok Exporters, and then loaded into the Havok SDK.

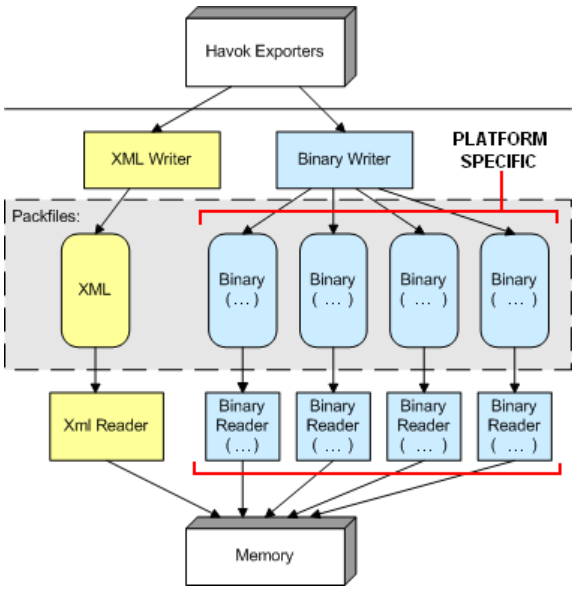


Figure 2.1: Loading packfiles.

Packfiles are loaded into your games using `hkXmlPackfileReader` and `hkBinaryPackfileReader`. XML packfiles are platform independent; therefore XML packfiles can be created on any platform and loaded onto any platform. However, binary packfiles are platform specific, so they can only be loaded by a reader running on the specified target. See also the best practices section.

Packfiles can also be converted from one type to the other. XML packfiles can be converted to binary for any platform using a `hkXmlPackfileReader` to read the XML, followed by a `hkBinaryPackfileWriter`.

Binary packfiles can also be converted to XML using a `hkBinaryPackfileReader` on the appropriate target, followed by a `hkXmlPackfileWriter`.

2.3.4 Loading Game Data

2.3.4.1 hkLoader Utility Class

If you have a file produced from the havok exporters, `hkLoader` takes care of the details. It will detect whether the file is XML or binary. Also if the asset was produced for a different version of the sdk, `hkLoader` can update the contents to the latest version on load.

```
hkLoader loader;
hkRootLevelContainer* container = loader.load("filename.hkx");
// loader owns the loaded data. You must ensure loader is
// not destroyed while you access container.
```

In the longer term, you may wish to use the lower level interface. This has a lower code footprint if for instance, you do not need the XML parser and loader or do not need the version compatibilty functions. The lower level interface also allows inplace loading of pre-read data.

2.3.4.2 Loading Example

The following code example illustrates how to load an XML packfile into your game. Note that the result returned from the reader must be cast as the correct type. In this case an instance of the `SimpleObject` class is loaded.

```
// Load the file contents. Pointers are automatically fixed up.
hkIstream iStream(FILENAME);
hkXmlPackfileReader reader;

reader.loadEntireFile(iStream.getStreamReader());
SimpleObject* object = (SimpleObject*)reader.getContents( &SimpleObjectClass );
```

The binary format may be loaded inplace without any I/O nor any memory allocations.

```
char* myBuffer; int myBufferSize; // user has preloaded buffer
hkBinaryPackfileReader reader;
reader.loadEntireFileInplace( myBuffer, myBufferSize );
SimpleObject* object = (SimpleObject*)reader.getContents( &SimpleObjectClass );
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

2.3.4.3 'Finishing' Loaded Objects

Since pointers contain memory addresses their targets are lost during serialization. However, if objects linked by pointers are serialized together then the packfile reader will fix any pointers as it loads the objects into your game.

Additional processing is often necessary after a packfile has been loaded. For instance, virtual function tables need to be set up for polymorphic objects. Also, cached data or pointers to non-serialized data may need to be initialized. To achieve this, any classes which require post-processing or 'finishing' are registered with a `hkTypeInfoRegistry`. Once the data has been loaded, the packfile reader calls the registered function for each object that needs finishing.

Usually this finish function will be the finish constructor (the one taking a `hkFinishLoadedObjectFlag`).

See the registries section for more information.

2.3.4.4 Versioning Support

The layout of the objects in the packfile need not match the layout of objects in memory. It is possible to update the loaded objects to match the in-memory versions.

The process is as follows: The file is first loaded and the memory image reconstructed. At this point all layouts are from the older version. Next, the version registry compares the object version embedded in the packfile to the current version and we choose a sequence of update functions to call. Each update function generally converts between two fixed versions (e.g. 3.0.0 to 3.1.0). Finally, we get the contents as before.

The list of updated functions is usually set at compile time in the variable `hkVersionRegistry::StaticLinkedUpdaters[]`. For example, in the havok demos this is in `demo/demos/AnimationClasses.cpp`, `PhysicsClasses.cpp` or `CompleteClasses.cpp` depending on your product. The supplied updater functions are defined in the `hkcompat` library.

```
hkIstream infile("file.hkx");
hkBinaryStreamReader reader;
reader.loadEntireFile( infile.getStreamReader() );
\emph{\emph{hkResult updateStatus = hkVersionUtil::updateToCurrentVersion( reader, hkVersionRegistry::
  getInstance() );}
if( updateStatus != HK_SUCCESS )
{
  // Updating failed. Most likely because the relevant updater was not
  // registred in hkVersionRegistry.
  // Applying the vtable fixups and getting contents will now
  // have unpredictable results.
}}
MyContents* contents = (MyContents*)reader.getContents( &MyContentsClass );
```

Note that if you're using the `hkLoader::load(char*)` or `hkLoader::load(hkStreamReader*)` convenience methods, they call `updateToCurrentVersion` for you with the `hkVersionRegistry` singleton.

```
hkLoader loader;
MyContents* contents = (MyContents*)loader.load("file.hkx", MyContentsClass );
\emph{// Contents are now updated to the current version. If the file could not
// be loaded or the contents could not be versioned (for instance if
// hkVersionRegistry::StaticLinkedUpdaters[] is empty) contents is null.}
```

2.3.4.5 Stream Requirements

If you are using custom streams, note that the readers may require some optional interfaces to be implemented. See the hkbase stream section for a shortcut to implementing the mark interface.

hkXmlPackfileReader	loadEntireFile	*mark*
hkBinaryPackfile	loadEntireFile	none
hkBinaryPackfile	loadEntireFileInplace	n/a
hkBinaryPackfile	loadSection	*seek*

2.3.4.6 Imported/Exported Objects

A packfile may contain pointers to data which is not contained in the packfile. e.g. created in code or located in another packfile. We call each pointer an *Import*. Imports are identified with plain C strings. Similarly a packfile may expose some of its contained objects using an *Export*. Use `hkPackfileData::getImportsExports()` to examine the Imports and Exports.

We do not impose a policy of how these references should be resolved, but a sample linker which places all identifiers into a single global namespace is included in `demos/Utilities/Asset/ResourceLinker.h`. The SymbolImportExport demo shows its usage.

There are several approaches to setting up imports/exports. If you have relatively few objects (e.g. mesh shapes), a simple option is to call `addExport` and `addImport` before `setContents`. For an indeterminate number of objects, you can use the `addObjectCallback` to select by adddress or type.

2.3.4.7 hkNativePackfileUtils

We also provide a special low level loader which loads from an in-memory packfile image into a single user-supplied buffer, discarding temporary data such as pointer patch tables. Also, using only this API for loading means that all the versioning code, compiled-in metadata and generic loading code can be dead stripped.

This loader does not support versioning, so you'll need to be sure to supply absolutely up-to-date assets. To minimize this burden during development, see the NativePackfileLoadDemo. A single #define enables and disables versioning support while the main body of code uses the minimal loader interface.

2.3.5 Saving Game Data

2.3.5.1 Saving Objects

In-game objects which have metadata can also be serialized and saved as packfiles, using the `hkXmlPackfileWriter` and `hkBinaryPackfileWriter` classes. If a pointer to an object without metadata is encountered, it is skipped and a null pointer is written.

```
hkOstream ostream(FILENAME);
hkXmlPackfileWriter writer;
writer.setContents(&simpleobject, SimpleObjectClass);
hkPackfileWriter::Options options; // use default options
writer.save(ostream.getStreamWriter(), options);
```

2.3.5.2 Exporting to a Different Platform

You can use `hkBinaryPackfileWriter` to export packfiles to platforms with different binary layouts. Do this by setting `hkPackfileWriter::Options::m_layout` to the desired platform. `hkStructureLayout` contains several static layouts corresponding to supported compiler/platform combinations. Use a `hkBinaryPackfileReader` on the target to read in the data as usual.

```
hkOstream ostream(FILENAME);
hkBinaryPackfileWriter writer;
writer.setContents(&simpleobject, SimpleObjectClass);
hkPackfileWriter::Options options;



// export the data in PlayStation(R)2 gcc3.2 format
options.m_layout = hkStructureLayout::Gcc32Ps2LayoutRules;
writer.save(ostream.getStreamWriter(), options);
```

If your platform does not have a static entry in `hkStructureLayout`, you can find the correct values by examining the `hkStructureLayout::HostLayoutRules` on the target platform. This always corresponds to the running programs rules.

2.3.5.3 Stream Requirements

If you are using custom streams, note that the writers may require some optional interfaces to be implemented. See also hkArrayStreamWriter which provides all required interfaces.

hkXmlPackfileReader	save	none
hkBinaryPackfile	save	*seek*

2.3.5.4 Embedded Metadata

Assets can optionally include metadata for all contained objects. This allows tools to operate on the files, even if the asset is newer than the tool itself. In many cases however, the application loading the asset may have metadata compiled in (`hkVersionRegistry` contains stores the available metadata). In this case, the file metadata is redundant and may be stripped by setting `hkPackfileWriter::Options::m_writeMetaInfo` to false.

2.3.6 Type Registration

2.3.6.1 Uses of the Registry

Much of the serialization infrastructure relies on having registries of `hkTypeInfo` and `hkClass` objects.

saving : vtable to hkClass: When saving polymorphic types, we need to know their exact type in order to save them correctly. This is done by registering each type's virtual table in a `hkVtableClassRegistry` along with its `hkClass` instance.

loading : name to finish function: When loading polymorphic types or types which need post-load "finishing", we need to be able to call an initializing function on the object, based on the type name. `hkTypeInfoRegistry` stores this information.

To ease the burden of creating and initializing these registries, there is the `hkBuiltinTypeRegistry` singleton which contains the above registries and is created once using a compiled-in list of types. Most serialization functions have variants which take explicit registries and ones which implicitly use the builtin registry. e.g. `hkPackfileWriter::setContentsWithRegistry()` and `hkPackfileWriter::setContents()`.

2.3.6.2 Register All Types

To start with, the easiest thing to do is to register all classes. To do this use the following code snippet, uncommenting the line relevant to your product.

```
//#define HK_CLASSES_FILE <Common/Serialize/ClassList/hkCompleteClasses.h>
//#define HK_CLASSES_FILE <Common/Serialize/ClassList/hkAnimationClasses.h>
//#define HK_CLASSES_FILE <Common/Serialize/ClassList/hkPhysicsClasses.h>
#include <Common/Serialize/Util/hkBuiltinTypeRegistry.cxx>
```

2.3.6.3 Custom Registration

Registering all types is easy but it can lead to bloat because storing virtual tables can cause linkers to link all the methods referenced from the virtual table. You can strip these methods by linking only the used types.

To set up the class registry, create a file which contains declarations of the types used. e.g. `MyClasses.h`

```
// these are always used
HK_STRUCT(hkClass)
HK_STRUCT(hkClassEnum)
HK_STRUCT(hkClassMember)

// declare non polymorphic types with HK_STRUCT
HK_STRUCT(SimpleObject)

// declare polymorphic types with HK_CLASS
// e.g. if BaseObject is an abstract class and
// PlayerObject, EnemyObject are concrete derived types.
HK_CLASS(PlayerObject)
HK_CLASS(EnemyObject)

// abstract base types should not be registered
// HK_CLASS(BaseObject)

// We can use the usual preprocessor directives
#if defined(EXTRA_TYPES)
#  include "MyExtraTypes.h"
#endif
```

Next, use the super macro file `hkBuiltinTypeRegistry.cxx` to generate the correct definitions. e.g. `MyClasses.cpp`

```
#define HK_CLASSES_FILE "MyClasses.h"
#include <Common/Serialize/Util/hkBuiltinTypeRegistry.cxx>
```

Note:
The lists in Common/Serialize/ClassList/*.h provide a good starting point for your custom class lists. You can remove an entire libraries classes by commenting out a single line.

2.3.7 Memory Considerations

2.3.7.1 Flattened Memory

Objects loaded from packfiles via the `loadEntireFile()` method are stored contiguously in memory, making memory management easy.

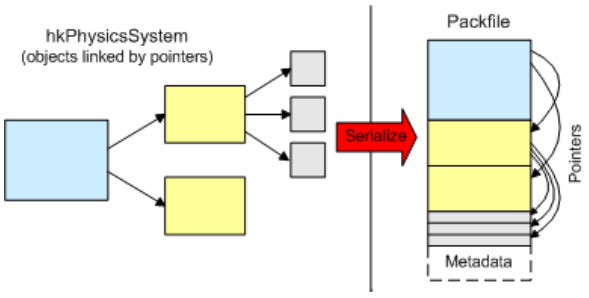


Figure 2.2: The internal structure of a packfile.

The hkPackfileData class contains (among other things) a handle to the memory allocations made when loadEntireFile() is called and any extra allocations which may result from versioning. Once a reference is added to the packfile data, the packfile reader can be deleted from memory. The memory containing the loaded data can be deallocated at any time by removing this reference.

```
// Get a handle to the memory allocated by the reader, and add a reference to it.
// This allows us to delete the reader.
m_packfileData = reader->getPackfileData();
m_packfileData->addReference();

delete reader;

// Later on, we can deallocate the memory containing the loaded data.
m_packfileData->removeReference();
```

2.3.7.2 hkArray resizing

hkArrays loaded into memory from a packfile are represented as a pointer and a size, with the array elements stored further down in the memory block. Most hkArrays loaded from a packfile will have a fixed size (i.e. their LOCKED_FLAG is set). However, objects can also contain dynamic hkArrays, which can be increased in size after being loaded. If a hkArray is increased in size its elements are copied elsewhere in memory to allow it to expand, as can be seen in the diagram.

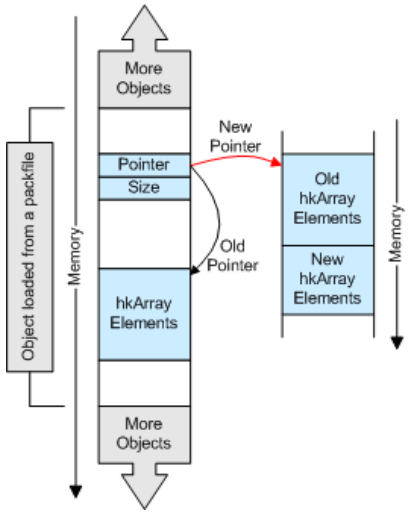


Figure 2.3: Memory re-allocation for arrays.

Therefore, objects that contain hkArrays and were loaded from a packfile must be parsed before they are deleted, and destructors called for any hkArrays found in case they have moved elsewhere in memory. The destructor checks the hkArray's LOCKED_FLAG and DONT_DEALLOCATE_FLAG, and if neither flag is set then the hkArray has moved and gets explicitly deallocated.

Note that once an array has moved its original memory is not referenced by anything and cannot be freed up until the section of memory that the packfile was loaded into is deallocated.

2.3.7.3 Reference counted objects

Reference counted objects are usually freed when their reference count falls to zero. However, because multiple objects are packed into a single allocation when serialized, it is not possible to free individual objects. Therefore, calls to increment and decrement the reference count of objects loaded from packfiles are ignored. More precicely, `addReference` and `removeReference` calls are ignored for objects which have `hkReferencedObject::m_memSizeAndFlags==0`.

2.3.7.4 Destructors of packfile objects

When objects with virtual tables are loaded, we call the finish constructors (see Finishing). The `hkPackfileData` remembers these constructors and will automatically call the corresponding destructors when the data is destroyed. Note that we almost always call `removeReference` in destructors instead of an explicit `delete`, because the former correctly handles packfile and heap objects.

Previous versions of havok did not call any destructors and required the user to explicitly call them before the data is destroyed to avoid memory leaks (such as from resized `hkArray`s, or from referenced counted objects created outside of the packfile). You can restore the old behavior with `hkPackfileData::disableDestructors()`.

Note that if you have several packfiles which reference each other, you will need to do cleanup in two passes - first call all the destructors (`hkPackfileData::callDestructors()`), then free the memory (`hkPackfileData::~hkPackfileData()`). This avoid problems such as calling `removeReference()` on memory which has already been freed.

2.3.7.5 Sections

Whereas XML packfiles are monolithic in structure, binary packfiles can be split into sections for modular saving and loading. By default binary packfiles are split into two sections; one containing type information for the serialized objects and the other the objects themselves.

However, it is possible to implement finer granularity if desired, using the `setSectionForClass()` and `setSectionForPointer()` methods in `hkPackfileWriter` to create custom sections.

- `setSectionForClass()` is passed a reference to an instance of `hkClass`. Any data of the specified class type is grouped into a section.
- `setSectionForPointer()` is passed a pointer to an object in memory. This object is serialized in its own section, even if it qualifies for inclusion in a 'class' section.

Both of these methods also take a section tag, which is used to identify the section. The tags for the default implementation are '`__data__`' and '`__types__`'.

To load data section by section use the `loadSection()` method in `hkBinaryPackfileReader`.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

2.3.7.6 Compiler independent types

The size and alignment characteristics of some types (most notably bool, enum and bitfields) vary between compilers or with compiler options. We define several types to combat this.

- `hkBool` - a bool which is always 1 byte in size.

  Note that while `hkBool` is typically used for object member variables, `hkBool32` (a typedef for int) is sometimes used in functions and local variables as it is more optimization friendly.

- hkEnum<enum,storage> - an enumeration with explicitly defined size

  hkEnum stores exactly one value of the enumeration. The storage type can be 8,16 or 32 bits.

- hkFlags<enum,storage> - a fixed sized bitfield of named values

  hkFlags stores bitwise combinations of the values defined in the enumeration. The storage type can be 8,16 or 32 bits.

2.3.8 Havok Tools and Data

2.3.8.1 Exporter Output Format

For consistency all Havok exporters export the same top level data structure. This data structure can be found in `hkserialize/util/hkRootLevelContainer.h` - it simply contains an array of named Variants. Each Variant is a structure containing a pointer to some data together with a hkClass structure describing how that data is laid out in memory.

Note however that the top level object is not fixed and some routines use different objects. e.g. the snapshot utility saves a `hkpPhysicsData` object.

Utility routines (findObjectByType and findObjectByName) are provided to make searching this root level container easier. In addition we provide a simple loader which will load one or more files containing this top level container. The loader maintains a reference to the each chunk of loaded data. This reference is not removed until the loader is destroyed. The loader can be found in `hkserialize/util/hkLoader.h` and is used throughout our demos.

2.3.8.2 Asset Conditioning Tools

`Tools/PackfileConvert` contains some sample tools for use in your asset pipeline (AssetCc1 and AssetCc2). A common approach is to use these tools to convert assets into their final game format. (See below)

2.3.8.3 Best Practice

You can export directly from a content tool to a game-ready platform specific format, which is easy and gives great turn-around times. However there are scalability disadvantages to this approach such as:

- It usually more awkward to automate export from a content tool.

- The produced file is only readable on the target system and thus is not amenable to further processing on the tools platform.
- There is no place to perform automated sanity checking.
- Using runtime versioning leads to runtime allocations and slower loading times.

Thus we recommend exporting initally to an easily malleable intermediate format (e.g. win32 or win64) so that you are insulated from changes in the target platform and can easily add custom processing steps such as verification or policy overrides before the final conversion into platform-specific binary.

2.3.9 Serializing Havok Physics Objects

When serializing Havok Physics objects, only the properties needed to recreate the object later on are saved - such as position, orientation, dimensions and impulses, but not contact points. There are a number of classes in the Havok Physics SDK worth discussing with regard to serialization.

2.3.9.1 The hkpRigidBody and hkpPhantom Classes

`hkpRigidBody` and `hkpPhantom` are two serializable Havok Physics classes containing `hkArray`s which are resized during simulation. Particular care needs to be taken when deallocating a section of memory loaded from a packfile if it contains either of these objects. As described in the Memory Considerations section, the `hkArray` destructor needs to be called for their arrays in case they have increased in size and are now located elsewhere in memory.

2.3.9.2 Userdata pointers

Some objects contain userdata pointers. They are typeless and may contain integer, float or pointer values. By default havok saves them as zeros. If in fact they are pointers to serializable objects, you can have the system treat them as such.

To do this, edit the header file and add an `//+overridetype()` comment after the `member` element. For instance if the pointer is really a shape pointer, you could use the following. After making this change, you will need to run the `build/serialize/regenerateXml.py` script and recompile the generated `Class.cpp`.

```
void* m_userData; //+overridetype(hkpShape*)
```

Alternatively, you can edit the `Class.cpp` file directly. e.g. in `hkpWorldObjectClass.cpp`. Note that running `regenerateXml.py` overwrites the `Class.cpp` files.

```
// This is the default, as shipped.
{ "userData", HK_NULL, HK_NULL, hkClassMember::TYPE_ZERO, hkClassMember::TYPE_POINTER, 0, 0, HK_OFFSET_OF(
  hkpWorldObject,m_userData) }
// This sets the userdata to be a shape pointer as in the previous example.
{ "userData", &hkpShapeClass, HK_NULL, hkClassMember::TYPE_POINTER, hkClassMember::TYPE_STRUCT, 0, 0,
  HK_OFFSET_OF(hkpWorldObject,m_userData) }
// This sets the userdata to be a 32 bit integer
{ "userData", HK_NULL, HK_NULL, hkClassMember::TYPE_INT32, hkClassMember::TYPE_VOID, 0, 0, HK_OFFSET_OF(
  hkpWorldObject,m_userData) }
```

Note:
When overriding a pointer as an integer, you'll need to be careful with 32/64 bit issues. Overriding a type as a type of different size can cause the binary writer to fail. In the case of `hkpWorldObject`, the alignment requirements of adjacent variables ensures that the object layout is still computed correctly for 64 bit machines.

2.3.9.3 The hkpPhysicsSystem Class

A `hkpPhysicsSystem` is a container class which stores collections of objects and their associated actions and constraints for serialization together. Conceptually, a physics system will be a single in-game entity that is always considered as a whole, such as a ragdoll or a vehicle. Even if two ragdolls are lying on top of each other, and hence in the same simulation island, they are separate game entities and can be considered as two separate physics systems for the purposes of serialization.

`hkpPhysicsSystems` are made up of just four component types: `hkpRigidBody`, `hkConstraint`, `hkpAction` and `hkpPhantom`. For example, ragdolls consist of a number of rigid bodies (usually eleven - two for each limb and the torso, and one for the head), and the constraints that hold them together, whereas vehicles are generally defined by a rigid body, a phantom, and an action. Collections of objects such as piles of crates or barrels can also be stored in a single `hkpPhysicsSystem`.

2.3.9.4 The hkpPhysicsData Class

The `hkpPhysicsData` class is the top level container class for Havok Physics data. It contains a `hkpWorldCinfo` and a `hkArray` of `hkpPhysicsSystems`, allowing anything from groups of ragdolls to entire game levels to be serialized together. A `hkpPhysicsData` scene can be captured and serialized at any time via `hkpHavokSnapshot`.

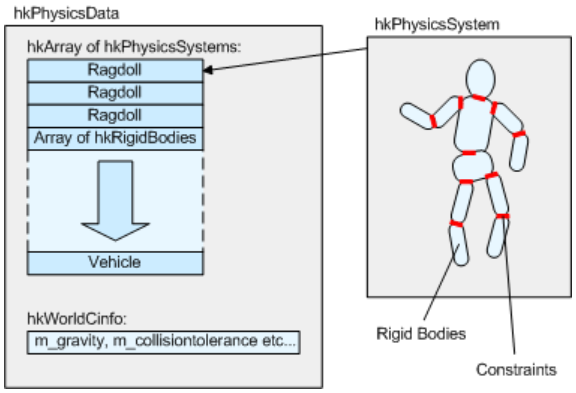


Figure 2.4: hkpPhysicsData and hkPhysicsSystems

The simpleLoadApi demo (in demos/api/serializeApi) demonstrates how to load Havok Physics data from a packfile. The getContents() method of the packfile reader is used to find the data loaded into memory from the packfile. This data is cast as a hkpPhysicsData object, which is used to populate the hkpWorld.

```
// Load the entire file
reader->loadEntireFile(infile.getStreamReader());
```

```
// Get the top level object in the file
hkTypeInfoRegistry reg;
hkRegisterHavokClasses::registerAll(reg);
hkpPhysicsData* data = static_cast<hkpPhysicsData*>( reader->getContents( &hkpPhysicsDataClass, &reg ) );

HK_ASSERT2(0xa6451535, data->getWorldCinfo() != HK_NULL, "No physics cinfo in loaded file - cannot create
  a hkpWorld" );

// Create a world and add the physics systems to it
m_world = new hkpWorld( *data->getWorldCinfo() );

// Register all collision agents
hkpAgentRegisterUtil::registerAllAgents( m_world->getCollisionDispatcher() );

// Add all the physics systems to the world
for ( int i = 0; i < data->getPhysicsSystems().getSize(); ++i )
{
    m_world->addPhysicsSystem( data->getPhysicsSystems()[i] );
}
```

2.3.9.5 Simple arrays

Simple arrays are provided as an alternative to khArray to create read-only arrays. A pair of typed pointer and int with specified naming convention is treated as a simple array. The type specifies array item type and must be one of the types supported by serialization framework, and int to store size of simple array. Currenly supported types are plain types, structures and c-strings. The names of the two class data members or structure members must follow the rule:

```
T* m_<member_name>; // T - type, <member_name> starts with non-capital letter
int m_num<member_name>; // <member_name> must start with capital letter
```

The following snippet demonstrate examples of declaration of simple arrays:

```
class MyClass
{
    public:
        HK_DECLARE_REFLECTION();
        ...
        // simple array of chars
        char* m_dataChar;
        int m_numDataChar;

        // simple array of structs
        struct MyStruct* m_dataStruct;
        int m_numDataStruct;

        // simple array of c-strings
        char** m_dataString;
        int m_numDataString;

        // not a simple array
        char* m_chars;
        int m_num; // to make simple array, change name m_num to m_numChars
};
```

The simple array item size must be known at compile time, so pointer to void type is not supported.

2.3.9.6 Homogeneous arrays

Homogeneous arrays extend functionality of simple arrays, and needed only when the type of data is unknown at compile type. A triple of hkClass pointer, void* and int with specified naming convention is treated as a homogeneous array:

```
hkClass* m_<member_name>Class; // <member_name> starts with non-capital letter
void* m_<member_name>; // <member_name> starts with non-capital letter
int m_num<member_name>; // <member_name> must start with capital letter
```

This is an example of homegeneous array:

```
class MyBuffer
{
    public:
        HK_DECLARE_REFLECTION();
        ...
        /// The pointer to a generic data buffer
        const hkClass* m_bufferDataClass;
        void* m_bufferData;
        int m_numBufferData;
};
```

The homogeneous arrays are rarely used and in most cases simple arrays and hkArray do the job.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

2.3.9.7 Havok Tools Output

The Havok tools for Maya and 3DS Max group Havok Physics objects into an instance of `hkpPhysicsData`, which is serialized and saved as a packfile. This `hkpPhysicsData` object contains a `hkpWorldCinfo` and a `hkArray` of `hkpPhysicsSystem`s as usual, with the physics objects serialized from the exporters grouped into `hkpPhysicsSystem`s in the following order:

1. Fixed Objects - All fixed objects are saved into the first `hkpPhysicsSystem`. Some fixed objects (e.g. the base of a chandelier) may be attached to the world and also conceptually belong to a `hkpPhysicsSystem` (the chandelier). To avoid such systems being grouped into one large `hkpPhysicsSystem` all fixed objects are isolated from any systems they would otherwise be a member of.
2. Dynamic Objects - All dynamic objects without constraints (e.g. crates, barrels etc.) are stored in the second `hkpPhysicsSystem`.
3. Keyframed Objects - All keyframed objects (such as moving platforms etc.) make up the third `hkpPhysicsSystem`.
4. Other Physics Systems - A `hkpPhysicsSystem` is created for each set of objects in the scene linked by constraints, such as ragdolls, vehicles etc. Any groups of objects constrained to a fixed object are saved as a `hkpPhysicsSystem` without the fixed object (e.g. the aforementioned chandelier is saved as a `hkpPhysicsSystem` without its base).
5. Phantoms - Any remaining standalone phantoms are grouped together in the final `hkpPhysicsSystem`.

2.3.10 Snapshot Utility

Havok includes a snapshot utility which can easily load and save your world. The utility takes care of most of the underlying details.

2.3.10.1 Saving a snapshot of the world

Note that when the file is being saved, the snapshot may print warning messages to the console if it encounters objects which it does not understand. See troubleshooting.

```
// Save the world into the file at "path"
static hkResult saveWorld( hkpWorld* world, const char* path, bool binary )
{
    hkOstream outfile( path );
    return hkpHavokSnapshot::save(world, outfile.getStreamWriter(), binary);
}
```

2.3.10.2 Loading a world snapshot

Note the `allocsOut` parameter for tracking allocations. See Memory Considerations

```
// Loads snapshot at "path".
static hkpWorld* loadWorld( const char* path, hkPackfileReader::AllocatedData** allocsOut )
{
    hkIstream infile( path );
    hkpPhysicsData* physicsData = hkpHavokSnapshot::load(infile.getStreamReader(), allocsOut);
    return physicsData->createWorld();
}
```

2.3.10.3 Troubleshooting snapshots

You can verify that your snapshot worked by changing the path in the `api/serializeapi/SimpleLoadApi` demo to point to your snapshot file. If the snapshot does not load, it is most likely that when saving, the snapshot encountered an object which it did not know how to deal with. In this case, the snapshot warns and replaces the object pointer with a null pointer.

Havok serialization uses objects vtables as type identifiers. When a polymorphic pointer is encountered (i.e. hkpShape*), we check a registry of vtable to hkClass instances to determine its exact type and thus how to save it.

If some of your object types are not in the registry (`hkVtableClassRegistry`) you have several choices.

- Register it as its base type

  This is the quickest method to implement and is a good choice if you are snapshotting for debugging and are not particularly interested in saving the extra information in derived classes.

  e.g. you have a class `MyRigidBody` which inherits from `hkpRigidBody`. You can cause the snapshot to treat them as if they were rigid bodies using the following code.

  ```
  hkVtableClassRegistry* reg = hkBuiltinTypeRegistry::getInstance().getVtableClassRegistry();
  reg->registerVtable( *(void**)anInstanceOfMyRigidBody, &hkpRigidBodyClass);
  ```

- Convert it to another type

  If there is no suitable base type (e.g. you derive directly from `hkpShape`) or your shape has data which you would like to save, you can save a proxy object instead.

  When objects are added to a packfile, there is a callback fired which allows one to modify or replace objects. For instance some classes (`hkpExtendedMeshShape`) contain exernal data not owned by havok. When saving, we can convert such types to "self contained" equivalents. (`hkpStorageMeshShape`).

  We can even change a saved objects type entirely. For instance to replace `MyShape` instances with `hkpCapsuleShape` instances.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
hkVtableClassRegistry* reg = hkBuiltinTypeRegistry::getInstance().getVtableClassRegistry();
const hkClass MyShapeClass(0,0,0,0,0,0,0,0,0,0); // dummy hkClass
reg.registerVtable( *(void**)anInstanceOfMyShape, &MyShapeClass);
// hkpHavokSnapshot::save( ... );
```

```
// in hkpHavokSnapshot.cpp
void StorageListener::addObjectCallback( ObjectPointer& ptrInOut, ClassPointer& klassInOut )
{
    extern const hkClass MyShapeClass;
    if( hkpExtendedMeshShapeClass.isSuperClass(*klassInOut) && klassInOut != &
      hkpStorageExtendedMeshShapeClass )
    {
        // convert meshes to storage meshes
    }
    else if( MyShapeClass.isSuperClass(*klassInOut) )
    {
        ptrInOut = new hkpCapsuleShape( myShape->getVertex(0), myShape->getVertex(1), 1);
        klassInOut = &hkpCapsuleShapeClass;
        m_storage.pushBack(ptrInOut); // we own the new proxy capsule shape
    }
}
```

- Add serialization information

  This is the highest fidelity method as your object is saved as it exists in memory. This is the best method if you need to reload the scene from a saved snapshot.

  This topic has its own section

2.3.11 Using the Infrastructure

This section describes how to make your custom class work with havoks serialization infrastructure.

Havok serialization is an *object reflection* based system. Each object type has a data block which describes the members of such objects. Object loading, saving and manipulation is done in a generic fashion using this reflection data.

2.3.11.1 Generating the Reflection Data

To ease the burden on the programmer, the reflection data does not have to be manually created, but can be extracted directly from C++ class definitions. The script (`tools/serialization/regenerateXml.py`) parses the C++ definition and writes the reflection data (C++ code) to the same directory as the header file. Only classes containing the `HK_DECLARE_REFLECTION();` declaration will be considered for reflection.

e.g. If a header `Foo.h` contains `class Foo`, the generated file will be named `FooClass.cpp` and will contain one const `hkClass` instance `FooClass`. You may wish to add an extern declaration to `Foo.h` (`extern const hkClass FooClass`) for users of `Foo`.

C++ parser notes

The parser is not a full ansi C++ parser. In particular, it scans each file independently and does not chase includes. Thus sometimes it does not have enough context to know exactly what a symbol denotes. e.g. given the member declaration "`Foo` m_foo;" it does not know if `Foo` is an object, enum or typedef.

You can disambiguate these cases with a little context:

- Declare non-pod member types unambiguously.
  e.g. "Foo m_foo;" -> "class Foo m_foo;"
- Add type aliases to `build/serialize/typeMapping.txt`
  e.g. Add the line "MyUnsigned32:hkUint32"
- Override the type explicitly with a specially formed comment.
  e.g. hkpShapeKey m_shapeKey; //+overridetype(hkUint32)

Also note that classes which require vtables (i.e. classes that have virtual member functions) must include the keyword "virtual" at least once in the class definition in order to enable proper vtable fixup at runtime. This requirement applies even to sub-classes, where in C++ the keyword is often optional.

Infrastructure notes

Because of the difficulty of handling multiple inheritance, we have decided to support only java-style inheritance. That is each class has at most a single parent, but may implement multiple interfaces (inherit from multiple pure abstract classes). The parent class must be the first class in the inheritance list.

Metadata overrides and annotations

Metadata annotations are specially formed comments which are extracted by the metadata generation script. They have the form `+identifier(value)`. Each annotation is applied to the *previous* item. Several annotations may be made per item. e.g.

```
struct MyStruct
{
  //...

    // TableIndex is a typedef for int.
    // Alternatively we could override globally in typeMapping.txt (see above)
  TableIndex m_index; //+overridetype(int)

    // This
  Reflected* m_pointer; //+serialize(false)

    // We don't save the cache - it is set to zero on load.
    // Cache is not a reflected object so "+serialize(false)"
    // won't work here.
  Cache* m_cache; //+nosave

    // Aggregate type default.
  hkVector4 m_startPos; //+default(100,100,0,0)
};
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The most important member annotations (with default values if unspecified) are:

+serialized(boolean=true): Indicates that the member should not be saved when taking a snapshot. By default all members are serialized.

+nosave: Indicate that a member should not be serialized. When serialized, the space will be zero filled. This is similar to +serialized(false) except that additionally the member is only partially reflected. e.g. `member.getType()` will be correct but `member.getClass()` will return null.

+overridetype(newtype): Force the metadata for member to be "newtype"

+default(value): Default values are especially useful for versioning - defaults are applied to new members when versioning.

+reflected(boolean=true): If false, the member is completely absent from metadata. Classes which have nonreflected members cannot be serialized.

+owned(boolean=true): For a pointer, indicates that the pointed to memory should be included in memory statistics for this object.

For classes:

+abstract(boolean=false): The class is abstract. Sometimes the C++ parser cannot detect that a class is abstract because it only sees the local definition and misses inherited methods.

+serializable(boolean=true): The class cannot be serialized, probably because it has some non reflected members.

+defineattribute(boolean=false): This class is used as a custom attribute. Currently used by the .NET language bindings which require special treatment for attributes.

For enums:

+defineflags(boolean=false): The enumeration values are intended to be combined bitwise instead of being used as discrete values. Currently used by the .NET language bindings which require special treatment for flags.

Custom metadata annotations

You can attach arbitrary metadata to `hkClass` and `hkClassMember`s. For example:

```
class MyGuiGridPosition // Stores the layout of attributes in the attribute editor grid
{
    //+defineattribute(true)
    HK_DECLARE_REFLECTION();
    int m_row;
    int m_col;
    int m_colSpan;
};
class MyHelpString
{
    //+defineattribute(true)
    HK_DECLARE_REFLECTION();
    const char* m_help;
};
class DebrisGameObject
{
    //+My.HelpString("Attributes can apply to classes and members.")

    hkVector4 m_position;  //+My.GuiGridPosition(1,0)
    hkQuaternion m_rotation; //+My.GuiGridPosition(1,1,1)
    const char* m_name;
        //+My.GuiGridPosition(colSpan=2)
        //+My.HelpString(help="The name is used to resolve the ...")
        //+My.Tooltip("Unique identifier")
};
///...
void GenerateGuiLayout( void* data, const hkClassMember& member )
{
    if( hkVariant* helpStringAttr = member.getAttribute("My.HelpString") )
    {
        MyHelpString* helpString = static_cast<MyHelpString*>(helpStringAttr.m_object);
        //... use the helpString object.
    }
}
```

The custom attribute class should specify `//+defineattribute(true)` when you need to expose it to .NET.

Custom attributes can be arbitrary reflected objects. If you have many attributes, you may spread them over several lines. The mapping from the attribute definition to the C++ data blob is implemented by means of a python function call. You can see that the attribute syntax matches the very flexible python function call syntax, so we can easily support positional, variable length and keyword arguments. See the python docs for how function arguments are mapped.

Note that custom attributes all contain a dot. When the C++ parser finds an attribute which is not built-in (i.e. contains a dot), it will try to import an attribute plugin from `Serialize/attributes` e.g. in this case `Serialize/attributes/My.py` and will call methods from that module to create the metadata. The method call should return a string containing C++ code which is pasted into the `Class.cpp`. See `Serailize/attributes/Example.py` and `Demos/Common/Api/CustomAttributes` for several examples.

The attribute name does not need to match the reflected class name. Instead it can be thought of as simply a tag which is used in two places:

- the string argument to `getAttribute()`
- to find the callable item in the python module

In fact the C++ type corresponds to the fully scoped name of the python class with the periods removed.

Thus same reflected class may be reused for several different attribute names. In the above example, we don't need to define separate classes for HelpString and Tooltip. We reuse the C++ HelpString class for both, using the code below.

```
## Serialize/attributes/My.py

import attributes
#...
class HelpString(attributes.Attribute):
    def __call__(self, value):
        return 'static const MyHelpString %s = { "%s" };' % (self.symbol, value)

# Tooltip is an alias for HelpString
# This line is similar to the C++ "typedef HelpString Tooltip;"
Tooltip = HelpString
```

Virtual Tables

The situation very commonly occurs where we have a pointer to a polymorphic type and need to access its associated hkClass. We do this by assuming that at the start of every polymorphic type is a *virtual table* (vtable), the address of which uniquely identifies each type. In practice, all compilers can be made to adhere to this assumption by having an empty base-most class.

Most "save" methods require such an association (`hkVtableClassRegistry`) as a parameter.

The other common situation is that we have loaded an object and need to initialize its vtable (it may change each time the executable is compiled) and optionally perform some other "load-time" initialization. We associate a string identifier (class name) with initialization function. Often initializing the vtable is as easy as placing the table address at the start of the object. However, it quickly gets more complicated with multiple inheritance when there are multiple vtables.

Most "load" methods require such an association (`hkTypeInfoRegistry`)

Because of the complications of initializing vtables, we let the compiler do the low level work. Each serialializable class should define a special constructor which does not initialize its members. Then calling placement new with the special constructor on the object initializes the vtable without disturbing the loaded values. Note that the flag is chained to the parent to avoid implicitly chaining the default constructor.

```
struct MyClass : public MyBase
{
  HK_DECLARE_REFLECTION();
  MyClass() {} // default constructor zeros the m_indices array
  \emph{MyClass(hkFinishLoadedObjectFlag f) : MyBase(f), m_indices(f) {} // non-initializing constructor}
  hkArray<int> m_indices;
};
```

Multiple Inheritance

For simplicity, we currently support only Java-style inheritance. That is, only the first parent may have data members and all others must be pure interface (no member data) classes. Additionally, virtual inheritance is not supported at all. Because most Havok types (except Listeners) derive from hkRefer-

encedObject, this means that a Havok base class must be the first parent in the hierarchy.

hkTypeInfo

The virtual table details are stored in an `hkTypeInfo` instance. This class stores the types name, vtable and a function which can initialize its vtable. The type info is generated automatically along with `hkClass` for each reflected class.

Summary

Here are the steps to reflect a `Foo` in a file named `Foo.h`

- Mark the class for reflection with `HK_DECLARE_REFLECTION();`
- Run `build/serialize/regenerateXml.py` to generate `FooClass.cpp`
- (optional) Add override types and defaults and rerun script
- Add vtable initializing `constructor (hkFinishLoadedObjectFlag)`

At this point all the infrastructure is in place. All that remains is to add the reflection data to the relevant registries as in the custom registration section.

Chapter 3

Havok Physics

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.1 Introduction

3.1.1 Introduction

Welcome to the Havok Physics SDK, a physics middleware technology for fast, real-time rigid body simulation. Havok can be used in applications where objects need to interact realistically within a 3D space. It has been deployed in over 150 game titles across many platforms and across many game genres, including action adventure, role-playing, first and third person shooter, sports, and vehicle games.

Figure 3.1: a small sample of game genres developed using Havok technology

This chapter introduces you to Havok and how it works within your games, and provides an overview of the Havok architecture. You will find more detailed information about the concepts and components introduced here in the rest of this manual.

3.1.2 What is Havok?

Before getting started, let's look at what Havok can do.

With the SDK, you can create a virtual 3D physical world, create physical objects within the world,

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

assign physical properties to those objects, continuously step the world forward in time, and examine the results of the simulation step. You can customize almost any aspect of the physical behavior during the simulation (e.g. change gravity, apply forces and torques to objects, add and remove objects dynamically).

Havok also provides you with higher level solutions for game genre-specific problems like vehicle simulation (for driving games), human ragdolls (for dynamic simulation of killed enemy characters), physical interaction of keyframed characters within a game environment (to simulate enemies getting hit in different parts of their bodies), character control (for first person or third person games).

In addition, it has a rich product set surrounding the API, including tools for content creation, profiling and debugging (e.g. the Visual Debugger), integration with third-party renderers like 3D Studio Max and Maya (including full source for scripted utilities for easy inclusion in your games existing art export pipeline), and tweaking and tuning.

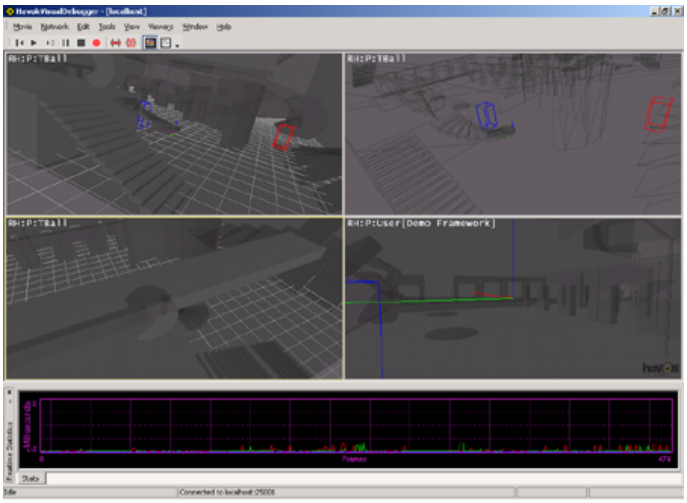


Figure 3.2: The Visual Debugger in action, which is a very useful tool for (remotely) viewing just the physical simulation and studying various in-game performance statistics

Havok is also an open, object-oriented framework. The SDK has been designed to be very modular and allows you to extend or re-implement many of the core components.

Finally, Havok is optimised for all supported platforms. Hand-coded assembly and optimised math routines help Havok to use each architecture in the most efficient way.

However, Havok is *not...*

- A packaged Game Construction Kit. The Havok SDK is just that an SDK. It provides building blocks that can be used and glued together in many different ways. Like any middleware technology, it is a generic technology that can be applied in a variety of game-specific contexts.
- A simple technology. It requires an investment in time to become familiar with its features, its documentation and its interfaces. To use it effectively, it demands a high level of programming expertise and a familiarity with the basic concepts of dynamics.
- A black box. Havok provides low-level access to core functionality so that you can construct complex physical behaviors that are specific to your game and don't come as standard with Havok.
- A commercial renderer. Although we provide a simple cross-platform display library, it is used

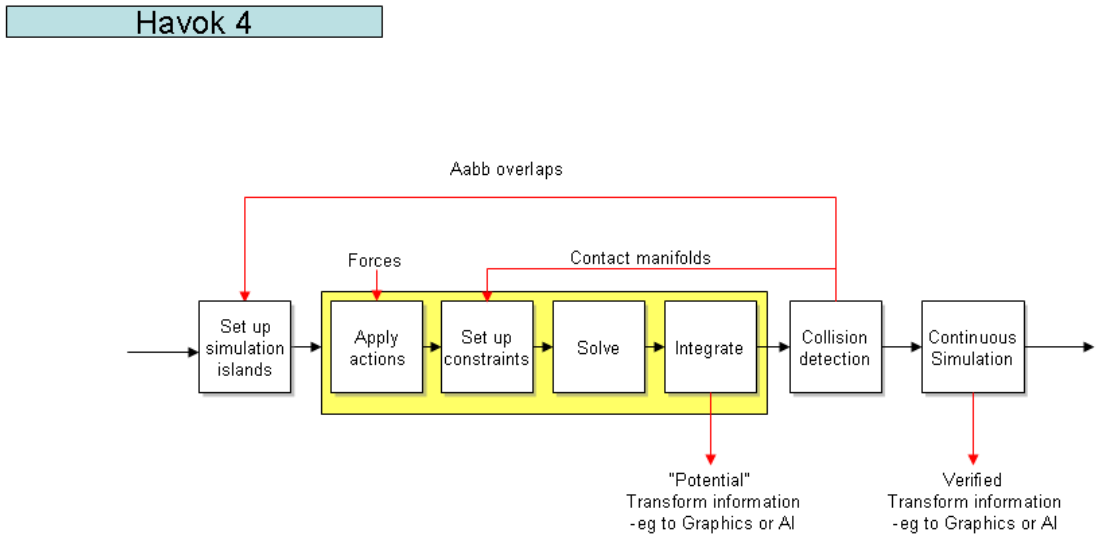
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

merely as a testbed for rapid prototyping and for deploying Havok demos.

3.1.3 How it works

When using Havok physics in your games, you want to create the impression of a continuous simulation and display. However, in practice this means performing single simulation and display operations several times a second - we call each of these time slices a frame. Lets assume that everything has been initialized properly and we are currently in our main game loop. During this frame, the physics engine needs to take a step forward in time and simulate to this step and then output its results for display.

The following diagram shows the work the Havok physics engine needs to do in each frame to step forward the simulation.



3.1.3.1 Setting up simulation islands

During the physics simulation separate groups of objects are partitioned into simulation islands. This allows them to be simulated independently for performance reasons. Simulating small groups of objects separately is good for CPU and memory utilization. In addition, if all the objects in an island are eligible for deactivation (which occurs when an object has come to a definite halt), the island can be deactivated and doesn't need to be simulated at all.

Objects share an island if there are certain physical effects between them. This includes objects that are potentially colliding according to the collision detection broadphase, those that are joined together with constraints, and (in some cases) those that are operated on by the same actions. Simulation islands are set up automatically by the system for these groups of objects. You do not need to set up or maintain simulation islands yourself.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

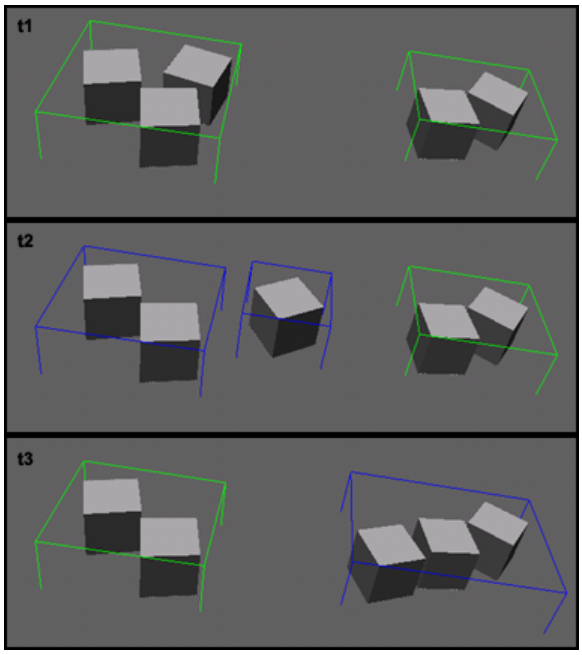


Figure 3.3: An object moving out and back into compound simulation islands

As indicated in the diagram above, there are two inactive (green) simulation islands. Between time t1 and t2, one of the physical objects is moved into the middle of the groups (which activates it, as well as the rest of it's initial simulation island). At time t2, there are three simulation islands, two active and one inactive one.

Between time t2 and t3, the object is placed beside the other set of objects, and is automatically added to their simulation island, activating them as a result. The other simulation island is automatically deactivated by the system.

Simulation islands govern the deactivation system in Havok. Any body that have come to rest in the simulation are eligible for deactivation. A deactivated body doesn't have to be simulated at all, thus saving valuable CPU time. Each body has a deactivator that tells the system when it is safe to deactivate the entity, based on its state.

Using the default deactivator type, if all the bodies in a simulation island are eligible for deactivation, the entire island is deactivated. Once a single rigid body is re-activated due to a collision or a user call, all rigid bodies in that island are reactivated. You can find out more about deactivation in the Deactivation section of the Dynamics chapter.

The following tasks in the step are performed on a per simulation island basis.

3.1.3.2 Applying actions

Actions enable you to control the state of the bodies in the simulation. At each simulation step, the `applyAction()` method of each Action you have added to the world is called by the simulation. You can write your own Actions by implementing the provided interface, or you can use some of the Action classes provided with the Havok Physics SDK, such as the vehicle action. You can find out more about actions in the Actions chapter.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.1.3.3 Setup Constraints

Constraints present in the scene are processed. This includes contact constraints - which are used by the system to prevent objects from interpenetrating - and constraints that you specify yourself, such as hinges or ball-and-socket joints between objects. Custom constraints can also be created.

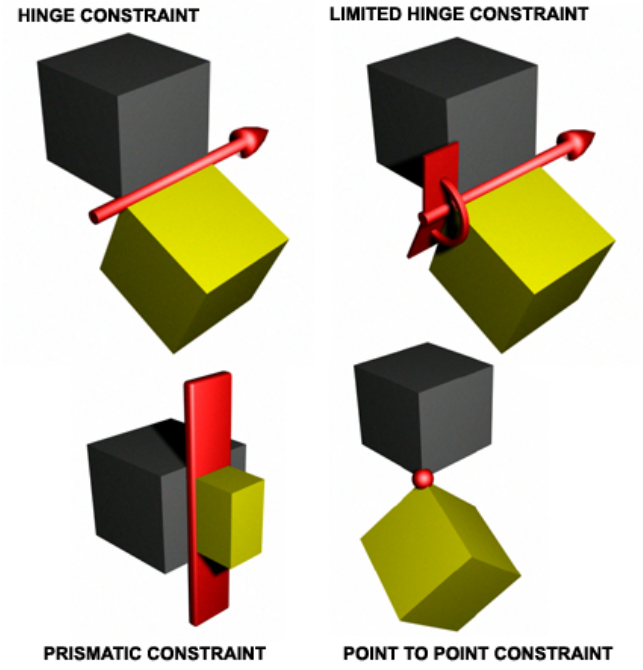


Figure 3.4: A sample of the some of the built-in Havok constraints

3.1.3.4 Solve

In a rigid body system, rigid bodies should not interpenetrate. The depth to which one rigid body is embedded in another, the penetration depth, is a measure of the error in satisfying their contact constraint. Similarly, if, for instance, a pair of objects joined with a ball-and-socket constraint become separated, the measure by which they are separated is an error in satisfying that constraint. Errors in satisfying constraints can be counteracted by moving the objects to a new position. This must be done carefully, however, because moving an object to deal with one error can introduce another error elsewhere. To deal with this, the system has a solver, which performs the next stage in our step.

The solver is responsible for calculating the necessary changes to reduce error for all constraints in the simulation. It passes this information on to the integrator.

3.1.3.5 Integration

The integrator calculates the new motion state (initial and final position and orientation, velocity, acceleration, and so on) for each object in the simulation, taking into account the information provided by the solver. The new positions and rotations can then be passed on to, for example, your renderer to update the corresponding in-game objects. The motion state of an object stores two transforms relating to its position and orientation at the beginning and the end of the simulation step. This way it is easy

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

to calculate position of the object in between the simulation frames. This is advantageous when running a slow-motion visualization and using one simulation step for a number of visualization frames.

Note that the set of operations [Apply actions, Setup Constraints, Solve, and Integration] are sometimes refered to simply as "integrate". For instance the function hkpWorld::integrate() actually performs all these operations.

3.1.3.6 Collision detection

The simulation needs to determine whether any of the objects in your scene are colliding/overlapping. In Havok Physics, collision detection is broken into three phases - broadphase collision detection, which is a quick approximation that finds any potentially colliding pairs of objects, midphase collision detection, which carries out a more detailed approximation of potential collisions, and finally narrowphase collision detection, which determines if any of these pairs are actually colliding. If they are colliding, any intersection information from the narrowphase is used to create collision agents and collision contact points. Those agents and contact points are used by the simulation solver in the following simulation frame.

Each physical object in the simulation has an hkpCollidable, which in turn has an hkpShape that defines the object's shape for collision detection purposes. These shapes can range from simple spheres and boxes to more complicated compound shapes and meshes. The hkpCollidable also has transform information, which matches the transform of the corresponding entity in the dynamics subsystem. You can find out more about working with Havok collision detection, including creating shapes and collision filtering, in the Collision Detection chapter.

3.1.3.7 Continuous Simulation

This functional block solves all "Time of impact" events, in time order, that were generated from the collision detection part of the pipeline. Each time of impact event can create new time of impact events, so this is an iterative functional block that can be quite computationally expensive. This part of the simulation is not called when the world simulation type is discrete. You can find out more about continuous simulation in the Continuous Physics chapter

3.1.4 About the documentation

Each section of this manual provides you with detailed information about how to use one of the SDK components described above. Useful concepts are explained, and code samples (usually from the Havok demos) are given throughout, along with helpful tips and 'gotchas'.

This help file also includes a Reference Manual for the Havok Physics SDK. This manual provides you with full API documentation for the SDK, including class and package descriptions, detailed class member descriptions, and inheritance diagrams. It has full hyperlinking in both the diagrams and descriptions, an index, and a search tool, making it easy for you to find the information you need. You can use the manual as a reference when using this manual and the demos, and when working with the SDK in your own games.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.1.5 About the demos

The `demos` folder of your installation includes a number of useful demos that you can use to learn about Havok. These demos are laid out in a logical manner which reflects the layout of this documentation. Source code is given for each demo, allowing you to quickly find out how to perform important tasks, and providing code that you can cut and paste into your own applications. Helpful comments in the code describe each demo and the tasks performed in it. Together with the documentation, we hope the demos will help you to familiarize yourself with the SDK and get started with Havok Physics in your own games.

Figure 3.5: Various screenshots of samples from our demos (shipped with source code)

3.1.5.1 The demo framework

All of our demos use the same framework, which abstracts the differences between the various hardware platforms supported by Havok and also separates the display and simulation aspects of each demo. This abstraction leaves the physics source in one place so that it is easy to read and understand without being confused with controller handlers and specific display information. The framework includes a base example demo class - hkDefaultDemo - and a GUI application that you can use to navigate between and play the demos.

You can find out how to build and interact with the demos for your chosen platform in its separate platform-specific manual.

3.1.5.2 Demo types

The demos are divided into two broad categories - *api* and *examples*. Each *api* demo illustrates a very specific aspect of working with Havok physics, such as setting up a single constraint between two rigid bodies. These will help you to gain familiarity with the feature set of the SDK, the concepts to which these features relate, and the key classes associated with individual features. In contrast, *examples* demos combine two or more features and show how they are used together, for example in a character or vehicle game. We have made these demos as simple as possible so as to isolate key concepts and show how they relate to one another.

3.1.5.3 Running the visual debugger with the demo framework

To use the visual debugger run the demo framework with a "-d" command line parameter.

More information on using the visual debugger (and on using it with the various consoles) can be found in the Visual Debugger section of this help file.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.2 Havok Dynamics

3.2.1 Introduction

The Havok Dynamics module, hkpDynamics, is the central module of the Havok system. It allows you to create a simulated physical world and populate it with simulated objects, step the simulation forward in time, and interact with the simulation.

This chapter will show how to:

- Create and use a Havok world
- Create and work with rigid bodies, including keyframed bodies
- Create and work with phantom objects

Figure 3.6: An example of a Havok world, populated with various rigid bodies.

It also introduces three other types of simulation elements - constraints, actions, and listeners - that are dealt with in more detail in their own chapters.

3.2.2 The Havok World

Every Havok simulation has one or more Havok worlds - instances of the class `hkpWorld`. The `hkpWorld` acts as a container for the physical objects in your simulation, and is also used to step the simulation forward in time.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The hkpWorld is partitioned into separate groups of objects called islands (hkpSimulationIsland) that can be simulated independently for performance reasons (these are set up automatically by the system).

3.2.2.1 Creating an hkpWorld

You create a Havok world using hkpWorldCinfo, which contains all the information needed to construct the hkpWorld object. You can specify any of the following details for your hkpWorld, or you can use the provided defaults:

- **Gravity**

  An hkpWorld's gravity is applied to all its simulated objects. The default value for this parameter reflects "real-world" gravity (-9.8 in the y direction, which is Havok's default up axis). Alternative gravity settings can be used for your game world (in space, you would define gravity to be 0 along all axes), and along alternative axis (your engine might use the Z axis as up). Gravity is specified as an hkVector4. The use of constant gravity is so common in games that it has been integrated into the main engine, rather than implementing it as an action. More complicated gravity models could be implemented as an Action.

- **Broadphase**

  The m_broadphaseWorldAabb.m_min and m_broadphaseWorldAabb.m_max allows you to configure the collision detection size of the broadphase for the hkpWorld. The default broadphase setting is a box with sides of length 1000.0 (the equivalent of a 1 kilometer cube if you are using the recommended scale, i.e. meters). The m_broadPhaseQuerySize allows you to set the initial size of the broadphase overlap local array. You can find out more about the broadphase and the effect of changing its size in the Collision Detection chapter.

- **Solver**

  The Havok simulation uses a solver to calculate the necessary changes to reduce error for all constraints in the simulation. hkpWorldCinfo has methods to allow you to set the solver types. Each solver type provides a different speed/accuracy balance for the simulation. The solver behavior depends on how many iterations the solver goes through in each step when calculating the changes, and the tau and damping values used by the solver. You can read more about the solver's role in the simulation in the Solver Introduction. As a rule, the higher the iterations, the better the accuracy. If you are feeling adventurous you can set the solver parameters directly, otherwise see hkpWorldCinfo for a complete list of solver types.

- **Collision tolerance**

  This allows you to specify a collision tolerance value for the hkpWorld - the default is 0.1 (or 10cm if you are using the recommended scale of meters). You can find out about this value and its effect in the Collision Detection chapter.

- **Deactivation**

  Most objects in a normal scene are stationary and thus they do not need to be updated the same as moving objects. A lot of the data can persist across simulation steps. The world can spot objects that should be deactivated by monitoring the relative movement they make as time progresses. It monitors the movement based on two frequencies to reduce the chance of deactivating objects that are oscillating the same as the sampling frequency. The m_enableDeactivation flag turns the monitoring on and off. To save some CPU we check each object not every frame but every 4th frame (high frequency checks) and as an extra backup deactivation check every 16th frame (low

frequency check). If all the objects in an independent group of objects are flagged for deactivation, this group will deactivate. Normally this group is one island, however for performance reasons the engine might merge independent groups into a 'sparse' island.

- **Linear Casting**

  Linear casting is the ability to sweep a convex shape through the world and find the points of collision. This can be very useful for accurate proximity checks for characters and movement planing. The casting is an iterative algorithm and so the max number of iterations is exposed to allow for performance<->accuracy tweaking along with the min dist tolerance value for the cast, m_iterativeLinearCastEarlyOutDistance. If you change the collision tolerance then this number should be change too, usually to a 10th of the collision tolerance.

- **Misc Collision Parameters**

  When using hkpCollide through the dynamics library (as opposed to by itself for just collision detection) you may need to tweak the collision tolerances. These are exposed in the `hkpWorldCinfo` along with the main collisionTolerance mention earlier. These extra parameters are explained in more detail in the Collision Detection chapter and also the in Reference Manual for the `hkpWorldCinfo`.

- **Continuous Simulation Parameters**

  Continuous simulation and collision detection use a group of internal safety and optimization parameters. They are mainly derived from a few parameter exposed in `hkpWorldCinfo`.

Parameter	Purpose, Values
m_simulationType	Selects type of simulation. Available values are SIMULATION_TYPE_DISCRETE, SIMULATION_TYPE_CONTINUOUS, SIMULATION_TYPE_MULTITHREADED
m_expectedMaxLinearVelocity	Assumed maximum linear velocity for all bodies. This does not set the linear velocity in the objects. If later on, a rigid body is significantly faster than this value, it might tunnel through other objects even if its qualityType is set to critical. It defaults to 200 meter/second
m_expectedMinPsiDeltaTime	Minimum expected physical simulation delta time. Defaults to 30 Hertz
m_contactPointGeneration	Allows contact points to be created at negative distance, which speeds up the simulation, but doesn't work well with small objects. See Collision Detection chapter for details. It defaults to CONTACT_POINT_REJECT_MANY. See the Reference Manual for details.

Table 3.1: Continuous Simulation Initialization Parameters

The following example shows how to create an `hkpWorld`. As you can see, you simply create the `hkpWorldCinfo`, assign the values you want to its members, and then pass it to the `hkpWorld` constructor.

```
hkpWorldCinfo info;
info.m_simulationType = hkpWorldCinfo::SIMULATION_TYPE_DISCRETE;
info.m_gravity.set( 0,-9.8f,0);
info.m_collisionTolerance = 0.1f;
info.setBroadPhaseWorldSize( 150.0f );
info.setupSolverInfo( hkpWorldCinfo::SOLVER_TYPE_4ITERS_MEDIUM );

m_world = new hkpWorld( info );
```

Once you have created the `hkpWorld`, you can use `hkpWorld` functions to get and set all the `hkpWorldCinfo` values.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Note:

When you are integrating Havok into your own games, rather than using our demo framework, you need to ensure that `hkBaseSystem::init(...)` is called to initialize the Havok memory system and other system services before creating and using your `hkpWorld`. This is called automatically within the demo framework by the relevant graphics system implementation. You can find out more about this function and its role in the Base System section of the Havok Base Library chapter.

3.2.2.2 Adding simulation elements

Simulation elements such as rigid bodies, phantoms, constraints, actions and listeners need to be added to the `hkpWorld` to take part in the simulation. `hkpWorld` provides a number of functions for adding and removing different types of simulation element - for instance, you add a constraint using `addConstraint()`. You will find out more about how to use these functions later in this chapter.

Note:

`hkpWorld::add( hkSerializable* )` is used internally by the serialization framework. You can also use it to add any serializable Havok object to the `hkpWorld`, but all it does is examine the type of serializable and call the corresponding add method.

3.2.2.3 Entities

The simulated physical objects that you add to your `hkpWorld` are known as *entities*. The base class for entities is `hkpEntity`, which provides a number of functions and data members common to all entities. There is currently only a single entity type provided with Havok, the `hkpRigidBody`.

3.2.3 Rigid Bodies

Any real-world object that doesn't change its shape - from a car chassis to a boulder hurtling down a mountainside - can be simulated in Havok as a *rigid body*.

Rigid bodies are central to Havok's dynamics system. If you use objects that are completely rigid, you can do rapid physical simulation in real time. Rigid body simulation is perfectly suited to making many real world objects move realistically within a game environment.

You model rigid bodies using the `hkpRigidBody` class, a subclass of `hkpEntity`. An `hkpRigidBody` has an `hkpRigidMotion` that stores all the information relating to how the rigid body moves, including its mass and velocity. The `hkpRigidBody` also has an `hkpCollidable` member that contains the information needed for the body to work with the collision detection system - a `hkpShape` and a `hkMotionState` pointer. This motion state pointer points to a structure holding the position and orientation of the rigid body, thus linking the collision detection system to the dynamics system.

Note:

The collision detection system is designed to work without the dynamics library, hence the apparent level of indirection (the `hkMotionState` for instance) that exists between the collidable and the rigid body that owns it.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.2.3.1 Creating a Rigid Body

As with the `hkpWorld`, you create an `hkpRigidBody` using a cinfo, the `hkpRigidBodyCinfo`. You can specify all the following details for your hkpRigidBodies, or use default values where they are provided.

- **hkpShape m_shape**

  An `hkpRigidBody`'s `hkpShape` defines the object's shape and size for collision detection purposes. An `hkpShape` can have a number of implementations such as simple shapes (a box or sphere), a compound shape made up of simpler hkShapes etc. You can find out more about the different shape types and how to create them in the Collision Detection chapter.

- **hkUint32 m_collisionFilterInfo**

  This value can be used by collision filters to identify the entity - for example, if a group collision filter is used, this value would specify the entity's collision group. By default, this is set to zero. You can find out more about how to use collision filtering in the Collision Detection chapter.

- **Properties**

  See `hkpProperty`

- **hkpMaterial::ResponseType m_collisionResponse**

  This value governs how the rigid body reacts to collisions. See `hkpMaterial` for a full list of possible values and resulting behaviors.

- **hkUint16 m_processContactCallbackDelay**

  This value specifies how often the entity should send process contact events to any collision listeners that you have added to the entity. The default is 65535, which has the effect that the callback is called once for the first collision and then once every 65535 frames, until the bodies move outside the collision tolerance. You can find out more about this value and its effect in the Collision Listeners section of the Listeners chapter.

- **hkVector4 m_position**

  The initial position of the `hkpRigidBody`, specified as an `hkVector4` in world space. By default, this is set to the origin.

- **hkQuaternion m_rotation**

  The initial rotation of the `hkpRigidBody`, specified as an `hkQuaternion`. By default, this is set to the unit quaternion (the identity rotation).

- **hkpMotion::MotionType m_motionType**

  Use this parameter to specify whether the `hkpRigidBody` is fixed, movable with an inertia type, or keyframed. There is *no* default rigid motion type (the default value is invalid and will assert in debug builds) - you need to specify this for every `hkpRigidBody`. See `hkpMotion::MotionType` for a full list of motion types and their implications. You can also find out more about keyframed hkpRigidBodies in the Keyframed objects section of this chapter.

  If you create an `hkpRigidBody` as fixed or keyframed you cannot make it dynamic later during the simulation (using `hkpRigidBody::setMotionType()`), but you can make a dynamic `hkpRigidBody` fixed or keyframed during simulation and switch back.

  MOTION_DYNAMIC is the other enumerated motion type and is used to switch back to the previous dynamic motion state (box or sphere inertia based, as per above) during simulation using the `setMotionType()` function that is described later.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- **hkMatrix3 m_inertiaTensor**

  The inertia tensor of the `hkpRigidBody`, specified as an `hkMatrix3`. An inertia tensor describes how difficult it is to rotate an object around any given axis. In practice, it's reasonable to assume that most non-fixed objects in Havok will use a box inertia tensor. The inertia tensor used is determined by the rigid motion type described below. You can use the `hkpInertiaTensorComputer` class, provided in the hkutilities module, to calculate an appropriate value for this. The algorithm uses other information about the body - such as its size and mass - to create a realistic inertia tensor, as in the example below.

```
hkpRigidBodyCinfo info;
hkpMassProperties massProperties;

hkVector4 boxSize ( 1.0f, 2.0f, 3.0f );
hkReal boxMass = 10.0f;

hkpInertiaTensorComputer::computeBoxVolumeMassProperties(boxSize, boxMass, massProperties);
info.m_inertiaTensor = massProperties.m_inertiaTensor;
```

  Note that the method shown is for a box. Different methods are provided to calculate the inertia tensor for different types of object. A convenient method is also the hkpInertiaTensorComputer::setShapeVolumeMassProperties(shape, mass, rigidbodyInfoOut)

  As with the body's rigid motion type, there is no default inertia tensor value provided - you need to specify this for every *non-fixed* `hkpRigidBody`.

  Note that depending on the motion type, this inertia matrix gets reduced to:

  - hkpMotion::MOTION_SPHERE_INERTIA: The highest value of the inertia diagonal is used, and all outer diagonals are set to 0. This simplification is valid for quite a few objects like spheres or equally sized boxes.
  - hkpMotion::MOTION_BOX_INERTIA: The diagonal is used and all outer diagonal elements are discarded. This simplification is valid for all symmetric objects. As Havok's solver does not support full inertia, make sure that all your objects are aligned to the local coordinate space.

- **hkReal m_mass**

  The mass of the `hkpRigidBody`. There is *no* default mass value (the default value is invalid and will assert in debug builds) - you need to specify this for every `hkpRigidBody`. Remember to specify this value in kilograms if the world scale being used is meters (which is recommended). A mass of zero implies that the `hkpRigidBody` is fixed and uses the fixed motion type.

- **hkReal m_linearDamping, m_angularDamping**

  These specify the initial damping used for linear and angular movement, respectively. The default linear damping value is zero while the default angular damping value is 0.05. Linear damping reduces the movement of the object over time, and angular damping performs the same function but with the rotation of the object e.g. if you set the linear damping to 0.1f, then every second about 10% of the objects linear velocity will be removed. (Or to be precise: newVelocity = oldVelocity * exp(-damping*deltaTime))

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- **hkReal m_maxLinearVelocity, m_maxAngularVelocity**

  Set hard-limits for the hkpRigidBody's linear and angular velocities. The default linear velocity limit is set to 200 m/s. If you need a higher value for fast projectiles consider setting hkpWorldCinfo::m_expectedMaxLinearVelocity as well. Angular velocity is set to a very high value initially. Note however, that there is another clipping mechanism in place which limits the angular velocity of bodies to ∼170 degrees per simulation frame. Those values are actually hard limits. A rigid body will never ever exceed this value. Note: This max linear velocity is a hard limit built directly into the integrator. This could lead to strange effects if a body reaches is maximum velocity for a longer period of time: Say you limit the max velocity of a box to 10 meters/second. If a fast car hits this box, the car won't be able to push the box faster than this 10 meters/second. However, the Havok solver does not know about this limit. This means the solver will not work as expected and the box will quickly sink into the car. If continuous physics is enabled between the car and the box, extra CPU resources will be invested in TOI resolving (see chapter on Continuous Physics), however the integrator will simply undo the work of the TOI solver. So this parameter should only be used as a safety net to stop the physics from getting crazy, if you want to limit the velocity for gameplay reasons (say a box slowly falling down due to an attached parachute) you should use either damping or write your own action.

- **hkReal m_friction**

  You use this value to specify an initial friction value for the `hkpRigidBody`. A body's friction value indicates how smooth its surface is and hence how easily it will slide along other bodies. General friction values range between 0 and 1, but can be higher (a maximum of 255) The default value is 0.5. You can see the effects of changing friction values in the FrictionApi demo.

- **hkReal m_restitution**

  You use this to specify an initial restitution value for the `hkpRigidBody`. This indicates how "bouncy" the object is - in other words, how much energy it has after colliding with something. A value of 1 means that the object gets all its energy back after a collision, a value of 0 means that it will stop moving completely. The default value is 0.4. You can see the effects of changing restitution values in the RestitutionApi demo. The implementation of the restitution is only a rough approximation, therefore you might want to experiment with different values in your game to get the desired effect. Note that there is a hard limit of 1.99 for this parameter.

- **hkVector4 m_centerOfMass**

  The center of mass of the `hkpRigidBody`, in the body's local space. By default, this is set to the local space origin. For boxes and spheres, this is the center of the shape, for other shape types this depends on how you set up the shape - see the Collision Detection chapter for more details. Note that changing an `hkpRigidBody`'s center of mass does *not* change the position of the `hkpRigidBody`.

- **hkpCollidableQualityType m_qualityType**

  The quality type of the hkpRigidBody determines types of its interaction with other objects. The type is stored in `hkpRigidBody`'s collidable. Havok has nine predefined categories of objects: `HK_COLLIDABLE_QUALITY_FIXED`, `HK_COLLIDABLE_QUALITY_KEYFRAMED`, `HK_COLLIDABLE_QUALITY_DEBRIS`, `HK_COLLIDABLE_QUALITY_MOVING`, `HK_COLLIDABLE_QUALITY_CRITICAL`, `HK_COLLIDABLE_QUALITY_BULLET`, `HK_COLLIDABLE_QUALITY_USER`, `HK_COLLIDABLE_QUALITY_CHARACTER` `HK_COLLIDABLE_QUALITY_KEYFRAMED_REPORTING`. Interaction quality types are determined

by hkpCollisionDispatcher on per-object-pair basis. For more information refer to the Continuous Physics chapter.

- **hkReal m_allowedPenetrationDepth**

  The maximum allowed penetration for this object. This defaults to .05f This is a hint to the engine to see how much CPU the engine should invest to keep this object from penetrating. A good choice is 5% - 20% of the smallest diameter of the object.

- **hkpRigidBodyCinfo::SolverDeactivation m_solverDeactivation**

  Allows you to enable an extra single object deactivation algorithm. That means the engine will try to zero the velocity of single objects if those objects get very slow. This does not save CPU resources, though it can dramatically reduce slow movements in big stacks of objects.

For more information on the above properties, see the Physics Primer documentation.

Now let's look at a simple example. The following snippet shows the creation of a convex `hkpRigidBody` with a mass of 1.0 and an appropriate inertia tensor for its shape and size. The default values are used for the remaining `hkpRigidBodyCinfo` details.

```
int numVertices = 4;

// 16 = 4 (size of "each float group", 3 for x,y,z, 1 for padding) * 4 (size of float)
int stride = sizeof(float) * 4;

float vertices[] = { // 4 vertices plus padding
    -2.0f, 2.0f, 1.0f, 0.0f, // v0
     1.0f, 3.0f, 0.0f, 0.0f, // v1
     0.0f, 1.0f, 3.0f, 0.0f, // v2
     1.0f, 0.0f, 0.0f, 0.0f  // v3
};

hkStridedVertices stridedVerts;
{
    stridedVerts.m_numVertices = numVertices;
    stridedVerts.m_striding = sizeof(hkVector4);
    stridedVerts.m_vertices = &(vertices[0](0));
}

// We do not plan to use raycast on this object, so we do not have to pass in planeequations
hkArray<hkVector4> dummyPlaneEquations;

hkpConvexVerticesShape* pShape = new hkpConvexVerticesShape(stridedVerts, dummyPlaneEquations);

hkpRigidBodyCinfo rigidBodyInfo;

rigidBodyInfo.m_shape = pShape;

// Compute the shapes inertia tensor

hkReal mass = 10.0f;
hkpInertiaTensorComputer::setShapeVolumeMassProperties(pShape, mass, rigidBodyInfo);

rigidBodyInfo.m_motionType = hkpMotion::MOTION_BOX_INERTIA;
rigidBodyInfo.m_qualityType = HK_COLLIDABLE_QUALITY_MOVING;

hkpRigidBody* pRigidBody = new hkpRigidBody(rigidBodyInfo);
```

After you create an hkpRigidBody, you can use hkpWorldObject, hkpEntity and hkpRigidBody functions to get or reset all the values described above, including the object's position and rotation.

3.2.3.2 Adding a Rigid Body to the World

Use hkpWorld::addEntity() to add hkpRigidBodies to the simulation, as in the snippet below. This adds the box-shaped hkpRigidBody created in the previous example to the hkpWorld.

```
m_world->addEntity(pRigidBody, [ HK_ENTITY_ACTIVATION_DO_ACTIVATE] );
pRigidBody->removeReference();
pShape->removeReference();
```

The second (optional) parameter allows to add the hkpRigidBody in an active or inactive state. The default HK_ENTITY_ACTIVATION_DO_ACTIVATE value always activates the body. The HK_ENTITY_ACTIVATION_DO_NOT_ACTIVATE value adds the body in the inactive state. However, the body may be instantly activated if it overlaps with another active body.

Note that in this example removeReference() is called on the hkpRigidBody and its hkpShape after adding the hkpRigidBody to the hkpWorld. This is because hkpEntities and hkpShapes inherit from hkReferencedObject. hkReferencedObjects have a reference counter that facilitates Havok's object management system. By adding an hkpRigidBody to the hkpWorld, the hkpWorld keeps a pointer to the hkpRigidBody and increases its reference counter. If we do not need to keep a pointer to the hkpRigidBody, we call removeReference() to decrement the counter and give full ownership to hkpWorld. When there are no longer any active references to an object, Havok deletes it. In our example the final removeReference call will be made by the hkpWorld destructor. You do not need to worry about deleting hkReferencedObjects yourself and are generally recommended never to call delete explicitly on a hkReferencedObject. You can find out more about reference counting and how it works in the Havok Base Library chapter.

There is some CPU overhead associated with adding bodies to the hkpWorld. If you need to add many bodies at the same time, then you can add them as a batch by using the more efficient addEntityBatch() function.

3.2.3.3 Removing a Rigid Body from the World

You can remove an hkpRigidBody from the simulation by calling hkpWorld::removeEntity(). Note that if you are removing an hkpRigidBody from an inactive hkpSimulationIsland containing other objects, you may want to manually reactivate the island e.g. if you remove a box from the bottom of a stack in an inactive island, the rest of the stack will remain floating until you manually activate the island. This can be done by activating the body you want to remove before you remove it.

3.2.3.4 Changing the motion state

hkpRigidBody provides a number of functions that you can use to apply physical impulses, forces, and torques to a rigid body, as well as changing its velocity.

Changing the motion state allows you to turn a simulated hkpRigidBody into a keyframed one and vice

versa.

A typical scenario would be: You have a piece of static geometry, which becomes dynamic as soon as a bomb explodes. However, if you have an object, which you want to turn into a dynamic object as soon as another object hits it, it is better to create the object as dynamic and deactivate it. As soon as another object hits this object, it is re-actived.

When changing the motion state you must pass activation information (like in `addEntity()` function). The parameter doesn't matter when changing into fixed state.

Static objects do not collide with other static objects, therefore if you change the state of the object to or from static, the Havok engine has to requery the collision detection engine and you get a significant CPU hit. To avoid the CPU hit you may suppress the collision detection query passing the `HK_UPDATE_FILTER_ON_WORLD_DO_NOT_QUERY_FOR_NEW_PAIRS` parameter to the function. Use this feature with caution

Warning:
Use the *checkBroadPhase* parameter with caution as it disables creation of new collision agents. Therefore, it is safe to use this parameter when changing motion type to fixed if we expect the body to keep all its agents except the ones linking to other fixed bodies. It is not safe to use this parameter changing the motion type back to dynamic, when the body needs to collide immediately with fixed bodies in proximity.

Having a dynamic object requires having an inertia tensor. As static or keyframed objects do not have this information, you cannot (in theory) convert static or keyframed objects to dynamic ones. In practice we actually can use a trick: You create the object as a dynamic one first, set it's state to be static or keyframed before you add the object to the world. In this case the object actually stores it's initial inertia tensor which it uses when you convert the object back to dynamic.

Changing an object's motion state is particularly useful when applying actions. You can find out more about this in the Actions chapter.

Warning:
Trying to change the motion state of a deactivated object can have unpredictable results. Read more about this in the Deactivation issues section.

Velocity

You can use the `setLinearVelocity()` and `setAngularVelocity()` functions to specify new linear and angular velocities for your hkpRigidBodies during the simulation.

The following code from the MarbleAction (used in the Pyramid examples) shows the marble's "brake". The `setLinearVelocity()` and `setAngularVelocity()` functions are used to set the marble's velocity to zero when the brake button is pressed.

```
if (m_brakePressed)
{
    hkVector4 zero;
    zero.setZero4();
    m_body->setLinearVelocity(zero);
    m_body->setAngularVelocity(zero);
    setBrakePressed(false);
}
```

Forces and torques

Applying a force to an `hkpRigidBody` over time has effect on the velocity of an object (accelerating or decelerating it). Applying a torque has a similar affect on the angular velocity of the object.

You can apply a force to an `hkpRigidBody`'s center of mass or to a specified point using its `applyForce()` functions. Similarly, you can apply a torque to a body using `applyTorque()`.

Any forces and torques that you specify for an `hkpRigidBody` during a simulation step are applied immediately as impulses. Step delta time is needed to calculate the value of impulses, therefore you must also pass the *hkStepInfo* parameter to the function..

You specify a force as an `hkVector4`, where the values of the x, y, and z elements indicate the strength of the force in the corresponding axis direction, in world space. For instance, if you want to accelerate the body in an upward direction, you specify a value in the y element of the `hkVector4`, as in the example below. As with the linear impulse, the change in the body's linear velocity after the simulation step is inversely proportional to the body's mass.

As a force causes a change in the velocity of a body over time, the change in the object's velocity is dependent on the simulation delta time value. For example, if you apply an upward force to an object every 0.1 seconds (or 10 times a second) and then apply the same upward force to the same object every 0.0166 seconds (or 60 times a second), you would end up with slightly different results within the simulation. This is an artifact of numerical integration.

See `hkpRigidBody`::applyTorque() for more details about how that function works.

In the following code from the Fountain example, upward forces are applied to all objects that fall inside the fountain:

```
for (int i = 0; i < m_phantom->getOverlappingCollidables().getSize(); i++ )
{
    hkVector4 force( 0,70,0 );
    hkpRigidBody* rb = = hkGetRigidBody(m_phantom->getOverlappingCollidables()[i]);
    if (rb)
        rb->applyForce( stepInfo, force );
}
```

The fountain area in this game is created using a phantom object. You will find out about how to use them in the Phantoms section later.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

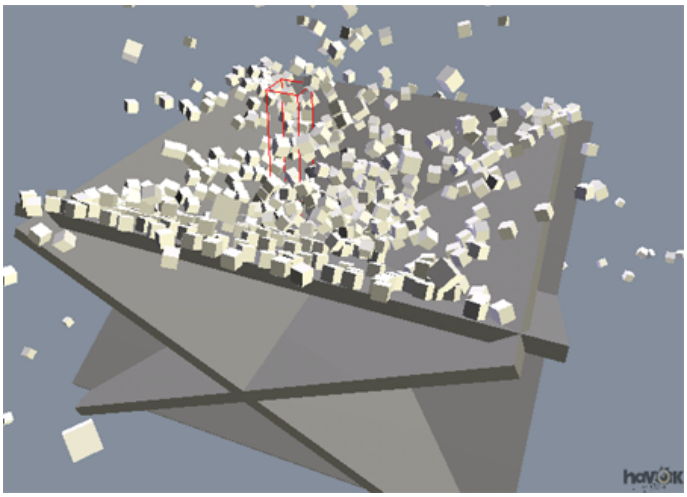


Figure 3.7: The Fountain example in action

Impulses

Applying an impulse has the immediate effect of changing the `hkpRigidBody`'s velocity - think of a car driving into a wall! An impulse delivers all of the specified force to an object instantly (think of it as a force which has been pre-multiplied by a time). If you had a box in a room and wanted it to jump, you would apply an upward impulse to it. If you wanted it to float, you would apply a force to it over time.

The `hkpRigidBody` class has several functions for applying impulses to hkpRigidBodies. You can use `applyLinearImpulse()` to apply an impulse at the object's center of mass, or `applyPointImpulse()` to apply a linear impulse at a specified point. The `applyAngularImpulse()` function lets you apply an instantaneous change in angular velocity (in world space) around the center of mass.

For a linear impulse you specify the impulse as an `hkVector4`, where the values of the x, y, and z elements indicate the strength of the impulse in the corresponding axis direction, in world space. So, for instance, if you want to change the body's velocity in an upward direction, you specify a value in the y element of the `hkVector4`, as in the example below. The change in linear velocity is inversely proportional to the body's mass (the difference in velocity is the impulse multiplied by the inverse mass).

You also specify an angular impulse as an `hkVector4`. However, in this case the normalized vector specifies the axis around which you want the body to move, and the magnitude of the vector specifies the strength of the impulse. For an angular impulse, the difference in angular velocity is inversely proportional to the body's inertia (or to be precise, proportional to the inverse of the inertia tensor matrix).

If you apply an impulse to a point other than the center of mass, it changes the body's linear velocity as described for a linear impulse at the center of mass, but also affects its angular velocity. The change in angular velocity depends on how far the point is from the center of mass as well as the body's inertia.

The following example, again from the MarbleAction, shows the use of an upward impulse to make the marble jump when the jump button is pressed. As you can see, the body's own mass is used when specifying the impulse, meaning that this method will have similar results with marbles of different masses.

```
if (m_jumpPressed && ((m_lasttimeCalled - m_lastJump) > 1.0f))
{
    m_lastJump = m_lasttimeCalled;
    hkVector4 imp(0, m_body->getMass() * 6, 0);
    m_body->applyLinearImpulse(imp);
    setJumpPressed(false);
}
```

The MarbleAction also uses impulses to roll and turn the marble. You can see the complete MarbleAction and Pyramid example in the demos provided with your Havok installation - as well as exploring the code, try playing the game and knocking down the pyramid with the marble!

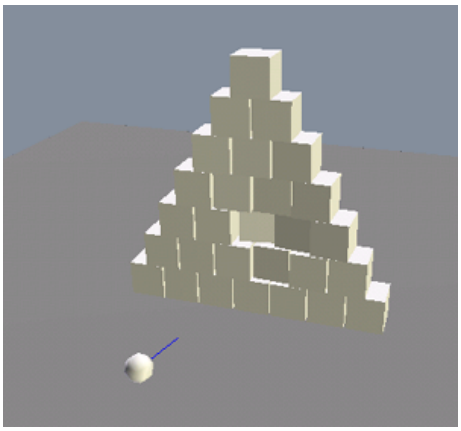


Figure 3.8: The Pyramid example in action

3.2.3.5 Rigid body collisions

During simulation, the collision detector will be called to generate contact points between hkpRigidBodies. Several contact points need to be generated between two hkpRigidBodies in order to resolve the collision correctly. Consider a box sitting on a flat surface. In this case we generate 4 contact points, one for each corner of the box that is touching the surface. This is called a "contact manifold". In general we do not create more than 4 points between two convex hkpRigidBodies. In certain conditions however, for example where an `hkpRigidBody` is colliding with a highly tessellated triangle mesh, many more contact points can be generated. It is not desirable to have this situation, because a large number of contact points adversely affects performance. Havok has a limit of 250 contact points which may be created between two hkpRigidBodies. If this limit is reached a warning is generated, and any more collision points generated will be automatically rejected. This warning usually indicates something is wrong - your mesh is too tessellated for example. However, if this cannot be remedied, the system will still work properly, as the contact points that are generated will compensate for the ones that are rejected.

3.2.3.6 Deactivation

Any hkpRigidBodies that have come to rest in the simulation are eligible for deactivation. A deactivated body doesn't have to be simulated at all, thus saving valuable CPU time. Each body has a deactivator that tells the system when it is safe to deactivate the entity, based on its state.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The only deactivator type is `hkpRigidBodyDeactivator::DEACTIVATOR_SPATIAL` (apart from the hkpRigidBodyDeactivator::DEACTIVATOR_NEVER). This can be accessed by calling `hkpRigidBody::getDeactivatorType()`. The initial deactivator settings are controlled by `hkpWorldCinfo`::m_deactivationReferenceDistance. The basic idea is that we deactivate an object if its position stays within a sphere (radius=m_deactivationReferenceDistance) for about 20 frames (high frequency deactivation). Or within a sphere (radius = 4*m_deactivationReferenceDistance) for about 80 frames (low frequency deactivation). If the object breaches this deactivation sphere, the center of the sphere is set to the current position of the object and the deactivation counters is reset to 0. Also the same rules apply to angular movement. To get detailed access to the deactivation parameters, you can change the values in `hkpSolverInfo` (`hkpWorld::getSolverInfo()`). If all the bodies in an `hkpSimulationIsland` are eligible for deactivation, the entire island is deactivated. Once a single `hkpRigidBody` is re-activated due to a collision or a call to `forceActivate()`, all hkpRigidBodies in that island are reactivated.

A more subtle deactivation can also occur when an object and its island aren't actually deactivated (so they still take simulation time) but the solver doesn't move them. This is known as "Solver Deactivation" and is especially useful when simulating stacked objects. See m_solverDeactivation for more details about Solver Deactivation.

Manual deactivation and reactivation

Normally hkpRigidBodies are deactivated when their entire island comes to rest, as described above. You can force a body to deactivate using its `deactivate()` method. Due to the nature of `hkpSimulationIsland`s, calling this method on an object will also deactivate all of the other objects in its `hkpSimulationIsland` (methods exist to find out what other objects are in the `hkpSimulationIsland`).

Deactivated hkpRigidBodies are automatically reactivated when the island they are in is activated. This most often happens with island-merging when an active object's aabb overlaps with the aabb of an object in the deactivated island causing the island to activate. To manually activate a hkpRigidBody, you call its `activate()` method. This also reactivates the body's entire hkpSimulationIsland. Objects are automatically activated when any of the methods that change the `hkpRigidBody`'s position, rotation or velocities are called, and so it is not necessary to call `activate()` when calling one of these methods (e.g. `setLinearVelocity()`, `applyImpulse()`, `setPosition()`). To preset a `hkpRigidbody`'s position, rotation, or velocities without the implicit activation, access the body's `hkpRigidMotion` directly and use its corresponding `setLinearVelocity()`, `applyImpulse()`and other functions.

Deactivation issues

Note that actions are *not* applied to objects while they are deactivated. This is because the simulation island which contains the object is also deactivated, and hence will not evaluate any of its actions.

Some hkpRigidBody settings can give the appearance that the island deactivation was applied, such as high values for hkpRigidbodyCinfo::m_angularDamping or hkpRigidbodyCinfo::m_linearDamping. Check that these are set to reasonable values when objects deactivate too quickly.

Another issue is when an object is removed from an inactive simulation island; the island isn't automatically reactivated (see Removing a rigid Body).

Switching off deactivation

Deactivation can be set for a single `hkpRigidBody` or for the whole simulation. For instance, you can specify that you never want a particular `hkpRigidBody` to be deactivated by setting its deactivator to DEACTIVATOR_NEVER as in the example below. You might want to do this, for instance, if the

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

hkpRigidBody is the chassis of a Havok vehicle.

```
chassisRB->setDeactivator( hkpRigidBodyCinfo::DEACTIVATOR_NEVER );
```

However, bear in mind that if an hkpSimulationIsland contains an hkpRigidBody that is set to never deactivate, the hkpSimulationIsland will never deactivate itself, which can cause a performance hit.

If you want to ensure that all of the objects in the simulation remain active (e.g. for testing purposes), you can disable deactivation in the whole hkpWorld upon its construction:

```
worldCinfo.m_enableDeactivation = false;
```

Other parameters that modify how the whole simulation handles deactivation are: m_highFrequencyDeactivationPeriod and m_lowFrequencyDeactivationPeriod.

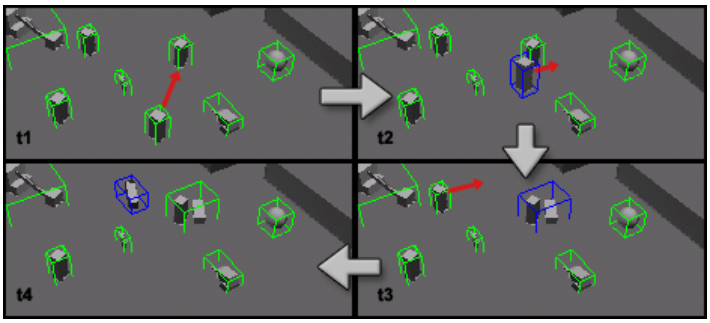


Figure 3.9: Green boxes are inactive Aabbs, blue are active (screens taken from the visual debugger)

In the figure above, a box is moved into the same hkpSimulationIsland as another box between time t1 and t3. At time t4 this hkpSimulationIsland is inactive, and another box in the area is active within its own hkpSimulationIsland.

3.2.3.7 Keyframed objects

As you know, a keyframed object is treated by the simulation as if it had infinite mass - however, unlike a fixed object, you can move a keyframed object around. A keyframed body with a velocity of zero behaves exactly like a fixed hkpRigidBody. However, if you set a velocity for it, it will move at that velocity (or you can set its position, though this is more risky as you might accidentally cause a major interpenetration between it and another hkpRigidBody). Objects will bounce off and roll over a keyframed object but will not affect its motion, and impulses and forces will not change its velocity.

You can even turn box inertia and sphere inertia objects into keyframed (or fixed) objects and back during the simulation, allowing you to gain or relinquish full control over the motion of objects on the fly within the physics simulation. This could be used for implementing a character picking up and dropping objects within a game for instance.

By default, collisions between keyframed objects and the static geometry are disabled. If you are

really interested in the contact point data, enabling it may generate useless contact points, which only cost CPU and memory. Those collisions are controlled by objects' `m_qualityType` setting and by the `hkpCollisionFilter`. Therefore, if you have no good reason to enabled such collisions, keep the `m_qualityType` setting synchronized with current motion type of the body or setup your `hkpCollisionFilter` in a way, so it rejects those collisions.

Changing the motion type of an object

To turn a movable object into a keyframed object, you call its `setMotionType(hkpMotion::MotionType)` method, setting the value to hkpMotion::MOTION_KEYFRAMED. To return it to its movable state, you use `setMotionType(hkpMotion::MOTION_DYNAMIC)`.

While you can turn a dynamics object into a keyframed or fixed object and back, you *cannot* turn a body that was originally keyframed or fixed into a movable object. This is because Havok maintains physical state information - such as mass - for objects that you have set fixed, and can restore it appropriately once you "unpin" them. Keyframed and fixed objects have never had this information (it's assumed that they will be keyframed for the duration of their creation, therefore no data required for the dynamics engine is calculated and stored for these objects), and so they cannot be "unpinned" into the physical simulation.

Keyframed vs. Fixed?

You should avoid using keyframed objects rather than fixed objects if you don't ever want to move them. This is because fixed objects are more efficient as Havok can perform certain optimizations for fixed objects when they are added to the world, as it knows that they will always remain in the same place. Fixed objects are also faster to use as they do not affect simulation islands.

hkpKeyFrameUtility

`hkpKeyFrameUtility` contains utility functions to help calculate and apply velocities to keyframed bodies so that they will follow your chosen position/orientation and velocity.

This can be particularly useful when using a keyframed object with a character controller - the keyframed object's position and orientation could correspond to the game character's position and orientation, allowing the character to realistically interact with the physics world. You can find out more about using keyframed objects like this in the Character Control chapter.

Another example of using a keyframed object would be when creating moving platforms within a game.

applyHardKeyFrame()

This function changes the linear and angular velocities of a body so that it will move to a particular position and orientation in a specified time. You pass it the target position (as an hkVector4), the target rotation (as an hkQuaternion), the *inverse* delta time (as an hkReal) and the keyframed body.

applySoftKeyFrame()

This function allows you to specify more detailed acceleration guidelines for moving the keyframed body. As well as the keyframed body and the inverse delta time, you need to provide the following:

- **KeyFrameInfo**

  A KeyFrameInfo struct contains the current keyframe position, orientation, angular velocity, and

linear velocity. It also provides functions for calculating the angular and linear velocities if you only have the current and target positions and orientations. The position must be specified as the position of the center of mass of the object rather than the local origin. See hkpKeyFrameUtility for more information.

- **AccelerationInfo**

  This specifies the following:

  - **Linear position factor**

    This governs how quickly you want the body to reach the target position. You specify this as 1/time. So, for instance, if this value is 2, then `applySoftKeyFrame()` will set the body's velocity so that it will reach the position of the next keyframe in half a second. Valid values are between 0 and 1/deltaTime.

  - **Angular position factor**

    This governs how quickly you want the body to reach the target orientation. You specify this as 1/time. So, for instance, if this value is 2, then `applySoftKeyFrame()` will set the body's velocity so that it will reach the orientation of the next keyframe in half a second. Valid values are between 0 and 1/deltaTime.

  - **Linear velocity factor**

    This governs how quickly you want the body to reach the target linear velocity. You specify this as 1/time. So, for instance, if this value is 2, then `applySoftKeyFrame()` will set the body's velocity so that it will reach the linear velocity of the next keyframe in half a second. Valid values are between 0 and 1/deltaTime.

  - **Angular velocity factor**

    This governs how quickly you want the body to reach the target angular velocity. You specify this as 1/time. So, for instance, if this value is 2, then `applySoftKeyFrame()` will set the body's velocity so that it will reach the angular velocity of the next keyframe in half a second. Valid values are between 0 and 1/deltaTime.

  - **Max linear acceleration**

    If the linear velocity changes required by `applySoftKeyFrame()` exceed this acceleration, they are clipped against this value.

  - **Max angular acceleration**

    If the angular velocity changes required by `applySoftKeyFrame()` exceed this acceleration, they are clipped against this value.

  - **Max allowed distance**

    If the distance between object and keyframe gets bigger than the max allowed distance, the object is immediately "warped" to the correct keyframe position / orientation, and corresponding velocities.

3.2.4 Stepping the simulation forward

As mentioned previously, the `hkpWorld` is used to step the simulation forward in time. You do this by calling `hkpWorld::stepDeltaTime()`, passing it the desired delta time value, in seconds. At a fixed frame update of 60Hz, the delta time would be 0.0166f. Very small or large delta values are not advised, and values that vary a lot reduce the accuracy of the solver. The How It Works Introduction describes the tasks carried out by the physics engine in each step.

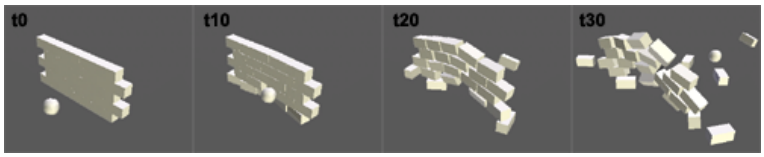


Figure 3.10: Snapshots of a Havok simulation being stepped forward.

3.2.4.1 Synchronous stepping

To simulate the world forward at 60 Hz as described above, you simply call:

```
m_world->stepDeltaTime(0.0166f);
```

After calling stepDeltaTime(), the positions and velocities are all updated based on the timestep and all the other simulation variables. At this point you can simply get the transform of each object and apply it to your display representation. If you wish to take results from the physics and apply it to your game this is also a good point to do so. The best point to apply impulses etc from your game to the physical world is just before stepDeltaTime().

3.2.4.2 "Half" stepping

The call hkpWorld::stepDeltaTime() does several things in order to advance the simulation. In single threaded simulations, i.e. hkpWorldCinfo::SIMULATION_TYPE_DISCRETE and hkpWorldCinfo::SIMULATION_TYPE_CONTINUOUS, it is possible to make finer grained calls to the world in order to advance the simulation. These functions are available in order to allow you to spread the cpu load of simulating the world over several graphical frames, or game loop steps. These functions are:

```
// Integrate the world forward in time - solves all constraints, and advances all transforms to the next
  timestep
   m_world->integrate(0.0166f);

   // Perform collision detection
   m_world->collide();

   // Perform continuous simulation
   m_world->advanceTime();
```

An example of "half" stepping the world using these functions is given in the demo Physics / UseCase / Fracture / City / FractureDemo. It performs integration and collision detection on alternate frames. It does so as follows:

```
void stepPhysics( physicsDeltaTime )
{
   hkTime timeForDisplay;
   if (m_halfStepCounter++ %2 == 0)
   {
      m_world->integrate( physicsDeltaTime );
      timeForDisplay = m_world->getCurrentTime() + physicsDeltaTime * .5f;
   }
   else
   {
      m_world->collide();
      m_world->advanceTime();
      timeForDisplay = m_world->getCurrentTime();
   }

   synchronizeTransforms( timeForDisplay );
}
```

In the above code, on alternate frames, we either call integrate() or collide() and advanceTime(). This allows us to split the physics processing time between rendering frames. i.e. we render at 60Hz, but we actually simulate at 30 Hz. Note you have to synchronize the transforms of the display objects using a time greater than the world current time after integrate(), because the world current time is not updated until advanceTime(). To synchronize a transform you would call rigidBody->approxTransformAt(timeForDisplay). Doing this gives you a smooth interpolation of transforms in the display frames. The following diagram shows how these three calls look in the context of the block diagram presented in the introduction section.

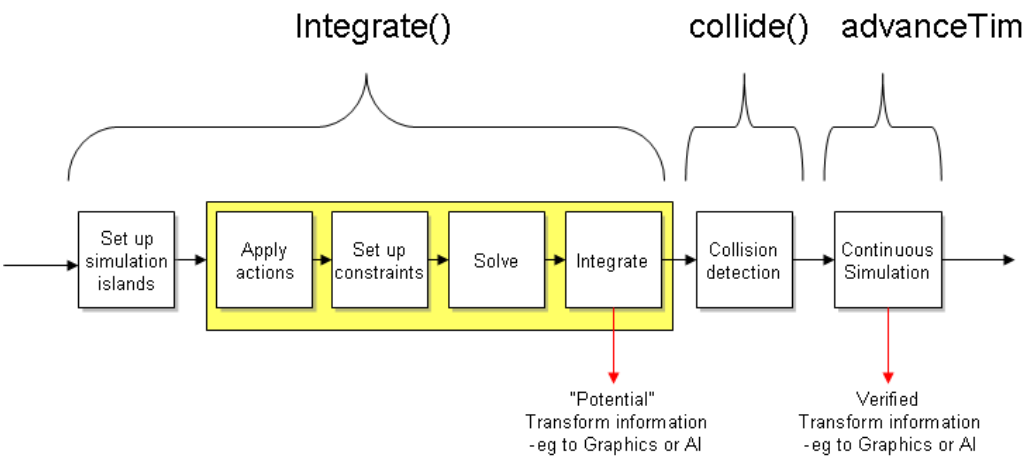


Figure 3.11: Integrate(), collide() and advanceTime()

The advanceTime() function performs continuous simulation. If the simulation is of type hkpWorldCinfo::SIMULATION_TYPE_DISCRETE, no continuous simulation is performed by advanceTime(), however it is still essential to call the function in order to advance the current time of the world.

3.2.4.3 Asynchronous stepping

Note that in the above case, we are stepping physics and graphics in parallel. This is our recommended approach because it keeps things simple, and is the most predictiable in terms of CPU load, and, particularly in multithreaded simulation, the most efficient way to simulate the objects.

However, there could be reasons (e.g. very varying framerate, slow framerate or running the physics at a much lower frequency) to decouple physics and framerate and step the physics asynchronuously. To do this, several more functions are provided in hkpWorld. These are hkpWorld::setFrameTimeMarker(), hkpWorld::isSimulationAtMarker() and hkpWorld::isSimulationAtPsi(). Please note that using asynychronous stepping throughout your game engine is challenging and only recommended for advanced users.

The code that you use to step the world asynchronously is:

```
world->setFrameTimeMarker( frameDeltaTime );

    world->advanceTime();
    while ( !world->isSimulationAtMarker() )
    {
        HK_ASSERT( 0x11179564, world->isSimulationAtPsi() );

        {
            // Interact from game to physics
        }

        world->stepDeltaTime( physicsDeltaTime );

        if (world->isSimulationAtPsi() )
        {
            // Interact with physics: physics data to game data
        }
    }
```

This code is best explained in conjunction with a diagram.

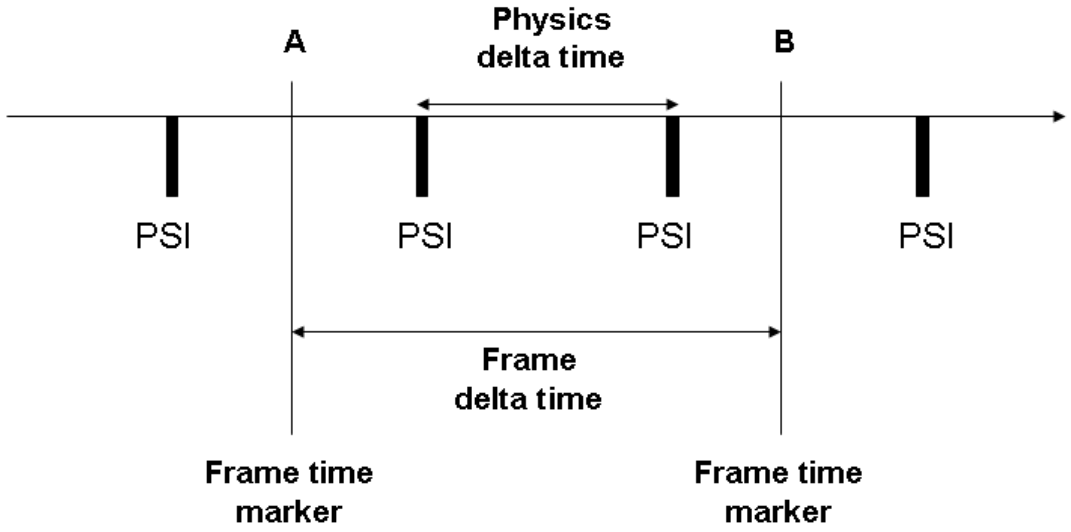


Figure 3.12: Integrate(), collide() and advanceTime()

The above diagram shows a picture of simulation time as the physics progresses in discrete timesteps, called PSIs. In synchronous simulation, i.e. when you are simply calling stepDeltaTime(physicsDeltaTime), the simulation simply moves from one PSI to the next. However, it is possible to "simulate" for any length of time by setting "frame time markers". These are set with the hkpWorld::setFrameTimeMarker(frameDeltaTime) call and are shown in the diagram above. Assuming that the world current time is at time "A", if you call setFrameTimeMarker(frameDeltaTime) you will set a marker at time "B". Here we are assuming that we have just completed an asynchronous step. The world current time (obtained by hkpWorld::getCurrentTime()) will be different to the world current PSI time (obtained by hkpWorld::getCurrentPsiTime()). The PSI time is always the same or greater than the current time. This initial situation is illustrated below:

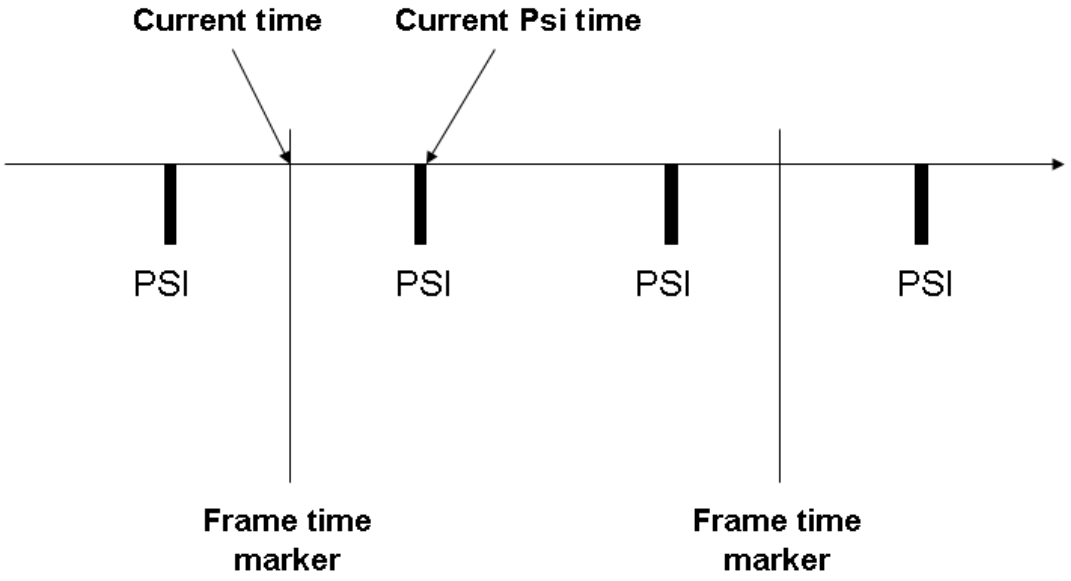


Figure 3.13: Integrate(), collide() and advanceTime()

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

After setting the new marker, you must then call advanceTime(). This will advance the current time to either the current PSI time or the frame time marker, whichever is earlier. If you are running a continuous simulation, this will solve all TOIs between the current time and the time it advances to. In our example, after advanceTime() is called, the time bar will look like this:

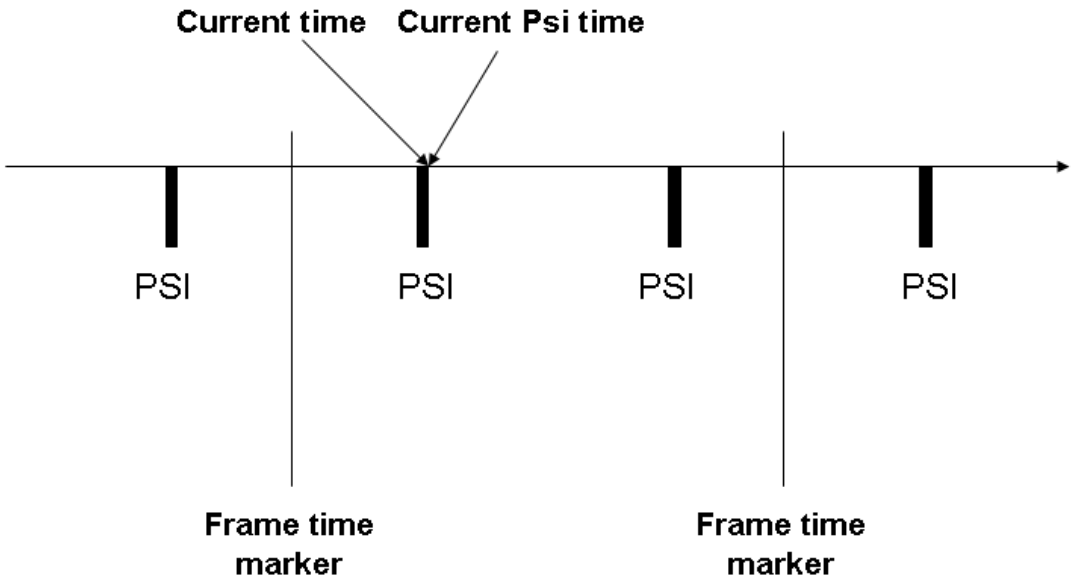


Figure 3.14: Integrate(), collide() and advanceTime()

You must then check if the simulation has reached the frame time marker, using hkpWorld::isSimulationAtMarker(). If it has not then you must step a PSI step, with stepDeltaTime(). Before calling stepDeltaTime(), you should write code to interact from you game to the physics. The call to stepDeltaTime(), will advance the simulation, and the current time, to either the current PSI time or the frame time marker, whichever is earlier. Actually, stepDeltaTime() simply calls integrate(), collide(), and advanceTime() in order. After this call you need to check if the simulation is at a PSI (rather than "mid-step" at a frame marker) and if it is, execute your code to interact from your physics to your game. After stepDeltaTime() the time bar will then look as follows:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

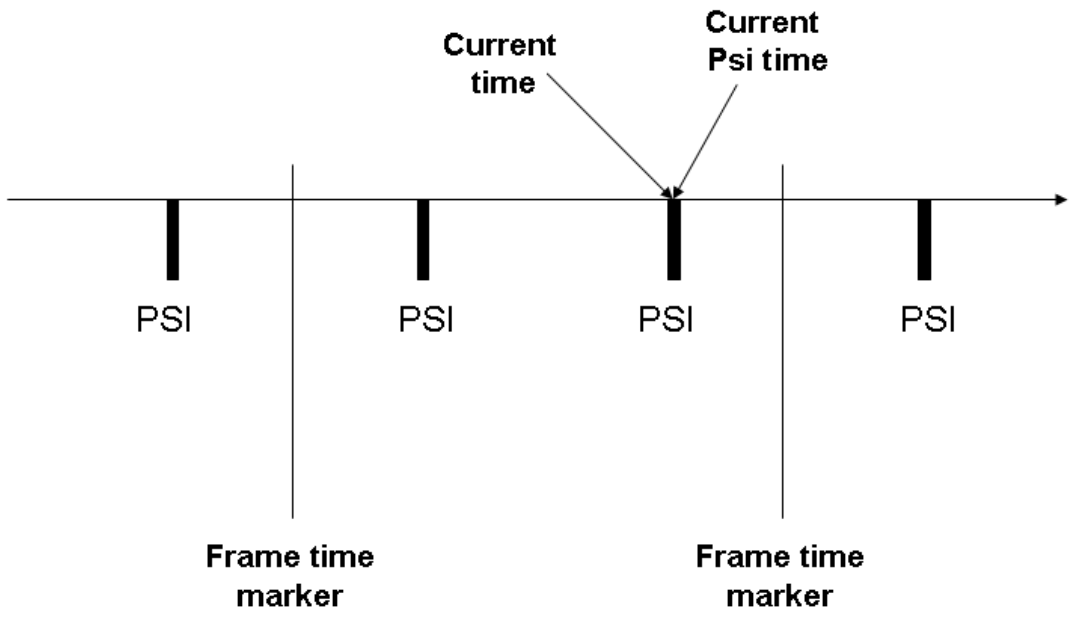


Figure 3.15: Integrate(), collide() and advanceTime()

At this point isSimulationAtMarker() still returns false, so another PSI step must be performed by calling stepDeltaTime(). After this call it can be seen (see the diagram below) that the current PSI time advances to the next PSI marker (this happens in the integrate() and collide() parts of stepDeltaTime()) and the current time only advances as far as the frame time marker (this happens in the advanceTime() part of stepDeltaTime()). At this point the simulation has been advanced to the desired frame time so the loop terminates.

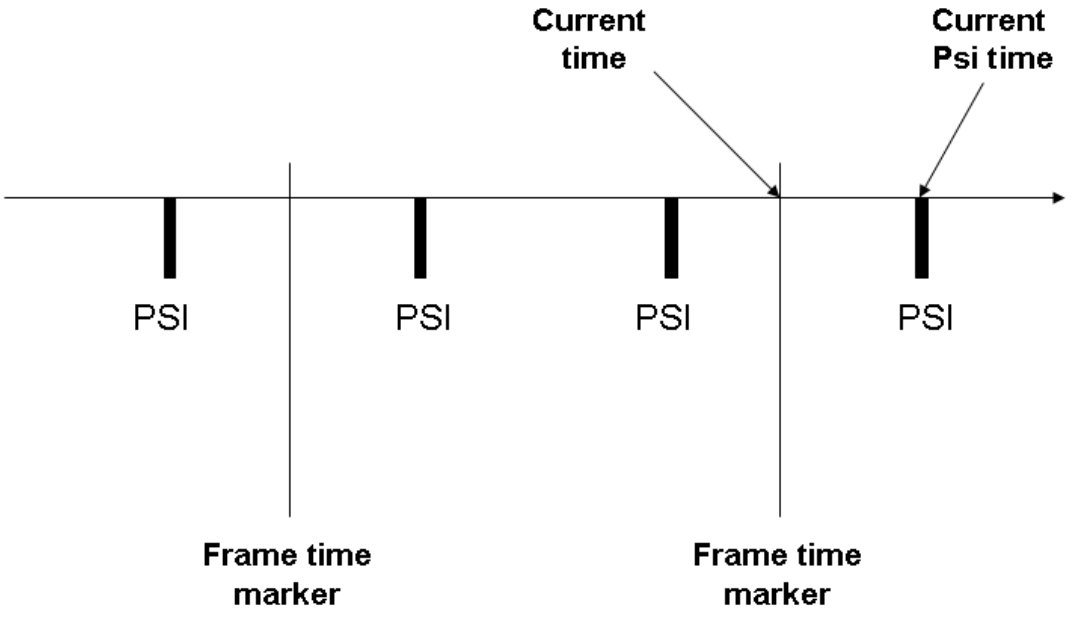


Figure 3.16: Integrate(), collide() and advanceTime()

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

It is important that you do game / physics interaction when the world current time is at a PSI rather than when it is between PSIs (as it is at the end of the loop). It is fine to render the objects at this position, because the system will interpolate positions and orientations using the hkpRigidBody::approxCurrentTransform, or hkpRigidBody::approxTransformAt functions. However, applying impulses and forces to these objects at this time is problematic. As well as having two positions and orientations at unique times, each object has a single transform, representing its transform at its "current simulated time". For some objects this transform will be at the current PSI time, while some will have a transform at the current time (depending on whether they have been involved in continuous simulation. Also to accurately apply a force to the object, you need to apply the force for the remainder of the timestep.

You can avoid these problems by interacting with the physics when the simulation is at a PSI. This is shown by the comments in the code block above. This code block is actually the implementation of the utility function hkAsynchronousTimestepper::stepAsynchronously(hkpWorld* world, hkReal frameDeltaTime, hkReal physicsDeltaTime). While you can use this function, when it returns it will typically give you a current time out of sync with the current PSI time, so it is not great to interact with the physics at this point. Rather, we recommend you take the implementation and integrate it into you main game loop, and add physics interaction code at the points specified.

3.2.4.4 Multithreaded stepping

The system also supports stepping with multiple threads. The functions hkpWorld::stepBeginSt(), hkpWorld::stepProcessMt and hkpWorld::stepEndSt() are used for this. The St and Mt tags on these function names refer single threaded and multi-threaded respectively. For a full discussion on multithreaded stepping please refer to the Multithreading Chapter.

3.2.5 Phantoms

An `hkpPhantom` has no physical presence in the simulation and is used to maintain a list of entities (and other phantoms) that overlap with it. The most common application of an `hkpPhantom` is to detect entities entering and leaving certain volumes of the hkpWorld, perhaps causing a scripted cinematic sequence to be played or a nearby AI character to be alerted to your presence.

This job can be done by simply calling `hkpWorld::getPenetrations( .. myObject ..)`. This is all you need. However, this approach can consume unnecessary CPU resources. To speed things up, Havok introduced the concept of Phantoms. The different types of phantoms reflect different types of caching mechanisms: `hkpAabbPhantom` and `hkpSimpleShapePhantom` cache at the Broadphase stage only, while `hkpCachingShapePhantom` caches at the broadphase and narrowphase stages.

All hkpPhantoms update their list of (Aabb) overlaps via two callbacks from the hkBroadphase, `addOverlappingCollidable()` and `removeOverlappingCollidable()`. Whenever any object in the Broadphase moves (updates its Aabb), these callbacks may be fired, and it is up to the Phantom to respond appropriately, for example maintaining or managing its own list of currently overlapping hkCollidables. Note as the overlap checks corresponding to these two callbacks take place at the *Broadphase* level, only overlapping Aabbs will be reported and as a result *Narrowphase* hkpShape overlaps are not considered.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Warning:

The `addOverlappingCollidable()` and `removeOverlappingCollidable()`callbacks are not completely symmetric with respect to collision filters.
The world collision filter is *always checked* before an `addOverlappingCollidable()`might be called, and if the corresponding hkCollidables do not have collisions enabled, no `addOverlappingCollidable()`is called.
However, *no collision filter is checked* before a `removeOverlappingCollidable()`so this callback may be fired when two overlapping hkCollidables move apart, *even if they already have collisions disabled.* Thus all hkpPhantoms must have special code in the `removeOverlappingCollidable()` method to deal with the case where they may be told about the removal of an overlap via `removeOverlappingCollidable()` with a body for which they never received a corresponding `addOverlappingCollidable()` callback. The three supplied implemetations of hkpPhantom: `hkpAabbPhantom`, `hkpSimpleShapePhantom` and `hkpCachingShapePhantom` all have this code, so if you inherit from them you will not need to add this extra check.

`hkpAabbPhantom` operates only at the Broadphase level. Its granularity is purely the axis aligned bounding box (Aabb) with which it is constructed and it only examines the Broadphase representation of any object it encounters when deciding whether or not an overlap has occurred. Due to this they may update their overlapping list very quickly and they are generally a good choice when accuracy is not overly important.

The second genre of `hkpPhantom`s are `hkpShapePhantom`s which are the finer-grain relatives of the `hkpAabbPhantom`. These phantoms may use any arbitrary shape to define their bounding volume and the user can subsequently choose to perform asynchonous queries to detect which objects' hkShapes are overlapping at the Narrowphase level. If accuracy outweighs pure speed then `hkpShapePhantom`s are preferred, but it is up to the user to decide when to perform these finer-grain Narrowphase collision queries.

3.2.5.1 hkpAabbPhantom

`hkpAabbPhantom` may be used in two ways: a) As a trigger zone within the hkpWorld, and b) for short raycasting. As `hkpAabbPhantom` is completely Broadphase based, it is ideally suited for triggering events which rarely depend on highly precise detection. For example, to know that the player has passed through a triangular door all that we really need know is that they have passed through the 'plane' of the door, and so a thin Aabb will do just as nicely as a more expensive triangular volume!

The second useful feature is short raycasting which benefits from the fact that the phantom already has a list of overlapping objects and so it may immediately proceed to perform a shape level raycast on each of them. Of course, you must make sure that the Aabb of the phantom is large enough to encompass the volume of space you are interested in! Using `hkpWorld::castRay()` for short raycasts on the other hand, requires that a list be built of overlapping candidates before proceeding to the shape level casting which will make it a little slower. For long raycasting, however, the world variant is preferred as a very large phantom would be prone to significant amounts of updating.

You can see an example of using a phantom object in the Phantom Object demo, shown below.

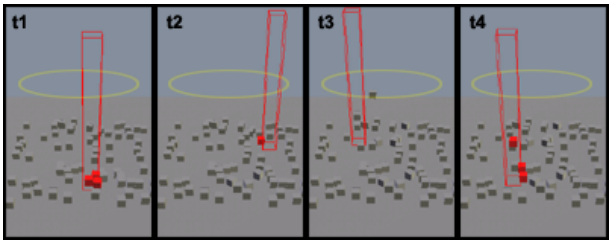


Figure 3.17: The phantom object in this example moves in a circle, applying an upward force to objects that overlap with it.

In the figure above, the phantom object is moved around by changing its Aabb using `m_aabbPhantom->setAabb( aabb )`, an asynchronous call which automatically updates the broadphase data as well as its internal list of entities which currently overlap it.

The code below illustrates an additional step in this demo which filters out any fixed objects, to ensure that only movable objects are contained within the phantoms list of collidable objects.

```
void hkMovingRBCollectorPhantom::addOverlappingCollidable( hkpCollidable* c )
{
    hkpRigidBody* rb = hkGetRigidBody(c);

    // Ignore other phantoms and fixed rigid bodies.
    if ( (rb != HK_NULL) && !rb->isFixed() )
    {
        hkpAabbPhantom::addOverlappingCollidable( c );
    }
}

void hkMovingRBCollectorPhantom::removeOverlappingCollidable( hkpCollidable* c )
{
    hkpRigidBody* rb = hkGetRigidBody(c);

    // hkpAabbPhantom::removeOverlappingCollidable() already protects its m_overlappingCollidables member,
    // regardless of whether collidables are present in it or not. However, this check needs to be
    // symmetric with the one above (ie. needs to check for non-fixed bodies), if we want symmetry
    // in callbacks fired within hkpAabbPhantom::removeOverlappingCollidable() and
    // hkpAabbPhantom::addOverlappingCollidable().
    hkpAabbPhantom::removeOverlappingCollidable( c );
}
```

The phantom object is then queried to get its list of pre-calculated overlapping objects, and an upward force is applied to each of them to raise them up in the air.

```
for (int i = 0; i < m_phantom->getOverlappingCollidables().getSize(); i++ )
    {
        // Note: it is OK to directly cast here, because we filter out
        //       all but moving rigid bodies in the add and remove collidable calls (above)
        hkpRigidBody* rb = static_cast<hkpRigidBody*>(m_phantom->getOverlappingCollidables()[i]->getOwner(
          ));

        rb->forceActivate();
        hkReal height = rb->getPosition()(1);
        hkVector4 force( 0.0f, 2.0f*(10.0f - height) ,0.0f );
        rb->applyForce( force );

    }
```

Creating an hkpAabbPhantom

Unlike `hkpWorld` and `hkpRigidBody`, `hkpPhantoms` do not use Cinfos. You can specify the following details for an hkpAabbPhantom:

- **Aabb**

  The m_min and m_max parameters allow you to specify the AABB that forms the phantom area. Each parameter is an `hkVector4`. By default, these are both set to the origin.

- **Collision filter information**

  This can be used by collision filters to identify the phantom. By default, this is set to zero. See the Collision Filter section for more information on its possible values.

The following example shows the creation of the phantom area from the Fountain example:

```
hkAabb phantomAabb;
phantomAabb.m_min = hkVector4( -2.0f, -10.0f, -2.0f );
phantomAabb.m_max = hkVector4( 2.0f, 10.0f, 2.0f ) ;
m_phantom = new hkpAabbPhantom( phantomAabb );
```

3.2.5.2 hkpShapePhantom

Unlike `hkpAabbPhantom`, `hkpShapePhantom` does not store an explicit Aabb for the broadphase but instead it computes it from its `hkpShape` and associated transform. This allows you to query for the closest points and penetrations within the phantom region with respect to its shape.

`hkpShapePhantom`s do not have a raycast functionality like `hkpAabbPhantom` as they are primarily designed to be used with a given shape with respect to other shapes in the region. Instead they have a linear, or shape, cast which is used to project the entire shape volume into the world.

The `hkpShapePhantom` has two main methods; `getClosestPoints()` and `getPenetrations()`. Both methods take collision point collectors (callbacks) that can be used to process the collision data as it is computed by the collision algorithms (See the `hkpCdPointCollector` and `hkpCdBodyPairCollector` respectively for the collector interface and `hkpSimpleClosestContactCollector` for a sample implementation).

The `hkpShapePhantom` has two default implementations; `hkpSimpleShapePhantom` and `hkpCachingShapePhantom`. The two varieties reflect opposite sides of the usual memory versus speed trade off, with the caching type boasting greater speed at the cost of increased memory usage.

In essence the key difference between the two is that `hkpCachingShapePhantom` caches the agents required to process the collision data. `hkpSimpleShapePhantom` just makes explicit asynchronous calls to the collision detector without storing very much of the state information. The simple type is most useful if you have strict memory budgeting to adhere to, or if the phantom moves significant distances every frame (invalidating the caches). Otherwise, if memory is plentiful and speed is more important then you should see a speedup using the `hkpCachingShapePhantom` instead. However, if you use a very simple sphere or capsule as your shape in your shape phantom, then caching does not increase performance significantly and you might prefer the non caching version.

You can find out more about querying the collision detection system directly, look in the *Getting collision details* section of the Collision Detection chapter, and find out more about character controllers (a common use of shape phantoms) in the Character Control chapter.

3.2.5.3 hkpPhantomCallbackShape

The Havok physics engine offers a different approach to caching user collision queries: The hkpPhantomCallbackShape. This shape is a special hkpShape (N.B. not a hkpPhantom), which simply detects

overlapping objects. As this hkpShape is part of the physics collision detection system, you can use this hkpShape with any other non-special shape: For example, you can associate it with a single-shape rigid body or make it a child shape of a list shape. This shape has no geometry per se. If you want to give a hkpPhantomCallbackShape a shape use the hkpBvShape to associate this callback shape with an actual geometry.

`hkpPhantomCallbackShape` can be useful in a number of game scenarios. For example, a vehicle in a city landscape: an `hkpPhantomCallbackShape` in the form of a triangle can be attached to the front of the car to detect if there are any collidable objects in front of it (for example another AI car, or some scenery), which could be fed back into the AI system in order to attempt to avoid the collision.

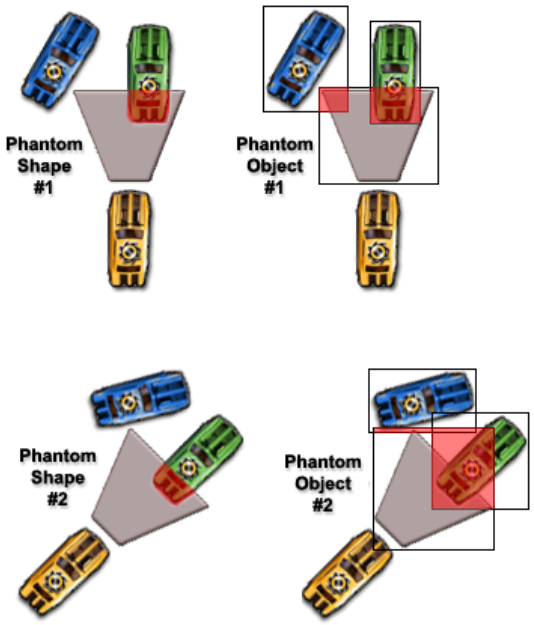


Figure 3.18: Typical in-game use of phantoms, illustrating the difference between phantom shapes and phantom objects

As you can see in the example above, the cars on the left have a phantom shape attached, and the cars on the right have a phantom object attached. The phantom shape on the left performs an accurate check between the shapes and doesn't detect that the blue car is colliding with it in either the top or bottom scenario.

However, with the phantom object on the right, the collision checks are carried out at the Aabb level, and in both the top and bottom scenarios, the blue car is detected to have collided with the phantom, which would feed back less accurate information to the AI system.

In practice, the shape being used for the phantom shape is typically the child shape of a `hkpBvShape`, where that child shape's Aabb is used for the phantom Aabb. If an object is found to collide with the Aabb, a mid and narrowphase collision check is carried out between the child shape and the object colliding with the phantom Aabb. Depending on the outcome, either the callback for the enter event or exit event will be triggered, or no event will be required.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

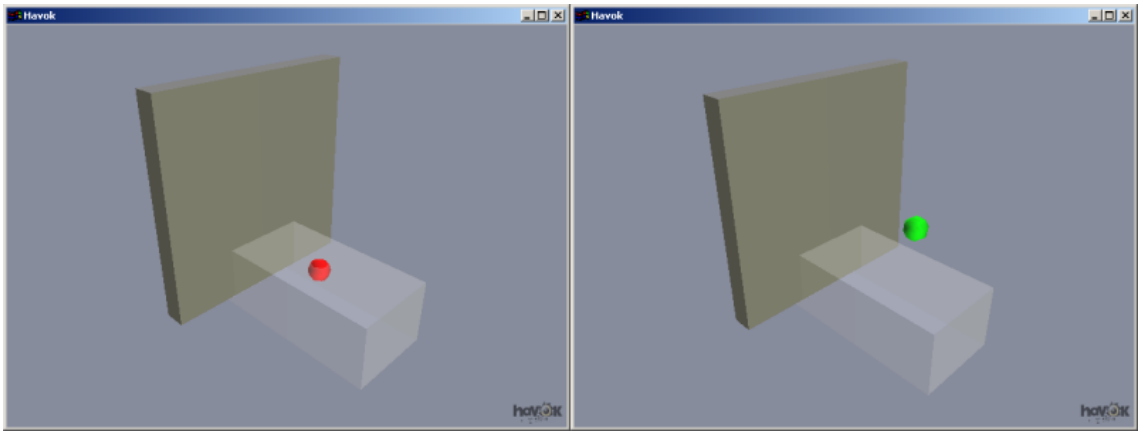


Figure 3.19: An example from the demo framework using a phantom callback shape

You can see the Phantom Events demo illustrated above. In this example, the transparent `hkpBoxShape` has been made the bounding volume of a `hkpBvShape` with a `hkpPhantomCallbackShape` as child. Within the entry and exit event methods (given below) we color the sphere to indicate whether or not it is inside the phantom volume. An upward impulse is applied to the body if it is leaving the phantom volume with a downward velocity. This impulse causes it to bounce back up through the box again whenever it "falls out of the bottom".

```
/// hkpPhantom interface implementation
virtual void phantomEnterEvent( const hkpCollidable* collidableA, const hkpCollidable* collidableB, const
  hkpCollisionInput& env )
{
        // Note: The color can only be changed once the entity has been added to the world
    hkpRigidBody* owner = static_cast<hkpRigidBody*> (collidableB->getOwner());

        // the "Collidables" here are "faked" so it's necessary to get the owner first in order
        // to get the "real" collidable!
    HK_SET_OBJECT_COLOR((int)(owner->getCollidable()), hkColor::rgbFromChars(255, 0, 0));

}

/// hkpPhantom interface implementation
virtual void phantomLeaveEvent( const hkpCollidable* collidableA, const hkpCollidable* collidableB )
{
        // Note: The color can only be changed once the entity has been added to the world
    hkpRigidBody* owner = static_cast<hkpRigidBody*> (collidableB->getOwner());

        // the "Collidables" here are "faked" so it's necessary to get the owner first in order
        // to get the "real" collidable!
    HK_SET_OBJECT_COLOR((int)(owner->getCollidable()), hkColor::rgbFromChars(0, 255, 0));

        // If moving out AND falling down, apply impulse to fire it towards "wall"
    if(owner->getLinearVelocity()(1) < 0.0f)
    {
        hkVector4 impulse(-70.0f, 220.0f, 0.0f);
        owner->applyLinearImpulse(impulse);
    }
}
```

`hkpPhantomCallbackShape` is described further in the Collision Detection chapter.

3.2.5.4 User hkpPhantom types

Currently the phantom types are only used by the demo framework to be able to display the phantoms properly.

`hkpPhantom` is the base class for phantom objects, and as you've seen above Havok provides two implementations of `hkpPhantom` that you can use, `hkpAabbPhantom` and `hkpShapePhantom`. You can also create your own `hkpPhantom` classes by extending one of the provided phantom classes and using one of the free 10 user phantom types (HK_PHANTOM_USER0 -> USER9) and return it in the virtual `getType()` method.

If you do so, you will have to implement `addOverlappingCollidable()` and `removeOverlappingCollidable()` (if you inherit directly from `hkpPhantom`). Be aware of the potential asymmetry in these calls with repect to collision filters, discussed in the Warning at the beginning of the Phantoms section above. These callbacks are called by the simulation when a new object enters or leaves the phantom's Aabb. This Aabb is acquired through the `calcAabb()` method that you must also implement.

3.2.5.5 Using hkpPhantoms

Adding and removing phantoms from the world

Add and remove `hkpPhantom`s using `hkpWorld` `addPhantom()` and `removePhantom()`.

Querying a phantom

You can query an `hkpAabbPhantom`'s list of overlapping entities using its `getOverlappingCollidables()` method. This returns an array of `hkpCollidable`s. `hkpShapePhantom` is queried using `getClosestPoints()` and `getPenetrations()`. Using `hkpPhantomCallbackShape` you get the events as they happen and store them as you see fit.

Moving a phantom

You can move an `hkpAabbPhantom` after adding it to the `hkpWorld` by calling `setAabb()`, passing it the new Aabb's min and max `hkVector4`s. The following example is from the Phantom Object demo, in which the phantom area is moved in a circle:

```
hkReal circleParam = m_time * 0.3f;
hkReal si = hkMath::sin( circleParam ) * radius;
hkReal co = hkMath::cos( circleParam ) * radius;
hkAabb aabb;
aabb.m_min = hkVector4( -phantomSide * 0.5f + si, 0.6f, -phantomSide * 0.5f + co);
aabb.m_max = hkVector4(  phantomSide * 0.5f + si,  20.0f,  phantomSide * 0.5f + co );

m_phantom->setAabb( aabb );
```

The moved `hkpAabbPhantom` automatically updates its list of overlapping entities.

`hkpShapePhantom` is moved by setting the transform in the phantom. Call setTransform() or just setPosition(). If you want to do a linear cast and move the phantom as well, it is more efficient to do

both at the same time as a linear cast alters the Aabb in the broadphase anyway, so if you move the `hkpShapePhantom` and then linear cast separately you are updating the Aabb in the broadphase twice. The call to do the linear cast thus assumes you might want to move the phantom as well, and hence is called `setPositionAndLinearCast()`.

Phantom raycast and linear cast

If you want to cast a ray against the entire world, the ray cast algorithm has to find all the objects that are likely to collide with the ray, and then collide all those objects against the ray. Typically, finding all those potential hit objects in the world is much more expensive than checking the ray against all those potential objects. So if we could speed up the broadphase of the raycast algorithm, we could make the overall algorithm much faster. As we already know that Phantoms are primarily used to cache the broadphase, it is no wonder that the `hkpAabbPhantom` has a `castRay()` function that you can use to perform fast raycasts during simulation. The `hkpAabbPhantom` simply uses its list of overlapping entities as candidates for ray casting. If the ray hits one of the objects stored in the Phantom, it will return the normalized hit normal at the point of intersection, and the ray parameterization value. All you have to do is make the Aabb of the hkpAabbPhantom big enough to contain the ray you want to use. However, be careful not to make this ray too big. For example, if you want to do line of sight checks, the ray might be very long. As a result, you would have to make the ray of the Phantom so big that it would overlap with nearly all objects in the scene, making your raycast horribly slow.

`hkpAabbPhantom` raycasting is useful when you have a small ray which you use every frame at roughly the same position. An example would be defining a phantom object around a particular room that laser targeting guns are operating in, or around a particular area of land within their range. Depending on which type of hit collector you use you can get the closest hit, or all hits, and so on.

ShapePhantoms don't have ray casts, instead they assume the queries will be related to its shape and thus it implements a linear cast. Linear casting is explained in the collision detection chapter, but basically it will sweep the shape held in the Phantom through the world and return the hits through the collector.

3.2.6 Scale

Havok is designed to work in *meters*. This means a linear velocity of length 1 implies a linear velocity of 1 meter per second, and a unit cube is one meter cubed. It is possible to change this scale although it is *not* recommended - If you are considering this option, please contact support.

There are some hard coded numerical tolerances in the engine which you do not have access to. This means that you cannot change the scale of the engine by a large factor (e.g. you cannot use units of centimeters, or of kilometers). However, if you bear in mind (and suitably adjust) some externally available parameters, you could use a scale of feet for example. Most of these parameters are in the struct `hkpWorldCinfo`. In each case you should scale the parameters by the required factor (in the case of using feet for example, set the values to 3.28 times their recommended default values). The default values can be viewed in `hkpWorldCinfo`.

Simulation Parameters:

- m_gravity
- m_expectedMaxLinearVelocity
- m_broadPhaseWorldAabb.m_min, m_broadPhaseWorldAabb.m_max

Collision Detection Internal Tolerances:

- m_collisionTolerance
- m_iterativeLinearCastEarlyOutDistance

In addition to these `hkpWorld` parameters, you must change the following shape-based values:

Per-Shape Parameters:

The *radius* of any convex shapes you create, either on construction or using the method `setRadius()`.

3.2.7 Constraints

Constraints are essentially restrictions on the freedom of movement of objects, allowing you to easily create complex systems such as hinges, body joints, and wheels. Whatever the type of constraint, these elements can help to make the dynamic environment very rich.

This document first introduces you to constraint spaces and constraint construction, and then outlines the different types of constraint that are supported by Havok and shows you how to create and configure them. Some guidelines are also provided for the use and limitations of constraints.

In addition to reading this section, you should take time to look at and play with the demos in the dynamics/constraints folder provided with your Havok Physics installation, examples from which are used throughout the text. Each of these simple demos illustrates a particular aspect of working with Havok constraints.

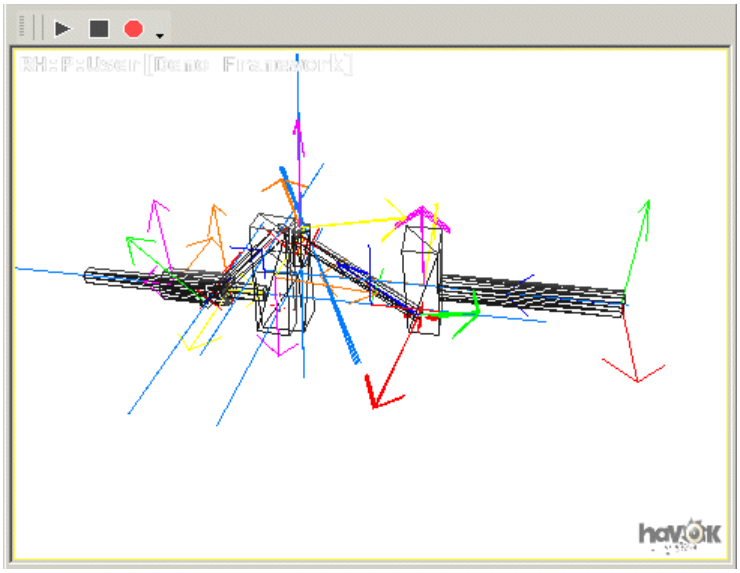


Figure 3.20: Constraints abound

3.2.7.1 About constraints

Constraint spaces

In rigid body dynamics each body has 6 degrees of freedom:

- 3 translational degrees of freedom
- 3 rotational degrees of freedom

Each type of Havok constraint can remove or limit one or more of these degrees of freedom for its constrained bodies.

Depending on the number and type of these limitations, we get different types of constraint, from the simple point-to-point constraint to the much more complicated ragdoll constraint. For example, with a point-to-point constraint, the constrained objects are completely free to rotate around the constraint pivot point, but have no linear freedom relative to each other in any direction - they are attached together at the point. However, with a ragdoll constraint, the objects not only have no linear freedom, but the possible relative orientation is also restricted, for example in the case of a shoulder.

In each constraint, these angular and linear limits are defined in terms of the constraint's coordinate system or *constraint space*. Because a constraint restricts the movement of its objects relative to each other, a constraint also needs to maintain a transform from each object's local space to the constraint space.

Each constraint stores the minimum amount of information it needs to construct this transform. For instance, because there are no angular limits in a point-to-point constraint, you only need to specify the pivot point in local space for each object - the objects' orientation is unimportant.

How Useful are Constraints?

Constraints which limit object movement in different ways can result in a number of useful applications. Take a look at some of these examples:

Swinging sign, door, elbow joint, spinning wheel	Limited Hinge Constraint / Hinge Constraint
Elevator, mechanical piston	Linear movement limited to a single axis (Prismatic Constraint)
Shoulder joint, caravan hitch joint	Ragdoll Constraint
Bell clanger, neck joint	Limited ball and socket joint
Car Wheel	Wheel Constraint

Table 3.2: Examples of Constrained Systems

More complex constraint setups can be designed and implemented using the constraint construction kit. However, the more complex the constraint, i.e. the more degrees of freedom it limits of constraints, the more expensive it is to compute.

Constructing Constraints

Each Havok constraint allows you to specify the rigid body or bodies to be constrained, and the necessary axes and pivot points, in either world space or the objects' local spaces. Some constraints also allow you to specify additional parameters, such as linear or angular limits and friction values.

Specifying axes and pivot points in world space is a quick and simple approach to constraint setup. These

are converted to the local space of both objects. However, this approach works only if the objects are already correctly positioned relative to each other in world space. If this is not necessarily the case, you should use the more complex local space setup. You can see an example of this in the section on Ragdoll constraints.

When you create a constraint in world space, the constraint's `isValid()` method is automatically called. This checks that you have provided valid setup information for the constraint - for instance, that unit vectors are actually unit vectors, perpendicular vectors are actually perpendicular, and that you have provided reasonable linear and angular limits. The implementation of this method depends on the constraint type. If the information you provide is invalid, Havok will assert in debug builds.

Constraint Data, Instance and Runtime

The basic idea of constraint sharing is that some constraints can be pretty large, and if you want to simulate 50 ragdolls, it makes sense to share common data between all the ragdoll constraints. Therefore, constraints are split into 3 distinct classes:

- hkpConstraintData

  This is the shared data between the constraints. Actually this class is just an abstract class and implemented by all constraint implementations like hkpBallAndSocketConstraintData or hkpRagdollConstraintData. This data can be used several times, however all instances uses this shared data will share the data. That means if you change the values in the data, it affects all instances. This is particularly important for motors.

- hkpConstraintInstance

  This class is used to create an instance of a constraint. An instance has two pointers to rigid bodies, a pointer to a hkpConstraintData and a priority. Note that this class is not abstract and is not subclassed.

- hkpConstraintRuntime

  Some constraints sometimes actually need some data for simulation. An example is the hkpBreakableConstraintData, which needs to get access to the impulses applied by its child constraint. This data is stored in the hkpConstraintRuntime memory. This runtime memory is allocated by the system when you add the constraint to the world. Actually it is up to the constraint data implementation to decide how and when to allocate the runtime data:

  - The constraint data implementation can decide not to be shared and store the constraint data internally. In this case no external constraint runtime is allocated.
  - The constraint data implementation can decide that it always needs a runtime information and requests an external runtime information allocation always.
  - The constraint data implementation can decide that it does not care about the runtime information. It checks the user flag hkpConstraintInstance::m_wantRuntime and allocates a runtime only if needed.

  The system tries to organize the runtime information of constraints in a cache friendly way. That means the system will move runtime information around in memory. So *never keep a pointer to a hkpConstraintRuntime.*

There are a few functions to get access to various information:

- If you have a hkpConstraintData class, you can call hkpConstraintData::getType() to get its type and then upcast the abstract interface to the correct implementation.

- hkpConstraintInstance is not subclassed, you can simply use it as it is.
- You can get access to the runtime information by calling hkpConstraintInstance::getRuntime(). In most cases you only get the runtime if you requested a runtime by setting hkpConstraintInstance::m_wantRuntime to true before you added the constraint to the world. When you have a hkpConstraintRuntime point, you need to cast this void* pointer into something useful. E.g.

```
// Example to get the detailed runtime information from a ball and socket constraint

    hkpConstraintRuntime* rt = myInstance.getRuntime();
    hkpBallAndSocketConstraintData::Runtime* runtime = hkpBallAndSocketConstraintData::getRuntime( rt );
    hkReal impulseX = runtime->m_solverResults[hkpBallAndSocketConstraintData::SOLVER_RESULT_LIN_0];
    hkReal impulseY = runtime->m_solverResults[hkpBallAndSocketConstraintData::SOLVER_RESULT_LIN_1];
    hkReal impulseZ = runtime->m_solverResults[hkpBallAndSocketConstraintData::SOLVER_RESULT_LIN_2];
```

Constraint Atoms

hkpConstraintAtoms are basic building blocks used to create hkpConstraintDatas. While hkpConstraintData has an intuitive real-world interpretation like hinge or wheel contraint, then hkpConstraintAtom is a lower level mathematical information on how certain degrees of freedom are restricted.

E.g. a hkpHingeConstraintData would be composed of three atoms. The first atom always specifies the constraint's space in reference to the constraint bodies. Here we need to define the pivot points and the free rotation axis of the constraint. The following atom restricts the linear position of the pivot points of two interacting bodies, and the next one restricts angular movement of the bodies to the hinge axis only. A hkpLimitedHingeData would be an expansion of hkpHingeData by add yet another atom for adding angular limit and another one for adding angular friction or motors.

We use hkpConstraintAtoms as they allow us to reduce code size by reusing the same types of atoms in different ways to form various hkpConstraintDatas. Once can use the atoms to create their own custom constriants.

Adding and removing constraints

Once you have created a constraint, you can add it to the hkpWorld using the world's `addConstraint()` function. Similarly, you can remove constraints using `removeConstraint()`.

```
hkpBallAndSocketConstraintData* bsData = new hkpBallAndSocketConstraintData();
    hkpConstraintInstance* bsInstance = new hkpConstraintInstance( bodyA, bodyB, bsData );
    m_world->addConstraint( bsInstance );
    bsData->removeReference();
    bsInstance->removeReference();
```

Or using a convenience function in hkpWorld

```
hkpBallAndSocketConstraintData* bsData = new hkpBallAndSocketConstraintData();
    m_world->createAndAddConstraintInstance( bodyA, bodyB, bsData )->removeReference();
    bsData->removeReference();
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Like entities, shapes, and actions, constraint data and instances are referenced objects. This means you can call `removeReference()` on them once you are finished with them - usually after you have added them to the world, as in the example above. Havok will look after deleting each constraint for you once it is no longer in use. You can find out more about reference counting in the Havok Base Library document.

Constraint limits

As you know, constraints can limit one or more degrees of freedom for their constrained bodies.

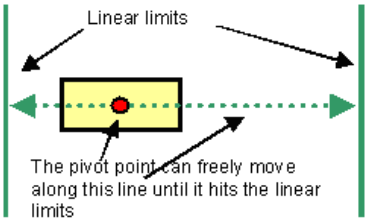


Figure 3.21: Linear limits along one axis

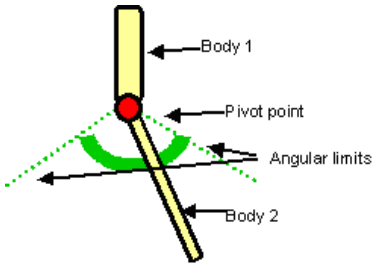


Figure 3.22: Angular limits around one axis

You can specify linear or angular limits like these for a number of Havok constraints. You can see examples of how to do this in the sections on limited hinges, ragdoll constraints, and prismatic constraints.

A number of Havok constraints have friction limits for certain degrees of freedom (usually unconstrained). For example, when creating a limited hinge, you can specify a friction value for the constrained bodies' angular movement around the hinge axis. Other constraints with friction limits include prismatic constraints and point-to-path constraints.

Updating constraints

Once you have created a constraint, you can update its data by calling the appropriate constraint class `set..()` methods. For debug builds, you can call `isValid()` yourself to ensure that the constraint still has valid information.

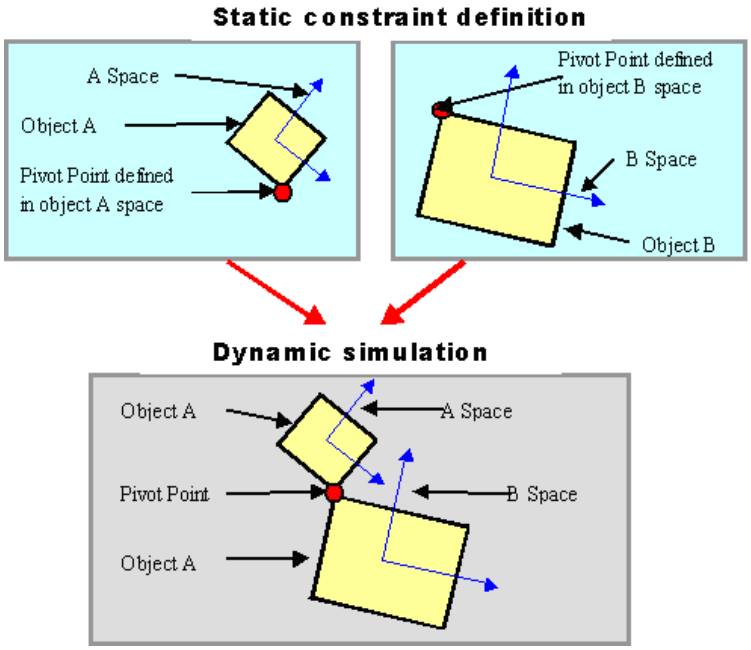
3.2.7.2 Constraint types

hkpConstraintData

The constraint classes in Havok all inherit from the abstract base class hkConstraint. This includes a number of methods for getting and setting basic properties, and updating the constraint.

Ball and socket constraints

A ball and socket constraint (also known as a point-to-point constraint) forces its objects to try to share a common point in space. The objects can move freely about this space, but always have a point in common. This point is the constraint's pivot point:



At creation of the constraint, the pivot point has to be defined in the object space of each object involved. During runtime the constraint tries to apply forces to the objects so that the two pivot points defined by the two objects match. Using the `setInWorldSpace()` method, you can also define the pivot point in world space, which is then transformed into the body space of each object. As you know, setting the pivot point in this way works only if both objects are already positioned in world space in the way you want them to be constrained.

The following snippet shows how to use a ball and socket constraint to connect two objects - rb0 and rb1 - defining the pivot point in world space. When the simulation is run, the bodies spring together in order to try and share this common point in space, and try and maintain this common point throughout simulation. You can see the complete code for this example in the Ball And Socket demo.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

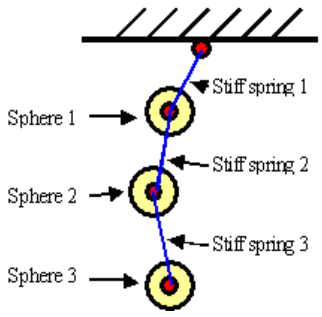
```
hkVector4 halfSize(0.5f, 0.5f, 0.5f);
    hkVector4 size;
    size.setMul4(2.0f, halfSize);
    hkVector4 position(size(0), -size(1) -0.1f, 0);
    hkpBallAndSocketConstraintData* bs;
    {
        // Set the pivot
        hkVector4 pivot;
        pivot.setAdd4(position, halfSize);
        pivot(0) -= size(0);    // Move pivot to corner of cube

        // Create the constraint
        bs = new hkpBallAndSocketConstraintData();
        bs->setInWorldSpace(rb0, rb1, pivot);
        hkpConstraintInstance* instance = new hkpConstraintInstance(rb0, rb1, bs);
        m_world->addConstraint( instance );
        bs->removeReference();
        instance->removeReference();
    }
```

You can use a series of such constraints between adjacent bodies to form a chain.

Stiff spring constraints

The stiff spring constraint is similar to the point-to-point constraint except for one important detail: it holds the constrained bodies apart at a specified distance, as if they were attached at each end of an invisible rod. A stiff spring requires a point in the body space of each of two bodies, and a constant distance by which these two points must be kept separated. Each body is free to rotate around its point.

You can easily create chains of objects using stiff springs, as in the diagram below:



You create a stiff spring between two objects using an hkpStiffSpringConstraintData, as in the example below. You can specify the points in world space, as shown, or in the bodies' local space. By default, the stiff spring's length is the original distance between these points in world space. If you like, you can set a different rest length using the `setSpringLength()` function.

You can see the complete code for this example in the Stiff Spring demo.

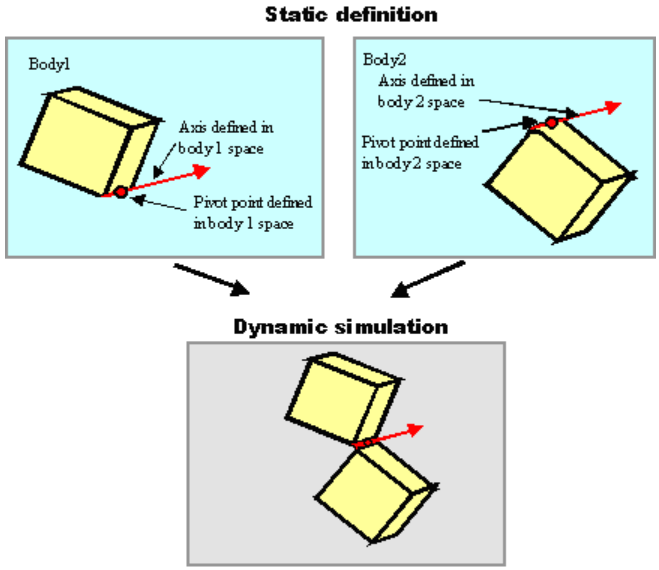
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
hkpStiffSpringConstraintData* spring;
    {
        // Create constraint
        spring = new hkpStiffSpringConstraintData();
        spring->setInWorldSpace(rb0, rb1, position1, position2);
        hkpConstraintInstance* instance = new hkpConstraintInstance(rb0, rb1, spring);
        m_world->addConstraint( instance );
        spring->removeReference();
        instance->removeReference();
    }
```

You can use hkpStiffSpringConstraintData to create a bouncy spring by making it malleable. You can find out more about how to do this in the section on malleable constraints.

Hinge constraints

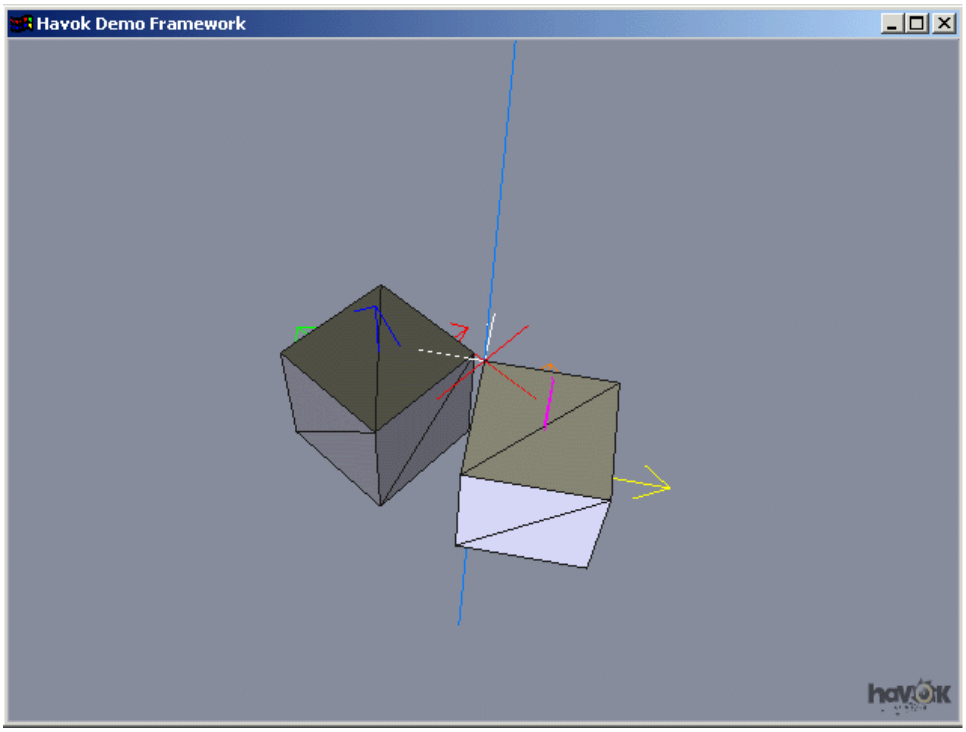
The hinge constraint allows you to simulate a hinge-like action between two bodies. You specify a line segment in body space for each body, with a position and a direction. The two lines then attempt to match position and direction, thereby creating an axis around which the two bodies can rotate.



The following snippet from the Physics / Api / Constraints / Hinge demo shows you how to constrain two cubes (one fixed and one movable) to be hinged together. To set up the constraint, you need to specify the point around which you wish to create the hinge and the common axis of rotation through this point (here aligned with the Z-axis). You can do this in world space using the hkpHingeConstraintData's `setInWorldSpace()` function, provided of course that the cubes are correctly positioned.

```
hkpHingeConstraintData* hc;
    {
        // Set the pivot
        hkVector4 pivot;
        pivot.setAdd4(position, halfSize);
        pivot(0) -= size(0);    // Move pivot to corner of cube

        hkVector4 axis(0, 0, 1);

        // Create constraint
        hc = new hkpHingeConstraintData();
        hc->setInWorldSpace(rb0->getTransform(), rb1->getTransform(), pivot, axis);
        m_world->createAndAddConstraintInstance( rb0, rb1, hc )->removeReference();
        hc->removeReference();
    }
```

When you run the demo, you can see the movable cube rotating around the hinge axis - shown here as the vertical line in the middle of the picture.



Limited hinge constraints

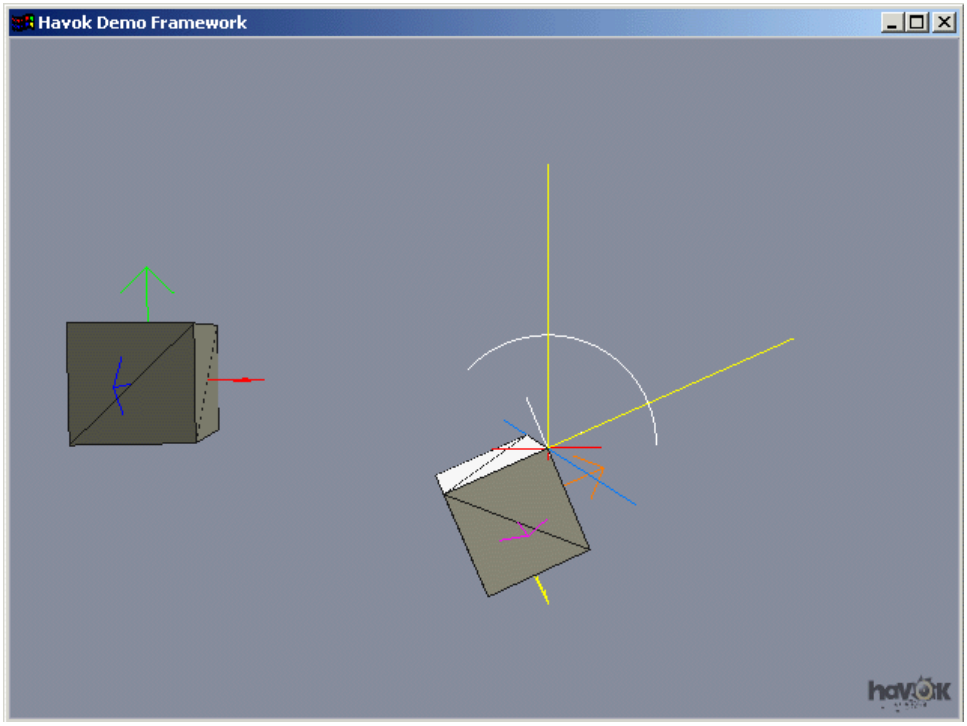
An hkpLimitedHingeConstraintData is similar to a hinge constraint, but also allows you to specify the maximum angle to which the hinged bodies can rotate. This can be useful for preventing interpenetration. Adding limits to a hinge requires another axis, perpendicular to the hinge's rotation axis. You specify the angular limits relative to this new axis, which is created automatically for you when you use the `setInWorldSpace()` function. The default limits are -PI and PI.

As mentioned previously, you can also provide a friction value for the constrained bodies' movement around the hinge axis. The default friction value is zero.

The following example, from the Physics / Api / Constraints / Limited Hinge demo, shows the creation of a limited hinge between two boxes - like the hinge example, one box is movable while the other is fixed. As you can see, you specify the angular limits using the `setMinAngularLimit()` and `setMaxAngularLimit()` methods. Note that the total range of movement should not be more than 2*PI.

```
hkpLimitedHingeConstraintData* lhc;
    {
        // Set the pivot
        hkVector4 pivot;        pivot.setAdd4(position, halfSize);
        pivot(0) -= size(0);    // Move pivot to corner of cube

        hkVector4 axis(0.0f, 0.0f, 1.0f);

        // Create constraint
        lhc = new hkpLimitedHingeConstraintData();
        lhc->setInWorldSpace(rb0->getTransform(), rb1->getTransform(), pivot, axis);
        lhc->setMinAngularLimit(-HK_REAL_PI/4.0f);
        lhc->setMaxAngularLimit(HK_REAL_PI/2.0f);
        m_world->createAndAddConstraintInstance( rb0, rb1, lhc )->removeReference();
        lhc->removeReference();
    }
```

You can see the angular limits in the following screenshot from the demo. The movable cube is free to rotate around the hinge axis within the limits indicated by the white arc.



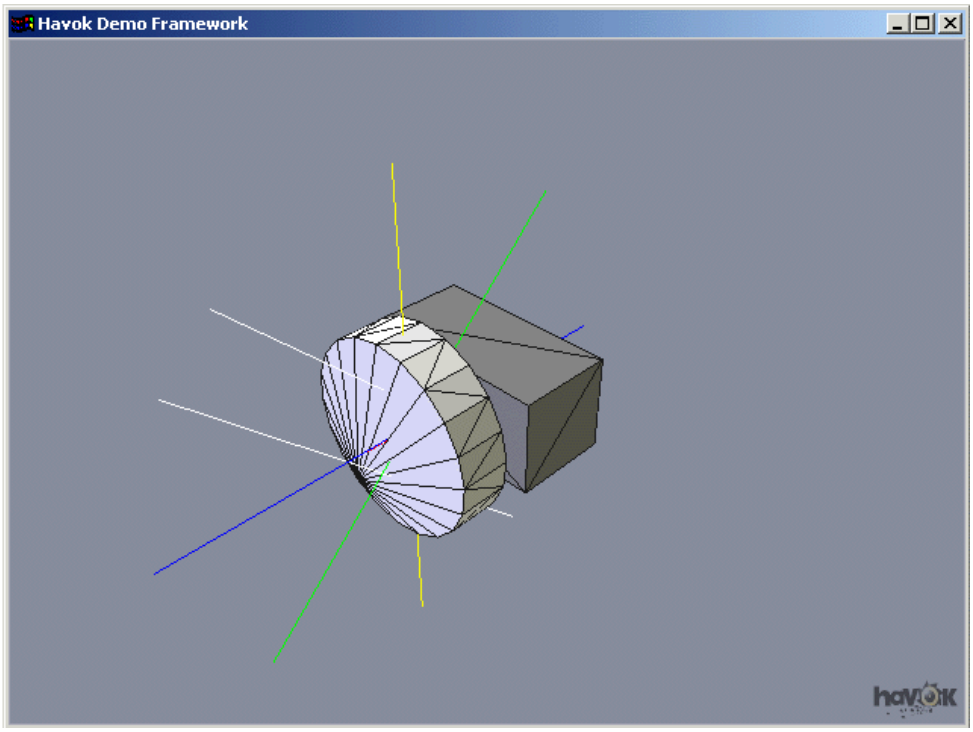
Wheel constraints

The wheel constraint is essentially a hinge with an extra free translation/suspension axis. This constraint allows you to attach a wheel to another object - for instance, a car chassis. The rotation axis provides the axle of the wheel, allowing it to spin freely without limits. The suspension axis works relative to

the chassis and allows movement along the direction of this axis. You can provide damping and stiffness values for the suspension, as well as limits for the movement of the wheel in the suspension direction, when creating the constraint.

The hkpWheelConstraintData lets you dynamically orient the rotation axis relative to the chassis to allow for steering, using the `setSteeringAngle()` function. To do this, the wheel constraint needs an additional steering axis, which you can use to specify this angle. In many cases, the suspension and the steering axis will be the same axis, though you can specify different axes when creating the constraint if you like.

You can see these axes and limits in the following screenshot from the Wheel demo. The rotation axis is shown in blue, the steering axis in yellow, the suspension axis in green, and the suspension limits are shown as two white lines.



Now let's look at how to create a wheel using the wheel constraint. The snippet below is also from the Wheel demo. As you can see, you can use the hkpWheelConstraintData's `setInWorldSpace()` function to specify

- the two rigid bodies (first the wheel, then the chassis)
- the pivot point for the rotation axis, in world space
- the rotation axis, in world space
- the suspension axis, in world space
- the steering axis, in world space

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

You can also, of course, specify the point and axes in body space using `setInBodySpace()`. Note that if you use this function, you only need to set the pivot point and rotation axis in both spaces - the steering and suspension axes are specified in chassis space only.

The example also shows setting suspension limits, suspension stiffness, and suspension damping for the wheel.

```
hkVector4 suspension(1.0f, 1.0f, 0.0f);
    suspension.normalize3();
    hkVector4 steering(0.0f, 1.0f, 0.0f);
    steering.normalize3();

    m_wheelConstraint = new hkpWheelConstraintData();

    m_wheelConstraint.setInWorldSpace( m_wheelRigidBody->getTransform(), m_chassis->getTransform(),
      m_wheelRigidBody->getPosition(), axle, suspension, steering );
    m_wheelConstraint.setMaxSuspensionLimit( .2f );
    m_wheelConstraint.setMinSuspensionLimit( -.5f );

    m_wheelConstraint.setSuspensionStrength( 0.01f );
    m_wheelConstraint.setSuspensionDamping( m_world->m_solverDamp * 0.25f );

    m_world->createAndAddConstraintInstance( m_wheelRigidBody, m_chassis, m_wheelConstraint )->
      removeReference();
```

You want users to be able to steer the wheel, so during the game `setSteeringAngle()` is used to change the steering angle depending on input from the user. The following snippet shows the code to turn the wheel to the left.

```
const hkReal maxAngle = (HK_REAL_PI * .25f);
    const hkgPad* pad = environment->m_gamePad;

    if( (pad->getButtonState() & HKG_PAD_DPAD_LEFT) )
    {
        m_steeringAngle += 1.25f * environment->m_deltaTime;
        // Clip steering angle to a max of PI/4 radians
        if( m_steeringAngle < maxAngle)
        {
            m_wheelConstraint->setSteeringAngle( m_steeringAngle );
        }
        else
        {
            m_steeringAngle = maxAngle;
        }
    }
```

Note:
In general, the Havok Vehicle Kit is a better way to construct four wheeled vehicles than using wheel constraints, especially racing or fast moving vehicles. Take a look at the Vehicle Physics chapter for more information.

Pulley constraints

Pulley constraint allows you to apply leverage between two dynamic objects. It represents a rope and pulley mechanisms where the pulleys have fixed positions in world, and the rope's ends are connected to dynamic bodies. By setting the leverage ratio you set the proportion of the forces applied by the rope onto the connected bodies. To initialize a pulley constraint you need to call either

`hkpPulleyConstraintData::setInBodySpace()` or `hkpPulleyConstraintData::setInWorldSpace()` passing the transforms of the linked bodies, the connection points where the rope is attached to the dynamic bodies, and the global positions of the fixed pulley blocks. Below is a simple setup code.

```
{
          hkpPulleyConstraintData* pulley = new hkpPulleyConstraintData();
          hkReal leverageRatio = 3.0f;

          pulley->setInBodySpace(body0->getTransform(), body1->getTransform(),
                                       bodyPivot0, bodyPivots1,
                                       worldPivots0, worldPivots1,
                                       leverageRatio);

          hkpConstraintInstance* constraint = new hkpConstraintInstance(body0, body1, pulley);

          world->addConstraint( constraint);
          constraint->removeReference();
          pulley->removeReference();
        }
```

Both of the initializing functions calculate the rope length corresponding to the positions of the linked bodies and assuming the rope is fully stretched. You can later modify the length of the rope and the leverage ratio calling the functions below.

```
pulley->setRopeLength(9.0f);
            pulley->getLeverageOnBodyB(2.0f);
```

A pulley mechanism when given an input force on one side applies `leverageRatio` times greater force on the other side. However, one also has to apply `leverageRatio` times the displacement to move the other end by a requested distance. The current rope length is computed as the sum of the distance from the first body to the first pulley attachment point and the distance from the second body to the second attachment point multiplied by the leverage ratio.

The pulley gets unstable when one of the attached bodies gets very close to the pulley block. Note also that the pulley is not a fixed-distance constraint, therefore it does not restrict the bodies from coming too close to the pulley attachment points.

This constraint is different from most other constraint in the sense that it

- the pulley attachment points must be specified in the fixed global space, and cannot be linked to a moving body;

- the constraint applies different forces to each of the bodies.

Ragdoll constraints

Ragdoll constraints allow you to realistically simulate the behavior of body joints, such as hips, shoulders, and knees.

Once you have decided what degree of movement a joint should have, you can model it by specifying limiting values for the ragdoll constraint. This constraint also requires you to specify which of its rigid

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

bodies is the reference body for the joint - this is used by the constraint as a reference when defining the limits for the motion of the attached body. For example, when you move your upper arm, your forearm always moves with it. So when modeling an elbow joint, you would specify that the upper arm was the reference body, making the forearm the attached body.

It is worth noting that ragdoll constraints (and limited hinge constraints) provide support for joint friction using a `setMaxFrictionTorque` function. The value passed to this function is not a parametric friction value between 0 and 1 but is dependent on the masses of the constrained bodies. To stiffen up a constraint try setting the friction high enough that the joint is completely stiff and then reduce the value until the desired behaviour is achieved.

The following sections will introduce you to the possible ragdoll constraint limits and how they work, and explain how to set up a ragdoll constraint using the Havok Physics SDK.

Ragdoll constraint limits

There are a number of limitations that need to be considered when dealing with ragdolls.

Choosing the parameters

An important part of creating a realistic ragdoll joint is choosing appropriate values for the various constraint limits. So before looking at the code, try stretching your right arm out in such a way that if you turn your head 90 degrees to the right, you are looking straight along it with the palm facing downwards. Imagine a pivot point at your shoulder. How would you model the freedom that your arm has to move using a ragdoll constraint?

First of all, you need to find the **twist axis** . This is easy in this case, as it is the axis running the length of your arm.

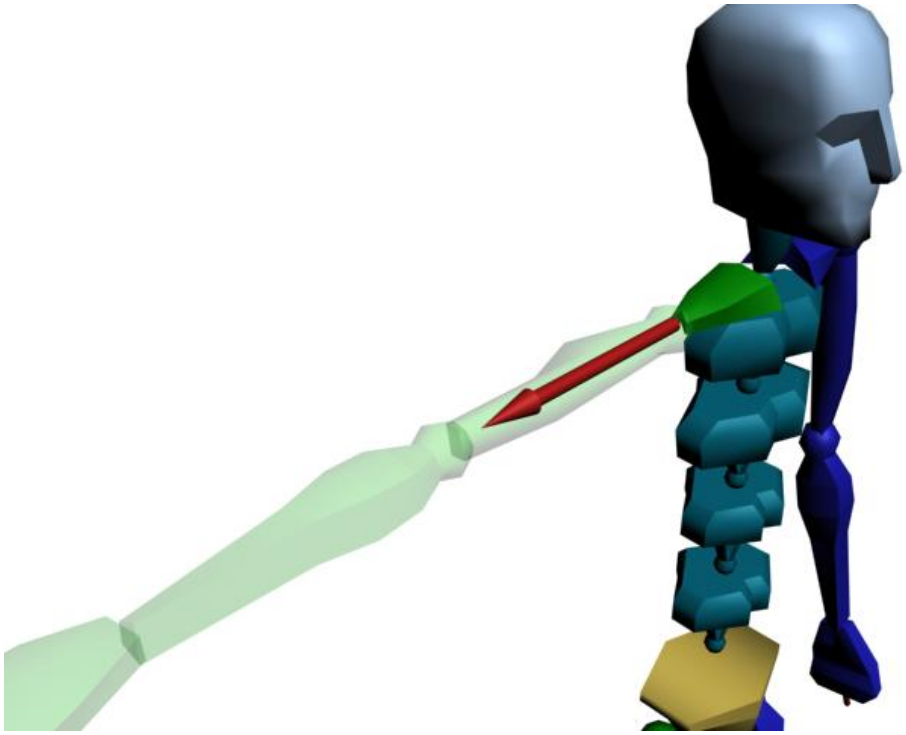


Figure 3.23: Twist Axis

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Now with your torso immobile, see how far your upper arm can twist around this axis relative to your torso. You can probably twist your upper arm forwards by about 5 degrees and backwards by the same amount (your lower arm and wrist can twist further than this). The **twist limits** for your joint will need to reflect this.

Next you need to figure out your arm's available movement space. Again with your arm outstretched, make the widest arc that you can with your arm. Try to imagine a circular cone that includes all of the areas your arm reached. We will prune this cone later so it takes into account the areas you couldn't reach.

We specify the cone of movement using the **cone angle** .

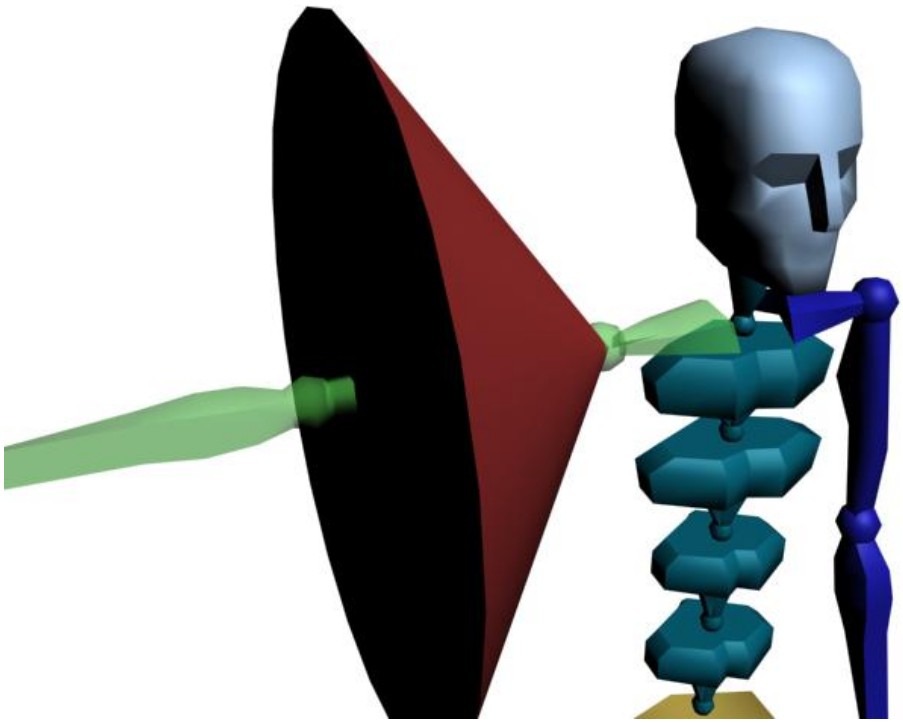


Figure 3.24: Cone of Movement

In the diagram, the cone is centred around the arm so there is equal movement upwards as downwards. However, you can probably swing your arm upwards until it is not quite vertical, and then around in an unbroken arc forward until it is pointing downwards. However, you will (almost certainly!) not be able to move your upper arm very far backwards relative to your torso. So, we need to tightly restrict its backward movement by clipping the cone of movement.

Havok uses two cones of possibly different angles aligned around a common axis to prune the movement cone. In Havok we call the common axis for these clipping cones the plane normal. To model the shoulder, let's align your coordinate system so that the plane axis is pointing straight forward from the shoulder.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

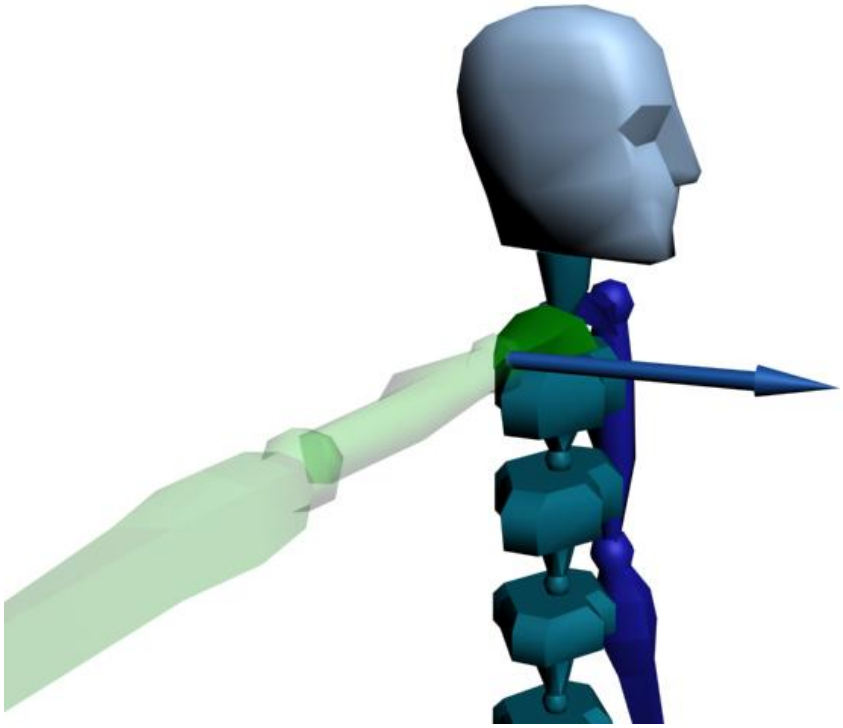


Figure 3.25: Plane Axis

We can specify a wide cone at the back so it significantly clips the arm's movement. The front cone is small so that it does not affect the arm's movement at all. The specification of these cones is done using the **plane max** and **plane min** angles, which are explained below.

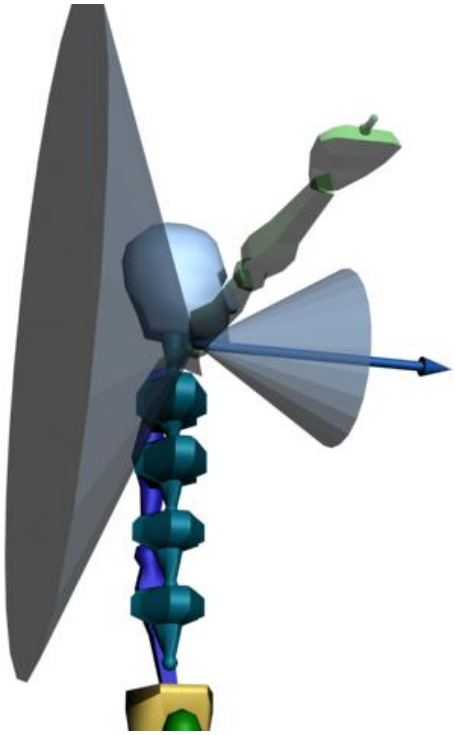


Figure 3.26: Plane Cones

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Here is the restriction of the movement given by the plane cones we've described.

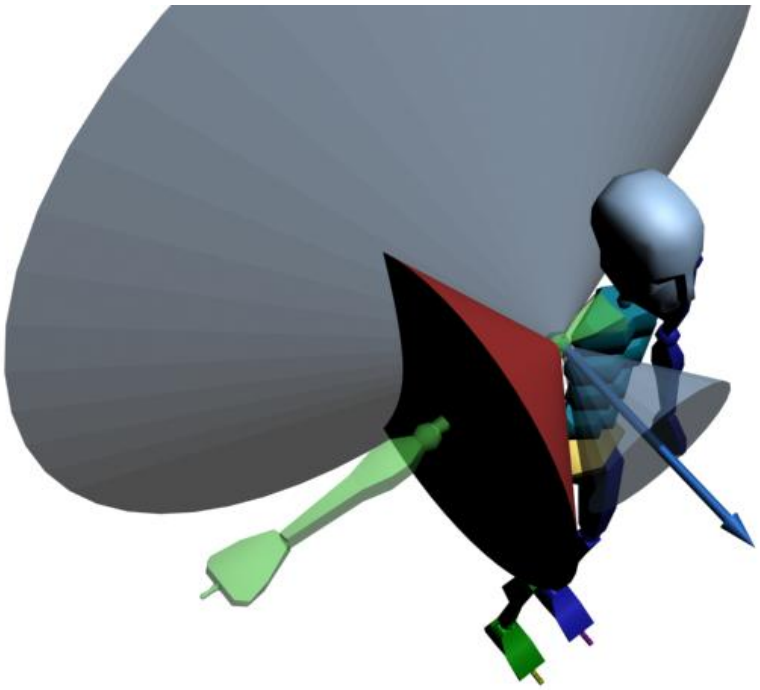


Figure 3.27: Plane Cones restricting Cone of Movement

The resulting movement is significantly restricted now in the backwards direction.

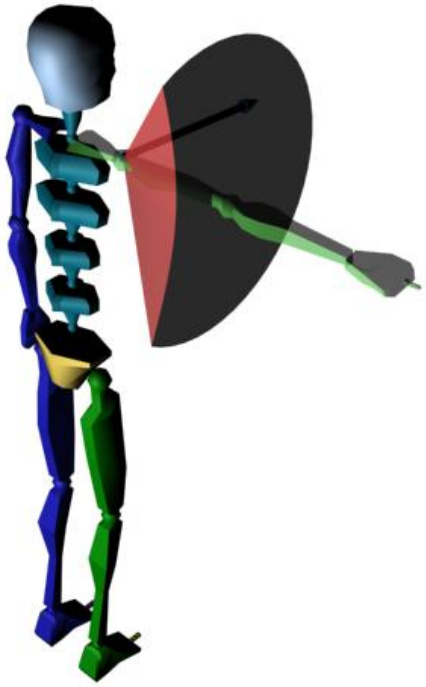


Figure 3.28: Resulting Cone of Movement

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Ragdoll axes

You restrict how a ragdoll constraint's attached body can move relative to the reference body using the constraint's cone, plane, and twist limits. These are specified relative to the constraint's three axes.

As you know, you can define a three dimensional frame of reference using an origin and three axes. The ragdoll constraint has a set of transformation matrices defining the transform from the objects' space to the constraint space. The constraint space frame of reference for a rag doll constraint is constructed by defining:

- the origin as the pivot point of the constraint between the reference and attached bodies
- the first axis as the twist axis of the ragdoll constraint - this is the axis around which the attached body can twist relative to the reference body.
- the second axis to be the ragdoll's plane normal, also known as the constraint's plane axis
- the third axis to be the cross product of the twist axis with the ragdoll's plane normal

The following diagram shows a ragdoll constraint's reference body, attached body, and axes, viewed from above. The twist crossed with plane axis is pointing out of the page. As you can see, the twist axis is aligned along the longest axis of the attached body. This is usually the case when modeling body joints, as this is the axis that is constrained within the constraint's cone of movement. You will see more clearly how this works in the cone and plane limits sections.

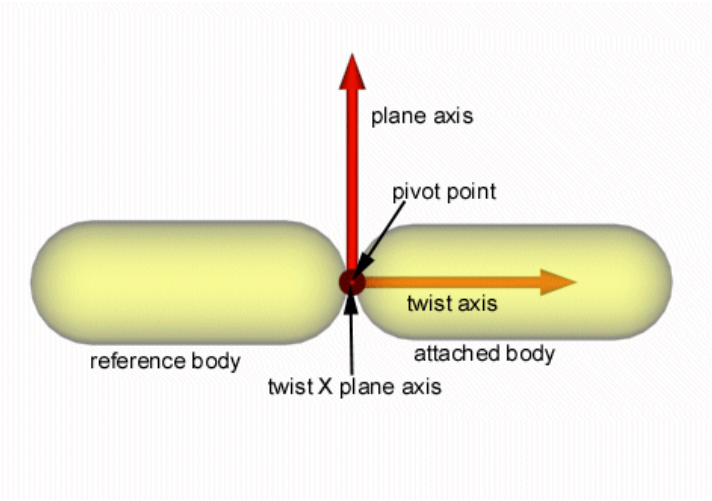


Figure 3.29: Ragdoll constraint bodies and axes, top view

Note that a ragdoll constraint's plane axis has to be at right angle to its twist axis, as shown in the diagram.

Twist limits

You use the ragdoll constraint's twist limits to define the relative permitted rotation around the twist axis between the two constrained bodies. The angular limitations around the twist axis - i.e. the twist limits - should be set in the range [-pi/2..0] for the minimum and [0..pi/2] for the maximum. You specify the twist limits relative to the twist X plane axis. The following diagram should help you to visualize the

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

effect of these limits. In this example, the attached body can twist relative to the reference body only within the range indicated in blue.

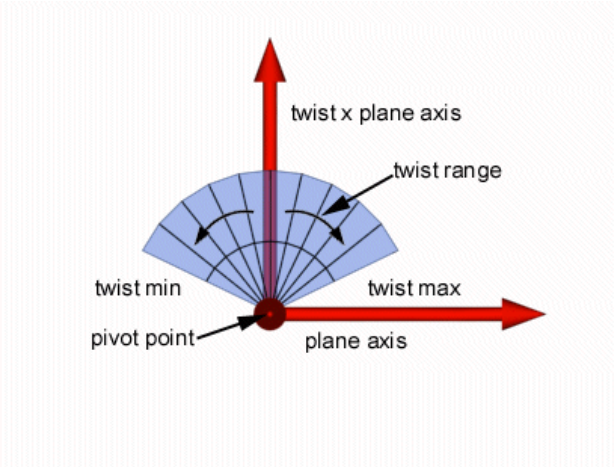


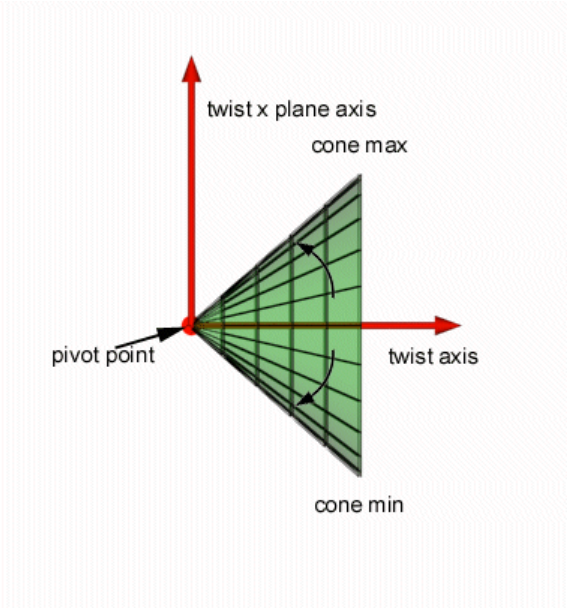
Figure 3.30: Twist limits, front view

Cone angle

The ragdoll constraint's cone of movement is circular and has its point at the pivot point. It limits the rotational movement of the attached body so that its twist axis can only move relative to the reference body *within* that cone. The cone is circular at all times and its size is defined using the cone angle, which is specified relative to the twist axis. The cone angle should be set in the range [0..pi].

The following illustration should help you to visualize how this limit works. You can see the attached object's cone of permitted movement and the constraint axes, viewed from the side and in perspective. In each case, the attached body's twist axis is free to move within the space indicated by the green cone.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



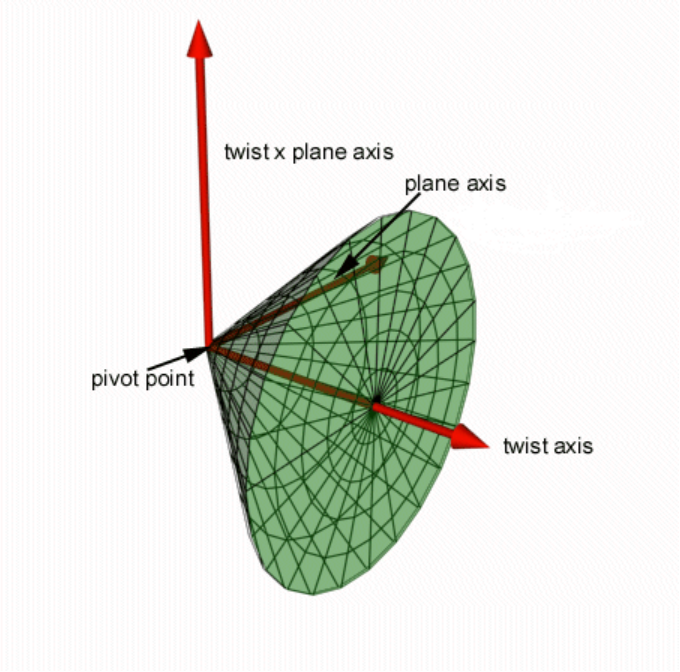


Figure 3.31: Cone of movement, side and perspective views

Alternatively, you can specify min and max angles for the cone. If these values are different, this has the effect of rotating the circular cone around the plane axis (the method you use to do this actually moves the twist axis in the reference body's space).

Plane limits

You can use a ragdoll constraint's plane limits to clip the cone of available movement specified with the cone limits. As you know, one of the axes you need to define for a ragdoll constraint is its plane axis. This is the plane normal to a plane in the twist axis / twist X plane axis space. The constraint's plane limits indirectly specify the radii of two circular cones with their points at the constraint's pivot point,

known as the plane min cone and plane max cone. Both cones are aligned with the plane axis.

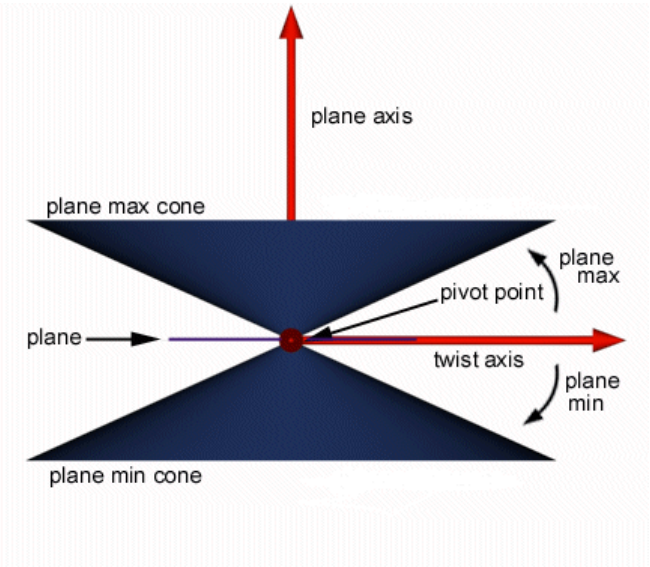


Figure 3.32: Plane and plane cones, top view

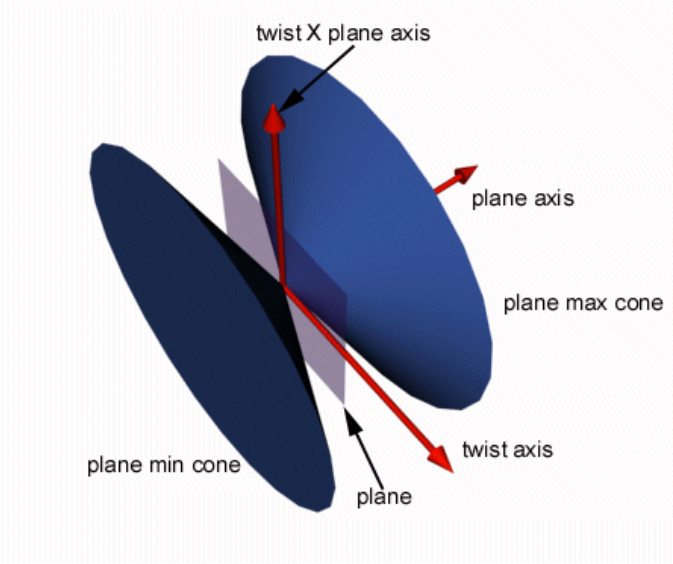


Figure 3.33: Plane and plane cones, perspective view

As you can see in the above diagrams, the plane max cone extends in the plane normal direction while the plane min cone extends in the negative plane normal direction. These cones limit the movement of the constrained body so that it can only move *outside* these cones. The plane limits should be set in the range [-pi/2..0] for the plane min value and [0..pi/2] for the plane max value, and are specified relative to the plane - large plane values produce small plane cones, and vice versa. While the illustrations show plane cones of equal size, it is, of course, possible to have differently sized plane cones by specifying different min and max values. Note that a min or max plane limit of 0 means that the corresponding plane cone becomes a plane, restricting all movement by the attached body in that direction.

You can see how plane values affect the constrained object's permitted movement space in the following illustrations. As in the previous section, in each example the attached object is free to move within the

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

area represented by the green cone.

Let's first look at what happens when large plane values are provided. As you can see, in this example the cones do not intersect the main cone of movement, and hence the plane values have no effect on the attached body.

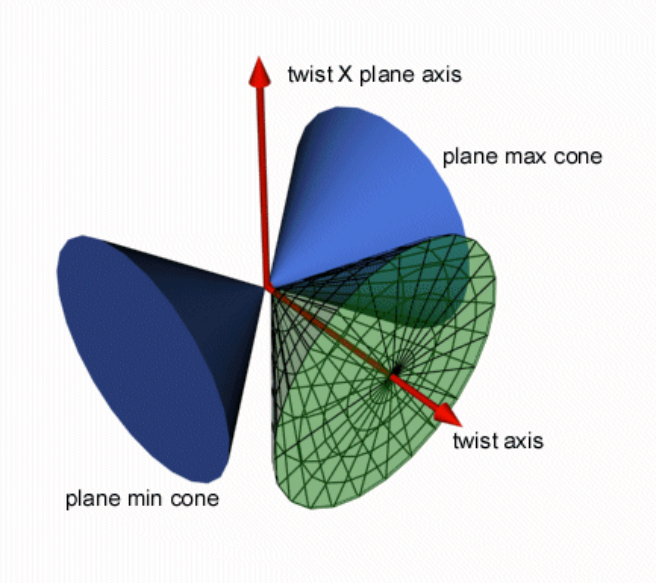


Figure 3.34: Plane cones not intersecting with cone

In the following example, however, lower plane values are specified and the cones intersect the main cone of movement. The subsequent diagrams show the constrained object's available movement space with these plane limits, the green cone having been clipped by the plane cones.

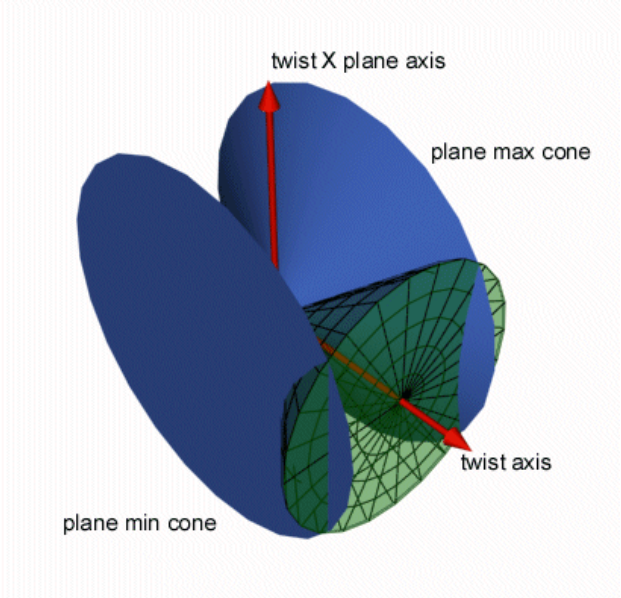


Figure 3.35: Plane cones intersecting with cone

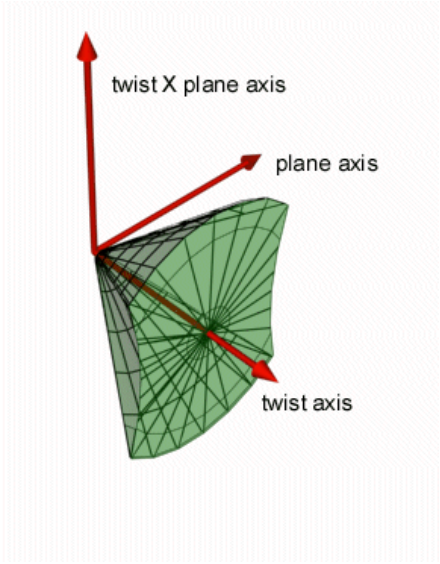


Figure 3.36: Permitted movement space

Creating Ragdoll Constraints

Now that you have been introduced to the ragdoll constraint's axes and limits, let's look at how to create a ragdoll constraint using the Havok SDK. In the following example, you're going to create a shoulder joint. Some suitable limit values are provided, but you can, of course, try your own parameters until you are happy with the movement of the joint.

Creating the shoulder

Now that you have decided how the constraint should move, let's look at how to create the constraint with the Havok SDK. The code snippet at the end of this section shows you one way of doing this.

First you need to create two rigid bodies to act as the torso and arm, or reference body and attached body. Once these are created and positioned in world space, they can be passed to one of the hkpRagdollConstraintData's `setIn...Space()` methods. In this example, you're going to specify the constraint details in body space. Hence, the torso and arm are passed to `setInLocalSpace()` - the first body passed to the method becomes the attached body, the second, the reference body.

You also use this function to specify the following information for each object, expressed in its own local space:

- the constraint pivot point
- the twist axis
- the plane axis

You don't need to specify the twist X plane axis, as it is calculated for you using the twist and plane axes. Note that when specifying the axes in local space, it should be possible to align the local axes. So, for instance, you shouldn't specify a different angle between the twist and plane axes for each object, as you may then get unpredictable behavior from the constraint. In the example, the objects are actually lined up correctly in world space, so the axis information is the same for each body.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Once you have set up the constraint's axes and pivot point, you can then specify the constraint limits using the appropriate methods. Because you want a cone with unequal values, you use the `setAsymmetricConeAngle()` function to set the cone limits after setting up the axes. Finally, as with other constraints, your new ragdoll constraint must be added to the world.

```
hkpRagdollConstraintData* rdc;
    {
        hkReal planeMin =  HK_REAL_PI * -0.05f;
        hkReal planeMax =  HK_REAL_PI * 0.5f;
        hkReal twistMin =  HK_REAL_PI * -0.025f;
        hkReal twistMax =  HK_REAL_PI *  0.025f;
        hkReal coneMin  =  HK_REAL_PI * -0.3f;
        hkReal coneMax = HK_REAL_PI * 0.45f;

        rdc = new hkpRagdollConstraintData();

        rdc->setPlaneMinAngularLimit(planeMin);
        rdc->setPlaneMaxAngularLimit(planeMax);
        rdc->setTwistMinAngularLimit(twistMin);
        rdc->setTwistMaxAngularLimit(twistMax);

        hkVector4 twistAxisA(1.0f, 0.0f, 0.0f);
        hkVector4 planeAxisA(0.0f, 0.0f, 1.0f);
        hkVector4 twistAxisB(1.0f, 0.0f, 0.0f);
        hkVector4 planeAxisB(0.0f, 0.0f, 1.0f);
        pivotA.set(-1.5f, 0.0f, 0.0f);
        pivotB.set(1.5f, 0.0f, 0.0f);
        rdc->setInBodySpace(pivotA, pivotB, planeAxisA, planeAxisB, twistAxisA, twistAxisB);
        rdc->setAsymmetricConeAngle(coneMin, coneMax);

        m_world->createAndAddConstraintInstance( rb1, rb0, rdc )->removeReference();
        rdc->removeReference();
    }
```

Where to use ragdoll constraints

Ragdoll constraints are more expensive to simulate than, for example, hinges, and should be used only where the special angular limits offered by the constraint are necessary. You can often simulate knee and elbow joints adequately using simple hinges, saving the ragdoll constraints for joints with more complex movement such as shoulders and hips.

Prismatic constraints

A prismatic constraint is a constraint between two rigid bodies that allows translation movement along one axis only. Hence, rotations as well as the remaining two translation axes are fixed. You can use it to implement "sliders", i.e. one dynamic rigid body moving along another fixed rigid body in a given direction (the movable axis of the constraint).

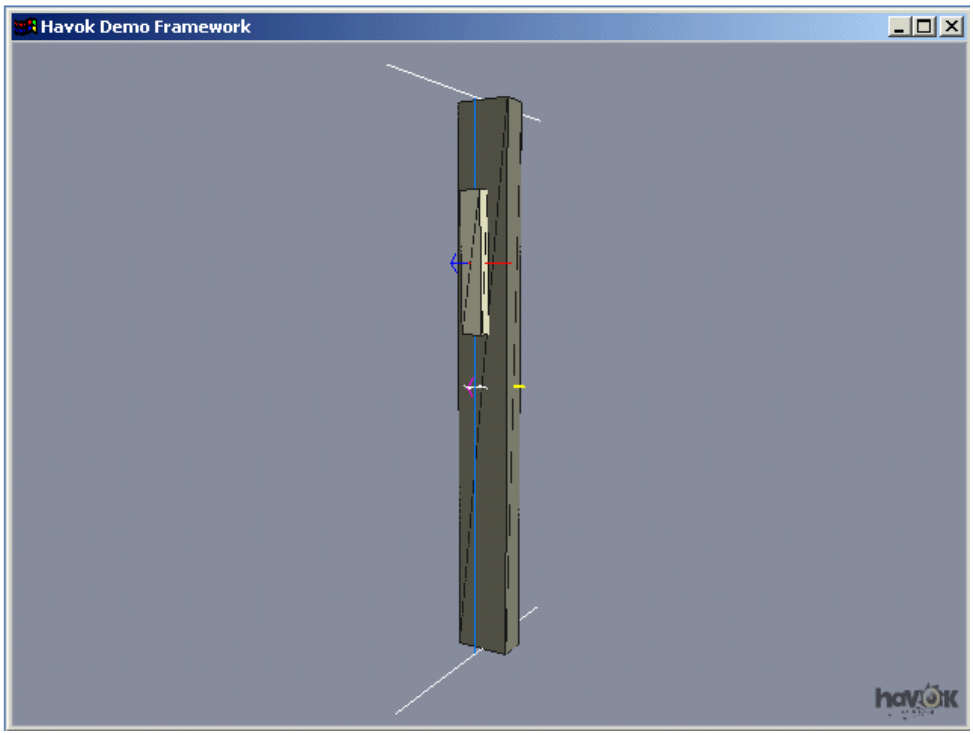
You use hkpPrismaticConstraintData to specify the two rigid bodies and the axis for the direction of movement. As with the other constraints, you can specify this axis in world space or in each body's space, in each case providing a position (pivot point) and a direction for the axis. You can also specify minimum and maximum linear limits and a friction value for the attached body's movement along this axis. By default there are no linear limits and a friction value of zero.

The following code snippet from the Prismatic demo shows you how to set up a prismatic constraint:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
hkpPrismaticConstraintData* pris;
    {
        hkVector4 axis(0.0f, 1.0f, 0.0f);

        // Create constraint
        pris = new hkpPrismaticConstraintData();
        pris->setInWorldSpace(rb1->getTransform(), rb0->getTransform(), position, axis);
        pris->setMaxLinearLimit(10.0f);
        pris->setMinLinearLimit(-10.0f);
        m_world->createAndAddConstraintInstance( rb1, rb0, pris )->removeReference();
        pris->removeReference();
    }
```

The following illustration shows the prismatic constraint created in the example. The small box is free to move along the blue axis within the limits indicated by the white lines.



Point to Path Constraints

The point to path constraint allows you to constrain two bodies so that one is free to move along a specified path relative to the other. You could, for example, use a point to path constraint to create a ski chairlift, where a chair is conveyed along suspended from a cable.

When creating a point to path constraint, you must provide two rigid bodies as well as the path the first body should follow. You specify the path using an hkpParametricCurve. hkpLinearParametricCurve is an implementation of hkpParametricCurve that represents a piecewise linear curve. It has a number of useful methods that you can use to set up your path, including the `addPoint()` method that allows you to add a point to the path.

The point to path constraint allows you to specify an orientation type for the attached body's movement.

The possible orientation types are:

- HK_DO_NOT_CONSTRAIN_ORIENTATION, where the attached body's orientation relative to the path is not restricted by the constraint. This is the default orientation type.
- HK_CONSTRAIN_ORIENTATION_ALLOW_SPIN, where the attached body is allowed to spin around the path that it's traveling along, but cannot change its orientation in any other way.
- HK_CONSTRAIN_ORIENTATION_TO_PATH, where the attached body's orientation is constrained to the path.

Note that for the HK_CONSTRAIN_ORIENTATION_TO_PATH orientation type to work, the hkpParametricCurve must have its `getBinormal()` function implemented. This should return a normalized vector that defines the "up" direction for the path. A very naive implementation is provided in hkpLinearParametricCurve - you will need to implement it yourself if you create your own path class.

You can also provide a friction value for the body's movement along the path. By default, this friction limit has a value of zero.

The following examples from the Point To Path demo show you how to set up a point to path constraint. In this demo, the attached body is only free to move along the specified helix curve.

First, let's look at how you create the path's curve. As you can see, the hkpLinearParametricCurve's `addPoint()` method is used to build up the helix.

```
m_helix = new hkpLinearParametricCurve();
    const hkReal twists = 3.0f;
    const hkReal height = 15.0f;
    const hkReal radius = 10.0f;
    hkVector4 pt;
    for (hkReal t = 1.0f; t > 0.0f; t -= 1.0f / 300.0f)
    {
        const hkReal angle = t * HK_REAL_PI * 2.0f * twists;
        pt = hkVector4(hkMath::cos( angle ) * radius * ( 1.0f + t), t * height, hkMath::sin( angle ) *
          radius * (1.0f + t) );
        m_helix->addPoint( pt );
    }
```

Once you have created the rigid bodies and path, you use the hkPointToPathData's `setInWorldSpace()` function to specify the attached body, the reference body, the pivot point in world space (which specifies the position of the path), and the path curve for the constraint. If you use `setInBodySpace()`, you can set the pivot in each body's space instead. This example also shows setting an orientation type and friction value.

```
hkpPointToPathConstraintData* p2Path;
    {
        p2Path = new hkpPointToPathConstraintData();

        // constrain angular DOFs so that it can only spin around the path
        p2Path->setOrientationType(hkPointToPathData::HK_CONSTRAIN_ORIENTATION_ALLOW_SPIN);

        p2Path->setInWorldSpace(rb1->getTransform(), rb0->getTransform(), rb1->getPosition(), m_helix);
        p2Path->setMaxFrictionForce( .1f );


        m_world->createAndAddConstraintInstance( rb1, rb0, p2Path )->removeReference();
        p2Path->removeReference();
    }
```

Breakable constraints

A breakable constraint (as the name suggests!) is a constraint that breaks when a specified threshold is exceeded - for example, you could use a breakable constraint to make a door that breaks off its hinges if it's kicked hard enough. An hkpBreakableConstraintData is a wrapper around one of the other constraint types that allows you to make the constraint breakable.

The following example from the Breakable Constraint demo shows how to create a breakable hinge. As you can see, you first create a hinge as usual. However, rather than adding the hinge to the world directly, you pass both the hinge and the world to the constructor of a new hkpBreakableConstraintData. Then you specify the threshold at which the constraint will break using its `setThreshold()` function.

As the example below shows, breakable constraints also allow you to specify whether you want the breakable constraint to be removed from the world when it breaks. You do this using its `setRemoveWhenBroken()` function.

Finally, once you have set up the breakable constraint, you add it to the world.

```
// set pivot
    hkVector4 pivot  ( 0.5f, -0.6f, 0.5f );
    hkVector4 axleDir( 0.0f,  0.0f, 1.0f );

    // Create hinge constraint
    // Do not add it to the world, only add the "wrapped" breakable version below
    hkpHingeConstraintData*  axle = new hkpHingeConstraintData();
    axle->setInWorldSpace(rb0->getTransform(), rb1->getTransform(), pivot, axleDir );

    // Create breakable constraint wrapper
    hkpBreakableConstraintData* breaker = new hkpBreakableConstraintData( axle , m_world );
    breaker->setThreshold( 10 );
    breaker->setRemoveWhenBroken(false);

    // Remove reference to "axle" since the BreakableConstraint now owns it.
    axle->removeReference();

    hkpConstraintInstance* constraint = new hkpConstraintInstance(rb0, rb1, breaker);

    // Add the breakable constraint to the world
    m_world->addConstraint( constraint );

    constraint->removeReference();
    breaker->removeReference();
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

When you play the Breakable Constraint demo, the hinged blocks remain hinged until a falling third block strikes them. The force of this collision exceeds the breakable threshold and the hinged blocks are separated.

Breakable listeners

If you want to be notified when a breakable constraint breaks, you can create an hkpBreakableListener class and implement its callback function, `constraintBrokenCallback()`. This is called by the simulation each time a constraint breaks, passing the listener a reference to the constraint, the force that broke it, and whether the breakable constraint has been removed from the simulation.

Malleable constraints

You can make a constraint softer or harder by making it into a malleable constraint. This changes the virtual mass used by the system when solving the constraint. This is useful, for instance, if you want to create a bouncy spring using the hkpStiffSpringConstraintData.

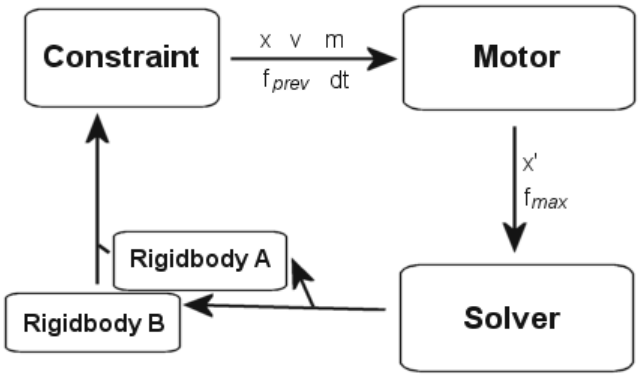
You make a malleable constraint by wrapping an existing constraint in an hkpMalleableConstraintData, in the same way as you create a breakable constraint. Then you can use the `setStrength()` function before adding the malleable constraint to the world.

Enabling motors in contraints

Havok also provides "powered" versions of some of its constraints - namely the hkpLimitedHinge-, hkpRagdoll-, and hkpPrismatic constraint. Each degree of freedom for these constraints can be controlled by a chosen motor. The motor implements a force law to apply calculated forces to that degree of freedom. These forces work to move that degree of freedom from its current position to a desired target position. For example, you can model powered machinery or muscles using motors.

Motors

Motors can be added to hkpLimitedHinge-, hkpRagdoll-, and hkpPrismatic constraints and also to constraints created with the constraint construction kit. You need to explicitly add your desired motors to the powered constraints.



The diagram above shows the basic feedback loop for the motors. The state variables passed in to the motor are in 1D "motor space", they are the dynamic values of the bodies converted into the one

dimension that the motor works on. The constraint is responsible for doing this conversion and calling the motors with this data. Position (x) and velocity (v) are relative between the two bodies connected by the motor. You can just treat the input and output values as if they described a particle on a line. The mass (m) is the effective mass that is "felt" by the motor, i.e. if you apply 2 units of force and the mass is 1 unit with a delta time of 1 unit, you should get a change in constraint space velocity of 2 units. The motor does not apply forces directly, it tells the solver what it would like done and the solver applies the forces to the rigid-bodies. The outputs from the motor (the inputs to the solver) are the new motor space position you are trying to reach and a maximum force that the solver can apply to get there.

You convert a simple applied force to the motor outputs by setting the maximum force (f_max) equal to the applied force, and the target position (x') equal to that force multiplied by the timestep squared and divided by the mass. Setting the target position follows from the basic laws of physics. f = ma, a = dv/dt, a*dt = v = dx/dt, (f/m)*dt*dt = x. This slightly more complicated interface to the outputs is provided to allow the motors maximum control over the stability and behavior of the system.

A number of motors are provided for you. The `hkpPositionConstraintMotor` is a good motor for blending keyframed animation targets with an articulated figure. It uses an implicit spring-damper model to try calculate the target for the current step, then allows the solver to act as a stiff driven constraint to get to that target. You may have a very specialized game-specific force law in mind for you motors. i.e. a motor that gets stronger as the distance from the target increases, up to a point, then becomes weaker past that point. To writing your own motor you should use the `hkpCallbackConstraintMotor` and provide a custom `CalcMotorDataCallbackFunc` callback function. Lets look at implementing a simple spring. A spring gets stronger linearly as the current position gets farther from the rest length of the spring.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
// Create a callback function
void HK_CALL SampleCallbackFunc(const hkpCallbackConstraintMotor& motor, const hkpConstraintMotorInput*
  input, hkpConstraintMotorOutput* output)
{
    // We're not interested in achieving a specific velocity
    output->m_targetVelocity = 0.0f;

    // We compute relative targetPosition for the motor i.e. our spring's rest point.
    // This is computed directly from the given input.
    output->m_targetPosition = input->m_deltaTarget + input->m_positionError;

    // These are desired spring constant and dumping.
    // Those values need to be converted into simple tau and damping parameters used by the solver.
    hkReal springConstant = 100.0f;
    hkReal springDamping  = 5.0f;

    hkReal invMass = 1.0f / input->m_virtualMass;
    hkReal dt = input->m_stepInfo->m_substepDeltaTime;

    // Forces applied by the solver are proportional to tau and virtual mass of involved bodies.
    // Spring force is independent of the bodies' masses therefore we divide tau by their virtual mass.
    // To convert tau to solver space we multiply it by dt^2
    hkReal tau = springConstant * dt * dt * invMass ;
    output->m_tau = hkMath::clamp( tau, 0.0f, 1.0f );

    // Solver damping is computed analogically to tau.
    // To convert damping to solver space we multiply it by dt
    hkReal damp = springDamping * dt * invMass;
    output->m_damping = hkMath::clamp( damp, 0.0f, 1.0f );

    // Explicitly specify minimum and maximum forces that the motor can apply.
    output->m_maxForce = motor.m_maxForce;
    output->m_minForce = motor.m_minForce;
}


// Create a callback motor which uses the above function.
// This motor can be then used to power a constraint.
hkpCallbackConstraintMotor* ccm = new hkpCallbackConstraintMotor( hkpCallbackConstraintMotor::
  CALLBACK_MOTOR_TYPE_HAVOK_DEMO_SPRING_DAMPER, SampleCallbackFunc );
```

Powered mode of Hinge and Ragdoll constraints

Motors of a ragdoll constraint can move the constraint's attached body to a specified target position, set using the constraint's `setTargetFromRelativeFrame()` or setMotorTargetAngle() function. You specify the target as a relative orientation between the attached and reference bodies. The constraint will then use three motors to try to achieve this desired pose target.

The ragdoll constraint's motors work differently than the motors for the rest of the constraints. Ultimately we are trying to match two orientations using the motors, rather than directly powering a degree of freedom to a target with a motor. The current position passed in will be the distance from the zero point of the motor. So all you need to do with the powered rag doll is set its orientation target - you don't need to access the motors directly.

The three motors apply torques around the twist, cone and plane axis. You can set the twist, cone and plane axis motors to be separate motors or they can all point to an instance of the same motor. Motors can be shared between axis and between different constraints. You must however take care that you don't share motors if you have implemented your own motors which contain non-shareable data.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.2.7.3 Tuning and tweaking constraints

Solving constraints

Havok deals with each constraint you add to the simulation as part of an overall constraint system. As a result, constraints are relatively stable and stiff, although they can get slightly softer with a lower simulation frequency or differences in mass. A constraint system can include contact constraints - which are used internally by Havok to prevent objects from interpenetrating - as well as the constraint types described in this document.

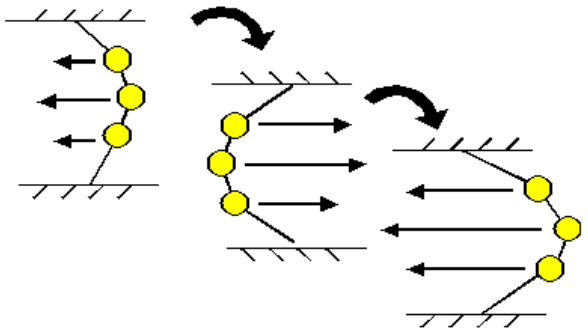
The overlap information for any pair of objects provided by Havok's collision detection system is an error in satisfying their contact constraint. Similarly, if, for instance, a pair of joined objects become separated, the measure by which they are separated is an error in satisfying their ball-and-socket constraint. Errors in satisfying constraints can be counteracted by moving the objects to a new position. This must be done carefully, however, because moving an object to deal with one error can introduce another error elsewhere in the system.

This problem is dealt with by the constraint solver. The solver is responsible for calculating the necessary changes to reduce error for all constraints in the simulation in each simulation step. It passes this information on to the integrator, which uses this information to calculate the new motion state (position, orientation, velocity, acceleration, and so on) for each object in the simulation at the end of the step. The new positions can then be passed on to, for example, your renderer to update the game display.

Reasons for instability of constraints

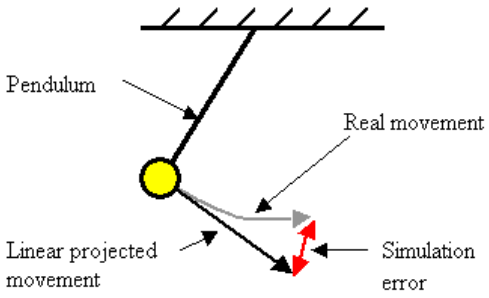
Simulating constraints is a hard problem to solve. For practical reasons a constraint solver has certain stability limitations:

- The forces involved in solving a constraint system can be too high and lead to frequency of object movements higher than the simulation frequency. For example, imagine simulating a swinging short rope that gets pulled harder and harder. In real life you will hear a higher and higher pitched sound, in a physical simulation with 30 physical steps per second pulling the simulated rope leads to strange artefacts like instability or soft constraints.



Swinging exploding rope (with three balls)

- The constraint solver solves a set of linear equations. In the physical simulation, however, angular movement leads to non-linear behavior. So if the time step is relatively big compared to the angular velocities of your object, the error introduced may lead to instability.



Getting stable constraints

If you want to make your constraints more stable, try following these simple rules. Start with the top rule and only if you do not get your expected results, try the next.

- Avoid low rotation inertia because low inertia means high angular velocity, which leads to high simulation errors (See Reasons for Instability of Constraints II). In particular, long and thin objects have one axis with a very low inertia. Try to artificially increase the inertia around this axis or use the oriented particle model. Very small objects also have low rotational inertia.

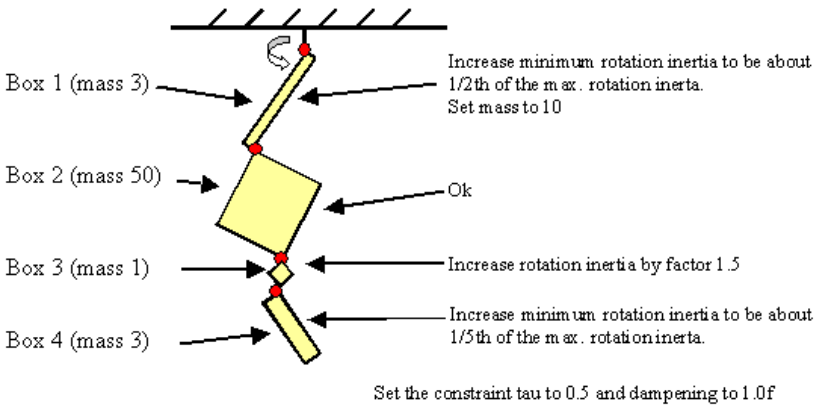
  The following rules are useful here:

  - In a multi body system, the minimum rotation inertia of one object should not be less then 1/10th of the maximum rotation inertia of the same object.
  - If a thin object is constrained between two heavy objects, the minimum rotation inertia of any object should not be less then 1/4 of the maximum rotation inertia of the same object

- If your constraints appear to be very soft, try to avoid constraining light objects between two heavy objects, or between a fixed object and a heavy object. Just make the light object at least as heavy as a 1/10th of the mass of the heavier objects.
- If you constrain a very large number of objects together, you may need to decrease the tau value of all constraints involved. Use the following rule: tau $\sim$ 1.0/sqrt(number of constraints in a system).
- Decrease the damping of the constraints slightly (to 0.6f).
- Increase simulation frequency by increasing the number of substeps.
- Increase solver tau and damping parameters. (See hkpWorldCinfo::SolverType)
- Contact Havok support ;)

Note:
These rules are rather strict. If you care more about physical accuracy, then you can either experiment with your system to find the real limits, or, if possible, just increase the number of substeps.

The following example shows a chain of objects with some hints to get a very stable solution:



CPU requirements

All constraints in Havok are O(n) constraints. This means that 100 constraints need 10 times more CPU than 10 constraints. As a result, there is no worse case CPU requirement if many objects are constrained together. There are a few simple rules about the CPU costs of constraints:

- In principle, the more degrees of freedom a constraint restricts, the more expensive it gets. A hinge with one degree of freedom restricts the remaining 5 degrees and is more expensive than a point-to-point constraint.

Constraint Limitations

In general, the Havok Vehicle Kit is a better way to construct four wheeled vehicles than using wheel constraints, especially racing or fast moving vehicles. Take a look at the Vehicle Kit chapters for more information.

Increasing Inertia Tensors for Ragdoll (e.g. hkpInertiaTensorComputer::optimizeInertiasOfConstraintTree)

Simulating a dynamics system of only linear equations is pretty simple. However, many of the problems when simulating dynamic objects are due to angular effects. A simple trick to reduce angular effects is to increase the inertia of the rigid bodies.

Examples:

- A rope of bodies blows up
- A ragdoll jitters a lot
- The ragdoll limits are too soft
- A ragdoll explodes
- An object causes too many TOI events
- Ragdoll motors are too soft

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

All of these problematic situations can be ameliorated by increasing the inertia of the bodies concerned.

So we want to:

- increase inertia to make things more stable but...
- ...without changing the behavior to the extent that a user can see the inertia tensor increase (e.g. extremely slow rotating objects)

Most of our use cases include constrained rigid bodies. Such constrained bodies cannot move entirely freely. As a result the virtual inertia tensor of each rigid body is much higher than its real inertia. Example: Say we have a human arm: .6m*.1m*.1m (using a box) with a real intertia diagonal of (0.0066, 0.12, 0.12). That means the inertia around the X-axis is 18 times smaller than the other inertia elements. Now consider two of those arms - rotate one by 90 degrees and glue them together (fixed constraint) at their center of mass. The resulting inertia tensor diagonal now looks like (.1266, .1266, .24) (both inertias added). If we modify the original inertia of one arm by lets say 1000% (which seems a lot) from .0066 to 0.066, the combined inertia will only go up to (.186,186,24). In short: Changing the inertias of individual objects (especially the smaller ones) in a constrained system only has a minor effect on the behavior seen by the user, but makes motors and limits much better/stiffer.

The hkpInertiaTensorComputer::optimizeInertiasOfConstraintTree() method tries to increase the inertia of the inner bones of a ragdoll as much as possible to improve motor, limit and friction stability, but not so much that the combined ragdoll inertia tensor increases significantly. As a result the motors should handle much better, however the behavior of the ragdoll as a whole will not look strange to the user.

3.2.8 Constraint Chains

3.2.8.1 Introduction

Constraint chains are useful for simulating long chains of linked bodies, which would or might fail to be properly simulated by Havok's iterative solver.

The advantages are

- More solid ragdolls
  - Characters limbs form chains of bodies of varied mass. In such cases, a low mass bone in the middle of the chain (e.g. a shoulder plate) will largely reduce accuracy for other heavy bones located further away from the torso. As the effect ragdolls' limbs might stretch and separate when ragdolls are thrown against obstacles, or dragged forcefully.
- Better motor accuracy for powered ragdolls
  - When powering a ragdoll with animation information, Havok's iterative solver could not deliver an accurate match to the requested pose. Creating the ragdoll with powered chain constraints gives greatly improved results.
- Much better results when simulating of ropes
  - Ropes can be simulated with a large number of bodies, i.e. more than 20 and up to a few hundred.

3.2.8.2 How to use chains

Chains are treated by the Havok engine almost identically to all the other constraints; they are slightly more complex to setup however. We now describe the setup, however sample code can be found in the demos listed in the demos section below.

To setup a constraint chain you need to initialize a hkpConstraintChainData and hkpConstraintChainInstance object, analogously to how it is done for normal constraints. You then simply add the instance to the world calling myWorld->addConstraint(myChainInstance).

The hkpConstraintChainData holds a list of constraint infos specific to the type of the chain. Each constraint info holds data analogous to data held by a regular Havok's constraint data, e.g. values such as pivot points. hkpConstraintInstance holds a list of linked entities. There is a one-to-one match between the ConstraintInfos and the list of entities.

Populate the data's list calling chainData->addConstraintInfoInBodySpace(..) and the instance's list calling chainInstance->addEntity(..).

Havok supports three types of chain data:

- hkpStiffSpringChainData
  - This is the simplest and most effective chain in terms of CPU usage. It is however also much less stable than the hkpBallSocketChainData described below. One may consider using this chain in conjunction with 'stabilized' motions for the bodies in the chain, or with objects with unphysically increased inertia tensor values.
- hkpBallSocketChainData
  - This is the preferred constraint allowing for quick and easy setup of ropes. This is also the preferred chain to use for non-powered ragdolls. As chains do not support constraint limits however, you will need to add additional limit constraints on top of the chains. There are two stripped-down constraints available for that purpose: hkpHingeLimitsData, and hkpRagdollLimitsData.
- hkpPoweredChainData
  - This is the chain to use when you want to create complex machines, crane arms, or ragdolls. Note that when adding chains to a ragdoll you can overlap the chains. You just need to ensure that the overlapping chains to not apply contradicting constraints to the linked bodies. Those chains can also be driven by Havok's animation data.

3.2.8.3 Interface to the chains

The constraint infos of hkpStiffSpringChainData and hkpBallSocketChainData contain only pivot points and - in the former one - spring length.

To setup constraint infos in hkpPoweredChainData you additionally need to specify:

- target orientation of the linked bodies (hkQuaternion m_aTc)
  - This specifies the relative orientation of the second (child) body in the first (parent) body's space. (The second body is the one added to the chain instance's list later.)

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- three motors to be used
  - The motors used to enforce the 3-dimensional orientation constraint
  - The motors available for use with the hkpPoweredChainData are hkpPositionConstraintMotors and hkpVelocityConstraintMotors.

Also you may make use of the additional parameters (however you should not use them before getting a good understanding of the above)

- hkBool m_switchBodies
  - This allows you to reverse the parent-child relation. When this flag is set to true, the m_aTc represents the first body's target orientation in the second body's space.
- hkQuaternion m_bTc
  - This allows you to specify a reference frame for the child body. The motor's space is aligned to that reference frame. This is useful when you want to vary motors' strengths or disable one of them and allow rotation along a specific axis. (See the "Motor Chain Demo/Test constraints space alignment" demo)

All the hkConstraintChainDatas have their own m_tau, m_damping, and m_cfm settings, independent of the global hkpWorld's solver settings.

Note:
You might note that hkpConstraintChainInstance also references a hkpConstraintChainInstanceAction. This is an internal action and it is added to the hkpWorld together with a hkpConstraintInstance. The action's purpose is to allow for the hkpConstraintChainInstance to keep track of its entities - that's because the hkpConstraintInstance's interface only can reference two bodies. This action ensures that all the entities will be treated as a group by Havok and always be simulated in the same simulation island.

3.2.8.4 Limitations of the Technology

- The powered chains do not support limits yet. However, you can use the hkpHingeLimitsData, and hkpRagdollLimitsData to use limits with chains.
- You cannot remove or disable a constraint in a chain easily.
- The chains are not handled during TOI events. As a workaround you can create normal constraints on top of the chain constraints and flag those as TOI constraints.
- The chain is solved perfectly, however it still works in an iterative solver. That means if any body in a chain is constraint by a non chain constraint, this introduces softness at that body. Based on the random order of constraint solving, this softness can either appear in the chain or at the other constraint.
  - A: a nice stable stretching between each pair or
  - B: if you pull harder a unstable stretching, where some bodies jitter between several positions.
  - The chain constraints solve A perfectly, but they still suffer from B. And because the forces are much higher, B will be more likely. To reduce this effect: INCREASE INERTIA (and mass) of the jittering bodies.

- Powered Chain constraints can be slow. Most of the time, chains are really fast. However, if you have a powered chain, where many motors have their maximum force limits breached, these chains get slow. Basically for each breached motor, we have to recalculate the whole LU decomposition of the chain again. So if you have 200 bodies and 100 motors are breached, you have to do 200*100*n matrix6 multiplies.
  - Workaround: If your motor is very weak and you expect many max forces to be breached, do not use powered chains, but ball socket chains with a normal powered ragdoll constraint on top.

3.2.8.5 What demos to look at

Demos using the constraint chains technology:

Physics / Api / Constraints / Chain / Ball And Socket Rope. This demo shows a rope with weights at each end that you can drag around and crash into obstacles.

Physics / Api / Constraints / Chain directory:

- SwingingRope
  - Ball and Socket chains variant
    - Chains of 100 bodies and more are stable
    - The demo also displays a rope solved with the old Havok constraints for comparison
  - Stiff spring chains variant
    - Note that these ropes are much less stable then their ball-and-socket counterparts
- Bridge
  - A bridge of a hundred or so elements constructed with two ball-and-socket chains.
- ChainNet
  - Show demos imitating cloth. They use array of bodies interconnected with a number of chains constructed at 90 degree angles.
- MotorChain
  - Illustrates accuracy of motors in the chain. You can modify the strength of all the motors and drag the boxes with the mouse. To get a better feeling for the behavior.
  - Velocity control
  - Enforce hinge
  - Test constraints space alignment
- Spider
  - Uses the chains to power the spider legs.

Animation / Api / Ragdoll / Blendtest

- This shows a powered ragdoll feed with an animation. Old Havok constraints hardly follow the requested animation. The new chains yield a very accurate match. Note how the characteristics of the movement changes as you control the maxForces of the ragdoll's motors. There's also a variant which illustrates how to use ragdoll and hinge limit constraints on top of the constraint chains.

3.2.8.6 Performance Issues

The computational complexity of the chains is linear in respect to the number of connected bodies. hkpPoweredChainData is several times more expensive than hkpBallSocketChainData, and hkpBallSocketChainData is several times more expensive than hkpStiffSpringChainData. Use of large chains in large simulation islands may slow down the solver due to memory caching issues. Chains have a considerable memory overhead and they store their intermediate results together with other solver data. When using hkpPoweredChainData be aware that their performance will drop noticeably when motors' forces are breached.

3.2.8.7 Powered Chain Mapper

The hkpPoweredChainMapper facilitates manipulation of systems of bodies linked by a number of hkPoweredChain instances. The mapper facilitates modification of target orientations for motors, which is useful when feeding the constraint chains with animation data. It also allows you to set maximum forces of the motors during simulation, which allows you to influence the characteristics of interaction with the environment.

Setup

The mapper is intended to be initialized from a previously setup physical system consisting of rigid bodies and normal Havok constraints (which connect exactly two entities, each), e.g. a ragdoll.

```
// Set options for the mapper building function
hkpPoweredChainMapper::Config config;
config.m_createLimitConstraints = true;
config.m_cloneMotors = true;

// Supply all the chains that we want to use
hkArray<hkpConstraintInstance*> allConstraints = theConstraintsThatWeWantToUse;

// Specify a list of start-end point pairs
hkArray<hkpPoweredChainMapper::ChainEndpoints>& pairs;
hkpPoweredChainMapper::ChainEndpoint& pair = pairs.expandOne();
pair.m_start = someEntityStart;
pair.m_end = someEntityEnd;

// Optionally you can pass an array which will be filled with constraints that were not used to construct
  chains
hkArray<hkpConstraintInstance*>* unusedConstraints;

// Create the mapper
hkpPoweredChainMapper* mapper = hkpPoweredChainMapper::buildChainMapper(config, allConstraints, pairs,
  unusedConstraints);

// Use the utility to add all the mapper's consraints to the world
hkpPoweredChainMapperUtil::addToWorld(world, mapper);
```

To create a mapper you need to specify an initial set of normal constraints which simply tell the mapper creation funciton which bodies can be linked in a sequnce and a list of pairs of hkpEntities, which specify the start and end points for each chain created. All the constraints which are in the created chains must be of the 'powered' type, i.e. have motors which can be extracted by the hkpConstraintUtils::getConstraintMotors functions.Also All the specified start-end entity pairs must be valid.

Note:
If there's more than one constraint connecting one pair of bodies, than it is unclear which constraint's info will be used when creating the chain.

Usage

As the chains may overlap, a single link index of the mapper may refer to more than one links in a number of chains. Setting a properly of one link modifies information in all the related chains at once. The index-to-link mapping is based on the ordering of the original constraints in the list passed to the mapper-building function. The Nth link connects the same bodies as the Nth constraint in that origina list.

```
// To weaken a constraint you have to set all it's three motor's forces to new values,
// as you can assign a separate motor to each dimension. If you use the same motor for all three
// dimensions then there is no need for the loop
const hkReal newStrength;
for (int i = 0; i < 3; i++ )
{
    mapper->setForceLimits(linkIndex, i, -newStrength, newStrength);
}

// And to set a new target orienation
hkQuaternion requestedOrienation = hkQuaternion::getIdentity();
mapper->setTargetOrientation(linkIndex, requestedOrientation);
```

3.2.9 Actions

Actions provide a simple and effective mechanism to control the **behavior** of the physical world during a Havok simulation. They can range from simple e.g. an "anti-gravity" action to make an object float, to complex e.g. an action to control the flight of a helicopter.

In addition to reading this section, take time to look at and play with the actions used in the dynamics/actions demos and the Pyramid example, as well as the example actions provided in `hkutilities/actions`.

3.2.9.1 About Actions

hkpAction::applyAction()

hkpAction is an abstract class so every derivation of hkpAction, no matter how simple or complex, has an `applyAction()` function that performs the "work" of the action. Once an hkpAction is added to the Havok world, `applyAction()` is automatically called during every simulation step. Find out more about how actions take part in the simulation in the Dynamics chapter.

Note:
Each hkpAction is automatically applied so long as its bodies are active. If bodies becomes inactive, the action is not applied. If the bodies reactivate, the action once again starts applying to the bodies.

The `applyAction()` function is passed an hkStepInfo data structure by the simulation each time it is called. hkStepInfo stores the simulation's current delta time and inverse delta time, and can be queried

by the action. For example, in the Unary Action demo the force applied by hkAntiGravityAction varies depending on the simulation time. hkAntiGravityAction gets the time from the hkStepInfo, as in the sample below:

```
inline void AntiGravityAction::applyAction( const hkStepInfo& stepInfo )
    {
        // Apply the antigravity force to the body.
        // The force ranges from [-2 * gravity to 0], modulated by a sine wave as a function of time
        // so sometimes it is stronger than gravity, sometimes less strong.
        m_currentTime += stepInfo.m_deltaTime;

        // Scaling factor that varies over time.
        hkReal scale = 1.0f + hkMath::cos(2 * m_currentTime);

        // Vector representing force to be applied.
        hkVector4 force;

        // Apply the antigravity force in the opposite direction to gravity. Its strength is
        // determined by multiplying m_gravity (also a vector) by scale.
        force.setMul4( -1 * scale, m_gravity);

        // This function call actually applies the force to the rigidBody.
        m_body->applyForce( force );
    }
```

hkpAction types

All actions implement the hkpAction interface. The Havok SDK provides three classes derived from hkpAction that can be used when creating custom actions. They are:

- hkpUnaryAction

  For actions that apply to a single rigid body. The motor action (hkpMotorAction) described later in this chapter, which can be used to spin a rigid body around an axis, is an example of a unary action.

- hkpBinaryAction

  For actions that apply to pairs of rigid bodies. The spring action (hkpSpringAction) described later in this chapter, which can be used to attach two rigid bodies together, is an example of a binary action.

- hkpArrayAction

  For actions that apply to many rigid bodies. A magnet could be represented by an hkpArrayAction. Care must be taken to use hkArrayActions only when appropriate. See Multiple actions versus single actions below.

These derived classes provide basic functionality for actions and have data members for the rigid bodies each action applies to. hkpAction also has a hkpEntityListener-inspired virtual function, `entityRemovedCallback(hkpEntity* entity)`. In all three implementations, `entityRemovedCallback()` removes the action from the simulation if any of the rigid bodies it applies to are removed.

Note that an hkpArrayAction is removed from the simulation if *any* of its rigid bodies are removed - to avoid this, re-implement `entityRemovedCallback()` for hkpArrayAction, or derive directly from hkpAction.

Adding and removing actions

To add an hkpAction to the simulation, use `hkpWorld::addAction(hkpAction* action)` e.g.

```
// Add antiGravityAction to the world. From now on antiGravityAction will automatically
// be applied to its body (boxRigidBody) during integration so long as boxRigidBody is
// active. If boxRigidBody becomes inactive, antiGravityAction is not applied. If
// boxRigidBody reactivates, antiGravityAction again starts applying to it.
// antiGravityAction can also be removed using hkpWorld::removeAction(antiGravityAction)
// after which it is no longer included in simulation.

// Add the action.
m_world->addAction( antiGravityAction );

// After addAction() m_world references antiGravityAction so the local reference can
// safely be removed.
antiGravityAction->removeReference();
```

hkpAction derives from hkReferencedObject. hkpWorld objects add a reference to each hkpAction in `addAction()` so it is safe to call `removeReference()` on the local hkpAction pointer after addition to the world.

To remove an action from the simulation, use `hkpWorld::removeAction(hkpAction* action)` e.g.

```
m_world->removeAction( antiGravityAction );
```

hkpWorld calls `removeReference()` on hkpAction in `removeAction()`. If the action's reference count reaches zero at this point the hkpAction will be deleted as normal. Find out more about Havok reference counting in the Havok Base Library chapter.

hkpAction::getEntities()

hkpAction has a pure virtual `getEntities()` function. For classes derived from hkpAction, `getEntities()` is used to retrieve the bodies the hkpAction applies to. When an hkpAction is added to an hkpWorld, the hkpWorld queries the first body the hkpAction applies to, to determine what simulation island the body is in and then adds the hkpAction to the same island. All bodies an hkpAction applies to are in the same simulation island. Find out more about simulation islands in the Introduction and Dynamics chapters.

Implementations of `getEntities()` are provided in the derived classes, hkpUnaryAction, hkpBinaryAction, and hkpArrayAction. In all three implementations all the bodies each action applies to are returned, be it one, two or an entire array of bodies. The following code sample shows hkpUnaryAction's implementation of `getEntities()`.

```
void hkpUnaryAction::getEntities( hkArray<hkpEntity*>& entitiesOut )
{
    entitiesOut.pushBack( m_body );
}
```

Multiple actions versus single actions

It is perfectly acceptable to have multiple actions apply to an entity. It is also fine for an entity to have an action with multiple functionality but this does not give an appreciable performance gain.

Note:

An action that applies to multiple entities merges the entities' simulation islands. This may result in a performance loss if some of the merged simulation islands would otherwise be inactive e.g. an action for wind or anti-gravity that acts on all the entities in a world will merge all the simulation islands. A separate action per entity is definitely a good idea in this case!

Actions versus Constraints

Often it is not clear whether a game feature should be better represented by an hkpAction or an hkConstraint. In general, actions are faster to simulate than constraints, but less stable e.g. the stability of dashpots is more sensitive to simulation frequency (large timesteps can make simulation unstable). Find out more about constraints in the Constraints chapter.

Just like constraints though, actions with more than one body are an association between those bodies and so affect the partitioning of simulation islands. As such you can not change the bodies in an action once the action has been added to the world. If you want to change the bodies, query the action for its properties and recreate another action from that information.

3.2.9.2 Example actions

In addition to hkpUnaryAction, hkpBinaryAction and hkpArrayAction, the Havok SDK provides a number of ready-made actions. These actions are provided with full source code, and are useful examples of what actions can do. These action classes can be found in the `Physics/Utilities/Actions` directory of your installation.

Dashpots

Linear dashpot

A linear dashpot (hkpDashpotAction) is a simple action that is used to attach two rigid bodies together. Its `applyAction()` function applies forces to the rigid bodies in an attempt to keep the bodies' attachment points in the same position. If other strong forces or impulses are applied to the rigid bodies the dashpot may momentarily "break" but it will attempt to force the bodies back to their specified positions in the next simulation step.

The following `hkpDashpotAction` properties can be specified:

- **Rigid bodies**

  The two rigid bodies the dashpot will act on.

- **Attachment points**

  The attachment points for each rigid body. The points can be specified in either world space or the rigid bodies' local space, and do not need to be inside the volume of either rigid body.

- **Strength**

  The strength of the restoring force between the two attachment points. A good initial range is 0.5 - 1 times the mass of the attached rigid bodies. The default strength value for both linear and angular dashpots is 1.0.

- **Damping**

  This value affects how quickly the oscillation of the dashpot stabilizes. It governs the impulse applied to the connected bodies due to their relative velocities. Low values create springy behavior, while high values reduce any oscillations very quickly with the size of the oscillations getting much smaller each timestep. A good initial value is 0.1 of the strength. The default damping value for linear and angular dashpots, based on the default strength, is 0.1.

For similar behavior to a linear dashpot but with greater stability, use a ball and socket constraint (hkpBallAndSocketConstraintData).

Angular dashpot

An angular dashpot (hkpAngularDashpotAction) is the rotational equivalent of a linear dashpot. An angular dashpot tries to align two rigid bodies so that they have the same orientation, with an optional offset. The dashpot's angular offset value is a quaternion defining the orientation of the first body relative to the second. In its `applyAction()` function, the angular dashpot works to maintain this offset between the two bodies. By default, this is the identity rotation.

As with hkpDashpotAction, the dashpot pivot points can be specified in either world space or each rigid body's local space. hkpAngularDashpotAction also provides damping and strength values.

Springs

Use an hkpSpringAction to join two rigid bodies with a spring. hkpSpringAction's `applyAction()` function applies forces to the bodies in an attempt to maintain the spring's rest length.

hkpSpringAction has the following properties:

- **Rigid bodies**

  The two rigid bodies that the hkpSpringAction will act on.

- **Attachment Points**

  The attachment points for each rigid body. The points can be specified in either world space or the rigid bodies' local space, and do not need to be inside the volume of either rigid body.

- **Rest length**

  A rest length for the spring. The default value is 1.

- **Strength**

  This governs the force the spring will apply to each attached body for each unit difference between its current length and its rest length - the default value is 1000.

- **Damping**

  This governs how quickly the oscillation of the spring stabilizes. A spring without damping will continue to oscillate up and down the same amount forever. A heavily damped spring will reduce any oscillations very quickly with the size of the oscillations getting much smaller each time. The default damping value is 0.1.

- **Extension and compression flags**

  You can use these values to specify whether the spring acts under extension or compression or both.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

For similar behavior to hkpSpringAction but using a constraint, use a malleable hkpStiffSpringConstraintData.

Motors

hkpMotorAction acts as a simple motor that can spin a rigid body. In each simulation step the action applies torque to the rigid body until it reaches a specified angular velocity. To create an hkpMotorAction, provide the following information to its constructor

- body - the rigid body to spin.
- axis - the desired angular axis of spin for the motor.
- spinRate - the rate of spin about this axis, specified in radians per second.
- gain - the speed with which the desired velocity is attained - the default value is 2.

The following sample shows how to create a motor that applies to the rigid body, myBoxRigidBody.

```
hkVector4 axis( 0.0f, 0.0f, 1.0f );
hkReal spinRate( 5.0f );
hkReal gain( 5.0f );

hkpMotorAction* motorAction = new hkpMotorAction( myBoxRigidBody, axis, spinRate, gain );
m_world->addAction( motorAction );
```

3.2.9.3 Creating a custom action

This section explains how to create a custom action. The example creates an action that applies a gravity-like force to a rigid body in each simulation step. As a result, the body "orbits" in an ellipse around a point during the simulation. The user of the action is able to specify the rigid body, the point around which the body will orbit, and the strength of the gravitational force.

Read the Dynamics chapter to find out more about applying forces to rigid bodies before exploring this example. For more details on the hkVector4 functions used in the example, see the Math chapter and the hkVector4 section of the Reference Manual. Read the Serialization chapter if you wish your custom actions to be serializable.

Derivation

First, decide which of the hkpAction classes to derive from. As the new gravity action applies to a single rigid body, it derives from hkpUnaryAction. hkpUnaryAction provides some basic functionality for the action and a data member for the entity that it applies to, `m_entity`.

```
// This action applies a "gravity" force to a satellite body, attracting it to a fixed point.
class GravityAction: public hkpUnaryAction
```

Construction

The action user must be able to specify the rigid body, the center around which it will orbit, and the strength of the force applied to it. As seen in the constructor below, the rigid body is passed on to the hkpUnaryAction constructor so that it becomes the action's `m_entity` member.

```
inline GravityAction::GravityAction(hkpRigidBody* satellite, const hkVector4 &gravityCenter, const hkReal
  gravityConstant)
    :    hkpUnaryAction(satellite),
        m_gravityCenter(gravityCenter),
        m_gravityConstant(gravityConstant) { }
```

Implementation of applyAction()

Now make the action do something! The implementation of `applyAction()`, should apply a force to `m_entity`, acting towards the specified gravitational center. The magnitude of the force should to take into account both the mass of `m_entity` - so that the action will work in the same way regardless of the rigid body's mass - and the distance between the body and the center.

First, find the direction in which the force should work:

```
hkVector4 dir;

// Sets dir to be the difference of m_gravityCenter and getRigidBody()->getCenterOfMassInWorld().
dir.setSub4( m_gravityCenter, getRigidBody()->getCenterOfMassInWorld() );
```

The `dir` vector now points from the body's center of mass to the specified gravity center.

Next, find the distance between the body and the point, and a unit direction. This example uses `normalizeWithLength3()`, which normalizes the vector as well as returning the distance. This is useful as the normalized vector is needed later in the example to create the force.

```
hkReal distance = dir.normalizeWithLength3();
```

Then find out the necessary magnitude of the force. To get the correct effect, it needs to be proportional to the body's mass, and inversely proportional to the distance squared: `F = (GMm)/d^2`.

```
// Use a version of the gravitational formula:
//
// F = (GMm)/d^2
//
// If G and M are fixed (rolled into m_gravityConstant), then F = m_gravityConstant * m / d^2
// i.e. the force is proportional both to the mass, and the inverse distance squared.
hkReal magnitude =  m_gravityConstant * getRigidBody()->getMass() / (distance * distance);
```

Next create the force as a vector acting along the correct direction (using the normalized direction vector), with the correct magnitude.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
hkVector4 force;
force.setMul4(magnitude, dir);
```

Finally, apply the force to the rigid body.

```
// Apply the gravity force.
getRigidBody()->applyForce( force );
```

Note that this action's work does not require time information from the simulation, so hkStepInfo is not needed.

Below is the complete `applyAction()` function for the ExampleGravityAction.

```
inline void GravityAction::applyAction( const hkStepInfo& stepInfo )
{
    hkpRigidBody* rb = getRigidBody();

    hkVector4 dir;

    // Sets dir to be the difference of m_gravityCenter and rb->getCenterOfMassInWorld().
    dir.setSub4( m_gravityCenter, rb->getCenterOfMassInWorld() );

    hkReal distance = dir.normalizeWithLength3();

    // Use a version of the gravitational formula:
    //
    // F = (GMm)/d^2
    //
    // If G and M are fixed (rolled into m_gravityConstant), then F = m_gravityConstant * m / d^2
    // e. the force is proportional both to the mass, and the inverse distance squared.
    hkReal magnitude =  (m_gravityConstant * rb->getMass()) / (distance * distance);

    hkVector4 force;
    force.setMul4(magnitude, dir);

    // Apply the gravity force.
    rb->applyForce( force );
}
```

Instantiation

Now that class for the action is written, create an instance of it, providing the target rigid body, center of gravity, and force strength.

```
hkpRigidBodyCinfo satelliteInfo;

// Set up satelliteInfo...
// ...
hkpRigidBody* satellite = new hkpRigidBody(satelliteInfo);

m_world->addEntity(satellite);

hkVector4 centreofgravity(0.0f, 0.0f, 0.0f);
GravityAction* gravityAction = new GravityAction( satellite, centreofgravity, 50.0f );
```

Call addAction()

Finally add the action to the simulation using `addAction()`. Remember to call `removeReference()` once the action is added.

```
m_world->addAction( gravityAction );

// After addAction() m_world references antiGravityAction so the local reference can safely be removed.
gravityAction->removeReference();
```

3.2.9.4 Important points when using actions

Top tips

Some important points to remember when using actions are:

1. Actions are normally **removed from the simulation** when their associated rigid bodies are removed.
2. Actions often need to **take time into account** when determining how large a force/implulse they should apply to each body.
3. Actions that act on more than one body **merge the simulation islands** the bodies are in.
4. Actions are **only applied to active bodies** .

Interpenetration

When using actions, be careful not to inadvertently cause interpenetration.

Interpenetration happens when one body is moved inside another, thus compromising the physical realism of the simulation. Havok uses its solver and special constraints to try to prevent this happening. However, if an action is added that directly sets the position or orientation of a body (e.g. using `hkpRigidBody::setPosition()`) then it risks causing an interpenetration that cannot be easily resolved.

For instance, if an action moves a body around by changing its position, it is possible that it will eventually move the body into something that is fixed e.g. the ground. Havok will detect this and attempt to recover physical consistency. However, in the next simulation step, the action will be called

again, (through `applyAction()`) which may again try to put the body under its control back into the ground.

A safer way to move a body is to use either forces or impulses, as in the example classes provided with the Havok SDK. The difference between a force and an impulse is that when a force is applied it has the effect of accelerating or decelerating the body. Just like a car, if it is traveling at 100mph and brakes it may be some time before it comes to a complete stop. Impulses change the velocity of a body directly (more like hitting a tree than applying a brake!). To change the orientation of a body apply forces and impulses off center, or apply torque and angular impulses. Find out more about applying forces, torques, and impulses in the Dynamics chapter.

3.2.10 Listeners

This section introduces you to listeners and events in Havok, and describes how to use the listener base classes provided with Havok Physics.

Listeners allow you to react to specific events that occur within the Havok simulation. These events can be anything from objects colliding, constraints being created, or entities being activated.

Unlike an action, which has a function that is called every single simulation step, listeners have functions that are only called when a particular event occurs. This makes them much more suitable for responding to infrequent changes of state, or for asynchronous events.

You simply add the appropriate listener to the simulation in order to register your interest in the event. When the event occurs, your specified callback code will run.

3.2.10.1 Callbacks and events

A listener in Havok simply implements a particular callback function that is called by the simulation when a corresponding event takes place. For example, an hkpActionListener's `actionAddedCallback()` function is called when an action is added to the hkpWorld.

Each callback allows the simulation to pass some useful information about the event to the listener, which the listener can then use when responding to the event. The type of information provided depends on the callback. For instance, an `entityAddedCallback()` is just passed a reference to the newly-added entity, while an hkpCollisionListener's `contactProcessCallback()` is passed a struct containing details of the event, including the hkCollidables involved and the collision detection results.

3.2.10.2 Listener types

Havok provides you with a number of base listener classes, each with virtual callback methods that you can implement to respond to particular types of event. You can create listeners that respond to events for a particular entity, or your listener can respond to events for the entire Havok world. Implementations of some of these listener interfaces are also used internally by Havok. The following sections introduce you to the Havok listener classes and tell you how to add these types of listener to the simulation.

World listeners

The following listener types can be added to the hkpWorld, using the appropriate hkpWorld `add..Listener()` function. You can see some examples of world listeners in the WorldListenerApi demo.

hkpIslandActivationListener

Listeners with these callbacks are notified when any of the world's simulation islands are activated or deactivated. Both the `islandActivatedCallback()` and `islandDeactivatedCallback()` are passed a reference to the relevant simulation island. You add and remove this type of listener using `addIslandActivationListener()` and `removeIslandActivationListener()`.

hkpConstraintListener

Listeners with these callbacks are notified when hkConstraints are deleted, or added or removed from the world. The `constraintDeletedCallback()`, `constraintAddedCallback()`, and `constraintRemovedCallback()` are all passed a reference to the appropriate hkConstraint by the simulation. You add and remove this type of listener using `addConstraintListener()` and `removeConstraintListener()`. An example of this listener can be found in the "Add Remove Constraints" demo (Physics Api / Dynamics / World).

hkpWorldDeletionListener

Listeners with this callback are notified when the hkpWorld is deleted. The `worldDeletedCallback()` is passed a reference to the hkpWorld. You add and remove this type of listener using `addWorldDeletionListener()` and `removeWorldDeletionListener()`.

hkpWorldPostCollideListener

Listeners with this callback are notified after all collision detection has been performed for all entities in the world. The `postCollideCallback()` is passed a reference to the hkpWorld, together with an hkStepInfo that stores the simulation's current delta time and inverse delta time. You add and remove this type of listener using `addWorldPostCollideListener()` and `removeWorldPostCollideListener()`.

hkpWorldPostIntegrateListener

Listeners with this callback are notified after the end of the `hkpWorld::integrate()` call. The `postIntegrateCallback()` is passed a reference to the `hkpWorld`, together with a `hkStepInfo` that stores the simulation's current delta time and inverse delta time. This type of listener can be added and removed using `addWorldPostIntegrateListener()` and `removeWorldPostIntegrateListener()`.

hkpWorldPostSimulationListener

Listeners with this callback are notified after each time the hkpWorld's `simulate()` function is called - they are fired after all the other callbacks in a given simulation step. `simulate()` is a protected function called inside `stepDeltaTime()`, which you use to step the simulation forward. The `postSimulationCallback()` is passed a reference to the hkpWorld. You add and remove this type of listener using `addWorldPostSimulationListener()` and `removeWorldPostSimulationListener()`.

A typical use of this callback might be to update the visual representations of the active rigid bodies in the games scene graph automatically, rather than performing the call after the simulation step call.

hkpIslandPostCollideListener

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Listeners with this callback will receive the callback after all collision detection has been performed for all entities in that island. This type of listener allows you to perform similar collision filtering etc. as the hkpCollisionListeners for hkWorlds and hkpEntities but work at a different level of granularity - i.e. hkSimulationIslands.

hkpIslandPostIntegrateListener

Listeners with this callback will receive the callback after `integrate()` is called for the hkpSimulation-Island. Again useful for situations where you want to filter or operate at a granularity of islands. Note that this callback is not raised when the simulation type is MULTITHREADED. This is because the integration and collision detection are interleaved to improve multithreaded resource utilization.

hkpActionListener

hkActionListeners have 3 callbacks:

- `actionDeletedCallback()` : Called when an action is deleted, note that hkpActionListener sub-classes must implement this function.
- `actionRemovedCallback()` : Called when an action is removed from the hkpWorld.
- `actionAddedCallback()` : Called when an action is added to the hkpWorld.

The `actionDeletedCallback()`, `actionAddedCallback()`, and `actionRemovedCallback()` are all passed a reference to the appropriate hkpAction by the simulation. You add and remove this type of listener using `addActionListener()` and `removeActionListener()`. An example of this listener can be found in the "World Listener" demo (Physics / Api / Dynamics / World).

hkpEntityListener

hkpEntityListener has three callbacks:

- `entityDeletedCallback()` : Called when an entity is deleted, note that hkpEntityListener sub-classes must implement this function.
- `entityRemovedCallback()` : Called when an entity is removed from the hkpWorld.
- `entityAddedCallback()` : Called when an entity is added to the hkpWorld.

Each callback is passed a reference to the relevant hkpEntity. You can either add an hkpEntityListener to a particular entity, or you can add an hkpEntityListener to the hkpWorld depending on whether you are interested in receiving events for a particular entity or all the entities in the hkpWorld.

Note that if you implement hkpEntityListener yourself, you should implement the `entityDeletedCallback()` function, calling `removeEntityListener()` to remove the listener from the entity's listener queue.

You add and remove hkpEntityListeners using the hkpWorld or relevant hkpEntity's `addEntityListener()` and `removeEntityListener()` methods.

hkpEntityListener callbacks are also used internally by Havok - for example, hkpCharacterProxy is also a hkpEntityListener.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

In a game scenario, hkpEntityListeners could be useful for maintaining the state of internal game data structures such as scene graphs etc.

hkpEntityActivationListener

Listeners implementing the hkpEntityActivationListener interface are notified when a particular entity is activated or deactivated. You add or remove an hkpEntityActivationListener using the relevant hkpEntity's `addEntityActivationListener()` and `removeEntityActivationListener()` methods.

Phantom Listeners

See hkpPhantomListener and hkpPhantomOverlapListener.

3.2.11 Postponing of Operations

Most of the functions in hkpWorld now use a locking mechanism. When the engine performs integration, collision detection or handles TOI events the world is internally locked. This allows the user to call any functions (e.g. addEntity, removePhantom, updateCollisionFilterOnEntity) at any time and they will be stored on an internal todo-list and executed at the earliest possible time when the hkpWorld is unlocked. The postponed functions are executed in the order which would be used if the world was not locked. Having a few postponed functions on the list, the first one is executed and then all recursive calls are executed. Only then the next postponed call from the list is processed.

The following functions are considered critical (all are members of hkpWorld, except where noted otherwise):

- addEntity()
- removeEntity()
- entityUpdateBP() – updates `hkpRigidBody`'s broadphase. Used by functions affecting a body's position/orientation and therefore its `hkAabb` representation; e.g. `hkpRigidBody::setPosition()`, `hkpRigidBody::setRotation()`.
- hkpRigidBody::setMotionType()
- addEntityBatch
- removeEntityBatch
- addConstraint
- removeConstraint
- addAction
- removeAction
- hkpWorldOperationUtil::mergeIslands – used internally
- addPhantom
- removePhantom
- addPhantomBatch

- removePhantomBatch
- updateFilterOnEntity
- updateFilterOnPhantom
- updateFilterOnWorld

3.2.12 Simluation Determinism

Havok simulation is deterministic, meaning that a simluation will yield exactly identical results if it's run several times from the exact same starting point on the same platorm and configuration.

To run Havok deterministically you must recreate the world from the same data. If you perform custom actions on the world, these must be done in the same order. This ensures that internal time of the simulation, and grouping and ordering of islands, entities, constraints, collision entries, etc. is the same. You must also run Havok in the same simluation mode, i.e. single-threaded simulation yields different results than multithreaded simulation. However multithreaded simulation will be identical independently of number of threads.

Simluation output is independent of how objects are laid out in memory, i.e. there's no sorting based on memory addresses. Therefore there's no need for reinitialization of the hkBaseSystem.

Havok run in a single thread may not yield the same or even similar results if you:

- shift the system in space, or offset position of all bodies by a vector – varying results are caused by finite numerical accuracy
- change the starting time of the simulation
- add a new body that doesn't even physically interact with the rest of the system – a new body may change the way simulation islands are organized

Additionally, Havok run in multiple threads will not yield the same results if you:

- modify the world in actions and most callbacks – callbacks order is nondeterministic when simulation is run concurrently on several processors
- change the simulation settings (e.g. minimum size of hkpSimulationIslands that are solved in multiple threads)

Havok may also yield different results when you run a simulation on different platofrms. Also, simluation on PLAYSTATION®3 may become non-deterministic when we change the default settings and allow the PPU to take SPU tasks related to collision detection and constraint solving.

3.2.13 Memory Issues

3.2.13.1 Memory Capping

It is possible to specify an upper limit for how much memory Havok uses for physical simulation. This is done by setting the `m_criticalMemoryLimit` member in `Common\Base\Memory\Memory\hkMemory` to your desired upper memory limit. This lets you guarantee that Havok will not use up valuable memory resources needed for other parts of your game.

All of Havok's small memory allocations (those less than 8k) are allocated to pages in its Pool Memory system. See the Memory Management section of the hkBase documentation for more details. Any particularly memory-intensive operations (i.e. those requiring more than 8 KB of memory) perform 'look-ahead' checks to ensure that there is sufficient memory available for them to run safely. Larger allocations are typically needed for:

- Broadphase calculations.
- MOPPs.
- Collision solving.

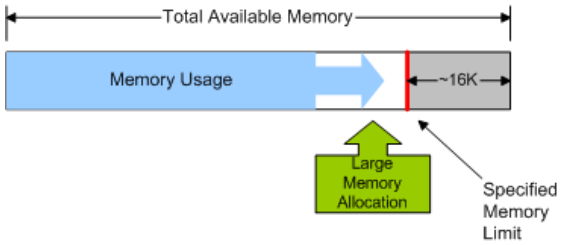


Figure 3.37: This large memory allocation's look-ahead check will fail because smaller memory allocations have made overall memory use too high.

Since smaller memory allocations do not perform any look-ahead checks, the amount of Pool Memory allocated by Havok can creep upwards over time. So although Havok frequently checks the total amount of memory in use (in addition to look-ahead tests), the memory limit may be exceeded by a small amount before the system detects failure.

It is possible to recover from such failures, but the process is computationally expensive. The following code example from the `LimitingMemory2` demo contains typical recovery logic:

```
if ( !hkMemory::getInstance().hasMemoryAvailable(wiggleSpace))
    {
        m_world->lock();
        m_watchDog->freeMemory(m_world);

        m_world->unlock();
    }
```

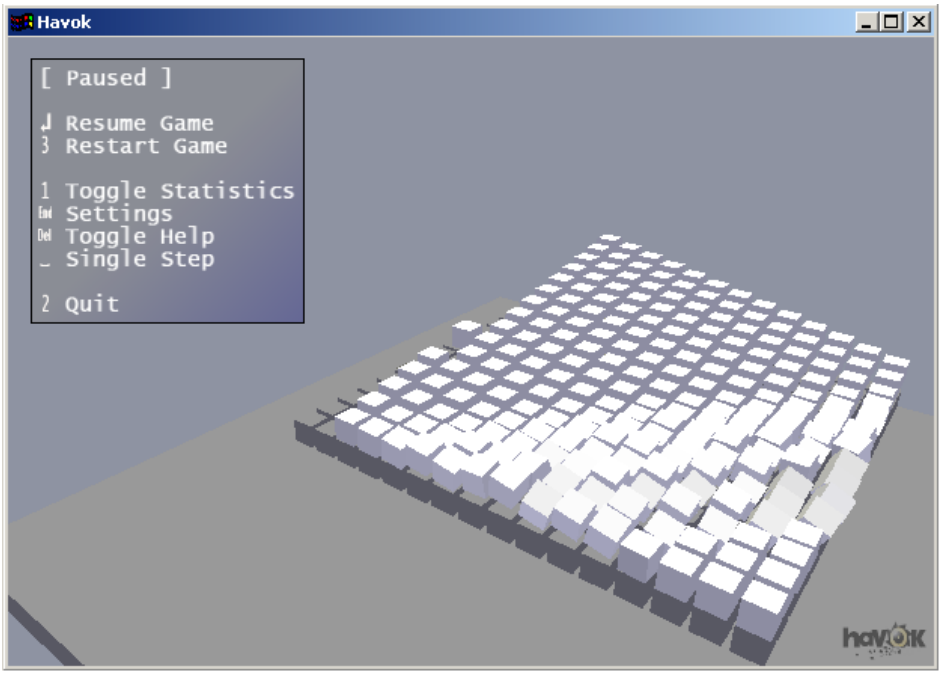


Figure 3.38: A screenshot from the LimitingMemory2 demo. Blocks are removed from the world to recover when memory usage exceeds the m_criticalMemoryLimit.

In the `LimitingMemory2` demo the `m_criticalMemoryLimit` is exceeded only occasionally, and by no more than around 1.5K. Since Pool Memory page allocations are 8K in size, it is recommended that you set your `m_criticalMemoryLimit` to a value roughly 16K below the amount of physical memory allocated to Havok. This safety margin ensures that recovery is performed before Havok runs out of physical memory and crashes.

3.2.13.2 Memory Watchdogs

The `hkWorldMemoryWatchdog` class (found in `Physics\Dynamics\World\Memory`) can be used to prevent the `m_criticalMemoryLimit` from ever being exceeded. This is done by setting its `m_memoryLimit` member to a lower value than the `m_criticalMemoryLimit`. The memory watchdog uses this lower limit to keep memory usage a safe distance from the critical limit. In order to achieve this, the lower limit should be further from the critical limit than the largest amount of memory likely to be allocated in a single step. As long as memory usage is kept below the `m_memoryLimit`, even the most demanding simulation step in your game will not exceed the `m_criticalMemoryLimit`.

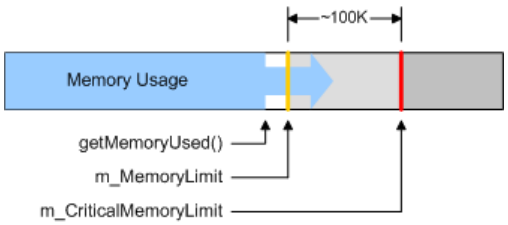


Figure 3.39: As long as Havok's memory usage is below the m_memoryLimit, the m_criticalMemoryLimit will not be exceeded in a single physical step.

The watchdog checks memory usage levels after every physical simulation step. If memory usage is greater

than the watchdog's `m_memoryLimit`, steps are immediately taken to reduce it to a safe level. The default memory watchdog implementation, `hkpDefaultWorldMemoryWatchdog`, removes entities from the world according to their `m_autoRemoveLevel`s until memory usage is safely below the `m_memoryLimit` once more. However, you are encouraged to develop a custom watchdog implementation more suited to your game.

The `LimitingMemory` demo shows how unimportant objects such as debris can be removed first when memory needs to be freed up. The demo continually creates ragdolls and debris and adds them to the world, and uses a `hkWorldMemoryWatchdog` with the following virtual implementation of `freeMemory()` to remove debris from the world before ragdolls.

```
void MySimpleObjectWatchDog::freeMemory( hkpWorld* world )
{
    int targetAvailable = m_minMemoryAvailable;

    hkMemory& mem = hkMemory::getInstance();
    while (!mem.hasMemoryAvailable(targetAvailable))
    {
        if ( m_debris.getSize() > 0 )
        {
            // Check if the debris object has been removed elsewhere
            if (m_debris[0]->getWorld() != HK_NULL)
            {
                world->lock();
                world->removeEntity(m_debris[0]);
                world->unlock();
            }
            m_debris[0]->removeReference();
            m_debris.removeAtAndCopy(0);
        }
        else
        {
            if ( m_ragdolls.getSize() > 0 )
            {
                world->lock();
                // Check if the ragdoll object has been removed elsewhere (just check the first entity)
                if (m_ragdolls[0]->getRigidBodies()[0]->getWorld() != HK_NULL )
                {
                    world->removePhysicsSystem(m_ragdolls[0]);
                }
                m_ragdolls[0]->removeReference();
                m_ragdolls.removeAtAndCopy(0);
                world->unlock();
            }
            else
            {
                break;
            }
        }
    }
}
```

It is important to choose a suitable `m_criticalMemoryLimit` for your game relative to the amount of physical simulation it contains. Tests run on the `LimitingMemory` demo reveal that the largest memory allocations per physical step (due to new collisions etc.) are around 11K. The amount of memory used to create a ragdoll and add it to the world is in the region of 120K, which is much more expensive. This total includes memory allocated to the Havok graphics renderer - the real cost to the physics engine is less than 100K.

Figure 3.40: The LimitingMemory demo, in which debris gets removed before ragdoll components.

Your `m_memoryLimit` value should be as high as possible, while still keeping it a safe distance from the critical limit. Using the figures mentioned above as a guideline, it seems advisable to make your `m_memoryLimit` about 100K less than your `m_CriticalMemoryLimit`, and to additionally call `watchMemory()` when adding large systems such as ragdolls to your game. `hkworldMemoryWatchdog`'s `watchMemory()` method will check current memory usage levels and call `freeMemory()` if necessary, ensuring that your `m_CriticalMemoryLimit` is not exceeded.

To find the best `m_memoryLimit` and `m_criticalMemoryLimit` values for your game, try running some performance tests, tweaking your chosen values until you are happy with the results. Remember that the `hkWorldMemoryWatchdog` is a fallback solution - if you find your game using it a lot then it might be worth allocating a larger chunk of memory to physical simulation or reducing the amount of physics in your game.

3.2.14 Saving Contact Points

The `hkpSaveContactPointsUtil` allows you to save contact information for chosen colliding pairs of bodies or for an entire scene at once.

For each collision entry (i.e. a pair of bodies in contact) we store `hkContactPoints`, their `hkpContactPointProperties` and the entire collision agent hierarchy in case when compound shapes are involved. The utility does not support all existing collision agents. Therefore when unsupported agents are encountered in an agent hierarchy, then saving of a collision entry is aborted.

There is also some limited serialization support for the contact point information, which allows you to store & restore full contact information, but puts restrictions on the runtime's configuration (see the Limitations section).

The Physics / Api / Dynamics / World / World Snapshot with Contact Points demo is a simple example

of how to save a hkpHavokSnapshot with contact points and how to load it back from a disk.

3.2.14.1 Methods and parameters for saving

There are three hkpSaveContactPointsUtil::saveContactPoints() function variations which let you save collision entries for the entire hkpWorld, for a list of hkpEntities, or an explicit list of collision entries (represented by hkpAgentNnEntry structures). (The variation with specified list of hkpEntities attempts to save all collision entries, where at least one of the involved hkpEntities is on the input list.)

If a single collision entry cannot be saved (i.e. when unsupported agent types are encountered), it's just ignored, and the utility continues to process the remaining collision entries.

Collision entries are saved into an array of hkSerializedAgentNnEntries, enclosed in a hkpPhysicsSystemWithContacts instance. (Be cautious with saving collision entries one-by-one as there is no checks implemented to check for duplications, i.e. multiple hkSerializedAgentNnEntries stored for one pair of bodies.)

The serialized collision entries have to be linked back to their respective hkpEntities later. To do that you may chose to either store hkpEntity pointers directly, or use custom unique ids. The former solution is useful if you're saving an entire hkpHavokSnapshot with hkpRigidBodies included in it. The latter solution is helpful if you want to remove involved hkpEntities from the hkpWorld after saving, destroy them and then recreate later before you load the points again.

Use the SavePointsInput structure to choose between saving hkpEntity pointers or custom ids. If using ids you'll also need to provide a callback function that assigns an hkUlong id for a given hkpEntity. You'll need to provide a symmetric function later when loading collision entries.

3.2.14.2 Methods and parameters for loading

There are two functions for loading contact points. One attempts to restore collision entries for the entire hkpWorld, the other function only attempts to restore collision entries involving hkpEntities from a list.

When loading points, the utility does not expect to match all saved collision entries. For each saved collision entry it attempts to find a matching pair of hkpEntities. If one does not exist that saved collision entry is ignored.

The LoadPointsInput structure allows you to customize the way that saved contact points are loaded.

- You can choose to destroy a serialized collision entry after it is loaded (that's useful, if you're loading contact points gradually one by one).
- You can zero m_userData stored in hkpContactPointProperties (might be useful, if you're using user data to store pointers to your own structures).
- You may choose to fire or skip collision callbacks. Namely contactPointAdded and contactPointConfirmed callbacks. Note however that using those callbacks usually requires special handling in your callback code, different to handling of normal callbacks.

  The contactPointAdded callbacks don't have the proper hkpCdBody pointers you may choose to pass the top level hkpCollidable pointer or HK_NULL. (That's because normally those callbacks are fired from within the collision pipeline, when they can reference temporary hkpCdBody objects.)

Also note that `conactPointConfirmed` callbacks are not called for all points. They are only called for points newly created in the previous simluation step, as it was in the orignal simulation, from which the contact points were saved.

- Finally you should provide a callback function for mapping your custom unite entity ids to `hkpEntity` pointers. This function will only be used by `hkSerializedAgentNnEntries` that are marked to use those custom ids instead of direct `hkpEntity` pointers.

3.2.14.3 Serialization

You can serialize a `hkpSerializedAgentNnEntry` or a collection of such entries stored in a `hkPhysicsSystemWithContactPoints`. When storing contacts separately from `hkpEntities`, remember to map the `hkpEntity` pointers to unique ids (see the Methods and parameters for saving' section above).

You can also use the `hkpHavokSnapshot::save(hkpWorld*, , bool saveContactPoints = false)` function to quickly create a physics snapshot with contact points.

There are platform restrictions for serialization of saved contact points. See the section below.

3.2.14.4 Limitations

Types of collision agents supported

The `hkpSaveContactPointsUtil` only supports collision agents working in the streaming-agents technology, i.e. all agents whose class name ends with the -3 suffix and which are defined in the hkinternal library. Other agents are not supported, and saving is aborted when they're found.

Restrictions on hkpCollisionDispatcher's configuration

It is advised to have the same set and order of `hkCollisionAgents` registered both when loading and saving contacts.

That's because when saving a collision entry, the utility saves raw data from an agent stream. That data also includes the agents' types, which in turn are simply indices of registration records in a table in the hkpCollisionDispatcher.

Cross-Platform and Layout Issues

Saved contact entries contain raw data from agent streams, have no reflection data, and cannot be versioned. Therefore, in the case of a change in the data structures of core collision agents in the future releases, the user will have to recreate the contact point data.

For similar reasons we cannot use our serialization framework to port the data between platforms with different data layout rules, except if the only difference between the platforms is their endianness. There is the specialized `hkpSaveContactPointsEndianUtil` that converts the endianness of the stored data.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.3 Collision Detection

3.3.1 Introduction

This chapter introduces you to the Havok collision detection system. As well as an overview of Havok collision detection and how it works, this chapter provides information on:

- Creating shapes for your rigid bodies
- Working with collision agents
- Using collision filters
- Optimizing and tuning collision detection
- Getting the results of collision detection
- Customizing the collision detection.

In addition to reading this chapter, take time to look at the relevant demos provided with your Havok Physics installation - these can be found in the `shapesapi`, `collisionapi`, and `raycastapi` directories. Each of these simple games illustrates a particular aspect of working with Havok collision detection. These demos are explained in tutorial form in this chm.

3.3.2 Overview

The primary focus of the Havok collision detection is to calculate all collision information between all bodies in the scene. Each body consists of a position (hkMotionState) and pointer to a surface representation (hkpShape). To perform this task efficiently, the collision detection is using a pipeline of algorithms to reduce the amount of CPU necessary:

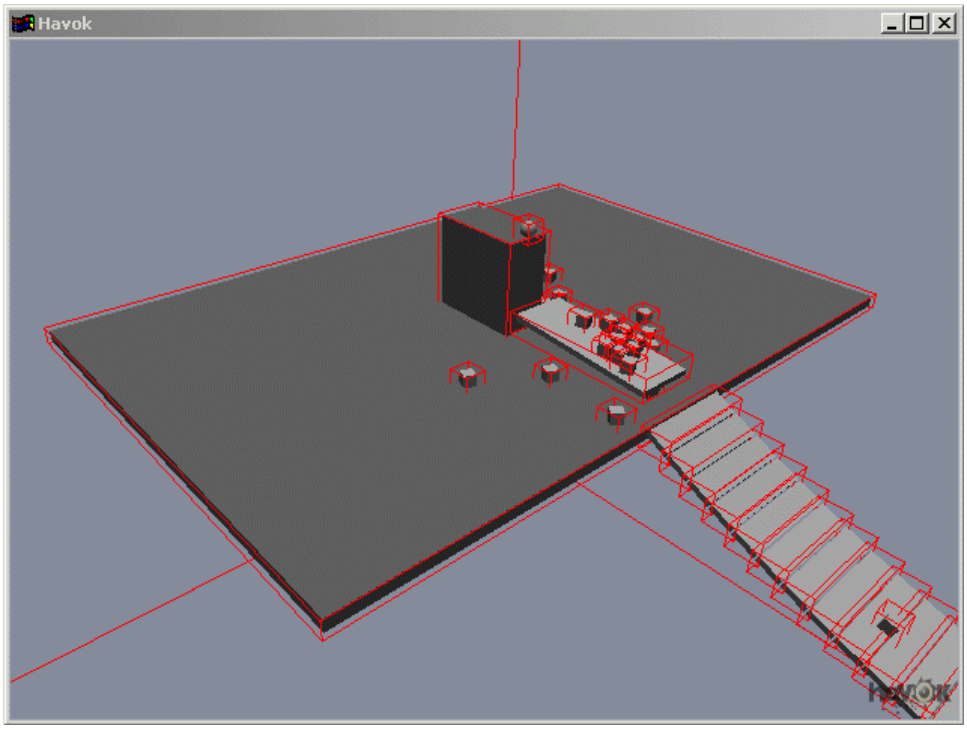
- The broadphase is used to find all pairs of objects, which are potentially colliding.
- The narrowphase is used to calculate the collision details. A simple example would be the collision between two spheres. However this narrowphase can also involve using some midphase algorithms, e.g. using an axis aligned bounding volume structure to identify subparts of a landscape that are potentially colliding with a players car.

One of the strengths of the Havok collision detector is that it can be used in a variety of ways:

- The user can query the system at any time for any type of collision information between bodies.
  - The closest points between two bodies
  - Whether two object penetrate
  - Linear cast a point through space and get the first or all hits with other rigid bodies
  - Linear cast any shape through space and get all collision information along the path
  - Do a raycast and get the first or all points where a ray hits an object in the world
- The physics system however needs a quite different interface to the collision detector:
  - The collision detector creates a high quality persistent manifold to be used by the physics engine
  - The collision detector calculates time-of-impact (TOI) events if two fast moving objects are colliding outside the 'normal' physical timestep (physical-synchronuous-instance = PSI).
  - The collision detector can use frame coherency to reduce the CPU load significantly.

3.3.2.1 The broadphase

Imagine a scene with 10 arbitrary objects. Each of these could potentially collide with any of the 9 others, giving a total of (10 x 9) /2 = 45 potentially colliding pairs of objects. As the number of objects in the scene increases, the potential number of colliding pairs increases $O(n^2)$, but on average during simulation the number of actual interactions is far less. To avoid checking all these pairs for exact collisions, we try to quickly identify the pairs likely to be colliding in the first phase (the broad phase). To do this efficiently, we build an axis-aligned bounding box (Aabb) for every object in the scene that fully contains the object at the current position. Then we only have to find pairs of overlapping Aabbs to see if we have to check any details of this pair. This representation of an Aabb is shown in the following illustration, where the red lines represent the broadphase bounds:



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Havok's implementation of the broadphase is a 3-Axis sweep and prune algorithm. The basic idea behind this algorithm is to calculate all Aabbs for all objects. For each axis (x,y and z), we reduce all Aabbs to line segments by projecting the Aabb onto this axis. Then we identify pairs of overlapping Aabbs by looking at their 3 projections. Only if all 3 projections of two Aabbs overlap, we say that these Aabbs are overlapping.

This approach has a number of nice and not so nice properties:

- Region queries, as required by raycasting or inserting and deleting objects, are slow.
- The algorithm uses only a small amount of memory and does only very few runtime memory allocations.
- The algorithm is unbeatable fast for solving the normal physical simulation problem.

In short, the broadphase in Havok is extremely fast for physical simulation and can be slow for other operations. However Havok provides a number of helper programs, which can help to avoid those bad cases: e.g. you can add multiple objects at once using the batch add and remove functionality of the hkpWorld.

3.3.2.2 The narrowphase

The broadphase gives us a list of potentially colliding pairs of objects. The job of the narrowphase is to check whether and how those pairs of objects are actually colliding. To get this done the narrow phase works in the following way:

- The type of the two shapes of the bodies is extracted.
- The hkpCollisionDispatcher is asked to select an appropriate function, which can handle the given types of shapes
- The function returned by the dispatcher is called, e.g. a sphere sphere penetration check

As many of the physics algorithms greatly benefit from frame coherency, Havok also uses a different approach, which allows the efficient use of collision caches: the collision agent. A Collision Agent can handle collisions between two shapes of given types and typically remains persistent across frames. So now the steps to be performed are:

- The shape types of the two bodies are extracted.
- The hkpCollisionDispatcher is asked to select an hkpCollisionAgent creation function.
- Create the collision agent.
- Call the appropriate function in the hkpCollisionAgent, typically this is to maintain a set of contact points for that pair. We call this set of points collision manifold. This manifold is used later on by the solver to keep the objects apart.

The collision agent continues to exist as long as the pair of objects overlap in the broadphase.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.3.2.3 Using the results

There is a number of ways to get results from the collision detector:

- Query the system directly, e.g. hkpWorld::getClosestPoint. The results are returned using a callback. To make things simple Havok provides a small number of default implementations of that callback (e.g. hkpClosestCdPointCollector)
- Implement the callbacks in the hkpCollisionListener class to retrieve and modify all information that is passed from the collision detector to the Havok dynamics.
- Redirect the output of the collision detector not to go into the constraint solver but to end up in your own data structures.
- Implement your own custom shape and agent to control the data flow within the collision detector directly. A typical example would be the hkPhantomCallback shape, which reports any overlaps with other shapes to the user and only to the user.

Using the first two techniques is simple and straightforward. The later two require some understanding of the underlying architecture and are not recommended for beginners.

3.3.3 Collidables

The core collision object is the hkpCollidable class. In order to do collision detection, each hkpCollidable needs a surface representation (hkpShape) and a position (hkMotionState), thereby linking the shape with the position and rotation of a dynamics object. If the collision detector is used with the hkdynamics library, this hkpCollidable class is an embedded class in the hkpWorldObject class and the owner of the hkpCollidable points to the hkpWorldObject. hkpWorldObject is the base class for all hkpRigidBody, hkpEntity and hkpPhantom objects.

As Havok dynamics is not 3 dimensional but 4 dimensional (the 4th dimension being time), the hkMotionState includes information about the movement of a rigid body between two physics frames. However if you want to use the collision detector asynchronously, you only need a position and rotation. Therefore we allow the hkpCollidable to either point to an hkTransform or an hkMotionState (Note: any calls to processCollision expects an hkMotionState pointer).

More members:

- hkpBroadPhaseHandle member, which means that the collidable can be added to the collision detection broadphase.
- hkUint32 m_collisionFilterInfo that allows you to specify information for filtering
- hkReal m_allowedPenetrationDepth which defines how deeply you allow objects to penetrate other objects.
- hkpCollidableQualityType m_qualityType which defines how much CPU you want to invest to stop this object from penetrating other objects. See the chapter on continuous physics for more information on this.
- hkInt8 m_type, whether the collidable is owned by an entity or a phantom

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

In order to find out who the owner of a particular hkpCollidable is, you can query the hkpCollidable type, which is either set to hkpWorldObject::BROAD_PHASE_ENTITY or hkpWorldObject::BROAD_PHASE_PHANTOM. Then you can safely cast the void* returned from hkpCollidable::getOwner() method:

```
int collidableType = myCollidable->getType();
if ( collidableType == hkpWorldObject::BROAD_PHASE_ENTITY )
{
  hkpEntity* myEntity = static_cast<hkpEntity*> ( myCollidable->getOwner() );
}
if ( collidableType == hkpWorldObject::BROAD_PHASE_PHANTOM )
{
  hkpPhantom* myPhantom = static_cast<hkpPhantom*> ( myCollidable->getOwner() );
}
```

Or you can use the two helper functions hkGetRigidBody(hkpCollidable*) and hkGetPhantom(hkpCollidable*). These functions check the type and return HK_NULL if the type does not match the requested return type. Otherwise they return the owner cast to the requested type.

3.3.4 Creating Shapes

All Havok entities have an hkpShape pointer stored in their hkpCollidable. This defines their shape and how they collide with other bodies. hkpShapes can be implicit like spheres and planes or explicitly represented like a polygon mesh exported from a modeler. We also use shapes to represent abstractions like lists of hkpShapes and transformed hkpShapes, thus an hkpShape can be a hierarchical structure. hkpShape instances can be shared within or even between rigid bodies and this is true for both fixed and dynamics bodies. Note however that hkCollidables are owned by exactly one hkpEntity or hkpPhantom.

3.3.4.1 The hkpShape class and shape types

All Havok shapes inherit from the hkpShape class, which specifies some common functions used by all shapes.

An hkpShape has a `getAabb()` function that gets an AABB for the shape. This function is used by the hkpWorld to get the AABB for each object that is added to the broadphase.

An hkpShape has a `getMaximumProjection()` function that gets the "furthest" point on the shape surface in a given direction, or an approximation thereof. It is used to help create bounding volumes for hkpShapes.

An hkpShape has a `castRay()` function that performs a raycast against the shape, calculating the parameterised distance to the minimum intersection point, as well as additional information about which subshape of a hierarchical shape was hit (if relevant).

Each Havok shape also has a `getType()` function that returns its type. The hkpCollisionDispatcher uses these shape types to select the most appropriate collision detection algorithm to deal with each pair of potentially colliding objects. You can find out more about how this works in the Collision Agents section.

hkpShape objects can also have one or more "alternate" types. These are registered dynamically with the

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

dispatcher. hkpAgentRegisterUtil registers the alternate types for all Havok shapes. For example, an hkpBoxShape has the type HK_SHAPE_BOX - however, an hkpBoxShape and an hkpSphereShape both have the alternate type HK_SHAPE_CONVEX, inherited from their common parent class, hkpConvexShape.

hkpShapes can either be of "basic" type (representing a sphere, box etc.), or of "aggregate" type (list of hkpShapes, hkpShape with additional transform etc.). This allows the user to create complex hierarchical shape structures, with aggregate types at the internal nodes of the shape tree, and basic types at the leaves. In the simplest case this allows the user to create "compound" shapes from a list of basic shapes with additional transforms. However, there is no limit on the complexity of the hierarchy, provided it is not recursive and does not contain a loop. There may of course be performance loss associated with large hierarchies.

Havok provides the following hkpShape interfaces:

Class	Type	Example Implementation
hkpShape	None	The base class for all shapes
hkpBvTreeShape	HK_SHAPE_BV_TREE	A Bounding Volume or BSP Tree with shapes at the leaf nodes
hkpConvexShape	HK_SHAPE_CONVEX	A convex shape, such as an egg, a pyramid, etc.
hkpPhantomCallbackShape	HK_SHAPE_PHANTOM	A floating power-up
hkpShapeCollection	HK_SHAPE_COLLECTION	A triangle soup
hkpHeightFieldShape	HK_SHAPE_HEIGHT_FIELD	A rolling landscape represented by a 2D array of heights.

Havok has implementations for the following basic shapes:

Class	Type	Example
hkpBoxShape	HK_SHAPE_BOX	A box
hkpConvexVerticesShape	HK_SHAPE_CONVEX_VERTICES	A point cloud, a convex mesh
hkpCapsuleShape	HK_SHAPE_CAPSULE	The limb of a ragdoll
hkpSphereShape	HK_SHAPE_SPHERE	A ball
hkpTriangleShape	HK_SHAPE_TRIANGLE	A wing
hkpCylinderShape	HK_SHAPE_CYLINDER	A barrel

Havok has implementations for simple wrapping shapes:

Class	Type	Description
hkpTransformShape	HK_SHAPE_TRANSFORM	Transforms the position and orientation of any child shape
hkpConvexTransformShape	HK_SHAPE_CONVEX_TRANSFORM	Transforms the position and orientation of its convex child shape. This is typically faster than using the hkpTransformShape
hkpConvexTranslateShape	HK_SHAPE_CONVEX_TRANSLATE	Translates the position of its convex child shape. This is even faster than the hkpConvexTransformShape

Havok has implementations for the following aggregate shapes:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Class	Type	Example
hkBVShape	HK_SHAPE_BV	An OBB (oriented bounding box) around another shape
hkpListShape	HK_SHAPE_LIST	A list of other shapes
hkpConvexListShape	HK_SHAPE_CONVEX_LIST	A list shape specially optimized for the case where every child shape is a convex shape. Not available on SPU.
hkpExtendedMeshShape	HK_SHAPE_TRIANGLE_COLLECTION	A collection of triangle lists and general hkpShapes. Use this to point to your graphics representations directly, or to efficiently combine static triangles with other static shapes.
hkpStorageExtendedMeshShape	HK_SHAPE_TRIANGLE_COLLECTION	A hkpExtendedMeshShape that owns its own triangle buffers. Useful for serialization in certain circumstances.
hkMoppBvTreeShape	HK_SHAPE_BV_TREE	A collection of shapes bounded by Havok's patented MOPP bounding tree
hkpSimpleMeshShape	HK_SHAPE_TRIANGLE_COLLECTION	A copy of an indexed vertex array

You can find out more about each shape type in the following sections.

3.3.4.2 Simple shapes

hkpConvexShape is the parent class for all the simple shapes described in this section. All hkConvexShapes have the alternate type HK_SHAPE_CONVEX, and inherit from this class. All hkConvexShapes can use the same convex collision detection code (the GSK algorithm) which requires implementation of `getSupportingVertex()` and `getFirstVertex()`. hkpConvexShape also has a radius member that allows you to add an extra "shell" to any convex shape, such as a sphere or box. You can find out more about how to use this in the Optimizing and Tuning section.

Boxes

An hkpBoxShape is a simple box centred on the shape's local origin. To create an hkpBoxShape, you must specify a half-extent for it, as in the following example. An (X by Y by Z) box has the half-extent (X/2, Y/2, Z/2).

```
hkVector4 halfExtent(1.0f, 2.0f, 3.0f);
hkpBoxShape* cube1 = new hkpBoxShape(halfExtent, 0);// creates a 2x4x6 box, with no 'shell' radius
```

You can change a box shape's half-extent after creating it using its `setHalfExtent()` function, but not after it has been added to an entity.

Spheres

An hkpSphereShape is a sphere around the shape's local origin with a specified radius. To create a sphere whose center is not the origin, you would need an additional hkpTransformShape as its parent. See hkpTransformShape.

```
hkpSphereShape* sphere = new hkpSphereShape(2.0f);
```

Triangles

An hkpTriangleShape is a simple triangle, stored in the shape as 3 hkVector4 vertices. To create an hkpTriangleShape, you specify the triangle's three corners in the shape's local space, as in the following example:

```
hkVector4 v0(  0, 0, 0 );
hkVector4 v1(  0, 0, 5 );
hkVector4 v2(  5, 0, 0 );

hkpTriangleShape* triangle = new hkpTriangleShape( v0, v1, v2 );
```

You can also create an "empty" hkpTriangleShape by passing no parameters to the constructor, and specify the triangle details later. The following example, from the Triangle demo, uses this approach.

```
int numVertices = 3;
int stride = 4;

float vertices[] = {
    -0.5f, -0.5f,  0.0f, 0, // v0
     0.5f, -0.5f,  0.0f, 0, // v1
     0.0f,  0.5f,  0.0f, 0, // v2
};

hkpTriangleShape* shape = new hkpTriangleShape();

int index = 0;
for (int i = 0; i < 3; i++)
{
    hkVector4 v(vertices[index], vertices[index + 1], vertices[index + 2]);
    shape->setVertex(i, v);
    index = index + stride;
}
```

Triangle Extrusion

Triangles also have an "extrusion" parameter. This is a hkVector4 that defaults to a zero length vector. When set, a triangle uses this vector to become a convex object represented by the triangle extruded by the vector. Effectively 3 new points are created on the fly during collision detection, one corresponding to each original point plus the extrusion vector. This is designed to help the bullet through paper issue in certain situations, without using continuous simulation, however it does not guarantee robust results. Currently it is supported by the hkpExtendedMeshShape, where extrusion values are set per subpart of the mesh (for triangle subparts). All triangles in the subpart are then extruded in the direction specified (which defaults to 0, i.e is disabled). If you have a ground made of triangles, with nothing beneath it, you can extrude the triangles a few metres in the downwards direction. When high speed non-continuous debris hits the landscape it will be more likely to bounce off rather than tunnel through. The effects of this extrusion is improved when also using one sided welding, as contact points will be correctly rotated to point up, even when objects are deeply penetrating.

Capsules

The hkpCapsuleShape class lets you create a capsule defined by two points. You need to specify these points in the shape's local space when creating the shape. If a non-zero radius is used, then the "shell" created will form a capsule / lozenge shape.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Cylinders

The hkpCylinderShape class lets you create a cylinder defined by two points and a cylinder radius. Note that this cylinder radius is separate from the convex shape "radius" which can be used to add an extra "shell" around the cylinder for performance reasons. You need to specify these parameters in the shape's local space when creating the shape.

Convex vertices shapes

An hkpConvexVerticesShape allows you to construct a convex shape from a specified set of vertices, as in the following example from the Convex Vertices demo.

```
int numVertices = 4;

// 16 = 4 (size of "each float group", 3 for x,y,z, 1 for padding) * 4 (size of float)
int stride = 16;

float vertices[] = { // 4 vertices plus padding
    -2.0f, 2.0f, 1.0f, 0.0f, // v0
     1.0f, 3.0f, 0.0f, 0.0f, // v1
     0.0f, 1.0f, 3.0f, 0.0f, // v2
     1.0f, 0.0f, 0.0f, 0.0f  // v3
};

hkpConvexVerticesShape* shape;
hkArray<hkVector4> planeEquations;
hkGeometry geom;
{
    hkStridedVertices stridedVerts;
    {
        stridedVerts.m_numVertices = numVertices;
        stridedVerts.m_striding = stride;
        stridedVerts.m_vertices = vertices;
    }

    hkpGeometryUtility::createConvexGeometry( stridedVerts, geom, planeEquations );

    {
        stridedVerts.m_numVertices = geom.m_vertices.getSize();
        stridedVerts.m_striding = sizeof(hkVector4);
        stridedVerts.m_vertices = &(geom.m_vertices[0](0));
    }

    shape = new hkpConvexVerticesShape(stridedVerts, planeEquations);
}
```

You specify the vertices in the shape's local space. As you can see, you also need to specify a striding value and the number of vertices. The striding parameter is the byte difference between one float triple and the next float triple (so usually 12 or 16 or greater).

Note that the vertices given in the constructor are not shared, but copied into the shape.

The set of vertices used does not need to be convex - the convex subset of vertices for a cloud of vertices can be calculated using `hkpGeometryUtility::createConvexGeometry()`. This function should be called off-line to generate that set of vertices as it is slow and memory intensive - see the hkpreprocess section for more information.

In addition, a set of plane equations must be supplied if the shape is ever to be tested for ray inter-

sections. These plane equations can be calculated from the set of vertices using the helper method `hkpGeometryUtility::createConvexGeometry()`. If no plane equations are supplied the shape will collide correctly with other shapes, but will fail any raycast queries, so it is always a good idea to supply these equations. It is usually a good idea to expand these equations by the convex collision radius that you are using for the shape aswell so that the raycasts and the normal collisions happen on the same virtual surface. You can use the `hkpGeometryUtility::expandPlanes(planeEqns, radius)` to shift the planes you computed for the surface out by the convex radius before giving them to the shapes constructor.

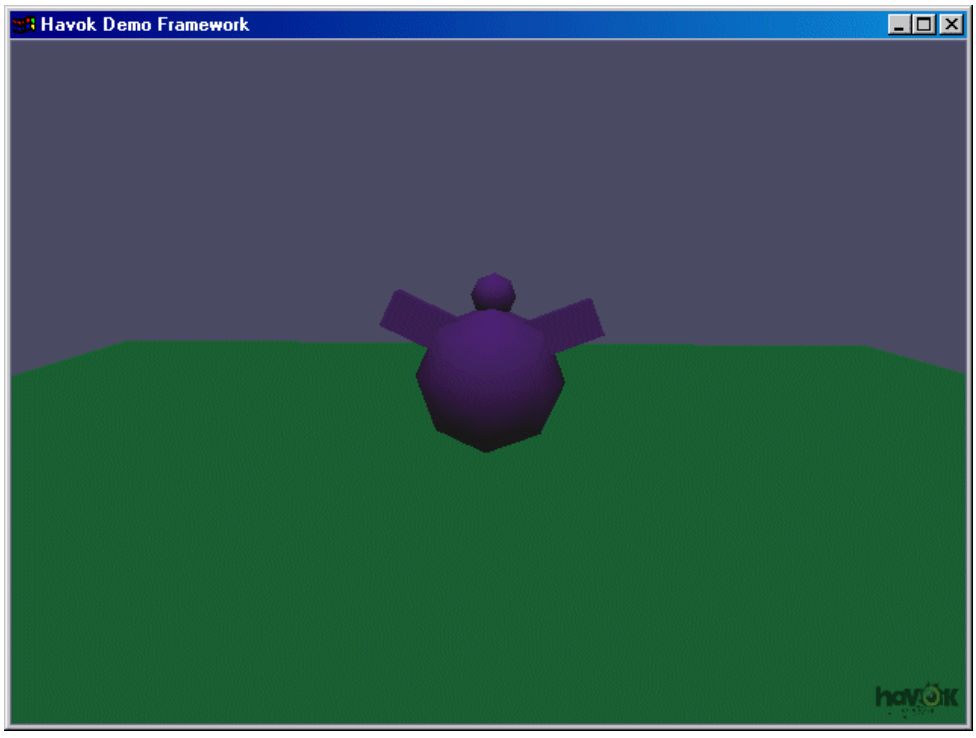
3.3.4.3 Compound shapes

Shapes which contain other shapes implement the `hkpShapeContainer` interface. You can use the following classes to create more complex shapes made up of several simpler shapes. Each shape in an `hkpShapeContainer` is identified with a shape key that can be passed to the containers `getChildShape()` function.

List shapes

An hkpListShape is a simple static list of hkpShapes.

The following example illustrates using an hkpListShape to build a compound shape - in this case, a little figure that wobbles made up of two spheres and two boxes.



As you can see, the first step in creating the compound shape is to create the simple hkpShapes that form the figure.

```
hkpSphereShape* sphere = new hkpSphereShape(radius);

hkpSphereShape* sphere2 = new hkpSphereShape(radius*0.3f);

hkVector4 halfExtent(boxSize(0) * 0.5f, boxSize(1) * 0.5f, boxSize(2) * 0.5f);

hkpBoxShape* cube1 = new hkpBoxShape(halfExtent);
hkpBoxShape* cube2 = new hkpBoxShape(halfExtent);
```

Then, because the hkpListShape constructor needs a pointer to an array of hkpShapes, you also create an array to store the child shapes.

```
hkArray<hkpShape*> shapeArray;
```

Before adding the child shapes to the array, you need to reposition them to form the wobbly figure, as otherwise the boxes and spheres will all be created around the same centre. To do this, you create a new hkpTransformShape with each simple shape - the following snippet shows the creation of the wibbly's left arm. The new transform shapes are added to the array as they are created.

```
hkpTransformShape* cube2Trans = new hkpTransformShape( cube2 );

hkTransform t;
hkVector4 v(0,0,1);
hkQuaternion r(v, -0.4f);
hkVector4 trans = hkVector4(-0.9f, .7f, 0);
t.setRotation(r);
t.setTranslation( trans );
cube2Trans->setTransform( t );
shapeArray.pushBack(cube2Trans);
```

Finally, the array of shapes is used to create the hkpListShape.

```
hkpListShape* listShape = new hkpListShape(&shapeArray[0], shapeArray.getSize());
```

Convex List shapes

A convex list shape, hkpConvexListShape, is basically a list shape, which uses only convex objects as its children and uses an implicit convex hull as an extra bounding volume structure. Compared to a list shape it can have good performance, very often nearly as good as a single convex body, and lower memory consumption.

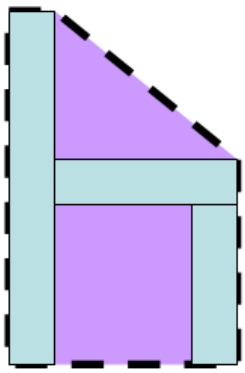
There are a number of drawbacks and limitations you must be aware of if you wish to use the convex list shape. These are:

- It cannot have hkpTransformShapes as any of its children; you must use hkpConvexTransformShape or hkpConvexTranslateShape.
- It is not available on SPU. If building for PLAYSTATION®3, the hkpListShape is recommended

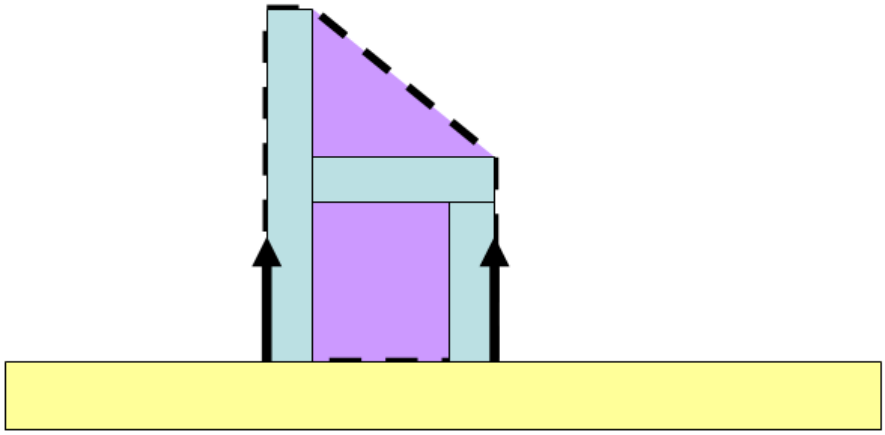
in its place.

- It can only handle child shapes with a maximum number of 256 vertices. For this reason it cannot handle cylinder shapes, i.e. you cannot have a convex list shape with a cylinder shape as a sub shape. This is because the cylinder shape implicitly defines itself to be an object with many hundreds of vertices.
- It only works with a limit of 256 child shapes. In practice you should try to keep the number of child shapes as low as possible, i.e. less than 10, for performance reasons.
- It does not work with welding. You can work around this by using a convex filter and treading the hkpConvexListShape as a list (see the last paragraph of this section).
- The radius of all child objects must be equal. For this reason you cannot combine hkSphereShapes, hkCapsuleShapes and hkConvexVerticesShapes in hkConvexListShapes.
- You cannot use collision filters to filter out sub shapes in a convex list shape. This is because the collision filter does not apply to the outer convex hull used in the convex list shape algorithms.

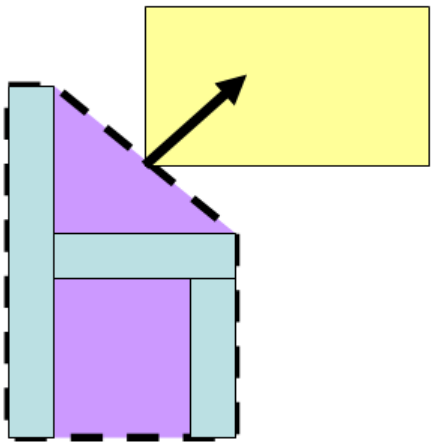
If these limitations are okay, then the hkpConvexListShape may be a good choice for your simple compound shapes. The way the hkpConvexListShape works is now illustrated. Consider the following example:



For simplicity we consider this example in 2D, but the issues all extend to 3D. This shape is composed of 3 sub shapes (the boxes that make the "h" shape). Also shown is the convex hull of these three shapes, the boundary of which is drawn as a dotted line. The convex list shape works by first considering this convex hull for collision prior to checking the sub shapes. Consider this object in collision with a box (shown below the convex list shape):

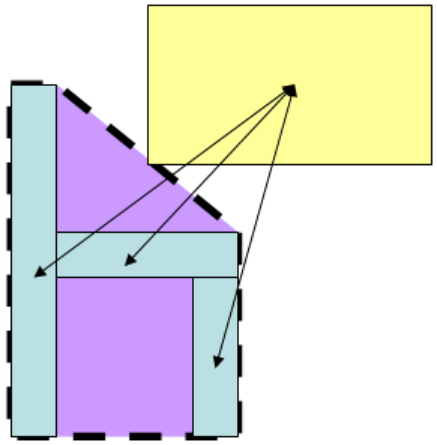


The outer convex hull is first collided with the box . The contact points with this outer convex hull are shown as 2 arrows in the diagram above. For each contact point on the outer convex hull, we can quickly determine if the contact point is a valid contact point on one of the sub shapes. If it is we do not have to check the sub shapes individually. In the above case, both contact points are valid points, so we do not need to check the sub shapes; collision with the outer convex hull and the box is all that is necessary here. However in other cases this is not true, for example:



In the above case, the contact point between the outer hull and the colliding box is not a valid contact point, as it does not lie on any of the sub shapes. This is easily detectable, and in this case the contact point is rejected, and collision detection is performed between the box and the sub-pieces of the convex list shape, as shown:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



As can also be seen from the above examples, the convex list shape performance varies depending on where it collides with another object. Effectiveness of the convex list shape will be determined by how much of the surface area of the convex hull is actually real surface area of the sub shapes. A special callback is also provided which is called, like a collision filter, whenever a new collision agent is created between a convex list shape and another object. You can implement this callback to change the properties of this collision.

This feature is however only supported by the hkpConvexListAgent (old type) and not by the hkpConvexListAgent3 ('3' at the end stands for the streamed version of the agent). To enable the feature you must not register the hkpConvexListAgent3.

Here are return values for the convex list filter and their effects are as follows:

Return Value	Notes
TREAT_CONVEX_LIST_AS_NORMAL	The collision will be processed as described above
TREAT_CONVEX_LIST_AS_LIST	The convex list shape will be treated as a list shape in this collision
TREAT_CONVEX_LIST_AS_CONVEX	The convex list shape will be treated as convex in this collision

Treating the convex list shape as convex will mean that all collisions with the bounding hull of the convex list shape will be treated as real collisions. In many cases, particularly when objects are quite fast moving, it can be very hard to detect this, and it can lead to a big performance increase. Also note that the shape id in the contact point added callback may not be correct (as the collision may not be a real collision with an actual sub piece).

This behavior is controlled by a filter which is set in the hkworldCinfo structure, the hkpConvexListFilter. If this is left at the default, the hkpDefaultConvexListFilter is created. This filter returns TREAT_CONVEX_LIST_AS_LIST for collisions between the convex list shape and landscapes, and TREAT_CONVEX_LIST_AS_NORMAL for all other collisions.

If you wish to improve performance you can create and set your own hkpConvexListFilter. For example you can return TREAT_CONVEX_LIST_AS_CONVEX for collisions between convex list shapes and fast moving debris. If you want correct welding and don't want to use the simplified convex hull, you should return TREAT_CONVEX_LIST_AS_LIST. You can also return TREAT_CONVEX_LIST_AS_NORMAL for collisions between convex list shapes and landscapes - this will give you faster collision detection, but imperfect welding (which may well be acceptable for some objects).

Transformed Shapes

An hkpTransformShape contains a child hkpShape and an additional transform for that shape. The transform is from the child shape's space to the transform shape's local space. Transform shapes are useful if you want to position child shapes correctly relative to each other when making a compound shape. You can see an example of how to create an hkpTransformShape (as part of the construction of a compound shape) in the section below on list shapes.

There are two types of transform shapes: hkpTransformShape, which requires a special agent to handle it and hkConvexTransform/hkConvexTranslate shape, which do not require a special agent to handle it, as it implements the hkpConvexShape interface itself.

Implications of using hkTransformShapes

hkTransformShapes are usually used in conjunction with hkListShapes to create compound shapes allowing you to easily position the child shapes correctly relative to one another. They may also be used to align "off-centre" implicit primitives such as an hkpBoxShape or an hkpSphereShape with a display representation having a non-zero transform. However, in order to achieve this flexibility and ease of use an extra transformation is needed which naturally incurs a greater overhead. But just how significant is this overhead and what parts of the engine are affected by it?

Using hkTransformShapes will incur a performance hit in two places:

- During Aabb updates in the broadphase for the body, and,
- During narrowphase collision detection an additional collision agent is needed (there will also be further overhead in the form of transformation concatenation). This extra collision agent will disable some of the Havok3 memory and performance optimizations.

However, using hkpConvexTransformShape or hkpConvexTranslateShape some of these overhead can be minimized:

- During Aabb updates in the broadphase for the body, which is particularly cheap for simple translations
- No extra agent is needed, just the call to getSupportingVertex is modified.

In order the quantify the potential performance loss due to these extra computations we have internally tested sample scenes consisting of many objects using hkTransformShapes. The results described below are for a test performed on the PlayStation®2 with approximately 250 rigid body - rigid body interactions. The sample scene consisted of 120 boxes arranged in three stacks of 40 boxes each which were allowed to collapse onto one another to create a large pile.

Upon measuring the timings for both the broadphase and narrowphase collision detection we found that the broadphase was 17% slower when using hkTransformShapes and the narrowphase 23% slower. The overhead for actually creating and using the additional collision agent was only about 16% of the total overhead incurred; the rest of the time was spent concatenating the transformations. Although this may seem to be quite an increase in simulation time, when all other computations performed by Havok are taken into account (such as integration, contact solving etc.) the total increase in CPU time is only about 5%. Although this is an extreme case and one unlikely to exist in your game it is worth noting that there is the potential for a noticeable performance hit.

So what should you take away from this? The key thing to remember is that an overhead is present and if possible, and performance is critical, you should try to remove the need for this. As you can see from

the above statistics almost all of the extra computations are used to concatenate the transformations and so if you can "bake-in" the transformations, such as in the case of an hkpConvexVerticesShape, and thus remove the need for the transform you should do so. If you are using implicit primitives, such as hkBoxShapes, to synchronise your display representations it is likely that the concatenation will have to be performed at some point in your code. However, if you handle the transformations yourself you may see a performance improvement as you will only have to do so *once* and not *twice* as is the case for Havok (once in the broadphase and again during narrowphase). In general, if you use hkTransformShapes sparingly throughout your scenes performance should not suffer too much. To aid with performance tuning Havok will issue the following warning whenever hkTransformShapes are encountered:

Warning : 'Collidable at address 0x01234567 has a transform shape as the root shape. This can cause a significant performance loss. To avoid getting this message compose the transform into the collidable and remove the transform shape.'

So if you feel that these objects are degrading performance you can quickly track down any occurrences within your code.

Havok provides utilities to detect and remove unnecessary transform shapes. Check the Optimizing Collision Detection section for details.

Phantom callback shapes

An hkpPhantomCallbackShape has no physical effect in a scene (that is to say its collisions do not create contacts which affect the body which owns it), but it will trigger events when other shapes penetrate it. This shape is typically the child shape of an hkpBvShape, where the bounding volume of the hkpBvShape is the hkpPhantomCallbackShape has two callbacks that you may want to implement in your own hkPhantomShape subclasses. The `phantomEnterEvent()` function is called when the phantom shape agent is created, while `phantomLeaveEvent()` is called when it is deleted.

The phantom callback shape has limited use. In general if you wish to perform user-collision detection queries you should use the hkpPhantom classes (in hkdynamics). However, if you have a rigid body to which you wish to attach a callback region, it can be convenient to add a bvshape and a phantomcallback shape to your shape hierarchy. The region will be part of the rigid body's collision detection and will automatically move around with the rigid body, as opposed to the hkpPhantom, which you must move around yourself.

Adding bounding volume information

An hkpBvShape or hkpBvTreeShape allows you to specify additional bounding volume information for a complex shape or set of shapes. This can speed up collision detection significantly as it can filter the precise collision detection by a simplified check. If the simplified check results in a non colliding state, no further checks are necessary and we can skip calling the precise collision detection. There are three versions of bounding volume information:

- The hkpBvShape shape wraps a single complex shape with a single simple bounding volume shape (user definable).
- The hkpBvTreeShape creates an optimized spacial tree for a big list of children, thereby extracting only those children which are in close proximity of the colliding body.
- The hkpConvexListShape creates an implicit convex hull around its children and uses this convex hull as a bounding volume structure. It uses some extra tricks to increase performance and should be used if possible instead of an hkpBvShape/hkpListShape combination. Note the hkpConvexListShape is not avaiable on SPU.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Bounding volume shapes

An hkpBvShape has two hkpShape members, one of which is the child shape, the other of which describes the bounding volume used for the child shape. An agent for collisions between an hkpShape S and the child shape is created only when S intersects (penetrates) the bounding volume. It is deleted if S ever becomes disjoint from the bounding volume. Thus collisions between S and the child will only be detected when S also intersects the bounding volume, as you would expect. Imagine you have a detailed car driving over a landscape. Normally a quick check with a simple box can quickly decide whether a collision with the detailed car is possible at all. So by using a box as a bounding volume shape we can speed up collision detection for our complex car as long as our box is not colliding with the landscape. If we expect our box to continuously collide, we still have to do the whole complex collision work and our bounding volume shape is pretty much useless, it just costs CPU.

This also permits the creation of "phantom" shapes where events are raised whenever another hkpShape overlaps such a shape. In this case you simply set the child shape to be an hkPhantomShape. When an hkpShape S intersects the bounding volume shape, the child shape agent will be created, which in this case is an hkpPhantomAgent, raising a `phantomEnterEvent()`. When S and the bounding volume shape become disjoint, the child shape agent is deleted, raising a `phantomLeaveEvent()`. The net effect is that events are raised whenever an hkpShape enters or leaves this type of hkpBvShape.

For example, you could use an hkpBvShape to create a phantom with a triangular bounding volume. In that case, the child shape would be an hkPhantomShape and the bounding volume shape would be an hkpTriangleShape.

The following example shows the creation of a triangular phantom area:

```
hkVector4 v0( -1, 0, 1 );
hkVector4 v1( -2, 0, 1 );
hkVector4 v2( -1, 0, 2 );

hkpTriangleShape* triangle = new hkpTriangleShape( v0, v1, v2 );
leftEyeShape = new MyPhantomCallbackShape();

shapes[1] = new hkpBvShape( triangle, leftEyeShape );
```

Whenever an hkpShape enters/leaves the space occupied by the hkpTriangleShape, a corresponding phantom event will be raised. This allows the triangle to act as a "sensor" area.

Bounding volume tree shapes

An hkpBvTreeShape adds extra bounding volume information to an hkpShapeCollection, such as an hkpExtendedMeshShape, in order to facilitate fast region queries. In fact the underlying information need not be in a tree structure, but it is more usual for it to be (for example an Aabb tree, a BSP tree), hence the naming. A bounding volume tree is very useful in situations where you need to check for collisions between a moving object and a large static geometry, such as a landscape. Before all shapes inside the collection are dispatched for collision with an hkpShape, a region query can be used to produce a smaller subset of shapes in the collection which may collide with, say, the Aabb of that shape. It may also be used to speed up other queries such as raycasts.

If you use an hkpBvTreeShape for a landscape, the shapes that make up the landscape are usually hierarchically grouped in a bounding volume tree. At every node in the tree there exists a bounding polytope, which encapsulates all of its children. The top-level bounding volume contains the entire landscape, while the nodes on the leaf levels encapsulate one geometric primitive, normally a triangle. The

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

fit of this bounding volume can be perfect (as in some Aabb trees), or can have an extra margin/tolerance built in (e.g. MOPP).

In the case of a landscape for example, instead of checking whether the moving object is colliding with each of the triangles in the landscape in turn, which would be extremely time-consuming, the bounding box of the moving object can be checked against the bounding volume tree - first, whether it is intersecting with the top-level bounding volume, then with any of its child bounding volumes, and so on until the check reaches the leaf nodes. A list of any potentially colliding triangles would then passed to the narrow phase collision detection. You can think of the bounding volume tree as a filter to the narrowphase collision detection system.

3.3.4.4 Landscapes

Meshes

Meshes in Havok Physics are generalized to contain both triangles and other hkpShapes. This generalization allows for more efficient application of bounding volume tree information (typically MOPP bounding volume information) simultaneously to both triangles and general convex shapes.

The hkpExtendedMeshShape is a shape collection that wraps an arbitrary mesh made up of multiple triangles, as well as a list of other hkpShapes. The hkpExtendedMeshShape can handle both triangle lists and triangle strips, as well as handling degenerate triangles gracefully. Note that it can not handle triangle fans directly as the striding between the vertices is not constant (vertex0 + vertex(n+1) + vertex(n+2) for the n-th triangle). As it is just a collection of triangles the mesh is always assumed to be *concave.* The groups of triangles that make up the hkpExtendedMeshShape are known as its subparts and the shape does *not* copy the data that each subpart points to. Thus the memory overhead of the mesh shape is roughly the size of each subpart structure by the number of subparts used. Each subpart is a struct containing the following information:

- (Vertex Info) a pointer to the first vertex of the triangle data
- (Vertex Info) the byte offset in memory between consecutive vertices, also known as the vertex striding
- (Vertex Info) the number of vertices in the subpart
- (Triangle Info) a pointer to the first triangle index
- (Triangle Info) whether 16 or 32 bits are used to index the triangles
- (Triangle Info) the offset between index triples in the triangle index, also known as the index striding
- (Triangle Info) the number of triangles formed by the subpart

Note that the vertex striding is between the vertices, and *not* the individual vertex (x,y,z) sub components. For example if you store all your vertices one after each other in one big float array (vertex buffer), the striding is `(3*sizeof(float))`, which is 12. If there was a 4byte (32bit) color in-between the vertices, the striding is `(sizeof(vertex)+color) == (3*sizeof(float) + 4) == 16`.

The triangle index (or index buffer) is an array used to indicate which triples of vertices form triangles - this is useful in cases where the same vertex is used more than once, and you don't want to store it multiple times. It also facilitates use of triangle strips, where adjacent sets of triangle indices overlap. A triangle strip can be thought of as a window of 3 indices that progress by one index along the index

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

array for each triangle, in which case the index striding is equal to the size of a single index element, i.e. either 2 bytes or 4 bytes (depending on whether 16bit or 32bit indices are being used). For instance, if you store 16bit indices in the a large unsigned short array, one after each other, as a triangle strip, the index striding is `(1*sizeof(unsigned short))`, which is 2. If you store your indices as 32 bits and say that they represent a triangle list (not a strip), the striding is `(3*sizeof(int))` which is 12.

Because the hkpExtendedMeshShape retreives its triangle geometry from vertex data and triangle indices rather than from hkTriangleShape instances, it must create a new hkpTriangleShape to return whenever its `getChildShape()` function is called. This shape is stored in the AllocBuffer parameter passed to the function by the caller by means of placement new. Each new hkpTriangleShape is given a user ID, which can be used to identify the triangle as the ID given to the returned hkpTriangleShape is the same as the hkpShapeKey used to query for the triangle and thus represents the triangle index in the subpart and subpart index itself.

As mentioned above the hkpExtendedMeshShape handles degenerate triangles gracefully. It does this by checking that a triangle is non-degenerate before returning it as valid triangle during shape key access. The degeneracy check used by hkpExtendedMeshShape is a utility method in hkpTriangleUtil (physics/hkcollide/hkpTriangleUtil.h). In this method a global tolerance is used: hkDefaultTriangleDegeneracyTolerance, which defaults to 1e-7. If may wish to extern this value and increase or decrease the tolerance. This can be useful if you find that degenerate triangles in your mesh geometry are being returned. This can manifest itself as a crash inside the MOPP engine during runtime. Currently hkpExtendedMeshShape is the only code that uses ::isNonDegenerate internally in the Havok Physics or Animation engines, thus it is safe to change this value. If you choose to use hkpTriangleUtil::isNonDegenerate() in your own code, it is best practise to specify the tolerance as a parameter rather than use the default (global) value.

Per triangle / per subpart material tagging.

You could use the shapekey as a key for indexing material information on mesh subparts or triangles. Alternatively, mesh shapes have builtin support for materials which is more memory efficient than a simple external mapping.

Each subpart has members `m_materialIndexBase`, `m_materialIndexStridingType` and `m_materialIndexStriding` which associate 8 or 16 bit integer indices with each triangle or for an entire subpart (by setting striding to zero). Note that the striding need not match the striding type for interleaved indices. Use the `hkpExtendedMeshShape::getMaterialIndex()` method to extract the material index from a shape key.

```
// Simple eight bit indices
          hkInt8* indices8 = ...;
          part.m_materialIndexBase = indices8;
          part.m_materialIndexStridingType = hkpExtendedMeshShape::MATERIAL_INDICES_INT8;
          part.m_materialIndexStriding = sizeof(hkInt8);
```

```
// Interleaved eight bit indices
          struct Interleaved { hkInt8 materialIndex; hkInt8 morePackedData; } * interleaved;
          part.m_materialIndexBase = interleaved;
          part.m_materialIndexStridingType = hkpExtendedMeshShape::MATERIAL_INDICES_INT8;
          part.m_materialIndexStriding = sizeof(Interleaved);
```

Each subpart also has `m_materialBase`, `m_materialStriding` and `m_numMaterials` members which refer

to the material data. Again, the material array may be interleaved. See the MeshMaterialDemo for an example of interleaving materials. Note that using `m_materialBase,` implies enabling collision filtering since the base class `hkpMeshMaterial` contains a filter value.

```
// externally stored materials - we use the materialIndex into an external array
part.m_materialBase = HK_NULL;
```

```
// Interleaved material
struct MyMaterial : public hkpMeshMaterial
{
   // inherited hkUint32 m_filterInfo for triangle collision filtering
   int m_extraData;
};
MyMaterial* interleavedMaterials;
part.m_materialBase = interleavedMaterials;
part.m_materialStriding = sizeof(MyMaterial);
part.m_numMaterials = ...;
```

MOPP bounding volume trees

hkpMoppBvTreeShape implements an hkpBvTreeShape using MOPP. A MOPP tree uses a number of special optimizations that make it more memory-efficient. Each leaf node bounding box in the MOPP tree has a "fit tolerance" that indicates how closely the box fits the enclosed shape. A default fit tolerance is provided, though you can further optimize the tree for your game needs by specifying custom fit tolerance details. You can find out more about this in the Optimizing and Tuning section.

The following example illustrates how you might use a MOPP bounding volume tree shape to create a landscape which has extra bounding volume information. As you can see, first you need to create an hkpExtendedMeshShape for the landscape's geometry. Then you create a single subpart - specifying the necessary vertex and index information - and add it to the mesh shape. In this example the mesh shape has only a single subpart, but in general an hkpExtendedMeshShape might have many subparts, corresponding to different areas of the landscape, or individual triangle strips for example.

```
m_mesh = new hkpExtendedMeshShape;

          // it is common to have a landscape with 0 convex radius (for each triangle)
          // and all moving objects with non zero radius.
          m_mesh->setRadius( 0.0f );

          {
          hkpExtendedMeshShape::TrianglesSubpart part;

          part.m_vertexBase = vertices;
          part.m_vertexStriding = sizeof(float) * 3;
          part.m_numVertices = numVertices;

          part.m_index16Base = indices;
          part.m_indexStriding = sizeof( hkUint16 );
          part.m_numTriangleShapes = numTriangles;
          part.m_stridingType = hkpExtendedMeshShape::HK_INT16_INDICES;

          m_mesh->addSubpart( part );
}
```

Next, you need to create the MOPP information for your mesh shape, using the hkpMoppUtility `buildCode()` function to build an hkpMoppCode. Here you are using the default fit tolerances, but it is important in general to set these tolerances according to the type/scale of data you are using. See the MOPP fit tolerance part of the Optimizing and Tuning Collision Detection section.

```
hkpMoppCompilerInput mci;
m_code = hkpMoppUtility::buildCode( m_mesh, mci );
```

Finally, you use the mesh shape and its code to create the hkpMoppBvTreeShape.

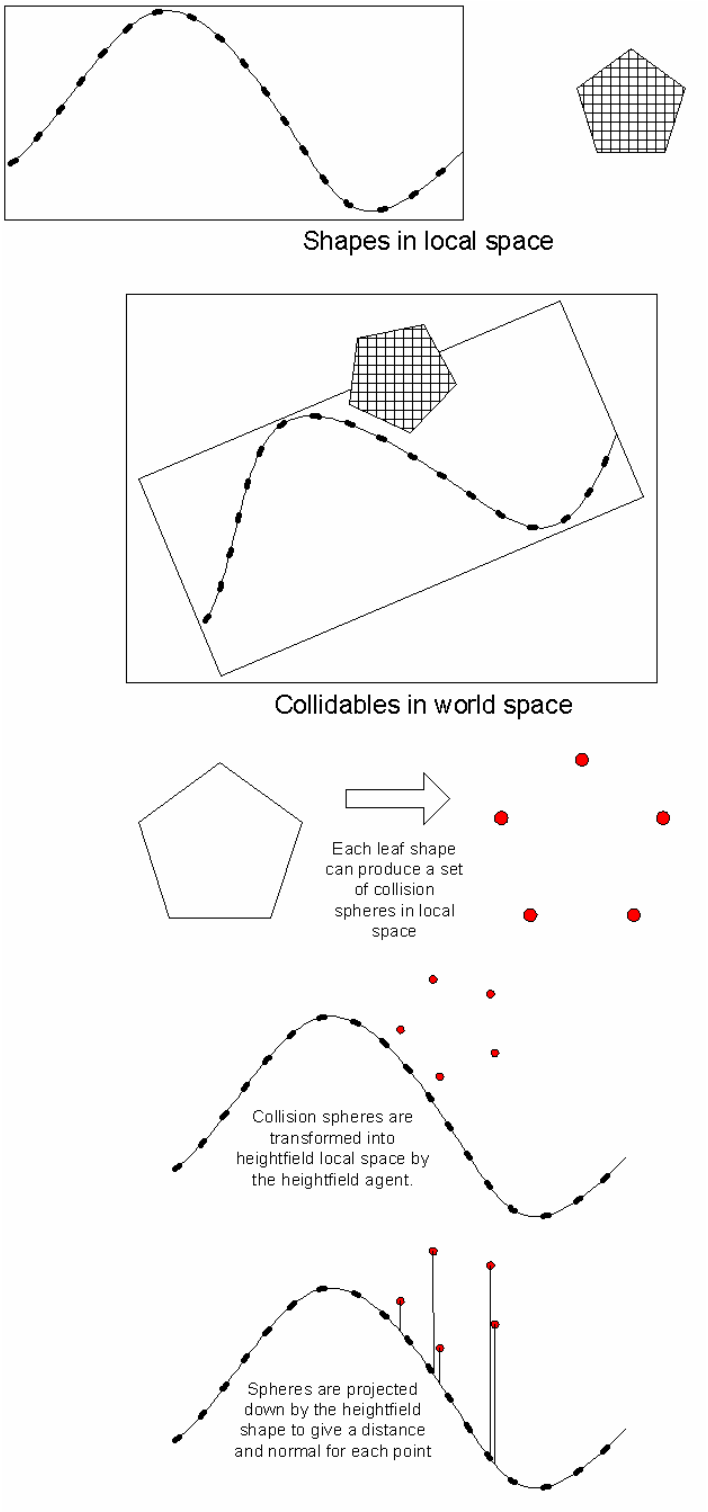
```
hkpMoppBvTreeShape* moppShape = new hkpMoppBvTreeShape(m_mesh, m_code);
```

Note that although the hkpMoppCode is not an hkReferencedObject, it is reference counted, so you can call its `removeReference()` function once you have used it to create the shape. The physics engine will then look after deleting the object for you. You can think of the hkpMoppCode as just another name for the bounding volume data. It is generally very low on memory, usually about 10 bytes per triangle in the mesh shape, and it varies depending on the parameters used in the fit requirements. This hkpMoppCode can be precomputed off line and streamed in at runtime, so see the Streaming MOPP Data section further on for more information.

3.3.4.5 Heightfields

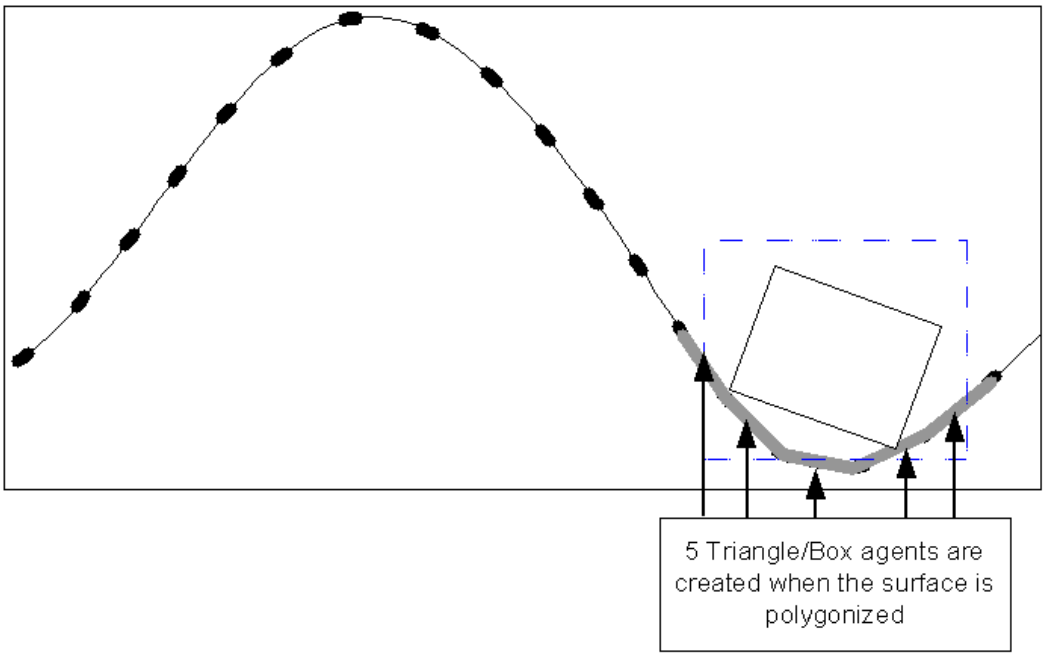
In Havok Physics, heightfields are a highly CPU and memory optimized way to do collision detection between displacement map landscapes and normal shapes. The heightfield class makes several assumptions (listed below) to keep the CPU and memory footprint required as low as possible. It is important that you understand the approximations made and their consequences before choosing to represent your terrain as a heightfield. However, if you can accept these approximations you will see clear improvements in the amount of CPU and memory required for collision detection.

The heightfield class is the base class for this surface representation. Its implementation requires that you provide ray casting support, sphere casting support (see approximations) and that you can report a set of distances for a given set of spheres. The spheres are provided using our hkpSphereRepShape interface. All leaf shapes in Havok implement this interface. For example a convex vertices shape implement this by creating a sphere with a small radius at each of the vertices of the convex vertices shape. Once you provide this implementation a single agent takes care of all collision detection.



Immediately the initial memory benefits are obvious. The static memory footprint depends on the underlying data structure used to represent the surface but is often far lower than storing a polygonal representation, e.g. A 16 bit heightfield stores a surface with just 1 byte per triangle. Also there is no bounding volume overhead that would be associated with a MOPP or octree. Also at runtime, by implementing the interface you end up with a single agent for each object that interacts with the

heightfield. If this surface were considered at a polygon level then potentially several agents would be created for each interacting object.



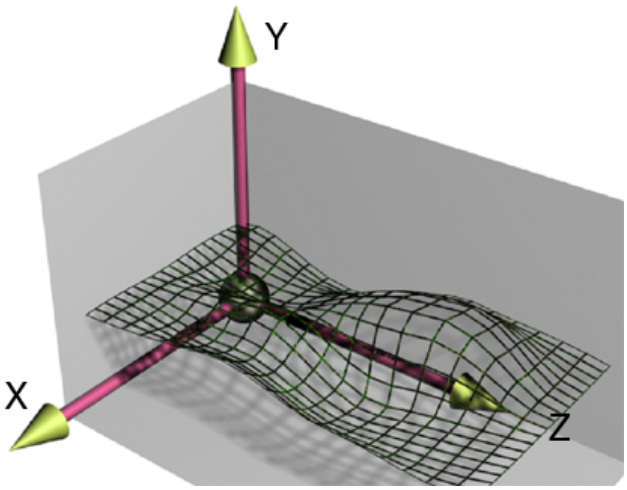
The data flow through the heightfield agent is optimized to avoid any heap allocations. The simplified intermediate sphere representation is created through the interfaces on the stack and only lives as the agent processes collisions.

On its own, this generic implementation may be useful if you have specialized surfaces in your title e.g. parametrically defined spline surfaces. E.g. The User Heightfield demo shows how to implement this generic heightfield. As you can see it does not assume a grid based displacement map. Instead it models the heightfield as a hollow sphere and does a simple sphere-sphere test in the main collideSpheres() method. However the castRay functionality is not implemented.

The most common implementation is a regularly sampled heightfield and we provide an optimized default implementation. The implementation allows you to quickly reuse displacement data that your renderer may be using. It also contains a highly optimized and exact castRay implementation that works well for both short and long raycasts by implementing a line drawing algorithm in the space of the heightfield.

Sampled Heightfields

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



The above image shows the default configuration for a sampled heightfield defined in local space. The interface requires the user implement a single method that can provide the height at each grid coordinate. To provide your own implementation you simply inherit from the hkpSampledHeightFieldShape and implement the getHeightAt() and getTriangleFlip() methods. Also forward the collideSpheres method to a templated implementation. This forwarding allows us to inline your user defined getHeightAt and getTriangleFlip functions. This significantly improves the performance for ray casting and collision generation.

Here is what a user defined heightfield class would look like (from the Sampled Heightfield demo)

```
class MySampledHeightFieldShape: public    hkpSampledHeightFieldShape
{
    public:
        MySampledHeightFieldShape( const hkpSampledHeightFieldBaseCinfo& ci): hkpSampledHeightFieldShape(
          ci)
        {
        }

            /// My simple height lookup
        HK_FORCE_INLINE hkReal getHeightAt( int x, int z ) const
        {
            return 0.0f; // return your height here...
        }

            /// this should return true if the two triangles share the edge p00-p11
            /// otherwise it should return false if the triangles share the edge p01-p10
        HK_FORCE_INLINE bool getTriangleFlip() const
        {
            return false;
        }

            /// Forward to collideSpheresImplementation
        virtual void collideSpheres( const CollideSpheresInput& input, SphereCollisionOutput* outputArray)
          const
        {
            return hkSampledHeightFieldShape_collideSpheres(*this, input, outputArray);
        }
};
```

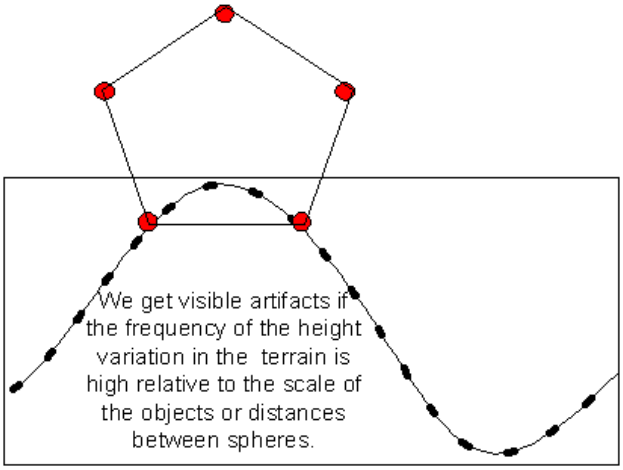
hkSampledHeightFieldShape_collideSpheres is templated on the type passed in its first argument. This means that the collide method for the heightfield can acquire the height at a given (x,z) with a quick inline call. The triangle flip is to allow for differences in the tessellation of the quads in the heightfield.

If the quad is defined as 4 points 00, 01, 11, 10 , the triangles can share either of the two opposing sets of vertices. Depending on how you render / interpret it you need to let the collision detection know via this call, so that the edge is along the correct diagonal.

To create a proper serializable heightfield you will need to add the extra couple of methods for the construction info, and that is dealt with in more detail in the Serialization section. Also you have to implement ray and sphere casting.

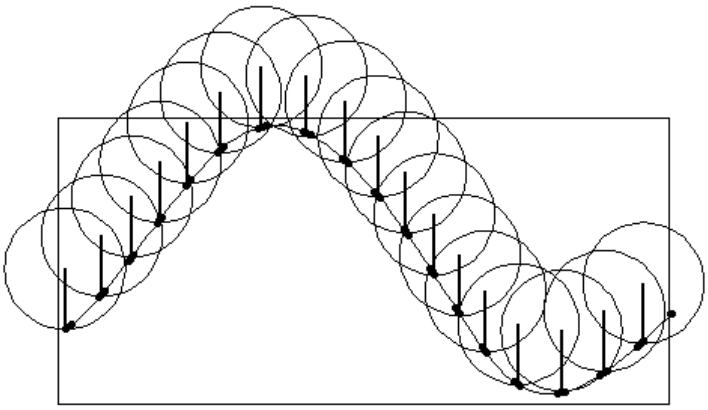
Heightfield approximations

The simplification of the incoming shape obviously does not take account of edge or face collisions. The higher the change in frequency the worse the artifact becomes. With sheer drops it is important to note that although objects collide correctly with the cliff face, they will not collide properly at the cliff edge because of the lack of edge/edge collisions. Since the simplification is done in the heightfield agent it will affect all implementations.

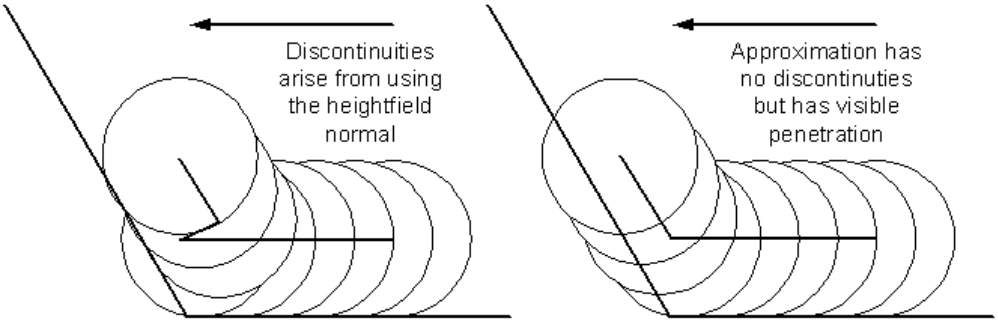


In the hkpSampledHeightFieldShape::collideSpheres implementation we project the centre of each incoming sphere down along y in local space to compute the required distance. This is then adjusted back along y by the radius of the sphere. This adjustment leads to visual artifacts as the slope of the heightfield or radius of the sphere increases. We could choose to take the normal into account when making the adjustment but this leads to more serious discontinuities in the heightfield.

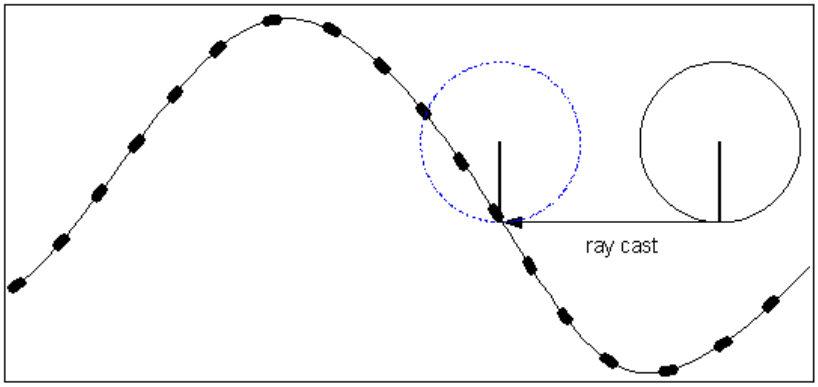
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Sphere collision approximation



Usually this approximation is only an issue for spheres and capsules (which have a sphere representation of 2 spheres) If the approximation becomes too apparent you could choose to derive new sphere and capsule shapes which return more spheres when the implement the hkSphereRep interface.



Sphere casting

When performing linear casting the heightfield agent casts the spheres for the incoming shape against the heightfield. It does this through the hkHeightfield::castSphere method. In our hkSampledHeightfieldShape implementation we approximate this sphere casting using the same approximation as above. We drop along the y axis and perform a raycast. This obviously suffers from the same artifacts as the above approximation and is probably most visible when using capsule characters with heightfields that have steep slopes. Examine the Heightfield Ray Vs Linear Cast demo to illustrate.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Triangulated Sampled Heightfields

If you wish to avail of the memory performance of heightfields (described in the section before last), but the approximations that heightfields use in collision detection (described in the last section) are not acceptable, you can simulate sampled heightfields as triangle meshes. Given a heightfield object, add the following two lines of code:

```
SampledHeightFieldShape* heightFieldShape;

// Create your heightfield here as normal...

// Wrap the heightfield in an hkpTriSampledHeightFieldCollection:
hkpTriSampledHeightFieldCollection* collection = new hkpTriSampledHeightFieldCollection( heightFieldShape
  );

// Now wrap the hkpTriSampledHeightFieldCollection in an hkpTriSampledHeightFieldBvTreeShape
hkpTriSampledHeightFieldBvTreeShape* bvTree = new hkpTriSampledHeightFieldBvTreeShape( collection );
```

Internally what is happening is that the height field is being "wrapped" in a special shape collection and a bvTree shape. These two shapes work together to present the heightfield to the collision detector as a bounding volume tree together with a triangle mesh. When an object collides with the heightfield, the bounding volume shape (hkpTriSampledHeightFieldBvTreeShape) will pull out all triangle IDs that correspond to the x and z coordinates at the area of collision. The shape collection (hkpTriSampled-HeightFieldCollection) knows how to convert these IDs into triangles. The collision detector then uses these triangles for detection with the colliding object. To see a demo of this (highlighting the difference between the heightfield and the triangulated heightfield), see "Physics/Api/Collide/Shapes/Height Field/Tri Sampled Height Field".

There is one very important performance caveat you must be aware of if using triangulated sampled heightfields. If you are using long linear shape casts across the landscape, the performance is likely to be prohibitively slow. The triangulated sampled heightfield does not use the same optimized linear cast algorithm as the sampled heightfield. Instead, it build a 2 dimensional Aabb around the linear cast and pulls out all triangles intersecting this Aabb. This is not a problem for short linear casts, so for example, using the character controller on triangulated sampled heightfield will not be a performance problem. Note also that raycasting is not a performance problem, because the triangulated sampled heightfield uses the same raycast as the heightfield (because this algorithm gives an accurate result).

In summary, you should use the standard heightfield if:

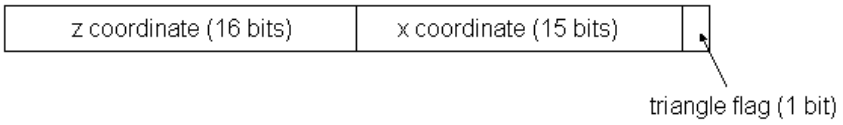
- You understand the approximations made in the collision detection (described in the previous section)
- The curvature of your heightfield is small in comparison to your objects (for example a gently undulating terrain)
- You are looking for the best performance possible from your landscape collision detection
- You only want one sided collision with your landscape - objects can not tunnel underneath the heightfield
- The edges of the landscape are not exposed - the heightfield does not perform edge collision detection.

You should use the triangulated heightfield if:

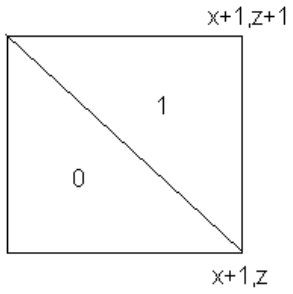
- You want exact collision detection with your landscape
- You want two sided collision with your landscape - objects can tunnel underneath the triangulated heightfield.
- You want holes in the heightfield or you want custom collision filters on parts of the heightfield - you can use the ids generated per triangle for filtering.
- You are NOT using the heightfield for long linear casts.

Heightfields, shape keys and getTriangleFlip().

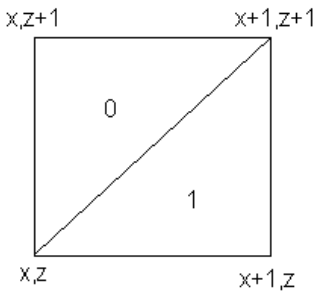
The triangles of the heightfield map to 32 bit shape keys. This mapping is used in the normal heightfield; when you perform a ray cast against the heightfield, you will get a shape key corresponding to the "triangle" of the heightfield that the ray hit. With the triangulated sampled heightfield, this shape key is used more extensively, in that you can actually use it to get an hkpTriangleShape corresponding to the triangle in question (using the normal hkpShapeCollection interface). The 32 bit shape key for a given triangle is broken into 3 components: the x-coordinate of the quad containing the triangle, the z-coordinate of the quad containing the triangle, and 1 bit flag representing which triangle in the quad it is. The lowest bit is used for the triangle flag. The next 15 bits are used for the x-coordinate, and the top 16 bits are used for the z coordinate. The key then looks like this:



There are two ways the heightfield can be triangulated, depending on the value it returns from getTriangleFlip. These are shown below. In each case, the triangles are marked with a "1" or a "0" . This will be the value of the triangle flag corresponding to that triangle.



m_triangleFlip = false



m_triangleFlip = true

3.3.4.6 Shapes and Scale

You can create several hkMeshShapes that refer to the same geometry but have different scales (set using `hkpExtendedMeshShape::setScaling()`). You cannot apply scales to the basic shapes in Havok, e.g. hkpSphereShape or hkpBoxShape, as they assume that you 'bake' the scale into the extents of the shape itself. If you are using hkpConvexVerticesShapes, you need to "bake in" a scaling factor into the vertices before you pass them to the constructor of hkpConvexVerticesShape.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.3.4.7 Shapes and Sharing

You can reduce the runtime memory footprint required to run a simulation by sharing shapes whenever possible. Each shape and all associated information is stored in the shape's local space. This means you can have a single instance of a shape and share it among several collidables.

Case1: 500 crates but only 1 shape. The simplest example of this is when we have a game object that appears many times in a level e.g. an ammo clip. We create one shape to represent the surface of the ammo clip and then reuse this shape every time it appears in the level by creating an hkpRigidBody for the clip and setting the shape pointer in the construction info to our single surface representation.

Case 2: Instanced geometry. As with the example above we can create a single instance of a piece of fixed geometry, e.g. a telegraph pole, and reuse this in several places in our level. The MOPP collision detection information need only be created and stored once.

Case 3: Compound shapes . When we have shapes that represent basic building blocks or components we will often want to put these together to build a final, more complex shape e.g. consider building a house with shapes for a door, a window, a roof and a wall. We can put this together using a list shape and use transform shapes to offset multiple instances of the same component shape in the final construction. The final building with 2 doors, 4 walls, a roof and 10 windows might contain 17 transform shapes. Each of these transform shapes would refer to one of the original 4 components (door, roof, window, wall) together with an offset transform to position and orient the component. It is worth noting that the hkTransformShapes do have a performance cost associated with them due to the extra transformation incurred, so where possible bake the transform into the shape data.

Havok provides utilities to detect and share duplicated shapes. Check the Optimizing Collision Detection section for details.

3.3.4.8 Legacy Shapes

This section lists shapes which are deprecated and marked for removal from future Havok Physics releases. Use of these shapes is no longer recommended.

- hkpMultiSphereShape
- hkpConvexListShape
- hkpFastMeshShape
- hkpConvexPieceMeshShape
- hkpMeshShape
- hkpStorageMeshShape

3.3.5 Broadphase Collision Detection

The job of the broadphase is to quickly find pairs of Aabbs that are intersecting, and thus to identify pairs of objects that require narrowphase collision detection. Objects that can be processed by the broadphase must have an hkpBroadPhaseHandle, which is used as an id for each object by the broadphase. For a

Havok object, this is its hkpCollidable, as hkpCollidable has a member hkpBroadPhaseHandle. Both hkEntities and hkPhantoms own an hkpCollidable.

The hkBroadphase class provides functions for adding, removing, and updating objects in the broadphase. These are used by the Havok world when you add, remove, or move entities in the simulation.

A single hkBroadphase implementation, hkp3AxisSweep, is currently provided with Havok.

3.3.5.1 Configuring the broadphase

For performance reasons the default hkp3AxisSweep converts all values to 15 bit integer. This gives roughly a 100% speed increase and 50% memory decrease compared to using 32 bit floats. As a result as part of setting up your Havok world, you need to specify the area dealt with by the broadphase, known as the broadphase size. You do this using the hkpWorld's hkpWorldCinfo. The size is specified by an Aabb given by maximum and minimum extents. This need not be centered around the origin. For convenience however, you can set it to be a cube centered on the world space origin, using the helper method `hkpWorldCinfo::setBroadPhaseWorldSize()`. Here you must specify the length of the side of the cube in world units The default size is 1000.

```
// Set broadphase size by setting Aabb extents
hkpWorldCinfo info;
info.m_broadPhaseWorldMin.set(-50.0f, -50.0f, -50.0f);
info.m_broadPhaseWorldMax.set(50.0f, 50.0f, 50.0f);
```

Warning:

Because the broadphase world has a definite size, it is possible to position objects so that they are *outside* the broadphase. *This must be avoided*. If this happens, the broadphase considers any Aabbs outside its limits to be projected onto the broadphase Aabb (it clips the object Aabb to the broadphase Aabb, in effect reducing the 3-D overlap test to a 2-D one, producing "spurious" overlaps). This incurs a significant performance cost as the relevant pairs are all passed to the narrowphase, greatly increasing the amount of work the narrowphase has to do. It also produces incorrect callbacks from other broadphase listeners, and from broadphase raycasts.

For example, if 100 objects are placed along the x-axis *all outside the broadphase, in a line but separated* then all their Aabbs will be clipped to have the same min and max x-extents. That is, their Aabbs each collapse to a rectangle, so the broadphase considers them to be overlapping each other on the X-axis. If the objects' Aabbs actually overlap on the other two axes then all 100 objects will be considered to overlap with each other. This will cause (100*99)/2 = 4950 agents to be created, even though most of these will return no collisions when the narrowphase is processed! Thus collision detection will still work, but be *massively inefficient* both in speed and memory consumption.

To avoid this, ensure that your broadphase world size is appropriate for your scene, and remember the broadphase limits when moving your objects. It is also important that you *do not set the broadphase size too large*, as internally the broadphase has to quantize the whole range. Too large a broadphase size leads to reduced accuracy in the overlap tests resulting in more narrowphase tests than needed.

To monitor objects leaving the broadphase during run-time, you can use the hkpBroadPhaseBorder class in the hkutilities/collide folder. This has a `maxPositionExceededCallback()` function that is called whenever an object leaves the broadphase Aabb. By default, it removes the entity from the world - you may implement your own version.

3.3.6 Narrowphase Collision Detection

3.3.6.1 Collision Agents

In narrowphase collision detection, hkCollisionAgents are used to determine if pairs of objects are colliding. Each hkpCollisionAgent implementation handles collision detection for a different combination of hkpShape types - so, for instance, there is an agent that deals with collisions between spheres, another that deals with collisions between transform shapes and other shapes, and so on. The appropriate collision agent for each pair is chosen by the hkpCollisionDispatcher based on the type information in their shapes, and the current set of agents registered by the user.

As you know, each Havok shape can have one or more alternate types. The actual type indicates a preferred interpretation for the surface, but one or more alternate types can be used for alternatives. For example, an hkpSphereShape will have a type of HK_SPHERE_SHAPE and an alternate type of HK_CONVEX_SHAPE.

At physics initialization, the collision dispatcher will receive all necessary information, namely all shape types and their alternate types followed by a list of agents with the types they can handle. The result is compiled in a 2 dimensional table.

At runtime the collision dispatcher simply gets the types of the two objects involved, looks up the pre-built agent creation table and creates the agent. If both objects are spheres, it tries to find an hkpSphereSphereAgent, which uses an algorithm designed specifically to deal with collisions between spheres. If no such agent has been registered, it will check probably find a more general agent on this basis - for example, an hkpGskConvexConvexAgent for two convex shapes. We call agents that deal with pairs of object types like this *binary agents*.

If there is no specific agent that deals with both the objects' types, we typically register *unary agents* with the dispatcher. These deal with collisions between one shape type and any other shape type. We also call these midphase agents, as they require the subsequent use of binary agents, and often reduce the amount of work the narrowphase has to do. So, for example, if a box collides with an bounding volume tree shape, an hkpBvTreeAgent is used to choose which child shape(s) the box is potentially colliding with. The appropriate binary agents can then be used to check for collisions between these shapes and the box. This allows hierarchical shapes to be collided using a corresponding hierarchy of agents.

These are the unary agents provided by Havok:

Agent	Type
hkpBvTreeAgent	HK_SHAPE_BV_TREE
hkpBvAgent	HK_SHAPE_BV
hkpListAgent	HK_SHAPE_LIST
hkpShapeCollectionAgent	HK_SHAPE_COLLECTION
hkpPhantomAgent	HK_SHAPE_PHANTOM
hkpTransformAgent	HK_SHAPE_TRANSFORM

Some of the agents listed above also have their Agent3 versions, which fully use the agent streaming technology.

There is one more binary midphase agent that specializes in colliding two shapes of the types HK_SHAPE_COLLECTION or HK_SHAPE_BV_TREE. This agent dispatches both collections at the same time, resulting in significant memory savings.

Agent	TypeA and TypeB
hkpCollectionCollectionAgent3	HK_SHAPE_COLLECTION or HK_SHAPE_LIST or HK_SHAPE_BV_TREE

If no appropriate agent is registered for a given shape type, it just uses the default collision agent. This is the hkpNullAgent, which does nothing except producing a warning.

A collision agent continues to exist as long as the pair of objects overlap in the broadphase. Once the objects no longer overlap in the broadphase the agent is destroyed. All "child" agents of any unary agents in the corresponding agent hierarchy are recursively destroyed.

3.3.6.2 Streamed Agents

One drawback of using agents is that they allocate a fixed size of memory per colliding pair. That means extra memory fragmentation and extra cache misses as well as wasting memory. Havok3 introduces a new type of agent: the streamed agent. A streamed agent is an agent which sits in a memory stream. This has a number of advantages:

- Each agent can change its size every frame.
- Streamed memory access is faster than random memory access.
- All memory allocations occur in 512 byte blocks. This greatly reduces memory fragmentation, however it can add some extra unused memory.

Havok only streams some of the most important binary agents:

- hkPredGskAgent handling discrete and continuous convex convex collisions
- hkPredGskCylinderAgent extends hkPredGskAgent for cylinders
- hkpCapsuleTriangleAgent typically handling ragdolls hitting landscapes
- hkpBoxBoxAgent for the box-box special case

Streamed agents cannot be called asynchronously, therefore Havok does not use streamed agents for hkCachingShapePhantoms

3.3.6.3 hkpCollisionDispatcher

In order to use the collision detector you must register the all shape and their alternate shapes as well as all collision agents with the collision dispatcher. Each hkpCollisionAgent (with the exception of the hkpNullAgent) has a `registerAgent( )` function that you can use to do this. You pass the function a reference to the dispatcher.

Alternatively, you can use the utility function `registerAllAgents()`, provided in the hkpAgentRegisterUtil class. This registers all the agents provided by Havok Physics with the dispatcher, as in the following snippet.

```
hkpAgentRegisterUtil::registerAllAgents( m_world->getCollisionDispatcher() );
```

You should use this utility. As well as registering all the collision agents, it also registers all the alternate shape types.

Note, however, that if you create your own collision agent type, it will not be registered by `registerAllAgents()`. You will need to use the agent's `registerAgent()` function instead. Note also, that `registerAllAgents()` will probably register more agents than necessary unless you are using all possible types of shapes, and this will increase your code size. Thus it is probably better to register only those agents required. If two hkpShapes overlap in the broadphase and no agent can be found, the engine will WARN the first time it creates an hkpNullAgent to let you know that collisions will be missed. The order that you register agents matters as well as the registration overwrites the previous entry in the agent lookup table, so the last agent registered for a type pair is the one that is called.

Advanced Notes:

The job of the collision dispatcher is to provide pairwise information about the different types in the collision detection system. It holds a number of dispatch tables:

- For each pair of shape types it holds a function to create the appropriate collision agent.
- For each pair of shape types it holds a list of static functions, which allow you to statically query the collision detector for this pair, e.g. check for sphere sphere penetration.
- For each shape type it holds a list of alternate shape types. E.g. a HK_SHAPE_BOX is also a convex shape. Therefore the collision dispatcher knows that the box shape has the alternate shape type convex.
- For each pair of hkpCollidable::getQualityType() it returns an index to an hkpCollisionQualityInfo. This type is used t o define how much CPU is invested for keeping two objects from penetrating.
- For each pair of hkpCollidable::getResponseType() it creates the appropriate contact manager for that interaction. A contact manager is used to link the dynamics lib with the collide lib.

In addition to that it performs a number of extra tasks:

- It automatically decides which agent to create when you request a discrete or a continuous agent. (Note: only the GSK agent is continuous).
- It has a helper function initCollisionQualityInfo() to initialize all major tolerances used by the continuous physics
- Given a pair, it knows whether it makes sense to exchange the two objects to get a better agent. For example Havok has a sphere-box agent. If you pass a box and a sphere to the dispatcher, the dispatcher knows that it is better to swap the box and the sphere to use the sphere-box agent.

Note that the order of registering agents is important. Agents registered later will override existing entries. Therefore getting the order right of how to register agents can be very tricky. To help you, the collision dispatcher has a debugPrintTable() function to print the current contents of the table.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.3.7 Collision filtering

You can use filters to disable or enable collisions between objects. If collisions are disabled between two objects, no collision agents are created for the pair, and they can move into or through each other. This can be useful, for example, if you have a number of objects constrained together that are likely to bump against each other frequently - let's say a ragdoll figure's upper arm and lower arm. If you disable collisions between the upper and lower arm, rather than repeatedly checking for collisions between them - which can be very time consuming - the simulation just uses the constraints to keep them apart and ignores any small interpenetrations that might occur.

Filtering happens at several stages of the collision detection pipeline:

- The first stage is the collision dispatcher. It ignores collisions between objects if the dispatched hkpCollisionQualityInfo is invalid. This is the default for collisions between fixed and keyframed objects.
- Then it tries to call the user implementation of the hkpCollidableCollidableFilter
- Then it tries to generate an agent for this pair. If it cannot find an agent, this collision is ignored.

The hkpCollisionFilter class implements four basic filter base classes: The four base classes broaden the range of filtering possible and allow for increased flexibility by splitting the filtering into four distinct types: hkpCollidableCollidableFilter, hkpShapeCollectionFilter, hkpRayShapeCollectionFilter and hkpRayCollidableFilter.

So now hkpCollisionFilter contains four overloaded isCollisionFilterEnabled(...) methods, one corresponding to each base class. You may use this class to create your own collision filter, or you may simply choose to use our default implementation, the hkpGroupFilter which will be described later. The four base classes allow for the following types of filtering:

- Collidable vs collidable (used to filter the results of the broadphase)
- Shape collection vs shape (used to filter the results of a shape collection, or bvTreeShape)
- Shape collection vs raycast (used to filter a ray against the children of a shape collection)
- Collidable vs raycast (used to filter a ray against the results of the broadphase. This is only used if you call hkpWorld::castRay())

You specify that you want to use a particular collision filter using the hkpWorld's `setCollisionFilter()` function.

```
m_world->setCollisionFilter( filter );
```

Warning:

Changing either the filter or the filter info will not necessarily result in the immediate creation or deletion of contacts, because that information can be cached by the system. In order to cause the creation or deletion of appropriate contacts, an update of this cached information must be forced based on the new filtering information. More details are provided in the Update Collision Filter section.

An example of this is adding a collision filter to the world *after* you have added filtered entities to the world, as an entity is considered "active" by the collision detection system as soon as it is added. This means that contact points will potentially already have been created for entities that are deemed to be colliding. If you then add a collision filter on the next line, you will have to force an update of the cached information in order to bring the system in line with the new filtering rules.

3.3.7.1 Implementation Details of Filtering

An hkpCollidable has an hkpTypedBroadPhaseHandle embedded. This handle has a hkUint32 member variable m_collisionFilterInfo, which - when considering a pair of hkCollidables - is generally used by the particular hkpCollisionFilter implementation attached to the hkpWorld, to determine if collision detection is enabled for that pair of hkpCollidables.

As a trivial example of an implementation of collision filtering (and, indeed, the creation of a user-defined collision filter), consider the User Collision Filter demo in the Havok Physics demo framework. This creates two fixed boxes, and a moving box (see picture below). A value of 0 is assigned to the m_collisionFilterInfo of the top-most (moving) box, and the bottom-most (fixed) box. A value of 1 is assigned to the (fixed) box in-between. The isCollisionEnabled method of the user-defined collision filter created in the demo simply checks the m_collisionFilterInfo of the pair of hkCollidables for equality; if the values are equal, collision is enabled. If not, collision is disabled.

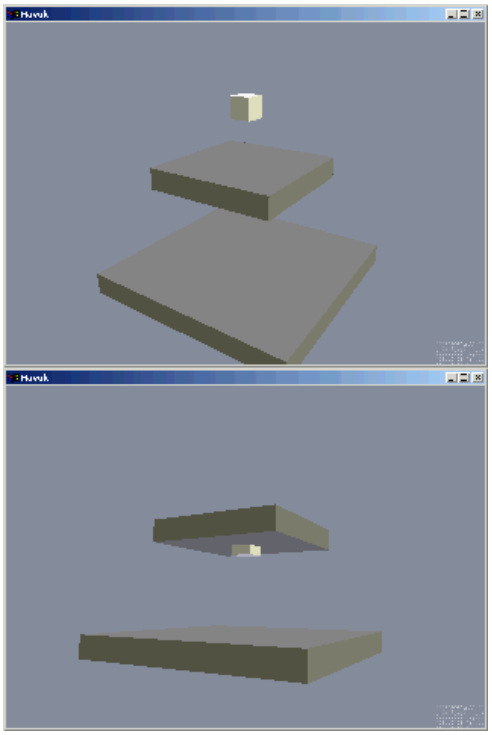


Figure 3.41: The User Collision Filter demo: before, and after

Hence, as can be seen, the moving box falls through the first fixed box, but collides with the second.

Phantoms and Collision Filters

From a filter point of view, phantoms and rigid bodies are identical. Both use a collidable.

3.3.7.2 Updating Collision Filters

You can update the collision filter info for an hkpCollidable (or, indeed, for the entire system). If you just set a collision filter info in the collidable, this info will only be used for new overlaps. That means existing collisions will not be affected and the desired effect will not happen. Therefore whenever you change the filter, you have to inform the physics to reevaluate some of its internal cached data. Also depending on the type of filter change you make, you can tell the system exactly what to verify. The API calls to update the collision filter are:

```
void updateCollisionFilterOnEntity( hkpEntity* entity,
                                    hkpUpdateCollisionFilterOnEntityMode updateMode,
                                    hkpUpdateCollectionFilterMode updateShapeCollectionFilter );
```

```
void updateCollisionFilterOnPhantom( hkpPhantom* phantom,
                                     hkpUpdateCollectionFilterMode updateShapeCollectionFilter );
```

```
void updateCollisionFilterOnWorld(  hkpUpdateCollisionFilterOnWorldMode updateMode,
                                    hkpUpdateCollectionFilterMode updateShapeCollectionFilter);
```

The parameters in the functions can be set as described below.

1: To do a broadphase region query and search for pairs, which have been filtered out before, but are now enabled:

Parameter	Value
hkpUpdateCollisionFilterOnEntityMode	HK_UPDATE_FILTER_ON_ENTITY_FULL_CHECK
hkpUpdateCollisionFilterOnWorldMode	HK_UPDATE_FILTER_ON_WORLD_FULL_CHECK

2: To not look for new pairs. Use this value if your filter change only removed existing collisions. This is way faster than rechecking the broadphase:

Parameter	Value
hkpUpdateCollisionFilterOnEntityMode	HK_UPDATE_FILTER_ON_ENTITY_DISABLE_ENTITY_ENTITY_COLLISIONS_ONLY
hkpUpdateCollisionFilterOnWorldMode	HK_UPDATE_FILTER_ON_WORLD_DISABLE_ENTITY_ENTITY_COLLISIONS_ONLY

3: To check for filter changes for shape collections. You have to use this if you changed the filter between an object and the subparts of a landscape:

Parameter	Value
hkpUpdateCollectionFilterMode	HK_UPDATE_COLLECTION_FILTER_PROCESS_SHAPE_COLLECTIONS

4: To not check changed sub landscape filtering:

Parameter	Value
hkpUpdateCollectionFilterMode	HK_UPDATE_COLLECTION_FILTER_IGNORE_SHAPE_COLLECTIONS

As changing a collision filter is a major operation for the engine, this cannot be performed as long as the engine is locking its internal data structures. Therefore calls to updateCollisionFilter might be delayed by the engine automatically until the next safe time to execute them. This typically happens if you want to change the filter in a callback.

In some cases you may not know in advance whether two objects should collide or not. You will have to decide at the point of collision whether they actually do collide, and if not, the collision must be filtered. For example, if a bullet hits an object it might collide with it or break the object and pass through. You may want to make this decision at the instant the bullet collides.

In such cases changing the collision filter does not help (the effects are delayed). However you should use the ability to reject contact points in the contactPointAdded callback. You can see an example of this in the code for hkSmoothOperator (hkutilities/collide) - but essentially, you achieve the desired behaviour by setting the event.m_status member of the hkpContactPointAddedEvent to be HK_CONTACT_POINT_REJECT, and allowing the engine to proceed. The contact will be ignored until the next simulation step (so, you could safely change the collision filter info of the object you no longer want to collide with, or delete it outright, when the simulation step has completed).

Note:

Changing the collision filter using the above functions is a slow operation. All the collision agents are destroyed for the entity (or phantom) in question, and then the broadphase is re-queried, and the changes propagated back through the system to recreate the agents with the new filter.

Calling updateCollisionFilterOnWorld is nearly equivalent in work to removing all your objects from the world, and re-inserting them. *Do not call this at runtime!*

3.3.7.3 The hkpGroupFilter

Havok Physics provides a powerful, fixed-function collision filter as part of the hkdynamics library. It is called the hkpGroupFilter, and as its name implies it separates hkCollidables into layers and groups - the interactions of which are determined by rules set in the filter.

For each of the four collision filters mentioned above (that is, the four base classes for hkpCollisionFilter from which hkpGroupFilter itself derives), the filter extracts 32 bit number from the two objects being queried for a collision call.

The hkpGroupFilter allows you to assign a distinct "collision layer" and "system group" to each hkpCollidable. There are 32 collision layers (numbering 0-31), and 65536 system groups (numbering 0-65535). Recalling that the m_collisionFilterInfo member is 32-bits long, the lower 5 bits represent the layer number, and the upper 16 bits represent the system group number in the current implementation. As can be seen, this scheme enforces a distinction in both the layer, and the group (i.e., an hkpCollidable can be a member of only one of each).

The hkpGroupFilter reports that two hkCollidables can collide if:

- collision between both of their layers is enabled, and
- they do not share the same system group (except for system group 0)

By default, the filter enables collision between all of the layers. Also collisions between layer 0 and any other layer are always enabled and can not be disabled.

3.3.7.4 An Example hkpGroupFilter

For an example group filter setup, you should check out <hkcollide/filter/group/hkpGroupFilterSetup.h>, which is thought to be representative of a typical game scenario: the filter is set up with a static landscape layer, dynamic object/player object/ai object layers, a "keyframed" (i.e., non-physical) object layer and normal debris/fast debris layers. One could derive the collision rules from looking at the code, but basically: collisions between keyframed objects and other keyframed objects are disabled, collisions between keyframed objects and static objects are disabled and collisions between fast debris objects and other fast debris objects are disabled.

It would be prudent at this juncture to point out the addressing convention for collision layers in the calls to enable/disableCollisionsBetween and enable/disableCollisionsUsingBitfield - the layers number 0 to 31, and this is the value that should be used in both the m_collisionFilterInfo and the enable/disableCollisionsBetween calls. When addressing the layers using a bit field, however, the bit representing group 0 is the zero'th bit (i.e. 0x0000000b or 1 << 0). In other words, the following calls should be equivalent (I wish, using the example setup, to also disable collisions between the keyframe layer and the fast debris layer):

```
groupFilter->disableCollisionsBetween( hkpGroupFilterSetup::LAYER_KEYFRAME, hkpGroupFilterSetup::
  LAYER_FAST_DEBRIS );
```

or:

```
groupFilter->disableCollisionsUsingBitfield( 1 << hkpGroupFilterSetup::LAYER_KEYFRAME, 1 <<
  hkpGroupFilterSetup::LAYER_FAST_DEBRIS );
```

The advantage to using the enable/disableCollisionsUsingBitfield calls is that several collision layer rules can be defined at once.

3.3.7.5 System groups and the hkpGroupFilter

The rationale behind system groups in the hkpGroupFilter is presented with a use case. Our game has two ragdolls, and a chunk of static mesh for them to fall on. We want the ragdolls to collide with the landscape, and with other ragdolls - but we don't want the body parts of an individual ragdoll to collide with each other (for the purposes of example, we assume the constraint limits will enforce the disjointedness that we want - but, that's also a good route to follow generally, because it's less computationally expensive). We can achieve the desired behaviour by assigning the same (non-zero) system group number to the pieces of an individual ragdoll. So, how does that work?

Recalling that the uppermost 16 bits of the m_collisionFilterInfo for an hkpCollidable represent the system group the hkpCollidable is a member of, the bitwise XOR (bits set in either A or B that are not set in both) of the m_collisionFilterInfo's of two hkCollidables, with the lower 16 bits masked off, will be 0 if both hkCollidables inhabit the same system group. Now, if the upper 16 bits of either (we only need to

check one) is non-zero, then both hkCollidables inhabit the same non-zero system group, and hence they do not collide.

Back to our use case. We want to assign separate systems groups to all of the bodies in each ragdoll. We can obtain a free system group number by calling:

```
int systemGroupForOneRagdoll = groupqueryBroadPhaseForNewPairsFilter->getNewSystemGroup();
```

which increments the internal system group counter of the hkpGroupFilter, and returns it. When setting the m_collisionFilterInfo member of the hkpRigidBodyCinfo for a piece of our ragdoll, we make the following call to the broadphase:

```
info.m_collisionFilterInfo = hkpGroupFilter::calcFilterInfo( NUMBER_OF_RAGDOLL_LAYER,
  systemGroupForOneRagdoll );
```

which transparently shifts the system group number up by 16 bits and ORs with the number of the ragdoll layer to give us our m_collisionFilterInfo.

3.3.8 Removing child shapes from compound shapes

hkpMoppBvTreeShape and hkpListShape have the ability to disable their child shapes. This is an inexpensive way to simluate destruction and doesn't cause a performance peak that you'd see when replacing the entire shape of a body.

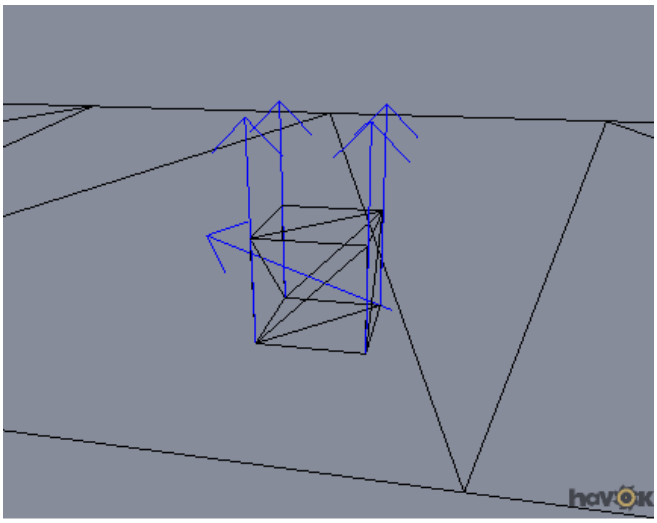
The hkpBreakOffPartsUtil::removeKeysFromListShape() function demonstrates the proper way to remove child shapes (or hkpShapeKeys) from a hkpListShape, and also from a wrapping hkpMoppBvTreeShape, if there's one. See the demo in Demos/Physics/Api/Dynamics/RigidBodies/BreakOffParts/BreakOffPartsDemo for an example of how the util is used.

Note that in order for hkpShapeKey removal to work:

- compound dynamic bodies must have an hkpListShape optionally wrapped with an hkpMoppBvTreeShape
- compound fixed bodies may have any of the collections shapes, but it must be wrapped with an hkpMoppBvTreeShape

3.3.9 Welding

"Welding" is the term used in Havok to refer to the problem of objects bouncing as they move from colliding with one object to colliding with another. This problem is illustrated by the following picture:



This picture shows a cube sitting on a triangle mesh. It is in contact with one triangle with 4 contact points (the 4 arrows pointing up). However it is also deemed to be in contact with the triangle to the right of this (as it is within the collision tolerance. The contact point between the cube and this triangle is shown by the arrow pointing to the left. As the box slides from left to right across the mesh this contact point will cause the box to "hop" to the new triangle. In versions prior to 4.5, this problem was resolved by a run time analysis of all contact points between the box and the mesh, rejecting any contact points that would cause this problem. In 4.5 and later versions, a new solution has been provided which is faster and more robust. It involves storing information in the mesh regarding the edges of each of the triangles. This information is used to "snap" the unwanted collision normals to the desired direction. This information adds an extra 2 bytes per triangle to mesh the storage requirement. To aid this new algorithm you should specify the winding of your triangles, assuming you can guarantee that all triangles in your mesh are stored with the same winding. If you cannot guarantee the winding of your triangles you can specify that your triangles are "two sided", by passing the parameter WeldingType::WELDING_TYPE_TWO_SIDED in the function below. However this is not recommended as it can lead to additional penetrations when objects collide with the mesh. See the api demos in "Physics/Api/Collide/Welding" for more info.

All Havok supplied meshes have a funcion that computes the welding information for the mesh. You must first build a mopp around the mesh shape and then call the following function:

```
void hkpExtendedMeshShape::computeWeldingInfo( const hkpMoppBvTreeShape* mopp, hkpWeldingUtility::
  WeldingType weldingType );
```

The weldingType parameter should be WeldingType::WELDING_TYPE_ANTICLOCKWISE if all triangles in your mesh when viewed from "outside" the mesh are wound in the anti-clockwise direction. This function builds the information and stores it in the mesh shape. If you have your own mesh shape you must build and store your own welding info and return the correct info in the triangles you create in getChildShape(). The hkTriangle constructor takes a welding info and a welding type for this purpose. To build the welding info for your mesh you must call the following function for each triangle (specified by the shapeKey parameter) in your mesh. The hkUint16 returned must be stored and set when the mesh creates requested triangles. See the implementation of any Havok mesh shape for an example of this.

```
static hkUint16 HK_CALL hkpMeshWeldingUtility::calcWeldingInfoForTriangle( hkpShapeKey shapeKey, const
  hkpBvTreeShape* moppShape);
```

It is recommended you use this welding as it is significantly more robust than the pre 4.5 welding algorithm. Also the pre-4.5 welding algorithm does not run on SPUs, so to enable welding on PLAYSTATION®3 you must use the new welding functionality. However the new welding currently only works on triangle meshes, so if you have "meshes" of higher order convex primitives and you need welding between them you need to re-enable the pre-4.5 welding. This can be done via the flag hkpWorldCinfo::m_enableDeprecatedWelding.

3.3.10 Optimizing and Tuning Collision Detection

Havok Physics provides you with a number of ways of optimizing the collision detection process. These allow you to "tweak" how collision detection works at the shape, world, and MOPP levels. You may need to experiment with combinations of these optimizations to find the most suitable solution for your needs.

3.3.10.1 Reducing the number of shapes

Decreasing the number of shapes used to represent objects will improve the collision detection process, by reducing memory usage (which may lead to performance improvements) as well as reducing the time required to traverse shape hierarchies.

Two utilities are provided in the hkutilities library that can be used to reduce the number of shapes and the shape hierarchy depth for one or multiple rigid bodies. They are CPU intensive so they are normally used offline while processing assets (the *Optimize Shape Hierarchies* filter in the Havok Content Tools uses these two utilities).

- The `hkpTransformCollapseUtil` utility class will attempt to remove unnecessary transform shapes by collapsing the transformation into the child shape (modifying its vertices for example). It will also replace `hkpTransformShape` objects with `hkpConvexTransformShape` or `hkConvexTransformTranslate` objects (which are faster) if the child shape is convex.
- The `hkpShapeSharingUtil` utility class will attempt to detect equivalent, duplicated, shapes in one or multiple rigid bodies and replace them by references to the same shape.

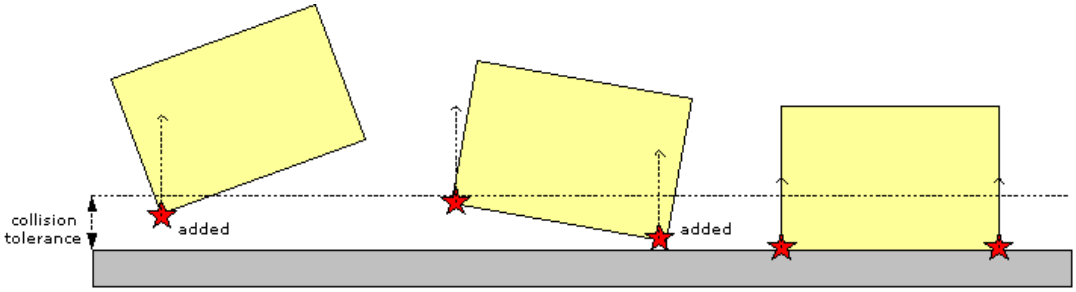
3.3.10.2 Collision tolerance

You can set a global collision tolerance for the entire Havok world by assigning a value to `m_collisionTolerance` in your hkpWorldCinfo. This value is specified in world units. By default, the world has a small collision tolerance of 0.1.

To see how collision tolerance works, let's say you have a collision tolerance of d. Each time the shortest distance between a pair of objects overlapping in the broadphase becomes less than d, the narrowphase collision agent for those objects creates a contact point or points for them (some agents create more than one contact point at a time). This information is then processed by the solver in each step until the

distance between the corresponding points on the objects exceeds the collision tolerance again, and the contact point is removed from the simulation.

The following illustration shows a box falling onto a table in a world with a large collision tolerance (to make things easier to see). As you can see, the first contact point is added as a corner of the box falls within the collision tolerance. As the box falls further, the first contact point is maintained and a new contact point is added. Finally, the box actually comes to rest on the table, with zero distance between the two hkpShapes. All contact points created are maintained by the agent until such time as they move outside the collision tolerance of the table again (if this occurs) in which case they are deleted by the agent. Also, should the Aabbs of the box and table become disjoint, this will result in deletion of the agent itself, hence automatic deletion of all contacts associated with that agent.

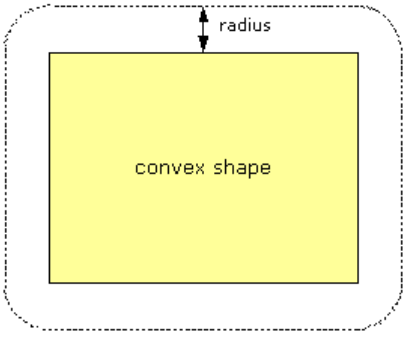


Having a non-zero collision tolerance can greatly help with two performance-related issues. Firstly, it can sometimes be useful for the system to pick up potential collisions just before they actually happen - when the distance between the objects is still greater than zero, as in our example. For fast-moving objects, this allows the collision solver to prevent interpenetration as early as possible.

It can also be useful for the system to maintain collision information even when the objects are slightly separated. For instance, when an object is sliding across or settling on another object, this allows the system to maintain the same manifold rather than having to create new contact points in each simulation step.

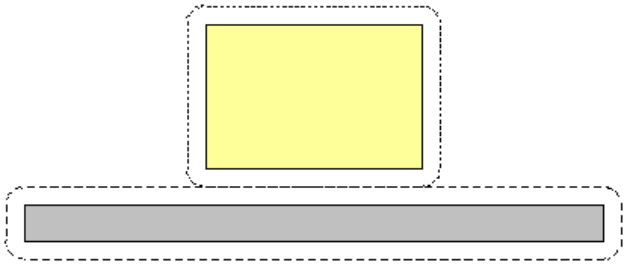
3.3.10.3 Convex shape radius

As you know, the hkpConvexShape class has a radius member that allows you to add an extra "shell" to any convex shape, such as a sphere or box. By default, an hkpConvexShape's radius value is just above zero (0.05). If a shape has a non-zero radius value, a shell with that radius is added around the shape.

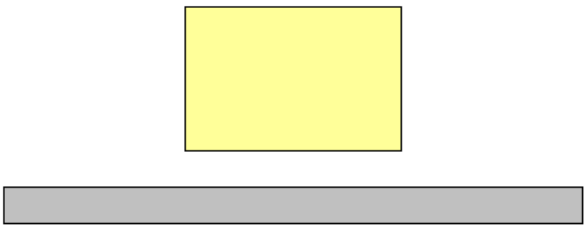


Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

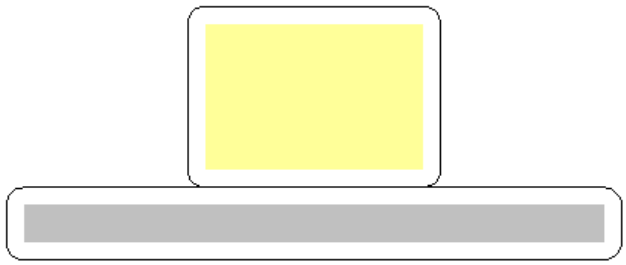
This shell is used as the shape's surface for collision detection purposes. The simulation will try to ensure that the distance between this shell and other objects is always more than zero, or in other words that the distance between the original convex shape and other objects is always more than the combined radii of the objects.



Adding a shell affects where the surface of the object is in the simulation, and hence where the object will come to rest. For instance, if a box with a shell comes to rest on a table, there may appear to be a visible gap between the box and the table, as in the next illustration.



However, as far as the collision solver is concerned, the two objects are in direct contact, as shown below.



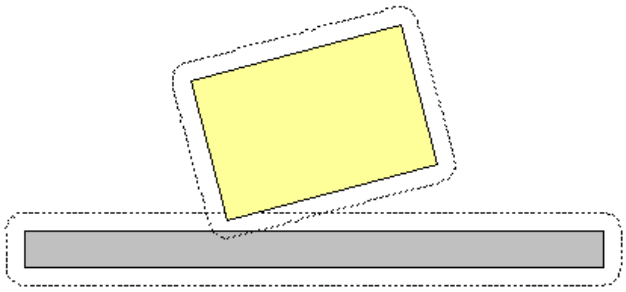
So why add a radius? Adding a radius to a shape can improve performance. The core convex-convex collision detection algorithm is fast when shapes are not interpenetrating, and slower when they are. Adding a radius makes it less likely that the shapes themselves will interpenetrate, thus reducing the likelihood of the slower algorithm being used. The shell is thus faster in situations where there is a risk of shapes interpenetrating - for instance, when an object is settling or sliding on a surface, when there is a stack of objects, or when many objects are jostling together.

Raycasting against objects with the radius will cast against that expanded surface if possible too, currently hkBoxShapes and hkCylinderShapes do, and hkConvexVerticesShapes assume the plane equations given to the shape are already expanded by radius. Raycasting against triangles does not raycast against the

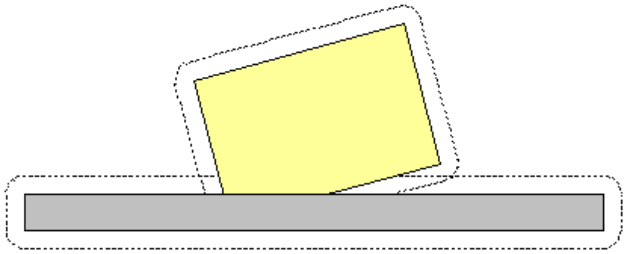
expanded form as it is computationaly prohibative, but as triangle based objects are normaly landscapes that can have zero radius, it is not normally an issue. The reason why landscapes usually have zero radius is that any two interacting bodies should have some radius between them, so all moving objects should, but the stationary landcape is obviously optional.

At any time you can view the convex radius of an object using the hkpConvexRadiusViewer in the VDB and/or the demo framework.

If a small shell interpenetration occurs, as shown in the next diagram, the system will work to restore the gap, the fast algorithm will still be used, and, as shown, the box will still not have appeared to penetrate the table.



Only if the "unshelled" shapes actually interpenetrate is the slower algorithm used, as shown in the following diagram.



Note:
Unpredictable behavior can sometimes result from using hkConvexVerticesShapes with a very small radius value. If this happens, try increasing the radius.

Visual Implications of Convex Radius

Collision detection thus appears to happen at the shell radius distance away from the object surface. The visual implications of this is that your graphical versions will hover above surfaces and bullets will hit of invisible barriers. Not exactly what you want, especially if you have shadows, and neither is setting the radius to zero and thus using more expensive algorithms more often. What you should do is create objects that such that the surface of the display version (your textured, high poly version) is the same as the physical version plus the radius. You can do this in your tool chain easily by creating the shape in the same dimensions as the display version and then calling hkpShapeShrinker::shrinkByConvexRadius(hkpShape* s) (found in hctUtilities/collide). It will return a new shape if one was created. It will return null if no new shape was required as the shape was able to be altered in place or did not have a radius. The function is recursive. Note that this is not meant to be done at runtime, but in the tool chain and

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

preprocess stages. When hkConvexVerticesShapes shrink, the vertices get shrunk by using the planes equations which get their plane distance reduced and then the planes' points of intersection are calculated and used as the new vertices. The new vertices and the original planes are used to create a new shape, thus it assumes that the plane equations of the non-shrunk shapes are on the normal surface, not the expanded one and that it has plane equations to start with.

3.3.10.4 Preventing Bullet Through Paper

Bullet-through-paper occurs when an object has a velocity such that it completely passes through another object within a single time step. The effect may also be witnessed if one object travels over 50% of the way through another object as the collision resolution algorithm will separate the pair by finding the shortest distance needed to resolve the interpenetration and pushing one of the objects in this direction. If this happens to be out of the backface of an object then this is where the interpenetrating body will emerge.

However Havok3 has solved this problem. Simply set the hkpEntity::getCollidable()->setCollidableType() to enable a higher quality. See the continuous physics section for more details.

3.3.10.5 Other tuning issues

A badly-configured collision detection broadphase can have serious performance implications. See the BroadPhase Configuration section for more details.

3.3.11 Offline generation of data

The hkinternal library contains some routines that are most appropriately used to generate data offline. For example, instead of running the convex hull generator at runtime in a game, the output geometry of the convex hull generator can be saved to file and read in when needed. This library can still be linked with and used at runtime but since it contains some very complex (and hence bulky) algorithms it has a large code footprint. The library contains the following:

3.3.11.1 Convex Hull Generation

The convex hull generator takes as input a set of vertices in 3 space and returns a set of triangles (an hkGeometry) that forms a closed convex hull and associated plane equations.

We use a convex hull algorithm in several places:

- Generation of plane equations for convex vertices shapes. These are used for ray-casting.
- Generation of visualisation code for display of convex vertices shapes (in the visual debugger)
- Generation of a hull for calculating an inertia tensor for convex vertices shapes.
- Generation of a convex hull as a utility in our modelling tools plugin / exporting geometries for convex vertices shapes

The algorithm used (rHull) has the following performance considerations (in terms of memory usage and speed):

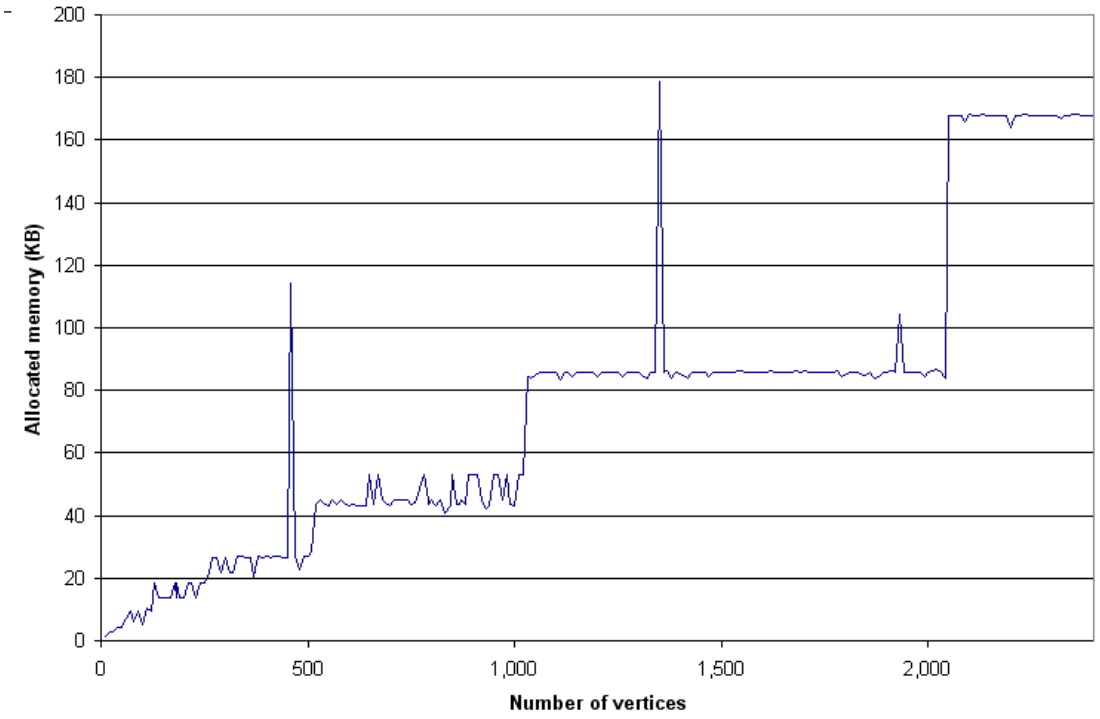


Figure 3.42: rHull memory usage

The spikes in the graph are due to the way the algorithm "searches" for the best convex hull and to the way hkArrays behave - for certain input sets, the algorithm makes allocations that double the current size of the arrays being used.

The convex hull algorithm in hkpreprocess (rHull, invoked by calling `hkGeometryUtil::createConvexGeometry`) was developed to be a more robust alternative to qHull. These two algorithms are compared against a brute-force algorithm, which works by simply generating plane equations for all tuples of vertices and checking which side of these plane equations every other vertex is on. These timings were run on a P4 1.6GHz with 512MB RAM.

Number of Vertices	Timings (milliseconds)		
	rHull	qHull	Brute-force
10	0.5	2.2	0.1
20	1.8	2.3	1.4
40	5.2	3.7	20.0
80	18.7	10.1	313.0
100	21.2	10.2	734.0
200	79.4	25.5	-
1000	1196.0	hangs	-

Table 3.3: Timing comparisons of convex hull algorithms

In general it is recommended to generate the convex hull offline. However, if you take into consideration the above timing and memory implications, there is nothing to stop you from linking with the hkpreprocess library and using the convex hull generation at runtime in your game, for example for the autogeneration of fractured debris (using small numbers of vertices).

3.3.11.2 Streaming MOPP data

Once you have calculated MOPP data for a landscape, it's possible to stream it to a file. This file can then be loaded at runtime to avoid the overhead of recomputation, which can be costly if the MOPP is optimised for runtime speed of collision detection. MOPP data is platform-independent, so you can, for example, use PC-generated MOPP data on a console. The internal representation for the MOPP data is a byte code format.

The hkpMoppBvTreeShape contains a pointer to an hkpMoppCode object. This is a serializable referenced object, and can be serialized normally using the hkserialize infrastructure. This allows you to write the hkpMoppCode into a platform dependent binary.

However, you can also hkpMoppCodeStreamer, provided as part of the hkutilities library. This will write out a MOPP to a platform independent binary file.

The following example from the Mopp Code Streaming demo shows you how to do this.

```
hkpMoppCompilerInput mci;
code = hkpMoppUtility::buildCode( meshShape , mci);

// Now write out to file
{
    // Create an output archive
    hkOArchive outputArchive(moppFilename);
    if( outputArchive.isOk() )
    {
        // Write MOPP data out.
        hkpMoppCodeStreamer::writeMoppCodeToArchive(code, outputArchive);
    }
}
```

To read in the data, you also use the hkpMoppCodeStreamer:

```
{
    // Create an input archive
    hkIArchive inputArchive(moppFilename);
    if( inpugArchive.isOk() )
    {
        // Read MOPP data in
        code = hkpMoppCodeStreamer::readMoppCodeFromArchive(inputArchive);
    }
}
```

It is up to you to ensure any "streamed-in" MOPP data files are in sync with the hkpMoppBvTreeShape you assign them to (originally build the code from), otherwise collision detection will fail. It may be advisable to "hash" the shape into a checksum, and write this as additional data into the file, to avoid any such conflicts.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.3.11.3 MOPP fit tolerance

When using a MOPP tree, you can customize the tolerance (called the fit tolerance) that describes how closely the bounding volumes at the leaf nodes (and the internal nodes) approximate the ideal bounding volume. You do this using an hkpMoppCompilerInput, which you pass to the hkpMoppUtility `buildCode()` function as part of building the MOPP.

Using the hkpMoppCompilerInput, you can choose which becomes the most highly optimised - the size of the tree, or how closely the leaf nodes bound the triangles. The effect of changing the fit tolerance is as follows:

- The smaller the fit tolerance, the less triangles are passed to the narrowphase collision detector and the less CPU is needed.
- However, the smaller the fit tolerance of the bounding volumes, the more memory is needed to store the bounding volume information.

For a detailed description of the parameters of the MOPP fit tolerance and background information on the MOPP please consult the hkpMoppCompilerInput section in the Reference Manual.

3.3.11.4 Case Study: Deciding how to represent your large scene

What you probably want to do is to split your scene into a single static mesh chunk per grid square, according to your memory requirements, so first off lets make a few assumptions about the style of the landscape. Lets say that the world is already segmented for display purposes (at least into triangle lists with bounding spheres). Each segment is mainly composed of static meshes but also contains some (probably deactivated) environmental objects like trashcans, crates etc. The main reason for segmenting the world then is to reduce the upper memory requirement for the physics and hopefully reduce the CPU requirement for collision detection (because the meshes are smaller). By segmenting the world we want to avoid any issues with physical objects straddling segments and any extra CPU / memory overhead for dynamically instantiating or removing a segment.

The main physics memory drain for the scene will depend on how you represent your static meshes. Havok Physics will need to be able to access the triangles for the static geometry and some bounding information which speeds up our collision detection with large meshes. This bounding information (in the case of MOPP) is extremely efficient and takes between 10 and 12 bytes per triangle. It can be entirely precomputed and is stored in a platform independent binary format (the hkpMoppCode). The memory requirement for the actual triangles will depend on your internal mesh representation. If you want to share visual and physical collision representations then you can (with the hkpExtendedMeshShape) and Havok will require no extra memory for accessing the triangles.

This has to make sense in the context of each geometry though, as there is no point in actually sending a 2000 poly representation of a fire hydrant into the physics collision detection system when a cylinder will do just as well. A good rule of thumb for sharing graphical and physical meshes is that the average triangle size in the mesh should be large or roughly the same size as the dynamic objects it interacts with.

If your game design or character controller requires that you have a separate collision mesh for your scene then obviously Havok will have to be able to access those triangles at run time. We store the mesh bounding information in a tree structure in memory. This means that is generally has O(log N)

performance in the number of triangles stored in it e.g. a query on 512 triangles only takes 1/8 more CPU than a query on 256 triangles. Because of this kind of trade off it is generally better to store fewer meshes with more triangles than lots of meshes with a small number of triangles each. In worst cases an object will have to travel down several of the smaller trees, making it worse than the worst case performance for the single tree.

With regard to simulation islands, Havok Physics automatically creates, partitions and merges islands. Dynamic objects belong to the same island if they share a constraint or action, or if they are in contact (as it is a constraint as well). So if you have three separate piles of objects in the world all colliding on a single static mesh, Havok will automatically separate these into three islands for simulation. By allowing Havok to handle the islands you never have to worry about the case where an object straddles two segments, and essentially is in two different islands at the same time.

So as we recommend generally doing things in this order:

- Throw the entire level into the physics system, let it calculate everything at run time and get boxes, crates etc. rolling and bouncing around.

- Decide on the collision representation for the scene: will it be a separate model or will it be the display mesh?

- Decide on the tool chain for the collision representation: most of the Havok related information can be baked in an offline preprocessing step.

- Estimate your memory budget for the physics engine and decide on an appropriate segment sizes, as a first pass dynamically create the geometry and bounding information at run time and check out the memory requirements.

- Later add in the post processing steps that dynamically stream in the bounding and geometry info. See the hkpMoppCode streaming example above.

3.3.12 Getting Collision Information

There are a number of ways to get information from the collision detection system. The two most important ones are intersecting the information flow from the collision detector to the dynamics using the hkpCollisionListener interface. The other is to do asynchronous queries to the collision detector using the hkpWorld or hkpPhantom classes.

3.3.12.1 Collision Callbacks

On every frame, collision agents create, maintain, and delete contact points. By implementing the hkpCollisionListener interface and adding your implementation as an hkpCollisionListener to the world or entity, you will be able to receive callbacks when one of those events are happening. This allows you to query information about the contact being processed or even to manipulate the data before resolution occurs. As a contact point is typically reused over several frames, only one contactPointAddedCallback event is raised, followed by a single contactPointConfirmedCallback() and several contactProcessCallback events until contactPointRemovedCallback event is raised:

contactPointAddedCallback(): This is called after a contact point is found between two bodies during narrowphase collision detection. It is passed an hkpContactPointAddedEvent struct. There are two possible types of hkContactPointAddedEvents: hkpToiPointAddedEvent and hkpManifoldPointAddedEvent. This struct provides detailed information about the event, including: the entity that fired the contact, the two hkCollidables involved, the contact point (including its position and collision normal), the relative velocity at the point of contact, and the contact point's hkpContactPointProperties. You can use it to get or modify the contact point information - for instance, you could override the friction and restitution values in the hkpContactPointProperties. In addition you can stop the contact point being added to the solver at all.

Note: When continuous physics is enabled, contact points are created optimistically, and may be changed or removed as a result of a TOI event. To prevent receiving spurious callbacks, you may wish to use contactPointConfirmedCallback() instead.

contactPointConfirmedCallback(): This is called when a contact point that has been added is confirmed to actually occur. For a discete simulation, the number of added and confirmed contact points will always be the same (unless an added point is explicitly rejected by the user). For a continuous simulation, however, some added contact points may be rejected and not confirmed. For examples of how this can occur, see the section on Continuous Physics or the "Confirmed Vs. Added" demo. It is recommended that important game events be triggered by a contactPointConfirmedCallback() instead of contactPointAddedCallback(). For more information on the order of callbacks for different contact point refer to the next subsection.

contactProcessCallback(): This callback is called after narrowphase collision detection but before the results (hkpProcessCollisionOutput) are passed to the solver. The callback is passed an hkpContactProcessEvent, this includes the hkpProcessCollisionOutput itself as well as the hkCollidables, and allows you to access or change any information in the result.

Note: When continuous physics is enabled there may be multiple contactProcessCallbacks per object per frame, because contacts can be processed before they are confirmed during TOI solving. See the Continuous Physics chapter for more details.

contactPointRemovedCallback(): This is called just before a contact point is removed, and is passed an hkpContactPointRemovedEvent struct. This includes the entities involved in the event and the entity that triggered the callback.

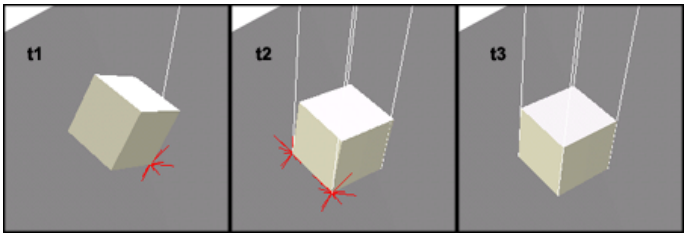


Figure 3.43: Using listeners to detect both added collision points and collisions (taken from the Collision Events demo)

In the demo shown above, one collision listener callback is used to detect when new collision points are added to the system and a second collision listener callback is used to detect collisions. The former is achieved using a contactPointAddedCallback(), the latter with a contactProcessCallback(). In this demo the contactPointAddedCallback() function has been customized to draw a red star at the location of the added point and the contactProcessCallback() is used to draw a white line from the collision contact point along the normal of the collision again. Some of the code for this demo is explained in the following section.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

In addition to reading this section, don't forget to look at the relevant demo code. You can see listeners being used in the Collision Events and User Collision Response demos, and also in the Fountain example.

Order of callbacks triggered for PSI and TOI contact points

A most common PSI contact point triggers callbacks in the following order:

- added callback – triggered when the contact point is generated by the system. This is the place to add custom data to the contact point.
- process callback – triggered after each time a contact point is processed in collision detection. This is the place to modify contact point data for custom response. Rarely there's more than one process callback before the first confirmed callback – this happens if the point is touched during a Toi processing.
- confirmed callback (called on the next frame) – confirms that the contact point has entered the constraints solver and actually had an effect in the simulation.
- zero or more process callbacks – the contact response might bounce bodies apart, in which case the contact point may disappear immediately. If it doesn't we get more processed callbacks.
- removed callbacks – called when the contact point is removed from simluation.

This changes when there are TOI collisions detected in simluation. A TOI collision may cause some PSI contact points to be removed before the next simluation frame. In this case we only get added, processed, and removed callbacks. Lack of the confirmed callback means that the contact point was never actually simulated.

TOI contact points the same callbacks as PSI points, except they only have one process callback.. And, like PSI points, they not always are confirmed. TOI contact point callbacks are triggered in the following order:

- added callback – when the TOI contact point is detected
- process callback – when TOI is processed incollisoin detection.
- confirmed callback – when TOI is solved by simulation and has an effect. TOI's are solved in chronological sequence according to their Time-of-Impact. As the result some later TOI's may be removed and never confirmed.
- removed callback – called when the TOI is removed from the system.

Process contact callback delay

You can use an entity's `setProcessContactCallbackDelay()` method to specify how often you would like an entity to send hkContactProcessEvents to collision listeners. A value of 0 means that `contactProcessCallback()` is called in every simulation step, whereas a value of 4 means that the callback is called every 5th step. The default is 65536, i.e. (for practical purposes) once for the first collision only, until the bodies separate to outside the collision tolerance.

Interface differences

There are a few differences between the interface of contactPointAddedCallback() and that of contactPointConfirmedCallback().

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Field	Contact Point Added Event	Contact Point Confirmed Event	Description
hkpCdBody& m_bodyA	yes	no	the hkpCdBody that was used (for entity A) to generate the contact point
hkpCdBody& m_bodyB	yes	no	the hkpCdBody that was used (for entity B) to generate the contact point
hkpCollidable& m_collidableA	no	yes	The first entity's hkpCollidable. Note that this is not the "leaf" hkpCdBody (as is the case for the hkpContactPointAddedEvent).
hkpCollidable& m_collidableB	no	yes	The second entity's hkpCollidable.
hkpEntity* m_callbackFiredFrom	yes	yes	the entity this collision listener was added to (or HK_NULL if the collision listener was added to the world)
hkContactPoint* m_contactPoint	yes	yes	the contact point. This cannot be changed by the user - it needs to be done in process callback
hkContactPointMaterial* m_contactPointMaterial	yes	yes	the contact point properties, including friction and restitution
hkReal m_projectedVelocity	yes	yes	the relative velocity of the objects at the point of contact projected onto the collision normal.
hkpContactPointAccept m_status	yes	no	whether the contact point should actually be added to the constraint solver or not (overridable by the user)
Type m_type	yes	yes	indicates whether the contact point is a TOI or manifold point
hkpToiPointAddedEvent& asToiEvent()	yes	no	gives safe access to the additional properties of a TOI contact event
hkpManifoldPointAddedEvent& asManifoldEvent()	yes	no	gives safe access to the additional properties of a normal collision point
hkpDynamicsContactMgr& m_internalContactMgr	yes	no	the contact manager
hkBool isToi()	no	yes	whether or not the contact point is a TOI contact point

Table 3.4: Interface Differences

These differences may be addressed in a future release of Havok.

Usage of Extended User Data in hkpContactPointProperties

hkpContactPointProperties can store additional user data of `hkUint32` type . The size of the data block can vary and depends on the setting of the colliding `hkpRigidBodies`. Use the `hkpEntity::m_numUserDatasInContactPointProperties` member variable to specify the number of extended user data slots for that body.

In the `contactPointAdded`, `contactPointConfirmed`, and `contactPointProcessed` callbacks you can query `hkpContactPointProperties` for number of extened user datas and retrieve the data. Note that for TOI events the sum of extended user data slots for both colliding bodies is limited by the `HK_NUM_EXTENDED_USER_DATAS_IN_TOI_EVENT` define.

The extended user data is always initialized with the `hkpShapeKey` hierarchy. The first user data for a given body stores the `hkpShapeKey` of the colliding leaf hkpShape. The second user data stores the `hkpShapeKey` of its parent. And so on, until we run out of the extended user data space or we end at the root shape, for which -1 is written in the user data slot.

The extended user datas are expected to store the `hkpShapeKey` hierarchy when you use the `hkpBreakOffPartsUtil`. Otherwise you're free to overwrite them with your own data.

Please refer to the `Physics/Api/Collide/CollisionCallbacks/ExtendedUserData` demo to see how extended user data is accessed in each of the callbacks for both Psi and Toi contact points.

Using a collision listener

You can see an example of using an entity collision listener in the Collision Events demo. The listener's `contactPointAddedCallback()` draws each new contact point for the entity on-screen as a little red star, getting the position from the contact point provided in the hkpContactPointAddedEvent.

```
void MyCollisionListener::contactPointAddedCallback( hkpContactPointAddedEvent& event )
{
    // draw the contact point as a little red star
    const hkVector4& start = event.m_contactPoint->getPosition();
    for ( int i = 0; i < 20; i++ )
    {
        hkVector4 dir( hkMath::sin( i * 1.0f ), hkMath::cos( i * 1.0f ), hkMath::sin(i * 5.0f ) );
        dir.setMul4(0.3f, dir);

        hkVector4 end;
        end.setAdd4(start, dir);
        HK_DISPLAY_LINE(start, end, hkColor::RED);
    }
}
```

Its `contactProcessCallback()` function also draws collision information on-screen - in this case, the collision normal each time a contact is processed by the solver. The listener finds out where to draw the line using the position and normal of each contact point in the hkpProcessCollisionOutput.

```
void MyCollisionListener::contactProcessCallback( hkpContactProcessEvent& event )
{
    // draw the contact points and normals in white
    hkpProcessCollisionOutput& result = event.m_collisionResult;
    for (int i = 0; i < result.m_contactPoints.getSize(); i++ )
    {
        const hkVector4& start = result.m_contactPoints[i].m_contact.getPosition();
        hkVector4 normal = result.m_contactPoints[i].m_contact.getNormal();

        // For ease of display only, we'll always draw the normal "up" (it points from entity 'B'
        // to entity 'A', but the order of A,B is arbitrary) so that we can see it. Thus, if it's
        // pointing "down", flip its direction (and scale), only for display.
        if(normal(1) < 0.0f)
        {
            normal.setMul4(-5.0f, normal);
        }
        else
        {
            normal.setMul4(5.0f, normal);
        }

        hkVector4 end;
        end.setAdd4( start, normal );

        HK_DISPLAY_LINE(start, end, hkColor::WHITE);
    }
}
```

Because you want this callback to be called in every step these callbacks are notified when hkConstraints are deleted, or added the relevant entity's process contact callback delay must be set to zero.

```
boxRigidBody->setProcessContactCallbackDelay(0);
```

You can see the listener in action if you run the Collision Events demo, and can view and change the complete code for the demo in the `Demos\Physics\Api\Collide\CollisionCallbacks\CollisionEvents` directory of your Havok installation.

Two very important notes:

- To optimize cache efficiency every entity has a m_processContactCallbackDelay member. This allows dropping calls to the hkpCollisionListener::contactProcessCallback. If you want to receive this callback every frame, you have to set m_processContactCallbackDelay to 0 for the entity to be examined.
- In Havok3 continuous physics, contact points are optimistically created for the end of the timestep. If time of impact event happens before the end of the timestep, this contact point will be deleted. That means that contact points will be created even if the objects never actually come into contact. Example: A bullet hitting a book with many pages. As the collision detector checks the collision between the bullet and each page separately, contact points between the bullet and inner pages of the book are created. Of course the bullet will hit the book cover and bounce off, never hitting any of the inner pages, so these "phantom" contact points are never actually simulated.

Let's examine the case where we want to identify some user-level info for the object involved in the resolution. We get a contact point added callback. There's a void* user data field in the contact point properties that we can use to store our user-level info. The two collidables involved in the collision are passed into this event. We use them to determine which user data is appropriate. This field is passed during the callback:

```
hkpContactPointAddedEvent::m_contactPointProperties
```

and we simply initialize it with our user data. Then, when in the contact process callback, we can look up the object type by calling:

```
hkpDynamicsContactMgr::getContactPointProperties( contactPtId );
```

It's easy to get the id's from within the callback. The contact process callback contains an hkCollisionResult which has a list of contact points, each of which has its own id.

Normals in Havok Physics

There are two common instances where you'll get back a normal from Havok; from an hk`pCollisionListener` or a `addClosestPointCallback`.

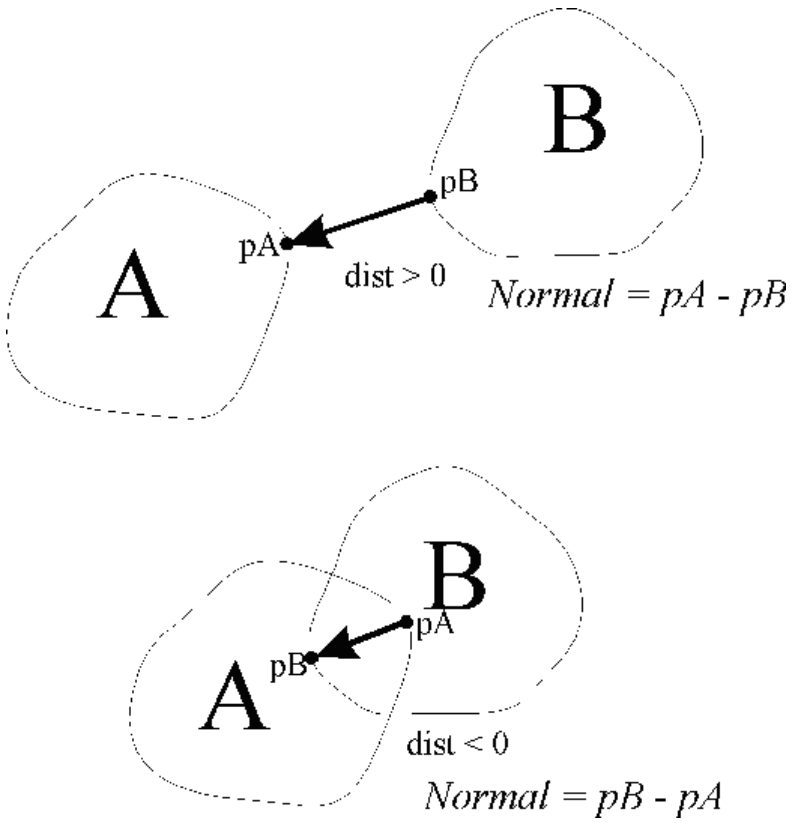


Figure 3.44: Normals for separating and penetrating objects

In Havok Physics, normals always "point" from B to A. i.e. moving A along the direction of the normal always moves the objects "further away" (or apart), as can be seen in the diagram above.

This ensures normals change continuously (no discontinuities, direction doesn't suddenly "flip") when moving from separating to penetrating cases.

However, normals are calculated from B to A when objects are separated, but normals are calculated from A to B when objects are penetrating (since pA is now "inside" B, and pB is "inside" A, hence direction pA-pB is reversed), which gives us the following possible 'gotcha':

Note:
norm(pA, pB) is not equal to pA-pB. This is only true in the separating case. For penetrating case, norm(pA, pB) = pB-pA

The Collidable Pointers

An hkpCdBody objects are returned in contact point callbacks. hkpCollidable inherits from hkpCdBody. An entity has an hkpCollidable.

In order traverse down a collision agent hierarchy, we need two collidables. Actually, most of the time the agents do not need the full information stored in an hkpCollidable. Therefore it uses a reduced version, the hkpCdBody. It still can access all the information of the original hkpCollidable by traversing up the m_parent hierarchy. The hkpCdBody which has no parent is actually an hkpCollidable. You can use hkpCdBody::getRootCollidable() to the hkpCollidable at the root level.

The hkpCdBody pointers returned in the callbacks are the hkpCdBody objects that where actually colliding at the lowest level. This may mean that you have the pointer to a temporary hkpCdBody, and not the actual collidable in any of your entities. Here is how this can happen and how to get the proper hkpCollidable pointer.

When dealing with a hierarchy of hkpShapes, for example an hkpBvTreeShape on top of an hkMoppShape colliding with an hkpBoxShape, we have two hkEntities, and thus two Collidables, one holding the hkpBvTreeShape pointer (CollidableA) and one holding the hkpBoxShape (CollidableB). When an Agent is created for the first level (a unary agent for the hkpBvTreeShape) it creates a temporary cdbody (CdBodyC) for each child shape that overlaps with CollidableB, and CollidableC is a copy of CollidableA but with the shape pointer changed to be the overlapping child shape. For each CollidableC,CollidableB pair a new Agent is created to deal with them based on the shape types, in this example it will be an agent to deal with triangle to box collisions (convex-convex). If these collide and result in a contact point, the details you get back in a listener are those of CollidableC,CollidableB NOT CollidableA,CollidableB as you might assume.

Example: Havok provides a number of default collector implementations. As these collector cannot store pointers to lowest level hkpCdBody objects, it traverses up the hkpCdBody hierarchy until it gets to the root hkpCdBody, which is a persistent hkpCollidable. Also it stores the 32bit shape key in the hkpCdBody. Say if you have a single landscape, your hit information will include the root collidable (typically an hkpMoppBvTreeShape) and the identifier for the triangle.

3.3.12.2 Asynchronous Queries

Havok allows you to listen for collision events raised withing the Havok step as described above, but also allows asynchronous queries to the collision detection system yourself to find out whether objects are colliding. This is useful, for example, when creating character controllers where you want to move a character around directly while still avoiding interpenetration and without having to place the character in the full physics system.

You can query the hkpWorld with the following 4 queries.

```
// Casts a ray into the world and get a callback for every hit
// Note: it uses the hkpWorld::getCollisionFilter() for filtering
// This function uses a ray-cast function for traversing the broadphase, so for
// long raycasts it is more applicable than hkpAabbPhantom::castRay
void castRay(const hkpWorldRayCastInput& input, hkpRayHitCollector& collector );


// Casts a shape within the worldAsynchronous aa
// The castCollector collects all potential hits.
// Note that the distance stored within the castCollector is a hitFraction (between 0 and 1.0) and
// not a distance.
// This function uses an Aabb-cast function for traversing the broadphase, so for
// long linear casts it this will perform better than hkpShapePhantom::linearCast
// The [optional] startPointCollector returns all the closest points
// at the start point of the linear cast. If you do not want this functionality, pass HK_NULL as the
// "startPointCollector".
void linearCast( const hkpCollidable* collA, const hkpLinearCastInput& input, hkpCdPointCollector&
  castCollector, hkpCdPointCollector* startPoCollector = HK_NULL );

// Get the closest points to an hkpCollidable (= [shape,transform,filterInfo] )
// Note: If you have call this function every step for a given object, use the hkpShapePhantom::
  getClosestPoints()
void getClosestPoints( const hkpCollidable* collA, hkpCollisionInput& input, hkpCdPointCollector&
  collector );

// Get all shapes which are penetrating the collidable
void getPenetrations( const hkpCollidable* collA, hkpCollisionInput& input, hkpCdBodyPairCollector&
  collector );
```

These queries are completely asynchronous, and no caching is done whatsoever. In each case, the broadphase is first queried for all possible overlapping candidates, and the function is performed on these candidates.

It should be noted, however, that if you simply call these methods via the world object the collision input will be that of the world. This has the subtle side-effect that collision details will only be returned if the distance is less than the tolerance. If you wish to perform the query with a tolerance different to that of the world you must explicitly set the tolerance to the required value:

```
hkpCollisionInput input = *m_world->getCollisionInput();
input.setTolerance( 100.f );
```

You may then perform the query directly by asking for the appropriate function from the dispatcher. An example of this is given below for the case of the getClosestPoints(...) function:

```
// Get the shape type of each shape (this is used to figure out the most appropriate
// getClosestPoints(...) method to use)
hkpShapeType a = m_bodies[i]->getCollidable()->getShape()->getType();
hkpShapeType b = m_bodies[j]->getCollidable()->getShape()->getType();

// Ask the collision dispatcher to locate a suitable getClosestPoints(...) method
hkpCollisionDispatcher::GetClosestPointsFunc getClosestPoints = m_world->getCollisionDispatcher()->
  getGetClosestPointsFunc( a, b );
getClosestPoints( *m_bodies[i]->getCollidable(), *m_bodies[j]->getCollidable(), input, collector );
```

Collectors

The result of most user collision queries is returned in form of calling a function of a collector. Examples are

Collector base	Implementations	Meaning
hkpCdPointCollector	hkpAllCdPointCollector, hkpClosestCdPointCollector	used to collect collision detection points for queries returning contact point information
hkpCdBodyPairCollector	hkpAllCdBodyPairCollector, hkpFirstCdBodyPairCollector, hkpFlagCdBodyPairCollector	used to collect information whether two object penetrate or not
hkpRayHitCollector	hkpAllRayHitCollector, hkpClosestRayHitCollector	used to collect ray cast hit information

See the Reference Manual for more details.

3.3.12.3 Asynchronous Queries in multithreaded environments

On multithreaded platforms it is possible to process the collision queries by using parallel worker threads or SPUs (on PLAYSTATION®3). These are the currently supported types of collision queries:

- Get Closest Points query
- Ray Cast query
- Linear Cast query
- Mopp Aabb query

We will take a closer look at the available types in a moment, but first some information on the basic concepts.

Commands, Tasks and Jobs

Usually a command represents one specific query (e.g. one ray to be cast). A job combines one or more commands with some common data, for example the MOPP we want to perform AABB queries against. Most of the available job types can take an arbitrary number of commands. Combining several commands in a single job allows you to spread the job overhead. Internally a job groups its commands into independent tasks, which are then handed off to worker threads or SPUs to be processed in parallel. You are able to control how many commands will be grouped together by setting the m_numCommandsPerTask variable manually depending on your actual query situation. For example if you know you only have 10 commands for a job in total it would make sense to reduce the number of commands per task to 5 or less so that at least two tasks can be processed in parallel.

Batching your queries

To get the most out of multithreaded collision queries you should consider batching all your queries instead of sending them off one by one. Grouping several queries into one job improves the performance overhead incurred by each job. Also try to put all different jobs together onto the queue before starting the multithreaded processing, this way the worker threads or SPUs can better utilized.

Available collision query types

Now let's have a closer look at the actual query types available. These are all defined in physics\hkcollide\multithreaded\hkCollisionJobs.h

- hkpShapeRayCastJob

  This job allows to cast one ray against an arbitrary number of collidables.

- hkpWorldRayCastJob

  This job allows to cast one ray into the broadphase.

- hkpPairLinearCastJob

  This job allows to cast one collidable against another collidable.

- hkpWorldLinearCastJob

  This job allows to cast one collidable into the broadphase.

- hkPairGetClosestPoints

  This job allows to calculate the closest point(s) between two collidables.

- hkWorldGetClosestPoints

  This job allows to calculate the closest point(s) between one collidable and all the surrounding objects that are located within a certain radius.

- hkpMoppAabbJob

  This job allows to perform an AABB query against a MOPP.

As you can see, there are 3 query types that try to emulate the hkpWorld queries mentioned in the previous section. These world queries will query the boradphase collision detection first to find any potientally affected objects. On PLAYSTATION®3 these world jobs will benefit from a significant performance increase as querying the broadphase is very fast on SPUs. Unfortunately there is a memory limitation involved with this. See below for more information on the PLAYSTATION®3 limitations.

Setting it up

You can find pretty simple examples to get you started in the "Demo\Demos\Physics\Api\Collide\AsynchronousQueries" folder hierarchy. These examples all end on "MultithreadingApiDemo". Here's a small summary on what needs to be done to start any of the available multithreaded collision queries:

- First you have to setup the multithreading environment. For this there's a CollisionQueryMultithreadedUtility available under "Demo\Demos\DemoCommon\Utilities\CollisionQuery".
- Now you have to create the queries themselves in the form of commands. Please see below for the alignment restrictions on PLAYSTATION®3.
- Also each job has be be passed a jobHeader.
- Now setup the jobs by passing in the commands and some additional parameters to its constructor. Note that each job needs a dedicated semaphore. This semaphore is released once all the job's commands have finished.
- Then put the jobs onto the job queue. You are able to state onto which queue (PPU or SPU) you want the job to ideally go. After that start the multithreaded processing.
- Each job will signal its 'finished state' by releasing its dedicated semaphore. You can use the time inbetween for any independent computation that might needs to be done.
- Once a job's semaphore has been released you can process the job's results.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.3.13 Contact modifiers

Collision utilities are used to customize collision resolution for chosen bodies or selected pairs of bodies. They can virtually change mass ratios of the colliding bodies, simulate surface velocity or soften collisions.

Collision utilities have the form of entity-collision listeners which override the characteristics of contact constraints created during the collision detection phase. A modfier keeps an extra reference to itself and deletes itself, when the body it's attached to is deleted. Note that only one modifier type can be active for any single pair of bodies.

3.3.13.1 Collision Mass Changer Utility

This utility changes relative masses of interacting bodies. This, for example, allows to some bodies as keyframed against selected other bodies or limit mass ratios in collision response.

To apply this utility use the sample code below.

```
hkpCollisionMassChangerUtil* util = new hkpCollisionMassChangerUtil(body0, body1, body0InvMassMultiplier,
  body1InvMassMultiplier);
    util->removeReference();
```

`bodyXInvMassMultiplier` parameters are the mass scaling parameters. The new mass is equal to the product of the original mass and the inverse of the parameter value.

3.3.13.2 Soft Contact Utility

This utility scales the forces applied during collisions between two specified bodies. This allows an effect of soft and bouncy contacts.

To apply this utility use the sample code below.

```
hkpSoftContactUtil* util = new hkpSoftContactUtil(body0, body1, forceScale);
    util->removeReference();
```

Note that `body1` can be set to `HK_NULL` to enable the utility for all rigid bodies colliding agains `body0`.

Note that this utility only works when the accurate collision response is disabled by setting `hkpWorldCinfo::m_contactRestingVelocity = HK_REAL_MAX`.

Note that this utility should not be used for pairs of bodies which use continuous collision detection, as it may cause significant performance hits. (As the forces during TOI event handling are scaled as well, this may result in a large number of consecutive TOI events being recorded for one pair of bodies.)

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.3.13.3 Surface Velocity Util

This utility simulates movement of the surface of a body. By specifying the velocity of the surface you can simulate moving surfaces like conveyor belts or imitate movement of objects moved by a river's flow.

```
hkVector4 surfaceVelocity(0, 0, 1.0f);
    hkpSurfaceVelocityUtil* util = new hkpSurfaceVelocityUtil(baseRigidBody, surfaceVelocity));
    util->removeReference();
```

Note: this utility has limited application for dynamic or even keyframed bodies, as it only allows you to specify the velocity in the world's space upon the creation of a contact point. Therefore when the body changes the orientation, its surface velocity will remain unchanged in relation to the world's space instead of the body's space.

Note that the surface velocity should be tangent to the contact plane, otherwise you may get unexpected penetrations or bouncyness at the contact.

3.3.13.4 Filtered Surface Velocity Util

This is an expansion of the functionality of the above utility. This utility gives you methods to enable or disable the surface velocity feature between the base body and any other body at any time. Look at the "Physics/Api/Dynamics/Rigid Bodies/Surface Velocity/Filtered" demo to see how the utility is combined with a PhantomShape to achieve a localized 'conveyerbelt' functionality.

Below is a snippet showing how the utility is created and how to enable and disable its functionality for a chosen body. When enabling the utility for a body the `propertyIdForEnabledEntities` property is added to it. Therefore the utility is disabled by default. Note that `propertyIdForEnabledEntities` should be set to a unique id.

```
hkVector4 surfaceVelocity(0, 0, 1.0f);
    int propertyIdForEnabledEntities = 100;
    filteredSurfaceVelocityUtil = new hkpFilteredSurfaceVelocityUtil( baseRigidBody, surfaceVelocity,
      propertyIdForEnabledEntities);

    filteredSurfaceVelocityUtil->enableEntity(someEntity);
    filteredSurfaceVelocityUtil->disableEntity(anotherEntity);

    filteredSurfaceVelocityUtil->removeReference();
```

3.3.13.5 Viscose Surface Util

This utility influences static friction between objects. It always allows for a slight relative movement between bodies in contact. It is useful in resolving situations where an object gets squished in and stuck in a narrow gap. In such situations normally friction might prevent the object from being moved at all. A typical application might be a car stuck between two buildings.

Note that you have to call `hkpViscoseSurfaceUtil::makeSurfaceViscose()` before you add the entity

to the world.

```
hkpViscoseSurfaceUtil::makeSurfaceViscose(hkpRigidBody* entity);
    world->addEntity(entity);
```

3.3.14 Raycasting

There are two ways of raycasting currently in Havok Physics, you can use an hkpAabbPhantom and you can use the method hkpWorld::castRay. In general use an hkpAabbPhantom when you have short rays and hkpWorld::castRay() when you have long rays, and this function calls castRay() on each possibly intersecting hkpShape. Each hkpShape's `castRay()` function finds the first intersection between the shape and a finite length ray defined in the shape's local space. The result (if the ray hits) is stored in an `hkpShapeRayCastOutput` struct. This includes the ray parameterization value - this specifies the relative position of the intersection point along the ray - and the surface normal at the point of intersection. If the ray has hit a subshape in a shape collection, the identifying shape keys of that shape are also provided. Certain speed ups are possible when casting multiple rays from the same point in the world, for more information please see the Optimized World Raycast demo.

3.3.14.1 Notes on using castRay(...)

Calling castRay(...) on a shape will perform a raycast in the local space of the shape and return *true* or *false* to indicate whether or not a hit was registered. As input we pass a both hkpShapeRayCastInput structure and an hkpShapeRayCastOutput structure; these are used to pass and return the details of the raycast. The output from a castRay(...) call for any given shape follows a strict set of rules which determine whether or not a hit is registered. These rules are outlined below:

- A start point of a ray that is exactly coinciding with the surface of an object and that does not intersect the object does not return a hit. There is one exception to this rule, for an hkpTriangleShape a ray that has a start point exactly coinciding with its surface does return a hit. This means that a path constructed as a series of rays will hit each object surface only once.
- An end point of a ray does not return a hit.
- Rays that are on a surface of the object and are parallel to it do not generate hits. The only exception to this is for an hkpCapsuleShape. If a ray is touching the exact geometric surface of an hkCapsule, a call to castRay will return a hit.
- If the start point of the ray is inside the object no hit will be registered.
- A hit will only be returned if the new m_hitFraction member variable is less than the current m_hitFraction (in the hkpShapeRayCastOutput structure). Note: The constructor for the hkpShapeRayCastOutput structure calls it's reset(...) method which sets the value of m_hitFraction to 1.0f.
- If a hit is found the hkpWorldRayCastOutput::hasHit() method will return true or the hkpWorldRayCastOutput::m_hitfraction variable will be less than 1.0f.

In addition to these rules there is one more very important rule:

- The m_radius member variable for hkpConvexShape derived classes is only taken into account for shapes that can do so reasonably. hkSphereShapes, hkCapsuleShapes, hkpBoxShape and hkpCylinderShape take m_radius into account for raycasting. All other shapes do not use this additional element, most notably triangle shapes and thus hkMeshShapes etc.

For both the hkpSphereShape and hkpCapsuleShape the shape itself is defined by the radius and without which the shape would be either a point or line segment. We have chosen not to give these shapes a secondary radius equivalent to the m_radius concept for other shapes. We feel that for these shapes it is more natural to visualise the shape as being completely defined by a single radius for collision detection and raycasting. hkpBoxShape and hkpCylinderShape take m_radius into account on top of their own extents as they can do so without a large performance hit. Radius changes traingles into 3D convex hulls though, so raycasting against them is much more intensive than a single triangle, and as triangles are commonly only used in landscapes where the radius can be zero, it is usally not an issue.

3.3.14.2 Raycasting and collectors

Whether raycasting through the hkpWorld, or the hkpAabbPhantom interfaces, you can select what hkpShapes are considered for raycasting, and what hit points are returned. For every hit a virtual function in the hkpRayHitCollector interface is called. Implementations of the hkpRayHitCollector can store the hits, sort them or process them in place. The hkpRayHitCollector also stores the "early out" distance. The raycast algorithms will use this early out distance to exit early and not return ray hits which are further away. If, in the callback the hkpRayHitCollector sets this distance to 0 (meaning a hit at the very start of the ray was found), no more ray-casts will be performed. So, by implementing a collector, you can get: all hits returned, the closest hit returned, or any hit returned (or any other similar behavior). There are some default collectors (which are open source) in hkcollide/collector/raycollector.

3.3.14.3 Raycast Hit Hierarchy

Raycasts return the full path of hkpShapeKeys from the root shape to the leaf shape, allowing you to identify exactly which leaf shape was hit. For `castRay()`, these are in `hkpShapeRayCastOutput.m_shapeKeys`. For `castRayWithCollector()`, these are in the `hkpCdBody` hierarchy. Note that in the collector interface all the (possibly temporary) shapes exist on the stack. For the non-collector interface they may have to be recreated if the shape instance is needed, not just the shape key.

```
// This function returns the shapekey which was returned by the older non-hierarchical raycast
hkpShapeKey getLastShapeKeyFromHierarchy( hkpShapeKey* keys )
{
  hkpShapeKey key = HK_INVALID_SHAPE_KEY;
  for( int i = 0; i < hkpShapeRayCastOutput::MAX_HIERARCHY_DEPTH && keys[i] != HK_INVALID_SHAPE_KEY; ++i )
  {
    key = keys[i];
  }
  return key;
}
```

```
// This function walks fromt the root down a shape hierarchy.
// It's better to use the collector interface if access to the shapes is needed, since
// they already exist on the stack.
void walkShapeHierarchy( const hkpShape* rootShape, hkpShapeKey* keys )
{
  // we potentially need a buffer for each level of the hierarchy
  hkLocalBuffer<hkpShapeContainer::ShapeBuffer> shapeBuffer( hkpShapeRayCastOutput::MAX_HIERARCHY_DEPTH );
  const hkpShape* shape = rootShape;
  for(int keyIndex = 0; shape != HK_NULL; ++keyIndex )
  {
    // maybe do something with shape

    // we should be at a leaf shape when we encounter HK_INVALID_SHAPE_KEY
    HK_ASSERT(0, (shape->getContainer()==HK_NULL) == (hit.keys[keyIndex]==HK_INVALID_SHAPE_KEY));

    // go to the next level
    shape = shape->getContainer() //
      ? shape->getContainer()->getChildShape(hit.keys[keyIndex], shapeBuffer[keyIndex] )
      : HK_NULL;
  }
}
```

3.3.14.4 Filtering Raycasts

To filter the results from raycasts, when creating your objects you can set their collision filter info using hkWorldObjectCinfo::m_collisionFilterInfo:

Then you can use an hkpCollisionFilter e.g. hkpGroupFilter to separate objects into into colliding and non-colliding groups e.g. see hkpGroupFilterSetup.

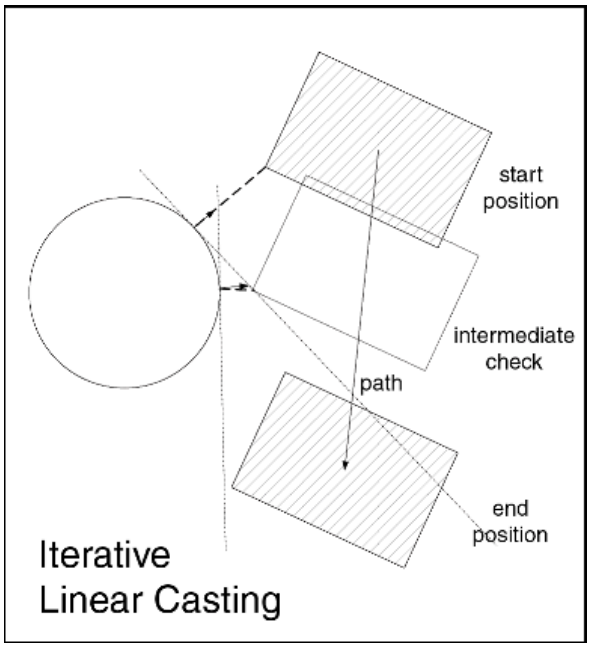
You can alter the filter attributes of hkpWorldRayCastInput or hkpShapeRayCastInput to set filter of the raycast.

See the demos in "Physics / Api / Collide / Ray Casting" for more detail.

3.3.15 Linear Casting

Linear casting is the ability to sweep a convex shape through the world and find the points of collision along the swept path. This can be very useful for accurate proximity checks for characters and movement planning. Note that the shape radius is considered so you may want to set the radius of the cast shape to zero. Linear casting is an iterative algorithm. We describe the algorithm below to give you an intuitive understanding of the associated parameters.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



The image above shows a box swept along a path. The path forces it to come in close proximity with a sphere. First the closest distance between the objects is calculated. This normal or separating plane between the objects is also computed (by getClosestPoints()). We project both the initial distance and path onto the normal and compute the fraction which will bring the closest feature on the box up to the plane defined by normal and the closest point on the sphere. We then recheck the separating plane and continue. In this example the box can complete the path before a third check is required.

The linear caster allows you to control

- hkpCollisionAgentConfig::m_iterativeLinearCastMaxIterations: The maximum number of iterations per cast.
- hkpCollisionAgentConfig::m_iterativeLinearCastEarlyOutDistance: The minimum distance the object must travel before exiting the algorithm.
- hkpLinearCastCollisionInput::m_maxExtraPenetration: The maximum distance the object is allowed to increase its current penetration before a hit is reported.

The last parameter in particular is important for handling worst case behaviour. This happens when an object is cast parallel to a surface it is resting on. This extra penetration prevents numerical instability in the algorithm and allows it to complete we fewer iterations. This use case happens frequently when you use the character proxy so it is important that this parameter is non zero.

3.3.16 Introduction to Customization

The Havok Collision Detector has an open and extensible architecture. This part of the Collision Detection chapter describes how you integrate your own collision detection routines directly. We also describe each of the components of the detector so you can reuse these to leverage the most value from your integration. Before you read this chapter, make sure you have read the sections of this chapter up to this point fully, In particular, you'll need to understand the following terms:

- Axis Aligned Bounding Box (Aabb)
- Broadphase
- Narrowphase
- Shapes
- Collision Agents

Our collision detection system is both open and extensible. It allows you to hook in at any phase of the collision detection pipeline. Deciding where to hook into this pipeline is the most important integration decision, as it will affect the overall performance of the system. It will also determine the level of effort (coding) required to integrate. For simplicity we'll consider the system to be broken into three phases, broad-, mid- and narrowphase. The broadphase identifies potentially overlapping pairs of objects. The midphase helps to decompose a pair of overlapping objects into pairs of overlapping basic shapes. Finally the narrowphase generates collision information for a given pair of basic shapes using a collision agent.

We do not recommend that you try to implement your own narrowphase collision agent, although it is possible to do so. If you wish to do this please contact Havok for further assistance.

As a recap, Havok has implementations for the following phases and basic shapes.

Broadphase

- 3 Axis Sweep and Prune

Midphase

- Bounding Volume Shapes
- Bounding Volume Trees
- List Shapes

Narrowphase

- ConvexConvex
- BoxBox
- SphereSphere
- SphereTriangle

Havok Basic Shapes

- Triangles
- Convex Polyhedra
- Boxes
- Spheres

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.3.17 Summary

You have seen how Havok Collision Detection system works and how to integrate your own algorithms into the collision detection pipeline. With this information you should be able to leverage the most from both Havok's collision detection and pipeline and your own internal data structures. By overriding and extending parts of the collision detector, you are embedding custom code directly into some of the inner loops in the collision detection pipeline. If you are considering integrating your own solution the best advice we can give is 'profile as you go'. Continually use the Havok timers to ensure your implementation is at least as fast as the defaults provided. Also when debugging creating a custom viewer for the visual debugger is guaranteed to save you time, especially if you are developing on a console.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.4 Continuous Physics

3.4.1 Introduction

Continuous physics is Havok's concept of high quality rigid body simulation. Continuous physics means that we do not ignore what is happening between two physical time steps, but handle all collisions at the exact time of their impact if necessary.

Most existing physics engines perform computations based on discrete states. That type of simulation will be referred to hereafter as discrete physics.

Discrete simulation breaks physical motion into discrete physical time steps. It performs collision detection at those discrete times and employs simple integration methods to advance the state of bodies in time. So all physical operations are handled at one physical synchronous instance in time (later referred to as PSI). Collision detection is only based on the relative position of bodies at each step in isolation. For small or fast objects it is hard to determine coherency between frames, therefore the simulation can easily break in these cases.

The continuous physics core is based on motion state representation, which contains information from two consecutive frames. With that information we can interpolate the state of simulated bodies in-between frames and perform continuous collision detection.

With such precise collision information we employ our solver to chronologically handle each occurring collision at its exact time of impact (TOI). This ultimately brings our simulation to a physically correct state in the following frame.

Using continuous physics is very simple. Even if you are new to Havok Physics you need only read the next section on basic API parameters to get started. Then you should be ready to start experimenting with the simple demos. Subsequent sections explain in more detail the concepts behind continuous physics, and are meant for more advanced users of the engine.

3.4.2 Fundamental Concepts

The following sections present an in-depth description of the concepts behind continuous physics. It is not necessary to understand this part in order to use the engine at a basic level.

Havok Physics has implemented new simulation loops and collision detection algorithms. This section progresses from the basics of motion data representation to the new functionality of the collision detector, and explains our implementation of the continuous physics simulation loop.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.4.2.1 Transform and SweptTransform

hkTransform is a representation of a body's state i.e. its position and orientation at a given moment in time. In discrete simlulation that position and orientation are constant for the duration of a frame and are updated at it's end.

In continuous physics we need to know the body's precise position and orientation at any moment in-between discrete 'frame ticks'. So we represent the rigid body's state by an `hkSweptTransform` structure which basically stores the position and orientation at two chosen moments in time.

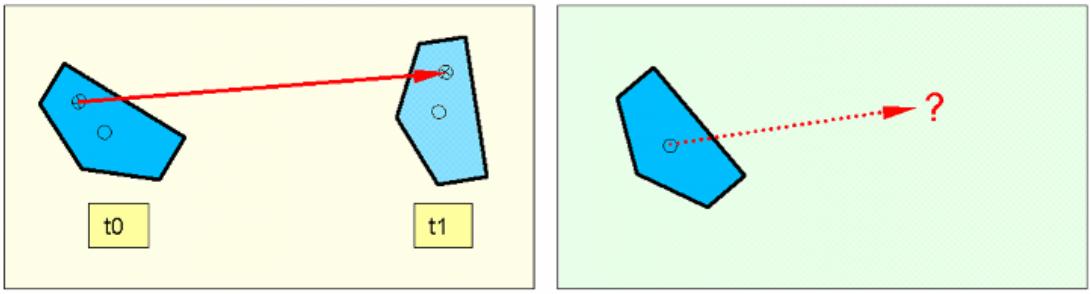


Figure 3.45: Illustration of a hkSweptTransform, which stores position and orientation information at two moments in time. Information about mass centre allows for proper interpolation of position in time. In contrast, discrete physics' representation uses position and orientation information from one time moment. Motion State values from consecutive frames are therefore not explicitly linked in any way.

`hkSweptTransform` also stores location of the body's mass center allowing for proper interpolation of position in time. In contrast, discrete physics' representation uses position and orientation information from one time moment. Body's state information from consecutive frames is therefore not explicitly linked in any way.

A `hkSweptTransform` stores:

- Time `t0` and `inverseDeltaTime`, such that `t1 == t0 + 1/inverseDeltaTime`.
- The body's mass centre position and the body's orientation at `t0` and `t1`.
- The center of mass in the body's local reference frame.

The position of the mass center is required to properly interpolate the body's transform between the two time endpoints.

To get body's transform at arbitrary time `tx` (where `t0 <= tx <= t1`) we interpolate the position of body's mass center and interpolate its orientation quaternions. Here's a simplified code snippet:

```
// Performs linear interpolation of positions
post = pos0 * (1 - t) + pos1 * t
    // Performs linear interpolation orientations
qt = lerp(q0, q1, t)
    // Calculates position of the body's reference point
    // using the mass center stored in local space mcPos
post = post - qt * mcPos
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Note:
One may ask why we need the center of the mass information and why we can't just associate the shape's origin with the mass center of a rigid body. The necessity for that information becomes clear when we consider `hkpShapeCollections`, where a number of shapes is associated with one rigid body. Collision detection is performed for each of those shapes in isolation from others and each of those shapes may be positioned far from the body's mass center. In such case the trajectory travelled by the geometrical center of the shape will not be linear and we need explicit information about the mass center to interpolate the shapes' position properly.

3.4.2.2 State Integration

To integrate a body's state we use simple Euler integration. At each 'frame tick' (also referred to as a PSI) assume that the body has reached its final position at `t1`. This becomes the initial position of the updated `hkSweptTransform`. After applying all gravity, constraint forces, reaction forces and impulses we get the body's final velocity that we can use to calculate its end position at `t1`. This new end position is perceived only as predicted or potential because it may be modified later when in-between-frames collisions are processed (we refer to such collisions as TOI or time-of-impact collisions).

3.4.2.3 Collision Detector

The collision detector performs two tasks – it verifies the motion of all simulated bodies throughout the frame step and it generates contact points between those bodies for the specified end time of the frame step.

Verifying Motion and Generating Time-of-Impact Events.

The collision detector is given two `hkSweptTransforms`, which overlap in time. It interpolates the transform of each of the associated bodies along time and checks for collisions.

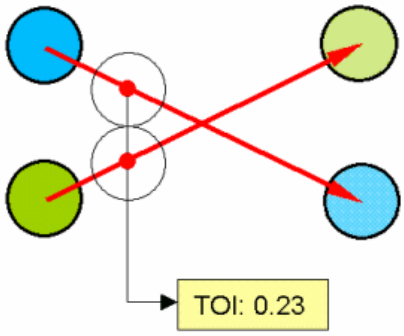


Figure 3.46: As the positions of the two bodies are interpolated they start moving towards each other. The moment when they touch each other is the time-of-impact and it is reported in a TOI event. If we interpolated their positions further the objects would interpenetrate significantly and eventually pass through completely.

When interpenetration of bodies is detected the collision detector generates a collision event. Such events are called time-of-impact (TOI) events. They contain the time of reported collision/impact and references to the two colliding bodies. Collision detection is performed for all pairs of object and ultimately yields a list of collision events.

Calculating Contact Points

Contact points generated by the collision detector are always calculated for one given moment in time. This operation is done independently of TOI events. By default, the collision detector calculates contact characteristics at the end of the current frame.

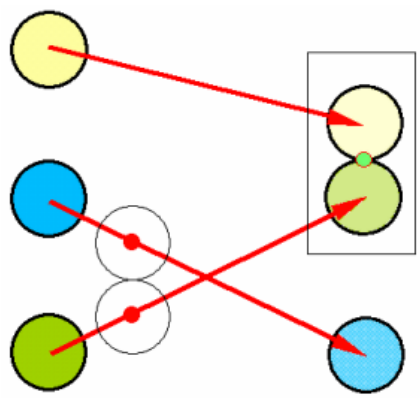


Figure 3.47: Pairs of objects found in proximity or contact at the end of the time step (objects selected within the box) generate contact points (small dot).

At that time the collision detector does not know whether or how any occurring TOI events are going to be handled. It therefore generates contact information, which may be used by constraint solver in the next integration step (or may be overridden with newer information).

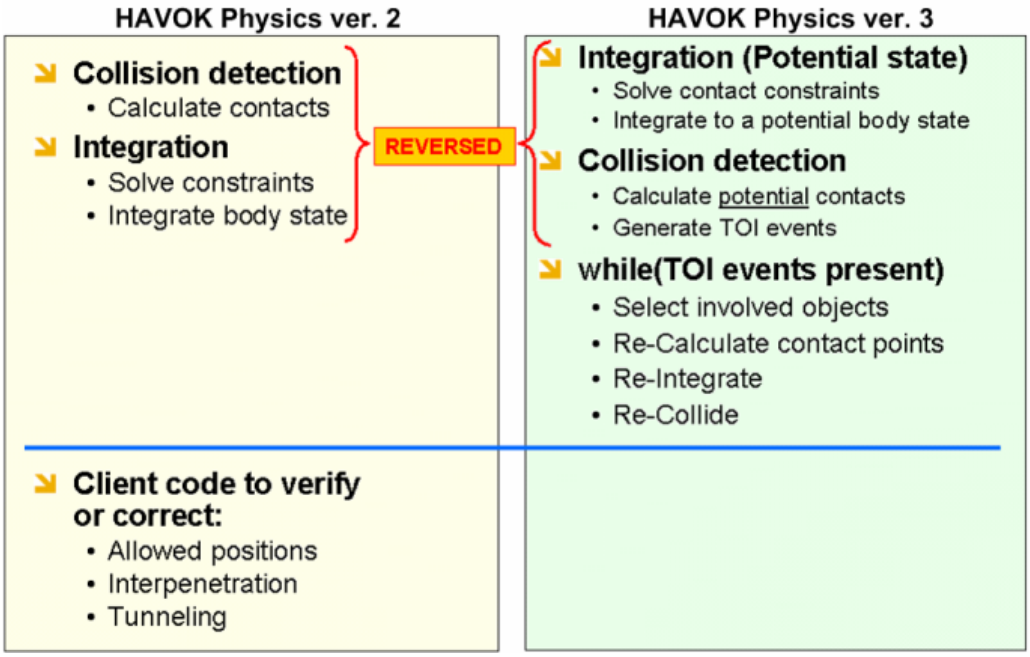
Warning:

When Havok generates and destroys such potential contact points it executes `hkpCollisionListener::contactPointAddedCallback` and `hkpCollisionListener::contactPointRemovedCallback`. Since the contacts are potential, you must be careful when using these callbacks as triggers, which drive game-play logic. The same limitation existed in Havok 2.

This part of collision detector functionality corresponds to the whole of collision detection in Havok 2. Note that thanks to TOI events there is no longer a need to have a large additional collision tolerance in hkpWorld. In Havok 2 the tolerance layer was used to generate contact points early to avoid object penetration or tunneling in the following frame. However it is still a good idea to have a big (∼10cm) collision layer as it can avoid CPU intensive TOI events. However a large collision tolerance can have the side-effect that objects bounce off each other before actually hitting each other. To avoid this side-effect, the collision tolerance can be reduced to a small value (around 1 cm), however more CPU will be used to handle the additional TOIs.

3.4.2.4 Continuous Simulation

The picture below highlights differences between Havok 2 and Havok 3. Collision detection and Integration are common to both engines. They are referred to as the Physical Synchronous Instance Step. Havok3's final while loop is referred to as the TOI-handling loop. It handles occurring time-of-impact events chronologically, and is intended to replace some of the user's motion verification code usually created outside of the engine.



Physical Synchronous Instance Step

Physical Synchronous Instance (PSI) Step corresponds to the Havok 2 simulation pass it is responsible for performing a synchronized collision detection and integration for all bodies.

Havok 2's approach was based on processing information from one physics frame only. Given the positions of bodies it performed collision detection and found any colliding or overlapping object pairs that needed to be corrected. The constraints solver used that information to apply proper forces to the bodies. Finally the correct new state was integrated into the system.

In contrast, Havok 3 assumes that it already has proper contact information for the current state at the beginning of each pass of the simulation loop. (Even if it doesn't, any unacceptable penetrations will be reported via TOI-events before the next frame is reached.) It runs the constraint solver and integrates the bodies' states first. This yields their predicted or potential final states.

Collision detection is responsible for verifying the bodies' movement from their initial to their final state. If there are no interrupting TOI events generated by the collision detector then the predicted state is validated and remains unchanged. Otherwise it may change as TOI events are handled (as is described in the following paragraphs).

Time-of-Impact Event Handling Loop

All TOI events for the entire world are kept on one list. The TOI-handling loop processes the events chronologically according to their time-of-impact.

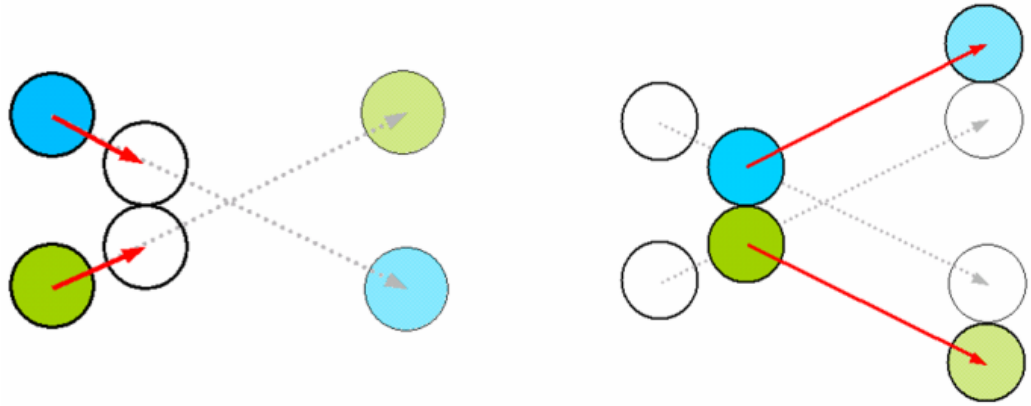
In each TOI-event the motion of some bodies is altered. For those bodies, collision detection is performed again which may result in removal, modification, or addition of some TOI-events.

Solving Collision Impact Response

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

For each event the two colliding bodies are identified and a simple collision response algorithm is used to process the impact in isolation from the rest of the world. (It will be shown in the Optimizations section below that we may also process a greater number of bodies in one TOI event.)

Once proper reaction is calculated, the motion states of the bodies must be recalculated. We set time `t0` to the moment of impact and use the corresponding interpolated transform as the new initial transform at `t0`. Then we integrate the state to `t1`. Note that this time we integrate only by a fraction of the original PSI step.



Note:

As the result of TOI-response, `hkSweptTransform`s of all bodies end at the very same moment in time the time of the next PSI. However, their `t0` values may differ.
Since each processed TOI event has a later or equal time-of-impact, each processed event always happens in between the beginning and the end of the `hkSweptTransform` time of every simulated body.

Partial Collision Detection

After altering the motion of bodies according to the calculated collision response we run collision detection again to verify the newly calculated trajectories of the bodies. We must therefore collide each of the colliding bodies with each other and with the rest of the world's objects.

This generates a list of new TOI events, which are inserted to the current world's list of TOI events to be handled.

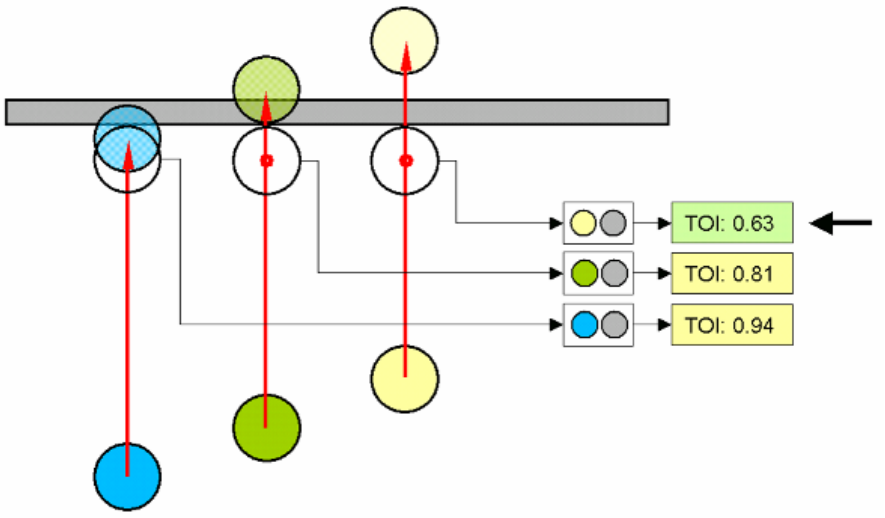
Also notice that previously calculated events, which referred to the reintegrated bodies, are now invalid. Such events are either removed or removed-and-replaced by the newly calculated events with new time-of-impact values.

3.4.2.5 Graphical Examples

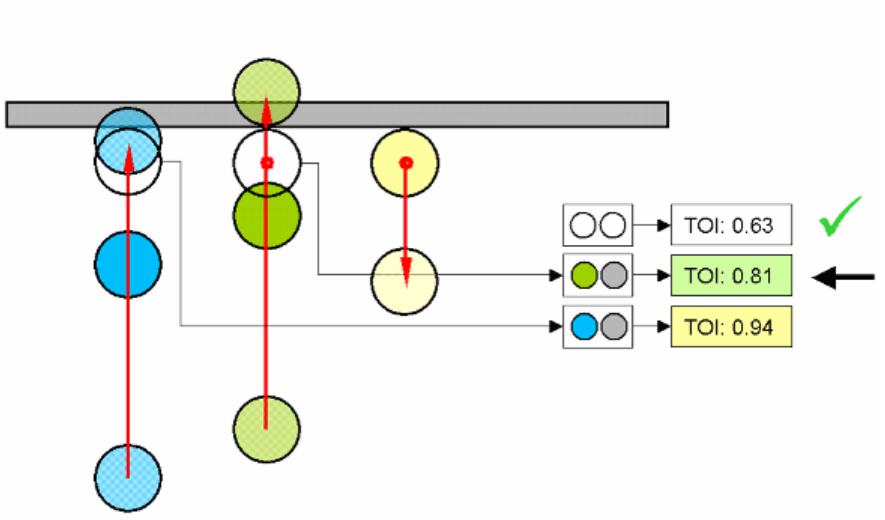
The following examples illustrate the operation of the collision detector in various situations.

Independent Collisions

Three spheres are moving in parallel, and are stationary relative to one another. In the picture below , the spheres at the bottom represent position at `t0`, and the uppermost spheres represent position at `t1`. They are about to hit a fixed wall (gray block).



The collision detector determines potential interpenetration and generates a TOI event for each of the spheres (represented by a black ring on the movement path of each of the bodies). Times for each of the TOI events are displayed in the table on the right. The system handles the earliest TOI event (for the rightmost sphere) first, marked by an arrow on the right side.



Once the first TOI collision is handled, the rightmost sphere's motion is reintegrated and the collision detection is re-run. It does not return any new TOI events in this example. The processed TOI event is taken off the TOI event list. The next earliest TOI is processed next.

Multi-body Collisions

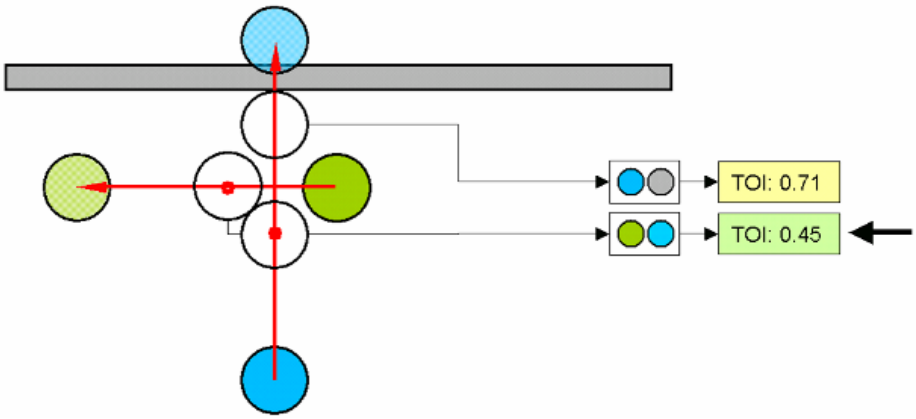
The symbols used here are analogous to those in the example above.

This example presents two spheres about to collide. Moreover, if that collision weren't handled, the blue sphere (the one with vertical movement) would hit the fixed wall.

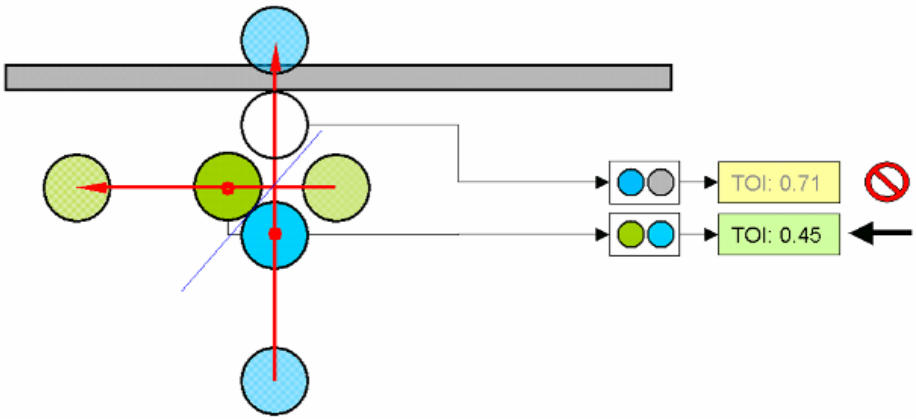
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Note:

Every body may have a number of `hkpCollisionAgents` with other bodies. Each of them may generate a TOI event. Therefore one body (like the blue sphere in the example) may end up with a list of a few TOI's.

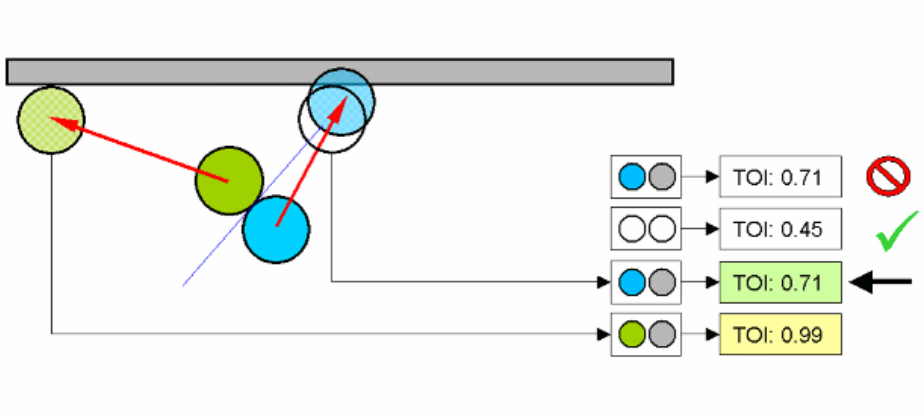


The sphere-sphere collision is handled first. Collision characteristics are calculated, and both bodies change in motion. Therefore the TOI-event between the blue sphere and the fixed wall will be removed from TOI list.



Collision detection is run with the newly calculated `hkSweptTransforms`. Each sphere generates a new TOI collision with the fixed wall. One event replaces the previously removed one; the other event emerges as a result of the handled collision.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



After both TOI events are handled, we receive the correct world state for the next PSI.

3.4.3 Priorities and Quality Types

Constraint priorities and the quality of objects control the effort Havok Physics puts into proper simulation of systems of bodies.

When simulating a massively populated world, we can often distinguish between a few critically important objects and numerous other objects that simply add complexity to the world to make it more realistic.

3.4.3.1 Constraint Priorities

Constraint priorities decide whether and how constraints are handled. There are three priority settings (listed in order of lowest to highest):

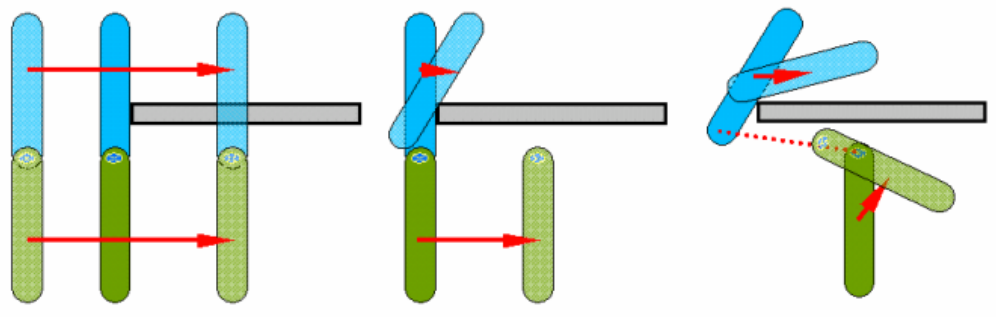
- `PSI` - Handled during PSI steps only.
- `TOI` - Handled both in PSI steps and TOI events.
- `TOI_HIGHER` - Handled both in PSI steps and TOI events. This setting overrides any normal constraints that it conflicts with. An example would be a very heavy car pushing a hand of a ragdoll into a wall. In this case we want the hand to penetrate the car but not the wall. Therefore wall-to-hand priorities should be set to `TOI_HIGHER` and car-to-hand priorities should be set to `TOI`. See the Collision Quality Level dispatch table in section Object Quality Types for details.

PSI and TOI Priorities

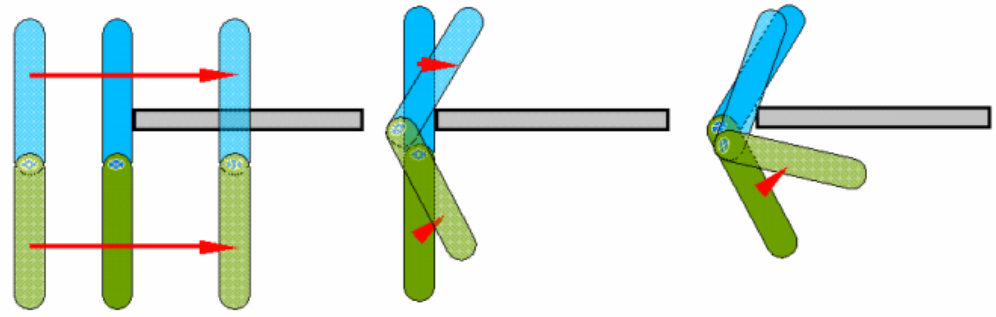
TOI constraints add extra quality to continuous simulation. When a body collides in a TOI event, its attached TOI-priority constraints are processed and may affect movement of other linked objects. If the lower quality PSI constraints are used, parts of constrained systems, e.g. ragdolls, may drift apart during TOI events.

The schematic below presents a hinge hitting a fixed obstacle with one of its attached bodies. On the left side we see the hinge at physical frame 0, and on the right side we see the hinge at the physical

frame 1. The hinge hits the fixed obstacles in between these two physical frames. If the hinge uses a PSI constraint, the hinge is not touched in the TOI event and as a result the two bodies drift apart.



In case of a TOI constraint, the hinge is processed in the TOI event and as a result the error is much less noticeable.



The following screenshots present PSI- and TOI-constraint-based ragdolls hitting a fixed obstacle.

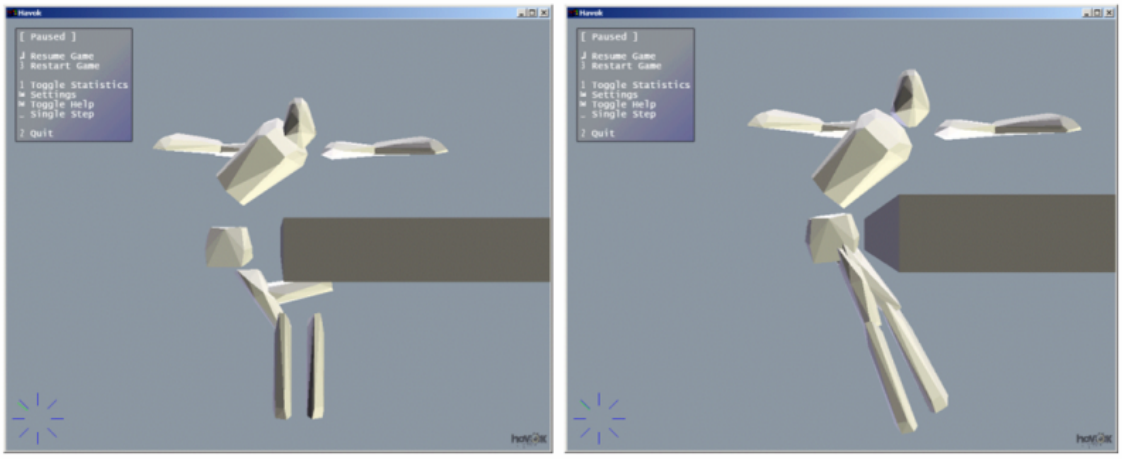


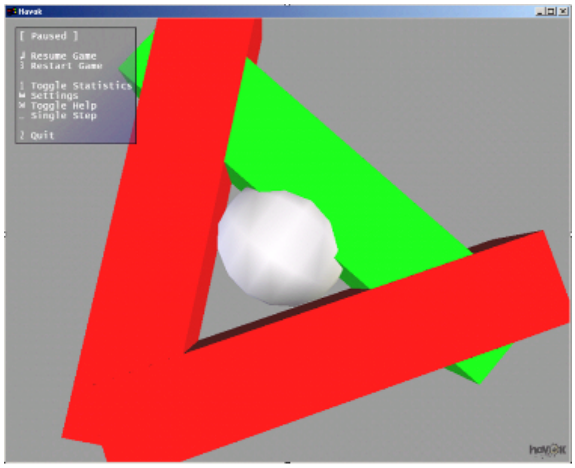
Figure 3.48: Left: PSI-ragdoll; Right: TOI-ragdoll. Note the clearly visible drift of the ragdoll's lower legs on the left picture.

The TOI_HIGHER Priority

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Constraints are assigned either a normal (PSI / TOI) or a higher priority level. In unsolvable or conflicting cases normal-priority constraints are violated before the high-priority ones. If two constraints are assigned the same priority it is ambiguous which will yield first.

The example below shows an 'impossible' scenario, where three boxes are being keyframed into a dynamically controlled sphere. Here, the sphere has higher-priority contact constraints with the red boxes and a normal-priority contact constraint with the green box. Note that only the normal-priority constraint is violated as the green box slides toward the ball and into a conflicting position.



3.4.3.2 Object Quality Types

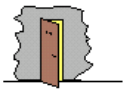
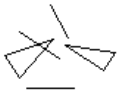
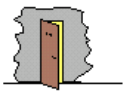
In a game you may want to use different quality settings for different kinds of interaction.

As it might be inconvenient to set a quality level for every occurring interaction in callbacks, Havok categorizes objects into six groups and uses a table in `hkpCollisionDispatcher` to determine the `CollisionQualityLevel` of each interaction.

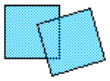
For example, you may expect critical game play objects (e.g. a key), to have solid high-priority collisions with the world so that they never accidentally sink into ground where they cannot be found. You may want to allow penetrations between the world and 'debris' objects, which only serve the purpose of enriching the visual side of the game, e.g. small car parts flying off from a car crash. Moreover, you might still expect those debris objects to interact with bullets you shoot.

Objects are categorized as follows:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

FIXED		For static world elements. Especially objects represented by a triangle mesh.
KEYFRAMED		Animated object with infinite mass.
DEBRIS		Low importance objects adding visual quality.
MOVING		Regular objects populating the world. Furniture, etc.
CRITICAL		Gameplay-critical objects that are never allowed to penetrate or leave the game world.
BULLET		Fast-moving projectiles.
KEYFRAME_REPORTING		Use this for moving objects with infinite mass which should report contact points and Toi-collisions against all other bodies.

The following types of collisions/interactions are available:

`PSI`		Collisions are handled only in PSI steps. Don't generate TOI events. Allow interpenetration of objects.
`TOI`		Collisions generate TOI events.
`TOI_HIGHER`		Collisions generate TOI events. Contact constraints have higher priority in the constraint solver. This prevents interpenetration even when other objects 'push' bodies into each other.
`TOI_FORCED`		Same as `TOI_HIGHER`. Additionally extra checks are performed when this collision is touched in a TOI event. If the constraint is violated then `hkpToiResourceMgr::cannotSolve()` callback is fired.
`---`		Ignored

Note:

`TOI_HIGHER` and `TOI_FORCED` types should be used only between a moving and a fixed object never between two moving objects. Both collision levels generate higher-priority contact constraints. Such constraints are used to prevent game play-critical objects from leaving the game world. For this reason it is wise to guarantee no conflicts between higher-priority constraints, and the easiest way to do it is to allow them only between pairs of fixed and non-fixed objects.

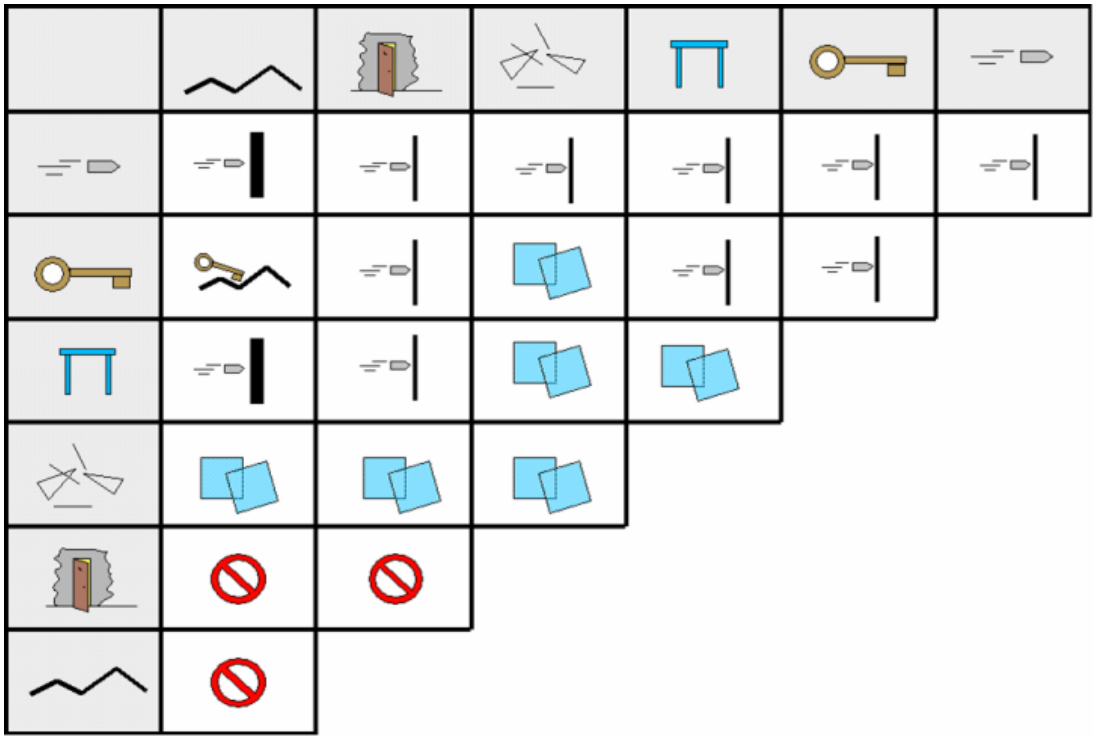


Figure 3.49: Collision Quality Level dispatch table.

Notes:

- No fixed and keyframed objects interact with each other.
- Debris can interpenetrate with any object but responds to bullet hits properly.
- Critical objects are forced not to penetrate with fixed objects, and do not have higher-priority contact constraints with any other objects.
- Moving objects generate TOI events with fixed, key framed, critical and bullets but are allowed to penetrate slightly with other moving objects or debris.

3.4.4 Performance Optimizations

The solution provided solves most problems in a finite number of TOI events. However this number might be large enough to stall the simulation for too long a time. To allow real-time operation of the engine in complex cases, we have implemented a series of additional routines, which trading quality for CPU time.

3.4.4.1 Problem: Too many TOI Events

In certain situations interacting objects generate too many TOI events for the simulation to handle, as the following examples illustrate.

Too many TOIs generated iteratively

A bouncy object is squeezed in between two heavy blocks and bounces in sequence from the top to the bottom block. As the blocks move closer to each other, the number of bounces (or TOI events to handle) in a given time period increases. This stalls the simulation if the restitution of collision between the bodies is high enough.

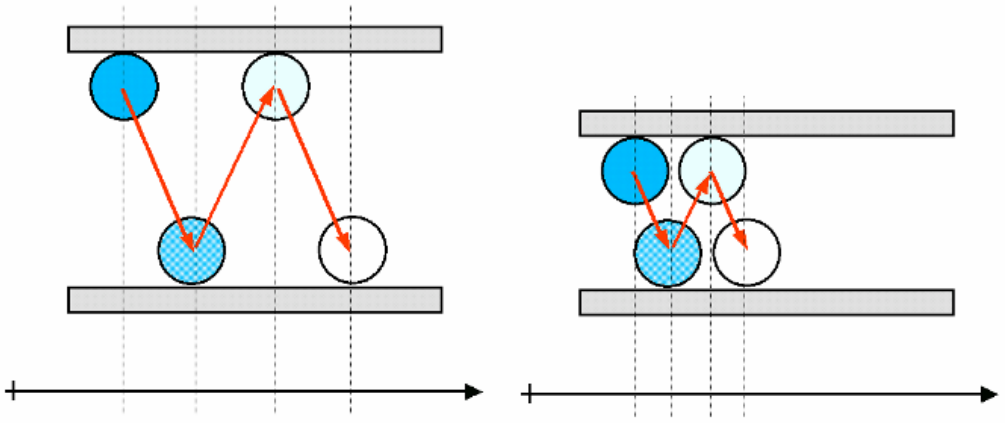


Figure 3.50: TOI events are marked on a time line. Both examples are for a sphere of the same initial velocity. No friction present; unit restitution.

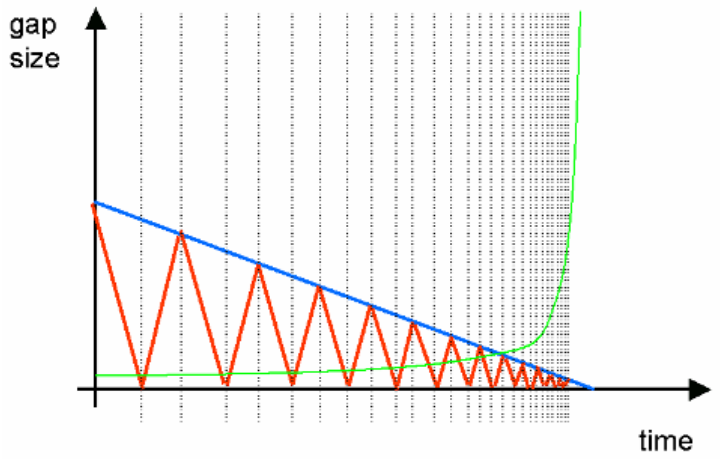
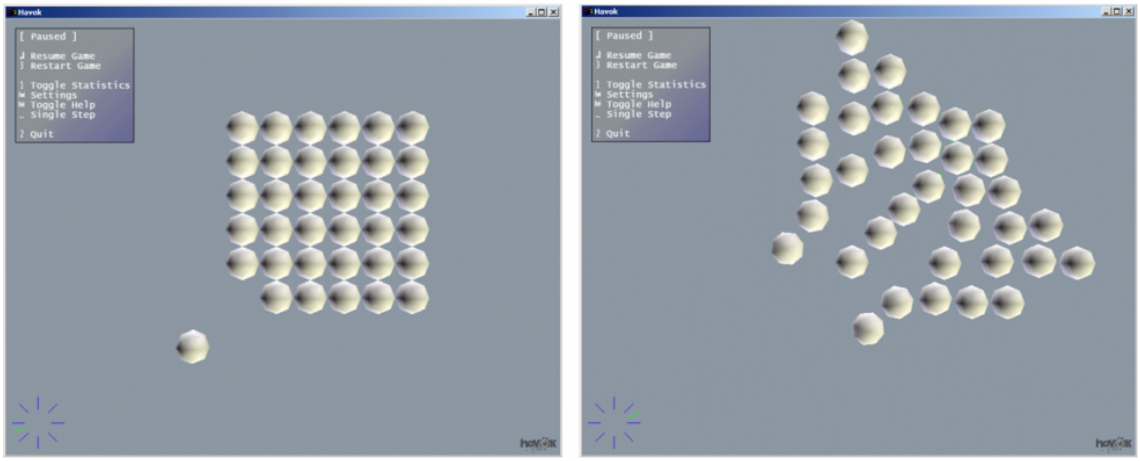


Figure 3.51: This graph shows how TOI events are generated more densely (vertical dotted lines) as the distance between two blocks decreases down to the point when the ball cannot move vertically anymore. The zigzag line shows the trajectory of a reference point on the ball. The green line shows the frequency of generated TOI-events.

Too many Bodies and/or Contact Points

A tightly packed array of objects is hit by a high-velocity particle. The fast particle transfers its momentum to a few objects it collides with. Those objects then instantly collide with further objects, which block their movement. As the result of those collisions the bounced-back objects collide again with the initially fast-moving particle. And those collisions trigger recursively resulting in possibly hundreds or thousands of TOI events.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



3.4.4.2 Solution: Allow Penetration Depth

To avoid TOI events accumulating into high numbers at one moment in simulation time we have introduced the `hkpCollidable::m_allowedPenetrationDepth` parameter, which is assigned to every rigid body. This value is used to allow some penetration for one object. If we increase this parameter we allow bodies to penetrate more and gradually trade the solid feeling of the system for CPU time.

As a result of allowing penetration depth a squeezed bouncing object (see problem 'one') will gradually penetrate the solid blocks more and more.

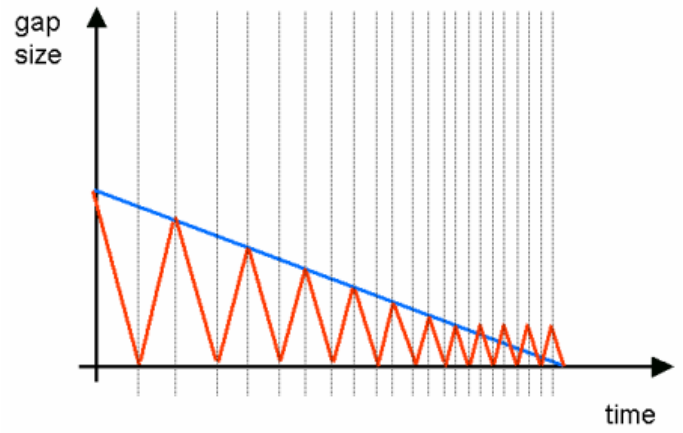
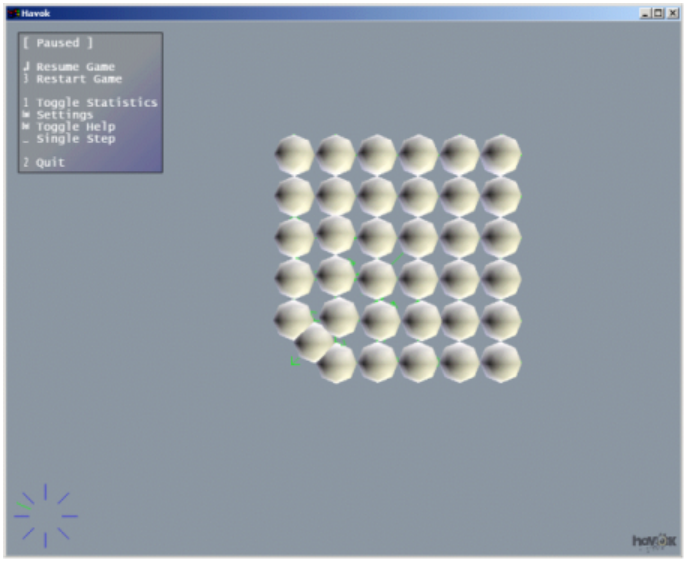


Figure 3.52: Note that the object (the red zig-zag path) penetrates the closing block (the straight blue line). As the block gets closer the system ensures minimum time distance between TOIs causing penetration.

Similarly the fast particle (see problem 'two') will initially penetrate the obstacles by a greater distance.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



3.4.4.3 Localized Solving and the hkpToiResourceMgr

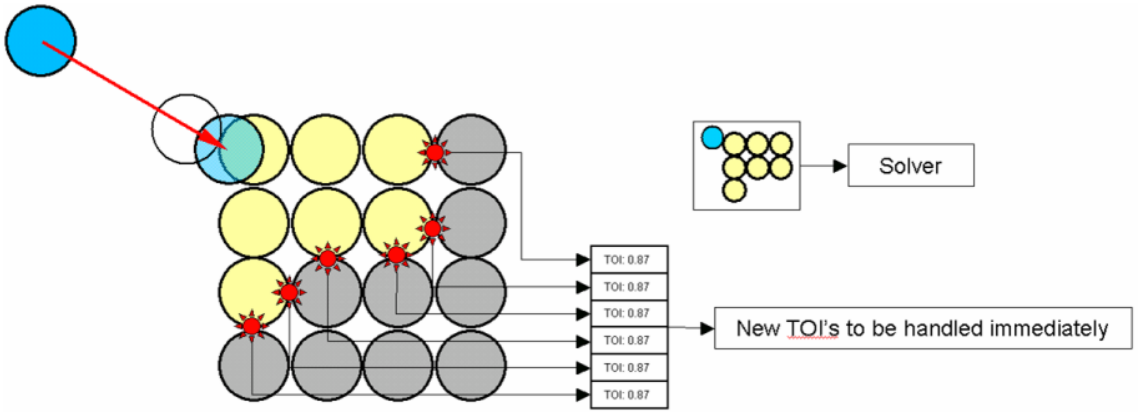
The `hkpToiResourceMgr` class is an interface to that grants fine grained control of the `hkpContinuousSimulation`. It allows the user to

- filter TOI events.
- set parameters and assign memory resources for each TOI event being handled.

Before handling a TOI event, the engine calls `hkpToiResourceMgr::beginToiAndSetupResources(..)`. This callback is responsible for allocating memory to be used by the solver and for setting hard limits on the maximum number of bodies and constraints to be passed into the solver. The callback may also return `HK_FAILURE` to command the simulation to skip this event. If the event is not skipped, the allocated memory should be freed in the `hkpToiResourceMgr::endToiAndFreeResources(..)` callback fired at the end of TOI-handling function.

When the group of activated bodies exceeds the limits set or when the system runs out of memory for the solver, a `hkpToiResourceMgr::resourcesDepleted(..)` callback is called to let you decide how to handle this case.

The TOI handling routine also assures that `TOI_HIGHER`-priority constraints are not violated. If it fails the user receives a `hkpToiResourceMgr::cannotSolve(..)` callback.



The example above illustrates a fast particle hitting an array of bodies. In order to calculate a perfect non-penetrating solution all of the bodies have to be passed into the solver. Assuming we have limited memory only a part of the array is activated, while the gray objects will be ignored. After handling this TOI event and performing collision detection new TOI events may be created (marked on the picture with the red spots). Possibly though the particle's momentum is redistributed well enough and the interpenetrations (at the red spots) are small. In that case presetting `hkpRigidBodyCinfo::m_allowedPenetrationDepth` to a large-enough value would suppress new TOI events being generated.

3.4.4.4 TOI Solving Causes Extra Performance Hit for Discrete Collisions

When two bodies of quality type MOVING interact, one might assume that their collision detection is only run once per frame. This is not the case if one of the bodies is involved in a TOI collision.

All bodies that take part in a TOI collision may have their motion affected and their collision detection is performed again.

The said two MOVING bodies may re-collide if one of them is involved e.g. in a TOI collison with fixed landscape.

Note that when there are many MOVING-quality objects in proximity, then a single TOI may cause a lot of collision detection work. Additionally, since TOI's are handled in a single thread, this collision detection work will not be spread across several CPU's causing even a greater performance hit.

3.4.5 Asynchronous Simulation

A discussion of stepping the world synchronously and asychronously is presented in the dynamics chapter in the section Stepping the simulation forward. In continuous simulation the advanceTime() call (referenced in that section) will advance the time of the simulation forward and solve any TOIs that arise between the current time (from the last time advanceTime was called) and the time it tries to advance to. This is visualized below for a ball travelling from one PSI to the next, with a frame time marker set between the frames. In discrete simulation advance time simply advances the world current time to "Frame 2" below:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

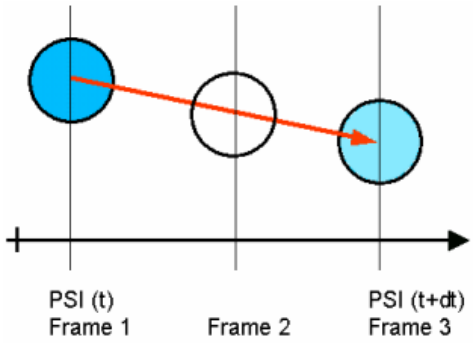


Figure 3.53: To display an object at an asynchronous point between two psis, you can set the frame marker, then call approxCurrentTransform to interpolate between the 2 psis.

In continuous simulation, the advanceTime() function will additionally process all TOIs up to the time it is advancing to. This is shown in the diagram below. The first time advanceTime() is called it will process TOI1 to advance to the "Frame 2" marker.

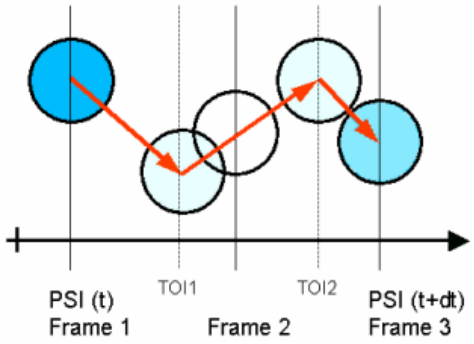


Figure 3.54: In continuous simulation, TOI events are handled when advanceTime() is called. When Frame 2 is rendered only TOI1 is handled, and TOI2 is simply kept on the TOI list until the next execution of advanceTime().

3.4.6 Advantages of Continuous Simluation

3.4.6.1 Small Objects

With continuous simulation it is safe to use small objects (<10cm in size). That's because, a TOI event is generated whenever the objects attempts to penetrate scene borders. With discrete simulation a small object could easily penetrate a triangle-mesh objects even when moving with small velocities. Additionally, varied constraint priorities can be used to keep critical objects from being pushed through floors or walls.

You may attempt to stack small objects. However accuracy decreases with object size giving poor results for objects below 10cm in size. To improve the quality of solver results for smaller bodies you may increase their rotational inertia. This causes noticeably unrealistic movement in larger objects, but helps to minimize jitter in smaller ones. The issue of small objects is well illustrated in the Pyramid demo.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.4.6.2 Low Frequency Simulation of Ragdolls

Ragdoll simulation frequency can now be set at low frequencies. Simulation at 15 Hz gives a fairly good visual quality, although limb separation is slightly noticeable when the rag doll collides with environment. The trick of increasing rotational inertia of body parts is also applicable to ragdolls and gives significant improvement in quality for high-velocity ragdoll impacts.

At even lower frequencies (e.g. <10 Hz) it is already noticeable that gravity only is applied to rag doll body at discrete moments resulting in instantaneous velocity change. This feature is well illustrated by the Ragdoll Vs Mopp example.

3.4.7 Limitations of Continuous Simulation

3.4.7.1 Angular Errors

When processing contact constraints, Havok Physics does not check angular movement as such but only looks at the resulting relative linear velocity of contact points on colliding bodies. Contact constraints prevent each of the bodies from penetrating each other along one direction, or through one contact plane.

This solution works efficiently with objects flying with high angular velocities as long as the solution uses a limited number of TOIs. That's the case for all convex bodies.

Consider however a hollow object which acts as a centrifuge. The example contains a sphere enclosed in a rotating hollow box. As the box's angular velocity rises the sphere is pressed against its walls.

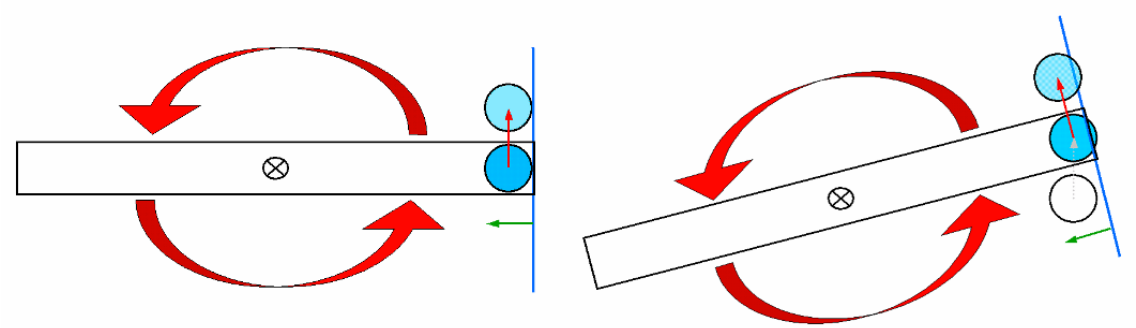


Figure 3.55: Note the penetration between the sphere and the centrifuge after the first step (right picture). This and consecutive incremental error will accumulate causing the sphere to tunnel through the centrifuge's wall.

In the first frame (left picture) the contact constraint only prevents the sphere from going further to the right through the blue contact plane. However after the next integration step the wall of the centrifuge moves to the left due to the centrifuge's angular motion. This error accumulates over consecutive simulation frames. It is corrected partially by penalty methods, but when the centrifuge's angular velocity reaches a certain value, the sphere will eventually tunnel through.

Consider continuous simulation now. As the hollow box accelerates the contact constraints keeping the sphere inside must be updated more and more often, resulting in more and more TOI events to handle. However since we have specified a minimum time distance between two consecutive TOI events handled, the sphere will start to penetrate the box's walls at a certain angular velocity. Eventually the result is

the same as in discrete simulation.

The effect of angular errors is well visualized by the Centrifuge demo in the Havok demo framework.

3.4.7.2 Angular Velocity Clipping

The Havok Physics engine has a hard limit for the angular velocity of bodies. This is necessary for performing interpolation of body orientation between two frames. In every simulation frame bodies can rotate by no more than ∼170 degrees. At simulation frequency of 60 Hz this corresponds to the angular velocity of the wheels of a car traveling at ∼190 km/h (assuming the wheels to be 60cm in diameter).

Additionally the angular velocities of individual bodies may be limited further with a `hkpRigidBody::setMaxAngularVelocity()` call.

3.4.7.3 Linear Velocity Clipping

In order to prevent the engine from stalling some internal safety values are used. You can set the maximum linear velocity of a body using `hkpRigidBody::setMaxLinearVelocity()`. Make sure that the tolerances in the `hkpCollisionDispatcher` are tuned correctly by setting the `hkpWorldCinfo::m_expectedMaxLinearVelocity` to the highest velocity you allow in your game.

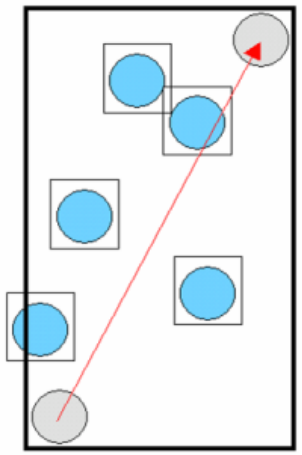
3.4.7.4 Less Efficient Use of Broadphase Collision Detection

Two positions are stored in the motion state of a rigid body - the position of the rigid body in the current physics frame and its expected position in the next frame. The `hkAabb` representation of a body in the broadphase encloses the entire volume the body passes through between the physics frames.

Note that this has changed from Havok 2, where the `hkAabb` of a body simply enclosed its volume at a given instant in time.

Linear Motion

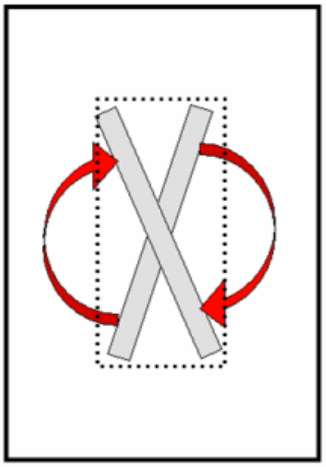
Fast projectiles have their `hkAabb`s stretched along their linear velocity vector. Note that depending on the direction of travel a `hkAabb`'s volume may be relatively small and stretched (if the projectile moves along one of the global reference frame's axes), or it may be large and stretched along 3 axes.



The above picture illustrates an emerging problem. Collision agents are created and queried as the `hkAabb` representation of the particle overlaps with other objects. On the next simulation step all agents must be deallocated.

Fast projectiles may instantly merge numerous separate islands containing stationary objects. Such islands may need to be first activated by the engine, which causes additional overhead.

Angular Motion



Long, thin, rotating objects must be appropriately expanded, as the space they move through cannot be simply based on their initial and final `hkAabb`s. Havok Physics uses a rough approximation to expand the `hkAabb`s in all directions, taking the angular velocity of the body into account.

3.4.7.5 Collision Agents

The `hkpPredGskfAgent` (predictive convex-convex) agent is the only agent that supports continuous collision detection. The agent has been improved in terms of CPU efficiency and memory usage as

compared to its predecessor `hkpGskConvexConvexAgent`. However it suffers from not being able to generate the full collision manifold instantaneously, as the box-box agent can. As a result some additional small jitter might be noticeable when stacking large numbers of small boxes.

3.4.7.6 Collision Callbacks

`hkpCollisionListener::contactPointAddedCallback`, `hkpCollisionListener::contactPointRemovedCallback` and `hkpCollisionListener::contactProcessCallback` are called for every point created by a `hkpCollisionAgent`. The collision detector generates those points optimistically and they may be changed as a result of TOI events. Therefore callbacks that are generated by high-velocity objects may be created between objects that will eventually never collide and may be perceived as side effects (see the Multi-body Collisions section for more details).

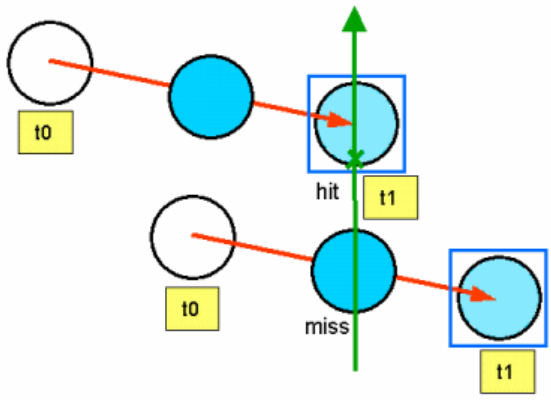
Such callbacks should be used with caution when driving game-play events. In case of a bullet it is advised to either use those callbacks for modification of the engine's information only, e.g. changing contact normals, or to perform extra contact verification.

3.4.7.7 Asynchronous User Calls

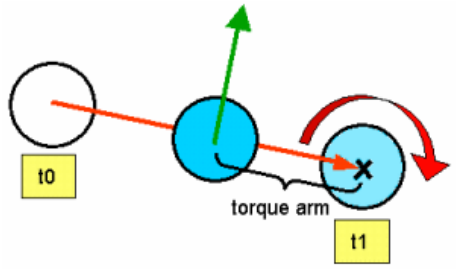
The Havok Physics engine is optimized to perform all 'out-of-main-simulation-loop' asynchronous queries and calls at the time of PSIs only. It does not store proper mid-frame data. The `hkpRigidBody` transforms used by collision detection (accessed via `hkpRigidBody::getTransform()`) are always set to the end of the PSI step (`hkSweptTransform`'s position/orientation at time `t1`).

As a result raycasts fired mid-frame may incorrectly miss or hit objects.

The following diagram shows a ray-cast (marked by the green vertical arrow) executed at frame time in-between PSIs. The objects move from the last PSI position `t0` to the next PSI position `t1`. We assume that we render a frame just between `t0` and `t1`. So if we fire our ray when we render the frame, we expect the ray to hit the upper object and miss the lower one. However the raycast uses the transforms stored in the object for time `t1`. That means our ray will hit the lower one and miss the upper object. This is only a problem if objects are moving at high velocities.



Similarly any forces applied to rigid bodies at specific points may use incorrect position of center of mass information and cause unexpected rotational motion.



In this example force is applied at the point of expected mass centre of a body. The system uses end-of-PSI-step transforms what causes unexpected rotational movement.

When running asynchronous simulation, it is a good practice to always use the `hkTransform` stored in the `hkpRigidBody` in the simulation engine, and not taking one from e.g. the graphics engine.

3.4.8 Overview - Basic Parameters

Below is a summary of the key parameters used to perform continuous simulation.

hkpWorldCinfo

- `m_simulationType` must be set to `SIMULATION_TYPE_CONTINUOUS` or `SIMULATION_TYPE_MULTITHREADED` to enable continuous simulation, or `SIMULATION_TYPE_DISCRETE` for discrete simulation.

hkpRigidBodyCinfo

- `m_qualityType` (also accessed in `hkpCollidable`) determines how the engine handles the body's interactions with other entities. Available settings are: `HK_COLLIDABLE_QUALITY_FIXED`, `_KEYFRAMED`, `_DEBRIS`, `_MOVING`, `_CRITICAL`, `_BULLET` or `_KEYFRAME_REPORTING`. For more information refer to the Object Quality Types section above. Note that the quality cannot be changed once the rigid body has been added to the world.
- `m_allowedPenetrationDepth` (also accessed in `hkpCollidable`) defines how deeply an object is allowed to penetrate in normal situations. However, the engine is allowed to breach this value to solve critical situations. The value should be small for small objects (about 5% to 10% of the object's diameter) and can be larger if quality is not a primary issue. If you reduce this value, you will get a higher quality at the expense of some extra CPU. The default value is 0.05 (metres) and works well for objects as small as 20cm.

hkpConstraintInstance

- `m_priority` determines whether a constraint should be considered in processed TOI events (PSI and TOI priorities). Higher-priority (`TOI_HIGHER`) constraints may be used to override normal-priority (PSI or TOI) when the two are in conflict.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.4.9 Overview - Advanced Parameters

The hkCollsionDispatcher's internal tolerances are set up according to parameters taken from hkpWorldCinfo (see the Collision Detection chapter for detailed descriptions):

- m_expectedMaxLinearVelocity - The assumed maximum linear velocity for all bodies.
- m_allowNegativeDistance - Allows contact points to be created at negative distance, which speeds up the simulation but doesn't work well with small objects. For small object this setting may cause TOI events being generated massively (instead of regular constact points created before interpenetration of objects) and therefore cause a drop in frame frequency. This setting is enabled when hkpWorld::m_contactPointGeneration is set to hkpWorldCinfo::CONTACT_POINT_REJECT_DUBIOUS or hkpWorldCinfo::CONTACT_POINT_REJECT_MANY (the default setting).
- m_expectedMinPsiStep - The minimum expected physical simulation step. This is only used when m_allowNegativeDistance is true. This is used to determine the distance a body travels in one frame time, having zero initial position and only gravitational acceleration applied.
- m_gravityLength - The length of the gravitational acceleration vector. This is only used when m_allowNegativeDistance is true. This is used to determine the distance a body travels in one frame time, having zero initial position and only gravitational acceleration applied.

The hkpContinuousSimulation class is used to handle continuous physics. To control its CPU resources used it uses the following functions (from hkpToiResourceMgr):

- markBeginningOfNewPsi - To mark the beginning of a new PSI step.
- beginToiAndSetupResources(hkToiSolveInfo, hkpToiResources) - To accept or reject a TOI event, and to set parameters and resources for the event.
- endToiAndFreeResources(hkToiSolveInfo, hkpToiResources)s - Called at the end of the TOI event to free up resources.
- cannotSolve() - To inform you that some of the TOI_HIGHER priority constraints have been violated.
- resourcesDepleted() - To inform you that a TOI needs more resources in order to fully expand its local group of bodies being collided and reintegrated.

3.4.10 Troubleshooting

What to do when:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

A brick wall simulation stalls the simulation.	• Increase the allowed penetration depth. • Consider changing body types to `MOVING` or `DEBRIS`
Objects penetrate walls.	• Change the object's type to `CRITICAL` • Ensure walls are of fixed type. • Decrease m_allowPenetrationDepth.
Objects sink through the walls and floor.	• Change the object's type to `CRITICAL` • Ensure the sinking object is not attached to any `TOI_HIGHER` priority constraints other than its contact constraints with the walls/floor. • Decrease m_allowedPenetrationDepth if it has been increased.
Systems momentarily fall apart when colliding.	• Make sure you use `TOI` priority constraints. • Sometimes you just can't help it, this feature is not bullet-proof.
The engine stalls.	• Check the timings file and check the number of TOI events generated in problematic frames. • Check the quality types of bodies involved and determine the types of agent involved. • Check the constraint priorities involved. • Increase `m_allowedPenetrationDepth` for the bodies involved.

3.4.11 Demo Walkthroughs

3.4.11.1 Continuous basics

This demo introduces the basic concepts of continuous physics. Basically a simple ball is shoot against another ball. Once the balls collide a line is drawn (and remembered) in which direction the balls travel after collision. We then continuously iterate the game. As Havok is deterministic, repeating the demo would give the same results over and over again. So to make things more realistic, we vary the starting position of the ball at each iteration. As a result the balls will hit each other at different times. This is similar to what happens in a game, where it is not defined when exactly a player's car will hit the road block.

Slow ball hitting other ball

This scenario is simulated using discrete physics. As we simulate the demo at 60Hz, there are enough timesteps available when the two balls collide. So this demo works as expected.

Faster ball using discrete physics

Here we simply increase the velocity of the striking ball. As discrete physics only has discrete time steps and the balls are colliding at varying times, it is kind of random how the balls are positioned when the physics time step is simulated. This gets more obvious when running the demo in slow motion. As a result, the outcome of the collision vary each time dramatically.

Fast ball in slow motion

This shows the previous example frame-by-frame. Notice that at the moment when collision response is applied the balls may either just touch or deeply interpenetrate. Such varying input information causes the collision results to be hard to predict. As we only perform collision detection at discrete moments in time we hardly ever happen to apply collision response at the exact time of impact of the two bodies. We are only able to apply it before the collision (using extra collision tolerance boundary which adds a thick layer around the bodies geometries) or after the actual collision in that case bodies already penetrate because their state was integrated from the previous frame by a fixed timestep without consideration of this collision.

Fast ball in slow motion interpolated

This demo introduces the concept of continuous motion. Havok3 represents the state of bodies with the hkSweptTransform structure which holds information about a body position and orientation at two specified moments in time. The transparent shapes visualize those end positions of the hkSweptTransforms of the moving bodies. Notice how hkSweptTransforms are only updated at the time of simulation step and how the current position of bodies is simply interpolated from them. If you use this interpolation technique for extraction the transforms of the physics for your graphics engine, it allows us to run the physics at a different frequency as the graphics (like in this demo) (e.g. by using hkAsynchronousSimulation).

With physics clock

This demo introduces the clock commonly used in the continuous physics demos. The clock is located in the lower left corner of the screen. The blue lines correspond to moments when physical simulation is run (called physical synchronuous instance = PSI). The green line indicates the current time of visualization. Whenever the green line passes over a blue line, a PSI is executed. Then, until the green line reaches the next blue line, the physics engine is suspended. To preserve the illusion of continuous motion of the bodies, the rendering engine calculates their positions from their hkSweptTransforms at every frame. The hkSweptTransform stays constant in-between the physical steps.

Very fast ball

This example accelerates the striking ball to even higher velocity. Many of the collisions are simply missed (or not recorded) by the physics engine. However even when they are recorded, the collision response outcome is highly unpredictable.

Very fast ball with continuous physics

hkpContinuousSimulation calculates the moment of impact of the colliding bodies. It then uses hkSweptTransform to determine their position at that time and applies precise collision response.

Note:

The majority of the iterations result in exactly the same outcome. However with time you may observe a slight variation in the final velocity of the objects. This inconsistency is related to the fact that somtimes the balls hit between two PSIs, sometimes the balls hit exactly at a PSI (within a certain tolerance). In both cases different code paths are used to process the collisions:

- If the collision is handled in between the main physical simulation steps, it is handled at its exact time of impact via a special time-of-impact event handler.
- When the time of impact is however close enough to the moment of a main simulation step then the collision is simply handled in the main simulation step

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.4.11.2 Discrete vs Continuous

A scenario where very-fast-moving cubes are collided against a landscape modelled with mesh geometry. Such scenario is likely to fail with discrete physics as fast objects can easily tunnel through the walls of zero thickness. Havok3 allows you to mark objects with the HK_COLLIDABLE_QUALITY_BULLET property which ensures that continuous collision detection is performed between bullet-objects and fixed landscape objects. Notice how the fast objects are shot into the corners of the simple landscape and how they may cause more than one impact collisions per frame. After their first collision the bodies are reintegrated and again re-collided with the environment. This yields the information necessary to process all subsequent collisions occurring before the next main simulation step.

3.4.11.3 High speed collision

Presents a grid of spheres, placed one next to another, that is hit by a fast particle. This demo presents a good use case for localized solving algorithm and for the hkpRigidBodyCinfo::m_allowedPenetrationDepth parameter. If localized solving was not used this example might cause a huge number of TOI events, as every application of simple collision response would be followed by more collisions with surrounding bodies in contact. Localized Solving brings the number of TOI events down by an order of magnitude. Also notice that the spheres are assigned a relatively high allowed penetration depth parameter, which allows for a major interpenetration before a TOI event is triggered. This is the reason for which the particle initially sinks-in into the grid of spheres and only then properly disperses its kinetic energy.

3.4.11.4 Squeezed ball

This demo illustrated the results of usage of incremental allowed penetration. As the small ball is being squeezed it bounces more and more often between the two slabs, finally it is allowed to penetrate into the moving slab (as it has a lower surface quality than the fixed slab) in order to keep the number of TOI events reasonable. See section 5.2 for more details.

3.4.11.5 Hinge hitting table

This demo shows the importance of using TOI_PRIORITY constraints in systems composed of bodies that use continuous collision detection. The first three variants of the demo show a hinge composed of two rigid bodies and a hinge constraint: 1. using discrete simulation, 2. using continuous simulation but with a non-TOI_PRIORITY constraint, and 3. using continuous simulation and a TOI_PRIORITY constraint. When the hinge hits a fixed obstacle, the first two variants fail to enforce the hinge constraint. The third example is simulated properly, as the hinge constraint is considered in the localized solving algorithm at each TOI recorded between the hinge bodies and the fixed obstacle. The next three examples show a ragdoll with analogical setting of discrete/continuous simulation and priority settings of the ragdolls constraints. The last two examples introduce the idea of incrementing the rotational inertia tensor for ragdoll limbs. This trick is an efficient way of improving the stability of the system, as it reduced the angular components of motion in the solver.

3.4.11.6 Three way squeeze

This demo illustrates use of normal- and higher-priority constraints.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.4.11.7 Centrifuge

This demo illustrates the problem of high angular velocities.

3.5 Multithreading

3.5.1 Introduction

The Havok SDK can be used in several different multithreaded configurations. The next section outlines these and highlights the areas of the SDK you need to pay particular attention to when using multithreaded simulation.

3.5.2 Traditional Single Threaded Approach

In a traditional approach, the different algorithms are ideally called in this order:

- User input
- Game Engine with Physics
- Graphics

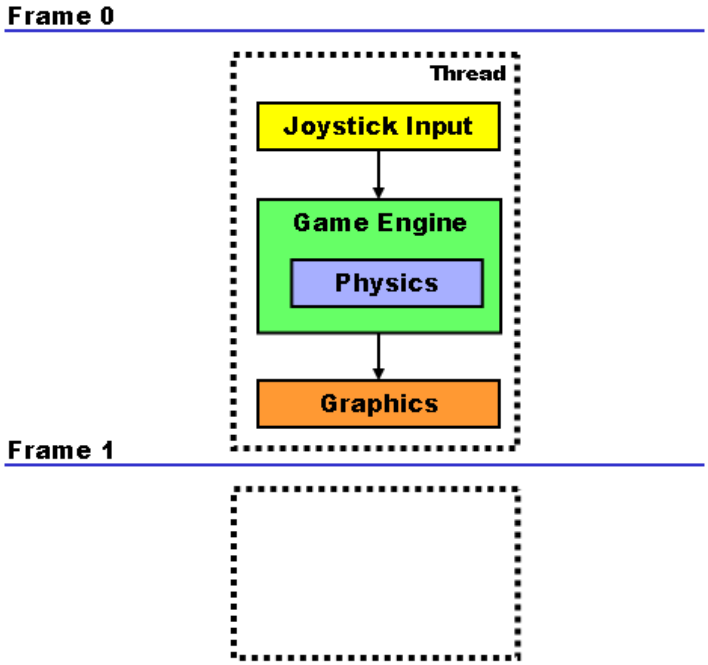


Figure 3.56: Traditional pipeline. All jobs are done sequentially.

This approach ensures that the user input is processed and rendered on the next frame with no delay. In this configuration the interface between the game engine and physics can be very tightly coupled. For example, the game engine will directly call the physics engine for proximity queries

3.5.3 Running graphics in a separate thread from the main game loop

Typically, after initialization, the interface between a game engine and graphics engine is just an array of transforms. So running graphics in a separate thread from the game loop is relatively straightforward.

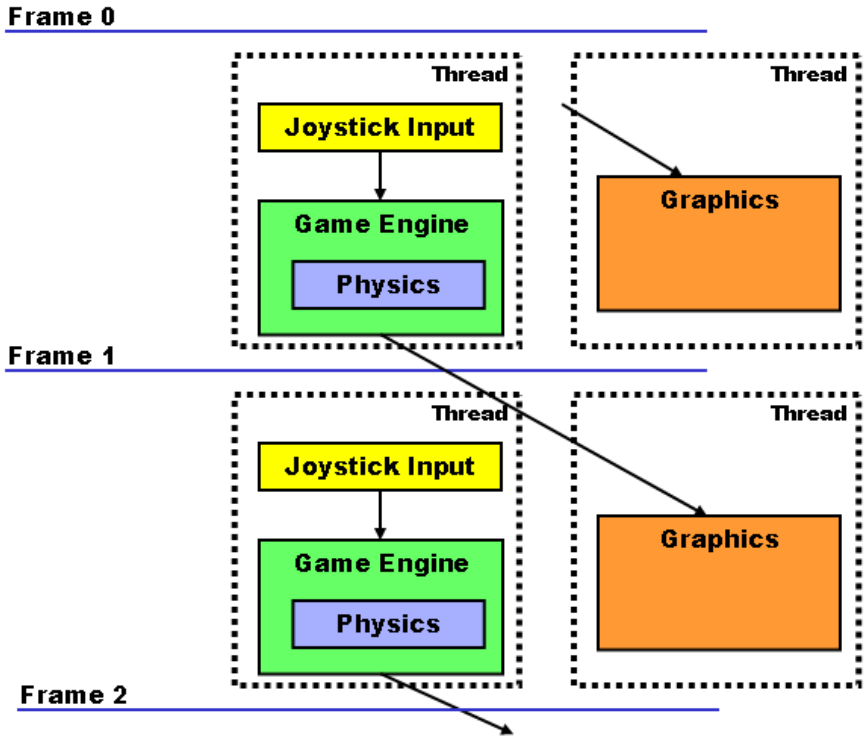


Figure 3.57: Graphics runs in a parallel thread. Graphics output may be delayed by up to one frame.

3.5.4 Running physics in a separate thread from the main game loop

One easy way to use a multi CPU system is to offload the entire physics simulation onto a second CPU. This is relatively easy to do. The only difficulty is in managing the communication between game engine and physics. There are fundamentally two approaches:

- If you just want to run the physics in a separate thread you need to decouple the communication between the game engine and physics simulation using some data structure e.g. a command queue. However, controlling the behavior here can be tricky, especially if you use the simulation for asynchronous queries. As a naive example, say you jump through a teleporter just before a bomb goes of nearby. The game logic places a command into the queue at time t "Teleport Player". Then the bomb goes off, also at time t,and uses the current game state to query for all objects in the vicinity (this is done in the game loop). It will also find the player object as the player in the physics is still not teleported yet (The command may be still in the queue and not yet processed). A new

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

command may be added which applies an impulse to the player . When the player appears on the other side of the teleporter they incorrectly receive a large impulse.

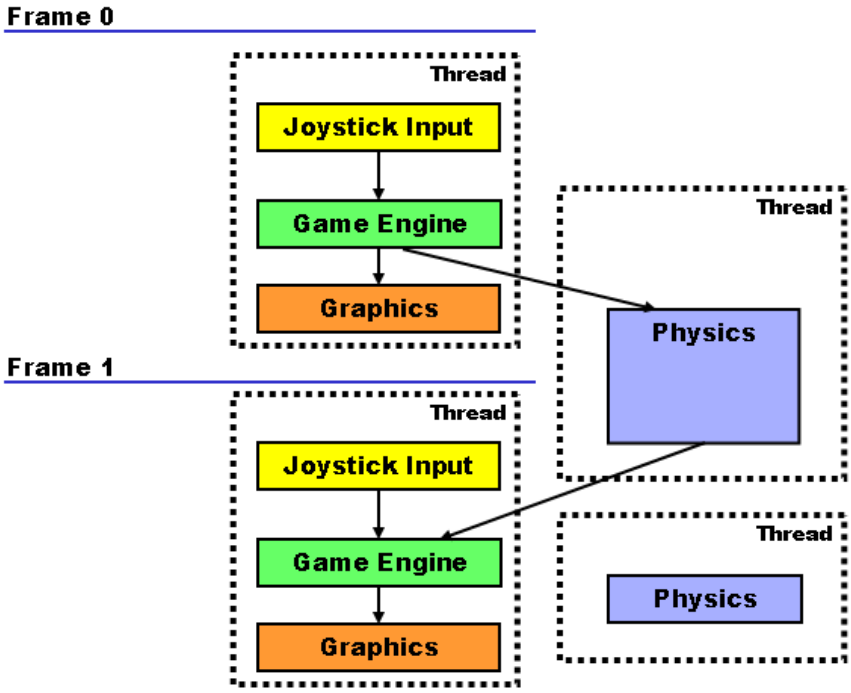


Figure 3.58: Physics runs in a parallel thread. Physics output may be delayed by up to one frame. Note that interface between the Game Engine and Physics may become complicated and inflexible.

- A better approach to this is to split the game engine into a portion wrapping the physics and a portion independent of physics. As a result you can send higher-level commands to our wrapped physics. In our example the commands will be: "Teleport Player" and "Explode Bomb". But they are processed synchronously.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

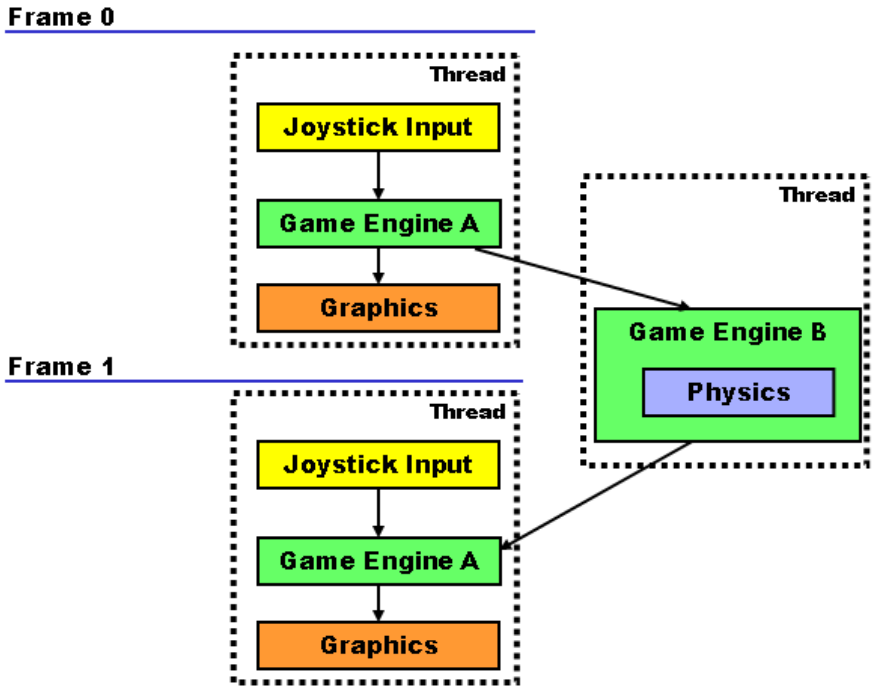


Figure 3.59: Game logic is split into two portions – one dependent and one independent of physics. This allows to simplify the physics-to-gamelogic interface. Inter-thread communication might be handled by a command queue.

3.5.5 Splitting Physics

Very often physics engines and game engine are very tightly coupled. In this case it is not possible to completely offload the physics onto a separate thread. A possible solution is to partition the physics world into a gameplay critical partition and an optional special effects physics. All objects in the gameplay physics partition are treated as keyframed objects in the special effects partition. The special effects physics partition has no influence on the game play. This has a few nice side effects:

- The special effects physics is totally optional. There is no need to save/serialize it or to simulate it at all on lower spec machines.
- The communication to the special effect physics is typically pretty simple.

The demo `SpecialEffectsPhysics` shows this in operation

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

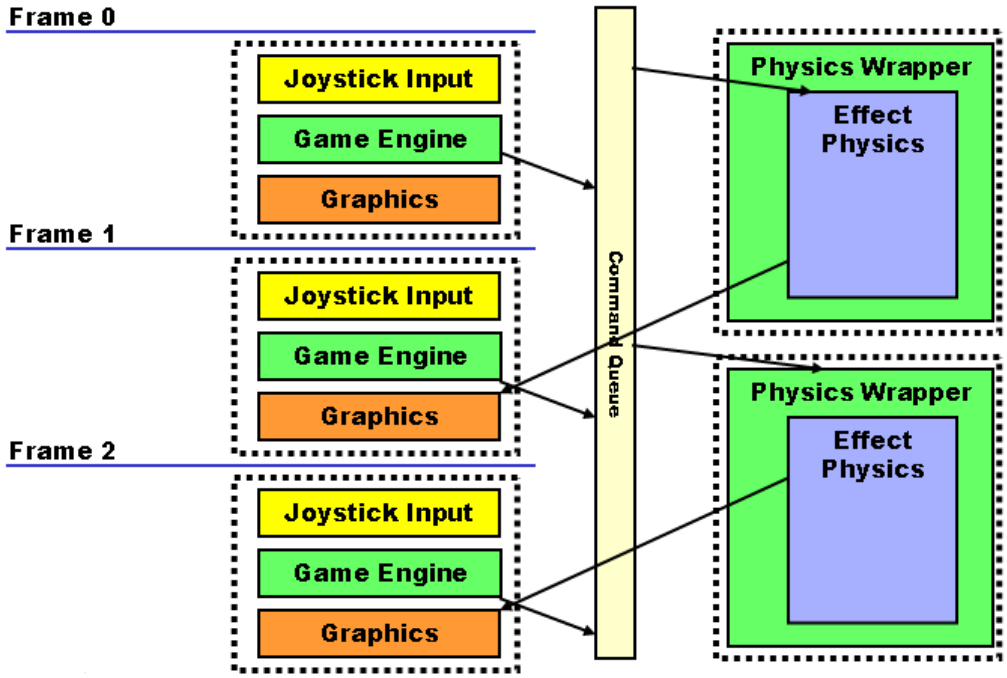


Figure 3.60: Game logic sends physics commands to a thread-safe command queue. The graphics engine may simply reference unlocked transforms from the physics engine (assuming that physical entities are not deleted or moved in memory).

An advanced version of this approach is to run the special effects physics asynchronously to the graphics update rate. You can access the transforms of a rigid body in an unsynchronised manner as long as no rigid bodies are added or removed from the scene.

Note:
In Havok all transforms actually consist of two sets of position and orientation, and we only need to interpolate the final transform.

If the special effect physics needs more than 100% of allocated CPU time, we simply put the special effect physics into a slow motion mode, giving it a chance to catch up with the games frame rate. In an actual game, this would require a bit of fine tuning, so this approach is not really trivial. As a result you get:

- Guaranteed frame rate as the graphics is never really waiting for the physics.
- As it does not matter whether the physics actually finishes in this very frame, you can use the same thread to also execute some extremely CPU intensive code e.g. for sound processing and use the physics to compensate for your CPU spiky sound code.

The demo and `AsynchronousSpecialEffectsThread` shows this in operation.

3.5.6 Running Physics Multithreaded

While we support the configurations above our fully multithreaded solution differs by *balancing the physics workload over several CPUs*. We split the physics calculations into a large number of independent micro jobs. The first thread entering the physics via hkpWorld::stepDeltaTime() will initialize the job queue and

perform some single threaded maintenance operations. Then, along with other threads, it will continue executing pending jobs until all are solved. In the end this thread will solve high-velocity collisions detected by continuous simulation in a single threaded way.

Any other thread which enters the physics using `hkpWorld::stepDeltaTime()` will jump straight to the job queue and start executing scheduled jobs. That means you can add threads to the physics as they become free from other tasks.

When all jobs are complete you call hkpWorld::m_simulation->resetThreadTokens(), indicating that the next thread should reinitialize the job queue.

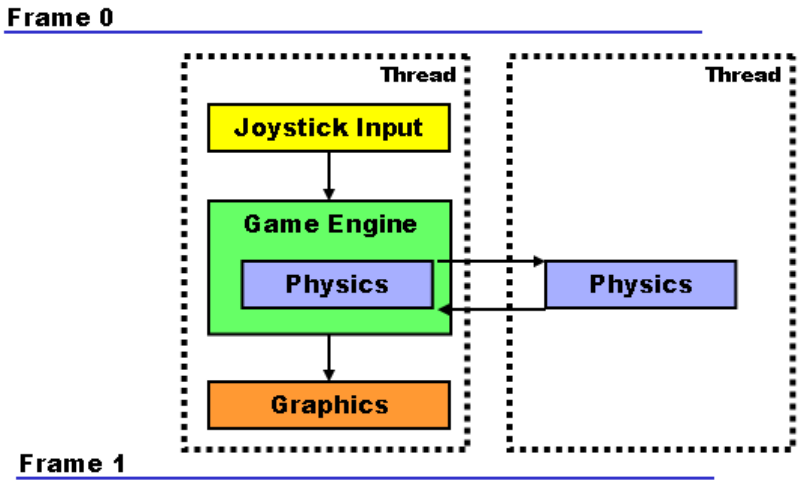


Figure 3.61: Running multithreaded physics is easy, but complex physics callbacks may be hard to implement and be error prone.

So basically to run Havok in fully multithreaded mode:

- You create your world using `hkpWorldCinfo::.m_simulationType = hkpWorldCinfo::SIMULATION_TYPE_MULTITHREADED`.
- Create your threads in a number appropriate for your hardware. E.g. if you have 6 processors and one thread is executing graphics, you should not use more than 5 threads for the physics. Otherwise you will get a significant slowdown.
- Each thread gets its local `hkThreadMemory` with its own stack area, which basically caches frequent access to the main memory manager.
- Instead of calling `hkpWorld::stepDeltaTime()` from the main thread directly, you simply tell all your threads to call `hkpWorld::stepDeltaTime()`. When all threads are finished, you have to call `hkpWorld::m_simulation->resetThreadTokens()`

3.5.6.1 Multithreading interface changes

Please note that from Havok 6.0 on the multithreading interfaces for the Multithreading utility will be deprecated and replaced with a simpler thread pool interface.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.5.6.2 Basic threading and synchronization

An example main loop for a worker thread is as follows:

```
struct hkThreadData
{
    int m_stackSize;
    hkReal m_physicsDeltaTime;
    hkSemaphore m_workerThreadFinishedSemaphore;
    hkSemaphore m_mainThreadSemphore;
    hkpWorld* m_world;
};

static void* HK_CALL WorkerThreadFunc(void *v)
{
    // The thread must be given some data through the void* v parameter passed in.
    hkThreadData* data = (hkThreadData*)v;


    // Initialize the thread memory for this thread, referencing the global memory instance
    hkThreadMemory threadMemory(&hkMemory::getInstance(), 16);
    {
        hkBaseSystem::initThread( &threadMemory );
    }

    // Allocate stack memory for this thread
    char* stackBuffer;
    {
        // stackSize is your desired stack size for this thread
        stackBuffer = hkAllocate<char>( data->m_stackSize, HK_MEMORY_CLASS_BASE);
        threadMemory.setStackArea( stackBuffer, stackSize);
    }

    // Wait for the main thread to release the physics thread
    data->m_mainThreadSemphore.acquire();

    // The thread "main loop"
    while ( killThreadFlag == false )
    {
            // step the world
        data->m_world->stepDeltaTime( data->m_physicsDeltaTime );

        // Release any thread (usually the main thread) which may be waiting for all worker threads to
          finish.
        data->m_workerThreadFinishedSemaphore.release();

        // Immediately wait until the main thread releases the physics thread again
        // - we do nothing but physics in this thread
        data->m_mainThreadSemphore.acquire();
    }

    // Deallocate thread resources
    threadMemory.setStackArea(0, 0);
    hkDeallocate(stackBuffer);
    hkBaseSystem::clearThreadResources();

    // Release the main thread
    data->m_workerThreadFinishedSemaphore.release();
}
```

This thread simply simply loops around, calling stepDeltaTime, until signalled to exit by another thread. It has memory setup and quit functionality at the end. The thread must be given some data through

the void* v parameter passed in. This data includes two semaphores, mainThreadSemphore and workerThreadFinishedSempahore, and a flag (killThreadFlag) used to signal termination by the main thread. The main thread executes a loop as follows:

```
void main()
{
    hkThreadData data;
    // Init Data here

    // Create threads
    spawnWorkerThreads( data );

    // Main loop
    while ( gameRunning )
    {
        // This must be called in single threaded mode
        m_world->resetThreadTokens();

        // Release all threads by releasing the main sempahore once for each thread
        releaseSemaphore( data.m_mainThreadSemphore, numThreads );

        // Step the world with this thread. We could do other work here instead, such as AI, preparation
          for rendering, etc..
        m_world->stepDeltaTime( data.m_timestep );

        // Wait until all threads are finished, by acquiring the thread finished semaphore once for each
          thread
        waitForSempahore( data.m_workerThreadFinishedSemaphore, numThreads );
    }
}
```

This code is pseudo code, but it shows clearly what the main loop needs to do. Each time round the main loop it uses the two sempahores to synchronize with the worker threads. resetThreadTokens() must be called in single threaded mode prior to releasing the worker threads. The main loop can also process physics with the worker threads if if talls stepDeltaTime() as shown. However ther is nothing preventing you from executing other code here instead, while the physics is runing (as long as you are aware of the physics synchronization issues outlined below).

3.5.6.3 The Multithreading Utility

The worker thread code in the previous section is actually a simpilfied version of the code in the hkMultithreadingUtility class. This class is designed to make multithreading any simulation very straightforward - you do not need to write the thread main loop yourself if you use this utility, simply make calls to the utility from your main thread.

After creating your `hkpWorld`, just pass it as an initialization parameter to a new instance `hkpMultithreadingUtil`.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
m_world = new hkpWorld( info );

// Worlds using multithreaded simulation must mark/unmark it for writes/reads before/after every access.
m_world->markForWrite();

hkpMultithreadingUtilCinfo mtcinfo;
mtcinfo.m_world = m_world;
mtcinfo.m_numThreads = numThreads;

multithreadingUtil = new hkpMultithreadingUtil(mtcinfo);

// Worlds using hkMultithreadedSimulation must mark/unmark it for writes/reads before/after every access.
m_world->unmarkForWrite();
```

The above code creates `numThreads` threads to handle the `world`'s simulation.

Note that the code constructs a hkpWorld and then marks it for write. The marking mechanism is a debug mechanism - it does not perform any locking. Internally it sets flags in the engine to indicate that you are certain that you are exclusively accessing the world. Methods inside the engine check the state of these flags to ensure simulation integrity. This utility is particularly useful when you have a callback in the engine and are unsure about which methods are safe or unsafe to call on a multithreaded simulation - if you access something unsafely the engine will assert. The section 'Synchronization Issues' below describes this in more detail.

Now, instead of calling `m_world->stepDeltaTime(timeStep)` directly call `multithreadingUtil->stepWorld(physicsDeltaTime)` or interleave physics calculations with your own computations with the following code. Remember to call m_world->ResetThreadTokens() single threaded from your main thread first.

```
m_world->resetThreadTokens();

multithreadingUtil->startStepWorld( timestep );
...
Your own code which does not reference the hkpWorld
...
multithreadingUtil->waitForStepWorldFinished();
```

You can arbitrarily add or remove physics threads between subsequent calls to `hkpMultithreadingUtil::stepWorld()` (when physics threads are suspended) with calls to `hkpMultithreadingUtil::setNumThreads(numThreads)`.

To kill all physics threads (other than the main one) call simply delete the multithreading utility:

```
delete multithreadingUtil;
```

To see an exmple demo using the multithreading utiltiy, see the demo physicsApi/dynamics/multithreaded/PhysicsInSeperateThread.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.5.6.4 Fine-grained multithreading

In the hkpWorld interface, there is finer grained control over multithreading than the stepDeltaTime implementation discussed above. When stepDeltaTime is called, the system must first perform some single threaded operations before all threads can process the simulation in multithreaded mode. When the multithreaded part is complete, there is once again a single threaded phase where one thread needs to perform continuous simulation etc. In fact as soon as the multithreaded part of the simulation is complete, all threads but one return from stepDeltaTime, the remaining thread is left to perform cleanup and continuous simulation. The functions to use for this finer grained control are stepBeginSt(), stepProcessMt() and stepEndSt(). The usage of these functions is shown in the following code snippet.

```
{
    hkpThreadToken token = state.m_world->getThreadToken();

    if ( token == HK_THREAD_TOKEN_PRIMARY )

        m_world->stepBeginSt( state.m_deltaTime );
        m_stepInitSemaphore.release( state.m_numThreads );
    }
    else
    {
        // Secondary threads could do additional work in this section, and could poll the
          m_stepInitSemaphore
        // in order to check whether stepBeginSt is finished.
    }

    m_stepInitSemaphore.acquire();

    m_world->stepProcessMt( token );


    if ( token == HK_THREAD_TOKEN_PRIMARY )
    {
        m_world->stepEndSt( );
    }
}
```

This code is designed to run on multiple threads. It is taken from the multithread util worker thread main loop. getThreadToken is first called to get the thread "Token". Here we assume that a separate main thread has called hkpWorld::resetThreadTokens(). This will be either HK_THREAD_TOKEN_PRIMARY (for the first thread that calls this function) or HK_THREAD_TOKEN_SECONDARY (for all subsequent threads). Note that if you have already identified your "primary" thread, there is no need to use this getThreadToken() / resetThreadTokens() functionality.

The thread with the primary thread token calls stepBeginSt and stepEndSt at the end in single threaded mode as shown above. All secondary threads wait until stepBeginSt is finished (via the m_stepInitSemaphore) before calling stepProcessMt(). All threads other than the primary thread exit the loop after stepProcessMt returns.

The code above allows you to easily avoid wait times in secondary threads during single threaded sections of the simulation.

3.5.6.5 Asychronous simulation and multithreading

The hkpMultithreadingUtil utility actually provides functionality to facilitate asynchronous stepping. As in the hkpWorld interface there is a call in hkpMultithreadingUtil::setFrameTimeMarker() which will cause the main loop to step until the marker is reached. For a discussion on asynchronous stepping please refer to the Asynchronous Stepping section of the dynamics chapter. The code to perform multithreaded asynchronous stepping is well documented in the WorkerThreadFunc() in the multithreading utility. It is effectively a combination of the asynchronous stepping code from hkAsynchronousTimestepper::stepAsynchronously() and the synchronous stepping version of the worker thread step function.

It is not recommended that you use the multithreading utility in asynchronous mode. This is for two reasons. Firstly the caveats outlined in the asynchronous stepping section apply. Secondly, if the utility takes multiple simulation steps, the thread utilization will be quite poor. For each step, all threads other than the primary thread will wait while the primary thread performs all continuous simulation. If you wish to use asynchronous simulation and multithreading we recommend that you take the code from the worker thread function in the multithreaded utility class and integrate it into your own game code.

3.5.6.6 Determinism and multithreading

Havok physics runs deterministically in multithreaded mode. However, it is quite easy to interact with the phyiscs in a way that produces non-determninistic behavior in a multithreaded environment. One way to be particularly aware of is the use of actions. Actions typically apply impulses to objects upon which they act. However if an action applies an impulse to an object outside its simulation island (i.e. an object that is not in the list of entities reported by getEntities()), then undetermined behavior will result - the object may be being processed in a separate thread simulaneously. Also if an action depends on a phantom's overlaps, it is not safe to check this in multithreaded mode, because any other simulation island being processed in another thread may penetrate the phantom during the action apply() call, and undetermined behavior will result. The hkVehicle is an action which does both these things. For this reason there is a flag hkpWorldCinfo::m_processActionsInSingleThread, which, when set to true, causes all actions to be processed in a single thread, before other threads can process simulation islands.

Non-deterministic behavior can result if you apply impulses to the physics in a user thread while the physics step is running. See the section on Synchronization Issues for help on how to detect these situations.

3.5.6.7 Continuous simulation and multithreading

In a continuous simulation, Time Of Impact (TOI) events have to be solved sequentially and 2 TOIs cannot be solved by 2 independent threads. As a result the possibilities of multithreading TOI solving are limited. However the collision detection part of each TOI can be parallelized and solved by different threads. As a result you should see a TOI solving speedup if a fast complicated (e.g. listshape) object hits a pile of other complicated objects. However you will not get any speedup (but rather a slow-down) if a simple object hits another single object, like a ragdoll hitting a landscape.

3.5.7 Memory Manager

We split memory management into:

- A global thread safe `hkPoolMemory` (or `hkDebugMemory`)
- A thread local `hkThreadMemory`, which
  - Works completely transparently
  - Caches frequently accessed blocks
  - Handles stack allocations
  - Can be disabled

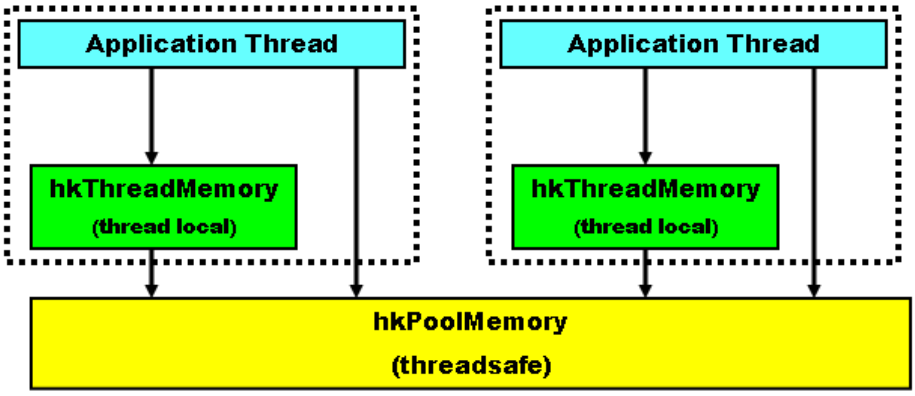


Figure 3.62: hkThreadMemory caches access to hkPoolMemory.

3.5.8 Synchronization Issues

The Havok SDK is not thread safe (except for the memory manager).

This is deliberate because

- Creating a fully thread safe engine would result in a very slow engine.
- A thread safe engine does not mean that the game using the engine is thread safe: Example: You shoot a ray and want to delete the object the ray is hitting. Lets assume Havok was thread safe and you are running multiple threads. So you fire the ray into the thread safe engine and identify an object, then you make a call to delete the hit object. Unfortunately between the call to cast ray and delete object, another thread was schedule and has deleted the object as well. As a result the engine crashes.

Most synchronization in our simulation is left explicitly to the user. We do not expose semaphores which control when it is safe to perform different operations, instead we leave this to you and simply warn and assert when synchronisation has to be enforced.

- `When you call hkpWorld::stepDeltaTime()` we internally manage the synchronization of the simulation by splitting the work into discrete jobs and distributing these among the available threads. The constraint solving and actions are distributed per island. In callbacks you have restricted access to different objects. For example if you apply an impulse to a random object in an action (not in the same island) then this is not appropriately syncronized - another thread may be writing to that object.

- If you are performing asynchronous queries which are only reading from the engine (read only access) then several threads can use the engine simultaneously. If one thread tries to access an operation, which requires a write, the engine will assert in debug mode. In this case the user is responsible for synchronizing all access to the Havok engine.

Havok has a debug utility build in that helps to identify missing or wrong synchronization. If you enable the multithreaded simulation in the world constructor, the engine will guard all functions to the physics system.

3.5.8.1 Synchronizing Callbacks

In a physics callback we cannot synchronize access to the physics (the `hkpMultiThreadedSimulation` is responsible for this). However many of the functions are actually thread safe as they are not executed immediately but put on a thread safe command queue. Many of the other functions also have a thread safe version. E.g. `hkpRigidBody::activate()` is not thread safe, but `hkpRigidBody::activateAsCriticalOperation()` is.

In debug the engine will assert if you call the wrong function.

Note You should never call `hkpWorldObject::markForWrite()` in a callback, doing so simply bypasses the synchronization checks.

3.5.8.2 Synchronizing Asynchronous Queries

To enable access to checked functions, you should synchronize your game engine before accessing Havok. In this case you than can tell the hkpWorld that you have synchronized by calling one of the following functions:

- `hkpWorld::markForWrite()`
- `hkpWorld::markForRead()`
- `hkpWorld::unmarkForWrite()`
- `hkpWorld::unmarkForRead()`

Examples:

- A single thread is creating and populating an `hkpWorld`. As the thread knows it is the only possible user of `hkpWorld` it calls `hkpWorld::markForWrite()`, just after the `hkpWorld` is created. Once the `hkpWorld` is populated, it call `hkpWorld::unmarkForWrite()`.
- In our ray cast, delete object example, from above, we have to create a single critical section object in our game guarding the physics access. Whenever we want to access the physics, we enter our critical section, mark the physics for write, do our work, unmark the physics and leave the critical section.

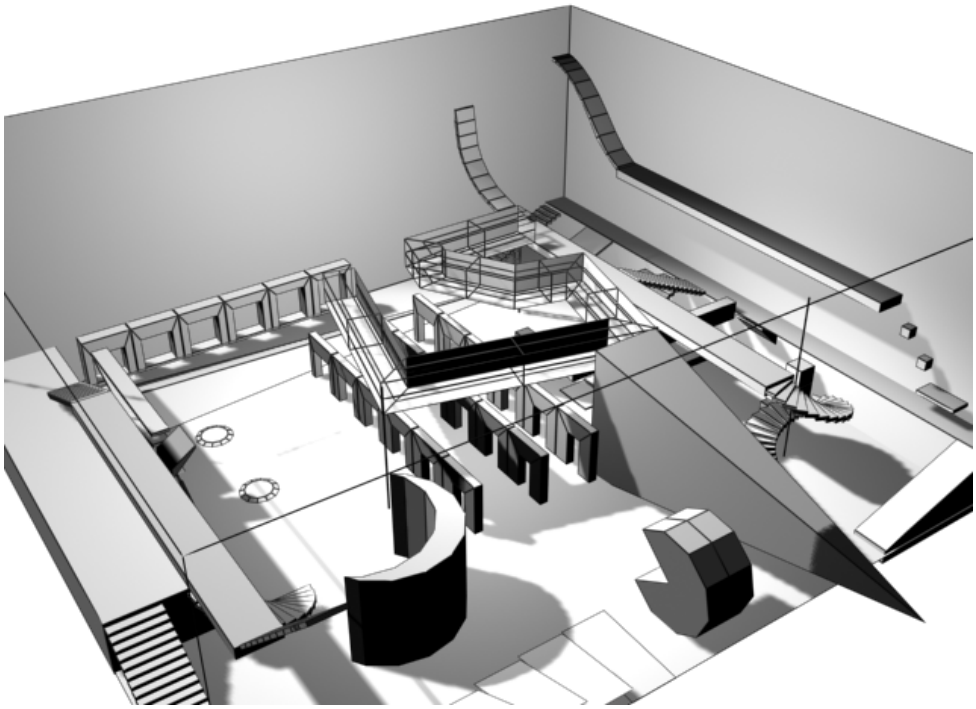
3.5.8.3 Warning

If possible do not call the `hkpRigidBody::markForWrite()` functions, try to use the `hkpWorld` version instead. If you call `hkpWorld::markForWrite()` you are indicating that you have exclusive read write access to the entire engine. If you have to call `hkpWorld` or `hkpRigidBody::markForWrite()`, be absolutely sure that you have synchronized your thread before calling this function.

Havok's debug access checks are very helpful, but if you call `markForWrite()` without doing proper synchronization you will have disabled the engine checks without solving the problem (its like disabling an assert). Eventually the engine will crash in a non-deterministic way.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.6 Character Control



3.6.1 Introduction

This chapter provides an overview of character control strategies in Havok. The chapter is designed as an overview of the core classes and methods that facilitate integrating our character controllers, or your own custom solution.

3.6.2 Overview

Often each character controller will have a very distinctive feel to entire gaming. Havok provides two different version of character controllers:

- The Rigid Body Character Controller

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- The Proxy Character Controller

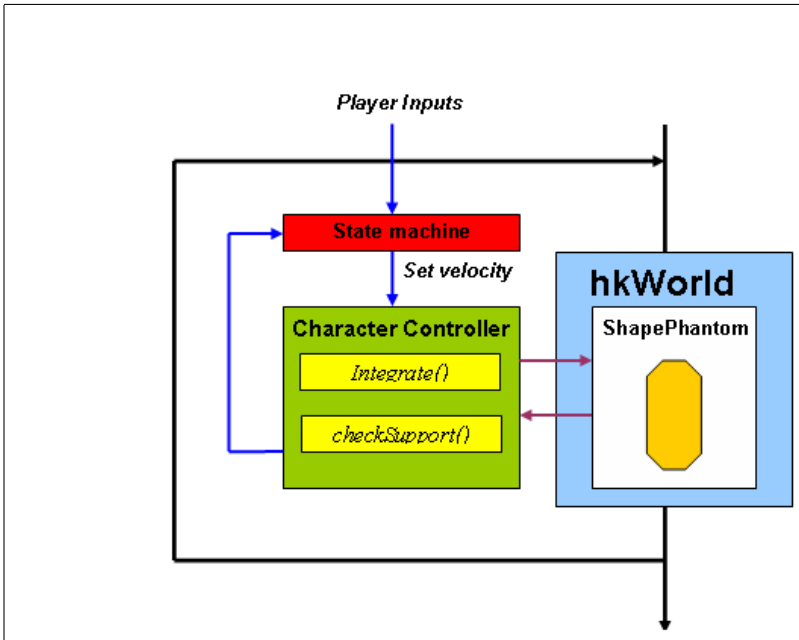
The main differences in the integration philosophies between the rigid body and proxy character controller are shown in the figure below. The rigid body character sits inside the normal simulation loop and it is an integral part of the dynamic world (`hkpWorld`). The rigid body is controlled by directly setting its velocities (linear, and in specific cases also angular). The proxy character controller sits outside the normal simulation loop and it is represented only with the shape phantom (`hkShapePhantom`) in the world. It's linked with the physical simulation in two ways. The first way takes the collision information from the simulation via shape phantom and the second way puts information into the simulation in the form of impulses applied to rigid bodies. Both controllers use the same state machine (`hkpCharacterContext`) to control the character state transition and calculate the required control velocity according to player inputs. Each controller has its own implementations of the `checkSupport()` method which provides informations about the surface under the character. More details about controllers can be found below in separated sections.

The rigid body character controller has several advantages over the proxy character controller. These include:

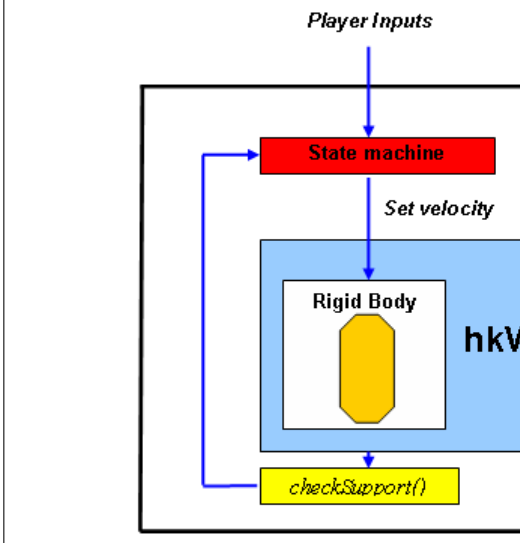
- Efficiency. The rigid body simulation is much more efficient than the phantom version. This is because of many internal optizations in the rigid body simulation calculations.
- Multithreading. On multithreaded platforms, the rigid body character controller multithreads naturally, as opposed to the proxy version which does not.
- PLAYSTATION®3 only: The rigid body character controller will run on the SPU, whereas the proxy version will not
- Code paths. Using the rigid body controller should actually result in simpler code, as the regular callbacks for rigid bodies are the ones that need to be leveraged to customize its behavior
- Character - rigid body and character - character interactions will be handled better by the rigid body character controller.

With these factors in mind we recommend using the rigid body character controller over using the proxy controller. However there are a number of caveats to be aware of in this choice. The behavior of the rigid body character controller may not be as smooth as the proxy controller, particularly for stair climbing. Also the character controller currently has less in game testing than the proxy controller.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



The Proxy Character Controller



The Rigid Body Character Controller

Table 3.5: Different integration philosophies for controllers

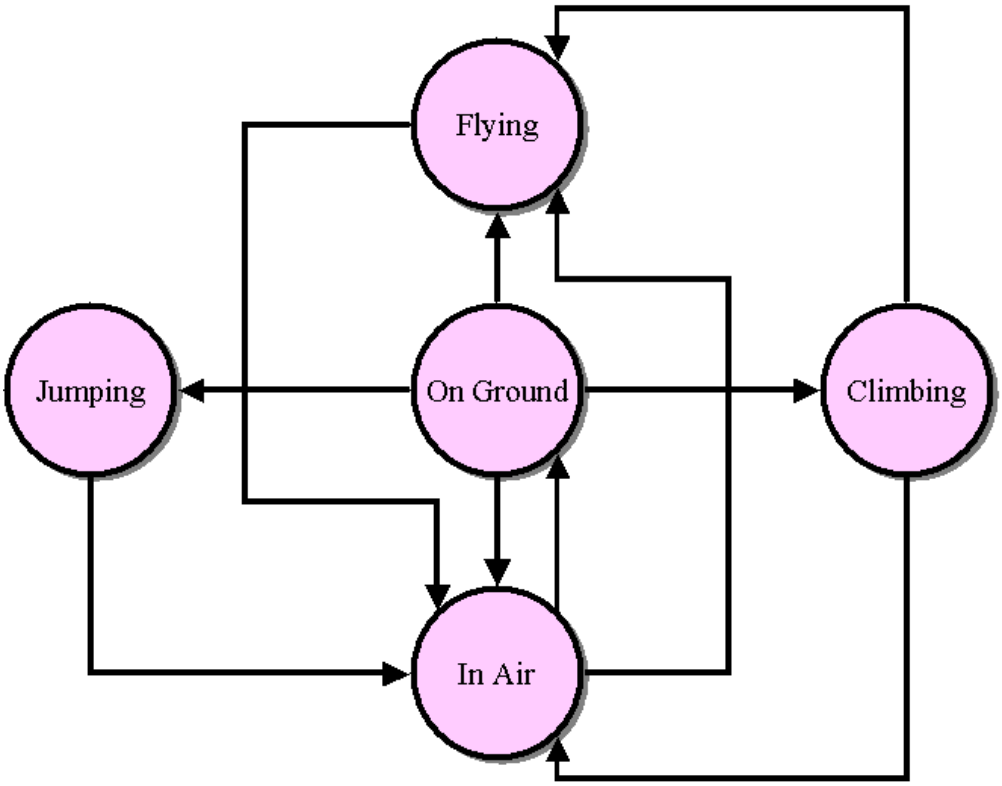
The structure we provide is designed to be extended and replaced as you see fit. The controllers consist of the following major distinct components

- The State Machine
- The Character Rigid Body or The Character Proxy
- The checkSupport Method
- The Simplex Solver
- The Interaction Listener

These components can be coupled together or used in isolation to produce the required behavior. The state machine and the simplex solver are the same for both controllers. The simplex solver is a major part of the character proxy movement calculation (`hkpCharacterProxy::integrate()`) and it is also used in both implementations of the `checkSupport()` method. The character rigid body and the character proxy are completely different as described above.

3.6.3 The State Machine

A reusable state machine controls the transition between discrete character states. The currently implemented state and their transitions are shown below. Full source code for the state machine can be found in `Physics\Utilities\CharacterControl\StateMachine`



The state machine is constructed through a state manager. The state manager allows the user to construct and register their own specific states at run time. Each state machine is designed to be shared. Usually one instance will exist for each type of character in your game e.g. one for Humans and another for Orcs. To construct a state machine you must first register each of you desired states with the state manager, for example

```
hkpCharacterState* state;
        hkpCharacterStateManager* manager = new hkpCharacterStateManager();

        state = new hkpCharacterStateOnGround();
        manager->registerState( state,    HK_CHARACTER_ON_GROUND);
        state->removeReference();

        state = new hkpCharacterStateInAir();
        manager->registerState( state,    HK_CHARACTER_IN_AIR);
        state->removeReference();

        state = new hkpCharacterStateJumping();
        manager->registerState( state,    HK_CHARACTER_JUMPING);
        state->removeReference();

        state = new hkpCharacterStateClimbing();
        manager->registerState( state,    HK_CHARACTER_CLIMBING);
        state->removeReference();
```

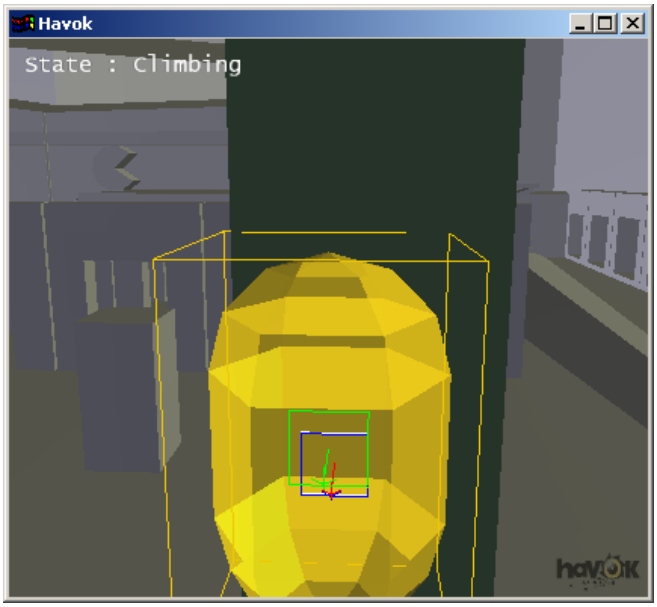
As you can see references to new states are managed appropriately by the state manager.

Once you have registered your state you create a context for each character you wish to control. All state transitions and updates are done through the character context. It contains the current state for the character. To create a context you pass in a manager and an initial state. The context will then use that manager to transition between states (when a transition is requested)

```
m_characterContext = new hkpCharacterContext(manager, HK_CHARACTER_ON_GROUND);
        manager->removeReference();
```

Once you have successfully created a character context you can fill out an input structure and pass it to the context's update method. This will cause the current state to update, potentially causing transitions in the state machine. The output from the update is a new desired output velocity for the character. The user chooses exactly how each of the fields in the input structure should be filled out. This decoupling allows the state machine to be run in isolation. It's output can be used with either the character proxy we describe below or with a normal rigid body.

If you run the character controller demo, the current state appears in the left side of the screen.



The same state machine is used for both types of character controllers. The most of the code is identical, but for example the state logic transition or the output velocity filtration can differ for the particular controller. To set the concrete character type use `hkpCharacterContex::setCharacterType()` helper method. The default is the character proxy (`hkpCharacterContext::HK_CHARACTER_PROXY`).

```
m_characterContext->setCharacterType(hkpCharacterContext::HK_CHARACTER_RIGIDBODY);
```

3.6.4 The Character Controller Behavior

The character in the demos is set up to behave like a typical first person shooter character controller. It is worth taking a moment to understand what affects this behavior, as it will help you if you wish to customize or modify the behavior for your own character. The best demos to try are the CharacterControllerDemos, which shows the character controllers in a static landscape which has been specially set up to create situations that are typically difficult for character controllers to deal with: slopes of varying angles, gaps which the character can just fit through / just not fit through, deep valleys, undulating terrain, etc.

3.6.4.1 Default Dynamic Behaviour

We define the speeds at which the character moves in the default state machine. The default walk speed is 10 metres per second. This is about 20 mph, which is 5 times faster than a character walks in reality, however it is a standard FPS walk speed. This speed is used in the "on ground", "in air" and "climbing" states. The jump height for the character by default is 1.5 meters. Gravity is passed into the character state machine, and in the demo it is set to -16 m/s/s. This is because a real world gravity of -9.8 m/s/s actually looks quite low, given the character's faster than life walk speed, however this higher value does mean that the character will fall faster than physical objects.

Because the state machine output velocity is used to control objects, which are part of dynamic system, it is a good practice to filter this velocity to ensure smooth and stable control signal. The filtration is done in two levels. The first level is performed inside the state machine in particular states, namely the "on ground" and "in air" states. The maximal acceleration is limited to 200 m/s/s for on ground state and to 50 m/s/s for in air state. Smoothness and delay of the signal is controlled by pre-multiply gains (0.95 for on ground state and 0.05 for in air state). The second level of filtration is performed on output velocity at the end of the update method. The limits are 625 m/s/s for maximal acceleration and 20 m/s for maximal linear velocity. Limitation of maximal velocity is applied first. We recommend that you decrease these limits with hkpCharacterContext::setFilterParameters() for highly populated scenes with a lot of interactions with abnormal objects (light bodies, hight inertia, etc.) or for complicated asymmetric shapes. If you prefer to have 'stronger' or 'more dynamic' character, don't forget increase maximal acceleration limits. If you want to model a character sitting on fast motion platform, don't forget to change maximal velocity limits.

When the character is running up a slope, (in the on ground state) we try to maintain its forward velocity as if it were on level ground, i.e. it still runs at 10 m/s up the slope. We also wish to make sure that the character runs in the direction it is pointing; so it does not get affected by the slope. This actually means we compute a velocity in the frame of the slope, and then re-project that velocity to the horizontal plane, which creates a new desired velocity that points slightly into the slope, and is slightly larger than 10 m/s. Intuitively this means that to maintain a desired velocity running up a slope at a glancing angle, we actually try to run "into" the slope at a faster speed. The code to do this can be seen in the "on ground" state machine. Note that the on ground state takes the surface velocity into account; it tries to maintain 10 m/s relative to the surface on which the character is standing.

There is a further inconsistency between the default on ground state and the default jump state. When the character runs up a slope of 60 degrees at 10 m/s, it has an upwards velocity of 8.66 m/s. With a gravity of -16 m/s/s, this is higher than the upwards velocity needed to jump to a height of 1.5 meters (6.29 m/s). The default jump state takes the current upwards velocity into account to give a consistent jumping behavior. This means that when the character runs up a steep enough slope, jumping does nothing (as the character has a velocity of greater than 6.29 m/s upwards, relative to the ground velocity). This code can be seen in the hkpCharacterStateJumping::update implementation. Because the character can run so fast, it can launch itself quite high into the air (in our default case, higher than it

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

can jump) if it runs off the top of a steep ramp. To prevent this from happening, the on ground state has a flag m_killVelocityOnLaunch, which is set to true by default. When set, the character sets the upwards velocity to zero when the it leaves the ground (unless it is jumping). This also has the effect of keeping the character closer to the ground when it is moving on uneven terrain and frequently going briefly to the in air state. Note that this flag only works correctly when the character is on a static landscape.

When the character is in air, by default it behaves as though it is on the ground, in that the character can still move at 10 m/s in any direction specified. This is also standard FPS behavior. There is no sliding state in the state machine currently. This could be implemented to give better behavior on slopes that are too steep for the character to climb. Currently the character stays in the in air state on these slopes, and so the player can try to move the character in the direction of the slope, and so slow down or stop the character falling down the slope. To implement custom sliding behavior a sliding state could be added.

3.6.4.2 The Check Support Method

Both characters have a function called `checkSupport()`. This function is used in all the character demos, as an input to the state machine, to determine whether the character is standing on the ground, and if so, what the resultant surface normal and surface velocity is. The implementations of these methods is very similar for both controllers, i.e. both methods use the simplex solver. Creating the input to the simplex solver is done in different ways depending on the character representation, otherwise the code is identical.

The check support function returns one of three states: SUPPORTED, UNSUPPORTED, or SLIDING. The SLIDING state is reported if the character is standing on a surface which has a slope greater than max slope. The SUPPORTED state is reported if the character is standing on a surface which has a slope less than the max slope. The check support function will return SUPPORTED even if the surface is moving away from the character. It is effectively a static query of the geometry around the character in a given direction. For example when the character jumps, at the moment of jump the character will have a velocity moving away from the surface. However as it is still touching the surface, check support will return SUPPORTED.

Sometimes it is necessary to register collisions with trigger objects in a scene, but not have those collisions affect their character controller. In addition to the normal checkSupport( ) method, the checkSupport-WithCollector( ) method is provided. This method takes the same arguments as checkSupport( ), along with a third argument for a point collector. Point collectors are essential for rejecting unwanted collisions and modifying character states. The following is an example of a custom point collector that ignores collisions with "jump boxes" so characters can pass through them while jumping:

```
class MyJumpCollector: public hkAllCdPointCollector
{
public:
    HK_DECLARE_NONVIRTUAL_CLASS_ALLOCATOR( HK_MEMORY_CLASS_DEMO, MyJumpCollector );

    // Constructor
    MyJumpCollector()
        : hkAllCdPointCollector()
    {
        m_jump = false;
    }

    // Accessor Method
    hkBool shouldJump() { return m_jump; }

    // Mutator Method
    void setShouldJump( hkBool jump ) { m_jump = jump; }


    virtual void addCdPoint( const hkCdPoint& event )
    {
        const hkCollidable* rootCollidable = event.m_cdBodyB.getRootCollidable();
        hkWorldObject* wo = hkGetWorldObject( rootCollidable );

        if (wo->hasProperty(HK_OBJECT_IS_JUMP_TRIGGER))
            m_jump = true;
        else
            hkAllCdPointCollector::addCdPoint( event );
    }

protected:
    hkBool m_jump;
}
```

3.6.4.3 The Controller Shape and Stair Climbing

The character shape is usually a capsule shape (hkCapsule) which is 2 metres high. It has a radius of 60 cm. This radius actually affects two important things: how wide the character is (in our case, 1.2 metres) and how the character climbs stairs. Current stair climbing is not explicitly handled by the demos or the state machine. In the examples provided the character climbs steps according to the geometry of the character. Some approaches to handle stair climbing include:

- Using a capsule allow the rounded bottom of the capsule to force the character upward
- Use a traditional approach - query the environment for a surface which the character cannot climb (you can use `hkpCharacterProxy::checkSupport`() for an indication of what's in front of the character) and then use a linear cast to move up to a maximum step height, retry the blocked move and then linear cast downward to contact the ground.
- Use your own hkpConvexVerticesShape to represent your own character controller with a sloped footprint to allow for smooth transitions over surfaces of a fixed height if you find a capsule does not give you desired behavior. Note however that a capsule will be faster on triangular meshes, as Havok has special a capsule triangle collision detection.

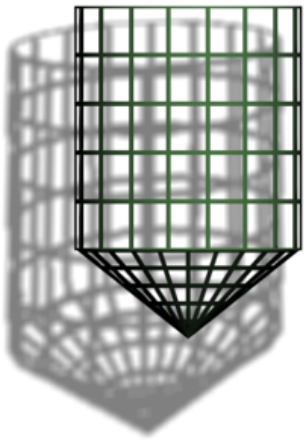
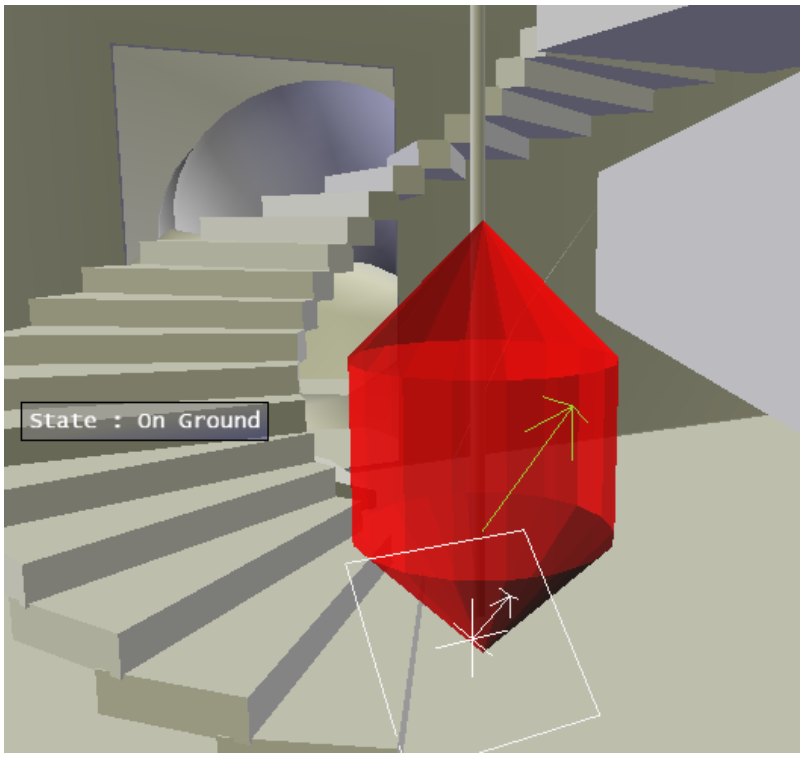
	
The special hkpConvexVerticesShape shape	The character on stairs (on ground state)

Table 3.6: The special hkpConvexVerticesShape character controller shape

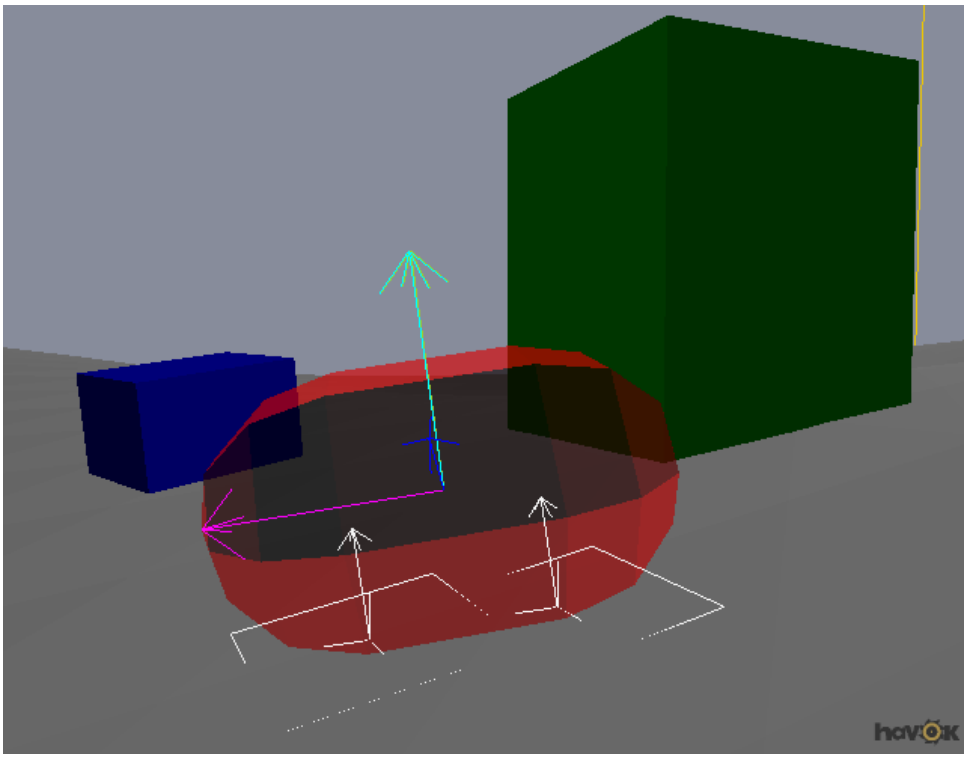
3.6.4.4 The Rotating Asymmetric Shape

Currently the character moves by sweeping the current shape along its current velocity and probes the environment for new surfaces. Obviously this linear sweep does not take rotational effects into account. If you want to use a shape that is not rotationally invariant about the up axis then you have to calculate desired rotation according to character current orientation and surrounding environment. Each character type has its own rotation shape calculation strategy.

- The Character Rigid Body - Because the character rigid body is a part of the dynamic world, you can safely set only its angular velocity. The angular velocity can be calculated as an angle difference between desired and actual orientation of character rigid body divided by timestep. The ideal approach is to apply a simple forward controller with pre-multiplication gain and with limitation of maximal angular acceleration to ensure smooth rotation.

- The Character Proxy - You should rotate the shape manually outside the proxy integration step. You may choose to check for penetration as crouching does in the section below, or you may allow the controller to recover from penetration as it normally does.

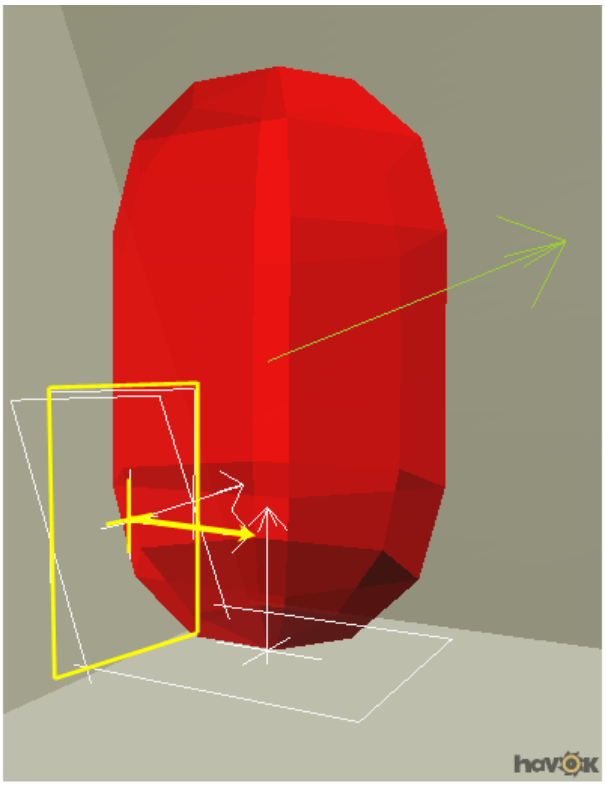
The corresponding asymmetric character demos show how this might be done for particular controller type.



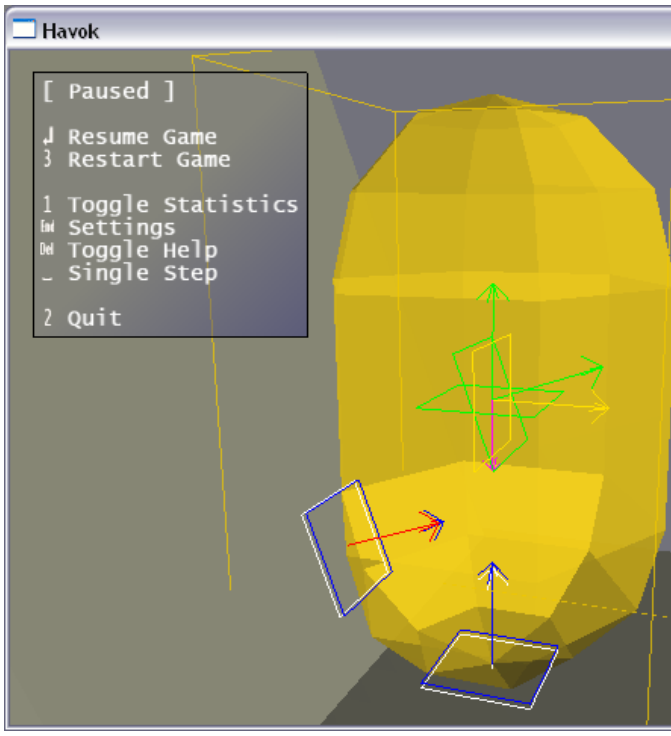
3.6.4.5 Restricted Movement

Both character controllers have built-in support for restricting the movement of the character on steep slopes. The character controller construction info structure has a max slope value which defines how steep a slope it can climb. If this is set to PI / 2 (i.e. vertical) so the feature is disabled (i.e. the character can climb any slope). Implementation of this feature significantly differs for controllers.

- The Character Rigid Body - Internally all normals of contact points are checked and if any of them exceed maximal slope an additional contact point with normal perpendicular to the rigid body up direction is added to the contact manager which prevents movement.
- The Character Proxy - Internally all surface planes are checked and if any of them exceed maximal slope an additional plane is added to the manifold which prevents movement. The additional plane is placed at the same position as the original and oriented vertically, preventing movement up the slope. In the image below you can see that one of the original green planes in the manifold has been classified as too steep to climb and an additional orange constraint has been dynamically added to the manifold.



The Character Rigid Body -Contact point added to the same position as original, but with new contact normal (yellow)



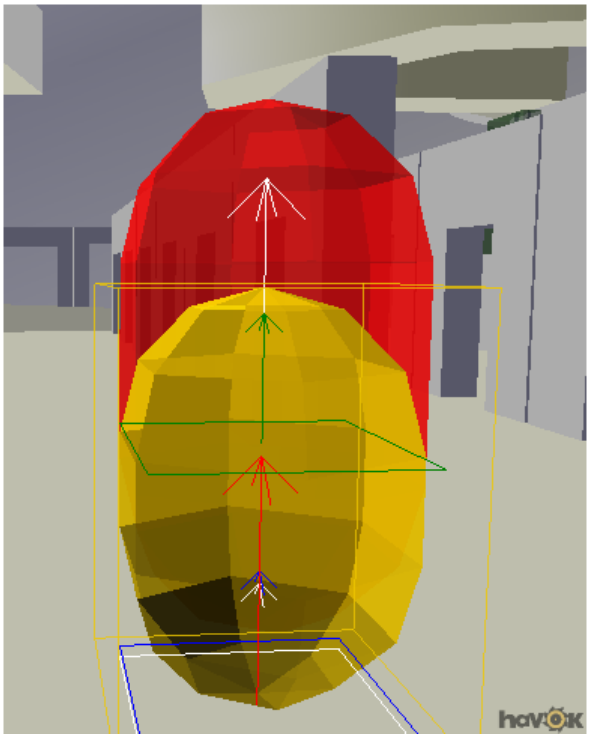
The Character Proxy - Orange constraint plane added

Table 3.7: Maximal slope implementations

3.6.4.6 Crouching

The character controller implements crouching by swapping the shape used to represent the character at runtime from a standing version to a crouched version. In examples we use two capsules to represent these states. The shape swapping is simple in the case of the rigid body character controller, we simply call `hkpRigidBody::setShape(newShape)` which will safely perform the update. Ths situation is more complicated for the proxy controller. See details in The Character Proxy chapter below.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



3.6.5 The Character Rigid Body

The character rigid body hkpCharacterRigidbody is used to represent a rigid body that can move dynamically around the scene. It is a rigid body analogy to the `hkpCharacterProxy` controller, where instead of using a phantom, and actual rigid body is used. This class creates a hkpRigidBody which you add to the world. There are several differences between the rigid body created by this class and a normal rigid body used for simuation:

- The character rigid body has an infinite inertia tensor (i.e. it cannot rotate)
- Gravity is not applied to the character rigid body (it uses a special motion called MOTION_CHARACTER which internally prevents gravity being applied).
- It uses a custom collidable quality type, HK_COLLIDABLE_QUALITY_CHARACTER, which is the same as HK_COLLIDABLE_QUALITY_CRITICAL except that it disables some internal optimizations which produce artifacts when used with rigid bodies that cannot rotate.
- It adds a hkCharacterRigidBodyCollisionListener to the rigid body to achieve some other custom behavior. See below for details.

It's movement is controlled by directly setting its linear and angular velocities. All source code for the character rigid body is in `\Physics\Utilities\CharacterControl\CharacterRigidBody.`

3.6.5.1 Constructing the Character Rigid Body

To create a character rigid body you must fill out a `hkpCharacterRigidBodyCinfo` structure. The structure variables can be divided into the following three groups:

- Standard parameters used to setup a rigid body such as mass, shape, collision info, etc.
- Character controller specific values for example up direction, maxSlope, maxForce.
- Helper parameters used by checkSupport for example maxRecoverySpeed, m_maxSpeedForSimplexSolver.

3.6.5.2 Interaction Listener

A listener `hkCharacterRigidBodyCollisionListener` is used to change a mass of rigid body character during collision with other dynamic rigid bodies and to restrain movement on very steep planes.

NOTE: Currently this listener will not run on the SPU, so for the PLAYSTATION®3 these two features are disabled unless the character rigid body is flagged as must run on PPU.

The listener is inherited from standard rigid body `hkpCollisionListener`. Instances of this listener is registered with a `hkpCharacterRigidBody`.

- The first usage is the modification of a mass of the character rigid body during collision with dynamic objects in the scene. This approach allows continuous regulation of "maximal force (power)" of character during interaction with other dynamics objects. This is done by modifying the apparent mass of the character rigid body during response calculation. The modification factor is calculated in every step in `contactProcessCallback()` using the following algorithm:

  - Calculate angle alpha between contact normal and current acceleration - Calculate current force applied to interaction object as *F = mass*acceleration*cos(alpha)* - If *(F > Fmax)* calculate mass modification factor - Apply mass modification factor to impulse calculation using `hkpResponseModifier::setInvMassScalingForContact()` The acceleration is calculated as difference between the required output velocity from state machine and current velocity of rigid body divided by timestep.

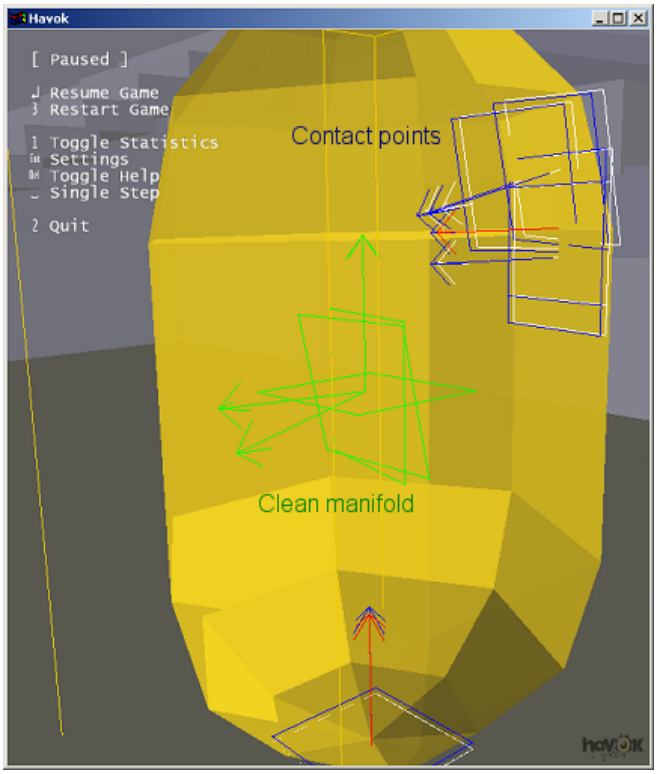
- The second usage is the addition of extra contact points to avoid movement of character on "steep" plane. "Steep" plane is defined by m_maxSlope parameter. It's a angle between UP direction and normal of the plane. So vertical plane has PI/2 (90 Deg) and horizontal plane 0. Default value is PI/3 (60 Deg). See restricted movement above for further details.

  The following is an example of a custom character proxy listener that restricts movement up steep planes:

```
class ZeroPlanesCharacterInteractionListener: public hkCharacterProxyListener
{
public:

    // Constructor
    ZeroPlanesCharacterInteractionListener( const hkVector4& up )
    {
        m_up = up;
        m_counter = 0;
    }

    // In this callback we examine the constraints passed to the simplex.
    // WARNING: This only works when the character is not in a moving
    // environment, as the velocities are zeroed.
    void processConstraintsCallback( const hkArray<hkRootCdPoint>& manifold, hkSimplexSolverInput& input )
    {
        // The remaining planes are vertical planes which have been added
        // in to prevent character movement up slopes which are too steep.
        // Unfortunately we do not have a corresponding manifold point, so
        // we just set all of these velocities to the character velocity.
        for (int i = 0; i < input.m_numConstraints; i++)
        {
            hkSurfaceConstraintInfo& surface = input.m_constraints[i];
            hkReal numerator = surface.m_plane.dot3( m_up );
            hkReal denominator = hkMath::sqrt( m_up.lengthSquared3() * surface.m_plane.lengthSquared3() );
            const hkReal dot = numerator / denominator;
            const hkReal minDot = hkMath::cost( HK_REAL_PI / 3.0f );

            if (dot <= minDot)
            {
                hkVector4 temp;
                temp.setCross( m_up, surface.m_plane );
                surface.m_plane.setCross( temp, m_up );
                surface.m_plane.normalize3();
            }
        }
    }

protected:
    hkVector4 m_up;
    int m_counter;
}
```

3.6.6 The Character Proxy

The character proxy represents the surface boundary for the character. The proxy encapsulates a shape with some spatial and temporal information and builds up a representation of the surrounding environment. For the user you should consider this simply as a shaped particle. It has a surface, a position and a velocity. To interact with it you simply set one or more of the attributes and call `integrate` explicitly. The character proxy will take care of the rest. It will maintain a clean manifold of plane equations that represents the surface contacts it currently has. These plane equations are fed to the next layer which produces a new position and velocity for the character. All source code for the character proxy is held in `\Physics\Utilities\CharacterControl\CharacterProxy\`



If you are using the debug versions of the Havok libraries then the Visual Debugger will display the contact planes shown in the image above. You can see how the character proxy uses linear casting to build up a manifold of planes for the simplex solver . The original hit position of each point in the start point collector is shown as a white plane. We also display this same plane displaced by the distance returned by the hit, in blue if the distance is positive or red if the distance is negative. Planes returned by the cast collector are shown in magenta. The character proxy uses these planes to verify and update its manifold. This maintenance converts the relatively noisy signal returned from the linear caster into a smoother cleaner signal that is passed to the simplex solver. The clean planes passed to the simplex solver are shown in the middle of the character in green. When looking at this debug output you should verify that the green planes will appropriately restrict the characters movement.

3.6.6.1 Maintaining the Manifold

When filtering the signal from the linear caster the character proxy uses both the start collector and cast collector results. The start collector contains all the contacts that surround the character in its current position at the start of the movement step or cast. The cast collector holds the distance to the first point of contact for each surface along the character's current path. These results are checked and a manifold of valid planes is constructed and maintained according to the following rules.

- All planes currently in the manifold must verify that they are still valid. This is done by checking that they are present in the start point collector.
- The closest (or most penetrating) plane picked up by the start point collector is added to the manifold, if it is not there already. This will ensure that an object which penetrates the character (say by moving in from the side) will be picked up by the manifold.

- All penetrating planes have their distance set to zero when passed to the simplex solver, but are artificially given velocity so the character is pushed away from them and out of the penetrating state.
- The first cast collector plane is added. This represents a surface immediately blocking the character and must be taken into consideration.

These rules produce a clean and minimal manifold that can be passed to the simplex solver. The simplex solver ultimately uses these to restrict the movement of the character. These rules are also set out specifically to ensure that the character does not hit triangle edges as it moves over a tessellated landscape.

3.6.6.2 Constructing the Character Phantom

To create a character phantom you must fill out a `hkpCharacterProxyCinfo` structure. When filling out this structure you will be asked to specify a `hkpShapePhantom`. The shape phantom is used by the proxy to probe the environment for surface details. When constructing this shape phantom you specify the shape you wish to use for your character.

3.6.6.3 Crouching

When switching between the shape we first ensure that the transition will not cause penetration. This prevents the character from standing in areas where the ceiling is too low. The code to perform this safe swapping is shown here

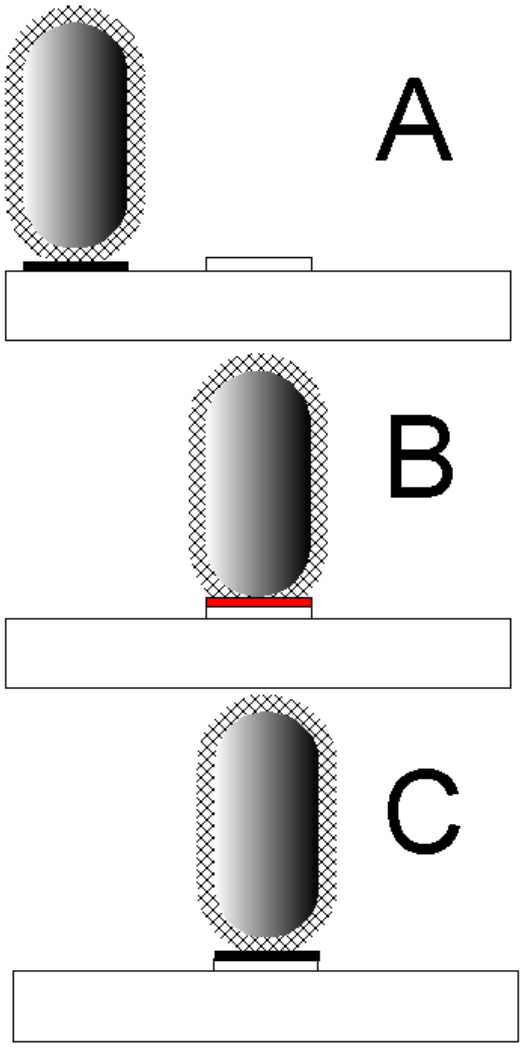
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
void CharacterDemo::swapPhantomShape( hkpShape* newShape )
{
    // Remember the current shape
    hkpShape* currentShape = const_cast<hkpShape*>(m_phantom->getCollidable()->getShape());

    // Swap to the new shape.
    // N.B. To be safe, we always remove the phantom from the world first, then change the shape,
    // then re-add, in order to refresh the cached agents in any hkCachingShapePhantoms which
    // may also be present in the world.
    // This also forces the display to be rebuilt, which is necessary for us to see the new shape!
    {
        // Note we do not have to add a reference before removing becasue we hold a hkpCharacterProxy
        // which has a reference to this phantom - hence removal from the world cannot cause this phantom
          to
        // be accidentally deleted.
        m_world->removePhantom( m_phantom );
        m_phantom->setShape(newShape);
        m_world->addPhantom( m_phantom );
    }

    //
    // We use getClosestPoints to check for penetration
    //
    hkpClosestCdPointCollector collector;
    m_phantom->getClosestPoints( collector );

    // Allow a very slight tolerance (approx 1cm)

    if (collector.hasHit() && collector.getHit().m_contact.getDistance() < .01f)
    {
        // Switch the phantom back to our current shape.
        // N.B. To be safe, we always remove the phantom from the world first, then change the shape,
        // then re-add, in order to refresh the cached agents in any hkCachingShapePhantoms which
        // may also be present in the world.
        // This also forces the display to be rebuilt, which is necessary for us to see the new shape!
        {
            // Note we do not have to add a reference before removing becasue we hold a hkpCharacterProxy
            // which has a reference to this phantom - hence removal from the world cannot cause this
              phantom to
            // be accidentally deleted.
            m_world->removePhantom( m_phantom );
            m_phantom->setShape( currentShape );
            m_world->addPhantom( m_phantom );
        }
    }
}
```

3.6.6.4 Keep Distance

The character proxy has an internal parameter called keepDistance which provides a soft penalty shell around the character shape allowing it to smooth the signal coming from the collision detector.

The controller gathers plane equations using both a cast collector and the start point collector. The start point collector is used to verify existing contacts in the manifold. It represent the environment around the character in its current position. When making this query the tolerance used is the keepDistance. The cast collector is used to check for objects the character is about to run into. These contacts are translated into plane equations and passed to the simplex solver. The keepDistance is also used to expand the planes returned by the collectors. The following image shows what happens when a character with a large keepDistance travels onto a small bump.



The diagram shows the shaded capsule which represents the characters shape and a hatched area showing the keepDistance. At point A the character is stationary. The plane is black was initially added when the character hit the ground. It was added by a cast collector and verified on successive frames because it remained present in the start point collector. Note that it has been expanded by the keepDistance so the character floats above the ground. At point B the character has cast along the ground. The cast collector does not use the keepDistance as a tolerance so the cast missed the small step. The step does appear in the start point collector and since it is inside the keepDistance it is marked as penetrating. Penetrating planes are handled by adding velocity to them which slowly pushes the character away - their distance are zeroed before they are passed to the simplex solver so they do not produce a sharp discontinuous movement. If the character stops on the step (shown at C) it will rise up slowly. If the character passed quickly over the step then the plane verified by the start point collector will drop back to the original height again and the penalty recovery will cease.

In summary : The shape plus radius defines a hard shell around the character which the cast collector and simplex solver enforce. The keepDistance provide an additional soft shell which is enforced using a simple penalty recover method. This soft penalty recovery filters high frequency discontinuities from the collision detector in the range of the soft shell.

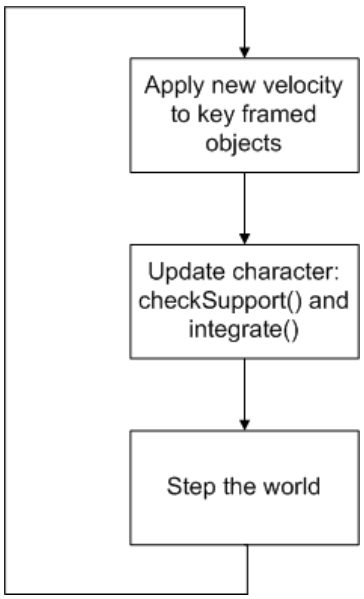
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.6.6.5 Moving Platforms and Low Frequency Characters

While Havok is a continuous physics engine, the character controller is effectively discrete in its simulation approach. Although this is fast, and means that characters can be stepped independently of the physics engine, and at lower frequencies, this can also lead to some accuracy problems. The first of these is that of the moving platform. When a character is in the air above a platform that is rapidly moving upwards, it may not see the moving platform until it is too late. When hkpWorld::step() is called, the platform will be integrated upwards some amount, depending on its velocity. At a certain point the aabb of the platform and the character controller will overlap. When this happens, at the next integration call for the character controller, the character will see the moving platform and if it is close enough, add an appropriate plane into its simplex solver. However it is possible that at this stage, the platform will already be penetrating the character proxy. The character proxy will be able to recover gracefully from this (see the section on maintaining the manifold, above) however you may see some frames where the character and the platform are visibly penetrating.

The situation is also apparent when a character is on a moving landscape. During the character controller integrate() call, all calls to the collision detection are with the landscape in the same position (i.e. the landscape does not move as time advances in the character controller). Because of this, additional artifacts such as occasional hitting of triangle edges may be apparent in fast moving landscapes; where possible for best simulation of the character controller, it is advisable to keep landscapes fixed.

Note that if you are changing the velocity of a platform with which the character is interacting, it is best to set the velocity of the platform (for example using the keyframed utility) prior to calling the character controller integrate call. The best order of calls is as follows:



If you apply velocities to your keyframed objects before calling the character controller, then the character controller will react correctly to the keyframed object. Otherwise, if for example you have a character on an accelerating elevator, the character will always see the elevator's velocity from the last step, and will sink slightly into the elevator as it accelerates (although penetration recovery should prevent it sinking too far).

As all character controllers are stepped independently, it is possible to step them at low frequency. For example you could step the physics simulation at 60 Hz, but step a character controller at 20Hz. Obviously

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

this will give you a performance gain (if you make sure that you step different characters out of sync, and so spread the CPU load between frames), however it makes the situation described above worse. The quality of interactions between the character and dynamic objects, keyframed objects and other characters will be degraded at lower frequencies. You are more likely to see penetrations such as those described above. In general, if you can afford it, it is best to step the character controller at the same frequency as your simulation. If you have many AI characters, it may be worth reducing the frequency for them, however you should experiment to make sure that any resultant penetrations are acceptable.

3.6.6.6 The Character Proxy Manifold

The character proxy has a stored array of hkpRootCdPoint structures, called the manifold, which you can access through the call getManifold(). This is the data that is used to build planes for the simplex solver which are used for both integrate() and checkSupport() calls, and it is the character's view of its immediate world. However, if you are using this data yourself, for example in the state machine, it is worth noting that the data can easily be out of date. In fact, at the end of the integrate() call, the character will have moved from the position at which the manifold was built (to the end of its step) and so the contact points in the manifold will not be exactly correct. The character proxy call refreshManifold() can be used to bring these points up to date. This call re-evaluates all collision detection at the current position of the character controller, and updates the manifold. Note that moving any other character controller, or calling the hkpWorld step functions may also make the manifold out of date. At the start of the checkSupport() call refreshManifold() is called automatically, so after checkSupport() the manifold will be correctly up to date.

3.6.6.7 Characters and Heightfields

Because of the logic used to maintain the character's manifold, you should be careful if you wish to use characters with heightfields (i.e. classes deriving from hkpSampledHeightFieldShape). Heightfield collision detection simply collides the vertices of the moving object against the heightfield - see the heightfield section of the collision detection chapter for more details. In the case where a capsule is used as a character controller's shape, only one point is ever generated against the heightfield, so only one point will ever be in the character's manifold. Unless the heightfield is virtually flat, this is likely to cause problems. If a character is in a narrow valley, the manifold should contain points on either side of the valley, but if the valley is represented by a heightfield, only one point will be generated, and jitter is likely to result. As a workaround for this, you should wrap the heightfield with hkpTriSampledHeightFieldCollection and hkpTriSampledHeightFieldBvTreeShape, which will give triangle accurate collision results.

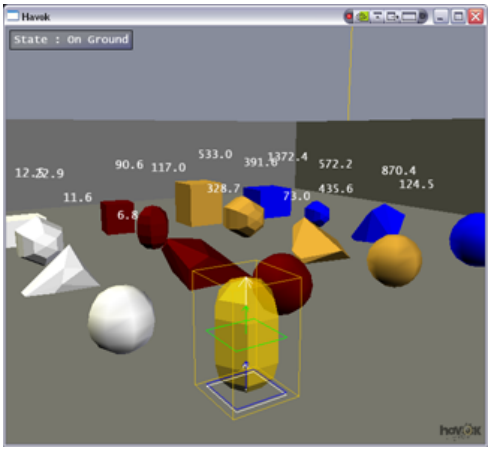
3.6.6.8 Physical Interaction and the Interaction Listener

The character proxy listener provides 2 methods specially designed to control character interaction. The first deals with character object interaction.

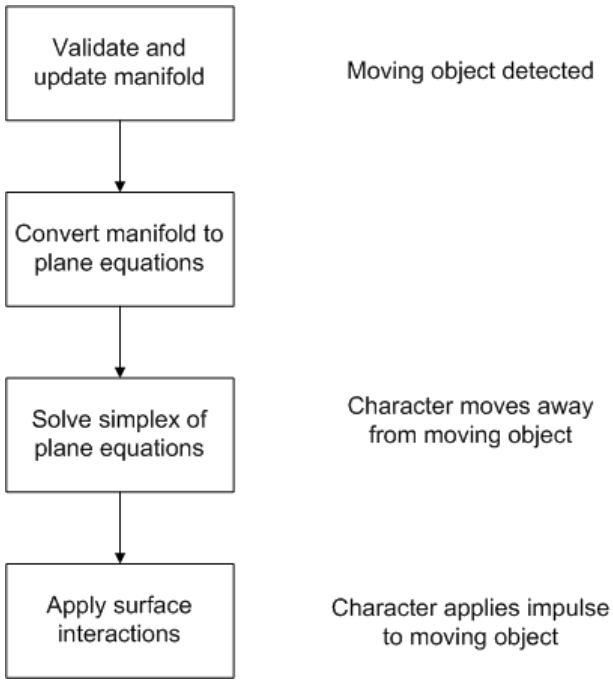
```
// Called when the character interacts with another object
virtual void objectInteractionCallback(hkpCharacterProxy* proxy, const hkpCharacterObjectInteractionEvent&
  input, hkpCharacterObjectInteractionResult& output ) {}
```

When the simplex has been passed a set of plane equations it solves these as described below. For each of the dynamic surfaces (not owned by a fixed or keyframed body) touched while finding a solution it

calls back the listener with an object interaction callback. Before calling back the listener it calculates an appropriate impulse to apply to the object and fills in the details of this calculation in the input structure. This information can be particularly useful when trying to play sounds or estimate damage to apply to the character. The output structure contains the impulse to apply to both the object and the character. When the callback returns, these results are applied immediately. You are free to completely override the suggested values passed in the input structure. The character interaction demo, shown below, demonstrates the default character interaction behaviour. It contains a scene with random objects with varying masses. The characters strength contained in the hkpCharacterProxyCinfo structure determines how easily the character will push the various objects around.



Interaction between the character controller and rigid bodies is not perfect, due to the fact that the character controller interacts by applying impulses to rigid bodies outside the constraint solver. While in general this gives good results, there are some drawbacks to this approach. A simplified version of the character integrate() loop is as follows:



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

It can be seen that the interaction between a character controller and a rigid body is a two phase process. Firstly the character sees and avoids the interaction plane generated from collision with the rigid body. After this the character applies an impulse to the rigid body. With the loop ordered this way, when an object hits the character, it will always be able to move the character slightly before the character gets a chance to apply an impulse to it. This can be unwanted, if you want the character to be infinitely strong. This can be seen in the character interaction demo, where you can fire boxes at the character from the side. It is possible to fix this problem by using the objectInteractionCallback described above to zero the velocities of the planes in the simplex solver. See the demo code from the character interaction demo for an example of this.

It might be tempting to rewrite the above loop, so that the character applies impulses before solving the plane equations, to avoid this problem. However this leads to a worse problem: consider a character pushing a box against a wall. The box will not be able to move, so from frame to frame, we should see a static plane in the character manifold resulting from this. If we were to apply an impulse to the rigid body before solving the simplex, however, the plane would be seen incorrectly to be moving away from the character. This leads to considerable unwanted jitter.

Another problem can arise when a character controller gets squeezed between two rigid bodies. Because the controller applies impulses to each rigid body without reference to the other, if the bodies are sufficiently heavy, the system will not be solved correctly (as would, for example a similar system where the middle body was not a character controller, but a rigid body). In this case the rigid bodies will penetrate the character controller, and may eventually hit each other. While this is not ideal, it is an unavoidable consequence of simulating the character controller as a phantom "outside" the simulation.

The character imparts a downwards force on rigid bodies it is standing on. This is controlled by the m_characterMass value in the character proxy. By default this value is 0, which means this feature is disabled. Setting the value to say 100 means the character will apply a force equivalent to that of a 100 kg rigid body. Note that the gravity parameter passed into the character integrate function is used with the m_characterMass vale to calculate the downward force. It must be set to the gravity of the hkpWorld to get the correct behavior. This value does not affect how fast the character falls when in air - that is controlled outside the integrate call by the state machine.

The second interaction method deals with character-character interaction. When two characters collide the proxy calls the following listener callback

```
// Called when the character interacts with another character
virtual void characterInteractionCallback(hkpCharacterProxy* proxy, hkpCharacterProxy* otherProxy, const
  hkContactPoint& contact) {}
```

This will be called once per character if all characters are running at the same frequency, or once per interaction if characters are running at different frequencies. This method is passed handles to both controllers involved in the collision. You are free to decide exactly how the characters should respond to the interaction by changing either linear velocity, position or even potentially the shape of the characters. The default behavior leaves both characters unaffected.

3.6.7 The Simplex Solver

The simplex solver is employed in both character controllers. It has a very similar usage in either implementations of `checkSupport()` method. However for the character proxy controller the simplex solver is a core of movement calculation as described above. The planes gathered from the character proxy

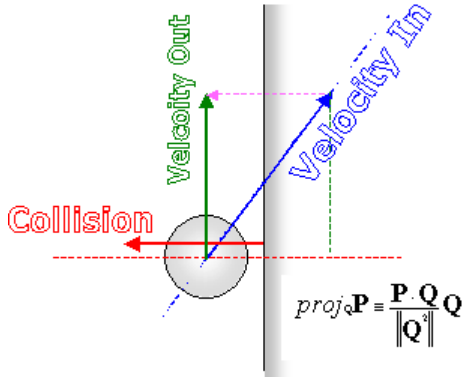
represent a set of simultaneous constraints that the proxy may not violate. Each of these constraints is detailed and includes surface, velocity, friction and priority information.

```
/// This represents a single constraint fed to the simplex solver
struct hkpSurfaceConstraintInfo
{
        /// The plane direction (normally identical to the contact normal). Note: .w component is distance
    hkVector4 m_plane;

        /// The velocity of the plane
    hkVector4 m_velocity;

        /// The static friction (behaves pretty much as expected from real friction)
    hkReal m_staticFriction;

        /// An extra friction in the up direction, if this function is used (value != 0), than
        /// the overall friction will be split into two components:
        ///  - one in the most up direction
        ///  - one which is perpendicular to the up direction
        /// Then each component is solved separately (box friction instead of friction circle)
    hkReal m_extraUpStaticFriction;
    hkReal m_extraDownStaticFriction;

        /// The dynamic friction:
        ///  - if set to 1.0f, then the simplex will simply project the velocity onto each plane
        ///  - if set to 0.0f, then the algorithm will try to maintain the input velocity
    hkReal    m_dynamicFriction;

        // The relative priority of the surface
    int          m_priority;
};
```

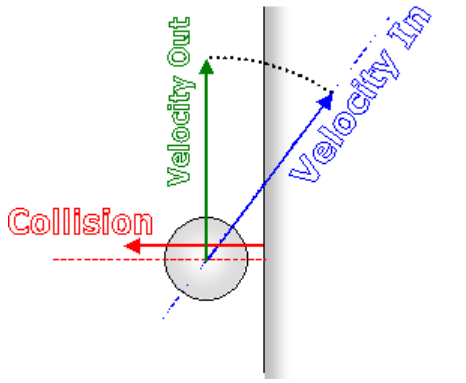
A collection of these constraints together with a velocity, position and time information is sent to the simplex solver. The solver then attempts to maximize the distance the character can move in the given direction over the given time step. It produces a new position and velocity. Many traditional approaches attempt to solve these surface constraints one after the other rather than in parallel. This can lead to unwanted jittering in the final controller. The user has direct control over the input structure passed to the solver. By registering a `hkpCharacterProxyListener` and implementing a `processConstraintsCallback` you will receive a callback containing the complete simplex input structure before it is passed to the solver. You may override any of the variables and dynamically add planes on the fly.

3.6.7.1 Friction

The solver handles both dynamic and static friction. Dynamic friction controls how much velocity or momentum is lost when a character hits a surface. When dynamic friction is set to 1 all velocity in the direction of the surface is clipped. This is the common behavior for first person shooters. The static friction gives you control over when the proxy will stick to a surface. If the ratio between the surface tangent velocity and the velocity projected in the surface normal direction is less than the static friction coefficient then the velocity of the controller in the tangent plane is clipped to the velocity of the surface at the point of contact. Note: If you are simply trying to restrict movement e.g. prevent the character from moving up a slope it is better to add additional planes to the manifold than directly change this value.



When dynamic friction is set to 0, momentum is preserved.



3.6.8 The Character Controller Examples and Use Cases

3.6.8.1 Character Proxy Demos

These demos introduce the basic concepts and behaviour of the character proxy controller. Demos can be found under "Character Proxy" in the demo "Character Control" menu (`\Physics\UseCase\CharacterControl\CharacterProxy\`).

- Asymmetric Character - Shows how to create a character with rotationally asymmetric shape and how to ensure its safe rotations. Use cases are swimming characters, horses, dogs, etc.
- Character and Triggers - Shows an interactions with rigid bodies and phantoms and how to implement phantom trigger objects using the custom collectors.
- Character Controller - Shows the behaviour of character in test static scene with many problematic objects from real game levels such as stairs, ladders, gaps, etc.
- Character Interaction - Shows interaction with objects of different sizes and weights.
- Character Phantom Interaction - Shows another technique how to interact with phantoms and how to implement phantom trigger objects.
- Character Priority - Shows how the character controller uses priority to handle insolvable situations.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- Low Frequency Characters - Shows how the character controller can be used efficiently for multiple characters at heightfield and how to simulate these characters at different frequencies.
- Multiple Characters - The demo used to test the performance of multiple characters.

3.6.8.2 Character Rigid Body Demos

These demos introduce the basic concepts and behaviour of the character rigid body controller. Most of them are analogies to the character proxy demos presented above. Demos can be found under "Character Rigid Body" in the demo "Character Control" menu (`\Physics\UseCase\CharacterControl\CharacterRigidBody\`).

- Asymmetric Character Rb - Shows how to create a character with rotationally asymmetric shape, how to control it by setting angular velocity to ensure safe rotations. Use cases are swimming characters, horses, dogs, etc.
- Triggers and Phantoms Character Rb - Shows an interactions with rigid bodies and phantoms and how to implement phantom custom listeners.
- Character Controller Rb - Shows the behaviour of character in test static scene with many problematic objects from real game levels such as stairs, ladders, gaps, etc.
- Platforms Character Rbs - Shows an interaction of the character rigid body with vertical and horizontal moving keyframed platform.
- Pushed Character Rbs - Shows an interaction of the character with another "free" and "AI" controlled characters.
- Interaction Character Rb - Shows interaction with objects of different sizes and weights.
- Multiple Character Rbs - The demo used to test the performance of multiple characters.

3.6.8.3 Character Proxy Vs Rigid Body Demo

The demo compares the behaviour of the character proxy and the character rigid body in a test static scene with many problematic objects from real game levels such as stairs, ladders, gaps, etc. Demo can be found under "Character Proxy Vs Rigid Body" in the demo "Character Control" menu (`\Physics\UseCase\CharacterControl\CharacterProxyVsRigidBody\`).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.7 Vehicle Physics

3.7.1 The Havok Vehicle Kit

3.7.1.1 Introduction

The *Havok Vehicle Kit* is a subset of the Havok SDK that helps you to develop physically-based vehicles that fully integrate with the Havok engine and the game environment.

The Vehicle Kit includes classes that define and implement the set of behaviors and algorithms associated with car simulation and driving. It also provides a complete game-oriented implementation of a vehicle and its components, such as its wheels and transmission. This default implementation is fully functional and easily tweakable, and a number of game developers have used it "as is" in their games.

Figure 3.63: Examples of vehicle games using Havok

Alternatively, thanks to the object-oriented nature of the Vehicle Kit, you can override any of the given default behaviors and extend the provided functionality to suit your game needs (for example, implementing a custom wheel friction model based on existing tire data).

3.7.1.2 About This Documentation

The Vehicle Kit section is divided into:

- An introduction to the *Vehicle Kit*, which also explains how the use of "unreal" physics can produce better gameplay, describes the output of the Vehicle Kit, and offers tips on customizing it for your own vehicle game.

- A *programming guide* that provides detailed information about the Kit, its classes, and the interaction between them. You can use the programming guide as a reference when using the tutorial or when working with the Kit in your own games.

Although learning how to use the Kit by modifying the various parameters is very straightforward, customizing your own drive model can take time. In addition to reading through the provided documentation and sample code, you should take time to play with the samples and experiment with different parameters.

3.7.1.3 Vehicle Physics

Introduction

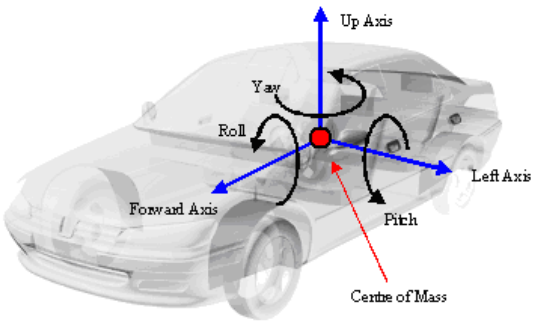
The primary focus of this chapter is to present the basics of the physics used within the Havok Vehicle Kit. It covers physical principles, Vehicle Kit parameters and some good parameter settings for various game scenarios.

Perfect vehicle simulations result in very realistic driving behavior, but in some situations you will want to employ various tricks within the simulation to make the driving experience more fun.

This document focuses on the default implementation provided with the Vehicle Kit. Keep in mind that you can always replace the default implementation of the vehicle or any vehicle components with your own custom implementation if you want to add a feature that doesn't already exist within the Kit. You can find more detailed information about the Vehicle Kit classes in the Vehicle Kit programming guide.

Coordinate System

A vehicles coordinate system defines all positions and directions relative to the vehicle itself. The Kit uses the coordinate system shown in the diagram below, which defines its coordinates using an origin at the vehicles center of mass and three axes:



You need to set the directions of these local forward, right and up axes when constructing a vehicle chassis using the Kit.

Suspension

A vehicles suspension is a mechanical system that attaches one or more wheels to the chassis of the vehicle. In real life there are many different types of suspension, each of which has its advantages and disadvantages. In principle all suspension systems allow the wheel to:

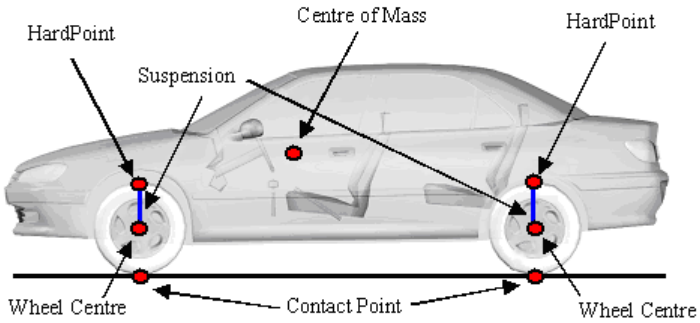
- Rotate around the wheel axis
- Move upwards and downwards within a given range. Normally this movement is performed in a straight line (due to mechanical restrictions in some suspensions, the wheel might move along a curve instead).

A spring, an anti-roll bar and a shock-absorber (known as a dashpot in Havok physics) are used to control the forces that act when the wheel moves up or down. If the spring becomes compressed, it tries to expand itself and raises the car chassis at its hard point (see the diagram below).

Combining a spring and the chassis of the vehicle will result a spring mass system, which tends to oscillate. In order to quickly dampen this oscillation, dashpots are added to create an opposition force to the wheels relative movement. Note that dashpots create different forces depending on the direction of the wheel. If the ground pushes the wheel upward, a much lower force is created compared to when the wheel moves downwards. The ratio is roughly 1:3.

The Havok vehicle Kit uses a slightly simplified suspension model:

- It assumes that the movement of the wheel is limited to the straight line of the suspension direction.
- The suspension strength is independent of vehicle mass.
- The suspension spring connects to the chassis at a point in chassis space called the hard point and extends down through the wheel to the center of the tire (half the wheel width). This is illustrated in the following diagram.



The spring and shock absorber force F depend on:

- The distance d between the wheels center and the hard point.
- The resting length l of the suspension spring.
- The spring constant c of the suspension spring.
- The relative velocity v with which the wheels center approaches the hard point.

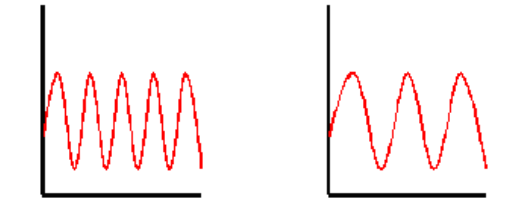
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- Two damping coefficients e, one if v is negative and one if v is positive.

$F = (d\text{-}l) * c + v * e$

The Suspension Spring:

When a spring is compressed, it creates a force. The higher the strength of the spring is within Havok, the stiffer the spring will be. In the Vehicle Kit, a suspension spring and the chassis form a spring mass system. The oscillation frequency of this spring mass system depends on the spring strength and the mass of the car.



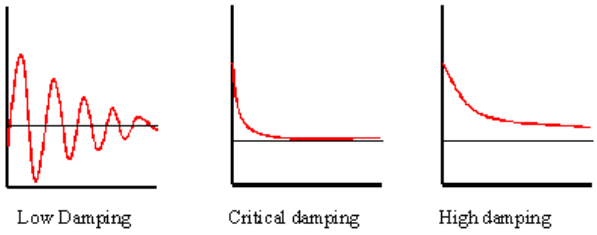
The Havok Vehicle Kit defines the spring constant c as a user-defined constant multiplied by the mass of the vehicle (c = cuser * mvehicle). Hence, as the mass of the vehicle is altered, the spring strength is altered proportionally, keeping the chassis in the same position. This means that the mass of the car can be altered without affecting the suspension length. Good values for the user spring constant cuser range from 5 to 50, depending on the type of vehicle you want to simulate. Some examples are given below:

- Sports car: 50
- Off road cars: 30
- Limousines: 15

Note that if the spring constant is too high, the physical simulation becomes unstable.

The Suspension Dashpot

Without any damping, the suspension spring and chassis mass would oscillate forever. Since damping is necessary, shock absorbers (i.e. dashpots) can be added that damp any movement in the suspension direction. The higher the damping coefficient, the quicker the dynamic energy is removed:



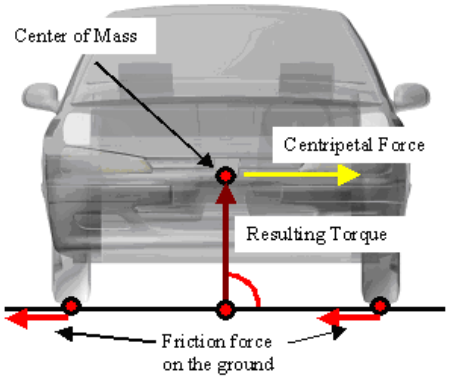
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The "critical damping" value refers to the situation where the dashpot completely removes the suspension movement in less than one full oscillation cycle. To keep the chassis forces low, real cars actually have suspension damping lower than the critical damping.

The critical damping value is dependent on the suspension strength. Therefore you should try varying the suspension strength before determining appropriate damping values for your vehicles. Good values for damping range from *10%* to *50%* of the suspension strength. If the damping values are very high, the physical simulation becomes unstable.

Friction

When taking a corner, a car tends to maintain its current direction of travel. However, the friction between the tires and the road forces the car to follow the curve:



As the friction force is applied at the tires and not at the center of mass, the resulting torque rolls the car. There is a limit to the friction force *maxF*, which depends on:

- The type of road and tire resulting in a friction function *f*.
- The relative velocity *v* between the tire surface and the ground. This is zero if the car is not sliding.
- The angular velocity of the wheel *a*.
- The force *F* acting on the tire, pressing it onto the surface.

Hence, we get $maxF = f(v, a) * F$

For simple vehicle simulation, the vehicle Kit assumes a constant function for *f*

$f(v, a) = constant$ *(Friction Coefficient)*

As a result, if the mass of the car changes, the resulting friction behavior of the car does not change. In real life, friction values around 0.8 are realistic.

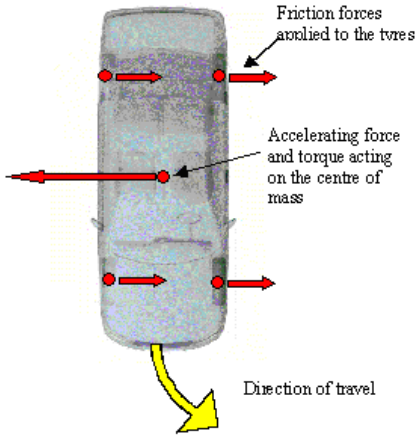
The Relationship Between Front and Back Friction

If you double the friction values for all tires, the car can take corners faster without sliding, however the overall driving behavior will still be quite similar provided the car does not flip.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

However if you increase the front tire friction coefficient by only 20 percent without changing the friction values for the other tires, the difference is dramatic. Lets look at an example.

Imagine a car taking a left turn:



As shown in the above diagram, he friction forces applied to the tires accelerate the car to the right. The steering angle of the front tires defines the turning radius of the car. If this turning radius results in friction forces exceeding the maximum allowed friction forces, the car loses its grip on the road and slides.

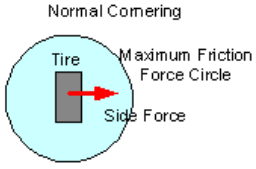
If the front tires lose grip first, the car "understeers", sliding on its front tires. It will not follow the curve defined by the steering angle. This usually means that the front friction coefficient is less than that of the back tires.

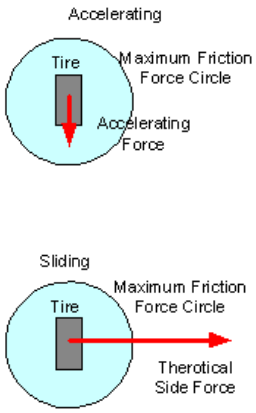
If the back tires slide first, the car "oversteers", sliding on its back tires. This means that the car may spin uncontrollably in curves.

If all four tires slide at the same time, the car tends to keep its current direction while sliding, making the car more difficult to control but fun to drive.

As you can see, the relationship between the front and back friction coefficients is mainly responsible for these effects. In a physical simulation, the brake and acceleration balance also influences this behavior. You will find out more about this later on in this manual.

The Friction Circle

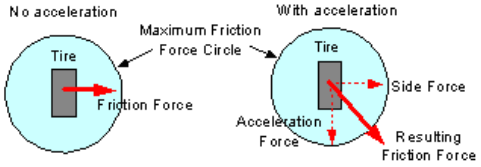




In reality, a tires friction circle is not 100% circular, but slightly deformed. However, in most cases it is sufficient to assume a perfect circle.

What does this circle mean?

Imagine a car taking a sharp turn without braking or accelerating. Assume the car is not sliding yet. If the driver accelerates, an extra force component adds to the current force, such that the resulting force exceeds the maximum friction force. The car starts to slide:



As a result, any forward or backward torque on the wheel effectively reduces the maximum side force, making it easier for the wheel to slide. A good example of this is a tire burnout or braking with blocked wheels.

Knowing this principle, the distribution of the brake torque and acceleration torque to the individual wheels is very important. This is also known as the brake or acceleration balance.

If you create a car with a very strong engine and use normal friction values for the wheels, you will experience:

- A normal acceleration
- A very slippery car

The reason for this is that the wheels can simply continue spinning without gaining grip.

The Brake and Acceleration Torque Distribution

The brake and acceleration torque distribution describes how the cars brake and acceleration torque is distributed to the 4 wheels, and has a significant effect on driving behavior.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

As you apply more forward torque to a wheel, its maximum allowed side force decreases. In other words, applying torque to a tire effectively reduces the friction coefficient. The wheel thus becomes more prone to slip.

In practice, this means that rear-wheel drive cars tend to oversteer in curves, whereas front wheel driven cars tend to understeer.

The Influence of the Center of Mass and the Rotation Inertia

In vehicle dynamics, the mass of the chassis plays a minor role. The mass tends to have more of a scaling effect than an actual influence on the entire vehicles behavior. However, the position of the center of mass inertia tensor has big effects.

The inertia tensor is a 3x3 matrix defining how difficult it is to rotate a body around the principal axis. The inertia tensor is dependent on the mass itself and on the mass distribution. For example, it is relatively easy to spin a one-kilogram iron ball compared to a one-kilogram bicycle wheel.

The inertia is roughly equal to the mass of the object multiplied by the average distance of all points in the objects volume to the center of mass. Hence, the inertia of a car with dimensions 4m X 2m X 4m for the principle axis is:

- Up axis: about 2 meters * 1500 kg
- Forward axis: about 1 meters * 1500 kg
- Right axis: about 2 meters * 1500 kg

A higher center of mass means that the car rolls more heavily in curves. It also means an increase in a braking cars rotation around the pitch axis.

Higher rotation inertia makes it more difficult to change the angular velocity of the car. It also reduces the frequency of rotational oscillations about the center of mass. A car oscillating around the roll axis at 2 Hertz only oscillates at a rate of 1 Hertz if its inertia is doubled. Hence, the inertia can be used to reduce the likelihood of a car spinning out of control when colliding or sliding.

Steering

Steering translates the direction of the steering wheel into a rotation about the yaw axis. In real life there is a simple linear relationship between the steering wheel of the car and the steering angle of the tires. However if you tried to steer a real formula-1 car at 200 mph with a game pad, you would not stay on the road for long!

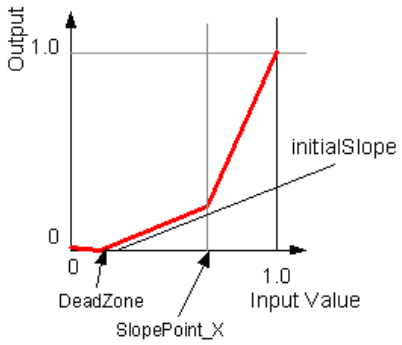
Hence, in a game environment you have to preprocess the raw input data of the input components using several techniques. The Havok Vehicle Kit does this preprocessing in two steps: first, a *DriverInputComponent* is used, followed by a *SteeringComponent* You can find more detailed information about these components and their default implementations in the Vehicle Kit programming guide.

Dual Slope (DriverInputComponent)

A dual slope input controller uses a very simple relationship between input value and output. For input values less than the controllers "dead zone", it returns zero. For input values x greater than the dead zone *dz* and less than *SlopePointX*, it returns *(x dz) * initialSlope*

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

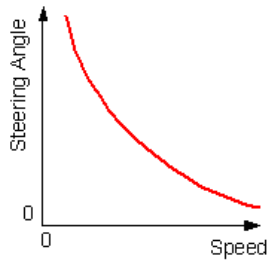
For input values greater than *SlopePointX*, the value is interpolated in a way such that it reaches a value of 1 for an input value of 1.



Speed-dependent Steering Angle (SteeringComponent)

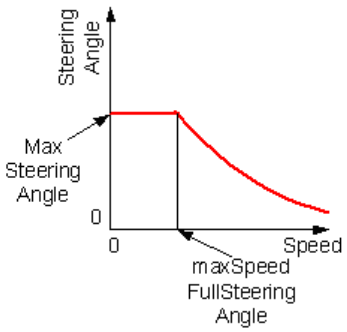
As the accuracy of a joystick is much smaller than that of a real steering wheel, the vehicle Kit can change the accuracy of the input depending on the current velocity.

The basic idea is that the higher the speed of the car, the less sensitive the steering should be. Given a certain friction coefficient and vehicle speed, a theoretical steering angle can be determined where the car is on the verge of sliding:



The steering angle as a function of vehicle speed, where the vehicle is on the verge of sliding.

For practical reasons the maximum steering angle is clipped:

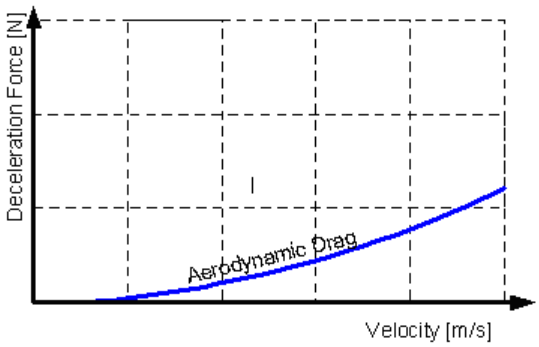


Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

To tune the car, decide a maximum steering angle for slow speeds and then play with the *maxSpeedFull-SteeringAngle* parameter to get the desired high-speed behavior.

Aerodynamics

As air has a certain density, it influences a moving car. These influences are called aerodynamic drag and lift. Aerodynamic drag creates a force opposite to the cars direction of travel. Aerodynamic lift creates an upward (or when negative a downward) force. Drag is the main effect that limits the top speed of the car.



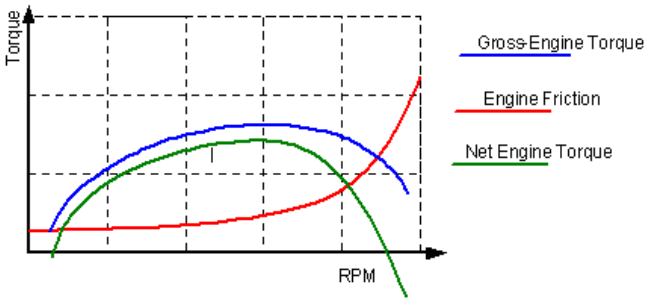
The formula used in the default vehicle implementation is:

F_{drag} = * airDensity * dragCoefficient * frontalArea * velocity^2

F_{lift} = * airDensity * liftCoefficient * frontalArea * forvard_velocity^2

Engine

The engine generates all the energy, or torque, that drives the car. The most common engines currently used for vehicles are combustion engines. The basic principle of this type of engine is that it burns petrol in a cylinder, thereby creating a gross-engine torque. Part of this gross-torque is actually eliminated by the engine friction. The engine torque and friction are dependent on the engines revolutions per minute. This relationship is illustrated in the following diagram.



The engines netTorque can be calculated as (grossTorque friction).

As a result, the performance of an engine is only optimal for a very small rpm range.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

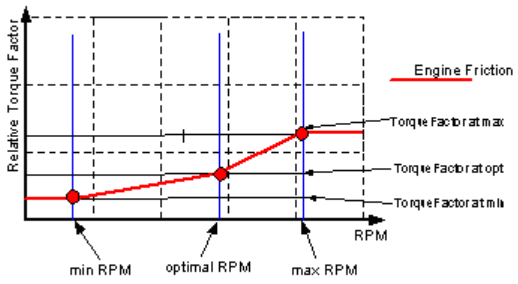
Havoks default implementation uses simplified curves. You only have to specify three points on the *RPM/Torque* diagram.

- *Min RPM/Torque:* If the engine runs slower than minRPM, the Kit assumes that the clutch kicks in such that the engines RPM never falls goes below minRPM.
- *Optimum RPM/Torque:* The RPM value where the engine produces its maximum net torque.
- *Max RPM/Torque:* Defines the maximum RPM for the engine. If the engine exceeds this value, the engine shuts off.

All of these values are just a factor to the engines net torque at optimum RPM. If the engine produces 700 [NM] net torque at 4000 RPM and the engine friction at *optRPM* is 0.5, the engines net torque is 350.

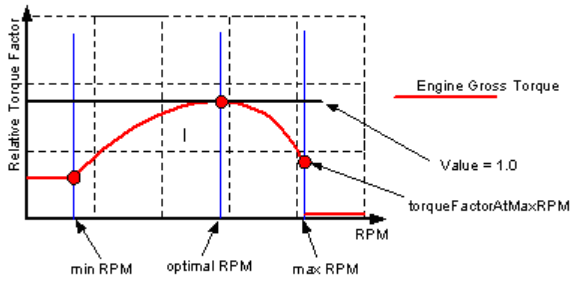
Engine Resistance

The engine resistance is linearly interpolated using two line segments, as shown in the diagram below.



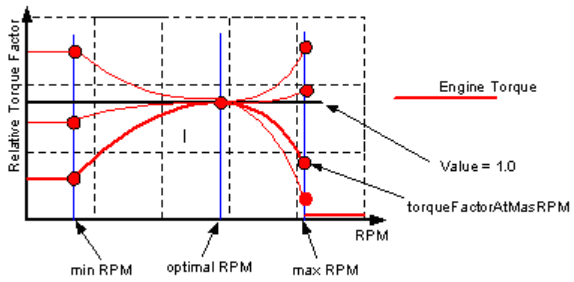
Engine Net Torque

The engine net torque is interpolated using two parabolic curves. The engine net torque at the optimal RPM is defined as the *torque* factor. The values at the minimum and maximum RPM are just relative factors, which the Havok default engine multiplies by the *torque* parameter during simulation.



The default engine uses piecewise parabolic curves to interpolate between three points. So possible engine curves may look like the following diagram:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

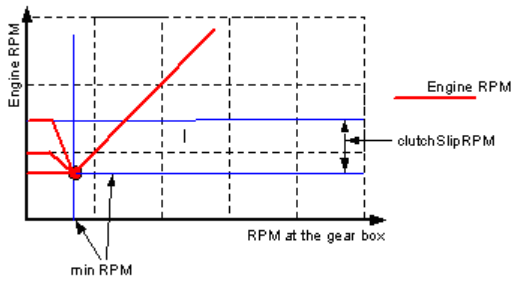


Engine Clutch

A vehicles engine cannot work properly below a certain RPM. To get it working above this lower limit, the driver of the car must press the clutch to allow for some slip. As a result the engine RPM is actually higher than the RPM at the transmission shaft. To simulate this behavior, Havok introduces a clutch slip for RPMs less than the minimum RPM.

For gearbox RPM less than *(0.5 * minRPM)*, engine RPM is calculated using the following formula:

*EngineRPM = minRPM + clutchSlipRPM * throttleInput*



ThrottleInput is a value between 0 and 1 describing how strong the driver pushes the accelerator pedal.

Transmission

To keep the engine working within optimal RPM ranges for different vehicle speeds, you use a variable transmission. The transmission is a device that uses gearing or torque conversion to change the ratio between engine revolutions and wheel revolutions. If the engine revolutions are increased but the wheel revolutions stay the same (making the ratio bigger), the car produces more torque. On the other hand, if the engine revolutions are decreased and the wheel revolutions stay the same (making the ratio smaller), the car achieves higher speeds.

A typical transmission includes a clutch, a gearbox and a differential. The clutch disconnects the engine from the gearbox while you change gear. The gearbox has several gears to translate the torque from the engine to the drive shaft. The differential translates and distributes the torque from the drive shaft to the left and right wheel.

The following formulas describe the relationship between wheels and engine:

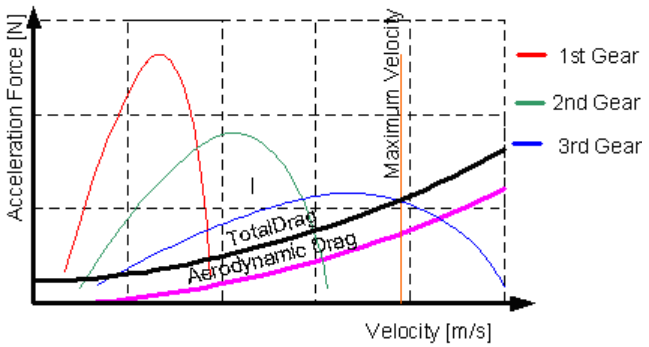
*EngineRPM = WheelRPM / (primaryTransmissionRatio * GearRatio[currentGear])*

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

*WheelTorque = EngineTorque * primaryTransmissionRatio * GearRatio[currentGear]*

- *EngineRPM* is the Revolutions Per Minute of the engine.
- *WheelRPM* is the average Revolutions Per Minute of the wheels.
- *PrimaryTransmissionRatio* is the transmission ratio at the differential.

When comparing the final wheel torque (which results in an acceleration force) to the total drag of the car, you can calculate the maximum velocity of the car.



For practical reasons the ratio between two consecutive gears should be constant:

$$\frac{gear0_Ratio}{gear1_Ratio} = \frac{gear1_Ratio}{gear2_Ratio} = \frac{gear2_Ratio}{gear3_Ratio}$$

The default Havok vehicle implementation provides an automatic transmission, which includes the following parameters:

- The gear ratio of all gears.
- The primary transmission ratio.
- The min and max engine RPM when gear shifting should be performed.
- The time it takes to shift a gear (ClutchDelayTime).
- WheelsTorqueRatio, which defines how the engine torque should be distributed to the wheels. For example if you want to create a front-wheel driven car, the two front wheels should have a value of 0.5 and the rear wheels a value of 0.0. The sum of all wheel torque ratios should be 1.0.

Tuning tips

A good way to start tuning the gear ratios is to set the gear ratio for the top gear to 1.0 and the primaryTransmissionRatio using the following formulas:

*TopSpeedCar[m/s] = TopSpeedOfCar[mph] * 1.605 / 3.6*

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

MaxWheelAngularVel = TopSpeedCar[m/s] / wheelRadius

*MaxWheelRPM = MaxWheelAngularVel * 60 / 2 * PI*

PrimirayTransmissionRatio = maxEngineRPM / MaxWheelRPM

The rest of the gears can be set using a gear-to-gear ratio of 1.5, e.g.

$$\frac{gear0_Ratio}{gear1_Ratio} = \frac{gear1_Ratio}{gear2_Ratio} = \frac{gear2_Ratio}{gear3_Ratio} = 1.5$$

Brakes

Brakes slow down a vehicle using a mechanism that converts kinetic energy into heat energy through friction. The most common example of this is found in the wheels of cars, where the brake shoes or disc pads press against the brake drum or brake disc, thereby creating a braking torque on the wheels.

The Havok Vehicle Kit multiplies the input value from the brake pedal by the maximum brake torque. The resulting torque is then distributed to the wheels.

Applying brake torque to a wheel actually means reducing the maximum side force the wheel can resist during corning. The driver of the car can often make use of this behavior to get the car into a sliding mode. The easiest way to achieve this is to use the handbrake, which totally blocks the wheels.

3.7.1.4 At the Limits or How to Cheat Physics

Introduction

Physically accurate driving behavior is very challenging and can be lots of fun, especially if your game is focused on realism. However, if the focus of the game is to be fun while easy to learn, you probably need to use several tricks to help or cheat the driving simulation. This chapter covers several tricks and algorithms used in the Havok Vehicle Kit and demos.

Subjective Time and Real-Time

In a driving game, players do not expect the game to run in real-time - they expect it to run faster than that! They want to drive around at 200 miles per hour but also to be able to take sharp turns around city corners. They expect the car to react faster to input commands than a real car would.

However, in certain situations, a real car behaves too fast for the player to react. In these situations the player wants a slow motion behavior, especially if the car is sliding or crashing.

This creates a dilemma, because you need to simulate the car faster and slower than real time, at the same time. Fortunately, the Havok Vehicle Kit allows you to "cheat" its physics simulation in order to deal with this. You can achieve desired behaviors relatively easily using a few parameters.

The following cheats allow you to increase the cars responsiveness:

- Increase gravity or add an extra gravity to the car

- Increase the friction coefficient of the tires up to 3.0
- Make the suspension harder
- Add extra yaw torque and extra yaw angular impulse

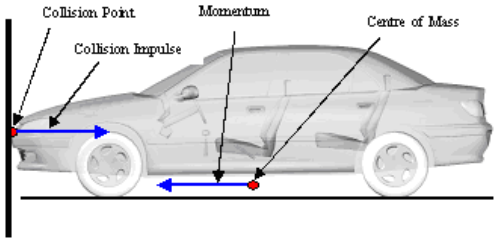
These cheats allow you to decrease the cars responsiveness:

- Lower the center of mass or virtually lower the center of mass by reducing all friction torque factors.
- Increase inertia
- Damp angular velocities
- Apply extra forces and torques (e.g. motorcycle controller)

Center of Mass

In general, a low center of mass results in improved car handling. Lowering the center of mass results in reduced roll and pitch torques during cornering, braking and accelerating. The car sticks to the ground better. It also reduces the likelihood of a car flipping over.

However, there are some limits to how low you can shift the center of mass and still have a realistic simulation. If the car crashes, a low center of mass may make the front of the car jump upward. The reason for this is that the point of collision is above the center of mass, thereby creating an angular impulse. If the center of mass is moved below the ground, the roll and pitch torques are reversed, resulting in a very unrealistic behaving simulation.



There are tricks that allow you to lower the center of mass without getting these negative effects. If the angular impulse at a collision is a problem, increasing the inertia up by a factor of 2 for the pitch axis reduces this effect. Alternatively, rather than lowering the center of mass of the rigid body, you can effectively lower the center of mass by setting the *torqueRollFactor* and *torquePitchFactor* to values less than 1.0.

Inertia

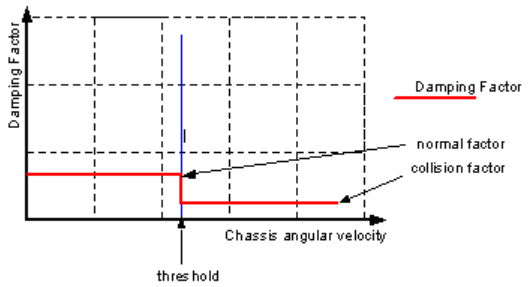
If a very fast car hits a solid object, it might start spinning very violently, which would be frustrating for a player. However, there are a few tricks that you can use to decrease the likelihood of high angular velocities. One way is to simply increase the inertia. As there are three values, one for each axis, you can tune the different axes independently. Changing the inertia a little (20%) changes the entire car behavior. Do not try to change the inertia dramatically.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Angular Velocity Damping

If a car takes a big jump, its angular velocity does not change while it is in the air. This may result in the car flipping over when it hits the ground. This accurate physical behavior might be frustrating for the player. Hence, a mechanism to damp the angular velocity is provided.

The Havok Vehicle Kit provides a controller that damps angular velocity. However, you might not want to damp the angular velocity for everything, since you want to see a spinning car after certain collisions. Havok measures the current angular velocity and if it is below a certain threshold it damps it using the normal angular velocity damping. If it is above the threshold, Havok uses the collision angular velocity damping:



The range for the normalSpinDamping should be between 0 and 3.0, and values between 0.5 and 1.5 should give reasonable results. A spin threshold of means that if the car spins at less than 180 degrees per second, Havok uses *normalSpinDamping*.

If your car uses realistic collisions, then *collisionSpinDamping* values between 0 and 0.5 are reasonable. However if your gameplay results in lots of collisions, then use high values between 3.0 and 10.0. This should remove any high angular velocities. Using this technique, the player never loses orientation.

Friction Equalizer

The tires of a stopped car on a flat surface are pressed against the ground with a certain force, called static load distribution. However, if the car is moving, these forces change due to various effects. This is called dynamic load distribution. The *frictionEqualizer* parameter allows you to change dynamic load distribution to be more equal to the static load distribution.

The effect is that the driving behavior of the car does not change when the car accelerates, brakes, goes over hilly terrain, etc.

Setting this parameter to values between 0 and 1.0 virtually averages all the suspension forces. For example, lets say the value is set to 0.8, and the back tires are not touching the ground while the front tires are. The front tires are pressed onto the ground at 10000N of force. In this case, the vehicle Kit redistributes this 10000N to the individual tires using the following equations:

Front tires: *5000N* (1-0.8) + 5000N*2/4*0.8 = 3000N*

Back tires: *5000N*2/4*0.8 = 2000N*

Hence, if you set *frictionEqualizer* to 0.0 you get physically correct behavior. If you set this parameter to higher values (up to 1.0) you improve the driving behavior on rough and off road terrain.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Extra Angular Torque

When a car starts to slide, it becomes very difficult to control. Even professional racing drivers sometimes have problems stopping their car from spinning. Applying some non-physical impulses and torques around the yaw axis can help to control the car in sliding conditions.

The *extraTorqueFactor* uses the current steering angle to create a torque around the yaw axis e.g. steering to the left creates a small constant torque to the left. Good values range from 0 to 2.

Due to coordinate system differences, this factor may need to be multiplied by 1.0. Check the behavior. If it is wrong, simply negate the value.

Extra Gravity

One of the most common and best tricks for improving the overall game driving experience is to add an extra gravity to the car. This has the basic effect of accelerating the subjective time. Good values are up to three times the normal gravity.

Extra Constraints

In many game scenarios, you need some extra forces to get the desired effects. A typical example is a motorbike game where a small helper function keeps the bike upright. There are endless ways to change the physical behavior of the car, just use your imagination.

3.7.1.5 Vehicle Kit Simulation Output

Introduction

When using a simulated vehicle in your game, you will obviously need to display the vehicle on-screen you may also want to make the players driving experience even more realistic by adding force feedback, sound, and tyremarks. The Vehicle Kit simulation provides you with all the information you need to do this. You can find more detailed information about how to use the relevant classes in the Vehicle Kit programming guide.

Graphics

To draw the vehicle, you just need the local to world transforms of the chassis and the wheels. As the chassis is already a rigid body in the physics simulation, you can simply get the transform from this rigid body. Because the current Vehicle Kit does not require the wheels to be simulated as rigid bodies, you can get the wheel transforms from the *WheelInfo* struct in hkpVehicleInstance.

Visual Debugger

The Havok Visual Debugger can be used to show debug information for Havok vehicles. The Vehicle Viewer will show the wheels and suspension of any vehicles in the simulation. For more information see the Visual Debugger section of this manual.

Force Feedback

The best approach when using force feedback is to take the values from the simulation and pass it to the steering wheel. Note that in reality the force feedback strength is a combination of different factors:

- *Engine rumble* The rpm of the engine can be translated in a sine wave that can influence the steering force. This value is stored in hkpVehicleInstance::m_rpm.
- *Wheel rotation rumble* Instead of using the engine, you can use the angular velocity of the wheels to define the frequency of the steering force (retrieve from the `WheelInfo::m_spinVelocity` member.)
- The *centripetal (side) force* that acts on the steered wheels: This can be retrieved using `WheelInfo::m_sideForce`
- Collisions may generate an impulse on the steering wheel.

Different force feedback devices have very different characteristics and you should test all major steering wheels and joysticks. You may need different settings for different kinds of steering wheels.

Tyremarks

If a car tire slides, the wheel produces rubber marks on the road. Drawing these tyremarks is important for the players immersion in the driving game. The darkness of tyremarks is directly proportional to the energy converted to heat divided by the area of the tyremark.

You can retrieve this parameter from the simulation from the `WheelInfo::m_skidEnergyDensity` member. The resulting value should be scaled to achieve the desired effect.

Sound

Sound is probably the second most important feedback the user gets from playing a racing game. A typical vehicle sound is a simple combination of the different vehicle sound sources:

- Engine sound based on engine RPM
- Skid sound based on `WheelInfo::m_skidEnergyDensity`
- Wheel sound based on terrain and the angular velocity of the wheels
- Air sound based on the velocity of the chassis

Sound is very important for tuning. If you intend to release a game with sound, then get the sound working before you start tuning.

Camera

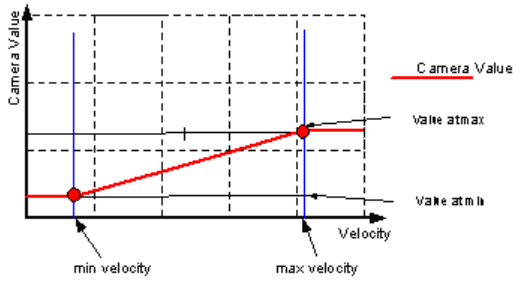
The player sees the whole scenery through the eyes of the camera. One goal for a car game camera is to be unobtrusive. The camera should just feel natural. A bad camera can easily destroy the gameplay without it being evident that the camera is the main problem.

In principle there are several ways to implement a camera. A damped mass-spring system technique is used in many past Havok demos. However, it is not good enough for game cars because:

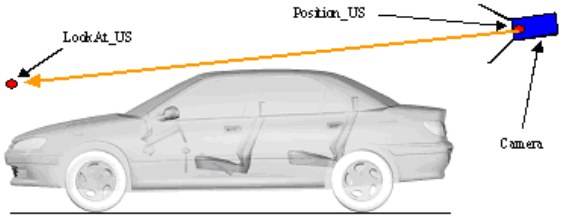
- It results in a very shaky camera
- It is difficult to control
- The distance between the car and the camera changes.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

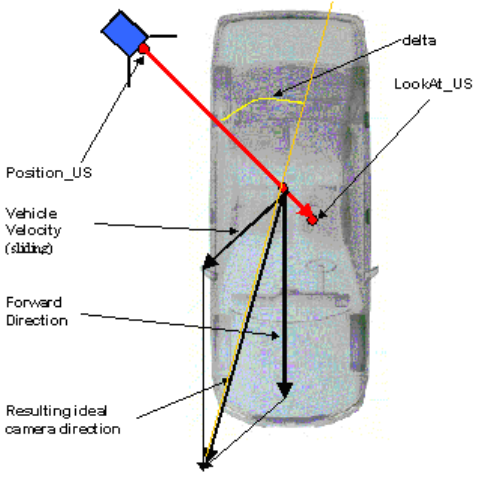
The default Havok vehicle camera, hkp1dAngularFollowCam, uses a different approach. The damped mass spring technique only applies to calculating the direction of the camera. All other values are fixed. The camera allows for changing its behavior for different car speeds. In order to get the value of one parameter, the camera implementation interpolates between two velocity values:



The default vehicle camera is positioned at a constant distance from the center of mass of the chassis rigid body, and looks at a fixed point, also relative to this center of mass. The camera always looks from its position to the lookAt position. Hence you need to specify these two positions relative to the center of mass of the chassis rigid body, using the simulations world co-ordinate system (world space):



The ideal camera direction is a combination of the direction of the vehicle and the direction of its velocity. The *m_speedInfluenceOnCameraDirection* value weighs the influence of the vehicles velocity. A value of 0 means only the vehicle direction counts, a value of 1 results in roughly equal weighting. Higher values base the ideal direction wholly on the vehicle velocity.



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The *current camera yaw angle* tries to match the ideal camera direction. The *m_angularRelaxation* parameter defines how quickly the camera follows this direction. Values between 2 and 6 are reasonable. The higher the value, the easier it is to keep the car in view.

Based on the current camera yaw angle, the two camera positions, the position and the lookAt position, rotate around the world up-axis to get the final camera position and camera lookAt position.

3.7.1.6 Pre Tuning Checklist

Before tuning a vehicle, it is highly recommended to check the following items.

- The size of the geometry should reflect real world sizes. If you use meters, Newtons and kilograms as your base units, it should make the job easier.
- Get a very simple car moving. Use your final game graphics and tracks if possible.
- Use the gamepad, joystick or steering wheel to tune the car. Do not use the keyboard for initial tuning.
- Make sure you get all possible feedback from your car:
  - Tyremarks
  - Motor sound and skid sounds
  - Shadows
  - Optional: Force feedback
- Set up a decent camera close to the car. Use popular driving games as a guide.
- Set the center of mass first as it influences all other parameters. Changing the center of mass is more difficult late in development than at the beginning.
  - The center of mass should be inside the chassis, never below the ground.
  - Collisions should feel realistic, banking and diving can be tuned without changing the center of mass.

  One interesting change to try is to lower the center of mass of the chassis rigid body. This has a big affect on the drivability of the car, as it is less likely to roll over on sharp corners with a lower center of mass. You can do this easily by calling the hkpRigidBody::setCenterOfMassLocal() method. Be careful, however, since the suspension hardpoints are specified relative to the center of mass of the chassis rigid body, you will have to move these in the opposite direction to your center of mass shift to put them back into the correct position after the shift has been applied.

3.7.1.7 Tuning

Tuning your car will take a long time. In principle it means repeatedly playing the game thoroughly and using all the parameters, thereby learning and improving their influence.

This tuning cycle should include:

- Normal driving situations

- Collisions
- Sliding
- Jumps

So take your time and play with it. Never forget that after a while you will become an expert at your own game - however, keep in mind that your game should also be playable by beginners.

3.7.2 Vehicle Programming Guide

This section provides an overview of the core classes that define and implement the set of behaviors and algorithms associated with car simulation and driving. The Havok vehicle SDK provides a complete game-oriented implementation of a vehicle and its components, such as its wheels and transmission. This default implementation is fully functional and easily tweakable, and a number of game developers have used it "as is" in their games.

3.7.2.1 Overview

We have organised the vehicle SDK so that it is easy to separate data that changes at runtime from data that is the unchanging specification of a vehicle. This makes it easy to share vehicle definitions between multiple vehicle instances.

A vehicle can be considered to be composed of several components. All of these components are provided a default implementation by Havok that can be easily overridden to satisfy your own games needs.

- **hkpVehicleInstance** : The class that holds the vehicle together. This contains pointers to all other components. It provides access to all output of the vehicle, as described in Vehicle Kit Simulation Output section. This class stores all cached and runtime data for the vehicle, and so cannot be shared between vehicles.
- **hkpVehicleData** : a container for vehicle specs such as: - the number of wheels - radius, mass, friction, etc. for each wheel - chassis orientation and the roll factors that affect the chassis - description of the vehicle for friction solving - any unchanging data that is not particular to any other component (see the refman item hkpVehicleData for full details)
- **hkpVehicleWheelCollide** : Handles the collision detection between the wheels and the ground. The default implementation owns an hkpAabbPhantom to perform raycasting (see vehicle raycasting) which ignores the chassis rigid body, and so because the rigid body for the chassis is specific to each vehicle instance, hkpVehicleWheelCollide cannot be shared. This component allows you to override the friction calculation in order to implement friction maps based on collision information input (shape keys etc.). See hkpVehicleRaycastWheelCollide::calcSingleWheelGroundFriction().
- The components hkpVehicleAerodynamics, hkpVehicleBrake, hkpVehicleDriverInput, hkpVehicleEngine, hkpVehicleSteering, hkpVehicleSuspension, hkpVehicleTransmission, hkpVehicleVelocityDamper all have a similar structure. They contain only data that is set during vehicle setup. Each has a method that is called every step to calculate the effect the component will have on the vehicle. This calc function has well defined inputs and outputs the output structure is given in the base class (e.g. AerodynamicsDragOutput in hkpVehicleAerodynamics). All these components can be shared.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- Additionally, we provide an hkpTyremarksInfo class to demonstrate how to handle tyremarks left by the vehicle, and an hkp1dAngularFollowCam as a sample vehicle camera.

Figure 3.64: Components of a hkVehicle

Note that the shareable components contain only public data members (that should be set by the user to specify the vehicle), while the non-shareable components, in addition to such data, also contain state information and data specific to each vehicle (such as the chassis rigid body).

3.7.2.2 Integrating Havok Vehicles into your game

Setting up a vehicle

You can use vehicles in their serialized form to instantly load a vehicle into your world, have a look at the vehicle serialize gameelements demo. This demo uses shows how to do this using binary and XML serialization.

To setup a vehicle without using serialization, have a look at the VehicleApi demo.

Note that:

- the hkpAabbPhantom in the hkVehicleWheelsCollide needs to be added to the world on construction of the vehicle and removed on destruction.
- init() must be called on an hkpVehicleInstance after all components have been setup.

Simulating a vehicle

Once a vehicle has been setup, the hkpVehicleInstance class can be added to the world like any other action since it is an hkpUnaryAction. It is also possible to call the applyAction method in the stepGame function yourself if you want more control, without adding the instance as an action.

For each physics step, you it is necessary to get the user input and steer the vehicle, update the camera following the vehicle, update any visible (or audible) tyremarks and apply any additional vehicle behaviour you want.

Additional vehicle behaviour

There are many cases that you could want to modify the behaviour of the vehicle when driving it. For example, during simulation you may want to:

- lean a motorcycle to the sides, simulating the motion of the driver.
- make sure that an off road car doesn't stay upside down for too long.
- damping the movements of a vehicle in certain directions to increase stability

and whatever other physical properties you can imagine your vehicle having. The Havok SDK is flexible enough to allow you to do all these. The Motorcycle demo shows how to tilt the vehicle during the steering, and some of the more common behaviours are already integrated in the SDK. For example, angular velocity damping is provided by the hkpVehicleDefaultVelocityDamper and there is a hkpReorientAction.

3.7.2.3 Simulating vehicles using ray casting

Why use a raycast solution?

The idea behind using a raycasting vehicle implementation is to avoid simulating the wheels as fully independent rigid bodies. Some of the reasons for this are as follows:

- In order to keep the wheels attached to the chassis, complex forces and constraints need to be applied between them during simulation, with the consequent expense of CPU. Rigid bodies in a simulation have a high amount of degrees of freedom, while wheels in a vehicle have not.
- Real vehicle wheels are not actually rigid bodies. They deform during driving, which is the reason why cars stick to the ground. Simulating wheels as rigid bodies doesn't provide good traction due to the fact that the interaction between a road surface and a tire is very different to that between two rigid bodies.
- Wheels spend most of their time on the ground (except when the vehicle is crashing or jumping). Thus, collision detection and resolution between wheels and ground can be greatly simplified, again reducing the CPU cost.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The approach taken by a raycasting vehicle implementation is to assume that the wheels will always be attached to the chassis, and that their position can be calculated at any time as a function of the current position of the chassis and the ground. This approach frees the vehicle simulation from the task of keeping the wheels attached to the chassis, with consequent improvements in speed and stability.

In a system where the wheels are simulated by the physics engine, there are two intertwined feedback loops: forces on wheels and on the chassis depend on the relative positions between them. The output of the vehicle simulation doesn't guarantee a valid state (a valid relative position between chassis and wheels): the forces calculated by the vehicle simulation try to reduce the error between the current state and the valid state. This complex feedback loop needs to be evaluated at relatively high frequency (small timestep) to guarantee stability.

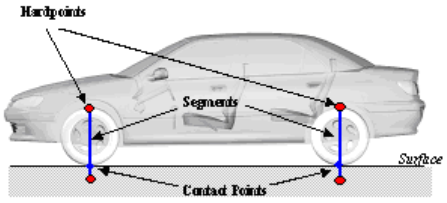
The figure above shows a system where the wheels are not simulated, but instead placed in a valid position by the vehicle simulation. The state after the vehicle simulation is always a valid state and no error needs to be corrected. The feedback loop is much simpler and lower frequencies (bigger timesteps) can be used.

Calculating the position of the wheels

As the name suggest, this approach calculates the new position of the wheels using raycasting. In order to calculate the position of each wheel, a ray is cast from the chassis at the point where that wheels suspension is connected also known as the hardpoint - in the direction of the suspension. The length of this ray is the maximum distance the wheels can be from the chassis.

If the ray intersects with the landscape, the wheel is considered to be in contact with the ground. The position of the wheel can be calculated from the intersection point (subtracting the radius).

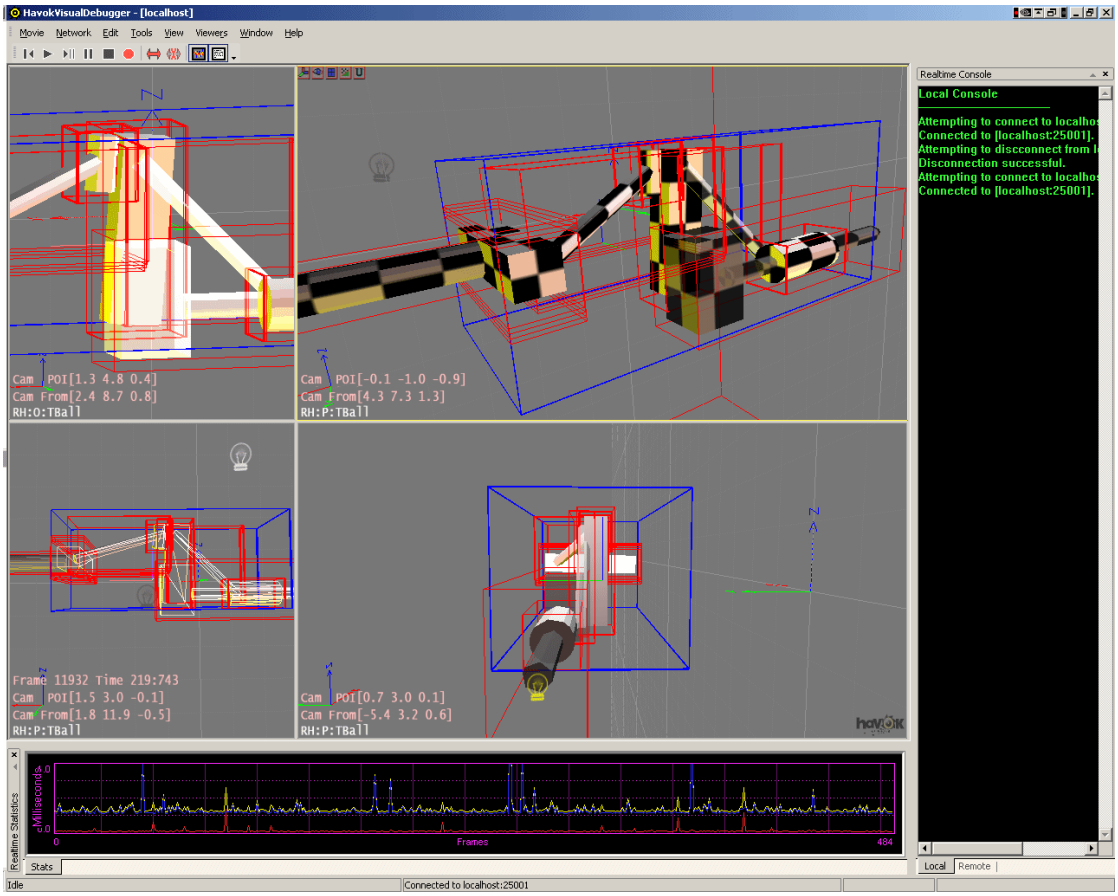
If the ray doesn't intersect with the landscape, the wheel is considered not to be in contact, and its position is set to the end point of the ray (again subtracting the radius).



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.8 Using the Visual Debugger

3.8.1 Introduction



The visual debugger connects from a PC (client) directly to your game running on a host platform (the game console, refered to as the Server) and displays a real-time graphical view of an active Havok simulation. These visualizations can be viewed in real-time or captured to a file for future reference. File capture can also be used for submitting support queries. The ability to view the physics independently of the graphics integration can make it easier to find and correct errors in both your simulation code and the graphics integration. Detailed performance statistics (both timings and memory) can also be displayed and captured.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The local area network is used to stream the state of the physics world between the simulation - your game - and the visual debugger application in real-time. It has a client/server architecture and so is split into two main components. These are:

- A stand alone Windows application that connects to your game and displays the state of the physics and user display information in real-time.
- A game side that links with your game along with the rest of the Havok runtime libraries.

3.8.2 Quick Start Guide

To get the visual debugger up and running follow the quick start guide for your specific platform.

3.8.2.1 Connecting to the demos

Once you have completed all the platform specific preparation for the visual debugger you should be able to get it up and running with the demos that ship with your Havok distribution. For help compiling the demos please refer to the main Havok Quick Start Guide in the docs directory of your Havok distribution.

- Run the Havok demos on a supported console or PC with -d as one of the command line arguments. You can use the `hkdemo.cfg` file in the demos directory to specify command line arguments as well, especially if running on a console.
- Find out the IP address or network name of the game side (which is actually the server). Consult the platform specific quick start guides above for information on how to obtain a suitable IP address. Please note that the visual debugger game side will only be created once a game has been run and the -d command line argument has been specified. Specifically, no visual debugger exists on the game side when browsing the available demos using the demos menus system, it is created when you run a demo and a Havok Physics world is created.
- Run the visual debugger application, this is located under the tools directory in your Havok distribution. The visual debugger application has been tested on Microsoft Windows 2000 or Windows XP and requires DirectX 8.1 runtime or above to function.
- Select **Network - Connect** and enter the IP address of the visual debugger game side where your game is running.
- The demo should be displayed in the visual debugger application main window. Try turning on and off some viewers by toggling the entries in the **Viewers** menu. Note that the **Viewers** menu will be empty until a valid connection has been established.
- You could also try recording a movie by selecting **Movie - Record...** and specifying a file name. Be warned though these files can be quite large as they are an exact dump of the network traffic from the server, but they compress well using zip for instance if you need to send them to Havok support.

3.8.3 The Visual Debugger Game Side

In order the use the Visual Debugger (VDB) with a Havok Physics game, you must perform the following steps:

- Include the visual debugger and the physics related contatainer:

```
#include <Common/Visualize/hkVisualDebugger.h>
#include <Physics/Utilities/VisualDebugger/hkpPhysicsContext.h>
```

- Instantiate a context so that the VDB will know the physics worlds it can visualize:

```
hkpPhysicsContext* physicsContext = new hkpPhysicsContext;
physicsContext->addWorld( myWorld ); // add all worlds as you have
```

  A Process is anything that gets run by the VDB, such as a Viewer. A ProcessContext is a way of getting information to those processes in a non physics-specific way. When a Process (Viewer) is created it is given this array of contexts. As such the context can hold whatever you like. In this case we want one that holds physics related information, so we use a hkpPhysicsContext, that is just a container that holds hkWorlds and listens to their deletion so that the list is always valid. Note that the hkProcessContext itself is not Havok memory managed, it is just an interface so that you can use your own container classes for whatever data you want to expose to processes you create and use with the VDB. The hkpPhysicsContext sub class is havok mem managed though.

- Register all processes (viewers) you want to expose to the VDB:

  There is a singleton called the hkProcessFactory that supplies any instance of the visual debugger with a list of what viewers are available (by name) and how to create them. When you create a Visual Debugger, it will know about the common processes that do not depend on Physics or Animation specifically. These common processes are the DebugDisplay and Statistics viewers. As we also want to be able to view physics specific data, such as broadphases and shapes, we need to register which ones we want. Normally you want to register all of them. hkpPhysicsContext::registerAllPhysicsProcesses() registers all the ones in hctUtilities (it is a static method). If you have processes (viewers) you have written yourself, register them with the hkProcessFactory too, but normally all you need to do is call:

```
hkpPhysicsContext::registerAllPhysicsProcesses();
```

- Create the Visual Debugger server:

  We can then create a visual debugger, given a list of the contexts that you want the processes to see.

```
hkArray<hkProcessContext*> contexts;
contexts.pushBack( physicsContext );
hkVisualDebugger* vdb = new hkVisualDebugger( contexts );
```

- Create a network game side:

```
vdb->serve(/* optional port number */);
```

or open up a capture file to stream directly to a file:

```
vdb->capture(captureFilename);
```

Note: You do not have to call hkVisualDebugger::serve(portNumber) method to use the hkVisualDebugger::capture(captureFilename) or visa versa, they function independently.

- Finally you must step the display by calling:

```
vdb->step( frameTimeInMs );
```

once per frame. If several physics steps pass before the debugger is stepped, then the debugger will only display the positions of bodies at the *last* step. If you don't give the VDB a 'current time' to report, by leaving it at 0, then the VDB will use its own internal timer to work out the time since the last step call. The time is used to govern movie playback so that it can optionaly happen in 'real time' and to provide information to the client during connection.

If you want to use the statistics viewers (highly recommended), then just before you call step() above, you should let the VDB know where all the timer data is. There is potentially a timer stream per thread you use for Havok (see hkMonitorStream::resize() in the hkBase documentation). So the code to pass each timer block would look like:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
struct TimerStreamInfo
   {
       const char* m_streamBegin;
       const char* m_streamEnd;
   };
   hkArray<TimerStreamInfo> threadStreams;

   //
   // In each of your worker threads in turn:
   //
   {PerThread} {
         TimerStreamInfo info;
         info.m_streamBegin = hkMonitorStream::getInstance().getStart();
         info.m_streamEnd = hkMonitorStream::getInstance().getEnd();
         timerStreams.pushBack(info);
   }

   //.. don't make any more Havok calls (or else the end pointer above will not take those times into
     account)

   //
   // In your main thread before vdb->step():
   //
   physicsContext->m_monitorStreamBegins.setSize(0);
   physicsContext->m_monitorStreamEnds.setSize(0);

   for ( int i = 0; i < threadStreams.getSize(); ++i )
   {
      physicsContext->m_monitorStreamBegins.pushBack(threadStreams[i].m_streamBegin);
      physicsContext->m_monitorStreamEnds.pushBack(threadStreams[i].m_streamEnd);
   }

   vdb->step( frameTimeInMs );

   //
   // In each of your worker threads in turn you can now reset the mointors for the next sim step etc:
   //
   {PerThread} {
        hkMonitorStream::reset();
   }
```

- hOnce you no longer require the VDB, delete it (it is reference counted) and whatever contexts you gave it. Any open network connections will be dropped and any files open closed.

```
vdb->removeReference();
physicsContext->removeReference();
```

An example of how to setup the visual debugger can be found at the bottom of the hkDefaultDemo.cpp file that ships with your Havok distribution.

Note for the visual debugger to compile, the application must link with the appropriate libraries and in some cases specific hardware must be installed. Please consult the quick start guide in this chapter for more information for the platform specific libraries required by the visual debugger game side.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.8.3.1 Required Havok Libraries

The following Havok libraries are required on all platforms in order to compile and link with the visual debugger game side:

- hkbase (where hkSocket etc. is defined)
- hkmath (where hkVector4 and hkTransform is defined)
- hkvisualize (where the class hkVisualDebugger etc. is defined, and the non physics related viewers)
- hkutilities (for the default physics viewers, but you can make your own viewers without it)

3.8.3.2 Viewers

Viewers are the link between you game and the visual debugger game side. It is possible to write your own viewers. The current list of viewers that ship with the Havok SDK and their corresponding string IDs is as follows (common ones are in hkVisualize, the physics ones in hkUiltities):

Common Viewers	ID (Unique Name)	Description
hkDebugDisplayProcess	"DebugDisplay"	Shows the global debug points and lines output that result from using HK_DEBUG_LINE etc.
hkStatisticsProcess	"Statistics"	Transmits detailed performance statistics. See documentation on the hkMonitorStreams and HK_TIMER calls for more information.

Physics Viewer	ID (Unique Name)	Description
hkpShapeDisplayViewer	"Shapes"	Displays the simulation geometries. This viewer is used by default.
hkpBroadphaseViewer	"Broadphase"	Displays the broadphase Aabbs in Red.
hkpSimulationIslandViewer	"Islands"	Displays the simulation islands as Aabbs. Active islands are colored blue and inactive islands are colored green.
hkpRigidBodyInertiaViewer	"Tensor"	Displays a magenta Aabb representation of the Inertia Tensor for each rigid body in the world.
hkpRigidBodyCentreOfMassViewer	"CentreOfMass"	Shows the center of mass of each rigid body in the world.
hkpContactPointViewer	"Contact Points (Active/Inactive)"	Displays all contact points in the simulation at the end of each frame. The contact points in active simulation islands are drawn in blue, the contact points in inactive simulation islands are drawn in green. This colour scheme matches the simulation island viewer's colour scheme.
hkpToiContactPointViewer	"Contact Points (TOI)"	Displays TOI contact points as a red arrow. This viewer has some limitations: The TOI contact point is drawn in the contactPointProcess callback. Users may modify the contact point in this callback, and the changes may or may not appear in the VDB. The SDK is free to call user callbacks in any order, so the VDB may draw the contact point before or after the user modifies it.
hkpConstraintViewer	"Constraint"	Displays a graphical representation of all the constraints in the simulation.
hkPhantomViewer	"Phantom"	Displays the Phantoms in the world as yellow Aabbs.
hkpSweptTransformDisplayViewer	"SweptTransforms"	Displays the Continous Simulation start and end frame for objects.
hkpWorldMemoryViewer	"WorldMemory"	Transmits per step memory usage for all hkWorlds (this asks each object how much it is using so is relatively slow).
hkpVehicleViewer	"VehicleViewer"	Displays debug information, including the wheels and suspension, for any Havok vehicles in the simulation.

Using viewers

Once a successful connection has been made to the visual debugger application a list of all the available viewers is presented under the **Viewers** menu. Using this menu viewers can be created and destroyed at runtime.

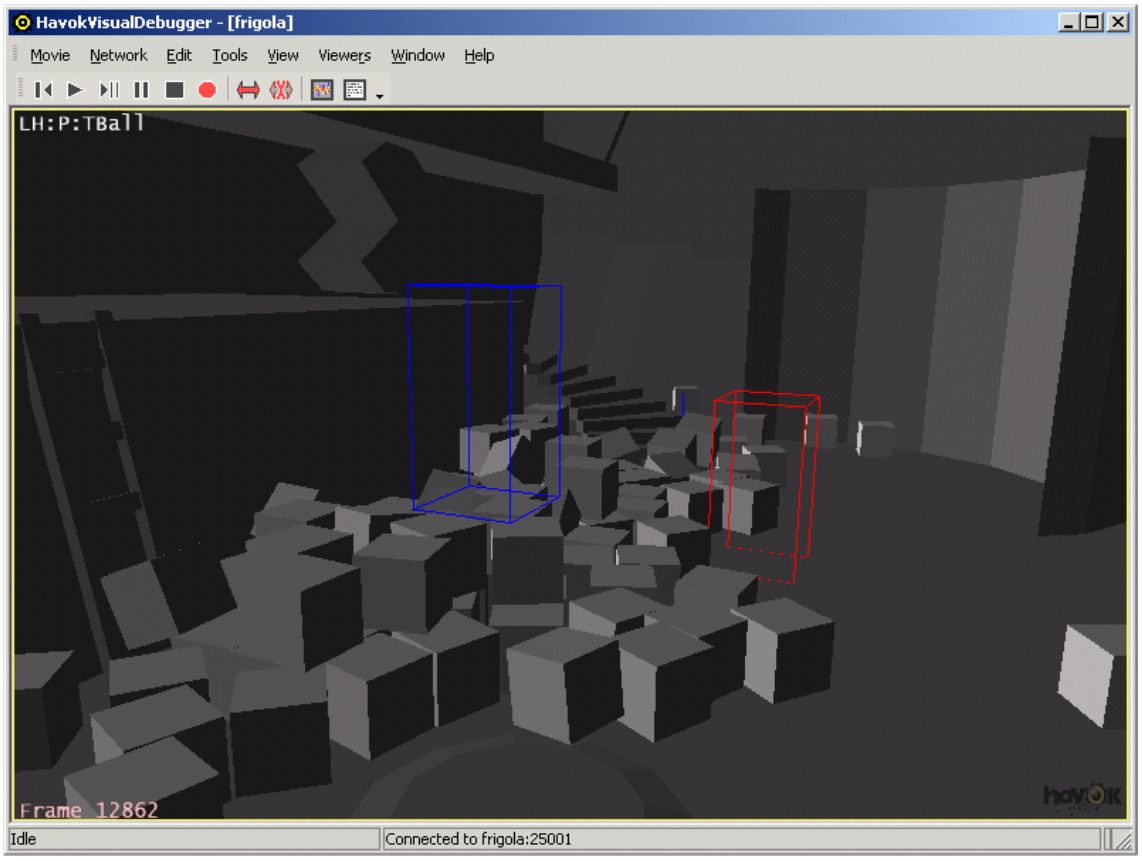
You specify that you want to use a particular process (viewer) by default using the VDB's `addDefaultProcess(const char*)` method and passing it the appropriate string ID (listed above). This registers interest in that process on the game side. This means that when a visual debugger application connects, an instance of that viewer will be instantiated by default. Similarly, you can remove interest in a viewer (such as the default hkpShapeDisplayViewer) using `removeDefaultProcess("Shapes")` - however, this will not remove the viewer from any existing instances of the visual debugger application. You can always enable and disable viewers from the visual debugger application all the default list does is specify which viewers should be enabled by default. This is important when capturing directly to a file using hkVisualdebugger::capture(...) as user interaction is difficult. You can see examples of how add viewers at the end of `hkDefaultGame.cpp` which is included with your Havok distribution.

From a performance point of view, bear in mind that as more viewers are instantiated more network bandwidth will be consumed, this can slow down the simulation. The Statistics based viewiers (perf and world memory) are the largest bandwidth users.

Viewers examples

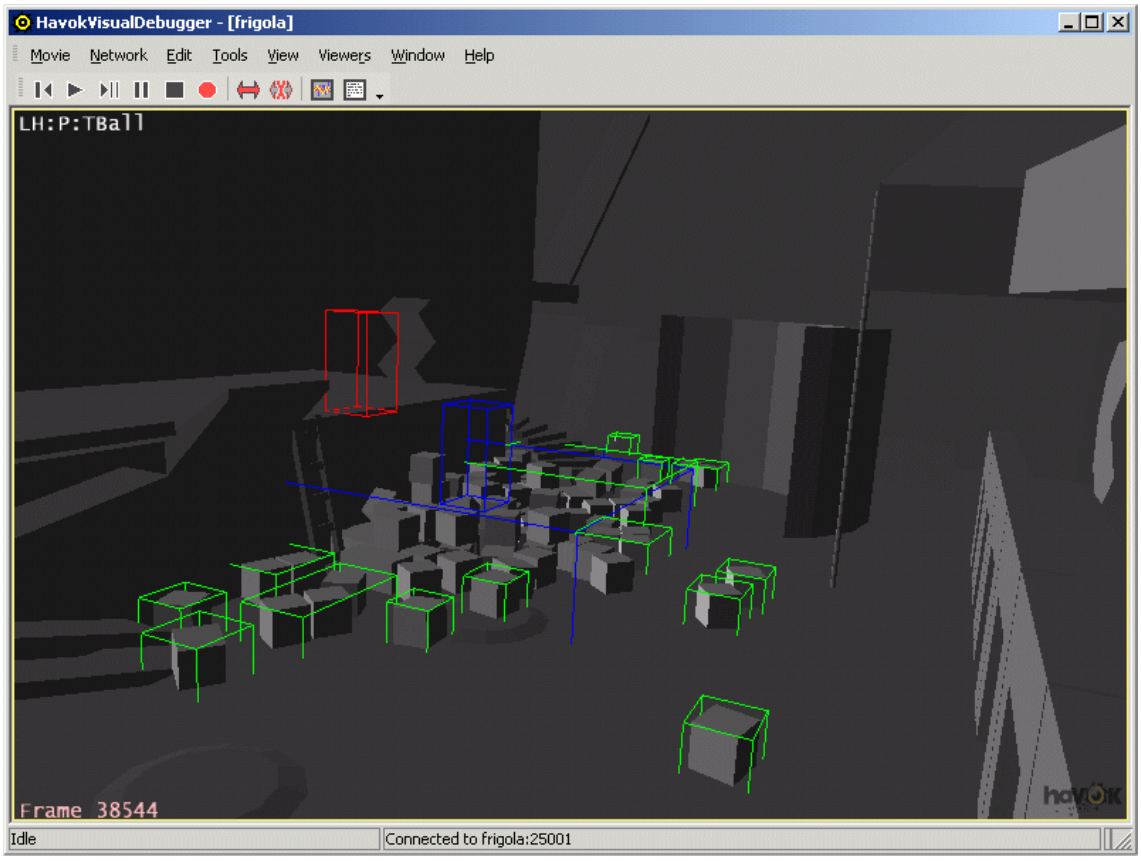
By default, the visual debugger enables just the hkpShapeDisplayViewer and the hkDebugDisplayProcess. This displays the geometries that exist in the simulation and all the global debug points and lines. For example, in the following image you can see all of the rigid bodies, the landscape and the Aabbs of the characters. The debug display was used to display the character Aabbs. This is a convenient way to display information that is not automatically displayed by the visual debuggers' built in viewers. The debug display is described in detail below.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



The hkpSimulationIslandViewer lets you see information about the simulation islands, for example, which objects are being simulated together and which islands have been deactivated. See the image below:

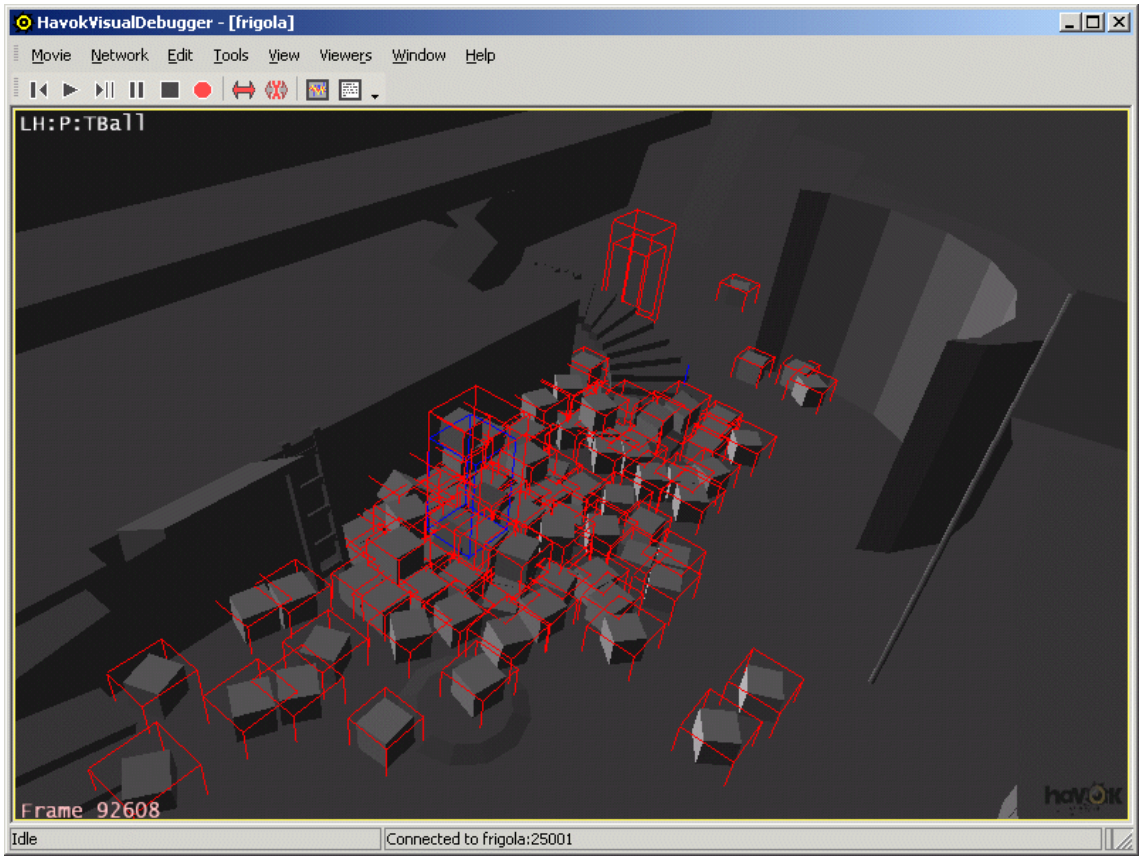
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



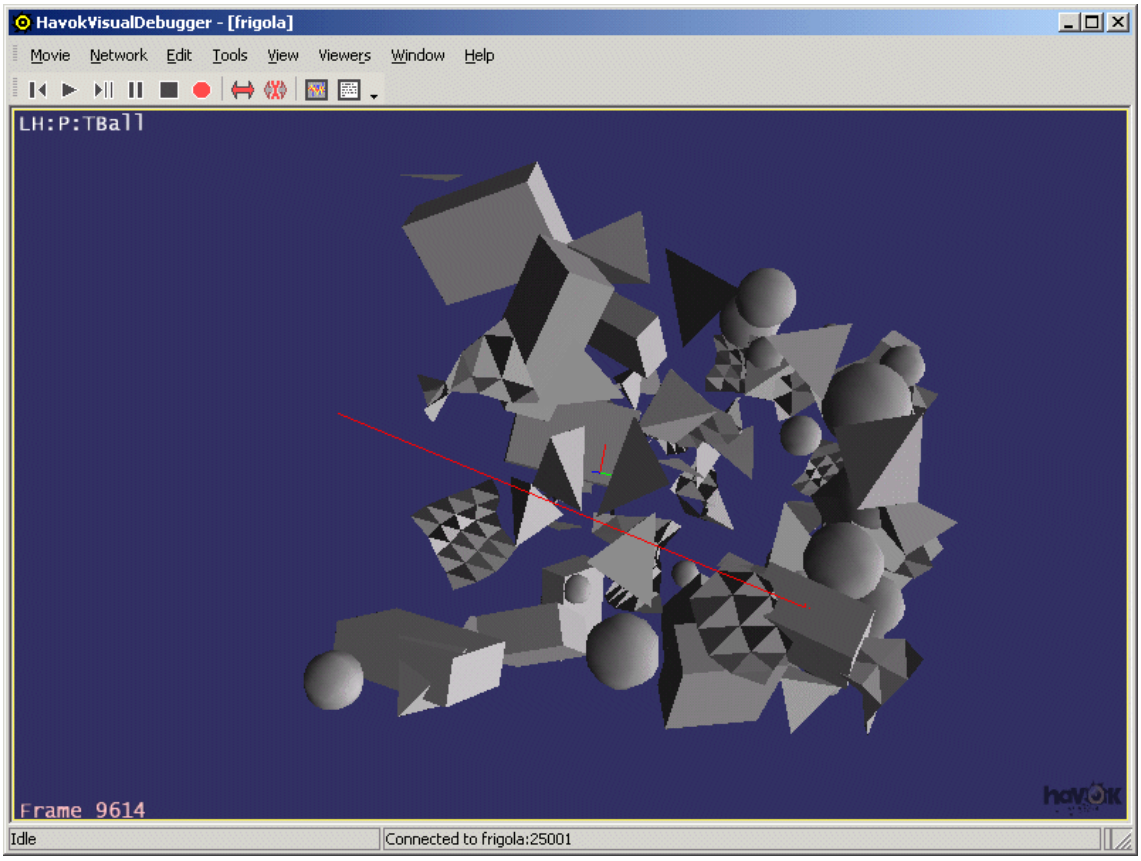
As you can see, inactive islands are shown in green and active islands are shown in blue. Fixed bodies have no island information displayed, as they do not take part in the simulation, they do however take part in the collision detection process. To have the simulation island information displayed by default (so that you don't have to manually select it in the UI) simply call the `addDefaultProcess("Islands")` on the visual debugger game side just after it has been created as in:

```
myVdb->addDefaultProcess("Islands");
```

In the next example, the hkpBroadphaseViewer has been turned on in the debugger by selecting the Broadphase viewer from the Viewer menu. Note that the Viewer menu will not be present (or be empty) until it has connected to a game. The Broadphase viewer displays the Aabb (Axis Aligned Bounding Box) used for each object in the collision detection broadphase.



Debug points and lines are a useful tool for debugging your game. They can be used to debug or visualize anything from forces, velocities and directions to game logic and AI. All debug display output can be viewed by turning on the DebugDisplay viewer (class hkDebugDisplayViewer), and only debug information issued after the viewer is turned on will be viewed. Again the simplest way to do this is to select it from the Viewers menu in the visual debugger application. In the example below the ray-cast is visualized using red user debug lines.



In order to display debug points and lines you can call the following macros in your code:

```
HK_DISPLAY_POINT(position, color);
HK_DISPLAY_LINE(start, end, color);
```

The example below outlines how to display a debug point and a debug line. Note the header file required to use the macros:

```
// Comment in the line below to compile out the debug points and lines...
//#define HK_DISABLE_DEBUG_DISPLAY
#include <Common/Visualize/hkDebugDisplay.h>

...

void drawMyLines()
{
   ...

   hkVector4 start(0.0f, 0.0f, 0.0f);
   hkVector4 end(0.0f, 100.0f, 0.0f);
   HK_DISPLAY_LINE(start, end, hkColor::BLUE);

   hkVector4 position(1.0f, 1.0f, 1.0f);
   HK_DISPLAY_POINT(position, hkColor::WHITE);
}
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The next below entitled The Debug Display details all the functionality provided by the debug display...

The Statistics Viewer

This viewer is unique in that it does not display information in the graphical viewports. Instead it outputs information to special real-time graph windows. There are 6 windows, one for each virtual channel in the timer stream. By default channel 0 is the perf timer on most platforms. This graph can be enabled by selecting the **Window - Perf Ch0** and so on for each of the 6 channels, or by selecting the appropriatly numbered Stat icon in the deafult toolbar. In a multithreaded envirnonment such as on PC or Xbox360, the 6 channels represent timer 0 from from each of the potential 6 threads. For more information on the graph display see the section entitled The Real-time Statistics View.

For Havok statistics to function properly the follow extra requirements must be met:

- Call hkMonitorStream::resize(bufferSize) at least once. Specify a buffer large enough to capture all the stats for a single frame typically at least 100 kilobytes (normally 1MB if you have a complicated simulation you want to time). If you want to check that there is enough memory call hkMonitorStream::memoryAvailable() at the end of a frame before calling hkMonitorStream::reset(). If it returns false then you may have run out of memory.
- Call hkMonitorStream::reset() at the beginning of every frame OR at the end of every frame but AFTER hkVisualDebugger::step() is called. (hkVisualDebugger::step() must be called for the visual debugger to function properly)

If you are not getting statistics as you expect then double check the following:

- The Statistics viewer is enabled and you have a valid connection to your game. Note that in the case of a movie the viewer will have to have been turned on when the movie was created. You can check if a movie has stats by looking at the **Viewers** menu during playback. If there is a Statistics entry and it is checked (but grayed) then stats are enabled.
- Check that enough memory is being allocated for the hkMonitorStream.
- Check that the range of the real-time graph is suitable for your statistics.
- Check that the statistics you are interested in are selected in the Select Stats or Legend dialogue boxes (right-click on the Real-time Statistics view).
- Ensure that your game is calling hkMonitorStream::resize(bufferSize) and hkMonitorStream::reset() appropriately.

When running a multithreaded simulation, you will notice that the performance in each channel seems to be quite variable. This is not spikes in simulation itself but just the jobs being done in different threads each frame. When you step a multithreaded simualtion, the first thread to call stepWorld is the master thread and take more of some types of jobs than others, and the rest of the threads take jobs as they arrise and add more etc.. Thus depending on which thread gets in first and how the subsequent threads arrive to stepWorld, they will get different jobs each frame. The order of work is all dependant on platform thread scheduling as to which thread does what. The overall work is the same, just the statistics for the jobs will jump from one channel to another each frame.

The Vehicle Viewer

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

When enabled, the vehicle viewer will draw some debug information for any vehicles in the scene. The viewer will draw this debug information for any objects of type `hkpVehicleInstance` or any classes derived from `hkpVehicleInstance` in the simulation.

For each vehicle the following components are displayed:

- Each wheel of the vehicle is drawn as a white circle.
- A red line shows the ray that was cast for each wheel.
- For each axle of the vehicle a yellow line is drawn, connecting the center of all wheels that are connected to that axle. This line is an indicator to show what wheels are connected to each axle. If there are more than 2 wheels connected to an axle, this line will zag-zag between all of these wheels.
- The suspension hardpoints are drawn as a yellow cross at the top of each wheel.
- The contact point between the wheel ray and the ground is shown as a yellow cross at the bottom of each wheel. If the wheel is not in contact with the ground, then the yellow cross indicates the current length of the suspension.

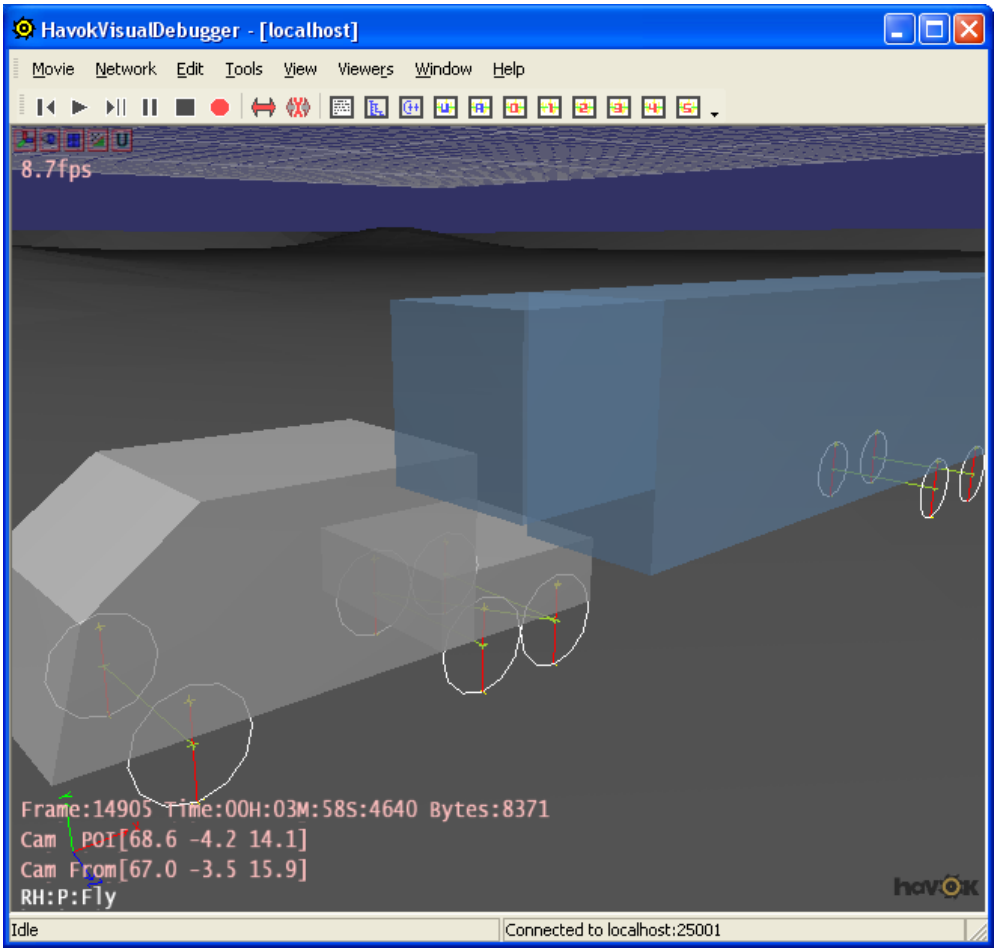


Figure 3.65: Visual Debugger Vehicle Viewer

The position of the information displayed by the vehicle viewer (wheels, ray casts) will lag behind the position of the chassis by one frame. This is because the vehicle simulation is an action, which is evaluated

before the Havok world is simulated, and the resulting forces are applied to the chassis rigid body as input for the world step. The Visual Debugger displays the state of the world after each Havok step. The vehicle viewer has been designed to show the input for the vehicle that caused it to reach the current location, to aid debugging.

3.8.3.3 The Debug Display

The visual debugger game side incorporates a debug display (class hkDebugDisplay in `hkvisualize/visualdebugger`). The header file defines many global macros which can be used anywhere in your code. Remember though that the calls have immediate effect only, so if the DebugDisplay Process is not enabled the calls will do nothing (apart from also going to the default local debug display if one).

Here is a list of all the macros available:

```
HK_DISPLAY_POINT(position, color)
```

Displays a debug point in your scene. The position should be specified as a hkVector4 and the color is an unsigned int in 0xARGB format. You can also used the hkColor class to make specifying colors a little easier.

```
HK_DISPLAY_LINE(start, end, color)
```

As above but takes a start and end point to draw a line.

```
HK_DISPLAY_TEXT(text, color)
```

Sends text to the console in the visual debugger application. This text is displayed in the Remote tab of the dockable Real-time Console view. Currently the color has no effect in the visual debugger application.

```
HK_DISPLAY_GEOMETRY(geometeries, color)
HK_DISPLAY_GEOMETRY_WITH_TRANSFORM(geometeries, transform, color)
```

Displays an array of geometries in wireframe. The array is a hkArray of hkDisplayGeometry pointers (i.e. a hkArray<hkDisplayGeometry*>). The hkArray class is documented in the hkbase documentation. The hkDisplayGeometry class and its derivatives (for example hkDisplayAabb) are in the hkgeometry2 Havok library. In HK_DISPLAY_GEOMETRY_WITH_TRANSFORM a transform can be specified to position the array of geometries in the world, transform is a reference to a hkTransform (see hkmath documentation for more information). hkDisplayAabb is used by the Broadphase viewer to efficiently transmit Aabbs over the network by sending only the min and max extents (6 floats) as opposed to 12 individual debug lines (70 floats!)

The following macros can be used but are a little more tricky to work with:

```
HK_ADD_GEOMETRY(geometries, transform, id)
```

Add a geometry to the display world which can be updated by setting it's transform using the HK_UPDATE_GEOMETRY macro and removed from the display by calling the HK_REMOVE_GEOMETRY macro below. The geometries parameter is a hkArray of hkDisplayGeometry pointers or a hkArray<hkDisplayGeometry*> as described above. This array of geometries all form part of the same local coordinate space whose position and orientation is governed by a single transform. The transform is a reference to a hkTransform (from hkmath, see Havok documentation for more information). The id is an unsigned integer and should be unique with respect to all other geometries added to the display world. The convention used by the Shape viewer for example is to use the pointer to the Havok hkpCollidable instance as an id. You can use whatever convention you like as long as your id's are unique.

```
HK_SET_OBJECT_COLOR(id, color)
```

This function will set the color of an object that has been added to the display world. Please note that this function does have it's flaws for example if you set the color on an object and then later connect to you game with the visual debugger application the color change will have been missed. However for events that happen periodically this is not usually a problem. So this simple implementation was preferred.

```
HK_UPDATE_GEOMETRY(transform, id)
```

Moves a geometry that has been previously added to the display world with a call to HK_ADD_GEOMETRY. The id must be the same as the id used in the original add geometry call and the transform is a reference to a hkTransform (in the hkmath library).

```
HK_REMOVE_GEOMETRY(id)
```

Removes a geometry from the display world. The id should be the same as the one used in the corresponding HK_ADD_GEOMETRY call.

```
HK_UPDATE_CAMERA(from, to, up, nearPlane, farPlane, fov, name)
```

The update camera command allows the user to specify their own cameras. Use this to have the visual debugger track your vehicles and characters in your game. You can supply as many as you like and update them as often as you like. (Please note that only the first 32 will be listed in the visual debugger applications drop down menu!). The from, to and up parameters are specified as Havok hkVector4's (see hkmath documentation), nearPlane and farPlane are specified as hkReals (which is just a float by default) and name is a null terminated c string identifying the camera.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.8.3.4 Visual Debugger Debug Info

When using the visual debugger it can be useful to see what the game side is actually trying to send to the visual debugger application. To cater for this, a macro is provided that can turn on debug info output. Note that this text should appear on your games console output, it is not transmitted to the visual debugger application.

```
HK_VISUAL_DEBUGGER_INFO_SET_LEVEL(LEVEL);
```

This debug info is defined as a series of levels. Level 0, the default, shows errors only. Level 1 shows the following high level information:

- Server has been created and the port number.
- All viewers that are registered with the game side and their names.
- A new connection has been received.
- All viewers that are instantiated for a new connection and their names.
- A geometry has been sent. Displays its ID and some high level information about it.
- A remove geometry message has been sent. Displays its id.
- The connection has just died and the game side is about to kill all resources relating to that connection.
- The visual debugger game side has been deleted.

Level 2 adds even more information about all of the geometries being displayed, such as the number of triangles in a given geometry.

3.8.4 The Visual Debugger Application

This is a GUI application which runs on Microsoft Windows 2000/XP. The user interface is configurable and has several components: a graphical viewport, a menu, a configurable tool bar system and dockable components for displaying console output and real-time statistical information. Each area has a context sensitive menu.

By default the application uses Direct3D, but you can make it use OpenGL if you so desire by using the *-ogl* flag at the application's command line. Some cards do not have good OpenGL support , but if you are experiencing visual difficulties try the OpenGL renderer, and vice versa. The main difference (feature wise) between them is that the Direct3D code is currently limited to 65K vertices per buffer, so large objects (landscapes etc) may exceed this and some crazy triangles will pop up. This will be fixed in future releases. By default, the VDB will try DX 9.0c first,. then DX 8.1, then OpenGL, in that order.

For mulitple monitors, the VDB will initialize by default on the adapter (monitor) that you last used. So if you start on your primary monitor, and want it on your secondary, simply pull the window on to the secondary and then the next time you run the VDB it will initialize on that secondary monitor (so giving full speed in DirectX etc). If you want to force the VDB to a specific montior (and ignore the last

window placement), use the command line option *-device0* and *-device1* (etc.) command line options. They will use the given number device to initialize on and will run the VDB maximize in the work area of that monitor.

All menu item shortcut keys are customizable through Tools->Customize and the UI keys for movement are customizable through the Edit->User Keys menu. The speed of movement is also customizable through the Edit->Movement Speeds menu. If you find something that is not customizable and it is reasonable to want it so, just let us know via support@havok.com.

3.8.4.1 Viewports and Cameras

The view port area can be configured to contain 1, 2 or 4 viewports and each of these can be used to display a graphical representation of different parts of your scene. The user can manipulate the camera in each viewport in many ways using several navigation modes. These modes can be selected via hot keys (customizable) or through the View menu in the main menu bar. This view menu is also available in the top left hand corner of the current viewport:

The first icon is the Axis icon, and holds the View->Axis submenu. The second icon is an eye and holds the View->Camera submenu. The third icon is the View->Viewports submenu that allows you to configure the number of views. The last normal icon is the View->Render State submenu. The extra icon, shaped like a U, only appears if there are User Cameras present. A User Camera is a camera sent by the game server to the Visual Debugger.

View –> Axis

Games tend to vary a lot in the convention they use for handedness (Left or Right) and also which direction the axis are in relative to the viewer. One axis can go any of 6 directions, one of the others can then go one of four directions (relative to viewer) and then the other is fixed for a given handedness. Luckily the VDB can display all combinations. First choose your handness in the Axis menu (use Left or not). The choose the Up axis as either X,Y or Z. Then choose if the Up axis is the +ve or -ve version of that axis.

You can see the Handedness of the current view in the lower left hand side information that shows, for instance *RH:P:TBall* RH means Right Handed, P means Perspective Mode, and TBall means Trackball camera control.

The other options in the Axis menu toggle the main Unit axis at the origin (Unit Axis), the unit axis at the Point of Interest (often referred to as POI in this doc) and the grid on the plane perpendicular to the Up axis. This grid will either be in units of 1 or 10, depending how close the viewer is too it. It is easy to see which resolution you are in if the POI axis is enabled.

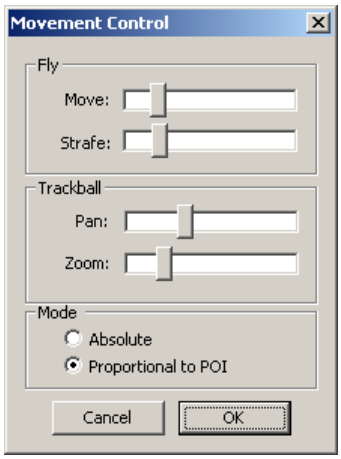
View –> Camera

Camera control is always an issue with an application that tries to cater for both large and small scales as well as users that prefer lots of different modelling packages. We provide two common modes, a virtual trackball (the SGI style) and a fly mode (a Quake style God-mode). The keys for both are for the most part customizable and the speeds at which the operate are customizable as well. It is handy to flick back and forwards between the two modes so choose a handy short cut key for each.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

In track ball mode (indicated by TBall in viewport label) the world can be rotated by left clicking with the mouse button, and panned with the right button, SHIFT-left or middle button will zoom (the Shift key is the 'Secondary' key in the User Keys dialog). The mouse wheel can also be used to zoom. All viewports will be in this navigation mode by default.

Fly mode (indicated by Fly) allows the user to fly through the scene using the WASDQZ keys (customizable of course) and steer using the mouse while holding the left mouse button down. Strafe is the right mouse button and does the same as the ADQZ default fly mode keys (left, right, up, down). Notice that a set of coordinate axes are displayed (of length 1 unit) at the camera's to position. The speed at which the camera flies through the scene is proportional to the distance of the camera to the POI. This can be adjusted by zooming using the wheel on the mouse or through the middle mouse button, or you can change the overall speed of the fly mode in the Edit->Movement Speeds dialog:



The speed sliders range from .1 to 10 times the default speed, and control the fly mode and the trackball. The Mode radio button at the bottom governs whether you want the speed to be absolute (same speed no matter what the distance to the POI is) or proportional to the distance to the POI (default, and handy if you can position your POI correctly).

Camera Object Tracking

There are two options to allow the camera to track the currently selected object local origin or the point of interest (POI) on the selected object. Select is the Space by default, for the object under the mouse pointer, and the POI is taken as being the exact point of the pick. The camera POI will either be set to the tracked object origin or selected POI on that object, and then the camera From position will optionally be kept the same distance away if the relative option is chosen.

Camera Fitting and Roll Back

You will notice the ability to Store the current camera (one stored per viewport) and recall it at a later time (Reset to Stored). The camera you have at the application shutdown is the camera at the start of the next, and also the default stored camera. The Default camera is a trackball mode camera, 10 units along Z, 10 units along Y, with +Y up and Right Handed.

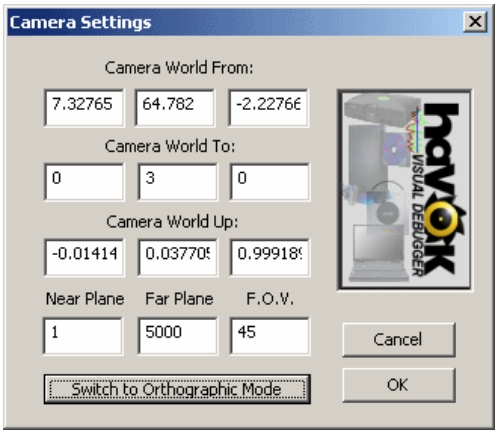
Fitting the camera to the world or current selection uses a simple computation of, given a 45 degree field of view, what distance the camera has to be along the plane perpendicular to the Up axis to see the whole bounding volume. It does not take occluding objects into account so it is easiest to use this fitting in trackball mode and just rotate around, as the POI will be set to the center of the bounding volume

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

and thus you will spin around it. You can choose to reset the POI to any object via the Use Selected POI that will just change the POI to the current selection point, but unlike tracking will not update it from that set point every frame. Thus it is handy to map this to a key near the select key on the keyboard so that you can select and reposition easily.

Exact Camera Settings and Projection Mode

Through the View->Camera->Camera Settings menu you can change the exact vector basis that makes up the camera, so can position the camera exactly where you want it:



The Field of View (FOV) is measured vertically. Notice that you can switch the mode between Orthographic and Perspective projections. With Orthographic mode, even though you specify the camera as a perspective camera (with F.O.V.), the camera internals will compute an orthographic matrix such that the same amount of the world will be visible at the given Point of Interest (To vector) and the given field of view.

Don't be tempted to set the far clip plane to far from the near plane as the accuracy of the Z buffer has to cover the whole space between the two planes, so small near and very large far planes result in a reduced accuracy and you will get artifacts etc.

View –> User Cameras

Viewports can also be set to lock to User Cameras, which are sent from the user's game (see the end of the section entitled The Debug Display for more information on generating user cameras). When recording a movie all cameras (all Viewports and User cameras) get recorded and are available during playback.

The User Camera menu is only populated when there are user cameras present. For instance when the Havok DemoFramework is running it sends its view as a user camera. You can have as many different user cameras as you like so they can be used to follow each AI character or car for instance.

View –> Viewports

This just allows you to have 1, 2 or 4 views on the world. Most options are specific to a view so this can be handy to not only look at the world from a different camera but also from a different rendering style. To resize the proportion of the views hold down right mouse button on the splitter bar of the two views, or the cross over of the two splitter bars in the quad view.

View –> Render State

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The viewports allow for some simple configuration of the render state: Wireframe, Culling, Blending, Textures, ShowLights and Background color. The first five are viewport specific, so you can set them on a preview basis. The last option, the Background color, is across all views (as is the clear color of the render device).

The Wireframe mode is self explanatory, as is the Backface Culling option. Additive Blending toggles the render state between 'modulate' alpha blending and 'additive' blending. With additive blending the alpha component drives the color component towards white when combined, so gives an x-ray like effect. The modulate preserves the object color more, but is affected by rendering order of the objects (and the objects are not sorted in the rendering engine). The Texture option enables a default checker texture that is mapped such that it repeats every 1 unit in world space (so each full black square in the texture is 0.5 x 0.5 units). This texture in conjunction with the grids and POI unit axis can really help in figuring out sizes in the scene. The ShowLights toggles the rendering (and thus picking) of the lights in the scene.

3.8.4.2 Picking and The Current Selection

The default key for picking is Space. It will select the closest icon or object to the camera at that point under the mouse. While the selection key is pressed the VDB will attempt to move that object to the new position. If the object is a light, then it is easy to move it exactly to where you want. If it is a geometric object from a movie then obviously you can't change its position.

When connected to a live simulation the scene can be manipulated with the mouse. Objects the represent Havok rigid bodies can be picked up and dragged by pressing and holding the select key (Space) while pointing at a geometry. Releasing the space bar will release the object.

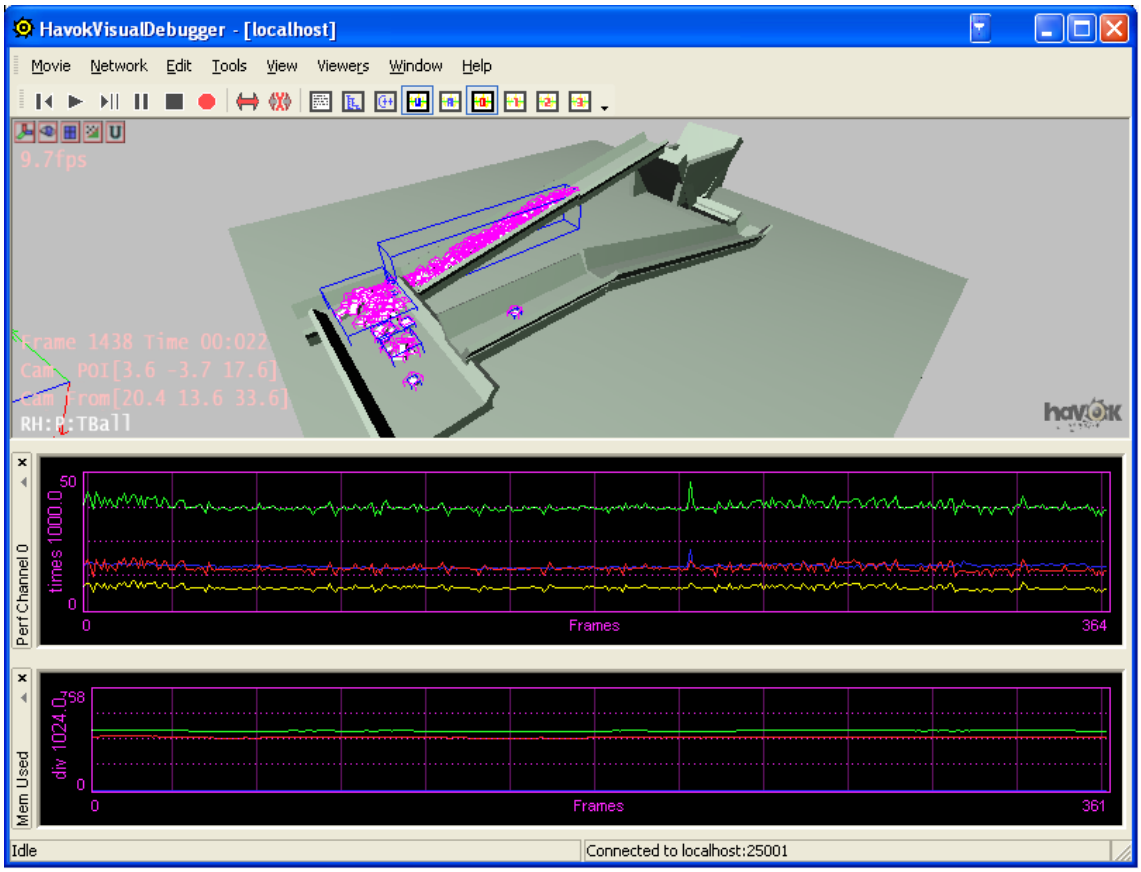
Note that if you double-left-click on a display object in a viewport you will get a summary of display geometry details. This does NOT relate directly to the Havok collision shape data as it has been converted into a graphical form. It is only there as a temporary feature which people may find useful in the absence of more sophisticated meta data. A future release of the visual debugger will improve on this dramatically. The color and alpha value of the display object can also be changed from this dialog box. This color is not persistent in movies or across reconnects.

In that double click object dialog there is an option to save to file. The format you can save the display geometry to is a .ttk file, which is the normal Havok ASCII .tk file, with a transform appended to the end, as a line of 16 floats, which is column major and the same conventions as a hkTransform. To create a hkTransform from it just skip the last row (every 4th elem of the 16 floats) to make a 3x4 array. This is so that you can save all the objects from your runtime (or in the case of Havok from your movie file) and recreate an approximation of the actual scene. In future it is planned that the VDB will be able to query for the actual Havok Serialization packets and save them to XML for instance and allow for exact replication of the problem scene.

3.8.4.3 The Real-time Statistics View

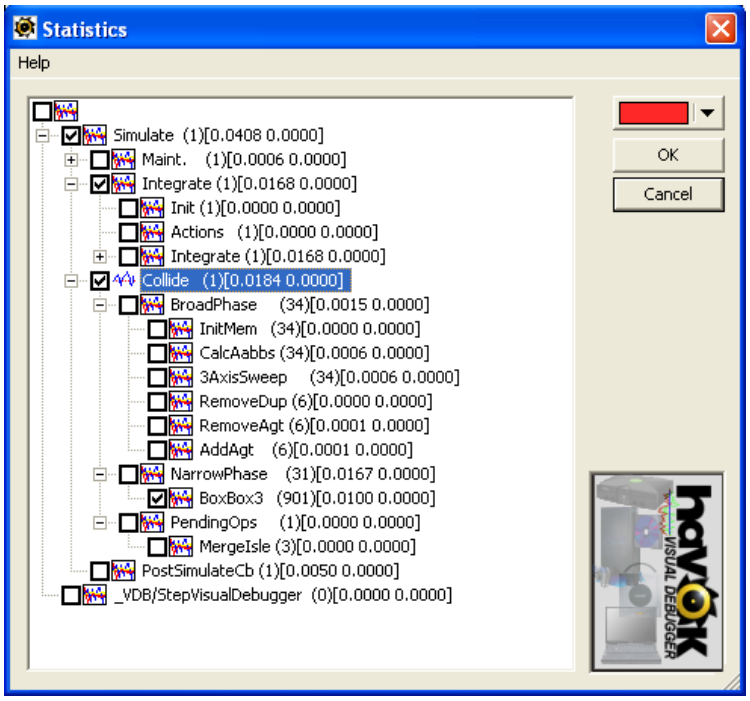
The image below shows the visual debugger displaying real-time performance statistics for a game. This information is transmitted from your game to the visual debugger using the statistics viewer (see hkvisualize, class hkStatisticsProcess). For information on how to setup the statistics viewer on the game side see the section above entitled The Statistics Viewer.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



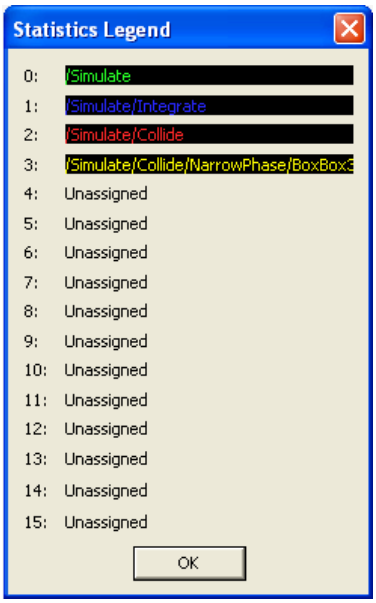
The Real-time Statistics output can be displayed by selecting the **Perf Ch0** option from the **Window** menu. Note that no information will be displayed by this view unless the Statistics viewer has been enabled. To enable this select it from the **Viewers** menu when connected to your game. To display the statistics you are interested in you must use the **Select...** option from the context sensitive window, see the image below. (Simply right-click on the dockable Real-time Statistics panel.) This will bring up a dialog box which displays a hierarchical representation of all the Havok performance timers in your game. Simply check the box beside the statistic of interest and select an appropriate color it.

If you are not getting statistics as you expect there is most likely a problem on the game side, please see the section above entitled The Statistics Viewer for help.



Initially the statistics are assumed to be in milliseconds so depending on the typical values of your statistics you may need to adjust the range of the graph by using the **Options...** option on the context sensitive menu.

A legend of the currently selected statistics and their corresponding color can be displayed with the **Legend...** option, see the image below.



Although there are 6 destination stat channels, not all may be updated each frame, the meaning of which depends on what channels you target with the FrameInfo in hkStatisticsProcess::step() and what setup

you have for the perf counter. You can alter this and recompile the viewer to suit what timings you want. On Xbox360 or a multithreaded PC the 6 timers represent timer 0 from each of the potential 6 threads.

Stats Dumps and the Perf Stat Summary

The bar graphs are all well and good to get a general runtime view of what is going on in the sim, but somethimes you may want to dig a bit deeper. One option is to bring up the Perf Stats Summary text window (the T icon in the tool bar). When you have the stats viewer enabled, it will show a navigatable stat tree, as produced by the hkBase lib (and as seen in the Havok demos). Give the window the keyboard focus by clicking on it, and then use the arrow keys to navigate the stats (Right Arrow to open a node, Left to close etc).

To save all the current stats for a frame or indeed a series of frames to file, right click on any of the Perf Stat Bars (graphs) and select Save. You can choose between three types to save to:

- TXT: A text dump, similar to the Perf Stat Summary, but can handle multiple frames and is a full dump
- TGA: A targa image file dump. A bar graph of important section timings, commonly used with multithreaded apps to find stalls and load imballance.
- HKS: A binary dump of all the current stats source information. This file can be read by Havok Dev Rel and they can produce from it all the different stats views available.

When you send a movie (See Movie section below) to Havok Dev Rel, they can extract this timer information too, as long as you enabled the Statistics Viewer while you where recording of course.

3.8.4.4 Advanced Profiling Analysis

From Havok 4.6 onward the visual debugger has been extended to provide real time profiling on a per thread / processor basis. This allows you and us to easily identify multi threading performance bottlenecks. When you save a VDB movies all timer information is also saved. This means that detailed performance analysis can be done either online or offline.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

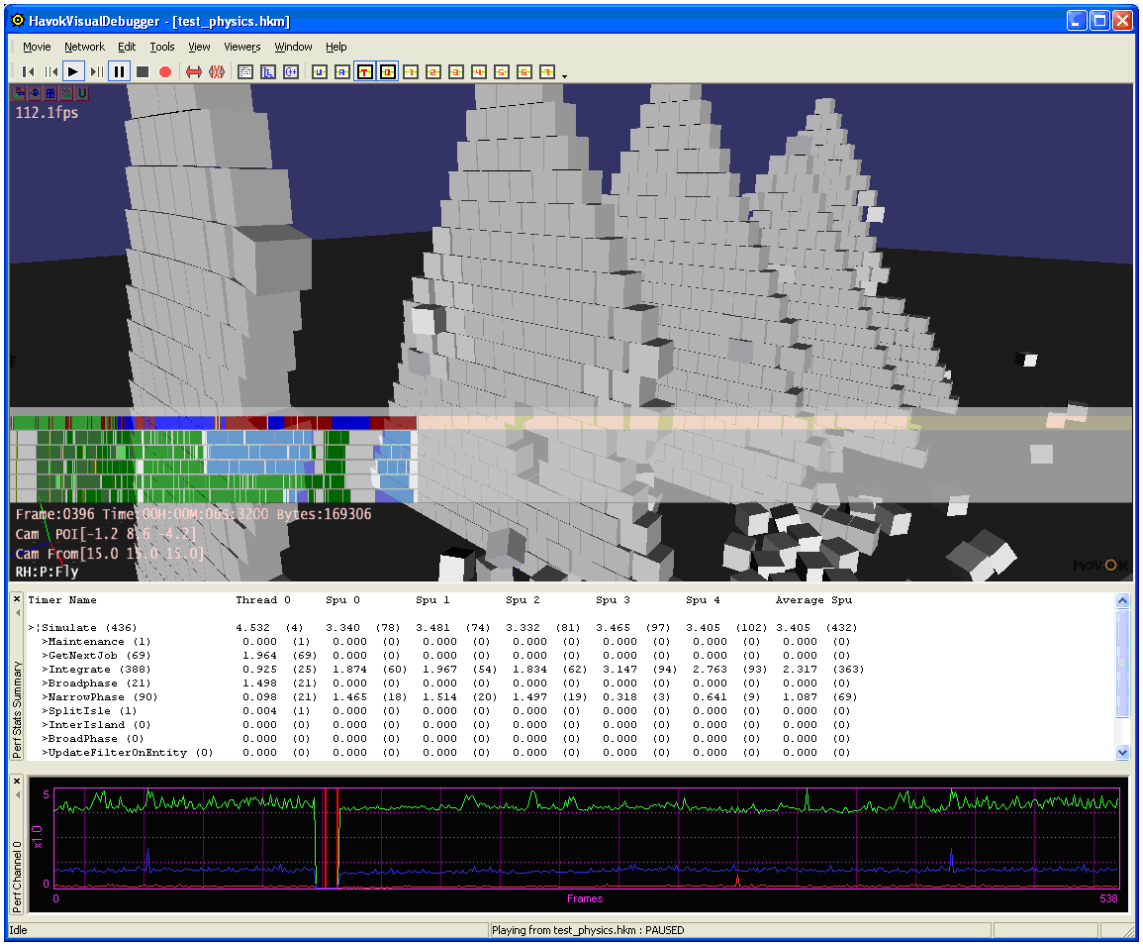


Figure 3.66: Advanced Profiling with the visual debugger

The diagram above shows both the per thread statistics overlay and the detailed performance statistics summary.

Per thread statistics overlay

The per thread statistics overlay give a real time view of the jobs running on Havok's Job queue. The job for each thread are shown horizontally in chronological order. Separate threads are stacked on top of each other. To display this view Choose View->Render State->Stats Graph Overlay from the menu.

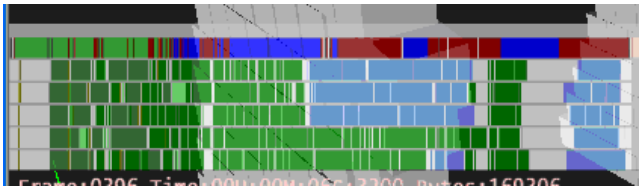


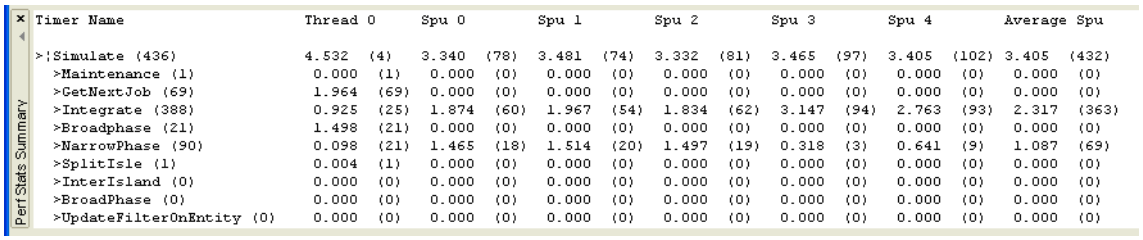
Figure 3.67: Detailed per thread stats overview

In the example above we see six running threads. Individual jobs are shown in different colours. By moving the mouse over any of the coloured block it will indicate what that job type is and how long it took. Selecting a region with the left mouse button will allow you to zoom in on a specific region. Double clicking allows you to zoom back out. In the example above we can see the top thread (in this case the

PPU thread running on Cell) is handling constraint setup jobs (light green) and broad phase jobs (dark blue). These jobs create work for the other threads to do (constraint solving in green and narrow phase collision detection in light blue) and then waits for this work to complete (shown in red).

Detailed performance statistics summary

The detailed performance statistics summary view can be called up by clicking on the 'Toggle Perf Stats Text' Icon (a Red T) in the top toolbar. This launches the following panel



Timer Name	Thread 0	Spu 0	Spu 1	Spu 2	Spu 3	Spu 4	Average Spu
>¦Simulate (436)	4.532 (4)	3.340 (78)	3.481 (74)	3.332 (81)	3.465 (97)	3.405 (102)	3.405 (432)
>Maintenance (1)	0.000 (1)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)
>GetNextJob (69)	1.964 (69)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)
>Integrate (388)	0.925 (25)	1.874 (60)	1.967 (54)	1.834 (62)	3.147 (94)	2.763 (93)	2.317 (363)
>Broadphase (21)	1.498 (21)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)
>NarrowPhase (90)	0.098 (21)	1.465 (18)	1.514 (20)	1.497 (19)	0.318 (3)	0.641 (9)	1.087 (69)
>SplitIsle (1)	0.004 (1)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)
>InterIsland (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)
>BroadPhase (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)
>UpdateFilterOnEntity (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)	0.000 (0)

Figure 3.68: Performance statistics summary

This is an interactive window showing total times for each thread. The timers are presented hierarchically. Any timer preceded with a '>' symbol can be expanded in more detail. Use the up and down arrow keys (when that window has focus) to navigate up and down through the list and use left and right arrow keys to expand and contract timers. For each timer two values are shown these are the total time in milliseconds and the number of calls made to th timer. The number of calls is enclosed in parenthesis.

3.8.4.5 The Real-time Memory View

The WorldMemoryViewer, when enabled, asks all hkWorlds to calculate the amount of memory they are using. This memory data is encoded in a large stream (not unsimilar to the hkMonitorStream) and sent across the network. The data has two values per object, amount used and amount actually allocated. There are thus two dockable windows that show these two values. They are based on exactly the same graph view as the real time performance stats, so the usage is the same. Through the context sensitive pop up menu (right click), you can change all the nodes you want to track through **Select...** and change the display settings through **Options...**

Note that in order to compute the memory stats all objects in the scene must be traversed and have a function called on them. It is thus a slow viewer to enable, but can give important insights in your debugging sessions, especially when used in conjunction with the normal perf stats.

3.8.4.6 The Console View

This displays local and remote textual information. To send text from your game use the HK_DISPLAY_TEXT macro described in the section entitled The Debug Display above. This text will appear in the Remote tab. The current limit on the console text is 1MB, and it will stop taking in text after that. A proper FIFO string management will be implemented in a future release.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.8.5 Visual Debugger Menu Options

This section describes the visual debugger menu options:

3.8.5.1 Movies

This menu includes all the visual debugger's movie options, along with the **Exit** command.

Play...

Plays a movie file. The default file extension for a Havok movie is .hkm. Havok movie files are created using **Movie - Record...** followed by **Movie - Stop** while you are connected to your game. It is also possible to record a movie by calling hkVisualDebugger::capture(const char* movieFilename) on the visual debugger game side. (See the bottom of hkDefaultGame.cpp or the visual debugger game side documentation for an example of how to do this.) If pause is selected, play will just toggle the pause to off.

If the movie is currently Paused, Play will just unpause (same as hitting Pause again). If the movie is playing and you hit Play again it will assume you want to play a new movie.

Single Step

Puts movie playback into single stepping mode. This option appears on the Standard toolbar by default and repeated clicking causes the movie to advance one step at a time.

Pause

Pauses the current movie.

Rewind

Goes back to the beginning of the current movie. There is no current support for rewinding just one frame as the stream of data from the movie is difference based and thus the VDB needs to store a lot more state information if it is to be able to rewind to previous states (such as re-creating a body that has just disappeared in the current frame from which we are rewinding etc.).

Stop

Stops playback or recording of the current movie.

Record...

Records a new movie to a file. The default file extension is .hkm, and it is an uncompressed form.

Real Time Playback.

When you step the VDB on the game (server) side, you can pass a time value. This is the 'time' displayed when you show the Frame Info in the viewports. If this time corresponds to the current time when the frame was captured (it may not), then you will probably want the movie to respect that time and playback at that speed. Respecting the frame time is the Real Time Playback option (on by default). If you find your movies stop for no reason at a certain frame, have a look at the Frame Time in the viewport. If it

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

jumps a lot the VDB waits until that amount of time has passed, thus appearing to stop whereas in fact it thinks the movie is just very slow.

Thus make sure that the time you give to the VDB step is insulated from the effects of breakpoints and other stoppages that may occur in your game for lengthy periods of time. Either that or don't use the real time playback option.

Exit

Quits the application.

3.8.5.2 Network

This menu allows you to connect and disconnect from the network. The **Network - Connect...** option displays a dialog box in which you can specify the IP address of the game side - the machine the simulation is running on - along with the port number. Use the **Network - Disconnect** to terminate the network connection. The default port number is 25001. **Network - Auto Reconnect** causes the application to automatically reconnect to the game if the connection is closed by the game. Exponential back off is used to avoid redundant network activity.

If you shut down the application with Auto Reconnect still on, the VDB will start to Auto Reconnect as soon as it starts up again. This can help speed up your workflow during debugging.

3.8.5.3 Edit & Tools Menus

Here you can edit the User Keys and the speed of Movement. Both of the settings in these dialogs are stored in the registry across sessions.

Tools->Customize allows the user to customize the key bindings and toolbars in the application. All settings are remembered, so please take the time to get all the keys and toolbars the way you like them.

3.8.5.4 Viewers

This menu becomes active when there is a valid network connection to a simulation. It displays a list of all viewers available on the game side. Selecting a viewer from this menu toggles the state of the viewer on and off by causing them to be created and destroyed on the game side.

There are several viewers available. In general viewers enabled by default (game side via hkpPhysicsContext::registerAllPhysicsProccess()) do not modify the simulation. They are passive and only transmit simulation state information to the VDB. There is however one exception to this rule. The Forced Contact Point viewer *DOES* modify the underlying physics simulation when it is enabled. Internally this viewer modifies the hkpEntity::m_processContactCallbackDelay member of all objects in the simulation. This can hinder performance if you have already specifically set this member to reduce the number of callbacks fired. For more information on the implications of setting hkpEntity::m_processContactCallbackDelay to 0 see Section 4.2.9.1 Callbacks and Events.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.8.5.5 Window

Creates and destroys the various dockable windows available.

3.8.5.6 Help

About The Visual Debugger...

Displays a small dialog box outlining what the Visual Debugger does it also contains protocol version information.

Help...

Links to the Havok documentation. The link is a relative path so it won't work if you move the application from it's original location. It just links to the Havok Physics chm in the doc directory.

3.8.6 Preparing a visual debugger movie for a support query

A Havok movie can be created by capturing the simulation directly to a file from your simulation or by using the **Movie - Record...** / **Movie - Stop** menu options in the application. To capture a file directly from a simulation see the section above entitled *The Visual Debugger Game Side* and take a look in hkDefaultGame.cpp for an example. Be sure that the appropriate listeners are created when capturing your movie, for example, if you want support to examine the performance statistics of your simulation the Statistics viewer must be instantiated/created when capturing your movie otherwise this information will not be streamed to the file. If you will be transmitting the file over a network it would be worth while zipping it as a compression ratio of 80% is typically achieved.

3.8.7 Platform-specific issues

3.8.7.1 Application Caveats

The visual debugger application requires DirectX 8.1 or later as it uses direct input.

3.8.7.2 Server Caveats

- Ray-casts are currently not displayed automatically and must be displayed using the debug display. See the section entitled The Debug Display above.
- Currently you must call hkpWorld->stepDeltaTime(...) and hkVisualDebugger->step(...) for the visual debugger to work properly. The world step is required for physics viewers that rely on data comming only from physics callbacks. The main VDB step first checks for new clients connecting and then sends its current frame of information, waiting for an acknowledgment back to continue.

3.9 Miscellaneous

3.9.1 Performance Tips and Common Errors

3.9.1.1 Introduction

This section provides you with a few useful hints, guidelines, and "gotchas" for working with the Havok Physics SDK. They should help you to write more efficient Havok code and avoid some common errors.

This is not intended as an exhaustive guide to using the Havok Physics SDK. If you are unfamiliar with any of the terminology used, or want to find out more about any of the topics, please see the relevant Physics chapter(s).

Don't forget to visit the Havok support site http://support.havok.com for the latest Knowledge Base items, downloads and user information.

3.9.1.2 Collision Detection

Broadphase issues

Because the broadphase world has a definite size, it's possible to position objects so that they are *outside* the broadphase. If this happens, the broadphase considers any Aabbs outside its limits that share a common axis to be interpenetrating. This can incur a significant performance cost, as the relevant objects will all be passed to the narrowphase.

In order to avoid this, you need to both ensure that your broadphase size is appropriate for your scene, and remember the broadphase limits when moving your objects.

You can use the hkpBroadPhaseBorder class, provided in the hkutilities module, to monitor objects leaving the broadphase. This is not used by default - you need to create an instance of it yourself. hkpBroadPhaseBorder has a `maxPositionExceededCallback()` function that's called whenever an object leaves the broadphase world. By default, it removes the entity from the world - you may implement your own version.

Configuring box shapes

Don't forget that you specify the size for a box shape using its *half-extent*. In other words, a box with the half-extent (10,10,10) has the dimensions 20 * 20 * 20.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

When to use transform shapes

hkTransformShapes incur a greater overhead than simple shapes with no transform. Because of this, you should avoid using transform shapes in situations where a simple shape would do. Remember that the transform in these shapes is from the child shape's space to the parent shape's local space, not from the shape to the world! Transform shapes are usually necessary only when constructing compound shapes, as they allow you to position child shapes correctly relative to each other.

Phantom performance

Phantom objects - hkPhantoms - that are used directly with the broadphase are much faster than hkPhantomCallbackShapes with a bounding volume.

Convex shape radius

The hkpConvexShape class has a radius member that allows you to add an extra "shell" to any convex shape, such as a convexVerticesShape or box. By default, the radius value is 0.05. For spheres and capsules this extra radius is the actual radius of the shapes themselves, and is not thought of as an extra shell, although it is treated by the collision detector in exactly the same way. (The sphere is just a "point", and the extra "shell" is the radius of the sphere).

This is used as the shape's surface for collision detection purposes. This can improve performance, as calculations for shapes that are actually interpenetrating are much more time-consuming than for interpenetrating shells.

If a box with a non-zero shell comes to rest on a table, there will be a visible gap of the radius (added to any radius the table has) between the box and the table. If interpenetration occurs, the system will work to restore the gap, and the box may still not have appeared to go into the table.

Note that if two convex shapes collide and both have zero radius, this can also introduce numerical instability. This is because the surfaces used for collision detection will be exactly touching when the objects come to rest, making the calculation of the collision normal unstable. This is another reason why convex objects should all have a positive radius value.

Registering collision agents

You must register collision agents explicitly if you want to use them - otherwise no narrowphase collision detection will take place! You can either register individual agents or use the `registerAllAgents()` utility function.

When to add collision filters

If you create a rigid body and add it to the world using `addEntity()`, it is immediately considered "active" by the collision detection system, which will start checking for overlaps and creating collision agents for it if it is potentially colliding with other objects. If you then add a collision filter on the next line, and turn off collisions for the object (perhaps on a pairwise basis with an hkpPairwiseCollisionFilter), it will only affect the body if its collision agents have not already been created - in other words, if the object is not potentially colliding with anything. Otherwise, collisions will remain enabled for the body. To avoid this happening, you should add the filter *before* you add your bodies to the world.

This is especially important when the objects you add are constrained together, as they may never leave the collision list and so the filter will never take effect.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.9.1.3 Constraints

Prismatic constraint pivot point

The pivot point of the prismatic constraint should be on/in the attached body and not the reference body. A pivot point in the reference body may produce strange/spongy behavior.

Prismatic constraint linear limits

If a prismatic constraint between two dynamic bodies has a linear limit, when the attached body hits this limit, large angular effects may occur and cause the constraint to become unstable. If this occurs, make sure the pivot point is specified inside the attached body, and also increase the mass of the reference body to many times that of the attached body. Note that normally it is desirable for constrained objects to have similar mass. Increasing the damping values for the bodies may also help if this occurs.

Where to use ragdoll constraints

Ragdoll constraints are more expensive to simulate than, for instance, hinges, and should be used only where the special angular limits offered by the constraint are necessary. You can often simulate knee and elbow joints adequately using simple hinges, saving the ragdoll constraints for joints with more complex movement such as shoulders and hips.

3.9.1.4 Base library

Memory system initialization

Note that the memory system is not initialized until after `main()` starts and `hkBaseSystem::init()` is called. This means that statically initialized classes that need to allocate memory will fatally fail. In practice, this means that most non-trivial Havok objects cannot be global or static.

3.9.1.5 General debugging

MSDev debug expansions

If you are using Microsoft Visual Studio with Havok, you can edit its `autoexp.dat` file to make Havok objects display in a more useful way in the Visual Studio debug window. For example, you can specify that you want a hkVector4's x, y, z, and w components to display on the top line, avoiding the need to click through several expansions.

Adding the following to the file will affect the display of hkVector4s, hkQuaternions, hkBaseObjects, hkArrays, and hkStrings.

```
hkVector3=<m_quad.m128_f32[0],g>, <m_quad.m128_f32[1],g>, <m_quad.m128_f32[2],g>,
            <m_quad.m128_f32[3],g>
hkQuaternion=<m_vec.m_quad.m128_f32[0],g>, <m_vec.m_quad.m128_f32[1],g>,
    <m_vec.m_quad.m128_f32[2],g>, (<m_vec.m_quad.m128_f32[3],g>)
hkBaseObject=<,t>
hkArray<*>=<,t> <m_size> / <m_capacityAndDeallocateFlag&0x7fffffff>
hkString=<,t> <m_string,s>
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.10 Physics Articles

This section contains in-depth discussions of general issues relevant to using Havok Physics in your games. The Physics Primer section discusses general physics concepts, while the section entitled 'Rotations, Handedness and all that' discusses and mathematical conventions and representations.

3.10.1 Physics Primer

3.10.1.1 Introduction

Welcome to the Havok physics primer. This document gives a broad overview of physical simulation, along with details about how Havok technology works. It aims to provide a general understanding of the terminology, methodology and behavior associated with a physics engine. With a reasonably good idea of how a physics simulation should behave, getting Havok to work within your game should be much easier.

3.10.1.2 Physical simulation

Physical simulation is not a new concept. Ever since computers started dropping off the assembly line, scientists and programmers have used them to simulate complex situations like rocket trajectories (ballistic motion), liquid flows (fluid dynamics) and other complicated projects. Many of these simulations are very expensive in terms of the CPU resources required. They are concerned with high levels of accuracy and as a result, calculations need to be performed with a very high level of detail. Havok is designed to solve a different problem - game physics; fast, believable simulation executed in a very limited number of CPU cycles.

Rigid body dynamics is the most common form of physical simulation used within video games. For performance reasons, a number of assumptions are usually made about the environment.

- The shape of each object is fixed. Each object is infinitely hard/rigid.

  There are many advantages if the shape of objects can be assumed not to change, irrespective of how large the forces applied to them are. Huge speedups can be achieved as lots of information about how objects will move can be precalculated.

  In real life, a car panel colliding with a wall will bend (and possibly be torn off) depending on the strength of the impact. In a rigid body simulation, the car panel will remain completely intact.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Figure 3.69: Rigid Objects

However game objects still need to show some damage so Havok helps out by signaling events corresponding to the collision between the car and the wall. Information about the impact can be queried (e.g. the position and the force of the impact) and fed back into the game engine. This could be used to dent the graphical representation of the car panel or even replace the physical shape of the panel to a pre-dented one if required.

- The landscape does not change dynamically.

  Simulating a completely dynamic environment (e.g. blowing holes in the ground with a grenade) can be very computationally expensive. In most games the base level of environment remains fixed and objects within the landscape (such as traffic cones, pedestrians, bins etc.) can be modeled as separate physical objects rather than as part of the landscape.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

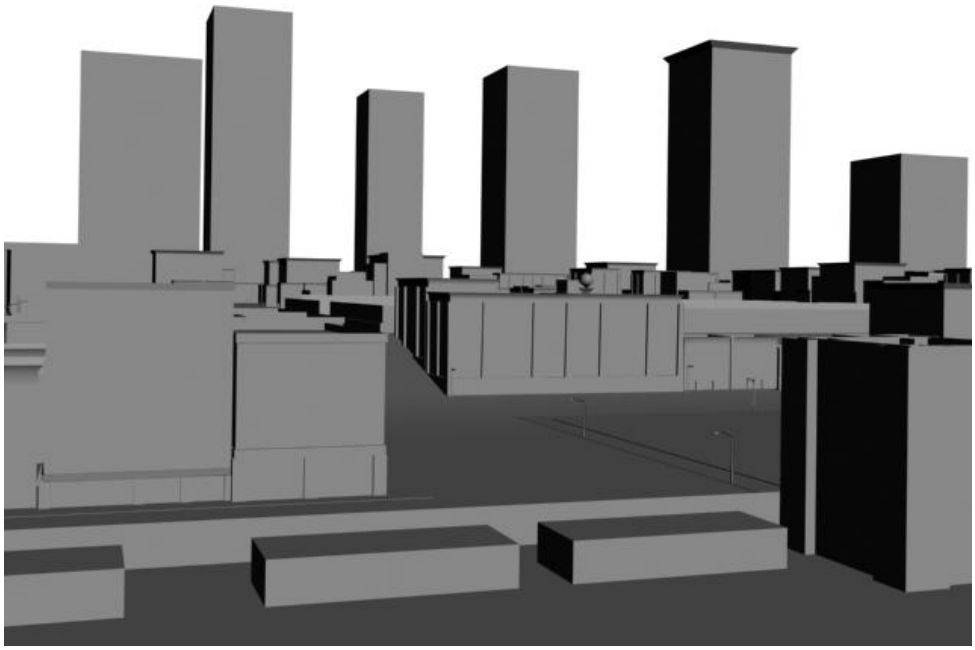


Figure 3.70: Base level Landscapes are generally solid

Sections of the game environment that are required to be dynamic (such as water, collapsible walls, destructible land etc.) can be layered on top of the base Havok physics.

- The shape of objects can be simplified for physical simulation.

  A replacement (or stand in) object can be used within the physical simulation to speed up the various calculations. While graphically a bus in a game may be a complex combination of rubber, glass, chrome and tin - to the physics engine it may just be a moving box! Havok has a simple set of primitive shapes like bounding boxes and spheres, and allows easy creation of custom shapes. Such a shape, can be used as replacement for a highly detailed in-game polygon mesh. Havok also contains functions to automatically extract simplified proxy meshes from arbitrary in-game geometry.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Figure 3.71: What the player sees is not always what the physics engine simulates

- Objects in a scene are not all active at the same time.

  Simulating something at rest is very efficient (zero CPU time) so huge worlds with many objects may be crated so long as only a reasonable number of them move at any one time. Within a typical game scene, there may be many potentially active environmental objects such as chairs, crates, barrels etc. Most of these remain at rest for most of the time, and so do not need to consume any CPU time.

  Havok automatically handles detecting when objects are at rest, and removes them from the active simulation process. It also automatically detects when they become active again and adds them back into the simulation. Complete control is given to the developer to allow them to override the default behavior and activate and deactivate physical objects (e.g. may never want a particular NPC to deactivate).

3.10.1.3 Realism vs. Believability

Havok's design aim is to provide simulation that gives the game-player a consistent environment to explore. The environment does not need to follow "real world" rules so players can drive faster, jump farther or bounce harder than normal, if the game designers see fit.

To be useful, simulation needs to be lightweight on the current range of consoles. To do this, many low level optimizations have been integrated into the core architecture of the Havok system. Havok Physics also allows developers to adjust the level of trade-off between accuracy and speed.

Game developers have always used pre-determined animations for some game scenarios. As consoles have become more powerful, players have started to expect a level of realism within games that can only be achieved by using more fully featured simulation.

A common, if simplified, example of a pre-animated sequence versus a physically simulated sequence

is driving or throwing an object through a stack of objects (e.g. a brick wall). An artist can animate one sequence showing the wall breaking down from the middle, or can spend a long time animating the wall being broken from several different pre-defined points. This process is time consuming and it still becomes clear to the player that hitting the wall in a slightly different area than the pre-defined points results in an unrealistic and unbelievable outcome.

Using the Havok physics engine, the stacked wall of bricks will react in a realistic manner and can result in a huge variety of different ways that the player can destroy the wall. When the wall is breached, some of the bricks that have fallen from the wall can be carried by the player to another part of the environment and perhaps used as part of a puzzle (e.g. to throw at an enemy, or to place on one side of a seesaw to increase the resulting weight).

In a game, if a character falls in exactly the same way each time they are killed, the belief that we are playing in a real environment is compromised. With the Havok physics engine, we get this expected chaotic behavior by default.

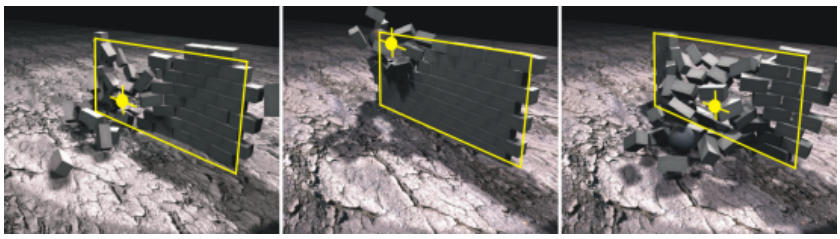


Figure 3.72: Chaotic behavior of a smashing wall impacted at different locations

3.10.1.4 World Scale

There is a lot of literature on physics and simulation. However not all of it is needed for game simulation! Havok deals with Newtonian mechanics, i.e. the high school laws of motion that describe the behavior of objects under the influence of other objects and external forces. These laws break down at very small (i.e. subatomic) and very large (i.e. galactic) scales but its unlikely a game will stray into these areas!

Havok is designed to work at the scale of objects that we interact with in the real world (cars, people, buildings etc). By default, the Havok engine works in meters and kilograms. If your existing game engine uses an arbitrary scale internally, a common method should be used to scale (up or down) the values the game engine passes to Havok at runtime (e.g. by using custom macros). This ensures that the physics simulation is processed using values within the preferred range while causing minimal change to the existing game code.

Be very careful with scale, particularly when using 3D modelers. Often modelers have their own mechanisms for displaying units in dialog boxes (e.g. 3ds max allows specification of the units used and automatically converts all values displayed to those units however internally it always works in inches, including when exporting geometry).

It is possible to use Havok with a scale that is not meters, however it is not recommended. If you are using feet as units for example, please see the scale section in the Dynamics chapter on how to adjust some of the internal Havok tolerances appropriately. It is not recommended that you work in a scale greatly different from meters, i.e. do not use centremeters or kilometers as your units in Havok.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.10.1.5 What Does a Physics Engine do?

A physics engine knows and cares little about how the objects it simulates are displayed graphically in a game. It simulates the motion and interaction of objects based on a physical (not graphical) description of the objects, but this information may be used to generate a display that "tracks" the simulation.

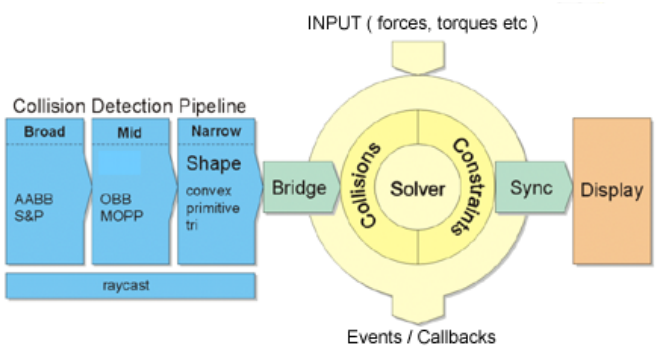


Figure 3.73: The structure of a physics simulation system

A physics engine like Havok has three basic tasks to perform. Having setup the initial conditions for a given scene, begin the main simulation loop which steps through some task, updates the graphics and then repeats.

1. Determine which objects are overlapping or about to overlap (e.g. a spaceship has collided with a hangar wall, or with another spaceship). This stage is divided into three sections: broadphase, midphase and narrowphase.
2. Solve all forces acting on the objects. Some of these forces might occur as a result of collision detection (e.g. two spaceships colliding together) or as a result of input from outside the simulation (e.g. the gravitational pull of the environment, the driving force of the ships engines or when the pilot presses the accelerate button, an additional force is calculated and applied to the physical representation of the spaceship etc.).
3. Having gathered up all the forces, the simulation is advanced by the time step size (say after a time step of $1/60^{th}$ of a second) and the new state of the objects (position, orientation, velocity, acceleration etc.) is calculated for this time in the future. This information is then used to update the corresponding display representations of the objects

   Do all other game logic and repeat....

It is assumed that at each step in the simulation we want to update the display. Later we will see why that might not be the case.

3.10.1.6 Simulating a cannon ball

Forget about collisions for now, and consider only the simulation of a cannon ball immediately after it has been fired from the cannon. We know the ball's position, speed and acceleration. We know its weight and we assume we know the state of the environment (i.e. air resistance, wind force, gravity). Armed with this knowledge we can start to make predictions using Havok.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

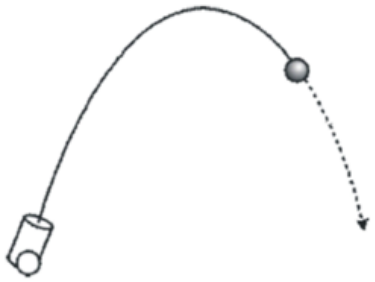


Figure 3.74: The expected ballistic motion of a cannon ball is an arc
This image illustrates what we would like to achieve. Over a period of time the cannon ball's rate of ascent should slow due to gravity, and it should eventually fall to the ground having traversed a classic parabolic arc (assuming no air resistance).

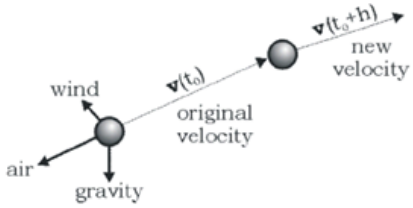


Figure 3.75: Given an initial state of an object, and knowing the forces acting on it, we can estimate the new state of a body in motion over time

At a given point in time we examine the state of the ball (its speed v and acceleration a) and knowing the external forces acting on it we make a prediction as to its change in position after a period of time has elapsed (call this period h seconds), as shown in the figure above. This prediction is a combination of a number of assumptions:

1. Newton's laws of motion govern the motion of the ball.
2. In the time period h, all external forces acting on the ball are constant (so air resistance and wind and gravity do not change during this time).
3. The math used to calculate the new position is accurate (see below).

In general 1) is usually a good assumption but 2) and 3) cause problems and are closely linked to the time period h over which the calculations are performed.

Imagine planning how to get from your home to your office. The factors effecting your trip are never quite the same and there is no simple equation that can guide you through traffic. You can get more accurate in your plan if you break the trip down into smaller segments and replan after each step.

This is also true of simulation math. As the simulation becomes more complex the math required to calculate the new positions and velocities of objects in a simulation also becomes more complex. As a result, the guesses produced by the math give less accurate results when using larger time steps.

The smaller the time step taken, the more accurate the result at the end of the time step so to step forward in time by a large time step h it is better to split this into n consecutive steps of a smaller time interval h/n.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

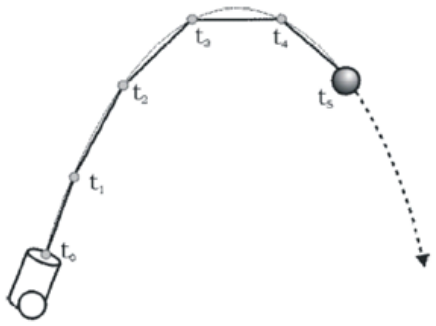


Figure 3.76: Evolution of a simulated cannonball captured in a series of "snap shots"

Iterations

With a physical simulation each result depends completely on the previous simulation step. If, in one step, a very inaccurate result is produced, then the next step is likely to be even more inaccurate, resulting in a spiral of decreasing accuracy until eventually the results are garbage (the simulation "becomes unstable" or simply "explodes"). Unfortunately you cannot simply stop mid step, reduce the number of objects or ignore collisions to compensate for a reduction in the available CPU resources you simply have to ensure that each scene can be simulated stably even with the lowest expected CPU bandwidth.

Assume we need to update the geometry in the game so it can be displayed once every $1/60^{th}$ of a second. For simulation accuracy and stability, we might still need to do more simulation steps every second. To achieve this, the Havok engine allows you to specify the number of "mini-steps" to take. In Havok these "mini-steps" are called iterations.

An iteration is an internal physics simulation step that does not interact with the 3D display. The iteration parameter specifies the number of steps the physics engine takes *before updating the 3D display.* This gives control over the granularity of the physics simulation independent of the display update frequency.

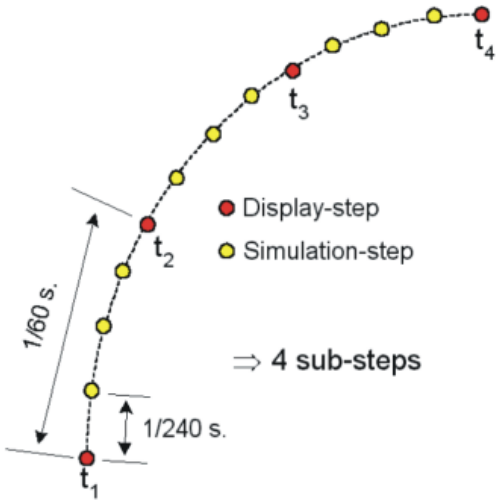


Figure 3.77: Iterations allow simulation frequency to be decoupled from display frequency

In the diagram above, the display is updated at $1/60^{th}$ of a second, but the physics simulation will be stepped at intervals of $1/240^{th}$ of a second. For every 4 iterations the display is updated only once. By

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

setting the number of iterations the accuracy of the physical simulation is controlled independent of the display.

Energy Management

One of the major factors determining the CPU load in a physical simulation is the number of objects that are active or moving i.e. being physically simulated. In a typical scene a large number of objects do not actually move at all and (in theory) can be ignored until interacted with. Temporarily removing stationary objects from the simulation remains the best single way to reduce the CPU load. Energy management is concerned with identifying objects in a scene that are not doing very much and removing these from the physical simulation (known as *deactivating* or turning off the object) until such time as they begin to move again. The important elements of energy management are:

- When should an object be deactivated?
- When should an object be reactivated?

Both these questions are difficult to answer precisely as the correct answer is dependent on the game context. Usually objects are deactivated when they have not moved much recently and are reactivated when hit by other moving objects. Havok does a good job of taking care of this automatically and should give near optimal results in most cases. If energy management for a particular game scenario needs tweaking, Havok exposes all the API controls necessary.

3.10.1.7 Linear Physical Units and Values

We already dealt with the issue of scale and switching between units of measurement. Physics engines are not only concerned with lengths and weights however. There are a large number of different measures in use at any given time. Despite the apparent complexity of dealing with all these units, in most cases you need not worry about them. It can be useful however, to know that you can analyze a scene and the units used to determine the approximate values required to achieve a desired goal.

In general the metric system is used in the following discussions. The metric or SI units are well behaved for our purposes (a pity as the Imperial system has great units like the Slug!)...

- m = meters
- kg = kilograms
- s = seconds
- N = Newtons (measurement of force: 1 N = force required to change the speed of a 1kg object by 1 m/s in 1 s. This can be difficult to grasp but think of it using an example of a 1kg weight sliding on 'perfect' ice. Imagine this weight slides at a speed of 1m/s (3.6 kph). If you skate along behind it and push it so that it speeds up to 2 m/s then the force you applied is 1 Newton!).
- rad = radians. 1 rad = 57.2958 degrees = (180 / 3.14159).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Name	Units, Metric	Unit, Imperial	Description
Position	m	ft	A better description of this term is displacement or distance
Velocity	m/s	ft/s	Speed - how fast an object is traveling relative to some frame of reference
Mass	kg	Slug	A measure of the amount of matter in a body. How difficult it is to move a body.
Momentum	kg m/s	lb/ft^2	Velocity multiplied by mass: this property determines the amount of force required to change the velocity of an object. For example a truck is harder to stop than a scooter even if both are traveling at the same speed, because the truck has greater mass.
Acceleration	m/s^2	ft/s^2	The rate of change of velocity over time (i.e. whether an object is speeding up or slowing down)
Impulse	kg m/s (or Ns)	lb s	A measure of the change in momentum (usually measured in Newton seconds). To instantaneously change the velocity of an object, apply an impulse to it.
Force	kg m/s^2 (or Newton N)	lb	The basic unit of a physics engine. This quantity measures the effort required to change the velocity of an object (i.e. to give an object an acceleration - either positive or negative).

Table 3.8: Units and terms

- x/y is read as "x per y" [for example velocity is measured in m/s or meters per second. Note that x/y is sometimes written as x y^{-1}].
- x/y^2 is read as "x per y squared" or more usually "x per y per y" [for example acceleration is measured in m/s^2 i.e. meters per second squared or meters per second per second]

Note : Sometimes, to distinguish from their angular counterparts (discussed later), velocity, momentum and acceleration are defined as *linear velocity*, *linear momentum* and *linear acceleration*.

3.10.1.8 Angular Physical Units and Values

Orientations are often one of the most difficult quantities to become familiar with. Most people (anyone who doesn't spend their entire day working on the space program or writing games!) find it difficult to get a good intuitive grasp of 3D orientation. Orientation can be specified in many ways, some of the most popular in 3D graphics being a transformation matrix or an axis and angle (a "standard" more familiar to pilots and astronauts might be pitch-yaw-roll. This is good for flying but not as useful for mathematical operations).

One important thing to remember is that orientations require a reference frame. Rotations are 'from' somewhere 'to' somewhere else so you always need something fixed to calculate where you are rotating 'from'.

Coordinate Systems & Reference Frames

Think about the location of the tip of your nose. You probably work out where the tip of your nose is in terms of something else. One answer could be "a couple of inches forward, an inch or two down" (i.e. relative to the point between your eyes!). This would mean that, your nose points forward, the top of your head is up and your right ear is to the right. This is like a subconscious coordinate system (also called a reference frame) to simplify talking about position.

A typical 3D coordinate system has three independent directions or *vectors* usually termed x, y and z and an origin (which is the [0,0,0] position). The vectors are chosen to be at right angles to each other and all positions and directions are specified with respect to these directions and the origin.

When you create geometry in a modeler (3ds max, Maya, SoftImage, LW etc.) it is created with reference

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

to the modeler's built-in 3D coordinate system. All vertices are specified in terms of their distance from the origin along the x,y and z axes. Objects may or may not be centered at the origin depending on how they are created.

Another reference frame at work in any simulation is the one dealing with each object's *center of mass* (COM). When an object is created in Havok it automatically calculates where theCOM is, depending on the shape of the object. The Havok engine needs this to simulate believable physical movement. The COM of an object is not difficult to visualize. It is the point that needs to be supported to keep an object balanced. For example, in a movie where a car teeters over a cliff, the cliff edge is under the COM.

When positioning an object in your game you need not consider the COM. In the modeler when you set or get the object's position you deal with the position of the origin of the object as created in the modeler. Havok takes care of the mapping between the COM and the object's origin or pivot as defined by the modeler.

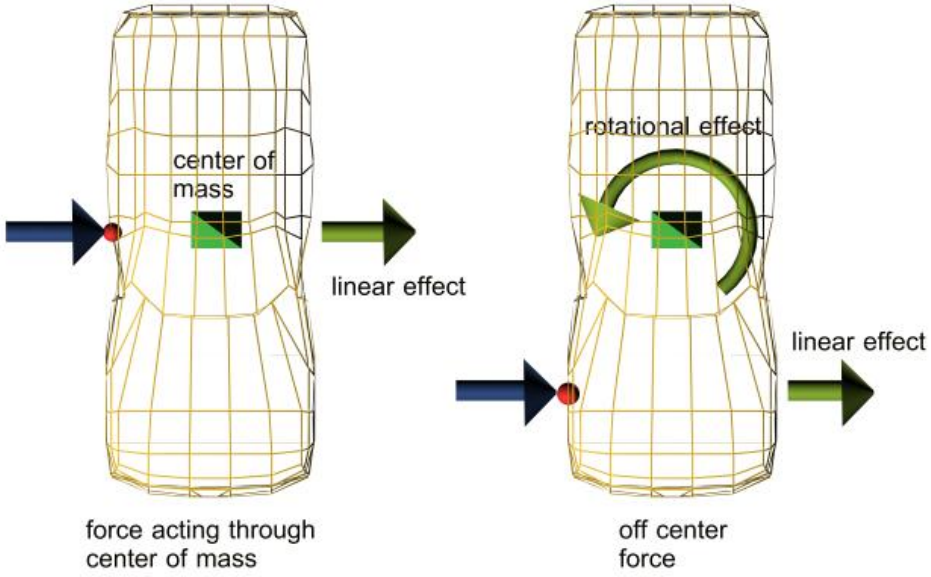


Figure 3.78: Depending on where forces are applied rotation may or may not happen

If a force is applied at or through an object's COM, the object's linear acceleration changes but no rotational acceleration is introduced. In the diagram above the force in blue is applied to the door panel of the car on the left. The line of force goes through the COM (the green box) and no rotation is created. On the other hand, when a force is applied so that it does not go through the COM a torque (or twist) is introduced. This torque alters both the linear and the angular acceleration of the object. In the diagram above the second car is hit from the side near the tire. This causes both movement left to right and an anti-clockwise twist. The further the force (or more accurately the line of force) is from the COM the bigger the twist that is introduced. The same is true for off center impulses.

When applying impulses and forces it is important to specify the point to apply the impulse or force with respect to the COM of the object. Havok gives easy access to this information and even allows manual movement of the COM to make gameplay more realistic. In the car example above, moving the COM to where the engine is will give more realistic "car like" behavior (the engine is normally the heaviest part

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

of a car).

If you'd like to know more about how Havok represents positions and orientations please take some time to read our article on the various common representations and conventions and how we manage them. You can find more information in the section entitled "Rotations, Handedness, and all that."

Name	Units	Description
Orientation	Axis + Angle (radians)	An object's orientation with respect to the world/game coordinate system.
Angular Velocity	Axis + radians/s	The speed at which the object is rotating: i.e. the number of radians per second the object rotates, usually specified with the axis around which it is rotating.
Angular Momentum	kg rad/s	Angular equivalent of momentum. This quantifies how hard it is to increase or decrease the rotational velocity of an object.
Angular Acceleration	rad/s^2	Angular equivalent of acceleration. This is the rate of change over time of the angular velocity.
Angular Impulse	kg rad/s or N rad	Angular equivalent of impulse. This is a measure of a change in angular momentum. Apply an angular impulse to an object if you want to instantaneously affect its angular velocity.
Torque	kg m^2/s^2 or Nm	This is the angular equivalent of force. Units are Newton meters.

Table 3.9: Physical terms and descriptions

3.10.1.9 Physical properties

These properties depend on the object's material(s) and affect its behavior in a physical simulation.

Name	Units	Description
Friction	Dimensionless	The friction coefficient (usually a value from 0 to 1) which specifies how "sticky" or "rough" an object is. See below for a more detailed explanation.
Restitution	Dimensionless	The restitution coefficient (usually a value from 0 to 1) specifies how much kinetic energy is lost during a collision between two objects e.g. with a value of 0, all energy is lost and objects appear to come to a complete halt when they collide. With a value of 1, no energy is lost and objects bounce off each other with an equal but opposite velocity. For values between 0 and 1, the objects lose some but not all energy with each collision. This is a per object or per object/material property.
Mass	kg	A measure of an object's resistance to change in motion, or the amount of matter in an object. Not to be confused with weight, which is the attraction of the earths gravitational pull on a certain mass. To make matters more confusing, mass is defined officially by the weight of a mass of platinum iridium stored in Sevres in France!

Table 3.10: Object properties

Dynamic and Static Friction

A friction force is one that resists the relative motion or tendency to such motion of two bodies in contact, that quantity which attempts to prevent surfaces sliding off each other. Friction is the key factor in allowing stable stacking (i.e. stacks or piles of objects that come to rest, held in place by the friction at the points of contact). During all collisions a certain amount of energy is lost due to friction (and mostly converted to heat in the real world).

Friction manifests itself in two forms, static and dynamic. Static friction operates when objects are at rest; it attempts to prevent objects moving or sliding. If a force is applied to an object large enough to overcome static friction, the object begins to move/slide. At this point dynamic friction kicks in and, so long as the object is in contact with another object or surface, dynamic friction attempts to slow the object down.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

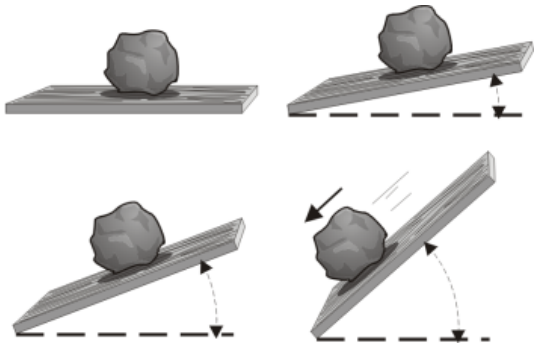


Figure 3.79: Static and dynamic friction in action

The figure above shows that the boulder is held in place by static friction until enough force is applied to break the contact (i.e. the plank has been raised to a sufficient height and gravity forces it to move). When the boulder begins to slide and dynamic friction kicks in which acts against the sliding action, generating heat. In Havok objects have a single friction coefficient value and the engine manages the transition between static and dynamic friction.

3.10.1.10 Collision Detection

Collision Detection is possibly the most crucial part of any physics engine. However if naively implemented, it can account for over 90% of the CPU time required for game simulation. We want to know when a player collides with the level, when bullets collide with the player, when bullets collide with the level etc. Given that we are usually interested in large numbers of interacting objects, we have a potential explosion in the numbers of collision tests required. At a worst case, with n objects in a scene we need to guarantee that every possible pair (sometimes called a *collision pair*) is checked. This requires $n(n\text{- }1)/2$ tests. This means 2 objects require 1 test (A with B). 3 objects require 3 tests (A with B, A with C and B with C). 100 objects require 4,950 tests! This gets CPU hungry very quickly. As the physics engine needs detailed information about each collision in order to resolve it correctly, each individual detailed collision test can be expensive so such a large number of tests is undesirable!

There are a number of ways to speed up the collision detection process:

1. Reduce the number of collisions that require detailed collision results to be generated (e.g. use a simpler collision test first)
2. Reduce the complexity of objects tested for collisions.
3. Reduce the number of simultaneously moving objects.

Obviously always try to achieve 3) first. The number of objects active in a given scene is the primary source of CPU load. If objects can be ignored this speeds up the simulation. In the case of 2), the difficulty in providing detailed collision results is directly associated with the complexity of the objects. This is addressed in a later section. First, we deal with how the Havok system attempts to achieve 1), a reduction in the number of complex collisions.

Multiphase collision detection

The Havok system employs a series of collision layers, each progressively more complex and returning more detailed results. After each stage as many objects as possible are eliminated from the test process, before moving to the next stage with the resulting data.

Broadphase collision detection

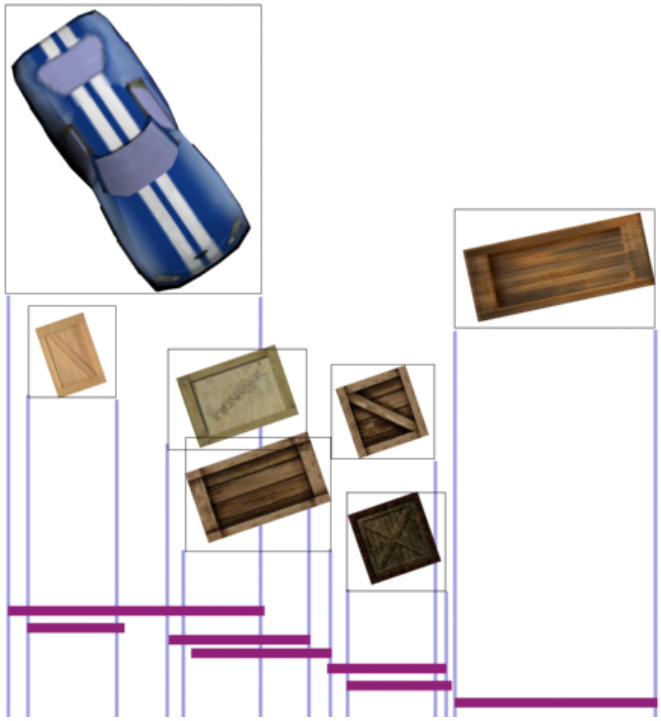


Figure 3.80: First pass in a collision detection system - attempting to eliminate as many collision pairs as early as possible

Above is an example of the first pass of a collision system. The car is driving toward a pile of crates. In this case we use *axes aligned bounding boxes* (Aabbs), boxes that are lined up with the world coordinate system and fully enclose each object. We test first to see if any boxes overlap (this is a much faster operation than doing full collision testing on arbitrary shapes). If they do, the collision pair is passed to the next, more complex collision layer. Of the potential list of 21 collision pairs in our example ((7x6)/2) we find that only 4 pairs have overlapping boxes (the car and the 3 left most boxes) so we can forget about all the other pairs. This is sometimes called *trivial rejection*.

Even though we identified possible colliding pairs of objects it does not mean that they are actually colliding only that they might be colliding so we move on to a more accurate collision detection layer.

Midphase collision detection

Now the set of potentially colliding objects is reduced, more complex tests can be carried out to determine if two objects are potentially colliding before finally passing them to the narrowphase stage.

In Havok, MOPP (Memory Optimized Partial Polytope) technology is an example of the Midphase layer. MOPP technology is specifically designed to allow large fixed landscapes to be stored with a very low memory overhead. It allows for extremely fast collision detection between moving objects and landscapes. MOPP as a Midphase layer reduces the number of possible collisions between game objects and landscape down to the much smaller set of likely collisions between game objects and specific landscape triangles.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

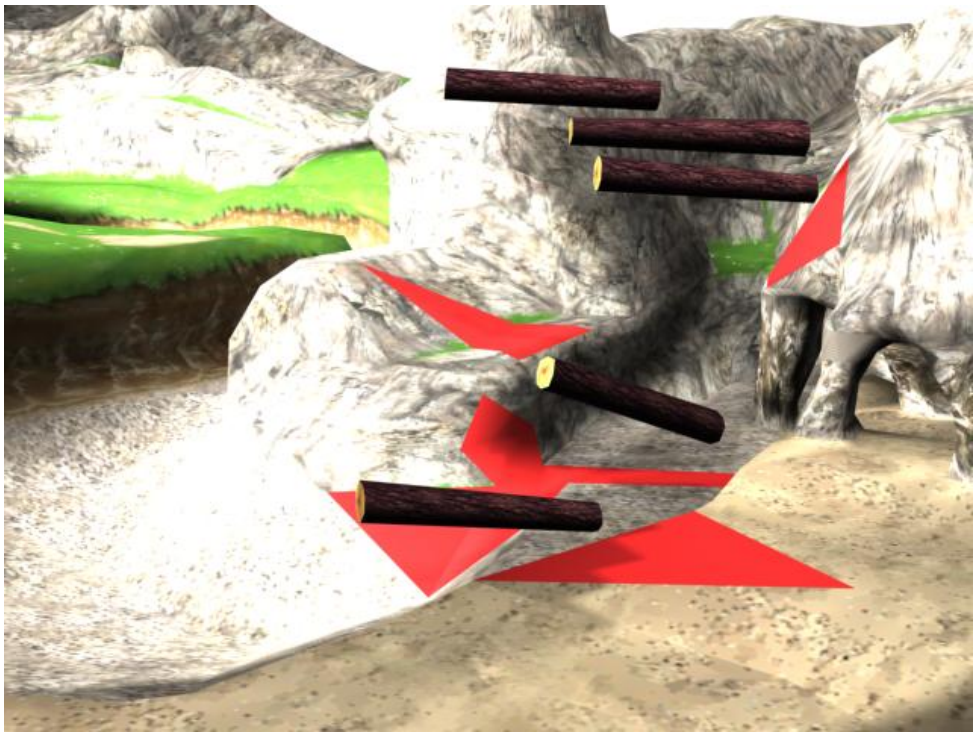


Figure 3.81: Midphase MOPP

In the figure above we see logs about to fall onto a static landscape (the landscape contains 1000's of triangles). The Broadphase will have detected that all of the logs are overlapping the Aabb of the landscape and all are potentially colliding. The Midphase (MOPP in this case) reduces the problem by identifying the triangles in the landscape which are likely to be colliding with the logs. This much reduced set of possible collisions is now suitable for passing to the final narrowphase.

Narrowphase collision detection

This final phase performs accurate collisions test to determine if objects actually collide. If they do, information like the point of collision, the normal to the objects at the point of collision etc. is calculated. These calculations can be quite CPU intensive, which is why the previous two stages are used to prune the set of objects that are processed at this stage.

The good news about this whole process is that Havok automatically implements collision detection management and can create bounding boxes, MOPPs etc. for all objects in the simulation. Note that Havok has been designed to be modular so if your game has a very fast collision culling algorithm already developed and implemented, it can be slotted into the system easily.

The user controls the complexity of the collision representation of the shapes of the physical objects. The next section deals with the various types of object shapes that can be used and their complexity for collision detection.

Rigid Bodies and Collision Geometries

The shape of objects tested for collisions has a major impact on the speed of collision testing. If assumptions can be made about object shape or the geometry simplified for collision testing, a lot of CPU time can be saved.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

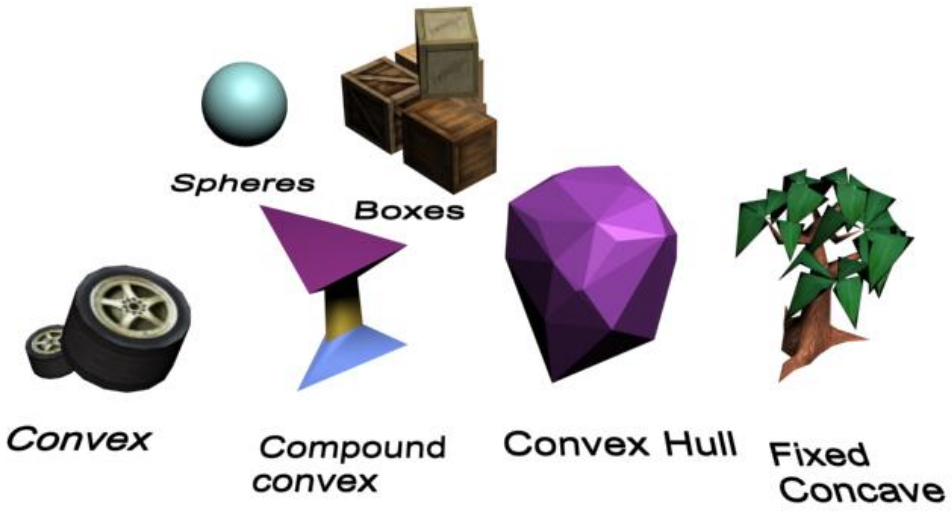


Figure 3.82: Some of the optimized geometry formats for Havok

In the Havok physics engine, objects are classified according to their shapes, and particular shapes have particular properties that make them easier to deal with during collision detection. The following list classifies object shapes in order of increasing complexity:

- spheres
- planes
- boxes
- polygons

For the simple shapes Havok has an implicit mathematical representation of the object and bases the collision test on this. For polygonal representations, Havok uses a description of each object in terms of these polygons (usually triangles). In this case objects are classified, in order of increasing complexity, as:

1. Convex: imagine wrapping an object in a cellophane wrap if the cellophane touches every part of the surface (i.e. there are no hollows in the surface that the cellophane does not reach) then the object is convex e.g. a beach ball is convex, a starfish is not. Treating an object as convex implies that it has in *inside* and an *outside*. For shapes that are "near enough" convex you can get big speed-ups by using the "convex hull" of the shape. The convex hull is just that cellophane-wrap representation. Havok can calculate this automatically and the approximation speeds up collision testing enormously.

   The amount of approximation will represent a trade off between speed of collision and accuracy required. This is typically very context specific; e.g. for the car chassis below we could get by with

a simple box representation. If we want to see our cars spinning and wobbling on their roofs then the convex hull gives a pretty good approximation of the graphics mesh.

Figure 3.83: Concave mesh and convex hull representations of a car chassis

2. Polygon Soup: a collection of polygons, not necessarily connected, all grouped and classified as a single object. This is the most expensive format to detect collisions with, but is also the most general. The majority of fixed level data will be stored using the polygon soup format.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

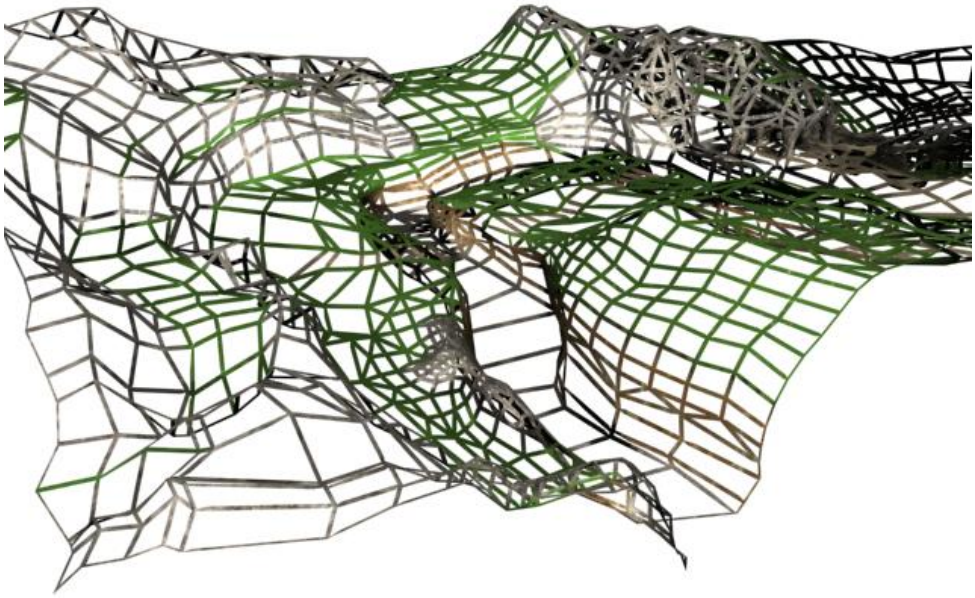


Figure 3.84: Fixed Landscape represented by arbitrary polygons

Inside and Outside

One major consequence of the assumption of solidity and the problems of interpenetration is that you cannot consider objects to be hollow. A common mistake is to create a box for a room and begin to place objects within the box, thinking these are being placed inside the room. A Havok box, by default, is a solid geometry. Any objects placed inside are tagged as interpenetrating and will simply be pushed out of the box (possibly at speed!) when simulation starts.

One way to avoid this is to represent the room/box using a polygon soup geometry (unconnected triangles). This will have the desired effect of a hollow shell for the room.

Setting Position and Orientation directly

Physics engines carefully store the state of all objects in a scene and update this state from one time step to the next. If an external system makes changes to the scene, the physics engine needs to update its state accordingly. This is OK *except* when changes made to the scene violate the simulation stability or consistency that the physics engine attempts to maintain.

A good example of this is setting object position or orientation directly (sometimes called *object warping*). It is easy to see how arbitrarily setting the position of an object in the physics scene may cause problems:

- If an object is warped to a position that causes interpenetration, without additional information the physics engine must simply guess how to resolve the inconsistency. Havok normally does a reasonable job of resolving such interpenetrations but if it is known in advance that such interpenetrations will occur frequently Havok provides *collision filters* to disable collisions between objects e.g. if a key framed object is always in close proximity to static landscape and risks interpenetration, but the designer is only interested with how the key framed object interacts with the player and dynamic

objects, then collisions can be disabled between the key framed object and the static landscape.

- If other objects are resting or stacked above an object that is warped away while the stack is deactivated, the remainder of the stack will remain floating above the space vacated by the warped object.
- If other objects are attached to the moved object by *springs* or *constraints*;: this is the worst case. By moving the object abruptly, the connections to the other objects are stretched instantaneously and can cause the system to destabilize and the constraints to oscillate wildly.

Objects may be moved outside of the control of the physics system if they are key framed in some way. For key framed objects, the Havok system needs to be informed that these objects can be expected to move non-physically Havok tracks these objects specifically and will attempt to resolve their motion in the simulation.

In general try to avoid direct manipulation of non-keyframed objects. Rather than moving objects directly it is always better to apply *forces* and *impulses*, either *linear* or *angular*, to the objects in order to "push" them towards the desired goal.

Collision tolerance

Havok has a global collision tolerance for the entire physical world, specified in world units. By default, the world has a collision tolerance of 0.1. A collision tolerance is an artificial buffer layer which tells the system that at some small distance (here 0.1) two objects are "nearly" colliding.

Having a non-zero collision tolerance helps with two performance-related issues. Firstly, it is useful for the system to pick up potential collisions before they actually happen - when the distance between the objects is still greater than zero. For fast-moving objects, this allows the collision solver to prevent interpenetration as early as possible.

It is also useful for the system to maintain collision information even while the objects are slightly separated. For instance, when an object is sliding across or settling on another object, this allows the system to maintain a manifold (a kind of 2D map of the collision) rather than having to create new contact points in every simulation step.

Convex object radius property (or shell)

Another speedup technique used is a collision radius property, set on a per object basis. The radius parameter allows you to add an extra "shell" to any convex shape.

Adding a collision radius to a shape can also improve performance. Convex-Convex collision detection algorithms are fast when shapes are not interpenetrating, but slower when they are. Adding a radius makes it less likely that the shapes themselves will interpenetrate, thus reducing the likelihood of the "slow" algorithm being used. The shell is thus faster in situations where there is a risk of shapes interpenetrating - for instance, when an object is settling or sliding on a surface, when there is a stack of objects, or when many objects are jostling together.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

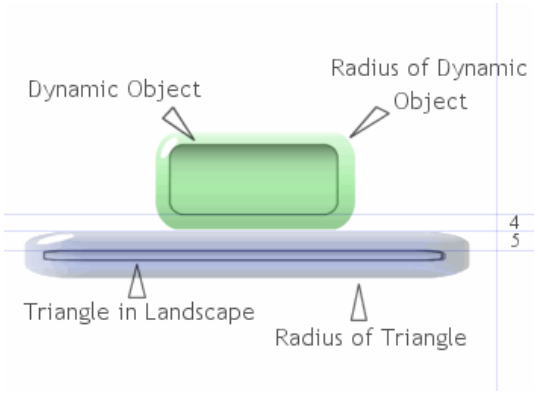


Figure 3.85: Shell tolerance

By default, an hkpConvexShape's collision radius value is 0.05. Havok tries to keep objects apart by the sum of their two radius measures. If you do not want to have the shell, you can change the radius value during runtime.

Setting the Scene

The first stage in any physics simulation is to create the scene. This is slightly different to constructing a 3D scene without physics and you should remember requirements like avoiding interpenetration. If you construct a scene in a purely geometric way you could experience difficulties placing objects so that they rest stably on other objects while avoiding collision problems.

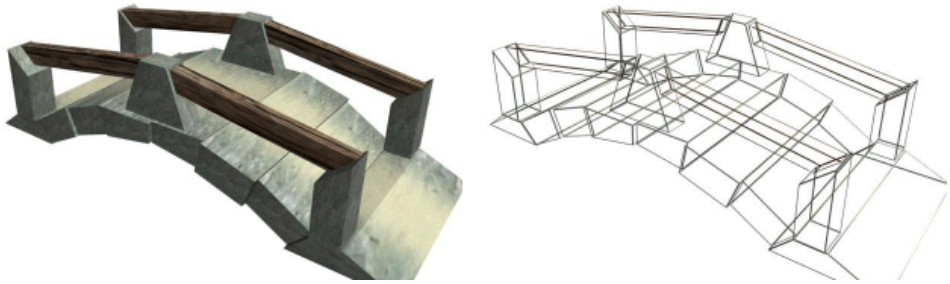


Figure 3.86: Creating a stable stack of objects

As an example take the case of creating a free standing bridge. If stones interpenetrate or if the collision radii overlap, the stones will all be active and immediately moved away from each other by the engine. Even if everything stabilizes you still get a big performance spike during the first simulation steps.

The best solution is to create the stones initially separated by a distance of more than the collision radii. Then simply simulate the scene for a period of time; the stones will fall the small distance and will settle into a stable stack. Eventually they will deactivate as the energy management system kicks in. Now you have a stable stack that is deactivated and ready to be saved. Store the positions of the bridge pieces now you can use these positions to create your bridge ready for streaming into your game.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

3.10.1.11 Constrained Dynamics

In many cases we want to construct physics systems that have joints or attachments between sub parts of a larger assembly. We need to be able to make machines, robots, traps and puzzles. Havok provides a full range of the simulation nuts and bolts required, including;

- Springs: forces are applied to objects connected by springs to attempt to keep the objects within some fixed distance of each other (the rest length).
- Dashpots (a more mechanically based spring - in real life dash pots contain oil to smooth out vibrations)
- Mechanical joints: The common joints in the world around us; prismatic joints (e.g. a piston in a machine, sliding door) hinge joints (e.g. a door, the sign swinging above the spooky saloon), axles (e.g. wheels, cogs), human joints (everything from the hinge-like elbow to the more complex body parts like shoulders and hips) and all kinds of vehicle components.

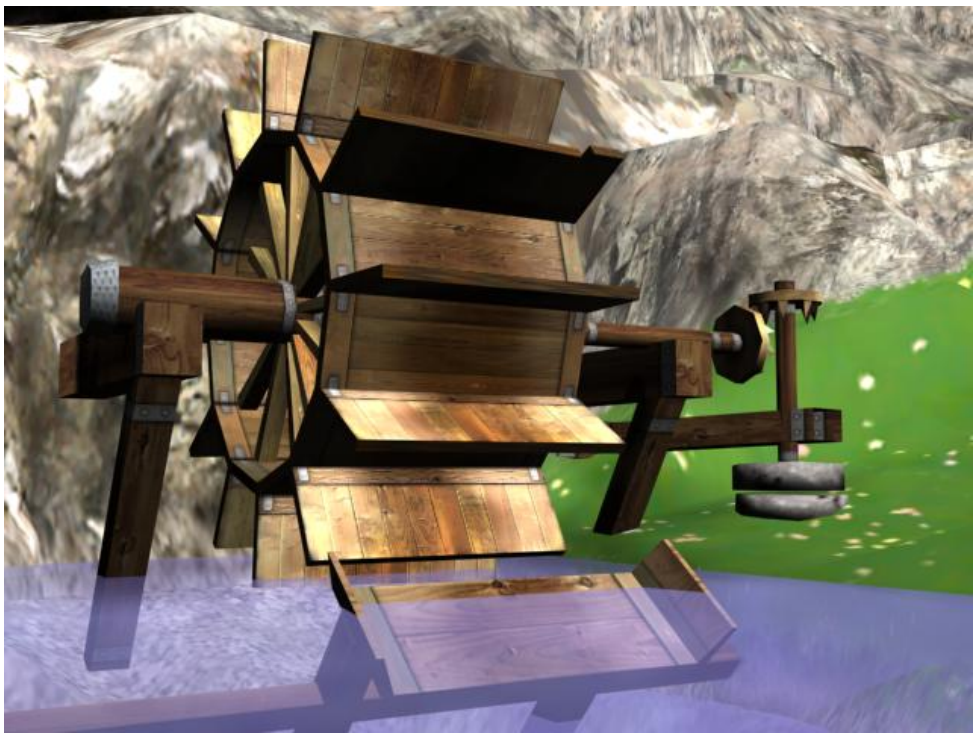


Figure 3.87: The water wheel is connected to the mill stones via an axle and some cogs

Constraints are a requirement for creating interconnected systems. Forces applied to one object are instantaneously applied to all other connected objects. In the example above, when the water turns the paddle, the axle turns, the horizontal cog turns the vertical cog which turns the vertical axle and the mill stones begin to grind.

3.10.1.12 What next?

You should now have a general understanding of all of the elements of a rigid body physics simulation. When you start using Havok's physics engine within your game, it should now be easier to plan how and

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

where you would integrate it into your existing code.

The SDK ships with a range of kick-start demos within our demo framework. The demos address specific API features and bigger game element concepts and if working on a new game, compiling and tweaking these demos is a good starting point to get used to the different features within Havok.

3.10.2 Rotations, Handedness, and all that.

3.10.2.1 Note On Notation

Note: In this doc we interchangeably refer to rotations/orientations, and also translations/positions. Strictly speaking rotations and translations are best thought of functions, and orientations and positions are the things they act on. However since theres an obvious mapping between e.g.. "the translation which maps the origin to (1,3,6)", and "the position (1,3,6)", we dont draw a distinction.

Notation: In this doc, the expression "R(Z,90)" is used to mean "the rotation of ninety degrees around the Z axis".

3.10.2.2 Introduction

This document explains common representations and conventions for rotation and transformation. It also explains how to identify and convert between different representations.

At Havok were in quite a unique position. Were exposed to more integration issues than most other middleware companies, mainly because most other middleware is either entirely adopted (together with a toolchain, math conventions etc. like most of the rendering middleware companies) or the flow of information is one way e.g. from toolchain to renderer, or from game engine to sound middleware. Information in Havok flows both ways, the scene is initially set up in a toolchain and this information makes its way to construct Havok objects and initial scene positions and orientations, and then when the simulation takes place this orientation information must flow back from Havok to the renderer. Generally most of this information is transformation information i.e. positions (translations) and orientations (rotations). At each integration interface you potentially have to convert from one set of conventions to the other.

- Part 1 of this document describes common sets of conventions found in the game industry used to represent transformations. It explains the reasons and sources of confusion, particularly with respect to rotation and handedness conventions. Its very useful if you are beginning integration with any 3D tool or if youre designing an engine from scratch.
- Part 2 describes how to convert between the different sets of rotation conventions. It gives a common practical example for converting between Right and Left handed systems and then generalises this to an arbitrary "change of basis" conversion. Finally we present an a working example going from a common modeller (3DS Max) to constructing a Havok scene and feeding the results straight into a DirectX rendering example.

3.10.2.3 Representations

Havok does not have a "handedness".

Havok is not "right-handed".

Havok stores rotations in two objects, a matrix format (hkRotation), and a quaternion format (hkQuaternion).* The way in which the rotation is stored in each case has been chosen to adhere to certain mathematical conventions, however it is a choice and other people will (may) have similar looking objects which represent (to them) the same rotation, but the data may be different.

Or, looking at it the other way, you may interpret one of our objects like one of yours and your interpretation of the resulting rotation may be different to ours.

Why is this?

Well, the thing about rotations is, theyre awkward to represent. To justify this, let us first consider "positions".

Note:

*Havok also stores rotations "in" two other formats, namely wherever "torque" or "angular velocity" is used. This will be dealt with later. The rotations inside hkTransforms are stored as hkRotations

Representing Positions

The thing about positions is, "you know where you are with positions".

For a start (and were considering 3D positions here), theres a pretty universal convention for describing them:

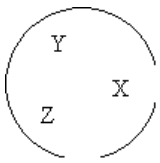
Implicitly there are three canonical bases:

(1,0,0), called X

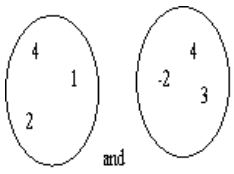
(0,1,0), called Y

(0,0,1), called Z

No-one calls e.g. (1,0,0) Y, but even if you called the bases Bob, Phil and Geoff it wouldn't matter because there is an implicit order here, given by the fact that we write them down right-to-left, rather than, for example as a set:



which would be not much use to anybody. How should one find the vector between:



So to avoid any confusion we normally write these as (for example):

(4, 2, 1) and (4, -2, 3)

Thats great, because we can then go and subtract them, dot then, cross them etc.

[Theres also this nice symmetry between the elements of a "position" : X,Y,Z are all "equal" (you can go into the mathematics of this to justify it, but basically the X-direction isn't any longer, or whatever than the other directions, and anything you can do to the x component of a position, like multiply it, or subtract it or zero it, you can do to the others in just the same way)].

So if I say to you "consider the position (1,2,3)", youll have a pretty good idea what that "looks" like. Now, youll probably imagine some orthonormal system where you've picked X to be "right", Y to be "up", Z to be "in", or some permutation of these. It doesn't matter. It doesn't matter if Z is "up", or if X is "left", thats all in the embedding of this abstract space into the real world. Mathematically no one of them is "up" or "right". The important thing is that if I then say you "now consider the position (1,5,7)", and ask you how far away it is youll say "3 Y units and 4 Z units" or somesuch, no matter how youre thinking about it. Also, if I ask you anything about dot products, cross products etc, youll get the same answer as anybody would.

How does this apply to Havok?

Well, Havok is abstract too. It deals with triples of real numbers (X,Y,Z). There is no "up", there is no "right". The default value for gravity is (0, -9.8, 0), but this is not hardcoded.

If you do the Havok equivalent of (1,5,7) (1,2,3) which is something like.

```
hkVector4 a(1,2,3);
hkVector4 b(1,5,7);
hkVector4 c;
c.setSub4(b,a);
```

then youll get the same result anybody else will with any other engine, i.e.. (0,3,4) because mathematically this is the right answer, and theres no reason they would have written a maths library which claimed to implement "Vectors" which didn't adhere to mathematical definitions. This applies also to the interface which gets/sets the X,Y and Z elements. They dont have to store them contiguously in memory as X,Y,Z, you can store them how you like, but it is highly probable that the storage in memory will be in the same order as X,Y and Z are alphabetically

So "positions/translations" are pretty much unambiguous / easy.

"Representing" Rotations

"Rotations" are hard. For a start, how do you represent them? Heres one representation:

$$\begin{bmatrix} .781 & -.48 & .394 \\ .550 & .832 & -.07 \\ -.29 & .272 & .916 \end{bmatrix}$$

Heres another:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

[0.0916, (0.183, 0.274, 0.939)]

And another:

(0.7, 0.267, 0.534, 0.801)

They all represent the same rotation in Matrix, Quaternion and Axis-Angle format respectively. [They all look pretty different, and they each have both 'good" and "bad" properties in terms of ease of use, efficiency etc. This is why all three are common in the games industry and are hence worth mentioning, though we shall not go into any comparative details here].

And theyre all *ambiguous*.

The easiest way to demonstrate this is to take the matrix case because its the one people are most familiar with (even though people tend to use Angle-Axis in conversation and examples, because its easier to say (less info), and fits on one line).

We all know you can multiply matrices by vectors in two ways: pre-multiplying and post multiplying, and in general these give completely different results. For example

$$\underset{R}{\begin{bmatrix} .781 & -.48 & .394 \\ .550 & .832 & -.07 \\ -.29 & .272 & .916 \end{bmatrix}} \underset{X}{\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}} = \underset{X'}{\begin{bmatrix} .781 \\ .550 \\ -.29 \end{bmatrix}}$$

But

$$\underset{X}{\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}} \underset{R}{\begin{bmatrix} .781 & -.48 & .394 \\ .550 & .832 & -.07 \\ -.29 & .272 & .916 \end{bmatrix}} = \underset{X'}{\begin{bmatrix} .781 & -.48 & .394 \end{bmatrix}}$$

which brings us to the core of understanding rotation representations:

"Defining" Rotations

Rotations are less defined by what they are, but rather what they do.

So we can uniquely define a rotation for example by specifying the action it has when applied to all possible vectors. Fortunately, due to the nature of vectors, its possible to simplify this to just specifying the action it has on each of the three basis vectors X, Y and Z.

So think of rotations as functions or operators. You havent fully specified them until you say how they apply to vectors.

So:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

$$\begin{bmatrix} .781 & -.48 & .394 \\ .550 & .832 & -.07 \\ -.29 & .272 & .916 \end{bmatrix}$$

with the operation of left-multiplication on a position interpreted as a column vector, i.e. $v \rightarrow Mv$

is unambiguous (because left-matrix multiplication is).

"[0.0916, (0.183, 0.274, 0.939)] as a quaternion [a,b] = q with the operation of

$p \rightarrow qpq^{-1}$, where $p = (0,v)$, and the result is the imaginary component of the resulting quaternion."

is unambiguous (because quaternion multiplication is).

"(0.7, 0.267, 0.534, 0.801) as a angle,axis pair [a,b] with the operation of

$p \rightarrow p+sin(a)bp+(1-cos(a))\ b\ (bp)$"

is unambiguous (because the cross product is).

Removing Ambiguity

The reason why this can cause problems is because people frequently dont even realize there is this ambiguity, hence dont supply the extra information necessary to resolve it. So long as you are "consistent" in your use of rotations, there *is* no ambiguity (as far as you know). It involves a bit of "stepping back" from the problem to even realize it exists:

For example, consider the "rotation":

"Ninety degrees around Z"

This certainly means a rotation which leaves (multiples of) the Z-axis invariant, and the angle between v and v (v after being rotated) is 90 degrees for any v perpendicular to the Z-axis. But theres two possibilities here - either (for example) X goes to Y, or X goes to Y. They are different rotations, in the same way that "clockwise" and "anticlockwise" are different. And you could associate one of them with being "clockwise" and one with being "anticlockwise", but in the absence of any convention theres no way to say which should be called which.

Once you have decided on one of the two choices, you have effectively determined which way a "positive" angle rotates, i.e. you have defined "positiveness" for an angle-axis representation.

As the diagrams below show, the same rotation can be thought of as either "clockwise" or "anticlockwise" depending on the "handedness" of your system.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

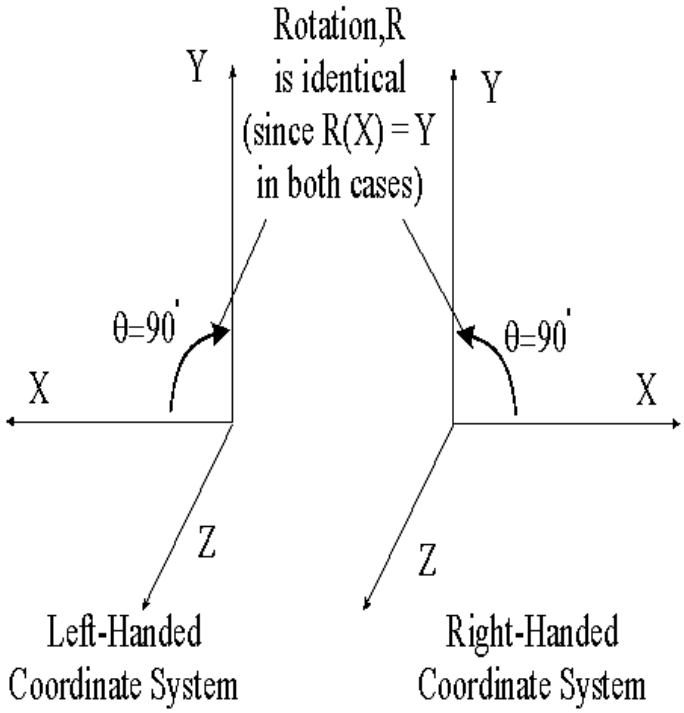


Figure 3.88: Left And Right-Handed Coordinate Systems

The Right-hand Rule (and why it almost helps)

Now there is also a convention for defining "positiveness" of rotation, without specifying the "handedness" of your coordinate system, known as the "right-hand rule". This says that a positive angle is measured *anticlockwise* around the axis of rotation. It is a convention that in such interpretations of clockwise/anticlockwise that the axis is pointing "out" of the clock towards you (thank goodness).

The right hand rule is frequently illustrated by something like this:

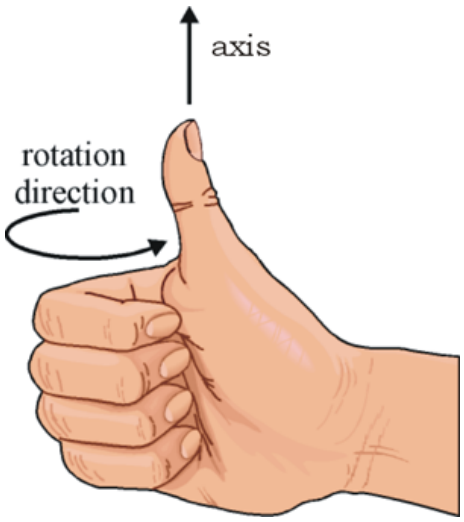


Figure 3.89: The "Right Hand Rule"

This *seems* to remove any ambiguity because it is clear in any diagrammatic interpretation, such as lining

up your right hand with a screen image, or either of the coordinate systems above, which way is a positive rotation. *But* as you can easily see, applying this rule to the "right-handed" coordinate system would define positive as rotating X to Y, whereas applying it to the other would mean wed get X rotating to -Y. These are two different rotations, so saying "we use the right hand rule" isn't enough.

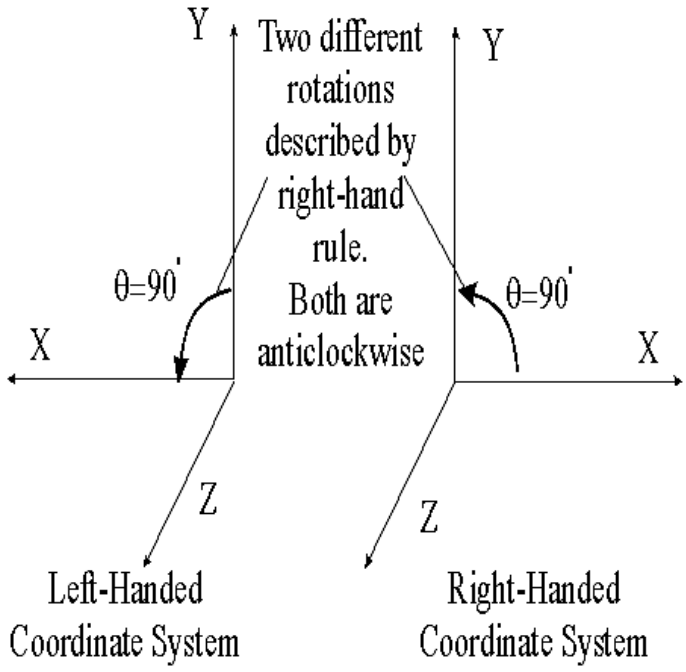


Figure 3.90: The Right-Hand Rule In Both Coordinate Systems

Ambiguity Resolved.

We know for certain that different people use coordinate systems of different handedness, its too late to change that now (even if we wanted to). One possible way of resolving ambiguity would be to get everyone to adopt the "right-hand rule", but this would mean (as illustrated above) that the actual rotation(as an operator) described by people, for example "Ninety degrees around Z", would depend on their coordinate handedness.

Fortunately, there is a corresponding "opposite" rule for rotations called the "left-hand rule" which defines positive rotations in the opposite direction, i.e. clockwise, and it happens that:

(Almost) everyone who uses a right-handed coordinate system adopts the "Right-hand" rule, and those who use a left-handed coordinate system adopt the "Left-hand" rule.

This is true of (amongst others): DirectX (Left), Renderman (Left), Unreal (Right), Renderware(Right), 3DSMax(Right), Maya (Right). It is mostly true of OpenGL which uses the right hand rule, but can use left or right-handed coordinates. It can be thought of as being true of Havok too, which (since it doesn't have a renderer as part of the core engine) is neither right not left handed, but defines positive rotations in the same way (their operation is the same) as all the others.

The net result is that (almost) everybody agrees that a rotation of+90 degrees around Z maps X to Y, or in other words, we all agree on the definition for a positive rotation (as an operation).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Warning:
We say "almost" because occasionally there are anomalies, and since the choice of handedness/rule is somewhat arbitrary, its perfectly OK to have a system which rotates "the other way".

Example: 3DS Max uses a "right handed coordinate system". Its user-interface adopts the "right-hand rule", which means that all rotations are counterclockwise around the axis. However, its internal SDK adopts the "left-hand rule", which means that all rotations are clockwise around the axis.

Similarly, Gamebryo/NetImmerse uses a right-handed coordinate system, but the left-hand rule.

Representation Conventions

Now that we have dealt with the two possible "mathematical" conventions which may be adopted, its time to look at the possible "representation" conventions which any implementation in code might adopt.

The following are 4 "rotation representations":

- Axis, Angle
- Euler Triple
- Quaternion
- Matrix

In this document we will not address Euler angle representation because direct conversion to/from Euler angles is normally done via a matrix format (which we will address).

We will also assume that the basic real-number representation chosen will be a *float*.

For *Axis,Angle* the obvious data representation is always chosen:

A single float for the angle, and a float triple for a *normalized* axis. The only storage choice to be made is whether to put the angle or the axis first in memory. With proper accessors (such as the hkQuaternion.getAxis() method), conversion of Axis,Angle to/from hkQuaternion, hkRotation is trivial, and dealt with in the next section on "Conversions".

For *Quaternion* the obvious data representation is also always chosen:

A single float for the real component, and a float triple for the imaginary component, where the real component is cosine(angle/2) and the imaginary component is sine(angle/2)**normalized* axis. Again, the only storage choice to be made is whether to put the angle or the axis first in memory.

For *Matrix,* there are two conventions (unfortunately). Since you can multiply a matrix on either side of a vector (in one instance the vector is considered a row, in the other it is considered a column), you have two choices of how to "represent" a rotation in matrix format.

In the following examples we assume a 4x4 affine matrix. Frequently some components are unused (ignored) or even not stored at all (effectively only 12 values are stored). We assume that the reader is familiar with such simplifications, and we do not deal with them here.

For example, R(Z,90) (rotation of 90 degrees around Z), which maps X to Y can either be represented as

0 -1 0 0

1 0 0 0

0 0 1 0

0 0 0 1

if the matrix applied on the left to column vector "pre" multiplies. Mx = x

or

0 1 0 0

-1 0 0 0

0 0 1 0

0 0 0 1

if the matrix applied on the right to row vector "post" multiplies. xM = x

An interface to these matrices which allowed us to, for example, get the first column would return

0

1

0

0

in the first case, but

0

-1

0

0

in the second case, so clearly were we to try and copy over data from one to the other via such an interface we would be accidentally changing the rotation (in fact, inverting it because for rotations R-inverse = R-transpose). Worse, if the matrix had a translational component, we might attempt to copy over

0 -1 0 7

1 0 0 4

0 0 1 3

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

0 0 0 1

directly into a matrix which is supposed to be applied on the right, and clearly the translational data would be put in the wrong place. The resulting matrix if applied on the right to the origin (0,0,0,(1)) would give us (0,0,0,(1)), which clearly has not been translated, hence the conversion cannot be correct.

Fortunately, it happens that a by-product of the conceptual way in which the rotations/transforms are applied (either to "column" or "row" vectors) means that *usually* the data will actually be *identical when laid out in memory*, no matter which "representation" is chosen.

To see why this is, consider the choice of using matrices which are to be "pre" multiplied (applied on the left). Because the mathematical object which these matrices apply to must be a "column" vector, it is *likely* that the matrix itself is also stored "by columns", or in column-major format. Column-major format of a matrix stores the elements linearly in memory, column by column. That is to say the elements in a single column are adjacent in memory and their address increases as the row index increases. For example:

0 -1 0 7

1 0 0 4

0 0 1 3

0 0 0 1

is stored as:

0 1 0 0 -1 0 0 0 0 0 1 0 7 4 3 1

Now consider the choice of using matrices which are to be "post" multiplied (applied on the right). Because the mathematical object which these matrices apply to must be a "row" vector (opposite to the case above!), it is *likely* that the matrix itself is also stored "by row", or in row-majorformat. Row-major format of a matrix stores the elements linearly in memory, row by row. That is to say the elements in a single row are adjacent in memory and their address increase as the column index increases. For example, the matrix *which represents the same transformation as above when applied on the right*, is

0 1 0 0

-1 0 0 0

0 0 1 0

7 4 3 1

and when this is stored row-major, it is stored as

0 1 0 0 -1 0 0 0 0 0 1 0 7 4 3 1

just like the other one.

Note that in both cases the *translational* components of the transform are "at the end" of the memory block.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

So, it is *very likely* that any two matrix storage formats (either 4x4 or, if ignoring the last column/row 4x3 or 3x4) will be identical in memory. Havok, Renderware, DirectX, Renderman and Unreal all have the same storage in memory for their matrix representation, even though Havok is column-major whereas Renderware, DirectX, and Unreal are row-major.

Note:

Unless a system explicitly ties "columns" or "rows" to the block of data which stores the matrix as a whole, the storage cannot be said to be either "row major" or "column major". OpenGL for example specifies only that the matrix is stored in 16 consecutive floats there is no reference to "columns" or "rows" except in the GL documentation, which does so for ease of explanation, but actually rather confuses things by implying an ambiguity namely whether the matrices are pre- or post- multiplied. In fact the actual definition simply requires that (like the "common" matrix storage format above), the translational components are the 13^{th} 14^{th} and 15^{th} floats

See http://www.opengl.org/documentation/specs/version1.4/glspec14.pdf p34 and p36 and http://research.microsoft.com/~hollasch/cgindex/math/matrix/column-vec.html

Another way to determine which parts of the matrix data are rows, and which are columns is to examine the result of a matrix multiplication where the order is specified, since you know that the resultant matrix must be formed from dot products of rows from the first, and columns from the second (mathematical convention).

Havok has accessors for elements by (row,column) index pair, and it also has explicit getRow() and getColumn() methods and in fact its internal storage for an hkTransform is as an hkRotation (3 hkVector4s, m_cols) followed by an hkVector4 (m_translation). This means that Havok definitely stores its transforms in column-major format, and as described above, since it also uses the "common" storage format, one can deduce that Havok applies its matrices on the left (pre-multiplies, conceptually).

As a contrast, DirectXs matrix is defined as:

```
typedef struct _D3DMATRIX {
    union {
        struct {
            float _11, _12, _13, _14;
            float _21, _22, _23, _24;
            float _31, _32, _33, _34;
            float _41, _42, _43, _44;
        };
        float m[4][4];
    };
} D3DMATRIX;
```

which means that row elements are clearly contiguous in memory, hence it is row-major and applies its matrices on the right (post-multiplies, conceptually).

Note:

Once again, although this is adopted in most of the major packages and standards, this is not a universally accepted convention, so it is (at least in theory) possible for a system to store what is the "same" rotation (the operation is the same) differently in memory to the way Havok (for example) stores it.

3.10.2.4 Conversions

"Positive Rotation Rule" conversions

The left-hand/right-hand rules concern the mathematical conventions which determine which way a "positive" angle will rotate, and hence whether conversion is required for any of the structures which have an interface allowing you to get/set an angle (or axis), or which implicitly store data related to this representation. In Havok these are:

- hkQuaternion
- hkRotation
- (hkTransform, which contains an hkRotation)
- "angular velocity" as an hkVector4
- "torque" as an hkVector4

As described in the section on Removing Ambiguity, the convention for defining positive rotations is that you should apply the same "hand" rule as the handedness of your coordinate system. If, for some reason, you do not use this rule, but rather define positive rotations in the opposite direction so that you have R(Z,+90) maps X to Y, then a conversion must be made if you wish calculations made in Havok to be the same as those made in your engine.

In this case the conversion is easy, and it simply involves changing the sign of the angle used. Since this is mathematically equivalent to inverting the rotation (the inverse of a rotation of theta degrees around an axis v is clearly a rotation of theta degrees around v), this is what we must do for each representation used.

hkQuaternions

For an hkQuaternion, on creation from an axis,angle pair, just negate the angle:

```
hkVector4 axis(myAxisX, myAxisY, myAxisZ);
q.setAxisAngle( axis, -myAngle );
```

If the hkQuaternion has already been created for some reason, call

```
q.setInverse(q).
```

Alternatively, we can directly alter the internal parts of q:

An hkQuaternion stores a rotation (theta,v) (say) as:

hkReal cosine(theta/2), and an hkVector4 sine(theta/2) * v.

Clearly then the rotation (-theta,v) is stored as:

cosine(-theta/2), sine(-theta/2) * v

which is equal to

cosine(theta/2), -sine(theta/2) * v.

So all we have to do is negate the imaginary (hkVector4) part of the hkQuaternion.

Alternatively, since it is a (nonobvious) property of quaternions that both q and q (both real and imaginary parts of q negated) represent the same rotation, we can just negate the real part instead.

In Havok, both parts are munged into the single hkVector4, the real part being stored in the (otherwise unused) 4^{th}component, so the simplest thing to do would be to negate this 4^{th}component like this:

```
q.setReal(-q.getReal());
```

Similarly if converting from your own quaternion format then you will need to copy over the imaginary part directly, but negate the real part.

To convert back to axis,angle, call getAngle() and (if the angle is non-zero*) getAxis() and then negate the angle.

Warning:
*Remember - the axis of rotation is undefined if the angle is zero!

hkRotations

For an hkRotation, if constructing using setAxisAngle(), then negate the angle.

WARNING: Now heres where the first problem may arise if you are directly copying over matrix data, then you must know whether you adhere to the "common storage" convention or not. Havok uses the common storage convention, so if you use it too then everythings fine, theres no extra work to be done, and the next couple of paragraphs will tell you how to fix your hkRotations to allow for the difference in "positive rotation rule" convention. *However*, if you store your matrix data differently, then it is almost certainly the "transpose" matrix representation, and hence your rotation will already be the inverse of what Havok uses, *so there is no need to do the extra step below.*

If an hkRotation has already been created for some reason, call

```
r.transpose()
```

which transposes the hkRotation in-place. Since a rotation matrix is always orthonormal, this will invert it.

If you are converting from your own data (and have read the warning above), then either transpose as you copy over, or copy directly, and then call r.transpose().

Angular Velocities/Torque

In Havok, angular velocities and torques are stored in hkVector4s, and they also follow the same "positive rotation rule" as used for axis,angle representations. That is to say, for example, a body at the origin, aligned with the world axes, with an angular velocity of PI/2 radians/second around the Z axis (0,0,1)

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

will have its local X axis rotated to be coincident with the Y axis after 1 second. In other world, positive angular velocities rotate anticlockwise when displayed in a right handed system, and clockwise when displayed in a left handed system.

Torques have the same property.

Since angular velocities are stored as vectors v, where length(v) = rotational speed in radians/second, and v (normalized) is the axis of rotation, converting these involves the same logic as converting axis,angle data we simply negate the angle. The result is that we negate the entire velocity, and the same is true of torques e.g.

```
angVel.setNeg4(angVel);

torque.setNeg4(torque);
```

"Handedness" conversions

It may be the case that you wish to take data that has already been "modelled" in a system which has a different coordinate frame "handedness" to that which you wish to render it in.

For example, you may have used the Havok Exporter, or your own toolchain to output "right-handed" data from 3DSMax, which you then wish to alter so that it appears identical (visually) when put through your renderer which may be "left-handed".

Clearly all positional data must be modified (flipped), but it is less obvious how to change, for example, rotational data, such as a hkQuaternion or hkRotation, so that it correctly reflects the change in handedness. Because the matrix format is more common, and easier to explain, we deal with this format first, then we deal with quaternions.

To illustrate what must be done we shall first take the simplest (mathematical) example and derive the necessary operations "from first principles". This section may be ignored if you wish to move on to the more general solution, which we shall explain with less rigor, but more completeness.

Example. Matrix "mirror" conversion.

We consider the "X flip" to be the canonical illustration of "conversion" from "left" to "right". The first Figure in Section 3.4 illustrates this. Imagine that your modeller was right-handed (the coordinate system on the right), but that your renderer was left handed (coordinate system on the left). Clearly were you to export your data directly, and render it, the resulting "views" of the scene would be mirror images of each other. This is why you must do some form of "left-handed to right-handed" conversion.

To be precise in order that (with the camera in the correct place) both the LH and RH views *look* identical, we must mutate the data we export in some way.

Let us assume we start off with some mesh data and an associated transform. For simplicity, we also assume that the transform here is a "rigid" transform, that is it contains only pure rotation (no reflection, no scaling), and translation components.

What we want to get out is another set of data, and another "rigid" transform, which, when plugged into the renderer, will produce the same view. We do this by "mirroring" all our data and transforms, i.e. "flipping X".

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

If you flip X, then clearly:

- The vertex data must be"mirror" flipped. "Proof": consider an asymmetric object with identity transform. Clearly, since the final transform must also be the identity, the data must be flipped for the object to be displayed correctly.
- The translation of the transform must be flipped. "Proof": consider any object (e.g. a point) displaced from the origin e.g. to the right. Clearly, the X component of the translation must be flipped in order to displace the object to the left in the other handedness.
- The rotation of the transform must be "mutated". This is the tricky bit, so well go into a little more detail below.

To work out what the new rotation must be, given the old rotation (as a matrix), lets remind ourselves of what the rotation matrix is:

Its the image of each of the canonical axes, stored as columns (or rows, see Representation Conventions Section). So in order to mutate our rotation matrix we just have to work out what the image of the "new" canonical axes would be under the same rotation, expressed in terms of these "new" canonical axes. Were preserving Y and Z, and flipping X, so the "new" axes are X,Y and Z.

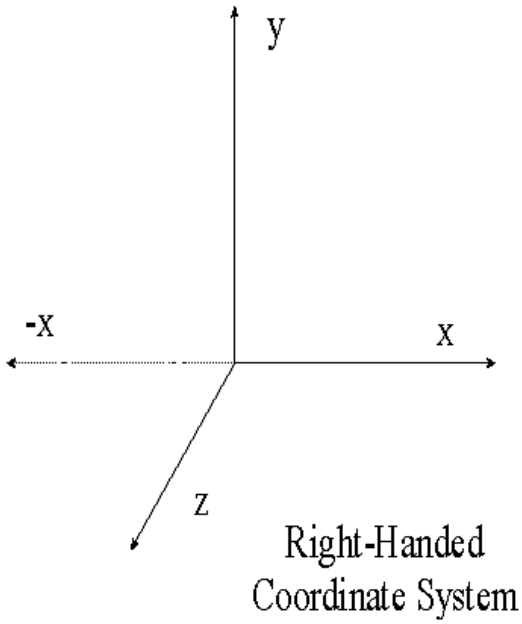


Figure 3.91: X, Y, Z and -X

We can "show" the rotation in our right-handed system by showing the vectors R(X), R(Y) and R(Z), representing the image of X,Y and Z under the rotation R:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

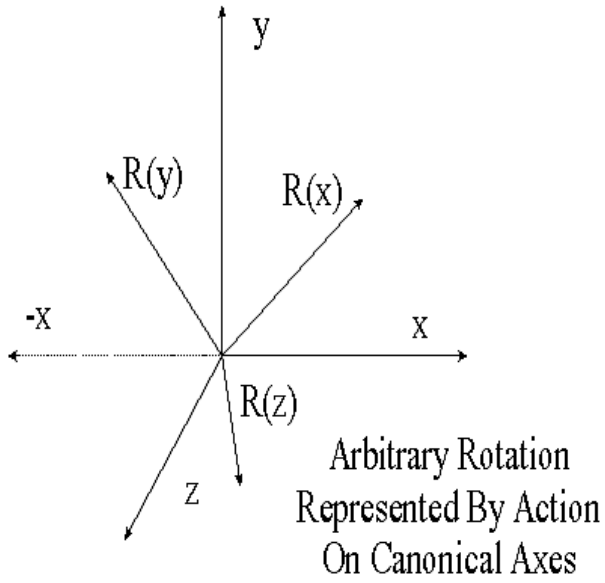


Figure 3.92: Our rotation in the right-handed system.

In the Left-Handed system, the same rotation would act on X,Y and Z. What we want now is to know what R does to X, but clearly R(-X) = -R(X):

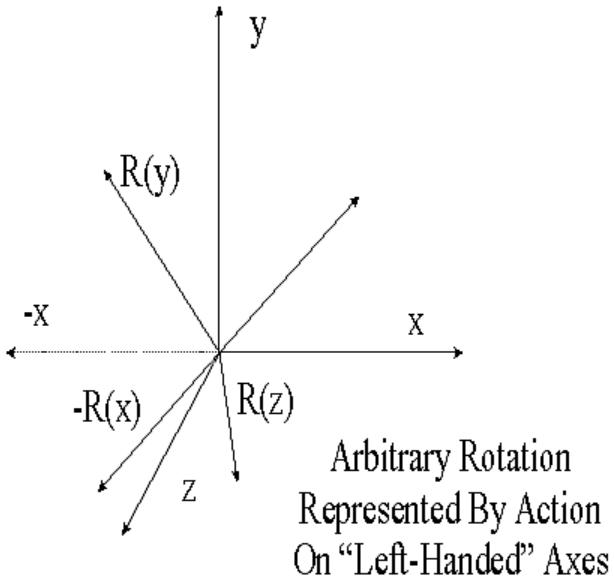


Figure 3.93: Our rotation in the left-handed system

We can now write down the rotation in the left-handed space, with canonical axes X', Y' and Z' say. In the left-handed space, all x components are flipped, i.e. Y' is Y and Z' is Z but X' is X. To keep things clear, we're going to specify which space every vector is written in by appending either RH or LH to the vector to specify which vectors are being used as the basis e.g.

(1,2,3)RH

which is equal to (-1,2,3)LH

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

because

(1,2,3)RH = 1*X + 2*Y + 3*Z

1*(1,0,0) + 2*(0,1,0) + 3*(0,0,1)

which is equal to

-1*(-1,0,0) + 2*(0,1,0) + 3*(0,0,1)

= -1*X + 2*Y + 3*Z

= (-1,2,3)LH

So if:

R =

R00 R01 R02

R10 R11 R12

R20 R21 R22

(written as a pre-mult matrix, i.e. X under this rotation maps to (R00, R10, R20))

then we want R (a rotation acting on the left-handed space generated by X, Y, Z) which maps:

X to R(X)

Y to R(Y)

Z to R(Z)

Now we know where each of X, Y and Z will go because, for example Y = Y, so

R(Y) = R(Y)RH = (R01, R11, R21)RH = (-R01, R11, R21)LH.

So the second column of R is (-R01, R11, R21)

Similarly

R(Z) = R(Z)RH = (R02, R12, R22)RH = (-R02, R12, R22)LH.

And finally

R(X) = -R(X)RH (note the minus sign here!)

= -(R00, R10, R20)RH = -(-R00, R10, R20)LH = (R00, -R10, -R20)

so

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

R =

R00 -R01 -R02

-R10 R11 R12

-R20 R21 R22

So this is how we observationally work out the matrix R (acting on vectors in the LH space), which has the same effect as R (acting on the corresponding vector in the =RH space. Hence, by the "rotations are defined by what they do" understanding, R and R are the same rotation, so we have successfully converted R from right handed to left handed.

General Solution (Change of Basis Conversions)

Alternatively, we could use the following intuition:

If we wanted to work out how a vector v = (*a,b,c*) in the LH space was affected by R, we could always:

- Flip *a* (i.e. the x-component)
- Apply Rotation to generate e.g. (*a, b,c*)
- Flip *a* again

What were doing here is representing v in the space we have the rotation in, then applying it, then reinterpreting our result back into the LH space. This obviously must give the correct result. In matrix form, the "flipping" can be expressed as:

F =

-1 0 0

0 1 0

0 0 1

since F(a,b,c) = (-a,b,c)

and we are applying FRF to the vector v. If you work out FRF you will indeed get the same result for R as we got before.

In general, we can say if T is a transform with zero translational component, and we wanted to work out how a vector (a,b,c) in the transformed space was affected by a rotation R, we could always:

- Apply T-inverse (maps point back to original space)
- Apply Rotation
- Apply T (maps T back to transformed space)

i.e. apply TRT-inv (expressed as pre-mult, obviously (T-inv)RT if post-mult).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

It simplified a bit in the "flipping" case because F-inv = F, but in general this is what youll have to do to convert your rotation. This will work not only for the "flip" case, but also for example like "swapping Y and Z", or "rotating so that Y points up/down". If you represent the munging you are doing by the correct transform, you can apply the above rule to convert your rotations as well as your translations.

It may happen that the actual operation of applying T and T-inv will simplify, like in the "flip" case where the final matrix could be computed by just flipping the sign of four elements. In this case, it may be useful to hardcode the conversion process, but be warned that where more complicated conversions occur it is easy to obfuscate what is going on, so unless there are performance reasons (and certainly when first writing this code!) it is better to use the full form: TRT-inv.

For example, here is a Havok function which will convert an hkRotation given a rotation matrix which may have reflection in it (using the changeBasis function of hkRotation, which does exactly this TRT-inv multiplication):

```
void convertRotationToNewSpace(const hkRotation& rotToNewSpace, hkRotation &r)
{
 r.changeBasis(rotToNewSpace);
}
```

Quaternions, and Axis/Angle

Now the above technique of map, rotate, map back will also work fine for any other rotation representation except that it is less general. Quaternions, in particular, cannot store reflections, such as those required to convert left-handed to right-handed. This means that you cannot pop the rotation part of your matrix T into a quaternion and compute qrq-inv as you might expect (where q is T as a quaternion, r is your original rotation as a quaternion, and we define the product using quaternion multiplication). In some cases you may be able to strip out the reflection component of the matrix which causes the problem, but we wont deal with that here, and well use a more direct approach.

Because quaternions are harder to "read" because they store the rotation in such a compact and computationally efficient format, we will clarify the conversion process by first examining axis/angle representations. See the picture below:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

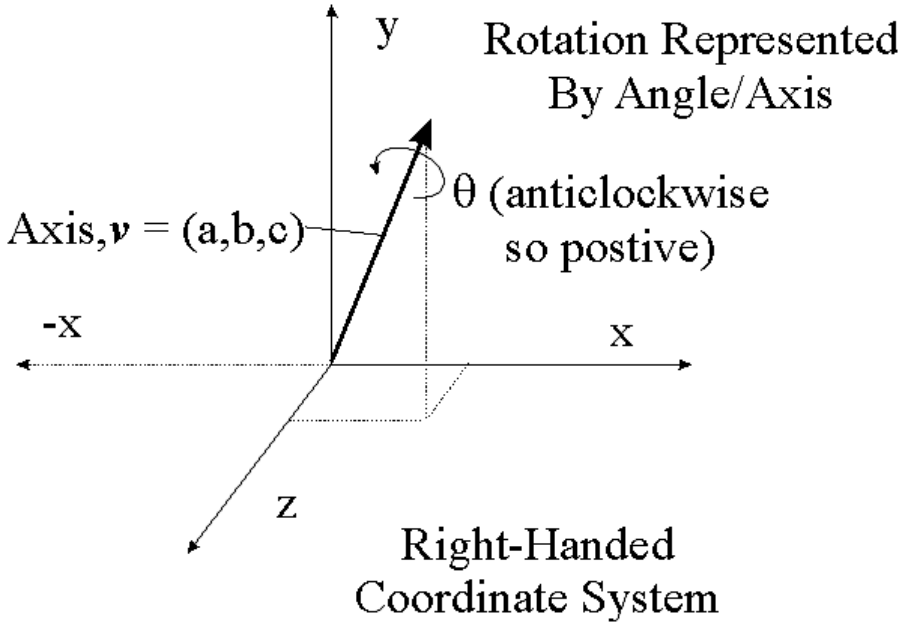


Figure 3.94: Axis/Angle rotation in right-handed system.

Intuitively, to covert the rotation in axis/angle format all we should have to do is change the axis about which the rotation takes place (it must take place around some axis!), and keep the angle the same (the amount of rotation surely does not change), with a minor caveat in that perhaps we will need to change the sign of the angle.

Our intuition is right here, the conversion of the axis is trivial, you just flip the x-component. As for the angle, the gotcha here is that now that were in a left-handed system, we will (almost certainly, according to the "Positive Rotation Rule) consider positive rotations to be rotating the other way. This means that we should flip the sign of the angle too:

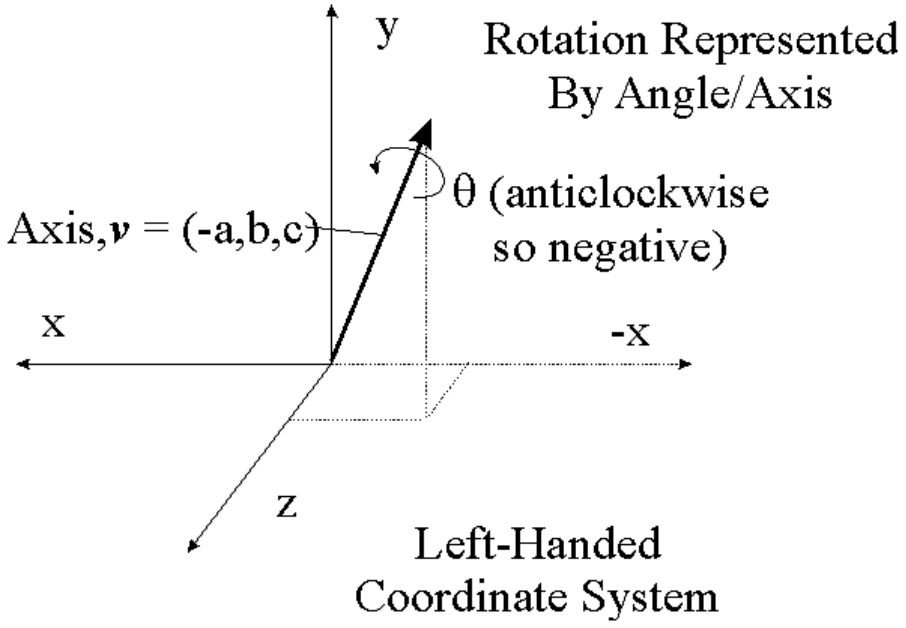


Figure 3.95: Axis/Angle rotation in left-handed system.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

In general, we can say if T is a transform with zero translational component you simply transform the axis by T:

converted axis = T*axis,

and then flip the sign of the angle *if T contains a reflection component*, such as when we change handedness by flipping X. Otherwise, for example in the case of a pure rotation where we are not changing handedness, we do not need to flip the sign of the angle.

Of course, wed like to avoid, in the case of a quaternion, actually extracting the angle, and the axis, which can be expensive.

Since a quaternion stores a rotation (theta,v) (say) as:

[cosine(theta/2), sine(theta/2) * v], then we wish to compute:

[cosine(theta/2), sine(theta/2) * Tv]

(Well deal with the angle flip shortly)

But since sine(theta/2) is a constant, we can move this into the computation sine(theta/2) * Tv and rewrite this as T * sine(theta/2)v. Thus we need only apply T to the imaginary component of the quaternion.

newImag = T * oldImag,

and the changing the sign of the angle, as discussed in the hkQuaternions subsection of the "Positive Rotation Rule Conversion" section only changes the real-component (i.e. the cosine(theta/2)). So if we need also to flip the sign of the angle, we do this by flipping the sign of the real component.

One way of determining whether there is a reflection component in the matrix if you do not already know (but you should), is to examine the determinant. If it is -1, then there is a reflection, and you will need to flip the angle. For example, here is a Havok function which will convert an hkQuaternion given a rotation matrix which may have reflection in it:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
void convertQuaternionToNewSpace(const hkRotation& rotToNewSpace, hkQuaternion &q)
{
 hkVector4 imag;
 imag = q.getImag();
 imag.setRotatedDir(rotToNewSpace, imag);

 q.setImag(imag);

 // Need to know whether there's a reflection here or not, so calculate determinant.
 {
  hkVector4 r0; r0.setCross( rotToNewSpace.getColumn(1), rotToNewSpace.getColumn(2) );

  hkSimdReal determinant = rotToNewSpace.getColumn(0).dot3(r0);

  HK_ASSERT2(hkMath::fabs(hkMath::fabs(determinant) - 1) < 1e-2f, "Matrix has scaling in it!");
   // If close enough to -1.
  if(hkMath::fabs(determinant + 1) < 1e-2f)
  {
   q.setReal(-q.getReal());
  }
 }

}
```

When To Convert

There are in effect two ways to sync your renderer/game engine with Havok, assuming that you need to do some data munging to swap between Left/Right handed, or even just to rotate you data so that your preferred canonical axis represents "up". You can either:

- Method 1.Export all your Havok data straight from the modeller in "modeller space", and ensure that at run time whenever you get or set Havok data, you apply the relevant conversion as specified above.
- Method 2. Apply the conversion as you export (just like you do for other data), in which case no run-time conversion is needed because Havoks data is already in the same space.

Note:

Note: Both these examples assume you are not changing the "Positive Rotation Rule" convention on export. If you are also changing this rule, you will have to apply the conversion process described in the section "Positive Rotation Rule Conversions".

It is far, far preferable to use method 2 because that way the export path is separated from your game runtime. It could be very confusing trying to view values in a debugger which were in two different "spaces", but which represented the same point. This will happen if you use method 1, so it is to be avoided.

Unless you have an additional "munge" transform which you concatenate into e.g. your camera matrix, using method 2 will ensure that Havok data and your game data are both interchangeable. For example, after setup, if using Havok to simulate the world, the transforms of all rigid bodies can be extracted directly for display (assuming you use the "common" matrix format if you use another format you will have to do some additional conversion).

Putting It All together: A Concrete Example

To put all this theory to the test, lets attempt a concrete example which requires us to convert data from

a right-handed system (in this case 3DSMax), to a left-handed system (in this case DirectX), including full integration with Havok.

There is an example app called directxexample (located in the demo/simpleapps directory) which implements the conversions discussed here. It is a DirectX application, integrated with the Havok SDK, and the Havok Visual Debugger. You will need DirectX installed to run the app. You will also need to get the Visual Debugger Client running to view the Havok "world" being simulated. It is necessary to get this example up and running so that you may follow the description below.

To build the DirectX app, open the .dsw/ .sln provided, and compile.

To get the visual debugger up and running, see the Advanced Profile Analysis section.

The example we shall use has 3DS Max as its modeller. There are two MAX files, hand.max and rotatedhand.max, which illustrate the mesh, and "scene" we shall use respectively. If you have 3DS Max you can open them to view the data, but if not the screenshots below illustrate the setup completely.

The Problem

The problem setup is as follows:

We have a "right-handed modeller" (in this case 3DS Max). If you open up the file hand.max, or examine the screenshot below you can confirm this. You can see that it is right handed because the World Coordinate Axes in the figure below are displayed in the bottom right of the four view panes. They use the standard colouring convention:

R,G,B = X,Y,Z

So X-axis is Red, Y-axis is Green, Z-axis is Blue. They are also labelled. You can clearly see that the axes form a "right-handed" coordinate system, with X and Y in the plane of the hand, and Z pointing up (like the thumb). Note that the hand mesh itself is not aligned to illustrate this point, what is important is the coloured coordinate triple should be obviously "right-handed".

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

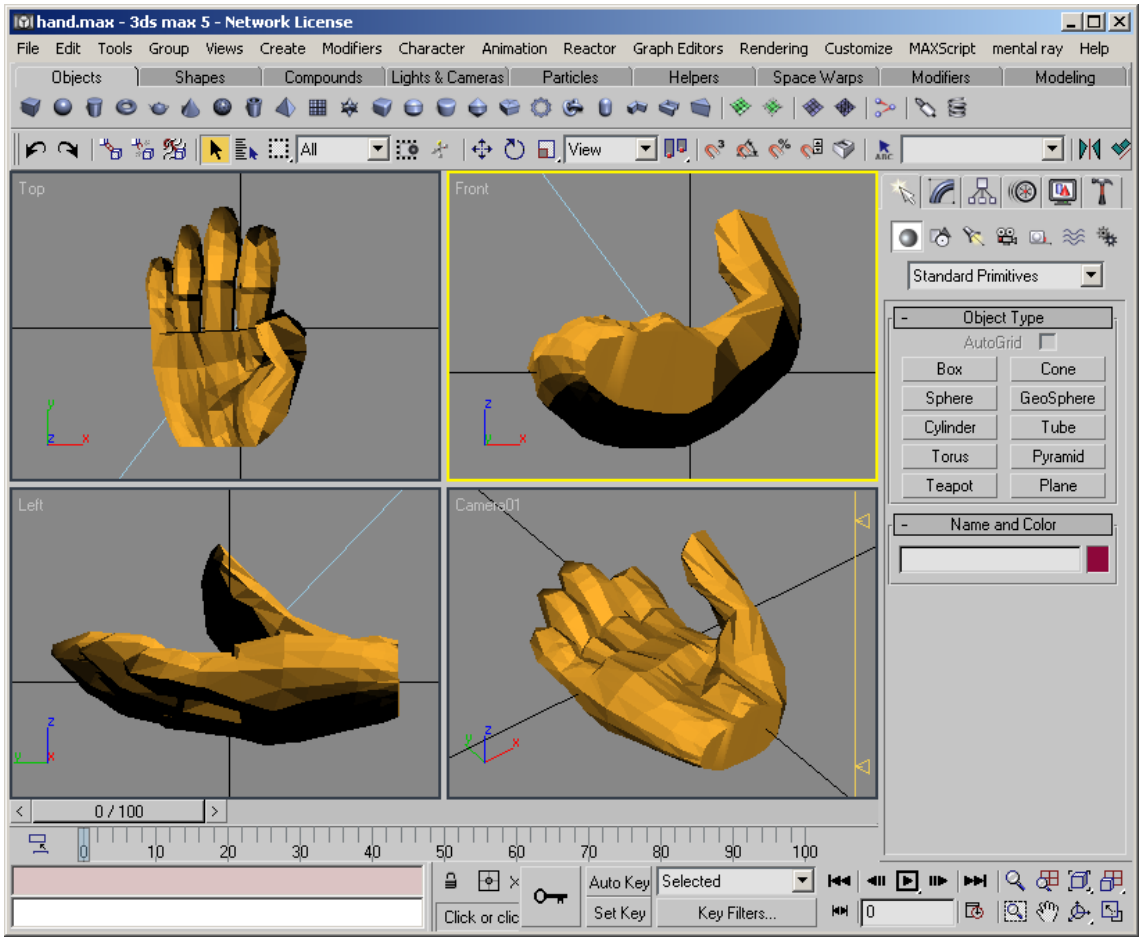


Figure 3.96: The "Hand" Mesh in Max (hand.max)

So, we have established the handedness of the modeller in question. Let's now set up a scene we wish to export using some obviously asymmetric data. The "hand" mesh below will be the object we place somewhere in the scene. The four panes show it from top/front/left and user camera views. Its pivot (centre) is currently over the origin of the World (i.e. the mesh object currently has the identity transform). It's clearly a "right-hand", so later when we compare other views of this mesh in both the DirectX window, and the Visual Debugger window, we can quickly sanity check things by confirming that what we see is still a "right" hand, not a "left" one!

If we take this mesh and both translate and rotate it in the world, we arrive at the scene stored in rotatedhand.max, and illustrated by the image below.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

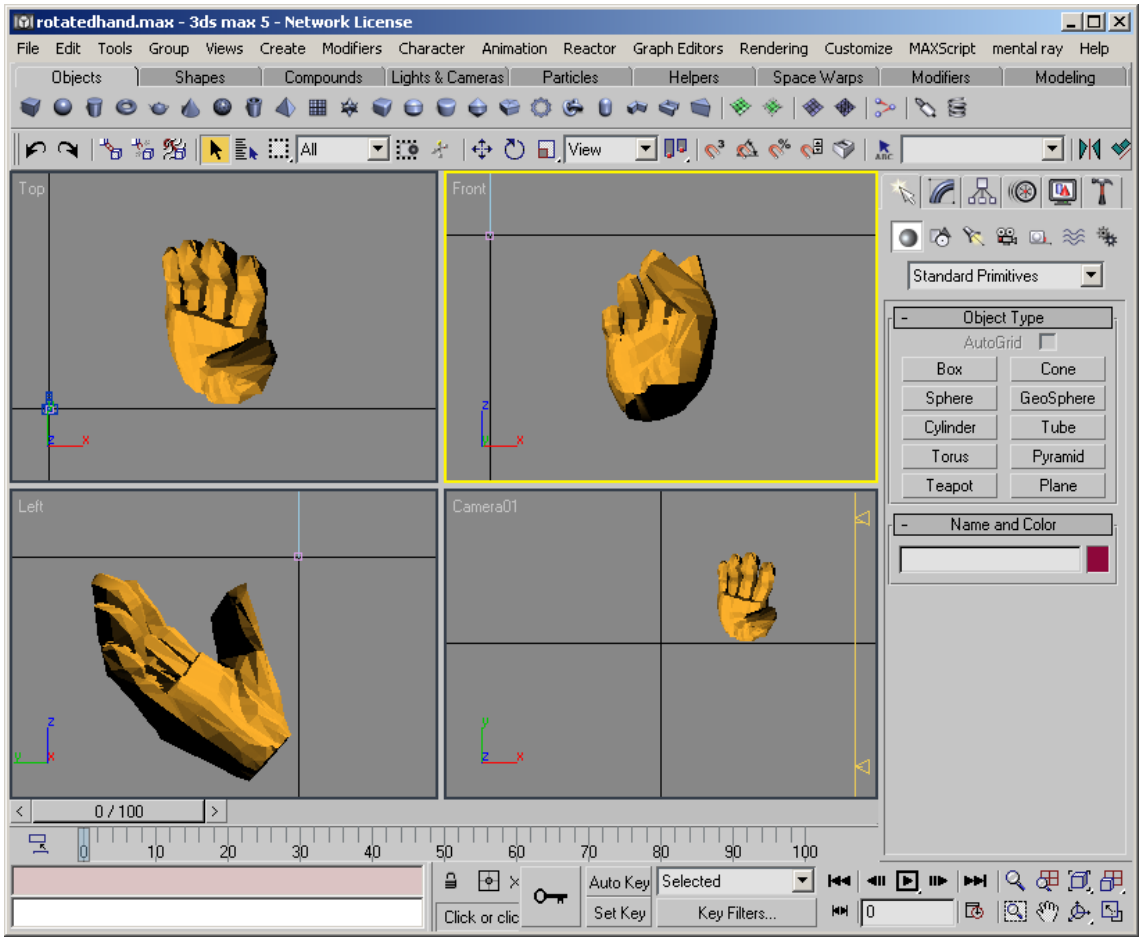


Figure 3.97: Initial Scene In Max (rotatedhand.max)

The important thing to note here is the bottom right-hand pane, labelled Camera01. This is the view of the world we shall attempt to export and match up in our DirectX renderer. We have set up our camera to point down the Z-axis, with the camera's "up" direction being positive Y.

Now we are going to export this data so that we may read it back in and display it using DirectX. DirectX is a "left-handed" system. Hence we will have to do some data conversion. For example, were we simply to export the mesh data directly, it would appear "backwards" in our DirectX display window, even if we positioned and rotated this correctly. This would be immediately obvious to us in this case because the mesh is the mesh of a person's right hand. Displaying this data in a left handed system would mean the mesh would look like a left hand!

"Mirror" Conversion

There are several ways to convert left handed data to right handed data. The example in the "Conversions" section above uses a "Mirror X" method. Another way to do it might be to "Swap" Y and Z values, which we shall call the SwapYZ method. These are but two examples, your own toolchain may have a different method, but in any case the "Conversions" section details how you can express this "toolchain transformation" in a Matrix format, which allows you to convert both positions and orientations. The DirectX application example illustrates both "MirrorX" and "SwapYZ" methods/toolchains, as is documented in the .cpp. It defaults to the "MirrorX" toolchain, and requires a recompile to use the "SwapYZ" toolchain. We describe the "MirrorX" toolchain example first.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

As described in the section 4.2.1, three pieces of data must be mutated to effect the conversion: the mesh data, the translation, and the rotation. We deal with the mesh data first. The conversion we are performing merely flips the sign of the X component. This can be easily done before export of the mesh to a DirectX .X file using the Mirror tool. The resulting file, called handMirrored.X is located in the application folder.

The rest of the conversions will be done on initialization of the DirectX app, and will all use the "toolchain conversion transform", stored as g_toolchainConversionTransformCM in the code. This is the key piece of data in the conversion process. In the case of the "MirrorX" conversion, the transform is:

F =

-1 0 0

0 1 0

0 0 1

which is stored in column major format as

```
float g_toolchainConversionTransformCM[16] =
{
  -1, 0, 0, 0,
   0, 1, 0, 0,
   0, 0, 1, 0,
   0, 0, 0, 1
};
```

and hence can be read directly by (or written directly into) a Havok hkTransform or a DirectX D3DXMATRIXA16 (which inherits from the D3DXMATRIX structure). We already derived this transform in Sections 4.2.1 and 4.2.2. The mesh data output into handMirrored.X has (implictly) already had this transform applied to it, i.e. all the vertices have been multiplied by this transform. It remains to apply this transform to all other data, namely the position and orientation of the object, as well as the position, direction and up vector of the camera, and (for simulation purposes), the direction of gravity. There is no actual toolchain here (we use no "exporter"), so we read each piece of data directly from examination of the MAX scene as follows:

We extract the modeller position/orientation of the object as a transform using the MAXScript Listener, as detailed in the code comments:

```
// Generate Object transform in Max from 'print selection[1].transform'
// (matrix3 [0.833395,-0.374604,0.406355] [0,0.735247,0.677799] [-0.552678,-0.564874,0.612751] [1,0.5,-0.5
  ])
// This is the transform given by the object in our toolchain, in column major data
// so, for example, the translation is given by the 13th, 14th and 15th floats.
// This is the "common" matrix format, referred to in the docs.
float g_objectInModellerCM[16] =
{ 0.833395f, -0.374604f,  0.406355f,  0.0f,
  0.0f,       0.735247f,  0.677799f,  0.0f,
 -0.552678f, -0.564874f,  0.612751f,  0.0f,
  1.0f,       0.5f,      -0.5f,       1.0f
}
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

We also extract the camera:

```
// This is the specification of the "camera", which we have extracted from Max by examining the
// Camera using
// print \$Camera01.Transform.position
// print \$Camera01.Target.Transform.position
// and assuming Y is up (which is also the direction "gravity" is expected to be defined in the scene).
float g_CameraEyeInModeller[3] = {0.0f, 0.0f, 5.0f};
float g_CameraLookAtModeller[3] = {0.0f, 0.0f, 0.0f};
float g_CameraUpModeller[3] = {0.0f, 1.0f, 0.0f};
```

And we "know" the gravity direction we want to be the same as the camera "Up" direction (but of opposite sign!), namely in the -Y direction:

```
// This is the specification of the "gravity" direction, which we have extracted from Max by examining the

// camera "Up" direction - usually your camera "Up" will be the same as the direction of gravity for
// obvious reasons.
float g_GravityModeller[3] = {0.0f, 1.0f, 0.0f};
```

If you examine the code you will see that everywhere a Havok object is set up, the "toolchain conversion transform" is applied first, exactly as described in Section 4.3 "When To Convert", i.e. using Method 2.

The net result is that when the app window comes up you should see the following:

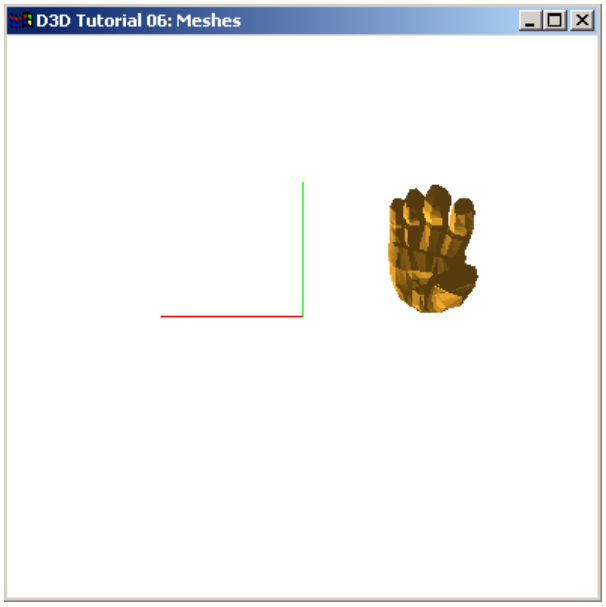


Figure 3.98: DirectX Window - "Mirror X" Conversion

Here the window looks identical (except for aspect ratio, and lighting) to the bottom right pane of the Max screenshot image "Initial Scene In Max (rotatedhand.max)", which demonstrates that we have correctly converted our "right-handed" data to be displayed in a "left-handed" system. I say "identical" but of course there is one small difference because the canonical axes are displayed in both cases, and you can quite clearly see that in Max the X-axis (in red) points to the right, whereas in the DirectX window it

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

points to the left, confirming that "MirrorX" conversion has been performed.

Now the app has been set up to continuously update the transform of the hand mesh from an associated rigid body, so we can be sure that not only was Havok "set up" correctly, but also it really is running "in the same space" as the renderer. To verify that Havok is indeed correctly "synced", we can bring up the Visual Debugger Client, and take a peek at the physics simulation happening in the app. I presume that the user had read the Advanced Profile Analysis section.

First we run the VDB, and connect to the machine running the DirectX app. We should see something like this:

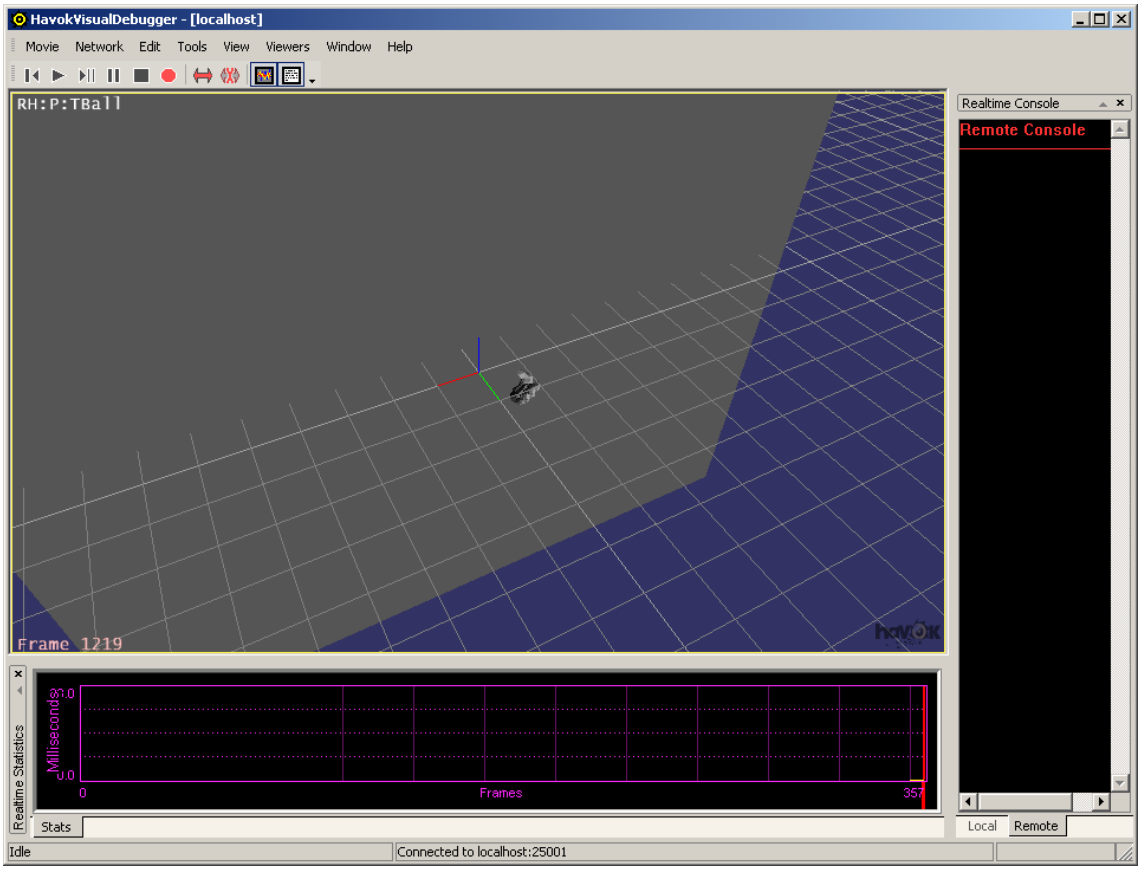


Figure 3.99: Visual Debugger Client Window - Initial View

This does not look like the DirectX window because we have not yet set and of the rendering options for the VDB (which is highly generic, and can be customized to render in many different ways). So now we set up the visual debugger to reflect the renderer settings of the app. There are three things to do here:

- Set the renderer to be left-handed. From the View menu at the top select

  Axis - Use Left Handed System

- Set the camera to be the same (user) camera set up by the app. As well as setting up the DirectX camera using the "converted" camera settings in the method SetupMatrices(), the app also sets up a VDB camera with identical parameters in the method StepVisualDebugger(). We can select this camera to be used by the VDB by going to the View menu and selecting

  User Cameras - DirectX Example

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- (For tidiness only) Set the "Up" direction (used to set the "world plane grid") to be Y (gives an idea of what direction really is "Up", for example we expect gravity to act at right angles to this plane). Set this by going to the View menu and selecting

  Axis - Y Up

Having done these three things, the VDB should now look like this:

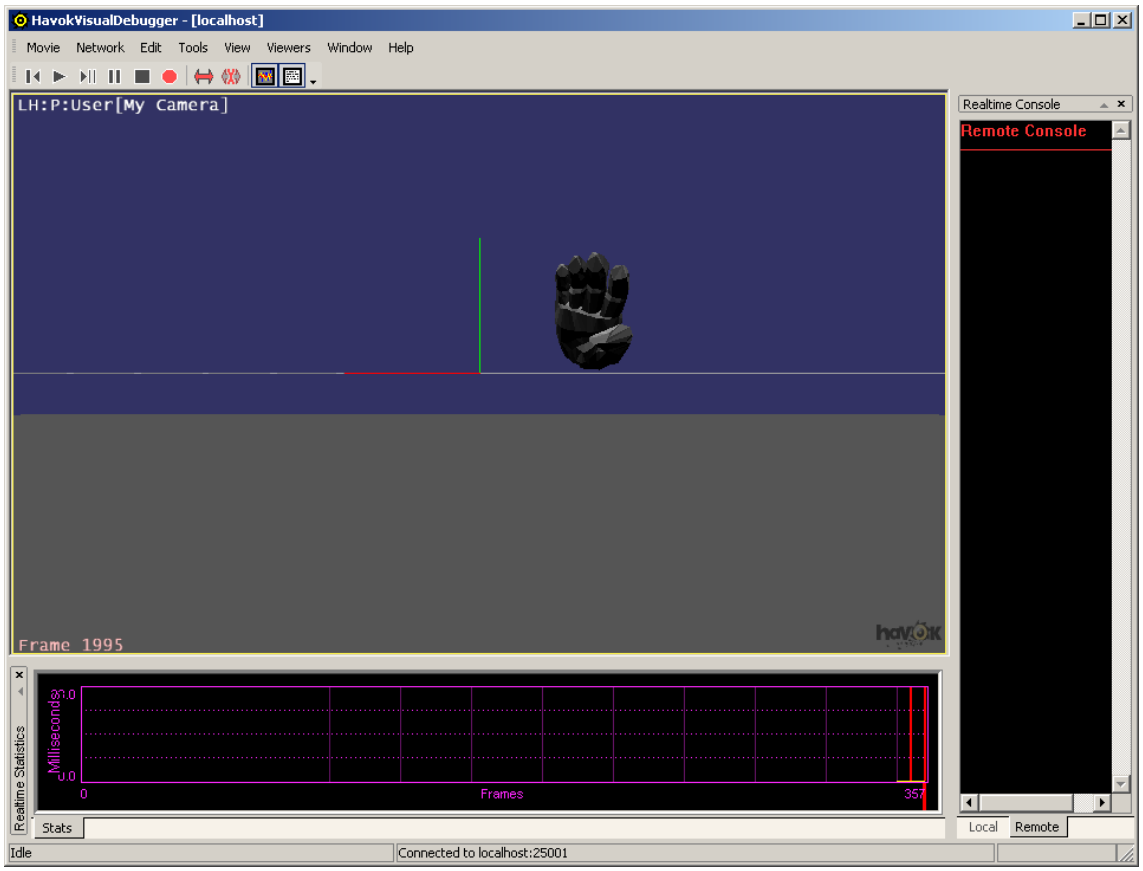


Figure 3.100: Visual Debugger Client Window - "Mirror X" Conversion

Again this is exactly the same (except for aspect ratio, and lighting) as the DirectX example, and if you use the mouse picking to grab and move the hand in the VDB you should see the hand in the DirectX example move in exactly the same way, proving that you have correctly set up all the Havok data to match the DirectX view. In fact you'll notice an grey block below the hand which was not visible in the DirectX window. This is a separate Havok rigid body which was added after the hand was, but no visual representation was added to the DirectX display, and the reason for doing this was so that you can turn on gravity in the app (set GRAVITY to a non-zero negative value and recompile), and see the hand fall and collide with the box, thus demonstrating that the physics is also fully working!

"SwappedYZ" Conversion

At this stage I hope you are convinced that the theory can be put into practice and produce the expected results. As a final example we can set the app to run using the alternative "SwappedYZ" toolchain, by recompiling the app with EXAMPLE #defined to be YZ_SWAPPED_EXAMPLE. In this case we convert from right to left-handed by swapping the Y and Z values. This transform can be expressed in matrix form as

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

SXY =

1 0 0

0 0 1

0 1 0

which is stored in column major format as

```
float g_toolchainConversionTransformCM[16] =
{
   1, 0, 0, 0,
   0, 0, 1, 0,
   0, 1, 0, 0,
   0, 0, 0, 1
};
```

The mesh file is already exported as handSwappedYZ.X.

If we run this example, the DirectX window looks like:

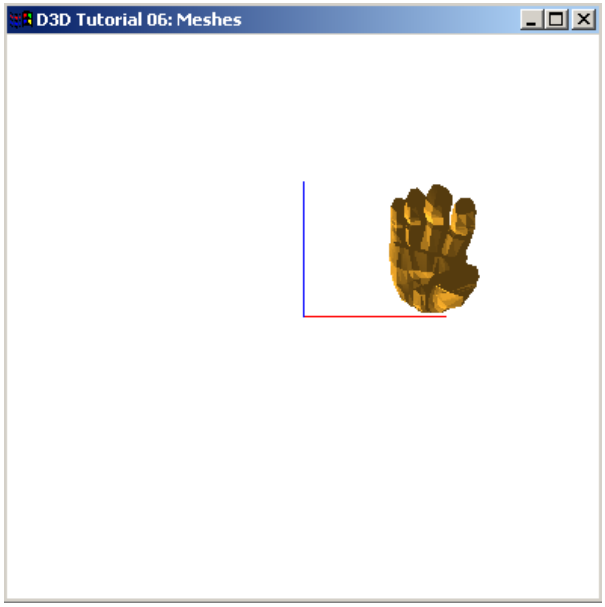


Figure 3.101: DirectX Window - "Swap Y and Z" Conversion

and we can see that again the hand is in exactly the right place, i.e. it looks identical (except for aspect ratio, and lighting) to the bottom right pane of the Max screenshot image "Initial Scene In Max (rotatedhand.max)". Note that here the X-axis (red) still points to the right as it has not been changed by the transform, but that Y (green) and Z (blue) have been exchanged (Y now points directly out of the screen, so is unfortunately not very visible!)

Once again we double-check that Havok has been correctly initialized by hooking up the VDB. You must re-select the user camera if you have not closed down the VDB to ensure it picks up the new camera values. Note that this time our renderer has Z as "Up", so we reflect this in the VDB by right-clicking

and selecting

Axis - Z Up

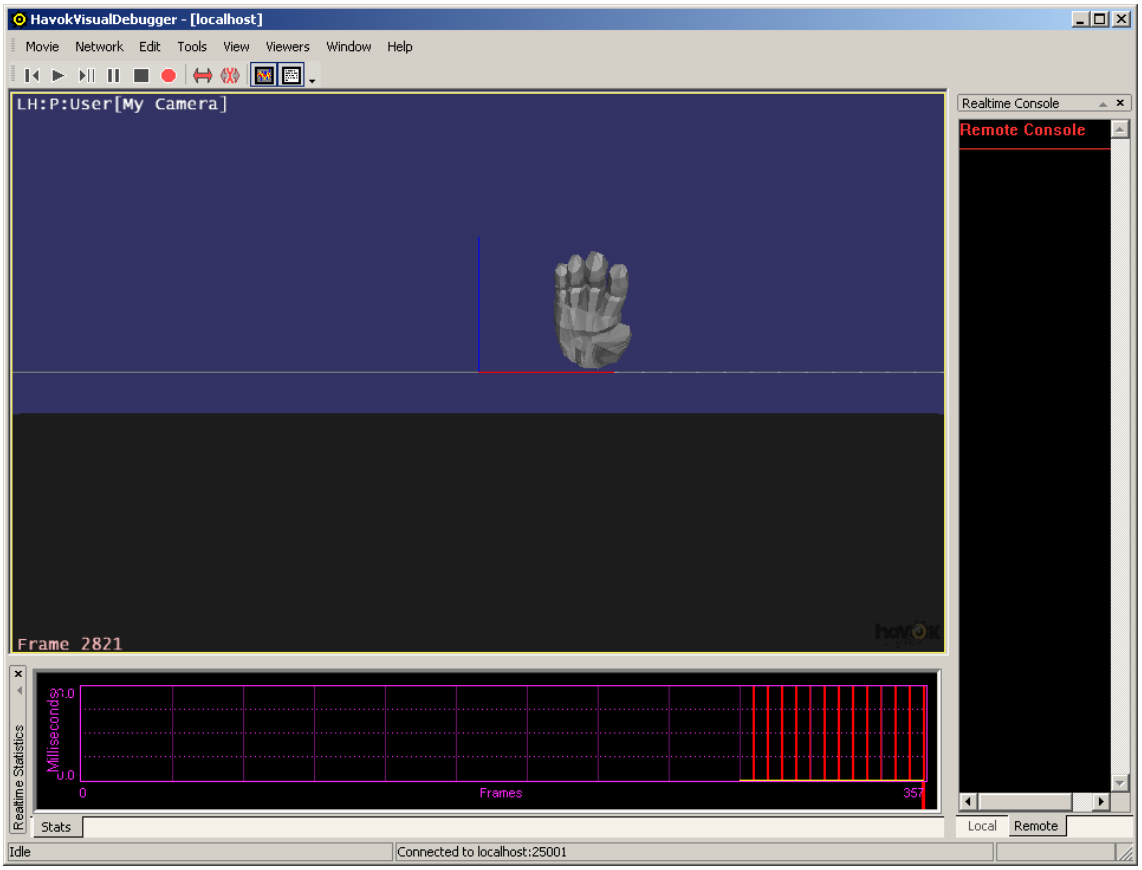


Figure 3.102: Visual Debugger Client Window - "Swapped YZ" Conversion

You can see that this correctly reflects the DirectX window view as expected. Again you may wish to turn gravity on/off, and /on grab the hand in the VDB to see that it updates correctly win the DirectX window.

These examples together with the detailed explanations of the transform derivations in previous sections should enable you to correctly adapt Havok to any toolchain conversion process.

3.10.2.5 Appendix A : Some references, and relevant quotations.

DirectX:

Left-handed coordinates:

http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dx8_vb/directx_vb/graphics_intro_8w8j.asp

"Microsoft®Direct3D®uses a left-handed coordinate system"

Left-hand rule (clockwise):

http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dx8_vb/directx_vb/graphics_intro_8wqb.asp

"Angles are measured clockwise when looking along the rotation axis toward the origin"

OpenGL:

No handedness.

http://www.opengl.org/documentation/specs/version1.4/glspec14.pdf

p 252. Appendix B, #15 "OpenGL does not force left- or right-handedness on any of its coordinates systems."

Right-hand rule:

http://www.opengl.org/documentation/specs/version1.4/glspec14.pdf

p 35, Section 2.10 Coordinate Transforms. "The computed matrix is a counter-clockwise rotation about the line through the origin with the specified axis when that axis is pointing up (i.e. the right-hand rule determines the sense of the rotation angle)."

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Chapter 4

Havok Animation

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.1 Introduction

Welcome to the Havok Animation Manual. Havok's Physics and Animation solutions form an integrated game animation, dynamics and collision detection pipeline providing a complete solution for a game's animation and dynamics requirements. Havok Animation is a fully featured game animation runtime with customizable asset pipeline.

Havok Animation has a number of key features:

- Highly optimized and feature rich skeletal animation runtime for games on all Havok supported platforms. The animation system features a state of the art compression pipeline enabling more animation data to be packed into the available memory on each of the supported platforms.
- A customizable asset pipeline with runtime class reflection model for on target tools creation.
- Full integration with Havok Physics for character dynamics and control.
- Full integration with Havok Behavior for interactive character behavior authoring.

Together with Havok Physics and Havok Behavior, Havok Animation empowers users with a complete solution for next generation character animation and dynamics, including runtime animation sequencing and productivity tools.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.2 Architecture Overview

The Havok Animation system uses the existing Havok runtime platform, including platform optimized math libraries, the `hkbase` library (for object and memory management), serialization and the Havok Exporter framework, providing both a complete end to end animation pipeline and an extensible framework to facilitate embedding the animation runtime and tools into an existing game production pipeline.

4.2.1 Object Overview / Glossary

This is a brief description of some of the more important classes in the runtime system. We use the mnemonics AC and AR to indicate a class that is typically allocated at construction time (load time) or runtime (stack / heap allocated). Classes marked with AC are typically shared. Abstract classes are *emphasized*.

4.2.1.1 Animation Classes

hkaBone - AC

- Describes a bone as it appears in the tool chain i.e. has a name, and rigging information. This is not the efficient structure we use at runtime to construct and blend pose information.

hkQsTransform - AC, AR

- This represents a decomposed 4x4 transform. It explicitly stores translation, rotation (as a quaternion) and scale along X, Y and Z. When combining or sampling animations we force them to drop down to this structure.

hkaPose - AR

- A class which holds a syncronized local and model space pose (as arrays of `hkQsTransform`s), and an array of float slots (as an array of hkReal). Allows you to work in either local or model space seamlessly and efficiently - it performs lazy updating.

hkaAnnotationTrack - AC

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- A tag for an animated variable (usually a `hkQsTransform`). Contains a name and a set of time stamped annotations.

(Hierarchy) - AC

- Not actually a class in the system. See `hkaSkeleton` definition below. Defines a parenting relationship between bones. A root bone has no parent. Currently represented as an array of 16 bit integers.

hkaSkeleton - AC

- A set of hkaBones, a reference pose (e.g. T-Pose) and a Hierarchy, and a set of float slots (char*). Either one of the bone or float slot sets may be empty. May have multiple root bones.

hkaSkeletalAnimation - AC

- An (abstract) interface.
- Represents a pose that changes over time. Has a finite period (in seconds). Can be sampled continuously. Can exist in any space for the bone data, but is usually in local space.

hkaAnimationBinding - AC

- Provides a mapping from animation tracks indices to skeleton indices for both bones (hkQsTransforms) and float slots (hkReal). Useful for partial animations (e.g. Upper body only).

hkaAnimationControl - AR

- An (abstract) interface.
- Provides runtime animation playback control. Controls the local time and overall weight for a playing animation.

hkaAnimatedSkeleton - AR

- Represents the runtime instance of a specific character. Keeps track of all active animations running on this instance. Samples and blends all active animations and extracted motion / locomotion. Provides support for per bone feathering and masking.

hkaAnimatedReferenceFrame - AC

- An (abstract) interface.
- This interface exposes the motion information for an animation, which is normally 'extracted' from e.g. the root bone.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.2.1.2 IK Classes

hkaCcdIkSolver - AR

- This is a general solver for chains of bones. It is an iterative solver. It has no support for limits.

hkaFootPlacementIkSolver - AR

- This is a specialized analytic IK solver designed to handle foot placement. It uses raycasting to find ground height information. Internally it uses the hkaTwoJointsIkSolver.

hkaLookAtIkSolver - AR

- This solver operates on a single bone and orients the bone so it faces a specified target. The solver has full support for cone limits.

hkaTwoJointsIkSolver - AR

- This analytic solver is designed to handle limbs such as legs and arms. It assumes the bones represent a ball and socket and hinge construction e.g. Hip and Knee or Shoulder and elbow. It has support for hinge limits. It has weighting support for gracefully easing in / out the result.

hkaThreeJointsIkSolver - AR

- This analytic solver is designed to handle three-jointed limbs. It assumes the first joint is resented by a ball and socket and that the second and third joints are represented by a hinge. It has does not support hinge limits and has no weighting support.

4.2.1.3 Deformation Classes

hkaMeshBinding - AC

- Contains the bind pose required for skinning. Provides a mapping from the bone indexes in a skinned mesh to the bones of a given skeleton. Useful when the tool chain has to split meshes (e.g. For Hardware rendering with a bone limit).

hkaMorphingDeformer - AR

- An (abstract) interface.
- Takes 2 input streams (Vertex Buffers) and interpolates to an output stream. The interface forces the user to bind the streams to the deformer first - this allows us to do error checking and set up the streams for efficient processing. Can bind once deform often (if binding is a slow process).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- An (abstract) interface.
- Takes a single input stream and skins to an output stream. The interface forces us to bind the streams to the deformer. User supplies input matrices when deforming. Bind once deform often.

4.2.2 Export Path

A critical component of any game production pipeline involves extracting information from the content creation tools (in particular the modelers), serializing this to an extensible stream format and efficiently loading, storing and accessing these assets at runtime. Havok's serialization framework is designed to integrate easily into existing pipelines, and Havok Animation provides a set of modeler export utilities, file formats and APIs which are fully customizable and extensible and which integrate seamlessly with the physics product.

4.2.2.1 Design Philosophy

Our philosophy is to provide a set of very simple components and formats that fit easily into your existing pipeline. These are built on our robust serialization infrastructure and are provided with full source. Our asset processing is carefully divided into modeler specific and modeler independent sections. This separation allows you to extend and enhance this pipeline quickly without having to learn any specific vendor API (including ours!). Once extended you leverage the full power of the system and get features like version control and batch processing for free.

In later releases we intend to extend this infrastructure to on-target asset viewing and tweaking.

4.2.2.2 Serialization Infrastructure

Our core serialization infrastructure uses standard object reflection techniques. Since serialization is common to both Havok Animation and Havok Physics, a full explanation of how it works can be found in the Common Havok Components chapter of this manual.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.2.2.3 Thin Export Utilities and the Filter Pipeline

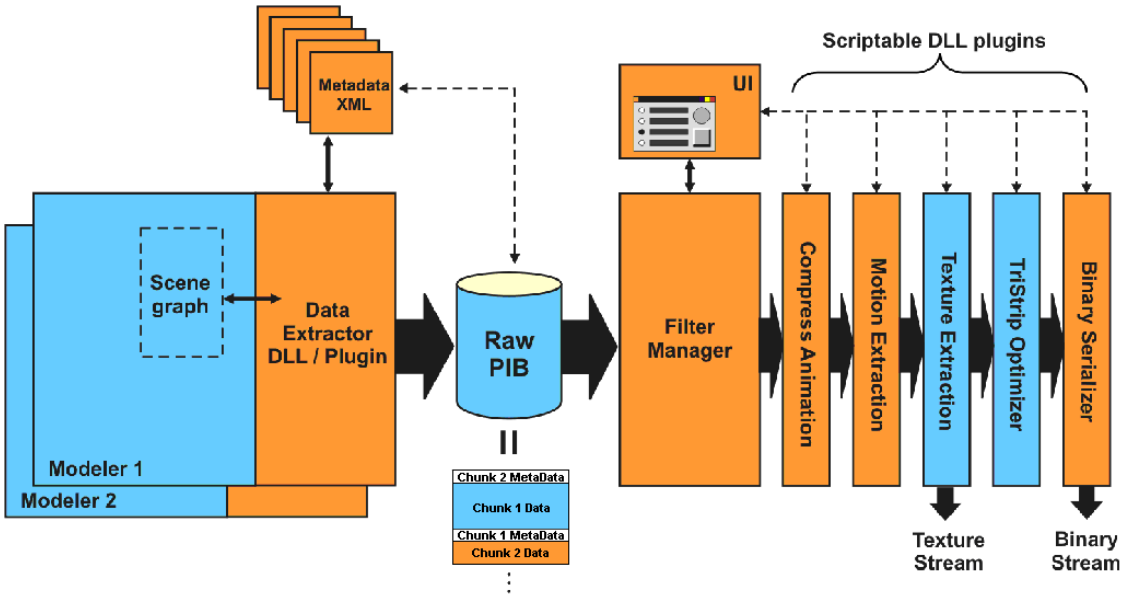


Figure 4.1: The Havok Asset Filter Pipeline

The asset pipeline that ships with Havok Animation is made up of a number of functional components as illustrated in the diagram. We provide "thin" export utilities for 3DS Max and Maya. These utilities have little user interface associated with them. Their job is to walk through the scene graph of the tool and extract specific data found into a single data stream. This data stream is then passed to pipeline of filters. Each filter in the pipeline can modify extent or remove data from the stream. Typical filters include scene transformation, rig and animation extraction or compression, data retargeting for a specific console or even data previewing. The section entitled 'The Havok Filter Manager' describes each of the current filters in detail.

The specific set of filters that process a given asset are interactively composed by an artist or technical director. This set of filters, together with any specific options, is automatically saved with the original assets. When the asset is next edited it retains the last set of filters and options. This simplifies both batch processing and asset maintenance. The filter manager also allows the user to load/save options from/to an external location.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.3 Animation Runtime

4.3.1 Animation Features

The Havok animation library provides a full runtime animation solution. Major animation runtime features include:

- Playback Control
- Sampling and Blending
- Compression / Decompression
- Motion Extraction
- Inverse Kinematics
- Mirroring
- Pose Representation and Transform Utilities
- Havok Physics Integration (see the Physics and Animation Integration chapter)
- Serialization and Tool Support (see the Havok Content Tools Chapter)

4.3.1.1 Design Philosophy

Havok's animation runtime is designed specifically to address the growing difference between CPU performance and memory bandwidth. During the design we made the following tradeoffs:

- Where feasible the system trades CPU performance for memory savings.
- Data pipeline interruptions are avoided where possible ; you won't see many callbacks in the API.
- We maximize shared data coming from the tool chain and minimize our runtime heap allocations.
- The user will want complete control over when and where animations are sampled and blended we impose no scheduling on the animation pipeline and leave that to your game loop.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.3.2 Animation Runtime Modules

The figure below gives an overview of the modules in the animation runtime.

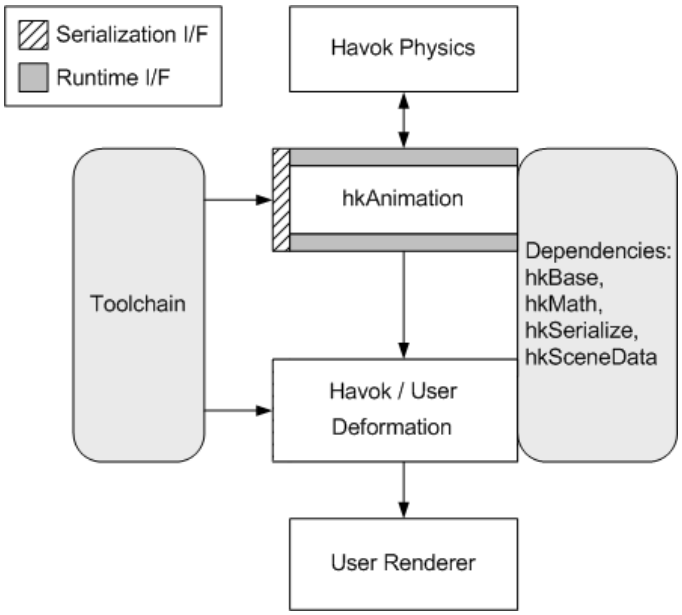


Figure 4.2: Runtime Animation Modules

Most object instances in the animation system can be shared. Animation, Rig and Mesh data, together with appropriate binding information come from the tool chain, through the serialization interface, and can be reused throughout your game. We have tried to keep the per instance data required for each character to a minimum. Only additional objects that deal with animation sampling (a control) and blending (an animated skeleton) are needed to have a fully animated character in your runtime.

The diagram below shows the most important relationships between classes in the animation runtime.

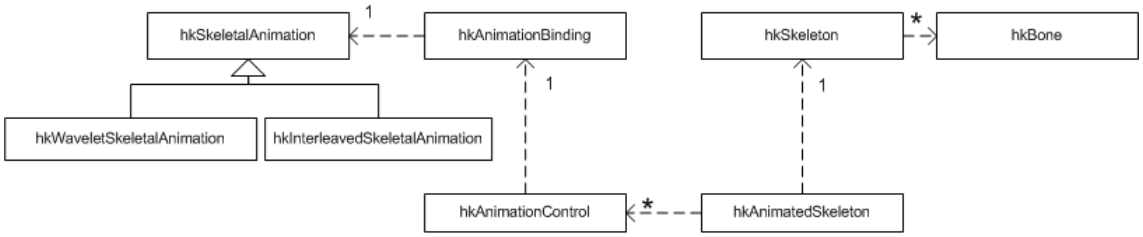


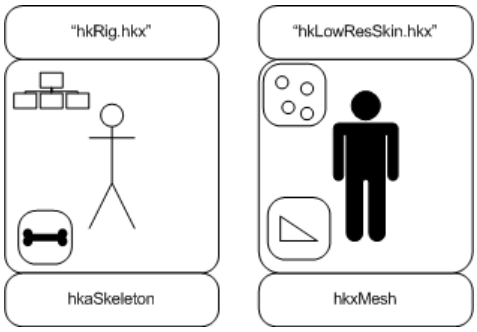
Figure 4.3: Animation Runtime Class Relationships

Only `hkaAnimationControl` and `hkaAnimatedSkeleton` are allocated at runtime. All other instances are created at load time and shared. The next sections go through each of these classes and their interaction in detail.

4.3.3 Dataflow Example

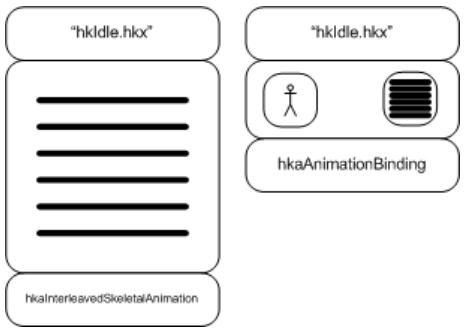
To illustrate how the various loaded and runtime-allocated objects can be used we'll walk through an example. The source code for this is in the *Demo\Demos\Animation\Api\MeshAndDeformation\Skinning* demo.

Initially we load a both a rig (`hkaSkeleton`), and a mesh (`hkxMesh`):



These stay in main memory as long as the character is being displayed / used. The `hkxMesh` represents a bind pose mesh. It contains a vertex buffer (shown as dots) and an index buffer (shown as triangles). The `hkaSkeleton` contains bones, a hierarchy, and bind pose info.

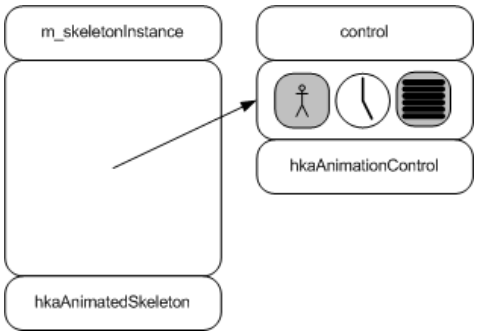
Next we bring the required animation (`hkaSkeletalAnimation`) and the binding (`hkaAnimationBinding`) for the skeleton into main memory:



The binding shows how to map the animation to the target skeleton. Both of these reside in memory as long as this animation is required.

We create a `hkaAnimatedSkeleton` ('m_skeletonInstance' on the heap). This is only required when an animation is playing on the character. Once created, we use the `hkaAnimationBinding` to create a `hkaAnimationControl` ('control' on the heap). We then add 'control' to the `hkaAnimatedSkeleton`.

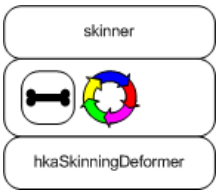
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Now 'm_skeletonInstance' can be queried for its current pose via sampleAndCombineAnimations(). It will access the control (which stores time and weight information), sample the bound animation, and blend this to produce a local pose for the skeleton.

To skin this result we copy an instance of the original mesh. The index buffer is shared with the original mesh. If were just going to skin and then render immediately we could create the copy of the vertex buffer on the stack.

Finally we create a deformer on the stack and bind it to the new output vertex buffer for the mesh instance and the original bind pose vertex buffer. The binding process may do complex things like download the vertices to the graphics card or vector unit. We take the sampled pose (from the `hkaAnimatedSkeleton`), convert it to world space, and combine its matrices with the original Skeleton inverse bind pose matrices (all locally on the stack). We check the mesh binding to arrange these matrices and then pass them to the hkaSkinningDeformer. The output vertex buffer is deformed and ready to be rendered.



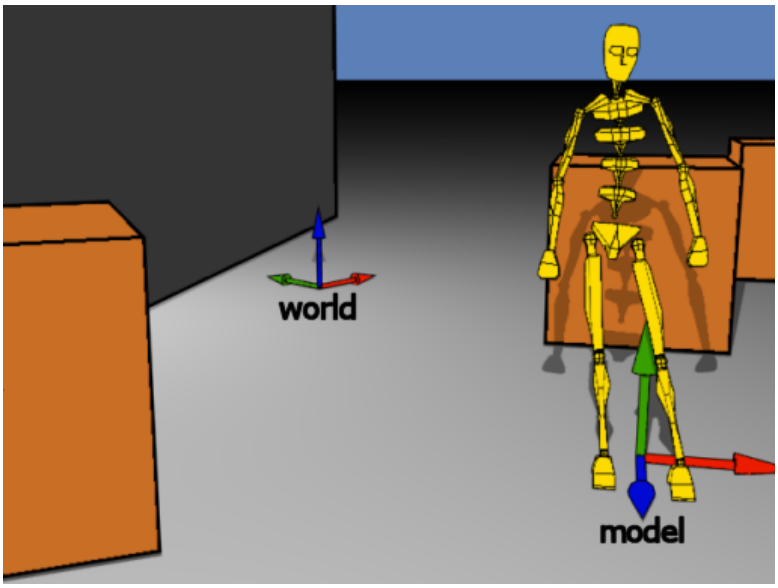
For each character we only require heap memory for the `hkaAnimatedSkeleton` and the space for each active controller. All other data is either temporary or shared (assuming we dont need to store the output vertex buffer).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.3.4 World, Model and Local Spaces

Throughout this documentation we will be using terms like World Space, Model Space and Local Space to describe transforms and poses. In this section we are going to describe the convention used to name those spaces.

4.3.4.1 World and Model Spaces



World space specifies "where the world is", including orientation of the world (what direction is up, for example). At any time, since there is usually only one world, you only have one world space in your application ("the world space").

Model space is specific to a character in the application, and it specifies "where the character is", including orientation (what direction is up, forward, right, etc). Therefore, you have a model space for each character in the game ("the model space of the character").

Notice that those two spaces are decoupled; for example, in world space the up direction can be the Z axis, while in model space the up direction can be Y instead (like in the diagram above, where X,Y and Z axis are represented with red, green and blue). In order to place characters in the world, i.e, in order to specify "where the character is in the world", you need to describe the model space defined in respect to the world space. This is what we call the world-from-model transform (the `worldFromModel` transform[1]).

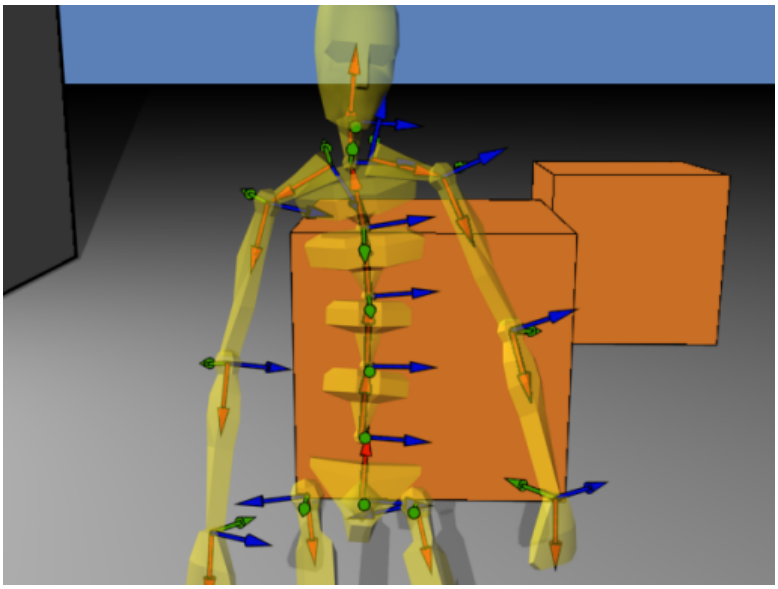
The world-from-model transform is usually tracked by the application. For example, it may be set based on output from the AI, the input from the user, the position of a physical character controller, the motion accumulated from the animations played on the character, or a combination of these.

[1] The world-from-model transform is also the model-to-world transform. We usually use the "from" convention since, in Havok, transforms are pre-multiplied and therefore the "from" convention reads naturally from left to right: It's easier to read b_from_a = b_from_c * c_from_a rather than a_to_b = c_to_b * a_to_c

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.3.4.2 Bone Space

Apart from the world and model spaces, when working with skeletal animations, there are also spaces associated with each bone in the skeleton:



For each bone B, the associated space is called the *space of bone B*. Very often this is also called the *local space of bone B*, although this is somewhat redundant.

4.3.4.3 World, Model and Local Space Poses

When working with animations, we are interested in describing the location and orientation of each bone, i.e., the space of each bone (the collection of all of these spaces is what we call a *pose*). As always, we need to define these spaces with respect to another space, by using a transform. The choice of the space determines whether our pose is a world, model or local space pose:

- If we define our bone spaces with respect to the world space, our pose is a *world space pose*: each transform in the pose is a `worldFromBone` transform.
- If we define our bone spaces with respect to the model space, our pose is a *model space pose*: each transform in the pose is a `modelFromBone` transform
- If we define our bone spaces with respect to the bone space of its parent bone, our pose is a *local space pose*[2] : each transform in the pose is a `parentBoneFromBone` transform. By definition, the parent space of the root (which has no parent) is the model space.

Notice that model space and local space representations of a pose contain the same information, just using a different representation - you can convert from a model space pose to a local space pose and vice versa without any extra transforms[3]. The hkaPose class allows you to work simultaneously in both

[2] A possibly more adequate name would be *parent space pose*. But local space pose is the most commonly used name.
[3] Although you will need the hierarchy (the index of the parent bone for each bone) to do this conversion.

representations. In order to convert from/to these representations to/from world space, you will need the `worldFromModel` transform.

4.3.4.4 Converting between Spaces

The `hkaSkeletonUtils` class in the `hkAnimation` library contains a number of very useful static methods for manipulating poses and skeletons. These methods allow you to:

- Convert poses to and from World, Model and Local space.
- Perform weighted blending of poses and bone chains (i.e. partial poses).
- Enforce bone constraints in `hkaSkeleton`s in both Model and Local space (currently the only bone constraint is 'lock translation').
- Find bone ids by name.
- Populate bone chains given the bone ids of the start and end bone (where the end bone is deeper in the hierarchy).
- Find all of a bone's descendants by id.
- Check that a `hkaSkeleton`'s animation binding is valid.

Please refer to the Havok Reference Manual or `hkanimation->rig->hkaSkeletonUtils.h` for more details.

4.3.5 Using hkQsTransform

In Havok Animation transformations are stored with a specialized representation, hkQsTransform[4]. hkQsTransforms have the following properties:

- They explicitly store three components : translation, rotation and scale
  - Translation is stored as an hkVector4
  - Rotation is stored as an hkQuaternion
  - Scale is stored as an hkVector4
- They define, among others, the following operations and elements
  - Product : hkQsTransform x hkQsTransform —> hkQsTransform
  - Inverse: hkQsTransform –> hkQsTransform
  - Identity value (neutral element for the product operation)
  - Interpolation : hkQsTransform x hkQsTransform x hkReal —> hkQsTransform
  - Application to a vertex : hkQsTransform x hkVector4 –> hkVector4
- They can be converted to and from hkMatrix4 (but, as we will see, with possible loss of information)

[4] The "Q" stands for Quaternion (used to store the rotation). The "S" stands for Scale, since these transforms can store scale.

hkQsTransforms can represent any orthogonal transformation, i.e., any transformation that does not involve skewing. The reason for that is that non-orthogonal transformations cannot be represented by a concatenation of translation, rotation and scale operations.

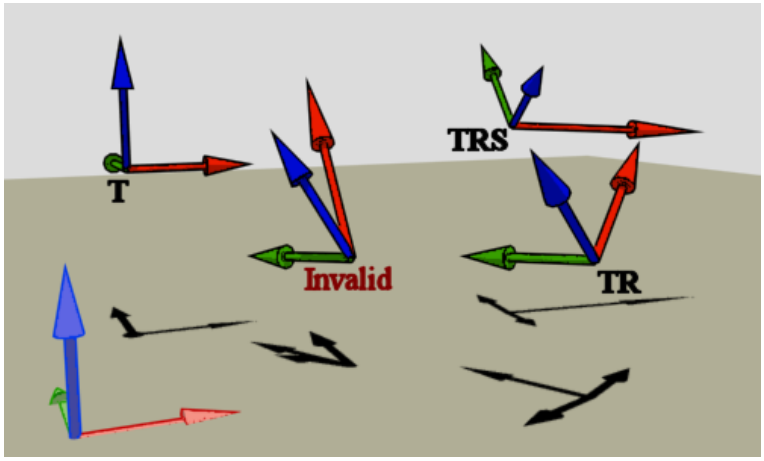


Figure 4.4: Example of different transformations (from a reference space, represented in lighter colors). Transforms T (translation only), TR (translation+rotation) and TRS (translation+rotation+scale) are all representable with hkQsTransform. However, the transform labelled in red cannot be represented with hkQsTransform since it contains skewing and therefore is non-orthogonal.

Notice that, since the set of valid hkQsTransforms (the set of all orthogonal transformations) is a subset of valid generic transforms (not necessarily orthogonal), care has to be taken when converting from one representation to the other, as that conversion may lose information (in particular about skewing)[5].

4.3.5.1 Multiplying hkQsTransforms

Translation and Rotation

If A and B are two hkQsTransforms, multiplying them yields a result C where the translation (vector) and rotation (quaternion) components are:

- Translation(C) = Translation(A) + [Rotation(A) applied to Translation(B)]
- Rotation(C) = Rotation (A) * Rotation (B)

These operations are consistent with considering an hkQsTransform as the composition of a Translation (T_) and a Rotation (R_) matrices:

- C = Tc * Rc = A * B = (Ta * Ra) * (Tb * Rb)

In this case, Tc and Rc (the translation and rotation components of the result C) can be calculated as above.

[5] In particular, many modelling tools allow the use of skewed transforms, since they use 4x4 matrix-based transformations. The Havok Animation Toolchain will always warn when performing conversions to hkQsTransform where the original 4x4 transform contains skew (as that skew will be lost).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Scale

Unfortunately, when introducing scale, the parallelism with matrices ends. This is because, if we consider the Translation, Rotation and Scale components to be matrices multiplied one after the other, then our result won't always be decomposable as an hkQsTransform; in other words, we would need a hkQsTransform C such as:

- C = Tc * Rc * Sc = A * B = (Ta * Ra * Sa) * (Tb * Rb * Sb)

The problem is that, in the general case, it is not possible to find the translation, rotation and scale matrices Tc, Rc and Sc that satisfy the equation above. This is because concatenating rotations and scale operations can produce skewing, which yields a non-orthogonal transform, not representable with hkQsTransform.

So we need to find an alternative interpretation of the multiplication operation which keeps the results inside the space of orthogonal transforms. To do so, we define the resulting scale component of multiplying two hkQsTransforms A and B as:

- Scale (C).x = Scale(A).x * Scale(B).x
- Scale (C).y = Scale(A).y * Scale(B).y
- Scale (C).z = Scale(A).x * Scale(B).z

That is, we define the resulting scale vector as the component-wise multiplication of the scale vector. By doing this we ensure that our multiplication operation works consistently, keeps the results inside the space of valid hkQsTransforms, and follows the expected properties of a transform multiplication operation.

4.3.5.2 Neutral Element (Identity)

The neutral element (the *Identity* hkQsTransform) is defined such as

- A * Identity = Identity * A = A, for any hkQsTransform A

From the definition of the multiplication described above, the resulting neutral element (identity) is:

- Translation (Identity) = (0, 0, 0)
- Rotation (Identity) = Quaternion_Identity = 1
- Scale (Identity) = (1, 1, 1)

4.3.5.3 Inverting hkQsTransforms

The inversion operation is consistent with the way hkQsTransforms are multiplied and the neutral element is defined, and as such the inverse A' of an hkQsTransform A is defined as

- Translation (inv(A)) = - (inv(Rotation(A)) applied to Translation(A))
- Rotation (inv(A)) = inv(Rotation(A))
- Scale (inv(A)) = 1 / Scale(A) [for each x, y and z component]

4.3.5.4 Interpolating hkQsTransforms

```
void hkQsTransform::setInterpolate4( const hkQsTransform& a, const hkQsTransform& b, hkSimdRealParameter t
  );
```

For convenience hkQsTransforms have an interpolation operation. Since this operation is used intensively in many animation-based applications, it has been implemented with speed in mind, rather than accuracy. The interpolation function is implemented as, given two hkQsTransforms A and B and an interpolation factor r (between 0 and 1), as returning C such as

- Translation (C) = Vector_Interpolation (Translation(A), Translation(B), r)
- Rotation (C) = Quaternion_Normalize (Vector_Interpolation (Rotation_Vector(A), Rotation_Vector(B), r))
- Scale (C) = Vector_Interpolation (Scale (A), Scale (B), r)

Vector4_Interpolation is the usual, component-wise, vector interpolation:

- Vector4_Interpolation (A, B, r) = A * (1-r) + B * (r)

Notice how we use this vector interpolation with quaternions as well, by converting them to four-component vectors and ensuring normalization after interpolation. This is much faster than performing SLERP, and returns similar results if the two rotations are in the same hemisphere.

4.3.5.5 Blending hkQsTransforms

```
void hkQsTransform::setZero();
void hkQsTransform::blendAddMul(const hkQsTransform& other,  hkSimdRealParameter weight = 1.0f );
void hkQsTransform::blendNormalize( hkSimdRealParameter totalWeight = 1.0f);
```

Another convenience functionality in hkQsTransforms is the ability to blend transforms. As with interpolation, these operations are defined using vector operations on the three components. Blending of a transform B, with weight w, into a transform A results in a transform C is implemented as

- Translation (C) = Translation(A) + w * Translation(B)
- Rotation (C) = Rotation_Vector(A) + w * Rotation_Vector(B)

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- Scale (C) = Scale(A) + w * Scale(B)

Notice how we don't automatically normalize the rotation component. This is because blending operations usually are performed with multiple transforms (starting with a "zero" transform) and weights, and therefore normalization is done at the end. We implement this blending normalization operation (of a blended transform A with total weight w) as:

- Translation (N) = (1/w) * Translation (A)
- Rotation (N) = Quaternion_Normalize ((1/w) * Rotation_Vector(A))
- Scale (N) = (1/w) * Scale (A)

4.3.5.6 Transforming vertices by hkQsTransform

```
void hkVector4::setTransformedPos(const hkQsTransform& a, const hkVector4& b);
```

The hkVector4 class defines the method setTransformedPos() which takes a position (vertex) and an hkQsTransform. For vertex transformation, we consider hkQsTransforms to represent a concatenation of Scale, Rotation and Translation operations on the vertex, such as (being P the vertex and T,R,S the components of the hkQsTransform):

- P' = Transform_Vertex (QsTransform(T,R,S), P) = Translate (T, Rotate (Scale (S, P)))

Or, if we consider the T, R and S components in matrix form, and P as a column vector

- P' = T * R * S * P

Notice that, for hkQsTransform, vertex transformation is a different operation to multiplication. This is in contrast to matrix-based transforms, where both operations are implemented as matrix multiplications. Because of this difference, some properties of Matrix4 involving these operations do not apply to hkQsTransform..

4.3.5.7 Converting to and from general 4x4 matrices

You can convert hkQsTransforms to and from 4x4 matrices. 4x4 matrices can represent any general affine transform, while hkQsTransform can only represent orthogonal transforms.

Converting to 4x4 matrix

Since the orthogonal transforms are a subset of affine transforms, you can convert to a general 4x4 matrix without loss of information, by using the method:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
void hkQsTransform::get4x4ColumnMajor(hkReal* p) const;
```

The matrix 4 is returned by filling an array of 16 floats, arranged in column-major format. That's the format used by the hkMatrix4 class in Havok, so you can cast any hkMatrix4 object into an array of hkReal and pass it to the method above.

Converting from 4x4 matrix

Converting from a 4x4 matrix to a hkQsTransform is not always possible since not all affine transforms (representable by a 4x4 matrix) are orthogonal (representable by an hkQsTransform). The method:

```
hkBool hkQsTransform::set4x4ColumnMajor(const hkReal* p);
```

Will attempt to decompose the 4x4 matrix into the translation, rotation and scale components of an hkQsTransform. It will return true if the decomposition is fully successful (i.e., if the 4x4 is orthogonal); otherwise it will return false.

4.3.5.8 Converting to and from hkTransform

hkTransform is an alternative transform representation available in Havok. They usually represents orthonormal transformations, that is, transformations where there is only rotation and translation involved. This is because hkTransform is extensively used in Havok Physics, where scale is irrelevant.

However, and since the rotation component in hkTransform is stored as a 3x3 matrix, rather than a quaternion, you can set a "rotation" in an hkTransform that also contains scale (although we recommend that you use hkMatrix4 in that case, and only use hkTransform for orthonormal transformations).

Converting to hkTransform

There are two conversions to hkTransform, depending on whether you want to ignore the scale component or not:

```
void hkQsTransform::copyToTransformNoScale (hkTransform& transformOut) const;
void hkQsTransform::copyToTransform (hkTransform& transformOut) const;
```

Converting from hkTransform

Similarly, there are two conversions to hkTransform. In this case, one will assume there is no scale (will assume the hkTransform holds an orthonormal transform), while the other won't. The first one (assuming no scale) is significantly faster but will fail if any scale is found in the hkTransform.

```
void hkQsTransform::setFromTransformNoScale (const hkTransform& transform);
void hkQsTransform::setFromTransform (const hkTransform& transform);
```

4.3.5.9 General Properties of hkQsTransforms

Considering

- A, B, Q as being any hkQsTransforms
- M, N as being any affine 4x4 matrix transforms
- Q @ P as being "the transformation of vertex P by hkQsTransform Q"
- A * B as being "the multiplication of hkQsTransforms A and B"
- M * N as being "the multiplication of 4x4 matrices M and N"
- M4_To_QS (M) as being "the conversion from 4x4 matrix M to a hkQsTransform"
- QS_To_M4 (Q) as being "the conversion from hkQsTransform Q to a 4x4 matrix"
- Inv(Q) as being "the inverse of hkQsTransform Q"
- Inv(M) as being "the inverse of 4x4 matrix M"

The following are properties of all hkQsTransform:

- A * (B * C) = (A * B) * C
- A * Identity = Identity * A = A
- A * Inv (A) = Inv (A) * A = Identity
- A = M4_To_QS (QS_To_M4 (A))

The following are *NOT* properties of all hkQsTransforms

1. A * B = B * A
2. (A * B) @ P = A @ (B @ P)
3. Inv (Q) @ (Q @ P) = P
4. M = QS_To_M4 (M4_To_QS (M))
5. QS_To_M4 (Inv (Q)) = Inv (QS_To_M4 (Q))
6. A * B = M4_To_QS (QS_To_M4 (A) * QS_To_M4 (B))
7. M * N = QS_To_M4 (M4_To_QS (M) * M4_To_QS (N))

(1) Transformations are not commutative. False for most cases.

(2,3) Due to the different nature of the "apply transform to vertex" and "multiply transforms" operations in hkQsTransform. Only true if no scale is involved.

(4) Since M can contain affine transformations not representable with hkQsTransforms (skew for example), information may be lost in the first conversion. Only true if M is orthogonal (no skewing)

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

(5) Due to the different interpretation of the inverse operation in hkQsTransforms and general 4x4 matrices. Only true if Q has no scale.

(6,7) Due to the different interpretation of the multiplication operation when scale is involved in hkQsTransform and general 4x4 matrices. Only true if A,B,M,N have no scale (are orthonormal).

4.3.5.10 Tips when converting poses between hkQsTransform and 4x4 matrices

The properties above, in particular 5, 6 and 7, raise the following:

Important:
When dealing with scale, and moving from 4x4 representation to hkQsTransform and vice versa, it is important to notice that operations (multiplication, inverse) in one space do not match in the other space, and therefore the results will differ depending on what representation is used.

So it is important to perform the conversion at the point where the best match is needed, because further operations will diverge in each space. This applies particularly when dealing with poses, since converting between local and model spaces involves multiplications and inversions.

For example, if you have a pose represented as an array of 4x4 matrices, each one defined in local space, and you need to represent that pose with hkQsTransforms, it is much better if you:

- First convert the local, 4x4 matrix-based, pose into model space (multiply each local transform by its parent)
- Then convert these model space pose of 4x4 transforms into a model space of hkQsTransforms by converting each transform
- Finally, if you need the local space pose, calculate it from the model space pose

By having done this, instead of directly converting the local, 4x4 matrix-based, pose to hkQsTransforms, we ensure that the two representations (4x4 matrix and hkQsTransforms) match best when considered in model space (what the artist sees in the modeller) instead of local space. This is what, for example, our export utilities do when exporting data from 3ds max or maya.

Similarly, if at runtime you have a local space pose using hkQsTransforms, and you need model space 4x4 matrices for skinning, you should convert the hkQsTransform local space pose to model space, and then convert the hkQsTransforms to 4x4 matrices. The rule is:

Tip:
In general, when dealing with poses, always do conversions in model space.

4.3.6 Animation Representation

The Havok Animation system supports animations of both transform data, and additional scalar data. Transform data is associated with the bones of a skeleton; skeletons also contain "slots" or "float slots" which are the analog of a bones for floating point data. Most of our examples use only the bone data, usually derived from a skeletal animation system, and so an animation can be thought of as a character animation. This is the core use-case for the Havok Animation system. However Havok Animation also

supports arbitrary animated floating point channels, which can be used to extend a character animation with 'auxiliary' single DOF tracks. In this case, both bone and float tracks would be present (created, sampled and blended). A third use-case would be to have no bone data at all, and use only single DOF float tracks - for example when representing morph target data for facial animation.

Even when no bone data is present, the same objects and object names are used for consistency, and we will refer to 'Skeletal' animations, and 'Skeletons' throughout this documentation.

4.3.6.1 Bone And Float Representation

Havok Animation handles two 'types' of data: transform data and float data. Transform data is associated with the bones of a skeleton and is used to define the pose of a character at a point in time. Float data is associated with the "Float slots" of a skeleton and allows for arbitrary floating point values to be animated. Both are treated independently of each other for creation, sampling and blending.

Bone Data

Some animation systems represent a bone animation as a collection of separate channels or degrees of freedom. Using a predefined rig these channels can be transformed into either a local or world pose for a given character. This pose can then be handed to the rendering system for skinning.

When blending inputs from various sources you have to either map them back to the original rig or drop down to some other intermediate representation to do the blending. Sometimes mapping back to the original rig is difficult or costly e.g. An iterative IK system may work in general 6 DOF space. Or sometimes we need to blend animations which we originally authored using other sources e.g. MoCap data must be fitted to our rig.

Since we do not wish to impose any specific rig configuration on our animation system we choose to perform all blending in a 10 DOF[6] local pose space. We force each of our inputs, or hkaSkeletalAnimations, to produce a set of decomposed transforms, hkQsTransforms, in this local space. Each transform is decomposed into a format suitable for efficient blending. This set of transforms is used together with the animation binding to blend the local pose onto a given skeleton.

In Havok Animation all bone animations may be considered as a set of separate channels, however each of the bone channels contains the full 10 degrees of freedom we use to represent a bone. When we refer to a bone track we are referring to the full position, orientation and scale for a specific animated bone. Often we use the term transform track to eliminate any ambiguity, and member/method names in the code may contain the string 'transform' to make things explicit, and to distinguish from 'float tracks as described below.

Float Data

Single DOF floating point data is represented in Havok by a single DOF floating point track, or 'float track'. No assumptions about range or distribution are made (although uninitialized values will be set to 0), and blending is done using straight linear interpolation.

[6] 3 for position, 4 for orientation and 3 for scale.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.3.6.2 Skeletal Animations

The hkaSkeletalAnimation class provides the interface for all inputs to the animation system. The interface enforces that any implementation can provide a full set of transforms, and a full set of float values for any time period that lies within the length of the animation. To facilitate this, the interface explicitly stores the length of the animation - we refer to this as the duration of the animation. In addition this interface class stores useful information that has come from the tool chain. This includes a handle to any motion information that has been extracted (see Motion Extraction) together with naming and annotation data. This tool chain information is stored in an array of hkaAnnotationTracks. If this array is not empty it will contain an entry for each bone track in the underlying animation.

We do not want to enforce either authoring or target sampling rates on our animations so we assume these can be sampled continuously. The core interface that enforces these restrictions is shown here.

```
virtual void sampleTracks(hkReal time, hkQsTransform* transformTracksOut, hkReal* floatTracksOut,
  hkaChunkCache* cache) const = 0;
```

4.3.6.3 Annotation Tracks

Each bone track (but *not* each float track) is tagged with the original name used to identify it in the modeler. Often the names of the tracks in an animation will exactly match the names of the bones in the skeleton it is designed for. If you are retargeting an animation to a different skeleton or only partially animating a given skeleton it is useful to have the original tool chain names. This allows you to construct an appropriate animation binding which can be used to playback the animation on the new skeleton.

It is often useful to mark up track specific events in the modeler. Examples are Left ankle down at frame 5, Left toe up at frame 10 and similar foot placement events used to synchronize gait. Another example might annotate an event on the hand to say, at frame 5, handle grenade thrown now. Different modelers have different techniques for marking up this annotation. At runtime this is baked into the annotation array which is stored in the animation track. This array contains a set of structures containing sorted events. Each event consists of a time specified in seconds and a text string which comes straight from the tool chain. The animation/annotation demo shows how you can access this data to drive game events.

4.3.6.4 Interleaved Animations

The simplest type of animation in Havok Animation is a hkaInterleavedSkeletalAnimation. This animation stores its data with an interleaved layout[7].

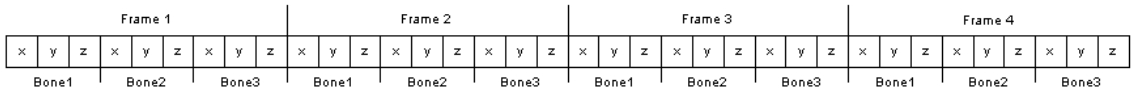


Figure 4.5: Animation Data in an Interleaved Layout

The float tracks are stored similarly ina separate block of data. Interleaving the data in this way gives a better memory access pattern and improved cache performance over storing the data per channel. This

[7] For simplicity we only show 3 of the 10 degrees of freedom of a single bone track here.

basic animation also stores the length of the total animation in seconds. We assume that each pose is uniformly spaced (in time). When fulfilling a request to sample the animations at a specified local time we:

- Find the samples immediately preceding and following the given time.
- Interpolate these poses to produce the final output pose.

Note:

When blending poses we simply linearly interpolate the position, rotation and scale for bone tracks. For the sake of efficiency we do not slerp rotations. In general the rotations for any bone will vary little from one frame to the next and lerp is appropriate. We also always linearly interpolate the float tracks.[8].

4.3.7 Rig Information in Havok Animation

Havok Animation has no predefined rig structure. Instead we simply expect poses to appear in local space (parent relative). The hkaSkeleton class contains the hierarchy that defines the parent/child relationship between bones.

4.3.7.1 Bones

The hkaBone class only exists to provide you with some useful tool chain information. It contains the name of the bone (useful for retargeting as mentioned before). It also stores a flag which indicates if translation is used when playing animations on this skeleton. This flag is used by the skeleton mappers to ensure that bones are not stretched when mapping from one rig to another.

For float tracks there is no corresponding 'hkaFloat Bone' class. We extend the concept of bone to floats by introducing 'Float Slots'. A Float Slot is the conceptual destination for all sampled float data. A Float Slot has only a name, and implicitly an order inside a skeleton - see below.

4.3.7.2 Skeletons

The hkaSkeleton class contains:

Bone Information A set of bones and a parenting relationship - the parent indices. The indices array is stored as an array of signed 16 bit integers. Each bone in the set has a corresponding entry in the parent indices array. This entry gives the index of the bone's parent. Indices begin at 0. Root bones that have no parent are flagged with a parent index of -1.

The skeleton also contains a reference pose. This pose is used during animation blending when no animation data exists for a given bone.

Float Information A set of float slots (names). There is no corresponding 'reference pose' for float slots. When no animation data exists for a given float slot, 0 will be assumed.

Note:

Unlike bones, all float slots are treated completely independently (there is no concept of 'local/model' space, and no hierarchy for float slots). Also there is no link between any given float slot and any given bone - in this sense bones and float slots are independent.

4.3.8 Playback

The state of a character in the Havok Animation library is defined by the `hkaAnimatedSkeleton` class. An instance of `hkaAnimatedSkeleton` contains one or more Playback Controls which each reference a single `hkaSkeletalAnimation` instance and specify a blend type. Each individual animation is sampled and blended together into a pose.

4.3.8.1 Sampling and Blending of Bones and Float Slots

The methods described below allow the pose of a character to be determined at a given point in time.

```
void hkaAnimatedSkeleton::sampleAndCombineAnimations( hkQsTransform* poseLocalSpaceOut, hkReal*
  floatSlotsOut, hkaChunkCache* cache = HK_NULL ) const;

void hkaAnimatedSkeleton::sampleAndCombinePartialAnimations( hkQsTransform* poseLocalSpaceOut, hkUint32
  maxBones, hkReal* floatSlotsOut, hkUint32 maxFloatSlots, hkaChunkCache* cache = HK_NULL ) const;

void hkaAnimatedSkeleton::sampleAndCombineSingleBone( hkQsTransform* localSpaceOut, hkInt16 bone,
  hkaChunkCache* cache = HK_NULL ) const;
void hkaAnimatedSkeleton::sampleAndCombineSingleSlot( hkReal* floatOut, hkInt16 slot, hkaChunkCache* cache
  = HK_NULL ) const;
void hkaAnimatedSkeleton::sampleAndCombineIndividualBones( hkQsTransform* localSpaceOut, hkInt16* bones,
  hkUint32 numBones, hkaChunkCache* cache = HK_NULL ) const;
void hkaAnimatedSkeleton::sampleAndCombineIndividualSlots( hkReal* floatOut, hkInt16* slots, hkUint32
  numSlots, hkaChunkCache* cache = HK_NULL ) const;
```

Sampling A Complete Pose

The `sampleAndCombineAnimations()` method allows the pose of a character, as defined by a series of blended animations, to be sampled at a given time. Calling `sampleAndCombineAnimations` queries all active controls and samples their animations. This animation data is then blended into a single local pose. An optional cache can be passed to improve decompression performance of Wavelet Compressed or Delta Compressed animations (Spline Compressed animations do not require a cache).

Sampling A Range of Bones

This is an advanced feature which supports LOD of a range of bones which have been topologically sorted by the user (the Content Tools enforce one such sorting - depth first sort). This range (0 to N) of bones is guaranteed to be sampled, but the efficiency of this method is dependent on the bone ordering in the corresponding `hkaSkeleton`. It is the user's responsibility to convert these bone transforms to world space (if desired), and to ensure that a full chain of bones starting from the root bone have been chosen. It is advised that the user contact Havok support before using this function.

Sampling Individual Bones

The `"Single" and "Individual" sampling` functions allow the user to sample an arbitrary set of bones

in the animation at a given time from a set of blended animations. One anticipated use case is supporting Level Of Detail animation in which only major bones such as the spline arms and legs are sampled, but minor bones such as fingers and toes are not. Another anticipated use case is to sample only a small subset of bones which are considered important, such as those belonging to the chain of bones from the root to the tip of the hand swinging a weapon. Note that the bones sampled are in local space. It is the user's responsibility to convert these bone transforms to world space (if desired), and to ensure that a full chain of bones starting from the root bone have been chosen. The corresponding Slot versions of these functions operate on float slot data.

Important:

This feature is currently only supported for blends of Interleaved, Spline Compressed, and Mirrored versions these animations, respectively. This feature must be enabled in the content tools for Spline Compressed animations. This feature is currently not supported for Wavelet Compressed or Delta Compressed animations.

4.3.8.2 Sampling of Transform Tracks and Float Tracks

The tracks of an animation may be sampled directly from the `hkaSkeletalAnimation` class. Sampling from the animation allows direct access to the local space transforms of each track present in the animation at a given time. The features described below are useful for users implementing their own animation controls and blending functionality.

```
void sampleTracks(hkReal time, hkQsTransform* transformTracksOut, hkReal* floatTracksOut, hkaChunkCache*
  cache) const = 0;
void samplePartialTracks(hkReal time, hkUint32 maxNumTransformTracks, hkQsTransform* transformTracksOut,
  hkUint32 maxNumFloatTracks, hkReal* floatTracksOut, hkaChunkCache* cache) const;
void sampleSingleTransformTrack( hkReal time, hkInt16 track, hkQsTransform* transformOut ) const;
void sampleSingleFloatTrack( hkReal time, hkInt16 track, hkReal* out ) const;
void sampleIndividualTransformTracks( hkReal time, const hkInt16* tracks, hkUint32 numTracks,
  hkQsTransform* transformOut ) const = 0;
void sampleIndividualFloatTracks( hkReal time, const hkInt16* tracks, hkUint32 numTracks, hkReal* out )
  const = 0;
```

Sampling All Tracks

The `sampleTracks()` interface is the simplest of the sampling member functions. Given a local time, the value of all transform and float tracks are computed. The associated `hkaAnimationBinding` class may be used to map the sampled tracks to the appropriate bone / float slots.

Sampling A Range of Tracks

The "Partial" sampling function allows the first N tracks to be sampled. It is the user's responsibility to convert these bone transforms to world space (if desired), and to ensure that a full chain of bones starting from the root bone have been chosen. It is advised that the user contact Havok support before using this function.

Sampling Individual Tracks

The "Single" and "Individual" sampling functions allow the user to sample an arbitrary set of tracks in the animation at a given time. One anticipated use case is supporting Level Of Detail animation in which only tracks corresponding to major bones such as the spline arms and legs are sampled, but tracks corresponding to minor bones such as fingers and toes are not. Another anticipated use case is to sample only a small subset of tracks which are considered important, such as those corresponding to the chain of

bones from the root to the tip of the hand swinging a weapon. Note that the transforms sampled are in local space. It is the user's responsibility to convert these transforms to world space (if desired), and to ensure that a full set of tracks corresponding to a chain of bones from the root bone have been chosen.

Important:

This feature is currently only supported for Interleaved, Spline Compressed, and Mirrored versions these animations, respectively. This feature must be enabled in the content tools for Spline Compressed animations. This feature is currently not supported for Wavelet Compressed or Delta Compressed animations.

4.3.8.3 Playback Control

The hkaAnimationControl interface class provides access to the information needed to sample and blend a specific animation. If an animation is playing then it must have an associated control. Currently controls cannot be shared and must be created for each instance of a playing animation.

Each control returns a state structure which contains:

- The local time used to sample an animation.
- A master weight to use when blending this animation.
- Access to the animation binding for this animation.
- Two sets of track weights used for per track feathering / masking (one for transform tracks, one for float tracks).

The Default Playback Control

The default control, hkaDefaultAnimationControl, is the default implementation for our control interface. This control automatically handles looping when stepped beyond the start or period of the animation. It also provides:

- Playback speed control.
- Direct access to the master weight for the playing animation.
- Direct access to the sampling time for the playing animation.
- Overflow and underflow callbacks (animation local time has past the end or beginning of the animation, respectively).
- Ease curve support.

Ease Curve Support

Ease curves are most frequently used when transitioning between animations at runtime. Easing modulates the standard control weight for an animation and provides a smooth transition from one animation to the next. Our ease modulation is always enabled.

To control the ease in or ease out curves you specify the control points of a standard Bezier curve. This parameterisation has some nice intuitive curves as shown below.

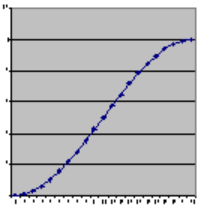
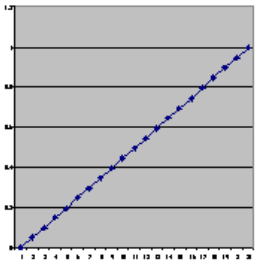
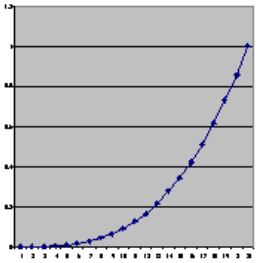
Curve Type	Graph	Parameters
Smooth / Sinusoidal		[0, 0, 1, 1]
Linear		[0, 0.33, 0.66, 1]
Exponential		[0, 0, 0, 1]

Table 4.1: Bezier Curves for Easing In / Out of Animations

Values for the control points are usually specified between 0 and 1, although other values can be used.

When easing in/out you specify the duration over which the ease occurs. Conceptually this horizontally stretches / squashes the curves. If you ease in an animation that is already easing in the curve continues from its current value rather than snapping back to the start of the curve.

If you ease back in an animation that is currently easing out then the ease-in continues from where the ease-out left off, if the ease curves are symmetric.

4.3.8.4 Animation Blending

The hkaAnimatedSkeleton class aggregates all playing animation controls for a specific instance of a character. It is responsible for advancing these controls, sampling their underlying animations, and combining these sampled poses into a single local pose that will ultimately be used for rendering.

It also provides support for sampling and blending any motion that has been extracted from each animation. This separation of motion blending and animation blending is efficient. It allows the character

motion to be checked independently from the rest of the animation system - this check may occur frequently per frame in, say, an NPC's state machine.

The interface also allows you to specify additional per track weights. These allow you to feather all animations playing on a particular bone or float slot. This could be used to dull effects from a limb that has been shot.

Normal Blending

Normal blending is the most common use case and corresponds to 'averaging' the contribution of multiple tracks, as follows:

```
For each active animation control bound to this animated skeleton
     If the binding specifies normal blending
       Use the control to sample the animation using the current control state
       For each track in the animation
           If this animation track is bound to a bone
               Weight = Motion Weight *  Control Weight * Track Weight
               Blend transform/float into bone/float slots according to Weight, and accumulate Weights

 For each bone in the skeleton
     If the total weight on this bone is below bind pose threshold
         Blend in the bind pose for the skeleton

 Normalize transforms and floats by sum of weights of that track.
```

Note:
If the total weight of all contributing controls on any particular bone drops below a user specified threshold (the *bind pose threshold*) then we automatically blend in the reference pose transform for that bone. This eliminates any popping artifact caused by animations suddenly ceasing or feathering that is too aggressive.

As in Interleaved Animations, we use linear interpolation to perform the blending. To be precise, blending happens as follows:

We add a weighted multiple of each of the position, rotation and scale components separately (with a hemisphere check for the quaternion representing the rotational component), and similarly with the float values. After all contributions have been added we 'normalize' the resulting output pose by division by the total weight (for position, scale and float components) or by quaternion normalization (for rotation components). This has the added advantage of being order independent - animation controls can be added to the skeleton in any order and are guaranteed to produce the same outcome.

Note:
The blend hint is not used for *float tracks* which are always blended the same way.

Additive Blending

Additive blending allows detail animations to be layered onto an existing base animation to add subtle detail to the final pose. To produce the detail animation use either the runtime utilities found in hkanimation/animation/util/hkaAdditiveAnimationUtility.h or use the CreateAdditiveAnimationFilter. The animation is created by extracting the detail from a base pose or animation. One common option removes the first frame of an animation from each of the subsequent frames leaving just the detail. When using the utility or the toolchain you have the following options when choosing the base:

- Use the first frame of an animation
- Use the reference pose from a corresponding skeleton
- Use raw animated data. This data must be same size as the original data (runtime only)

Once the detail animation or additive animation data has been created we need to inform the blending routines in hkaAnimatedSkeleton that this data should be layered on during blending. This is done by setting the blend hint flag in the hkaAnimationBinding for the given additive animation. This blend hint flag currently has two values, NORMAL and ADDITIVE. Normal blending follows the pseudo code above. For additive blending the routine below is run

```
For each active animation control bound to this animated skeleton
     If the binding specifies additive blending
       Use the control to sample the animation using the current control state
       For each track in the animation
           If this animation track is bound to a bone
               Weight = Motion Weight *  Control Weight * Track Weight
               add in in bone according to Weight
```

This is very similar to normal blending except the type of blend operation is different and no renormalization occurs afterward. During sampling all normal animations are blended first and then all additive animation are layered on. For an example of additive blending see the demo demos/AnimationApi/Blending/Additive this demo shows how to create an additive animation at runtime.

Note:
If you are using the runtime utilities to create additive animations you must remember to manually set the blend hint for the corresponding animation binding. Also if you are creating additive animations directly in your modeller (instead of using the CreateAdditiveAnimation filter) you will have to manually set the blend hint at runtime.

```
// Switch the binding to additive so this animation will be blended differently in sample and combine.
 m_binding[HK_ADDITIVE_ANIM]->m_blendHint = hkaAnimationBinding::ADDITIVE;
```

Note:
Additive blending is not explicitly supported for float tracks - float tracks are always blended 'additively'.

4.3.9 Motion Extraction / Locomotion

4.3.9.1 Overview

Most game engines use a simplified representation, a collision proxy, for characters. It is very important that the general motion of the proxy is tied appropriately to the underlying animation system. If this isn't done adequately the results often contain noticeable artefacts like foot sliding.

Nave pipelines force animators to animate on the spot and then match either character speed or the original animation to the movement of the proxy in-game. Usually these setups force artists to remove

all the natural acceleration and deceleration from the cycle (because of foot sliding) leaving an artificial result. This is a programmer driven integration model.

Other pipelines allow the artist to author natural run cycles with full displacement. A constant linear velocity is removed from this animation and used directly to drive the character proxy at runtime. This artist driven model faithfully reproduces the original intended motion.

Havok Animation extends the second model to handle more complex forms of motion, including abrupt changes in motion and direction. This allows artists to fully author complex motions and have these reproduced and blended appropriately at runtime.

4.3.9.2 What is Motion?

We use the term *motion* to specify the displacement of an animated skeleton (a character) during playback.

During the playback of an animation (a running cycle, or a hand waving animation), the position and orientation of the different bones of a skeleton will be animated. The motion of character is a concept that applies to the animated skeleton as a whole (instead of an animation track, which applies to individual bones), and basically is defined as the displacement of the animated skeleton as interpreted by the application. In other words, it represents how the overall animation of the individual bones affects the animation of the character's *reference frame* (this *reference frame* defines where the character *is* at any given point of time)

Take the example of a simple walk cycle. The original animation (in 3DS max, for example) would usually start with a character located at position A and would end at position B, a few meters away from the starting position.

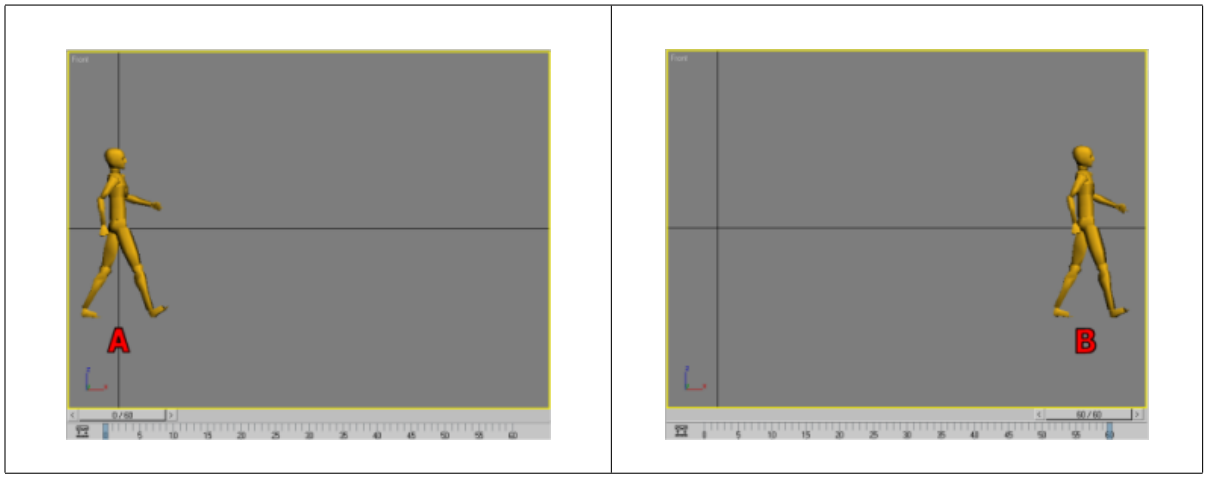


Table 4.2: Simple Motion caused by a Walk Cycle

If we loop this 60-frame (1 second) animation, though, what will happen is that the character will move from A to B, then snap back to A, move to B, snap again to A, etc. What we can do, of course, is make sure that once we reach the end of the animation, we then move our character to position B; that is, we add the translation vector B-A each time a full animation cycle is reached.

By doing this, we are starting to use the concept of motion, although in a very simplistic manner. The translation B-A is effectively the motion of our character after a single animation cycle.

The motion of animated character, therefore, is an abstraction of the concept of "how much a character moves" or, alternatively, "how much a character should move", during the animation. Motion can usually be estimated, as we did in our example, by looking at the animation for the root node of a character, and projecting it to a particular axis or plane.

We define then motion extraction as the process with which we split the animation (track) of the root of a character into a motion (track) and local animation (track).

For example, if in the example above we consider the motion of our character to be a constant velocity displacement between points A and B, our motion extraction would look like this:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

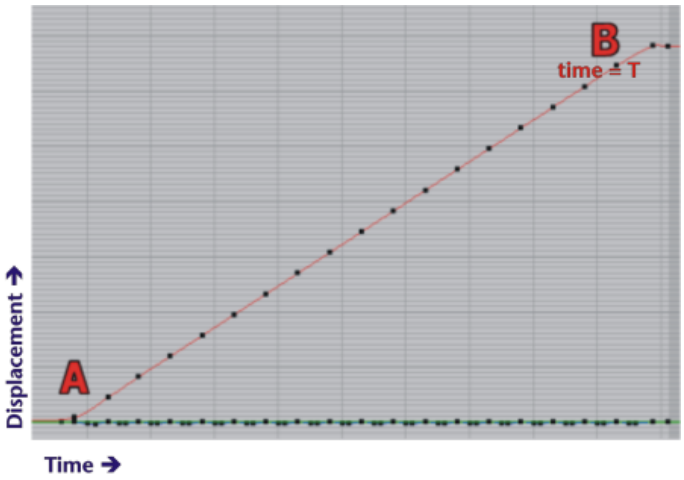
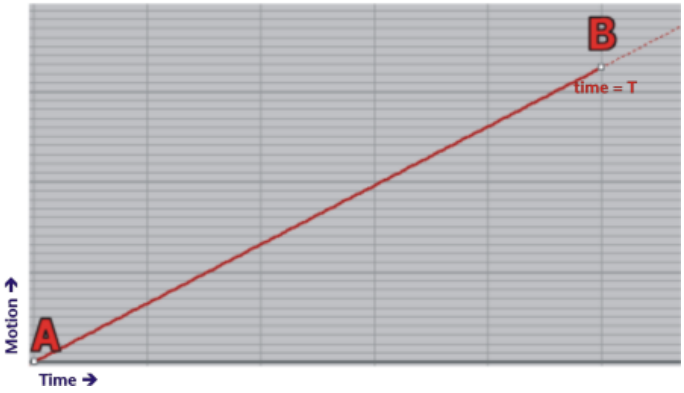
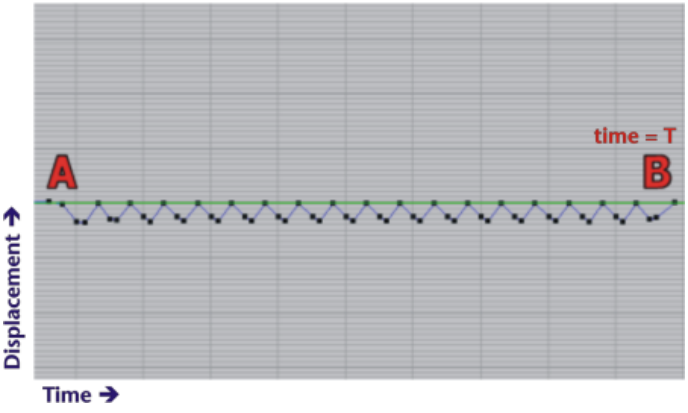
	The root track of our walk animation contains a displacement along the X-axis (in red) from A to B, plus some smaller displacements in the Z (up) axis (in blue), caused by the up/down hip movement.
	After motion extraction, we get a motion track (depicted here) that represents the net displacement of the character (in the X-axis). *One of the most important things that differentiates motion from the rest of animation is the fact that motion accumulates during playback. This is here: notice how, after reaching B, the value of X in the motion track continues to increase, where the animation would however loop again to its values at time A.*
	After motion extraction, the root track now only contains the smaller local displacements on the up (Z) axis.

Table 4.3: Character Motion at a Constant Velocity

Notice that after motion extraction, our animation will be a "walk in place", since all displacement in the forward (X) direction has been extracted.

Using Multiple Samples

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

In our example above, our motion along the X-axis is constant in velocity so we only needed two samples (one at A and one at B) to specify the motion (the motion was a straight line). There are some cases, however, where we may need to represent motion with more accuracy. Imagine this animation: our character walks a few steps, then stops, and then runs for a few steps. If we only look at the motion using start time A and end time B, our motion extraction would be:

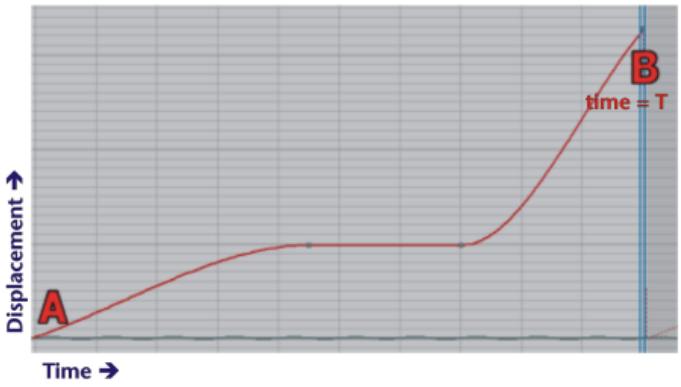
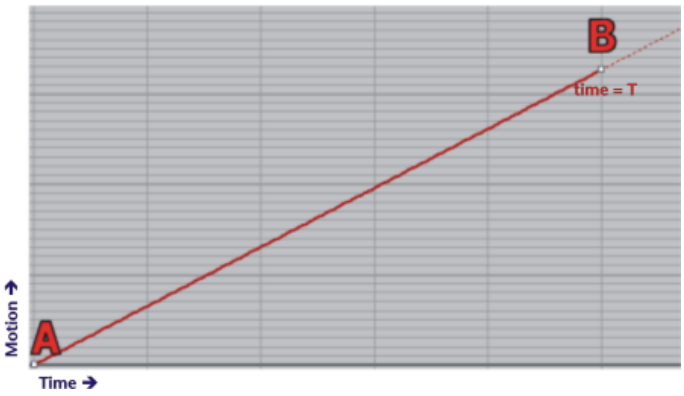
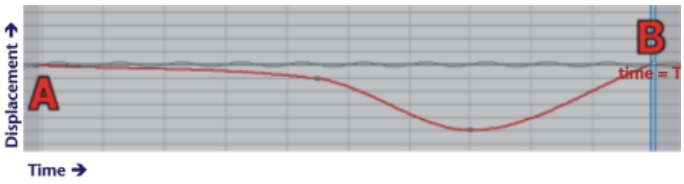
	The root track of our walk+stop+run animation contains mainly displacement along the X-axis (in red) from A to B, but this time it is not constant in velocity.
	If we only sample the motion using the X-axis at points A and B as we did in the previous example, our motion will have again constant velocity over time.
	After motion extraction, the root track now only contains the difference between the original animation and the extracted motion. Notice there are still big displacements on the forward (X) axis, resulting from the difference between the original animation and our constant velocity motion.

Table 4.4: Character Motion at Multiple Velocities

As we said earlier in this section, one of the uses of motion is that it tells us "where the character is", trying to disregard small local variations in the animation of particular bones. In the example above, the root node would still have a considerable amount of animation in the X (forward) direction, since a constant velocity forward motion doesn't accurately reflect the movement of our character. This can sometimes be a problem, as we will see in some of our examples later on.

If, however, we use multiple samples for the motion (instead of just sampling A and B), we can construct a much better representation for the motion. In particular, if we use as many samples as the animation:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

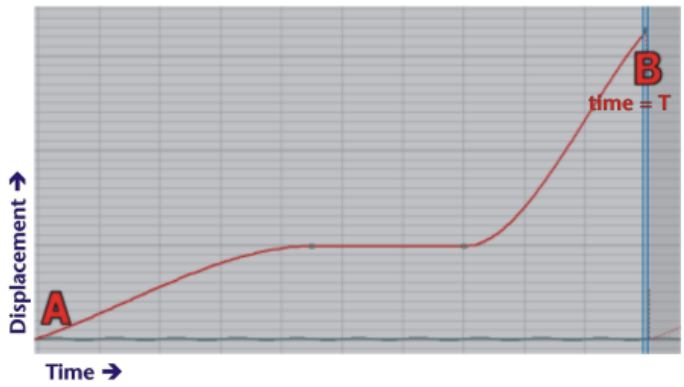
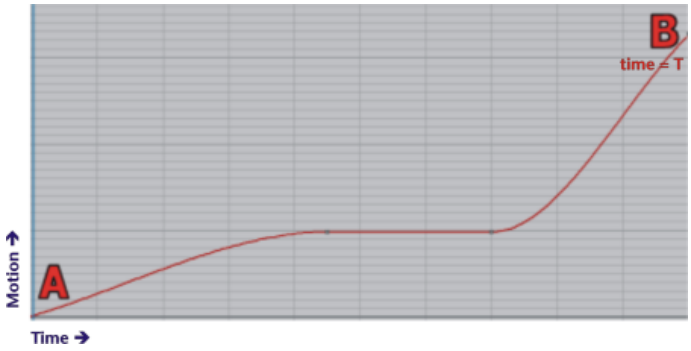
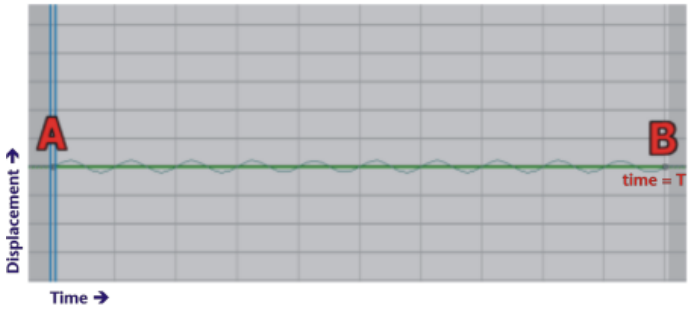
	Lets look again at our walk+stop+run animation.
	We now sample the motion in the X-axis by taking samples every frame. We basically reconstruct the curve for the X component of the roots translation.
	After motion extraction, the X translation component in the root track is now 0. The rest of components (Z, i.e. Up, in particular) only contain local displacements.

Table 4.5: Motion Extraction using multiple samples

The motion track now gives us a much more accurate description of "where the character is" at any time during playback.

Choosing the Motion Components

In the previous examples, we used a single component (X) to describe the motion extracted from the animation. This is because in our example, X was the "forward" direction in our scene, and the animations involved forward motion.

Since the choice of axis is arbitrary, from now on we will be talking about forward/back axis, right/left axis and up/down (or vertical) axis instead of X, Y or Z.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Not all animations have forward motion. The motion of a vertical jump, for example, could be described using the up/down axis; the motion of a forward jump may be described by using both the forward/back axis and the up/down axis. Finally, some animations, like a hand waving animation, may not have motion at all.

Also, apart from translations on particular axis, there is also a *turning motion* that we may want to extract from an animation. We define this turning motion (or *yaw motion*) as a rotation around the vertical (up/down) axis.

So, in total, we have 4 *motion components*:

- Translation on the forward/back direction.
- Translation on the right/left direction.
- Translation on the up/down direction.
- Turning (rotation) around the vertical axis.

The choice of which motion components should be extracted from the root animation is application-specific. However, most game applications have similar requirements, and therefore would follow these rules:

- If an animation is meant to loop seamlessly, you want to extract motion of those components where the initial and final poses differ (so the initial and final poses match).
- Even if a component loops seamlessly, the application may be interested in keeping track of motion for particular components during the animation cycle. In those cases, you want to extract the motion on those components using multiple samples.

Example 1

We have a jumping animation:

Figure 4.6: A Character's Jumping Animation

If we want this animation to loop seamlessly, we need to extract the motion in the forward direction, since the character ends up at some distance along the forward axis; otherwise the animation will snap back to the original position during playback.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Extracting motion along the vertical axis is not required for seamlessly looping, since the animation finishes with the character at the same height as the initial pose. However, in many situations we may want to extract the vertical motion from animations like this one. Imagine this situation:

We are using the animation to drive a character in a game. For collision detection, we are representing the character with a capsule, and we move this capsule representing the character according to the motion extracted from the character.

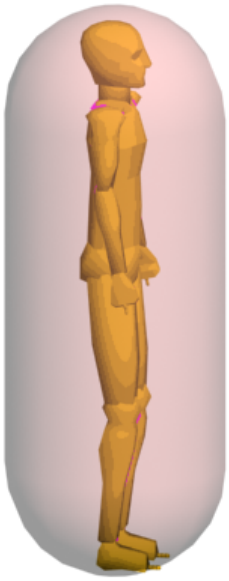


Figure 4.7: The Character with its Collision Detection Capsule

While with animations like walk or run cycles, considering only forward motion suits our needs, jump cycles are normally different. Consider the time when the animation has the character in the air:

Figure 4.8: The Capsule's Position when the Character jumps

Since our capsule representation follows the motion of the character, and we only extracted this motion along the forward axis, our game assumes the character is still at the same height (on the ground).

Imagine now that there is an obstacle on the way of the character (and thats why the player decided to jump which triggered the jump animation). Since our capsule representation (which we use for collision detection) doesn't follow vertical movements of the character, the capsule will collide with the obstacle,

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

a collision will be reported and the player will be stopped.

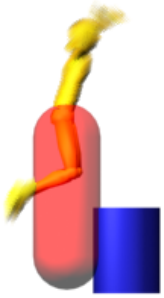


Figure 4.9: The Capsule Collides when the Character jumps over an obstacle

If, however, we allow vertical motion to be also extracted (and use multiple samples, otherwise the information about the jump is lost), we can use that motion to drive our capsule and therefore we dont get this false collision:

Figure 4.10: No Collision when Multiple Motion Samples are used

Example 2

Imagine we have an animation for a character turning around a corner:

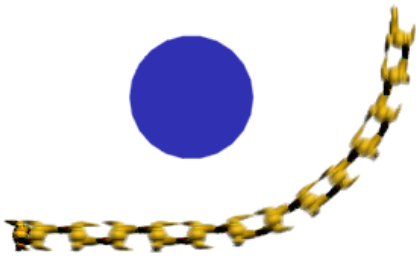


Figure 4.11: A Character's Turning Animation

In this case, we want to extract motion in both the forward/back axis and the right/left axis. We dont

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

need to extract up and down motion.

We need to use multiple samples in order to properly reproduce the motion curve of the character; if we use only two samples we would assume the motion is in a straight line (from the start to the end point) and therefore, if running an application like the previous example, our character representation (capsule) would report collisions with the blue obstacle in the picture.

Notice that for this animation we also want to extract turning (yaw) motion from the animation since the character actually turns 90 degrees from the start to the end (the animation finishes with the character facing a different direction) by doing this the end and start pose will match (and therefore loop seamlessly).

4.3.9.3 Storing Extracted Motion

Sometimes the animator can extract the motion of an animation before the animation is exported. For example, in any of the examples above, the animator could remove the root track (or parts of it) in the modelling tool and therefore have the animation walk, run or jump in place.

While it is not uncommon practice to do this, the main disadvantage is the fact that valuable information is lost if the removed tracks (extracted motion) are not exported alongside the animation. If motion is extracted manually by animators, it becomes the responsibility of the programmers to ensure that proper motion is applied at runtime in order to reproduce suitable animation. Otherwise artefacts (sliding feet for example) may occur.

By extracting the motion during export (and therefore leaving the original animation untouched) the extracted motion (the motion track) is exported and stored alongside the animations and can therefore be used at runtime without loss of information.

4.3.9.4 Applications of Motion Extraction

During the previous section we saw some of the reasons for extracting the motion from an animation. Here we'll review those and some other examples:

Animation Playback

When playing an animation of period T using a loop control, we need to ensure that the final pose at time T is the same as the initial pose at time 0 so the animation runs smoothly.

By extracting the necessary motion of the original animation, we can ensure that the final pose (at least for the root node) will match the initial pose.

Also, and since motion always accumulates (as opposed to animations, whose values repeat every period T), we can properly sequence or loop animations during playback, having the character move properly.

Use the Extracted Motion to Drive a Character

There are two main approaches that can be taken in order to control and display the movement of a character in a game:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

1. The application uses the user input to drive a character controller, which calculates a new position and the proper animation (and weights) to play. The animation system then calculates a new pose. The new pose and position are then used to display the character.

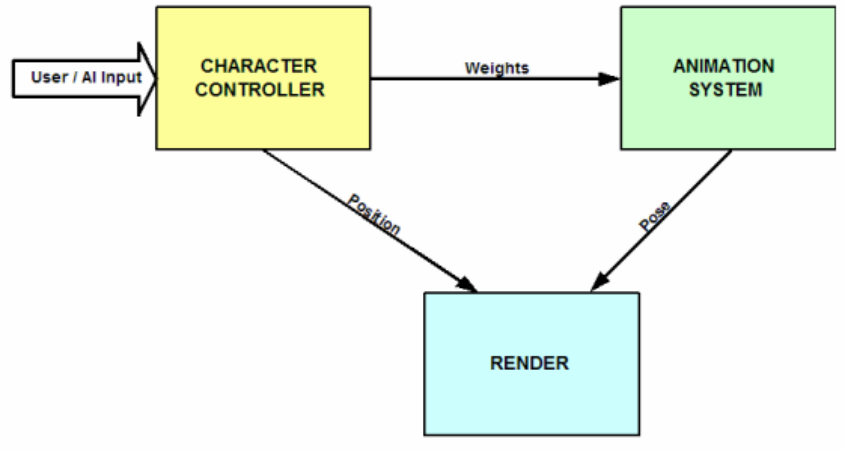


Figure 4.12: Animation System Driven by a Character Controller

The advantage of this first approach is that the application has total control over the position of the character, and it doesn't depend heavily on the animation system. This approach is commonly used in, for example, network-based first person shooters, where ensuring the correct positioning of the players is more crucial, and artefacts in the animation (sliding feet for example) are easily tolerated.

2. The application uses the user input to drive the animation system, which calculates a new pose and desired motion. This desired motion is used by the character controller to drive the character to a new position. The character is then displayed using the new pose and position.

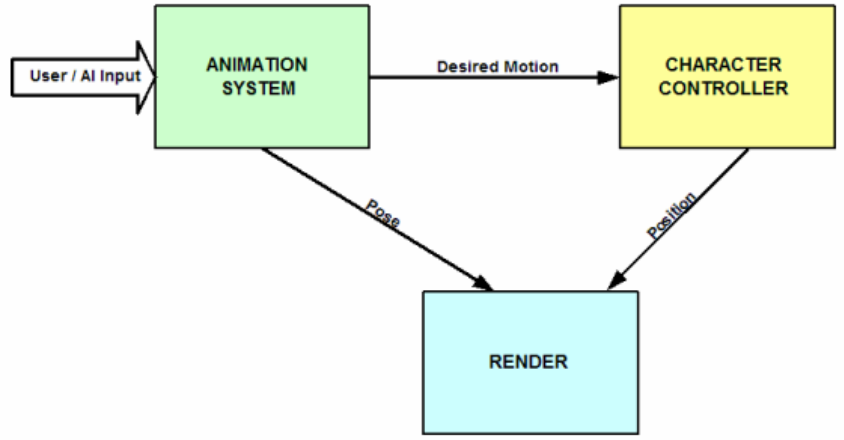


Figure 4.13: Character Controller Driven by an Animation System

The advantage of this second approach is that, by driving the character based on the motion extracted from the animation we ensure that the final result (position+pose) closely matches what the animator intended. For example, replacing a slow walking cycle with a fast walking cycle will automatically drive the character controller faster without requiring any adjustments in the controller code. This approach is particularly suitable for third-person games where emphasis is put on high-quality, smooth animation playback.

Notice that the two diagrams above are an oversimplification of the logic in game; they are just meant to emphasize the responsibilities of each component regarding the control of the characters position in

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

both approaches.

4.3.9.5 Motion in the Havok Animation System

Usually motion extraction is done through using the Havok Toolchain (see the Motion Extraction filter in the Havok Filter Manager chapter). At run time, the classes involved are as follows:

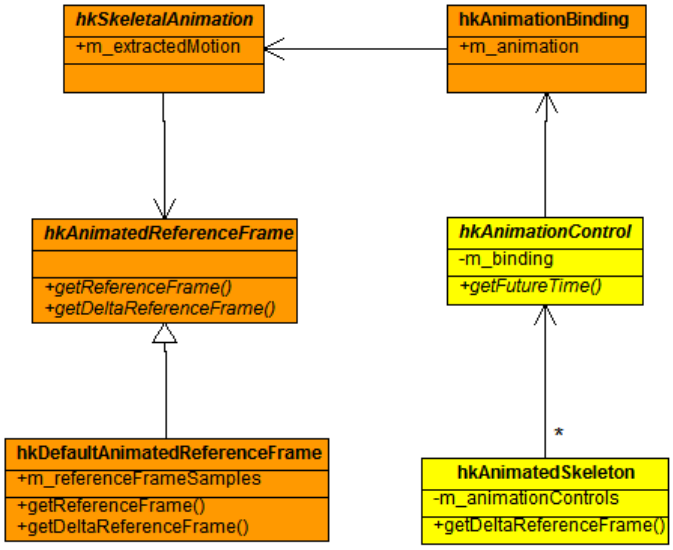


Figure 4.14: Motion Extraction Classes (at Runtime)

The classes in orange (on the left) are created (exported) by the toolchain; the classes in yellow are run-time classes only.

Some comments on how motion is handled by the different classes:

- **hkaSkeletalAnimation:** Has an `m_extractedMotion` member (possibly NULL) that points to an instance of `hkaAnimatedReferenceFrame` (the extracted motion).
- **hkaAnimatedReferenceFrame:** Defines the `getReferenceFrame()`, which returns the location of the reference frame (the extracted motion) at a given point in time (between 0 and the period of the original animation), and the `getDeltaReferenceFrame()` method, which returns the movement of the reference frame (the accumulated, or delta, motion) between two times.
- **hkaDefaultAnimatedReferenceFrame:** Implements both `getReferenceFrame()` and `getDeltaReferenceFrame()` by querying the samples that were extracted and stored from the original animation during toolchain operations (motion extraction).
- **hkaAnimationControl:** Keeps track of the current time and calculates how a time increase would affect the local time of an animation (it may handle looping, etc).
- **hkaAnimatedSkeleton:** Defines and implements `getDeltaReferenceFrame()` which samples the motion from all playing animations (through their respective animation controls) and blends them accordingly.

Extracting Motion using the Havok Toolchain

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The Havok Toolchain provides a *Motion Extraction Filter* that performs motion extraction operations in interleaved (`hkaInterleavedSkeletalAnimation`) animations. For more information about this filter, please check the Toolchain documentation.

Extracting Motion at Run-Time

While it is most common to perform motion extraction during toolchain operations, it is possible to also do so at run-time. You can create an instance of `hkaDefaultAnimatedReferenceFrame` based on an `hkaInterleavedSkeletalAnimation`, and then extract that from the animation using `hkaAnimatedReferenceFrameUtils`.

4.3.10 Mirroring Animations

4.3.10.1 Overview

Mirroring animation allows animators to synthesize a new animation from an existing one in which motion is transferred to opposite sides of the body. An example of an original animation and its mirrored counterpart are shown below in . Note how the opposite hand is waving in the mirrored animation yet asymmetries of the model and its textures are preserved. The Havok logo and black knee patch remain on the character's left side in both poses.

Figure 4.15: Example Mirrored Animation

Mirrored animation can also be used as a form of compression by using a left handed data set to represent both left and right handed motion simultaneously. The overhead of storing mirroring information is typically very small compared to the animation track data itself, allowing a series of left and right handed animations to be stored in roughly one half the space of exporting both animations directly.

4.3.10.2 hkaMirroredSkeleton and hkaMirroredSkeletalAnimation

Two classes are used to create a mirrored animation: hkaMirroredSkeletalAnimation and hkaMirroredSkeleton. The hkaMirroredSkeleton class is used for each skeleton (rig) which is to be mirrored. Just as multiple hkaSkeletalAnimation instances can share the same hkaSkeleton instance, multiple hkaMirroredSkeletalAnimation instances can share the same hkaMirroredSkeleton instance.

The hkaMirroredSkeleton class, shown below, encapsulates all settings relevant to mirroring an animation. A mirrored animation requires a mapping of bones from one side of the body to the other. This can be done by specifying the mapping manually, or using bone names to perform the mapping automatically.

```
class hkaMirroredSkeleton
{
public:
    hkaMirroredSkeleton( hkaSkeleton *skeleton );
    $\sim$hkaMirroredSkeleton();

    // MIRRORING SETTINGS
    void setMirrorAxis( const hkVector4 &v );
    void setRootOrientation( const hkQsTransform &T );
    void setChildOrientation( const hkQsTransform &T );
    void setNubOrientation( const hkQsTransform &T );
    void setMixedMode( hkBool mixed = true );

    /// PAIRING
    void setBonePair( hkInt16 bi, hkInt16 bj );
    void computeBonePairingFromNames( const hkObjectArray< hkString > &ltag, const hkObjectArray< hkString
      > &rtag );
};
```

In addition to bone mapping, several settings are available to adjust the mirroring algorithm to fit the convention of the model. Once the hkaMirroredSkeleton class is created, it can be added to an hkaMirroredSkeletalAnimation.

```
class hkaMirroredSkeletalAnimation : public hkaSkeletalAnimation
{
public:

    hkaMirroredSkeletalAnimation( hkaSkeletalAnimation* originalAnimation,
                                  hkaAnimationBinding* originalBinding,
                                  hkaMirroredSkeleton* mirroredSkeleton );

    $\sim$hkaMirroredSkeletalAnimation();

    hkaAnimationBinding *createMirroredBinding();
    static void HK_CALL destroyMirroredBinding( hkaAnimationBinding *binding );
};
```

Mirroring Settings

- void setMirrorAxis(const hkVector4 &v)
  - This is the axis in local space about which mirroring takes place. It defines the local axis of mirroring. Components of rotation orthogonal to this axis are mirrored, while components of rotation parallel to this axis are not. An illustrative example is motion about the spine

of a character. In the mirrored animation it is desireable that a forwards bow be retained (this would be a rotation about an axis pointing to the side), but a twist or bend to the side (rotation about a vertical axis or an axis pointing to the front) be mirrored.

- void setRootOrientation(const hkQsTransform &T)
  - This value should be set if the animations to be mirrored contain a constant offset in the root node track, such as a rotation which orients the character from facing along one axis to another.

- void setChildOrientation(const hkQsTransform &T), void setNubOrientation(const hkQsTransform &T)
  - This value should be set if the root node has a single child which also contains a constant offset in its animation track.

- void setMixedMode(hkBool mixed = true);
  - Many rigs will have a particular local space axis (say z) running "down the bone" for all bones. In other rigs this axis is "down the bone" on the one side, but "up the bone" on the other. If so, setMixedMode(true) should be called. See the tutorial in this section for further explanation.

4.3.10.3 Mirror Settings Tutorial

Properly choosing the hkMirroredSkeletalAnimation settings is necessary to successfully mirror an hkSkeletalAnimation. Many animations can be mirrored using only the default parameters, but others require more care. This tutorial will take the user through the steps of using the mirroring settings to successfully mirror and orient an animation. For a code-based example, please see the Animation/Api/Playback/Mirroring/MirroringDemo demo.

Default Settings

The rig shown on the left of the figure below represents the first frame of an hkSkeletalAnimation. The rig shown on the right of below shows the hkMirroredSkeletalAnimation version of this hkSkeletalAnimation with default parameter values. The local axes are drawn and colored red green blue for the local X Y Z axes, respectively. While the pose is clearly mirrored positionally, the orientation of several bones is not correct. This is immediately apparent when the rigs are skinned as shown in .

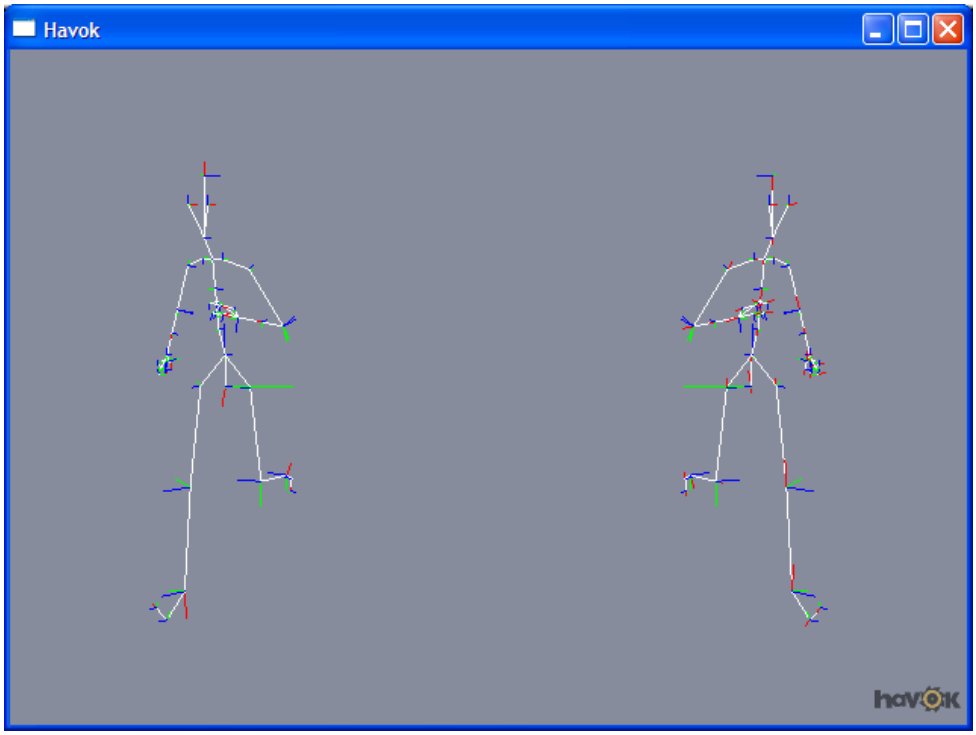


Figure 4.16: Original and Mirrored Skeleton, Default Settings

Figure 4.17: Original and Mirrored Skinned Models, Default Settings

Notice that the original animation (left) skins properly, while the mirrored version (right) does not. The cause of this problem is that the mirror axis is not properly set. The proper choice of local mirror axis for this rig is the Z axis: (0,0,1). For a biped model the proper choice is the local space axis which the knees

and elbows of the rig rotate around. It is also the axis which would cause the spine to bend forward and back (but not side to side).

Mirror Axis

The result of setting the local mirror axis by calling setMirrorAxis(hkVector4(0, 0, 1)) is shown in and . Notice that the model is now properly skinned. However the orientation of the rig is not correct. This is due to the fact that the hkSkeletalAnimation contains an implied root node rotation of 90 degrees about the X (red) axis throughout the animation; the natural configuration of the rig would be lying on its side in world space. The hkMirroredSkeletal animation unknowingly mirrors this rotation to become -90 degrees about the X, resulting in the character appearing upside down.

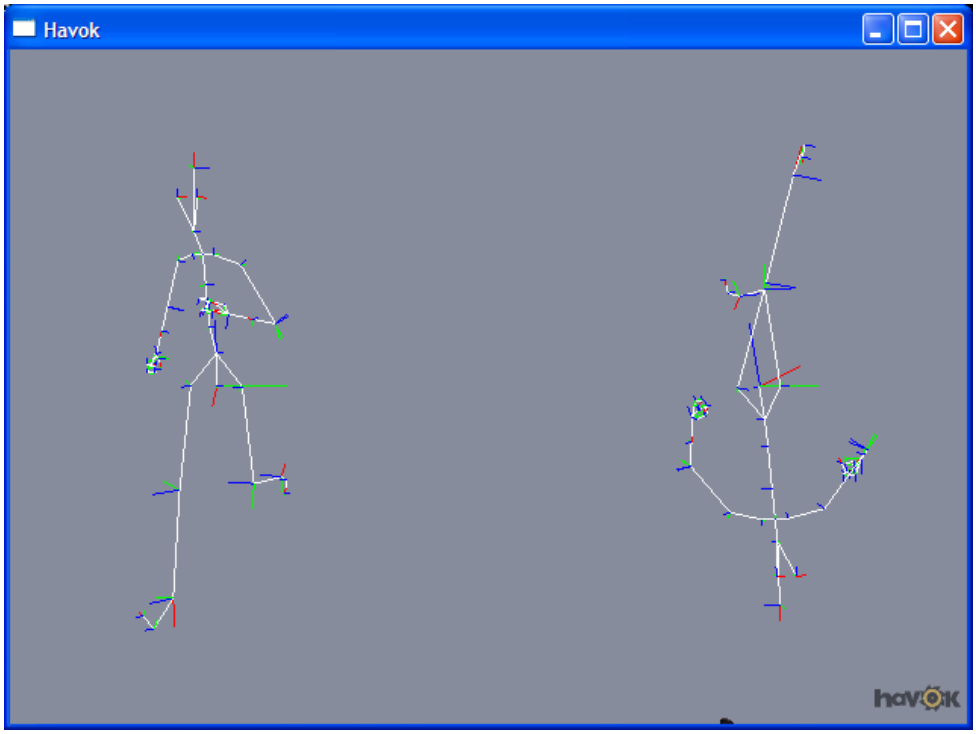


Figure 4.18: Original and Mirrored Skeleton, Proper Mirror Axis

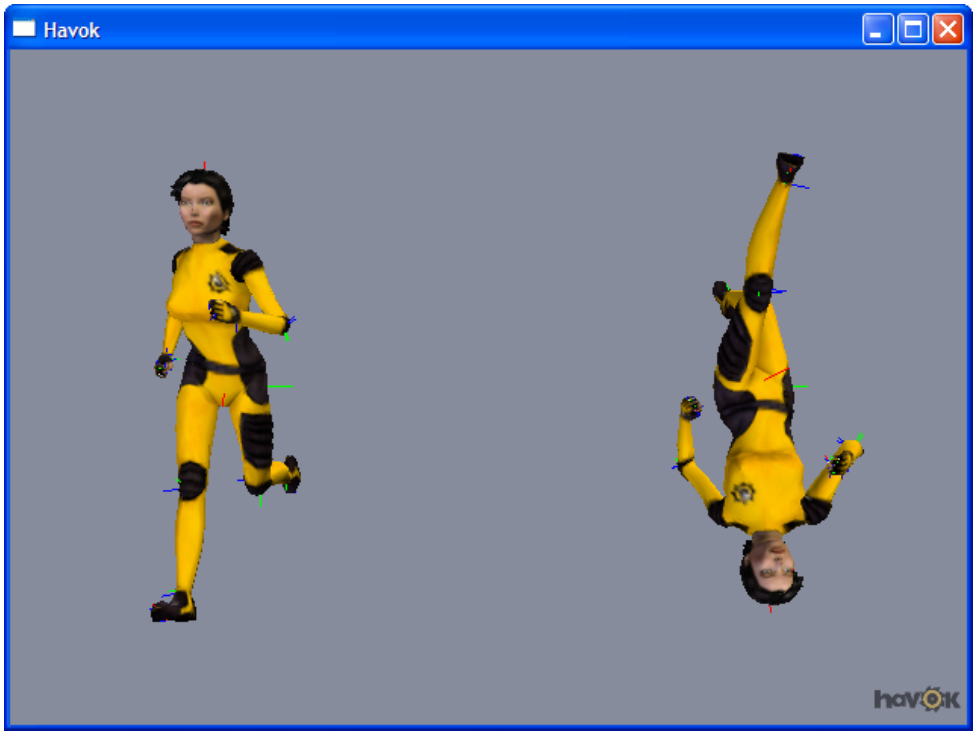


Figure 4.19: Original and Mirrored Skinned Models, Proper Mirror Axis

Root Orientation

To place the model right side up, the constant rotation of the root bone present in the animation track must be accounted for. With the root bone rotation set to identity, this character would be lying on its side. In all animations the root bone track contains a constant implied rotation of -90 degreees about the Z axis (Quaternion [0, 0, sqrt(2)/2, -sqrt(2)/2]) to orient the character right side up. This value may be determined by inspection of the rotation of the root bone in the first frame of any animation in which the character is facing approximately forward. The mirroring algorithm must be told to account for this rotation or the mirrored version will anot appear correctly oriented. The value to set for the Root Orientation is simply a rotation of -90 degrees about the Z axis. The result of this setting is shown in Figures and . Note that the character still does not appear properly right side up in this instance.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

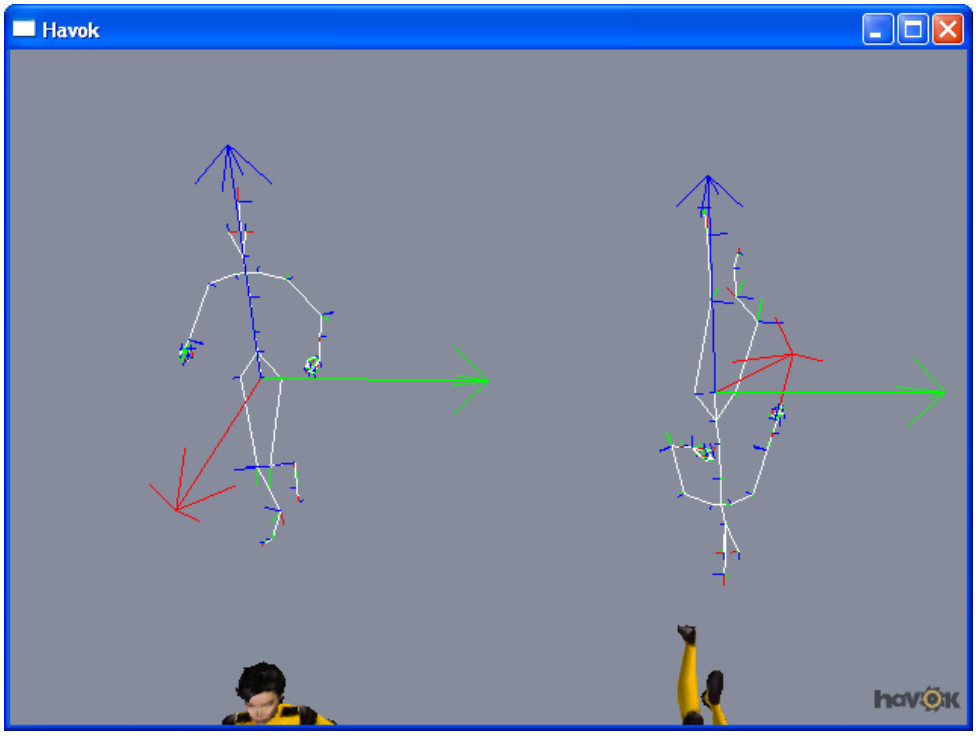


Figure 4.20: Original and Mirrored Skeleton, Root Node Properly Oriented

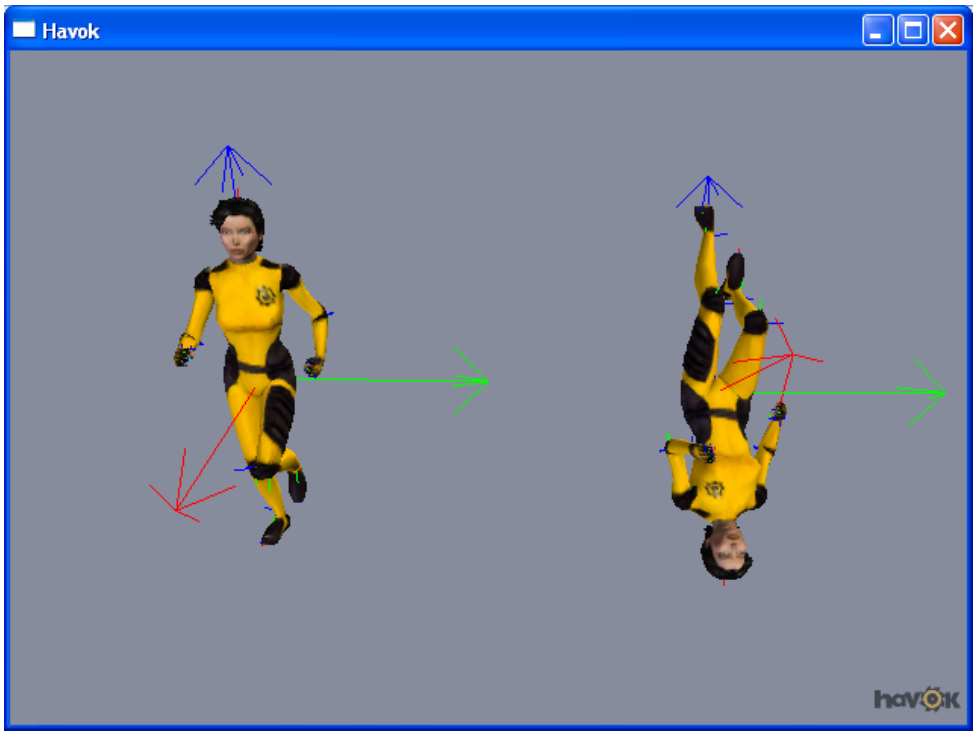


Figure 4.21: Original and Mirrored Skinned Models, Root Node Properly Oriented

Child Orientation

This animation contains a second bone, a direct child of the root bone which also contains a constant

offset. Because of this offset the character is still not properly oriented. For this animation the offset value is 120 degrees about the axis [1,1,1]. This value may be determined from animation data in the same manner as the Root Orientation . The results of setting the Child Orientation to this value (Quaternion [0.5, 0.5, 0.5, -0.5]) is shown in Figures and . The body of the character is now properly oriented. Typically animation assets require a Root Orientation, but may not require the specification of a Child Orientation.

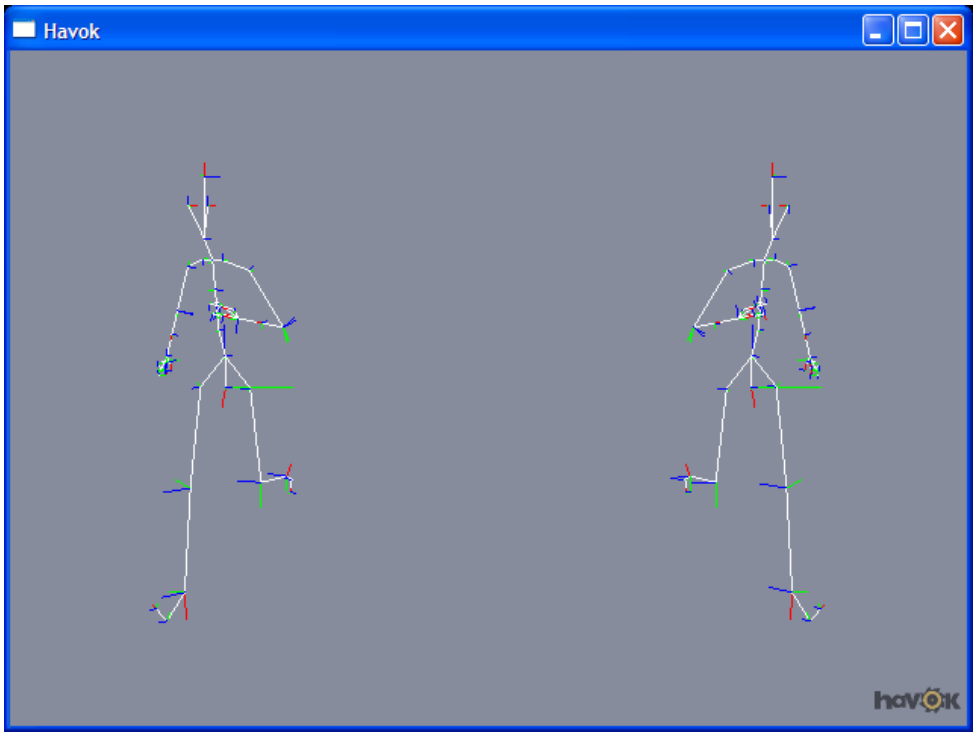


Figure 4.22: Original and Mirrored Skeleton, Root Node Oriented

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Figure 4.23: Original and Mirrored Skinned Models, Root Node Oriented

Nub Orientation

Lastly the nubs of this model do not follow bone convention as seen by the eye bones in . Applying a rotation of 180 degrees about the axis [1, 0, 1] resolves this as shown in Figure . Because there are no objects currently attached, this change is not seen in .

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

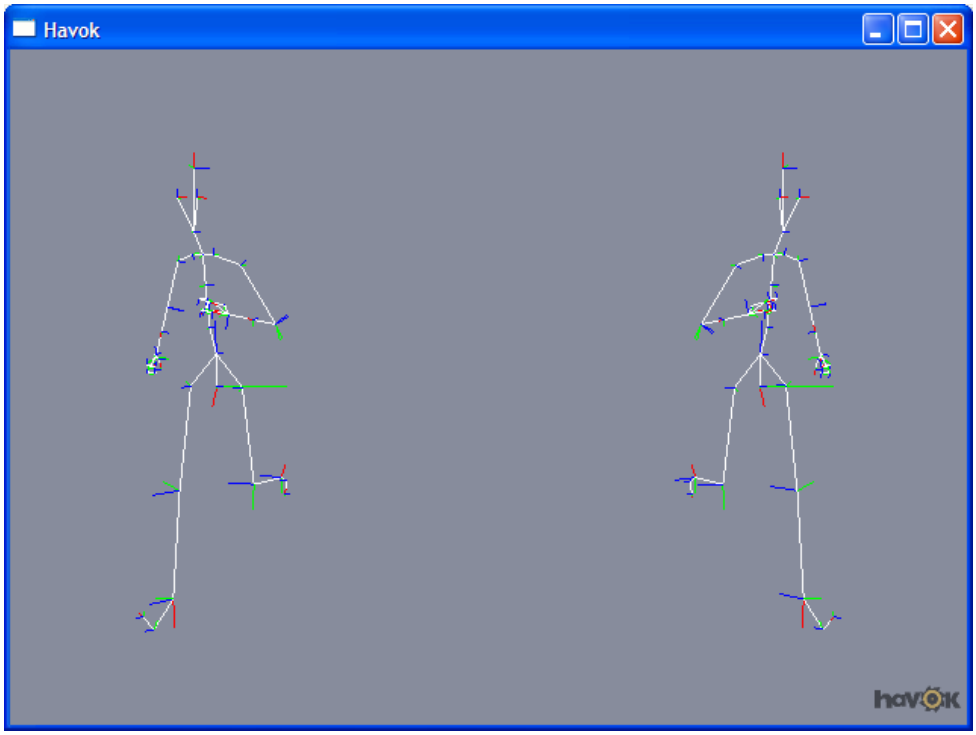


Figure 4.24: Original and Mirrored Skeletons, Nubs Oriented

Figure 4.25: Original and Mirrored Skinned Models, Nubs Oriented

Mixed Mode

The model used for this tutorial did not require the use of the setMixedMode() setting. This setting is

only needed if the construction of the rig used opposite initial orientations for bones on the corresponding left and right sides of the body. A rig requiring the use of setMixedMode is shown in ; the shoulders and arms of the model are illustrated. Here the green local Y axis points "up" the bones on the "down" the bones on the right side.

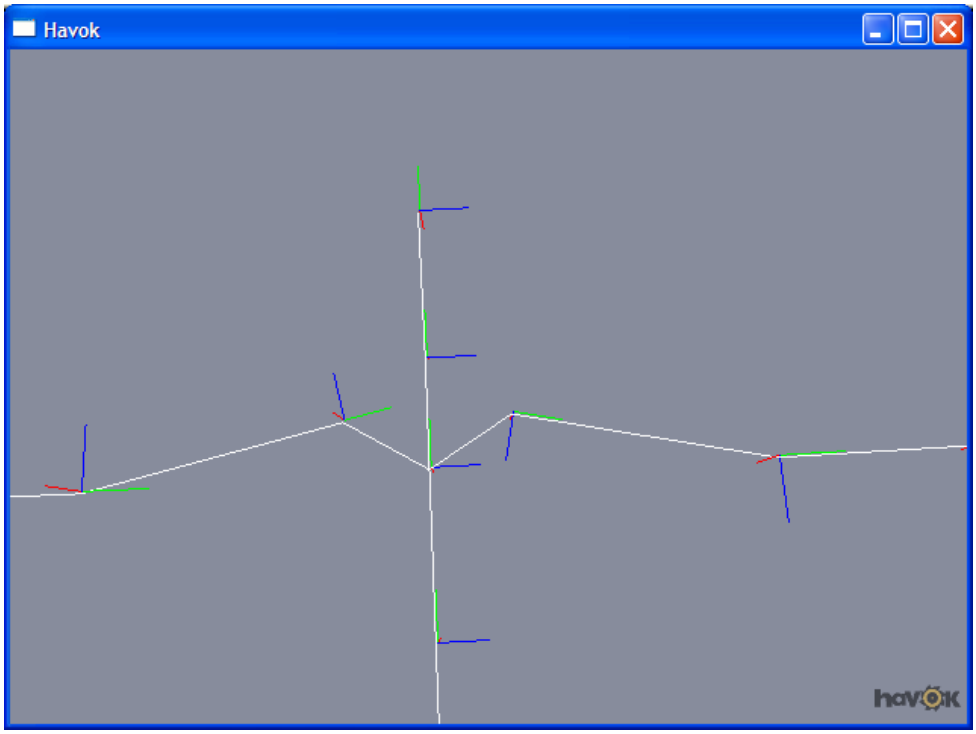


Figure 4.26:

4.3.11 Skeleton Mappers

4.3.11.1 Overview

There are situations where you may want to represent the same character using different skeletons. A common usage of this is when integrating animation with physics : while the animated character can have a high bone count (let's say 50), usually its physical representation (ragdoll) will have fewer (let's say 15)

In those situations, we want to be able to convert a pose from one representation to the other. For example, we may want to convert the animation (50 bones) pose to a physics/ragdoll pose (15 bones) so the ragdoll starts with the same pose the animation had. Similarly, we may want to convert a ragdoll pose (15 bones) to an animation pose (50 bones) so we can put skin around the ragdoll.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

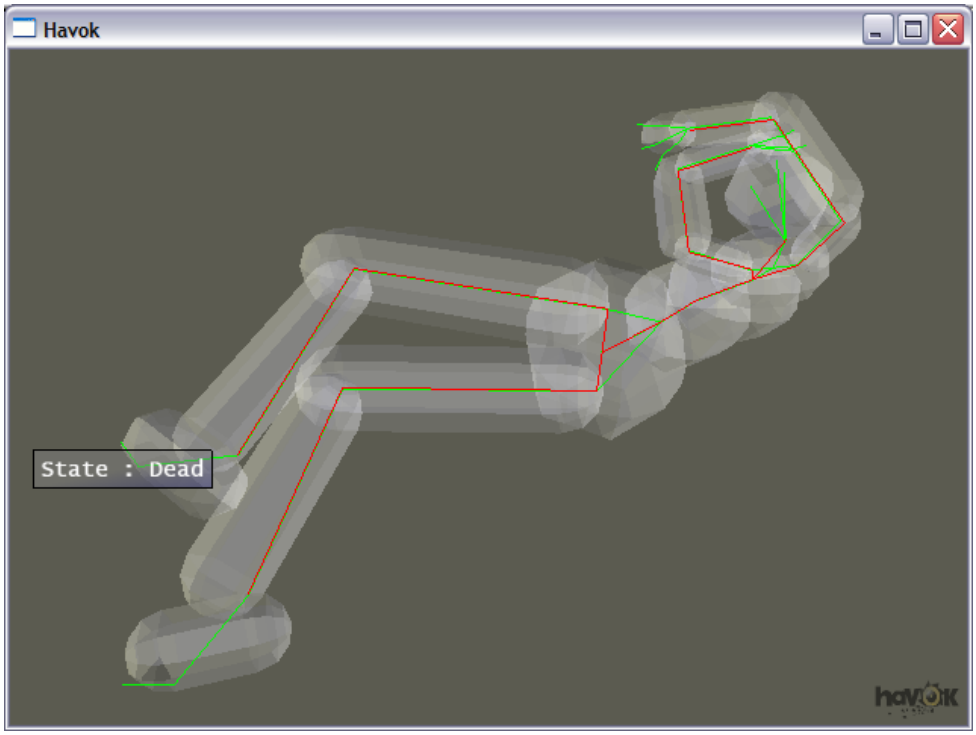
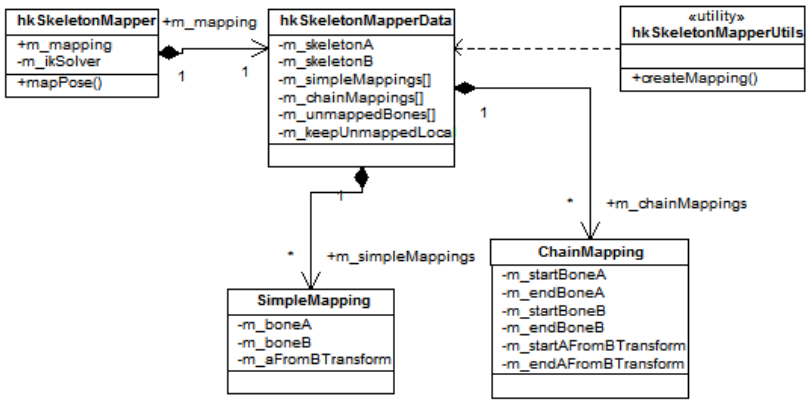


Figure 4.27: Here we map the physics pose (in red) to a higher bone count pose (in green), which we can then use for skinning the ragdoll.

The run-time classes involved on doing this kind of retargeting (where the skeletons represent the same character, with different resolutions) are called skeleton mappers (**hkaSkeletonMapper**).

4.3.11.2 hkaSkeletonMapper and hkaSkeletonMapperData



The hkaSkeletonMapper is a run-time class that is responsible for transforming a pose from one skeleton (A) to another skeleton (B). All the information required by this hkaSkeletonMapper is stored in a hkaSkeletonMapperData structure, which is passed on construction. As we will see later, the **hkaSkeletonMapperUtils** class can be used to create this data based on any two skeletons.

How is it done

The skeleton mapper does its job of mapping a pose between two skeletons through its method **mapPose()** .

The **mapPose()** method works with two poses : *poseA* is an input pose (it is not modified), while *poseB* is an input/output pose - it will be modified based on the *poseA*, but the previous values may also be used (for example, for bones that are not present in A).

This process is done in model space, were bones in the output pose (B) are gradually driven by the input pose (A). This job is done by looking at the information in the **hkaSkeletonMapperData** , which contains three main sets of mapping information: simple mappings, chain mappings, and unmapped bones.

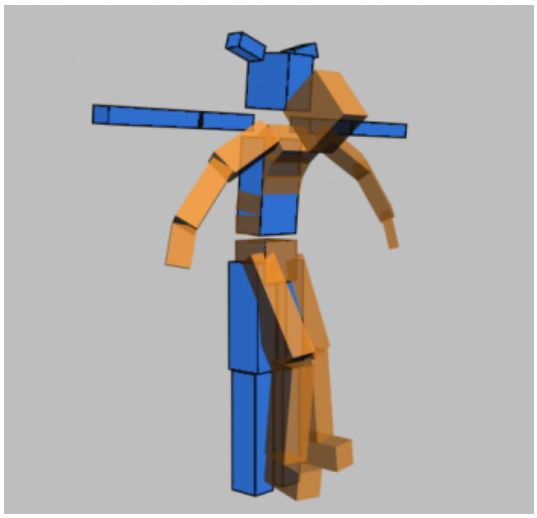


Figure 4.28: The two poses used by the mapper : the input pose A (in orange) and the output pose B (in blue) that will be modified.

Simple Mappings

Simple mappings are one-to-one bone mappings. A simple mapping specifies that the transform for bone B should be driven by the transform in A. An extra transform is also used for the cases where the pivots of these two bones have different location or orientation.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

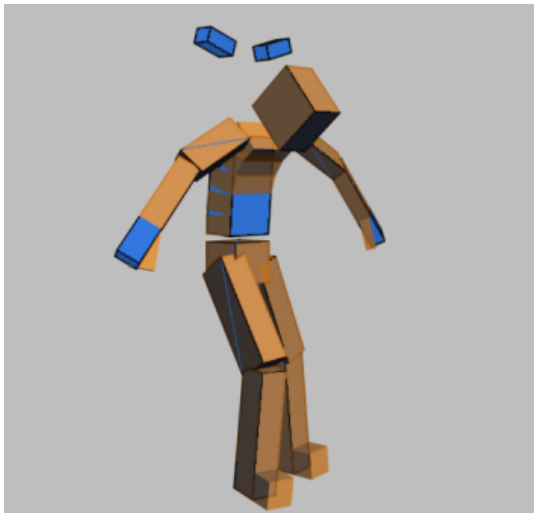


Figure 4.29: Simple mappings are one-to-one bone mappings, and are applied first.

Chain Mappings

Chain mappings are more complex mappings, used to map chains of bones from one skeleton to the other. The best example is the spine: while one skeleton may use 5 bones for the spine, another skeleton may use only 2 or just one. The mapper treats these chains a single problem to solve.

The way chain mappings are solved is by matching the start of the chains in both skeletons and then modifying the chain bones in B (output pose) by trying to reach the same position for the end bone as in A, without modifying the local orientation of the bones in the chain

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

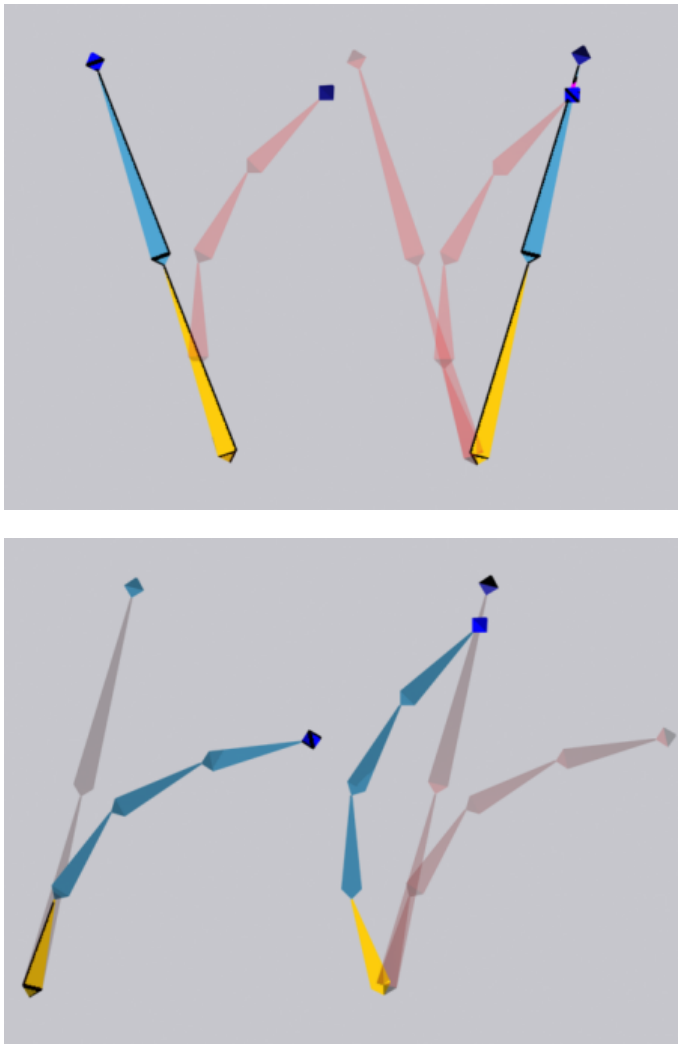


Figure 4.30: When solving chain mappings we try to reach the desired end target position by just rotating the whole chain to align it with the line going from the start bone to the target position. By doing so, the results are reversible.

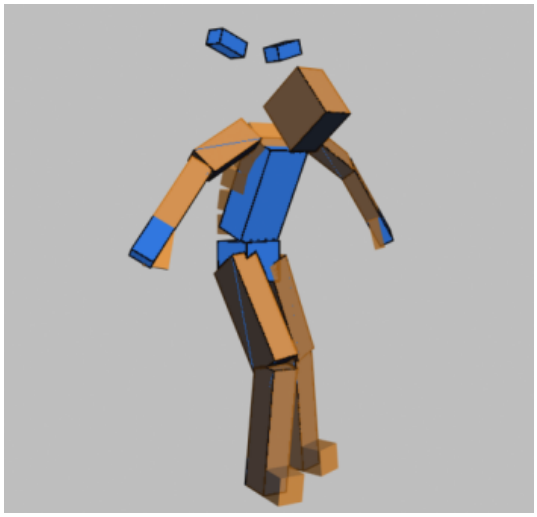


Figure 4.31: Chain Mappings are applied after simple mappings - in the example, the spine is aligned so it points towards the head (its end effector)

Unmapped Bones

Not all bones can be mapped by simple mappings or chain mappings. Sometimes the output pose B has a higher resolution than the input pose A, and therefore some bones cannot be driven by pose A. The mapper data has a list of unmapped bones, and can do two things with them (based on the value of the member **m_mapping.m_keepUnmappedLocal**):

- Keep those bones in the model pose they had in pose B
- Keep those bones in the local pose they had in pose B

Keeping the bones in their world pose would, in the example we are working with, leave the two antennae where they are in the picture above (we will see in the next step, though, that skeletal constraints may move them to their proper location -but wouldn't change their orientation). Keeping the local pose of the unmapped bones, however, will keep these unmapped bones local transform and, therefore, they will move with their parent (the head in the example).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

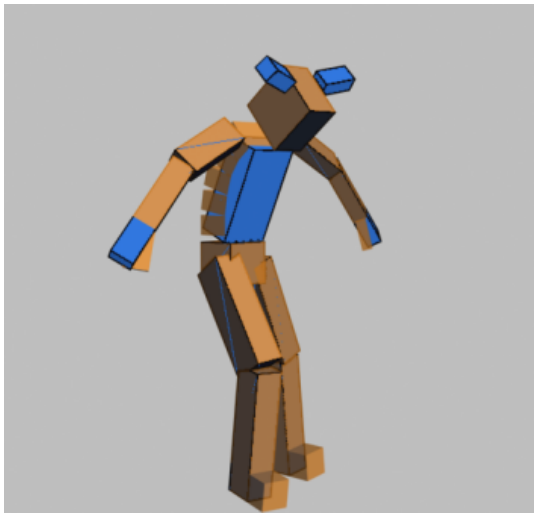


Figure 4.32: Keeping unmapped bones local ensures that the bones in pose B that have no equivalent in pose A follow their parent

Enforcing Skeleton Constraints

During the final step, which is one of the most important, the mapper goes through the transforms in the output pose and ensures that none of them breaks any of the constraints for that bone defined in the skeleton.

Currently, the only skeleton constraint that can be specified for a bone is "lock translation" - whether this bone can translate from its parent or not. If "lock translation" is true, the only translation allowed is that defined in the initial transform of the bone.

By default, all translations are allowed in an skeleton, so it is important that you lock the translations for all bones in your skeleton where you don't want the mapper to add extra translations. In the most common case, you only want translations allowed in the root bone(s) (center of mass / root / pelvis). In some rigs, you may need to allow translations for other bones as well (see "Triangle Pelvis Setup" for another example).

There are many reasons why you want to limit the amount of translations added by the mapping. In the case of mapping from animation to physics, for example, adding extra translations to the physics pose would likely mean setting a pose where physical constraints are broken, causing popping artifacts or instability. Similarly, adding unwanted translations when mapping to an animation pose may cause artifacts on the skinning (stretching, etc).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

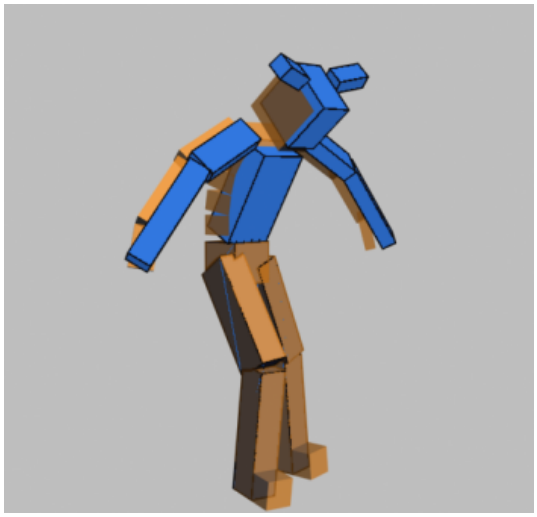


Figure 4.33: Enforcing skeleton constraints will make sure that no unwanted translations were added to the output pose. In the example, since the single-bone spine reaches further than the 5-bone spine (because it cannot bend), the head and arms no longer match pose A in world.

The Triangle Pelvis Setup

Since, as we've seen, the skeleton mapper works in world space (modifies subsets of the pose independently), it can handle the situation where the input and output skeletons have different hierarchies.

The most common example is what is know as the "Triangle Pelvis" setup in Character Studio.

Character Studio allows the artist to optionally use this setup for their bipeds. When doing so, the thighs in 3ds max are parented to the first link in the spine instead of the pelvis (their "natural" parent) [9].

It is important to notice that skeletons using that structure should keep the "lock translations" flag off (should allow translations) for both thigh bones since, although they don't translate with respect to the pelvis (their "natural" parent), they may do with respect to the spine (their actual parent).

[9] The reason seems to be that the Physique skin modifier works better with this "unnatural" parenting.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

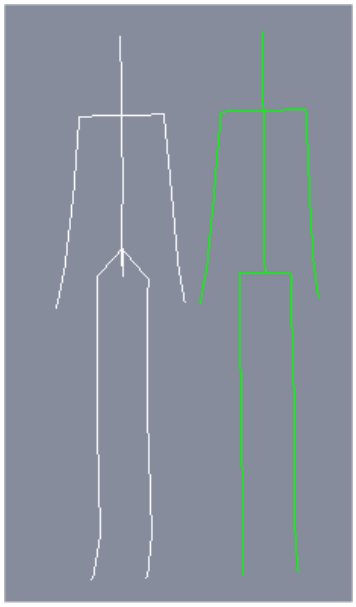
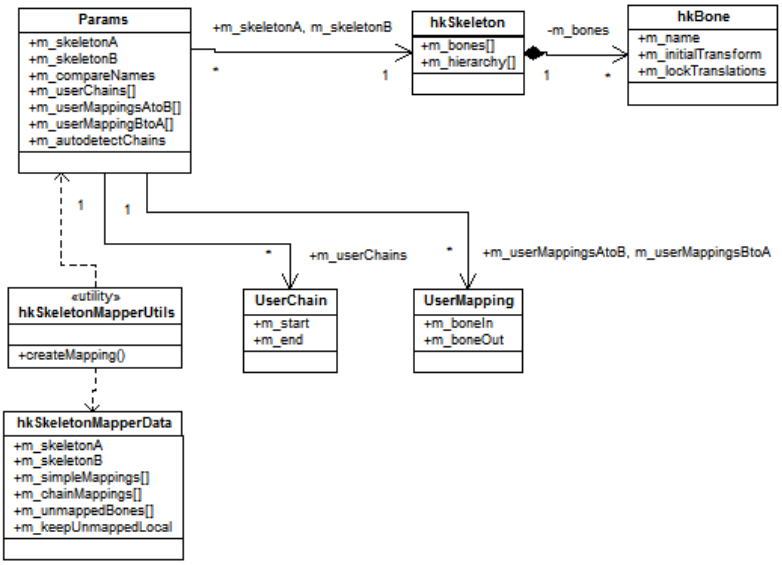


Figure 4.34: The skeleton on the left has triangle pelvis setup - the legs are parented to the spine. The skeleton on the right has a natural setup - the legs are parented to the pelvis. The mapper works in world space and therefore can map poses from one skeleton to the other - but it is important to allow translations on the thighs in the skeleton with triangle pelvis setup.

4.3.11.3 Setting Up Mapping Data

In the previous section we saw how the mapper works at run time, and how it uses a combination of simple mappings, chain mappings, a list of unmapped bones and skeletal constraints data to do so.

All this information is passed on construction using an **hkaSkeletonMapperData** object. While you can create this data by hand, Havok provides a utility, **hkaSkeletonMapperUtils** , to facilitate this setup.



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

This utility works based on the names of the bones from two skeletons A and B, and creates two sets of mapping data, one for going from A to B, the other one for going from B to A.

Bone Names

Each bone of a skeleton has a name parameter. The utility will use these names in order to match bones for the mapping. Since in most cases the names of the bones in the two skeletons will be similar, but not identical (for example, the head can be called "Girl Head" in one skeleton, but "Ragdoll_Head" on the other, a user-define string compare function (m_compareNames) is passed to the **hkaSkeletonMapperUtils** .

The Initial Pose

In order to generate the proper mapping information, the **hkaSkeletonMapperUtils** utility compares the initial pose of the two skeletons (the pose generated by the initial transform of each bone) in world space. It is, therefore, very important that both skeletons are properly aligned in that initial pose.

Simple Mappings

The utility will create a simple mapping for all pairs of bones in A and B whose name match[10]. Their world transforms will then be matched, and the offset will be stored and used for this mapping at runtime. Notice that, for optimal mapping, you want this offset to have no translation (otherwise a rotation in one skeleton will map to a rotation and a translation in the other skeleton); as a precaution, a warning will generated by Havok if large translations are found.

You can also force the utility to create simple mappings (from A to B or from B to A) for bones whose names don't necessarily match[11].

Chain Mappings

By default, the utility will try to autodetect chain mappings. If **m_autodetectChains** is on, a chain mapping will be created when:

- Two simple mappings have been created for bones (which we call *Start* and *End*)
- You can go from *Start* to *End* through the hierarchy of each skeleton without finding another bone that has already been mapped (the bones you go through form the chain on each skeleton)
- At least one of the chains has length >=3 (including *Start* and *End*)

You can also force the utility to create specific chains, by adding **UserChain** objects. You can do this easily by specifying the names of the start and end bones[12].

Skeletal Constraints

It is important to remember that maintaining skeletal constraints is one of the jobs of the mapper at run

[10] Unless they are part of a user chain

[11] One example of this can be found when mapping from a low res skeleton to a high res skeleton, where the high res skeleton has an extra "forearm twist" bone to help with skinning. This bone has no match in the low res skeleton, but it should be driven by the "forearm" bone when mapping from the low res to high res.

[12] For example, if one skeleton has a spine of 3 links (spine1, spine2, spine3, head) and the other skeleton has a spine of 5 links (spine1..spine5, head), the system by default will create a chain from spine3 to head. However, commonly you want to force a chain from spine1 to head.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

time, and it's one of the main reasons for using the mapper. Therefore, make sure that the bones in both skeletons have their **m_lockTranslation** flag set up properly.

As a precaution, the mapper (**hkaSkeletonMapper**) will generate a warning on construction if the target skeleton has no translations locked.

4.3.12 Animation Compression

This section describes animation compression features available within the Havok Animation SDK. Without the use of compression the space requirements of animation assets may exhaust the memory resources available in gaming consoles. Animation compression allows for these assets to stored in a compact form which significantly reducing the amount of memory required. The goal of animation compression is to produce a result that is perceptually similar to the raw input animation while achieving a significant savings in the size of the representation.

The Havok Animation SDK supports three compression schemes. Spline compression is described in section . The Wavelet and Delta compression schemes and are described in section . Each of these compression techniques implements *lossy* animation compression in which there is a trade off between fidelity to the original animation versus the size of the compressed animation in bytes. Each of the compression schemes allows the author control over the settings which define the space quality trade off. This trade off is conceptually similar to those encountered in popular "lossy" multimedia file representations for images, video or music.

Runtime performance statistics for each of the available compression schemes are detailed in the Quickstart guide for the appropriate platform.

4.3.12.1 Spline Compression Overview

Havok's spline compression technology provides state of the art compression of animation assets. Compression ratios of 10x-20x may be expected for near-lossless compression, while compression rates of 30x or more may be expected for lossy compressions. Features of Havok's spline compression technology include:

- Full control over the accuracy of the approximation
- Smoothly degrading quality from lossless to lossy compression settings
- No runtime cache requirement
- Multiple decompression threads may be executed in parallel
- Optimization for several platforms
- Per-bone and per-track decompression

The fundamental technique of spline compression is the fitting of low order piecwise polynomials to regularly sampled input animation data. The resulting polynomials are stored as NURBS, which are a common curve representation. NURBS allow for a smooth fit to input data while preserving data discontinuities when appropriate.

Spline Compression Pipeline

The spline compression pipeline consists of three stages: Track Analysis, Spline Approximation and Quantization. This process is Illustrated below in . These stages are are detailed in , , and respectively.

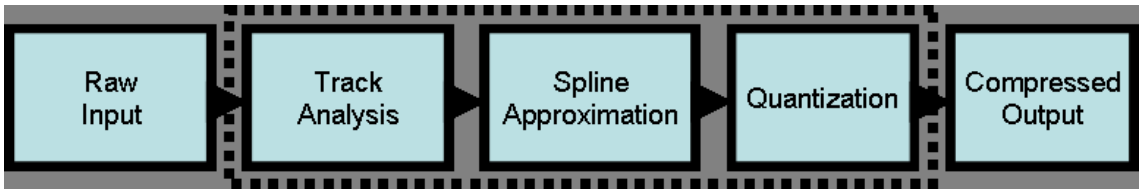


Figure 4.35: Spline Compression Pipeline

Havok's spline compression technology is implented by the `hkaSplineSkeletalAnimation` class. Spline compression settings may be declared uniformly for all tracks using the `hkaSplineSkeletalAnimation::TrackCompressionParams` class, or individually for each track using the `hkaSplineSkeletalAnimation::PerTrackCompressionParams` class. Spline compression settings are uniquely defined by three values: tolerance, degree and quantization type. These three settings may be specified independently for translation, rotation, scale and float data.

The tolerance value defines the accuracy to which the track analysis and spline approximation steps approximate the input data. This setting typically has the greatest control over the end quality of the compressed animation. The smaller the tolerance value, the closer the resulting spline approximation will follow the input data. The larger the tolerance value, the smaller the compressed output memory footprint will be.

The degree setting determines the polynomial degree of the spline representation. Piecewise cubic curves (degree 3) produce the smoothest results, while piecewise linear (degree 1) curves may provide a more compact representation at the cost of producing a less smooth result.

Quantization determines the number of bits used to represent scalar values in the spline data stream. Spline animation compresses input data in blocks, typically 256 frames of data. The block size is described in section .

Track Analysis

The track analysis stage of the spline compression pipeline identifies unchanging components of the input animation and stores this data in an efficient and compact manner. Translation, scale and float data is analyzed component-wise (x,y,z for translation and scale) to determine whether the maximum and minimum values are within tolerance of the mean. Any components which pass this test are denoted as *static*, and are represented by the mean value for the entire block. Rotations are treated similarly, but all four components must pass the tolerance test for the rotation to be considered static. This requirement is due to the fact that rotations are quantized as quaternion values and individual (x,y,z,w) components may not individually be marked as static. Additionally, tracks which are found to be within tolerance of special *identity* values: (0,0,0) for translation, (1,1,1) for scale and (0,0,0,1) for rotation are represented by a single bit in the spline data stream. All output of the track analysis stage of spline compression are guaranteed to be within tolerance of the input data as measured by the Infinity (Supremum) Norm.

Spline Approximation

The spline approximation step approximates input data as a series of piecewise polynomial curves using a standard NURBS representation. NURBS curves consist of a temporal component called knots, and a spatial component called control points. The spline control points are the same dimensionality as their respective input types (translation: 3, rotation: 4, scale: 3, float: 1) minus any degrees of freedom marked as static by track analysis. The specific knots and control points used to represent the compressed data

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

are found by a spline optimization and fitting routine which is specialized for compressing animation data. The spline fitting routine guarantees that the resulting spline curve is within tolerance of the input data as measured by the Euclidean (L2) Norm. Compression is achieved as typically many fewer control points are necessary to represent the input curve than the number of input sample points.

Quantization

Once the control points of the spline compressed animation have been created, they are quantized to a specified number of bits. The quantization values are defined in the `hkaSplineSkeletalAnimation::TrackCompressionParams` class. Translation, scale and float values may be quantized as either 8 or 16 bit values. The error introduced by n bits of quantization is of the order 2^(-n). Rotation values are stored and quantized as compact quaternion representations. As quaternions are always 4 dimensional unit vectors, a significant savings may be had by storing only 3 components and implying the value of the 4th. Quaternions may be compressed as 32, 40, or 48 bits. 40 bit quaternions are considered the default and other values should be chosen only as special needs dictate. The 32 bit quaternion representation uses a polar decomposition which introduces a speed penalty. Additional bit sizes including 16, 24, and 128 bits are available in the SDK, but are not exposed through the content tools. The error introduced by quaternion quantization to n bits is of the order of 2^(-n/3). An additional step in which quaternion control points are renormalized to unit vectors introduces a small additional error.

It is the user's responsibility to ensure that the errors introduced by the tolerance and quantization settlings are roughly of the same order. For example, 8 bit quantization introduces an error of approximately 0.004 and a tolerance setting in the range [0.01,0.001] is quite appropriate in this case. However a tolerance of 0.00001 would be of limited use as the error introduced by compression will be quantization dominated. In typical cases the animator can avoid this problem by simply avoiding opposing trends in quantization and tolerance settings; tightening tolerances while simultaneously decreasing the number of quantization bits is generally not productive.

Per Track Compression Settings

The `hkaSplineSkeletalAnimation` class offers two constructors using the `hkaSplineSkeletalAnimation::TrackCompressionParams` or the `hkaSplineSkeletalAnimation::PerTrackCompressionParams` classes for uniform or per-track compression settings, respectively. Each constructor also takes a single instance of the `hkaSplineSkeletalAnimation::AnimationCompressionParams` class to define settings which affect the entire animation. The `hkaSplineSkeletalAnimation::PerTrackCompressionParams` class itself contains an array of `hkaSplineSkeletalAnimation::TrackCompressionParams` instances which defines a palette of compression parameters. Additional arrays map each transform and float track to the appropriate compression parameters. The idea of having a compression palette was envisioned to support uses cases such as Level Of Detail compression (with a "major" bones compression setting and a "minor" bones compression setting) or an upper-lower body split. The number of palettes created is arbitrary (though having more palettes than tracks serves no useful purpose). To apply a compression setting to a specific bone, the track to bone mapping must be reversed to determine the appropriate track for the current animation. The content tools provide this functionality in the form of a GUI to the user.

Block Size

Spline compression operates on a "block" of data at a time. Block size is 256 frames of input animation by default. Animation sequences longer than this are automatically broken into multiple blocks without the need for user intervention. For most platforms it is not necessary to change the default block size setting. The exception is the PLAYSTATION®3 SPU implementation of spline compression where certain memory limitations need to be respected. Please refer to the PLAYSTATION®3 QuickStart

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Guide for more information.

Individual Bone / Track Sampling

Spline compressed animations can allow individual transform tracks to be sampled through the `hkaSplineSkeletalAnimation::sampleIndividualTransformTracks` member function, and individual bone poses to be sampled through the `hkaAnimatedSkeleton::sampleAndCombineIndividualBones` member function. Allowing individual tracks / bones to be decompressed requires a small additional memory overhead, thus this feature is considered optional and disabled by default. This feature may be enabled through the `hkaSplineSkeletalAnimation::AnimationCompressionParams` class.

4.3.12.2 Wavelet and Delta Compression Overview

The diagram below shows a typical compression pipeline used in Havok Animation. Usually you see this pipeline in action as you compress animations during asset processing. It is useful to describe each of the components here to give an intuitive understanding of the parameters and tolerances exposed in the tool chain.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

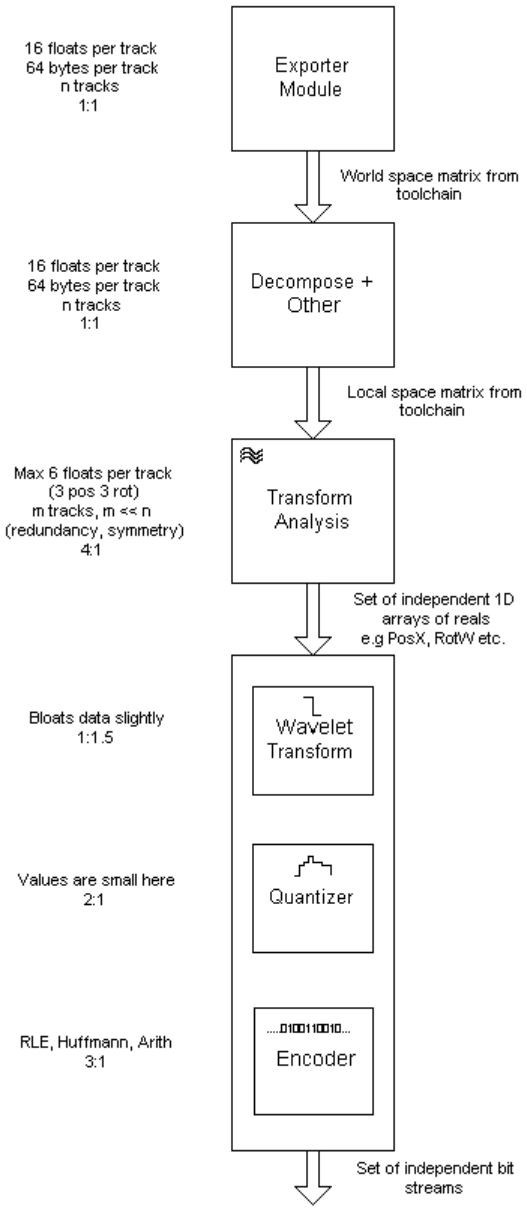


Figure 4.36: A Typical Compression Pipeline

The first two stages of the pipeline analyze the scene data from the export utilities and produce an interleaved animation (hkaInterleavedSkeletalAnimation). This data is passed to an analysis stage where redundant degrees of freedom are identified and removed. The data is split into separate components and each of these is passed to the compressor. The compressor consists of a transformation stage, a quantization stage and optional encoding stage. This produces an endian safe bit stream which can be loaded later and decompressed.

Compression Pipelines

Havok Animation has two preconfigured compression pipelines. The first uses delta transformation and variable bit quantization. The second uses wavelet transformation, coefficient truncation, variable width quantization , and a simple dictionary encoder. All settings can be set on a per track basis, and both pipelines can optionally split the signal into fixed size blocks for runtime caching of decompressed data.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The following diagram describes the data flow through the system:

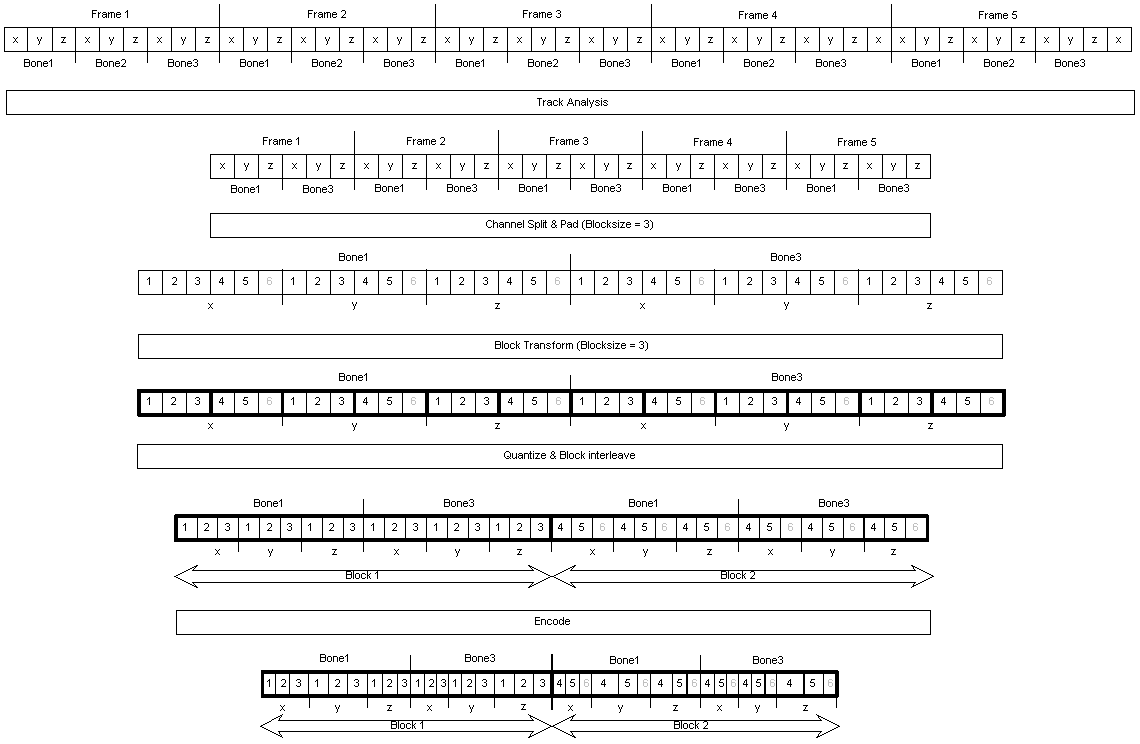


Figure 4.37: Data Flow through the Compression Pipeline

- The data is presented to the track analysis stage in interleaved format.
- Track Analysis identifies Bone 2 is redundant and removes it from the data stream. We get lossless compression here.
- Each remaining component (e.g. The real component of a quaternion, or x component of a position) is separated back into a continuous stream of floats, essentially undoing the interleaving. These channels are padded out to match the blocksize. The padding is shown in gray.
- These floats are then transformed by either a delta transformation or wavelet transformation. This transformation is done in fixed sized chunks. In our example we use a block size of 3.
- Once transformed the data is quantized and re-interleaved to improve memory access. Once re-interleaved all the data for a single block now sits contiguously in memory. This stage is performing lossy compression.
- Delta compression stops here and stores this quantized data.
- Wavelet compression continues with an encoding stage. This further reduces the memory requirement for each block.

Track Analysis

The track analysis stage removes redundant degrees of freedom from the set of decomposed transforms. Often if a rig is well constructed its decomposed representation will contain a significant amount of redundant data. Consider a rig where all the bones have fixed lengths. Once decomposed, most of the positions[13] will remain fixed throughout the animation. Obviously storing this information per frame is

[13] The root node may have movement.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

wasteful.

The track analysis stage takes the decomposed animation and identifies these redundant positions, rotations and scale components. A 16 bit identifier is stored for each bone in the animation. The analysis uses 2 bits per component (position, rotation and scale) to identify if it is static, clear[14] or dynamic. The remaining 10 bits explicitly identify which of the subcomponents (e.g. x, y, z) are dynamic.

In addition to this bitmask, which we call the static mask, the track analysis also stores a static array of hkQsTransforms, the static pose. This static pose is used to reconstruct the redundant parts of the animation during playback.

In order to identify different tracks as static or clear you must specify some tolerances. These include:

- **Absolute Position Tolerance:** If the absolute value x, y, or z elements of the position of a bone vary from its static pose by more than this tolerance then the position component is flagged as dynamic.
- **Relative Position Tolerance:** The range of the x, y and z components for the position of the bone are calculated. The relative tolerance allows each component to vary by a proportion of its calculated range. If any component varies from the static pose by more than this relative proportion then the position is flagged as dynamic. This tolerance is useful when you do not previously know the range of values in the animation.
- **Rotation Tolerance:** The rotation is stored as a Quaternion. All components are guaranteed to lie in the range -1 to 1, therefore we use an absolute tolerance here. If any component varies from the static pose by more than the rotation tolerance then the rotation is marked as dynamic.
- **Scale Tolerance:** As above we assume that the range of values for scale are typically well known and usually lie in a relatively small range. Therefore the scale tolerance is an absolute tolerance used to identify dynamic scale components.
- **(Use 3-Component Quaternion Compression Flag):** This is a boolean value. See details below

In addition to the tolerances above above, track analysis may optionally compress the *rotation* component of all tracks by using '3-component quaternion compression'. It uses the observation that encoding rotations in quaternion format is redundant in that 4 floating point values are used to encode a 3-dimensional quantity. In Havok, quaternions are always stored as unit-length quaternions, hence one can reconstruct any of the 4 components from the other 3.

Note:

Since the remaining coefficients will be quantized (and in the case of a wavelet-compressed animation, wavelet-transformed before quantization), reconstruction may introduce some error if care is not taken to identify which components might be suitable for dropping and later reconstruction - Havok compresses only those tracks which will not introduce this error.

The optional identification of suitable tracks during the track analysis phase of compression gives up to a 25% improvement in compression of the dynamic *rotation* tracks of an animation. Depending on the nature of the animation (how much of the dynamic data is positional versus rotational or scalar, and the length of the animation), users should expect an overall reduction in size for the same quality of about 10-25%. By default both Delta and Wavelet compressed animations will use this new improvement. We anticipate that this improvement should always be used, however we offer backward compatibility in the runtime, which may be useful if very aggressive compression settings are being used in conjunction with this new algorithm - the error introduced by the dropping/reconstruction may in these cases

[14] 'Clear' implies zero for position, the identity for rotation and (1,1,1) for scale.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

become nonnegligible. Both Delta and Wavelet compressed animation constructors have a hkBool useThreeComponentQuaternions argument which can be set to false to disable this compression.

Note:

At runtime, track reconstruction works for both 'old' *serialized* assets (or assets where 3-component quaternion compression has been turned off) and 'new' serialized assets with a negligible run-time performance overhead. So no versioning or re-export of old assets is required to maintain functionality, but re-export of the old assets will be necessary to avail of the new compression improvement.

Delta Compression

As shown above, delta compression stores the quantized data directly in the stream. When performing quantization the range of each subcomponent of position, rotation and scale is individually stored alongside the data. This allows each independent degree of freedom to minimize quantization error.

When delta compressing data you specify:

- Track analysis tolerances.
- A quantization bit width - we support any bit width up to 16-bit quantization.
- An optional block size for block compression.

When block compressing a delta compressed animation, a full floating point value is stored at the beginning of each block. For example, 8 bit encoding with a block size of 10 will require 13 bytes to store the quantized track (4 bytes for the initial floating value plus 9 bytes for the remaining delta values).

Wavelet Compression

Wavelet transformation is similar to delta transformation. After transformation the data is passed through a truncation/thresholding filter. This allows the user to clamp a proportion of the values or *coefficients* to zero. Often when a signal has been wavelet transformed many of the remaining coefficients are zero, or close to zero. By clamping these we introduce loss into the signal, however this also allows us to perform an encoding stage, which significantly reduces the storage space required for the quantized signal.

The diagram below shows the effect of truncation on 4 different signals (all taken from an animation). The ratios indicate the proportion of data that has been clamped to zero (2:1 implies 50%). As you can see a significant proportion of the signal data can be discarded with relatively little effect on signal reproduction.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

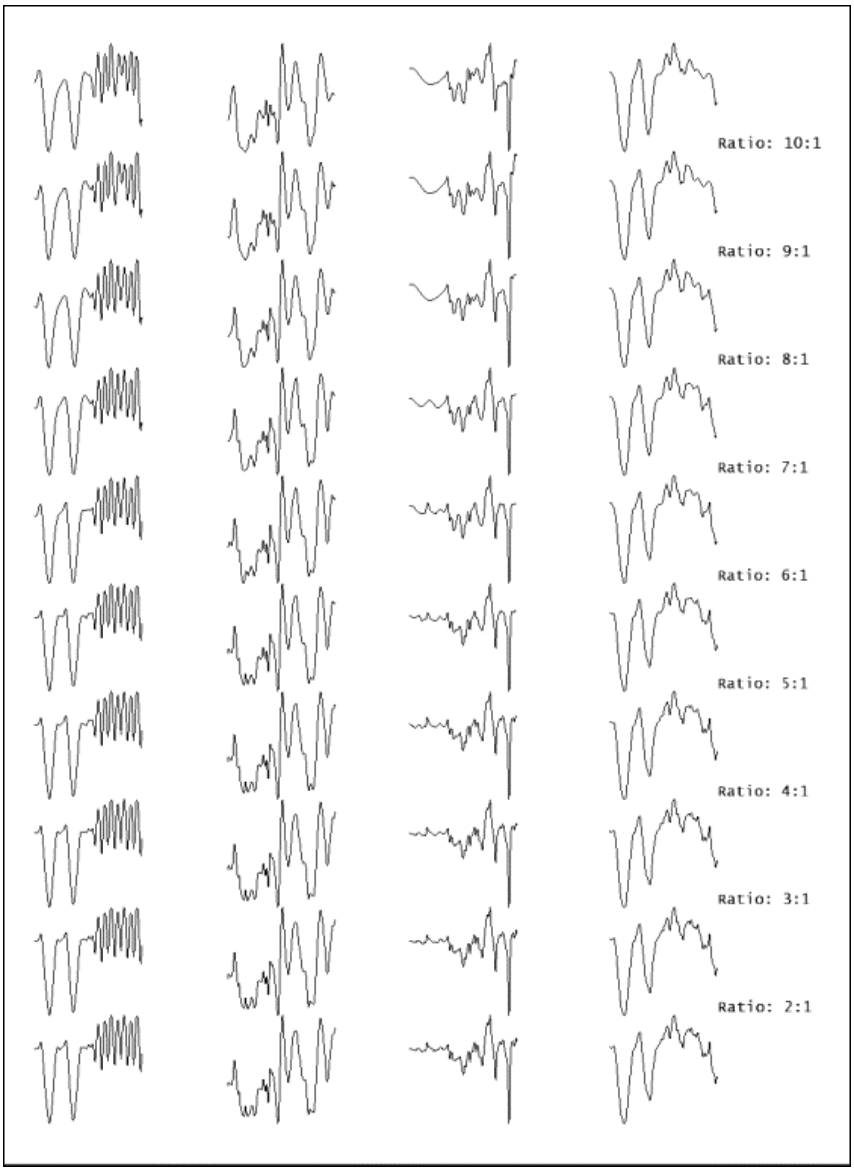


Figure 4.38: Wavelet Coefficient Truncation

Once transformed, clamped and quantized many of the wavelet coefficients are now zero (or to be exact, a quantized representation of zero). We further reduce the storage space required by exploiting this fact with a very simple encoding scheme. For each block we store a bitfield indicating which values are actually zero. We then simply discard the zeros from the original signal as we can reconstruct them using the bitfield. This encoding is similar to a dictionary encoding scheme with a single dictionary symbol.

Havok supports two heuristics for determining which wavelet coefficients to set to zero: *truncation* and *thresholding*.

- **Truncation** This drops a fixed proportion of the smallest coefficients, independent of their magnitude/importance. It gives a constant and predictable compression ratio (before the entropy encoding pass described above), but does not give the best quality for a desired compression ratio as it does not necessarily take into account the acceptable error eg. hinted at by the track analysis tolerances set by the user.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- **Thresholding** This drops all coefficients below a given absolute threshold value. of the smallest coefficients, independent of their magnitude/importance. It gives better control acceptable error hinted at by the track analysis tolerances set by the user. In fact it will uses the actual track analysis tolerances directly corresponding to the type of data (positional/rotational or scalar).

Overall thresholding tends to give a better compression ratio (by about 10-20%) for the same visual quality so is recommended over truncation.

When wavelet compressing data you specify:

- Track analysis tolerances - these will (by default) be used to clamp the coeficients to zero.
- A flag 'm_useOldStyleTruncation' and a corresponding truncation proportion for wavelet components. This lies between 0 (lossless) and 1 (fully lossy). It is not recomended to use this heuristic (see notes above) - but for certain type of animation this may be preferable.
- A quantization bit width - we support any bit width up to 16-bit quantization.
- An optional block size for block compression and encoding.

Per Track compression

Separate compression parameters can be specified for each individual track of an animation. Previously the same parameters applied to all tracks. To use this feature at runtime, construct a PerTrackCompressionParams palette to construct a delta compressed or wavelet compressed animation. The advanced compression demo shows an example of this in operation.

4.3.13 Caching Decompressed Animation Data (Wavelet and Delta Compression)

The Wavelet and Delta animation compression schemes described in section require a runtime cache of animation data to achieve best performance. Note that this section is not applicable to the Spline animation compression scheme described in section . Spline compression does not need nor benefit from the use of an animation cache. Since we use a block compression system for the Wavelet and Delta compression schemes we have to decompress the same block of data several times for successive frames of an animation. The additional processing time used to continuously decompress the same block of animation data may be avoided by caching the data on decompression and simply re-using it as necessary. To properly manage this data in memory we have implemented a cache management system. The system manages fixed sized chunks of data associated with a key.

4.3.13.1 Cache Management Architecture

The cache itself is simply a managed memory area. The cache interface provides read and write access to fixed size data chunks allocated from the managed area. Our default implementation of this interface, described below, also provides the ability to lock and flush cache memory. When sampling animations an optional pointer to a cache is passed. This cache used can be our default cache or any suitable user supplied implementation of the cache interface. In all cases where a cache implementation is unable to service a request the system automatically falls back to uncached decompression.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.3.13.2 Default Chunk Cache Implementation

The heap memory assigned to the default cache is distributed between cache pools. Our cache may handle an arbitrary number of pools, with each pool representing data chunks of differing sizes. The chunk size is specified in bytes and is chosen by you at construction time.

Each pool is further subdivided into buckets and slots which map out the entire memory space available to that pool. The buckets and slots form a square grid with each bucket having X slots available. Each slot stores a key, which may simply be thought of as an identifier for a certain section of memory (outside of the cache managed memory). The grid structure was chosen to enable the keys be quickly hashed to a certain bucket, thus decreasing key lookup time.

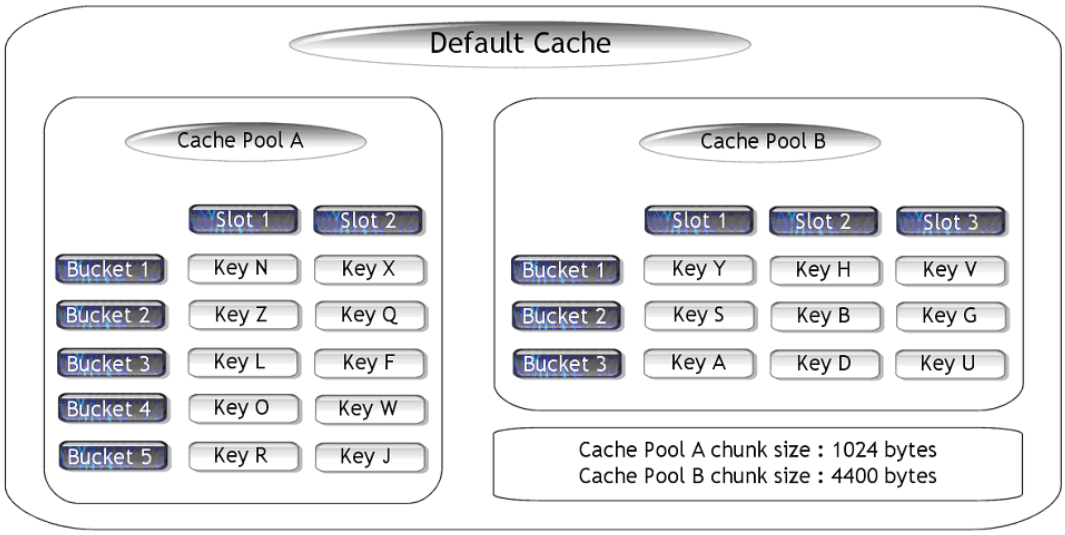


Figure 4.39: The Default Cache Model

The diagram above shows a typical cache instance that contains two different pools; one enabled to deal with data chunks of 1024 bytes and less and the other to handle data chunks between 1025 and 4400 bytes. Each slot simply holds a key which is used to retrieve the corresponding memory chunk.

In the example above, the cache manages a total of 49,840 bytes of memory as follows:

```
Cache Pool A :  2 slots * 5 buckets * 1,024 byte chunk size = 10,240 bytes
Cache Pool B :  3 slots * 3 buckets * 4,400 byte chunk size = 39,600 bytes
```

4.3.13.3 Using the Default Cache

The default chunk cache provides basic methods for reading and writing cache memory and some additional methods for locking and flushing certain keys. It should be stressed that the cache has no implicit knowledge of the nature of the data it manages. This allows it to be decoupled from the animation system and in fact enables the cache to be flexibly used throughout a codebase if desired.

The table given below describes the methods available with the default implementation of the chunk

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

cache interface.

METHOD	DESCRIPTION
retrieveChunk (key, chunkSize)	Retrieves a read only pointer to the cached memory chunk. Returns NULL if the chunk is not available.
allocateChunk (key, chunkSize)	Allocates a chunk from the cache. Returns a pointer to the allocated chunk or HK_NULL if a slot cannot be allocated.
flushKey (key, chunkSize)	Forcibly removes the specified key from the cache. The slot occupied by the key becomes the next slot to be consumed when a new request is made of that bucket.
printCacheStats (hkOstream*)	Debug Statistics (see below for more details).

Table 4.6: Chunk Cache Interface Default Implementation Methods

The following table shows the methods provided by our default implementation.

METHOD	DESCRIPTION
retrieveChunk (key, chunkSize)	Retrieves a read only pointer to the cached memory chunk. Returns NULL if the chunk is not available.
allocateChunk (key, chunkSize)	Allocates a chunk from the cache. Returns a pointer to the allocated chunk. This will return NULL if the chunk size is too large for any of the available pools or if all the slots in an available pool are locked.
lockKey (key, chunkSize)	Locks the specified key (thus locking a slot in the cache pools bucket). This forces the key and cached chunk to always remain in the cache until it is unlocked.
unlockKey (key, chunkSize)	Unlocks a previously locked key (slot). This allows the key (slot) to be replaced if the cache needs a slot in its bucket.
flushKey (key, chunkSize)	Forcibly removes the specified key from the cache. The slot occupied by the key becomes the next slot to be consumed when a new request is made of that bucket.
flushCachePool (cachePool)	Forcibly removes all keys managed by the specified cache pool.
printCacheStats (hkOstream*)	Debug Statistics (see below for more details).
lockKeyForRead (key, chunkSize)/ **unlockKeyForRead** (key, chunkSize)	*(Empty implementation - See note below and hkaMultithreadedChunkCache)*
lockKeyForWrite (key, chunkSize)/ **unlockKeyForWrite** (key, chunkSize)	*(Empty implementation - See note below and hkaMultithreadedChunkCache)*
isKeyLockedForWrite (key, chunkSize)	*(Empty implementation - See note below and hkaMultithreadedChunkCache)*
enterCriticalSection ()/ **leaveCriticalSection** ()/	*(Empty implementation - See note below and hkaMultithreadedChunkCache)*

Table 4.7: Our Default Implementation Methods

Both the key and the chunkSize are unsigned 32 bit integers and describe a unique identifier and the size in bytes of the desired chunk respectively.

The decompression pipeline automatically generates key IDs for the animation data and retrieves and allocates chunks as necessary. If you do not require low level manipulation of the cache then all you need simply do is pass a pointer to a valid cache when you call `hkaAnimatedSkeleton::sampleAndCombineAnimations( ... )` to enable animation decompression caching.

If you do not pass a pointer to a cache, or if at any point during the decompression stage the cache should fail to retrieve / allocate data, the decompression pipeline will default to decompressing the data on the fly.

Note:
The default chunk cache (hkaDefaultChunkCache) is *not* thread-safe. See hkaMultithreadedChunkCache below.

The hkaMultithreadedChunkCache

Both wavelet and delta decompression implementations utilize the lock/unlock and enter/leave critical section interfaces of the hkaChunkCache, but these implementations are empty for the hkaDefaultChunkCache. In order to use the hkaDefaultChunkCache with multiple threads you would need to have a hkaDefaultChunkCache instance for each thread. In the hkaMultithreadedChunkCache however these are correctly implemented and will ensure that multiple threads can access the same cache instance in a thread-safe manner. Clearly this will be more memory efficient and likely will result in more cache hits (so expected improved performance), however blocking and other MT issues may make actual in-game performance unpredictable.

4.3.13.4 Profiling the Cache

To maximize the performance of the cache it is important to choose both an appropriate pool chunk size and an optimum number of buckets and slots for that pool. To determine these values by estimation alone would be quite difficult so the default cache provides statistics on slot usage and requests for unavailable chunk sizes. Using this information in conjunction with the detailed timing functionality available from the Animation system [via HK_TIMER_BEGIN( ) and HK_TIMER_END( )] it is possible to construct an optimum cache layout.

In order enable statistics generation you must define the following variable:

```
HK_CACHE_STATS
```

If this variable is not defined the statistics code will not be included with the cache. This is to ensure that cache performance is not compromised when statistics are not required.

A call to *printCacheStats(hkOstream* stream)* at runtime will give a detailed report on cache activity up to that point. The method takes an hkOstream* which allows the output to be directed to a console window or to a file for later inspection. A sample extract of the debug statistics is given below.

```
--- BEGIN CACHE STATISTICS ---

Requests for unavailable chunk sizes:
-------------------------------------
Chunk Size: 4256    # Requests: 13740


Cache Pools slot activity ( hits / misses ) information:
--------------------------------------------------------

....................................
  Cache Pool: 0    Chunk Size: 4096
....................................

Bucket 0
^^^^^^^^
        Slot 0 ( 975 / 272 )
        Slot 1 ( 984 / 272 )
        Slot 2 ( 1003 / 272 )
        Slot 3 ( 1054 / 272 )
        Slot 4 ( 1023 / 272 )
        Slot 5 ( 1122 / 273 )
        Slot 6 ( 1060 / 273 )
        Slot 7 ( 1108 / 273 )
        Slot 8 ( 1000 / 273 )
        Slot 9 ( 1011 / 273 )
        Slot 10 ( 973 / 273 )

Bucket 1
^^^^^^^^
        Slot 0 ( 995 / 175 )
        Slot 1 ( 1041 / 175 )
        Slot 2 ( 988 / 175 )
        Slot 3 ( 990 / 175 )
        Slot 4 ( 1114 / 175 )
        Slot 5 ( 1026 / 176 )
        Slot 6 ( 996 / 176 )
        Slot 7 ( 1113 / 176 )
        Slot 8 ( 1086 / 176 )
        Slot 9 ( 1045 / 176 )
        Slot 10 ( 1143 / 176 )

Bucket 2
^^^^^^^^
        Slot 0 ( 1056 / 267 )
        Slot 1 ( 1030 / 267 )
        Slot 2 ( 1150 / 267 )
        Slot 3 ( 1115 / 267 )
        Slot 4 ( 1065 / 267 )
        Slot 5 ( 1061 / 267 )
        Slot 6 ( 1152 / 267 )
        Slot 7 ( 993 / 267 )
        Slot 8 ( 1062 / 267 )
        Slot 9 ( 1066 / 268 )
        Slot 10 ( 1126 / 268 )

 --- END CACHE STATISTICS ---
```

The first statistic given is the number of requests for unavailable chunk sizes. In this example a request for a chunk size of 4256 bytes has been made 13740 times. Since the only cache pool available deals with chunks of 4096 bytes or less all requests for the 4256 bytes chunk force the decompression pipeline to decompress the animation data on the fly. In this case serious thought should be given to either extending the chunk size of the pool or introducing a second pool to handle the 4256 byte chunk size.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The second part of the statistics output displays the hit and miss value for every slot in each pool. These figures can be used to gauge cache performance. By varying the number of buckets and slots in the pool you can tailor the cache for an optimum hit / miss ratio.

A simple way of profiling your animation decompression requests is to initially create a single pool to deal with a 1 byte chunk size. Since no decompressed animation data will fit into such a chunk all cache requests will be tracked as unavailable chunk requests. This will produce a detailed statistics output of all possible chunk sizes required.

Note:

1. The hashing function used to hash the keys to a given bucket gives optimum key distribution if the number of buckets in a prime number.
2. The chunk size of each additional pool in the cache should be larger than that of the pool before it. That is to say, the first pool should deal with the smallest chunk size, the second with a larger chunk size, the third with an even larger chunk size and so forth.
3. The cache will assign a given chunk size to the first pool capable of dealing with that size. Therefore if you wish to dedicate a certain pool to a specific chunk size you should also have a pool 1 byte smaller to force all other lesser chunk sizes to use it. For example, to keep all 700 byte allocations in a specific pool you should (assuming you will have other sub 700 bytes chunk requests) create two pools; one with a chunk size of 700 bytes and another with chunk size 699 bytes. This will ensure all 700 bytes requests use the 700 byte pool and all <700 byte requests use the other pool.

4.3.14 Deformation support

The current deformers fall into two categories: skinning deformers (`hkaSkinningDeformer`)and morphing deformers (`hkaMorphingDeformer`). The deformers are then split into two sub groups: one is where you are using a SIMD (SSE on PC etc) enabled hkMath library and one where you are not (a plain FPU version). The SIMD versions require that *at least* your input data members (position, normals) are aligned to 16bytes (16bytes == 4 floats == 1 simd vector) and works faster if your output data members are aligned as well. It autodetects the data alignment and uses an appropriate algorithm automatically. If you choose to use the SIMD deformers when you don't have SIMD enabled you may well discover that it is slower than just using the plain old FPU version due to needless copies etc.

Important:

It is important to note that the deformers are there to be used by Havok's demos but are not production quality. It is assumed that deforming will be done most commonly by your graphics engine, usually in hardware on GPUs or VUs. That hardware deformation is usually performed at the same time as per vertex lighting operations, so Havok cannot provide optimized deformers for all such game specific usage.

Havok animation provides both tool chain and runtime support for animated meshes. This breaks down into support for skinned meshes and support for attachments. We explicitly break these into distinct sections as they are often treated separately by different sections of the rendering pipeline. Both types of objects refer to our base mesh stored in the hkxScene description so we begin with a discussion of that.

4.3.14.1 Meshes

Meshes exported through our export utilities are made from one or more mesh sections. Each mesh section is composed from a material description, a vertex buffer and an index buffer. The section represents some segmentation of the mesh and although the description itself allows complete flexibility, our export utilities typically divide based on material.

4.3.14.2 Vertex Buffers

Mesh data is typically held in the vertices on a per vertex level. Our vertex description is completely flexible specifying only a raw data buffer and number of elements and a data description (a `hkClass`) of the layout of a buffer element.

We provide data descriptions for the common vertex formats we export. The names of these classes are generated from the component they contain. For example the class `hkxVertexP4N4C1T2` contains a position as 4 floating point values, a normal as 4 floats, a colour value, and 2 texture coordinates. To create your own format simply write the header file for the class description, include the `HK_DECLARE_REFLECTION` Macro once the serialization scripts are run on the header it will produce a class description which can be used to construct a vertex buffer.

Note:
All positions, normals, binormals and tangents are exported in world space.

4.3.14.3 Skinned Meshes

In our scene data all meshes are represented in local space (i.e. the positions, normals and tangents etc. are all in local space). When skinning, meshes are often represented or stored in world space, and the inverse of the binding pose is used to construct a matrix palette for skinning. Storing the skinned meshes in this format means we only need to store each vertex in the skin once in local space, while storing the inverse bind pose in world space saves us from having to store vertex data multiple times in the local space of each bone the vertex is bound to. Since meshes are exported in local space the transformation from local to world space is implicitly included in our inverse bind pose (given in hkaMeshBinding).

Skinned meshes are exported in 2 phases. In the first phase the export utility places the character in its bind pose and exports the mesh in local space. The exported mesh will contain weight and bone indices for each of the vertices. Today we export a maximum of 4 weights per bone, and these weights are quantized and stored as bytes to reduce mesh size. These skinned meshes are usually accessed through the `hkxSkinBinding` array in the `hkxScene` class. Each skin binding contains:

- A pointer to the underlying mesh in bind pose space
- A list of the nodes it is bound to
- A copy of the original bind pose of the bones
- The skin world transform at the time of binding (since the skin mesh vertices are stored relative to this)

Usually this information is adequate to deform a skinned mesh using matrix palette skinning. However, often the user will redefine the rig in the filter pipeline and hence this description requires extra data to map to a new skeleton.

This extra data is stored in the `hkaMeshBinding` and this is the class we usually use at runtime. This contains a mapping from the bones in the original rig as seen and stored in the modeller and the rig as defined by the user in the filter pipeline. Including the CreateSkin filter in the filter stack creates one of these mesh bindings. A hkaMeshBinding's m_mappings array can be null, indicating that the mapping is just a trivial one to one map (skeleton bone indices are used as is by the mesh indices). If the m_mappings array is of size one, then the mappings held in that structure apply to all mesh sections and

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

are the mapping between the mesh indices to the skeleton indices (as the mesh uses less than the full set of skeleton bones for instance). If the m_mappings array is more than size one, then there is a mapping for each primitive (each index buffer, in each hkxMeshSection) in order. This happens when you break up a mesh based on limited bone influences per mesh primitive (e.g: 20 bone limit per DrawPrimitive call in DirectX when using a vertex shader that takes a palette of 20 bones). The CreateSkin filter can do this limitation and segmentation for you using the hkxMeshSectionUtil functions.

The hkaMeshBinding also combines the skin world transform (`hkxSkinBinding::m_initSkinTransform` [*worldFromSkin*]) and the binding pose transform array (`hkxSkinBinding::m_bindPose` [*worldFromBone*]) into a single transform array, `hkaMeshBinding::m_boneFromSkinMeshTransforms` [*boneFromSkin*].

4.3.14.4 Attachments

During asset filtering the CreateSkin filter adds attachments to the animation container. These attachments are found by examining the nodes that form the skeleton rig. If any of these nodes refer to scene objects a `hkaBoneAttachment` is created. Each attachment has:

- The bone index of the bone it is bound to
- The world transform of the attachment
- A description of what has been attached.

The description of what has been attached is given as a `hkVariant`. This allows anything to be attached but in practice we only attach lights, cameras and meshes.

Note:
Though our meshes are stored in local space it is common to use the world space position of a given bone to drive the model space matrix from the attached mesh. Using the `m_boneFromAttachment` transform is the most common way to do this.

4.3.15 Inverse Kinematics

Havok Animation provides different inverse kinematics algorithms for run-time. By using inverse kinematics, you can modify a pose according to external information based on the environment. Examples of usage are:

- Make the character look at particular objects in the scene

- Place the feet of the character firmly on an uneven ground surface

- Reach for a doorknob

Some of the algorithms provided by Havok Animation are specialized for some of these situations (for example, foot placement or look-at controllers), while more generic solvers (like the CCD solver) can be used for arbitrary situations.

The IK solvers provided by Havok Animation are:

- Cyclical Coordinate Descent (CCD) Solver : iterative solver that operates with chains of bones of any length

- Two Joints Solver : analytical solver that operates on two joints : a ball-socket joint and a limited hinge joint (useful for limbs like arms and legs)

- Three Joints Solver: iterative solver that operates on three joints : a limited ball-socket joint (2 rotation axis) and a two limited hinge joints (useful for objects like robotic arms or spider legs)

- Look At Solver : analytical solver that operates on a single, limited, ball-socket joint

- Foot Placement Solver : A high-level IK solver that tracks changes in ground height and incline and modifies the extension of a leg and the orientation of a foot accordingly.

4.3.15.1 The CCD IK Solver - hkaCcdIkSolver

The CCD solver method employs an iterative IK process that improves the accuracy of the bone chain configuration with each pass. By repeating the CCD procedure on the bone chain for a given number of iterations the chain will either reach its target position or be as close as possible. There is a direct correlation between the number of iterations performed and the accuracy of the solver.

Note:

The CCD solver has been implemented under the assumption that it will be used for simple animation fix ups and as such is not suitable for complex, highly constrained limb movement. The CCD solver satisfies a positional constraint only and does not attempt to achieve a given end effector orientation.
Although this produces agreeable results most of the time, it is worth noting that unexpected bone chain configurations may occasionally occur. This is due to the fact that the solver gives complete freedom to the bone rotations and thus they may move in an unintuitive manner. An arm, for example, may rotate in a physically impossible fashion. The simple CCD solver does not currently support joint constraints to prevent these undesired configurations and some care may need to be taken to ensure they are not produced.

The CCD solver takes two parameters for construction, namely the number of *iterations* to perform and the *gain* to use during solving. The number of iterations used directly relates to the quality of the final configuration (although there will be a point whereby further iterations do not visibly improve the result). The gain influences how quickly the solver converges on a solution and effectively dampens the movement of the bones to try reduce oscillations about the desired target. It may be worthwhile to modify the default values for these parameters to try find a sweet spot between CPU time and accuracy required for your needs.

At run time, the CCD solver modifies a pose (wrapped with an hkaPose object) based on a series of constraints:

```
virtual hkBool solve ( const hkArray<IkConstraint>& constraints, hkaPose& poseInOut );
```

The constraint array contains a set of `IkConstraint` structures. This structure is shown below:

```
struct IkConstraint
{
    /// The start bone in this chain
    hkInt16 m_startBone;

    /// The end bone in this chain
    hkInt16 m_endBone;

    /// The target position for the end bone, in model space
    hkVector4 m_targetMS;
};
```

A constraint is simply a set or chain of bones to be manipulated (defined by the `m_startBone` and `m_endBone`) with a target or desired position for the `m_endBone` (given by the `m_targetMS` member). The solver will take each constraint in the array in turn and determine new transforms for each bone in the chain.

4.3.15.2 The Two Joints IK Solver - hkaTwoJointsIkSolver

The Two Joints solver (**hkaTwoJointsIkSolver**) is an optimized solver for IK configurations where the only joints involved (the only joints modified for the IK) are:

1. A ball-and-socket (spherical) joint
2. A hinge joint, with a range of allowed angles around the hinge axis of [a,b], where a>=0 and b<=180 degrees.

We find this configuration in many situations, in particular limbs (arms and legs). For example, in an arm, the spherical joint would be the shoulder, and the hinge joint would be the elbow.

As well as the pose to be modified, the solver takes as its input:

- The indices of the bones for the first (spherical) and second (hinge) joint
- The index of the end effector bone
- The hinge axis of the second joint, specified so positive rotations (using the right hand rule) extend the joint.
- Limit angles (specified with cosine) of the hinge joint (defaults are 0 deg [1.0] and 180 deg [-1.0])
- Gain values for modifications applied to both joints. The modifications calculated by the IK solver will be scaled by these values (a value of 0.0 won't modify the pose, a value of 1.0 will apply the IK fully). You can use these gains in order to "ease in" and "ease out" the IK.
- The target position, in model space, of the end effector bone

The figure below shows the Two Joints IK solver in action:

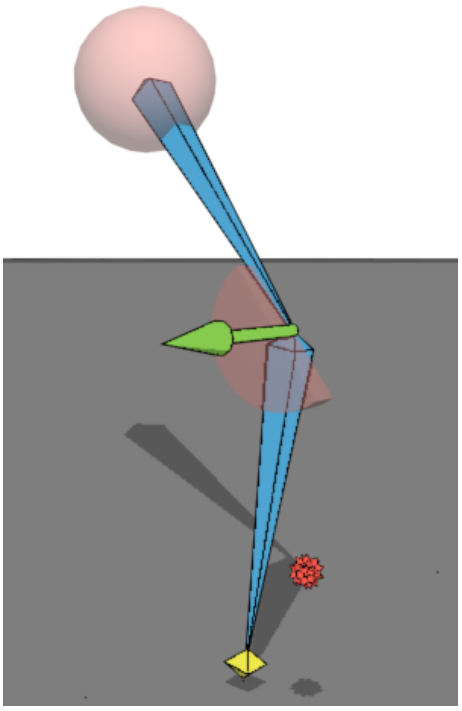
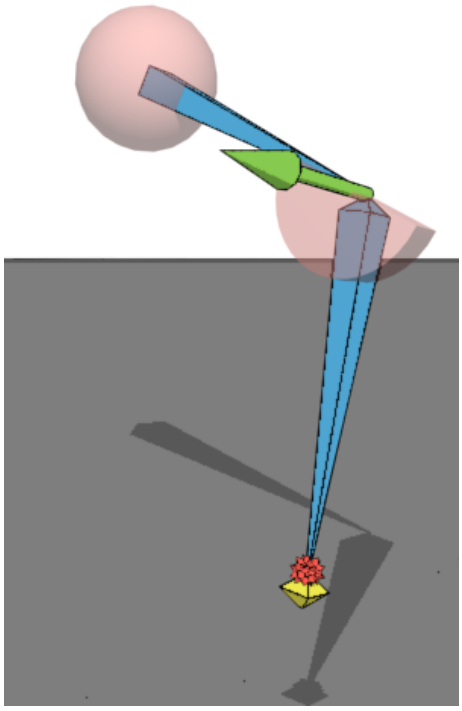
	
The input pose	The output pose

Table 4.8: The Two Joint IK Solver in Action

The solver will always apply the minimal modification in order to reach the desired target. That is, from all possible solutions to the IK problem, the closest solution to the input is chosen. Thus, the original pose passed to the solver influences the output pose, as shown below.

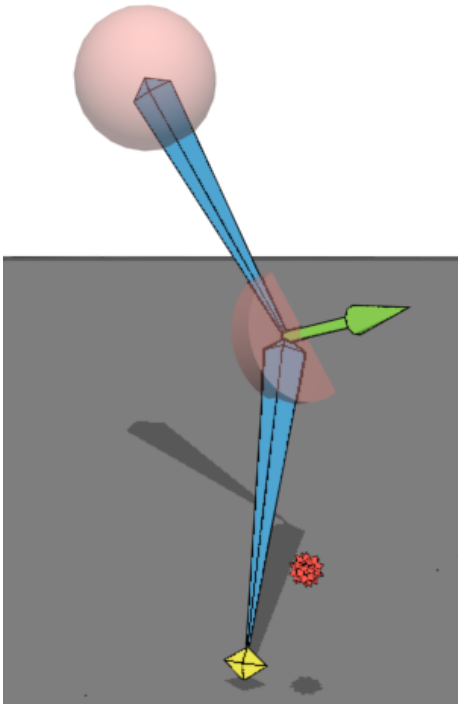
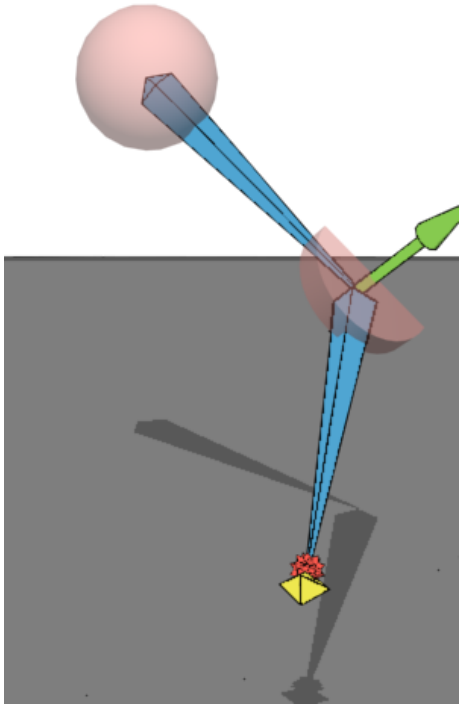
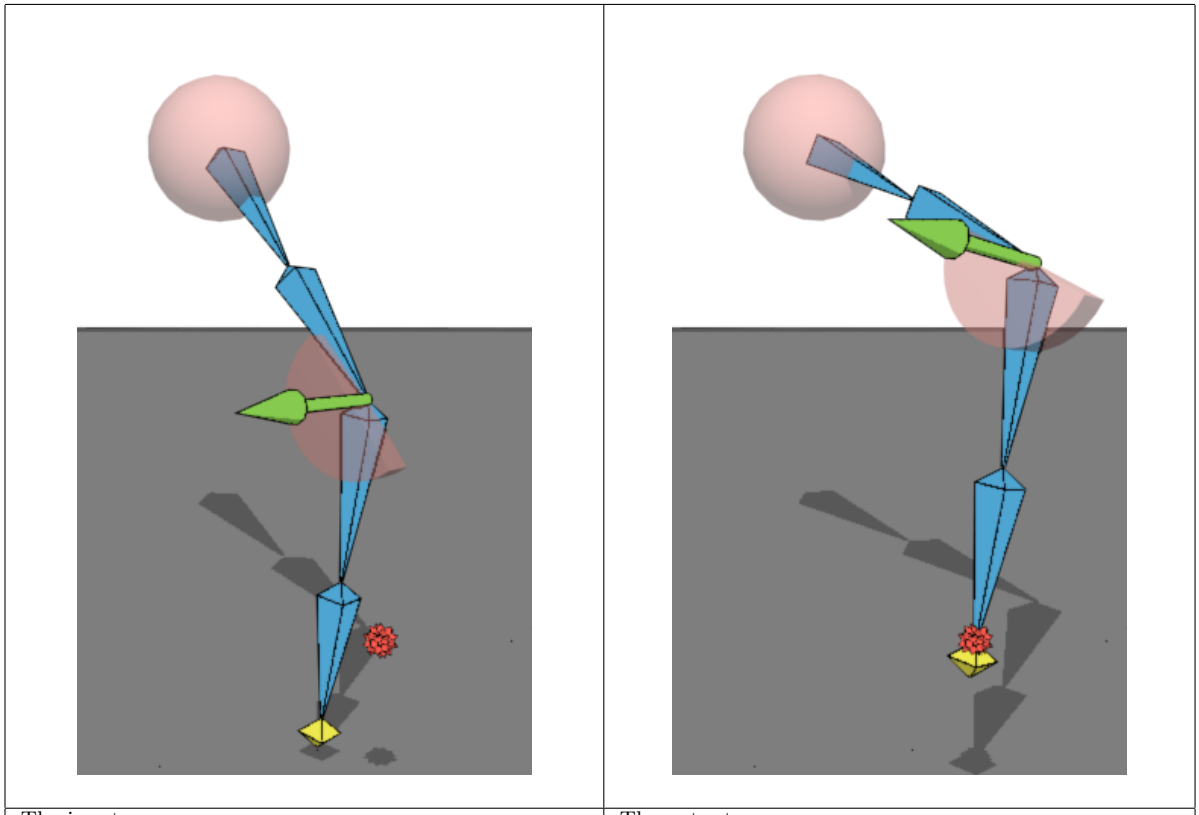
	
The input pose	The output pose

Table 4.9: Different input poses will yield different (but still valid) results

Notice that the first (spherical) and second (hinge) joint do not necessary need to be immediate parent and child in the bone hierarchy; that is, as long as the second joint is deeper in the hierarchy than the first joint (and the target joint is deeper in the hierarchy than the second joint), the IK solver will operate correctly. Any joints in between the first, second and target joint will be kept rigid (their local transform won't change), as illustrated below:



The input pose	The output pose

Table 4.10: The Two Joint IK solver can operate on chains of any length, but it will only modify two joints

4.3.15.3 The Three Joints IK Solver - hkaThreeJointsIkSolver

The Three Joints IK Solver (**hkaThreeJointsIkSolver**) is a fast iterative solver for IK configurations where the joints involved (the only joints modified for the IK) are:

1. A first hinge joint with two rotation axes. The first axis defines rotation of the whole bones plane and the second axis is perpendicular to this plane.
2. A second hinge joint with a rotation axis perpendicular to the plane.
3. A third hinge joint with a rotation axis perpendicular to the plane.

We can find this configuration in many situations, in particular unnatural limbs (insects or creatures legs, robotic manipulators, space robotic arms). For example, in a spider leg, the first joint would be the joint between the body and leg, the two hinge joints would be the elbows and the end effector (target point) represents contact point with the ground. A major restriction of the solver is the fact, that all joints and end effector must lay in a plane.

As well as the pose to be modified, the solver takes as its input:

- The indices of the bones for the first (2 axes hinge), second (hinge) and third (hinge) joints

- The index of the end effector bone
- The hinge axis of the first joint, around which the whole plane is allowed to rotate. This axis must lay in the bones plane.
- The target position, in model space, of the end effector bone

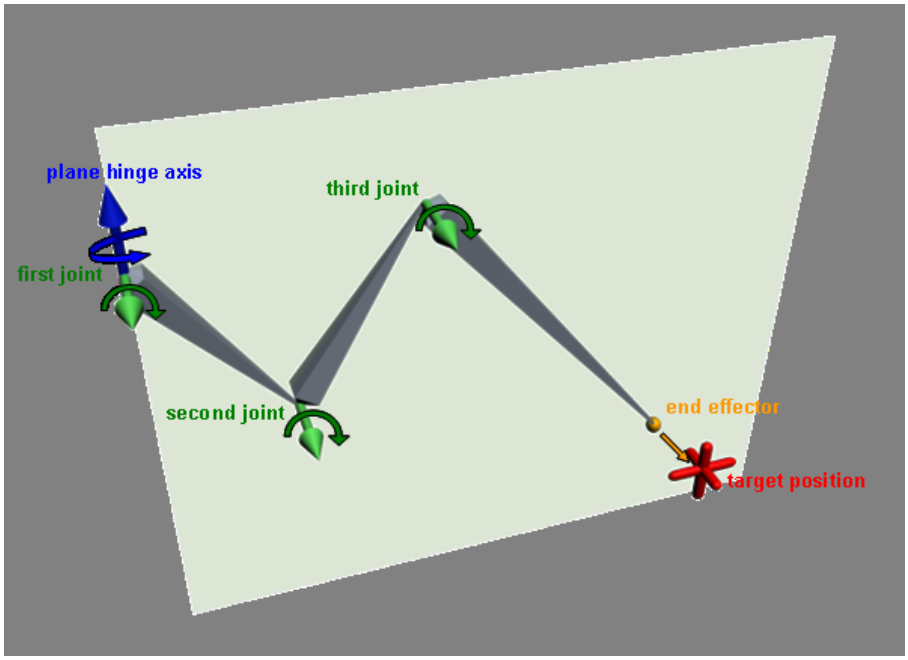


Figure 4.40: The solver configuration in 3D

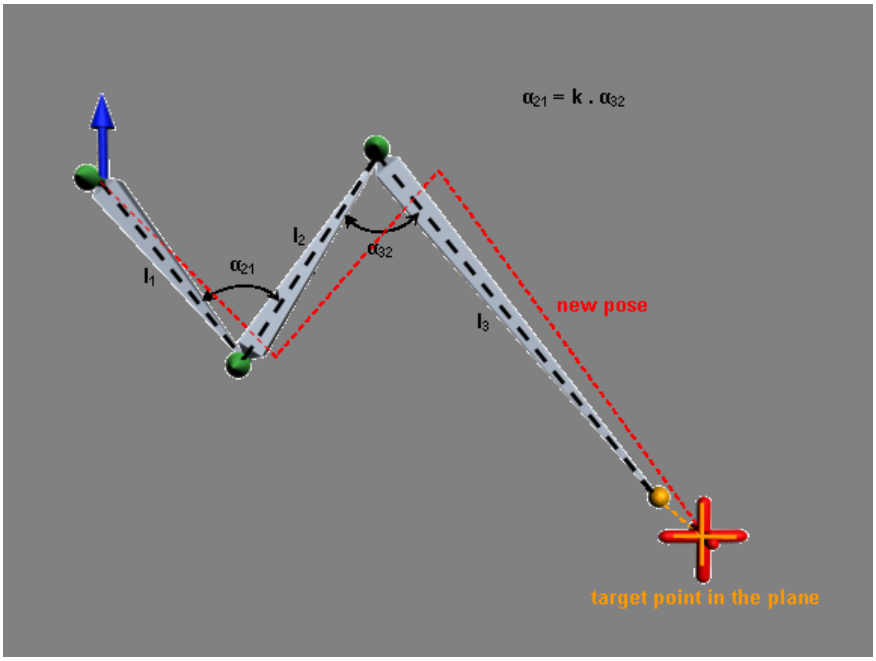


Figure 4.41: The 2D solution in the plane

The solver determines a new pose with the end effector at the target position in the three steps:

1. The solver rotates the whole bones plane around the first joint hinge axis (blue) to the target point position.
2. The solver calculates three new rotations in each joint around hinge axes (green) perpendicular to the bones plane to reach a desired position in the plane (red configuration).
3. The final output pose is updated.

In general, the mechanism in this configuration is kinematically under-estimated and the additional condition has to be defined. Our solution utilizes the restriction that the angles between the first and second bones(21) and second and third bones (32) are linearly dependent (21 = k 32). The constant k is calculated from the initial pose in the constructor. The core computation is performed by iteratively solving in the plane.

The target position has to occupy the operational space of the configuration. The operational space is defined as a sphere with the center in the first joint and with inner radius (collapsed configuration) and with outer radius (maximal reachable configuration). In a situation where the target point is out of the operational space (too close or too far from the first joint), its projection to the sphere is automatically calculated to ensure smoothness of the iterative solution.

The solver will always apply the minimal modification in order to reach the desired target. That is, from all solutions satisfying additional condition to this IK problem, the closest solution to the input is chosen.

Considering the nature of this kinematic problem and its solution, the restrictions of the solver are the following:

- All joints and the end effector must lie in a plane
- The angles between the first and second bones and the second and third bones are linearly dependent.
- The target position, in the model space, has to be changed continuously. It is recommended to apply only small steps between the previous and current target position.
- The target position, in model space, has to occupy the solver operational space.

4.3.15.4 The Look At IK Solver - hkaLookAtIkSolver

The Look At solver (**hkaLookAtIkSolver**) is a very simple IK solver that modifies the rotation of a single joint/bone so the bone points towards a particular point in model space. The output is limited by a cone of movement. Alongside the transform of the bone, the following information is passed to the Look At IK Solver

- A "forward" axis, defined in the local space of the bone. This axis is the axis that will be aligned towards the target point.
- A "eyes" position vector in the local space of the bone (head). Usually it defines a point between eyes. The default value is (0,0,0).
- A limit axis, defined in model space. This axis is the center axis of the cone that limits the movement of the bone.
- A limit angle, which alongside the "cone" axis defines the cone of allowed movement for the bone

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- A gain value. The rotation calculated by the solver will be scaled down by this value (so a value of 0.0 won't modify the bone, while a value of 1.0 will apply the IK fully). You can use this value in order to ease in and ease out the Look At IK.
- A target position, defined in model space. The IK will try to align the target vector to the direction defined by the vector between the "eyes" and target positions.

The following illustrations show the Look At IK Solver in operation, both when the target direction is inside the limiting cone movement and when it's outside.

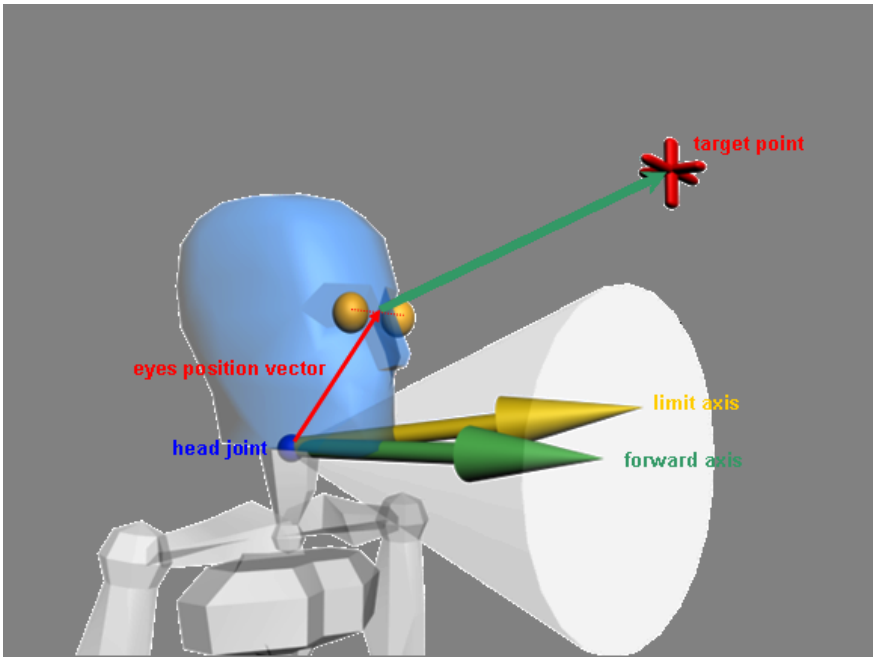


Figure 4.42: The input to the Look At IK Solver (target is inside the limiting cone)

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

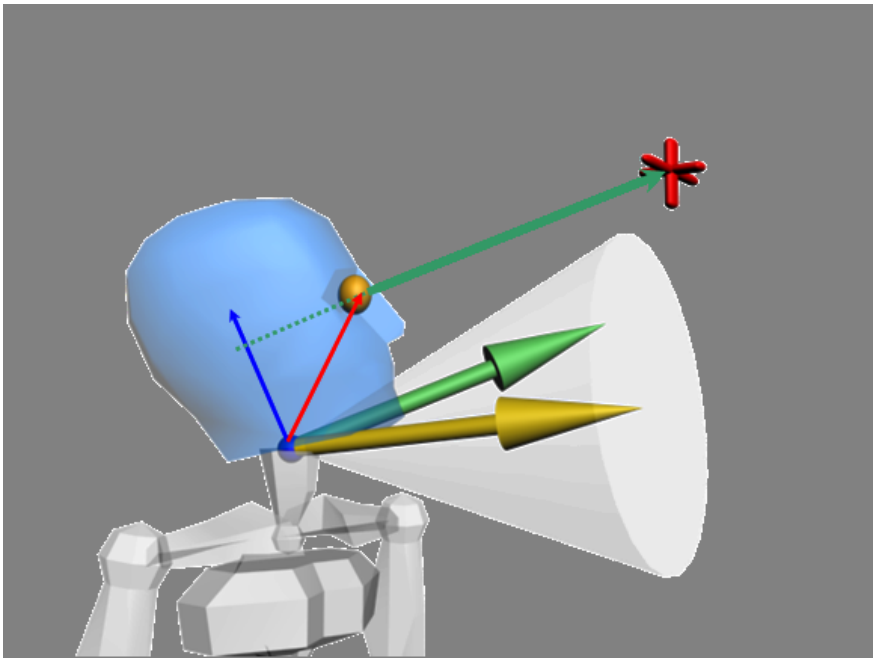


Figure 4.43: The output of the solver (target is inside the limiting cone)

Notice how in this second case the "forward" axis (in green) is kept inside the limiting cone (in yellow):

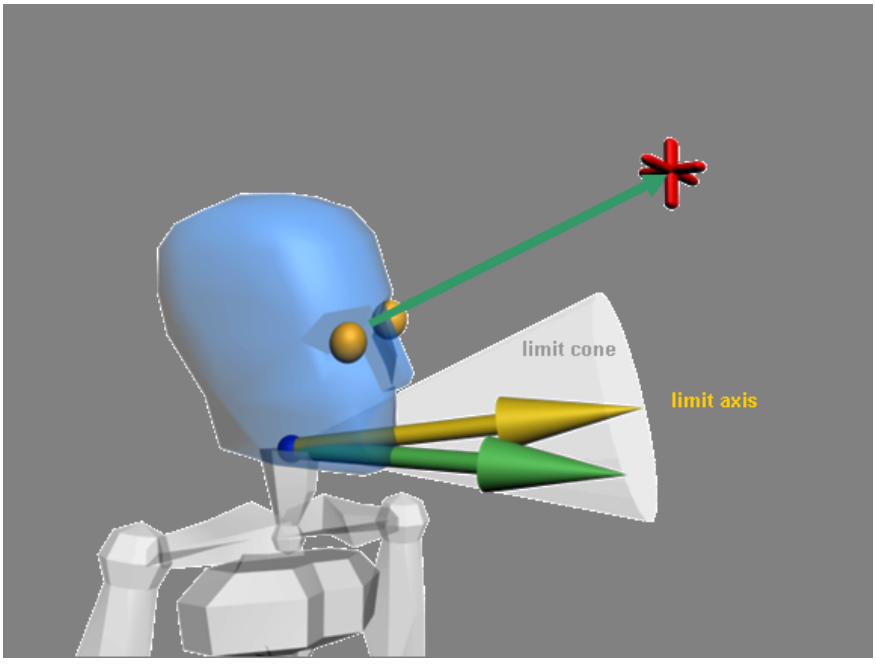


Figure 4.44: The input to the Look At IK Solver (target is outside the limiting cone)

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

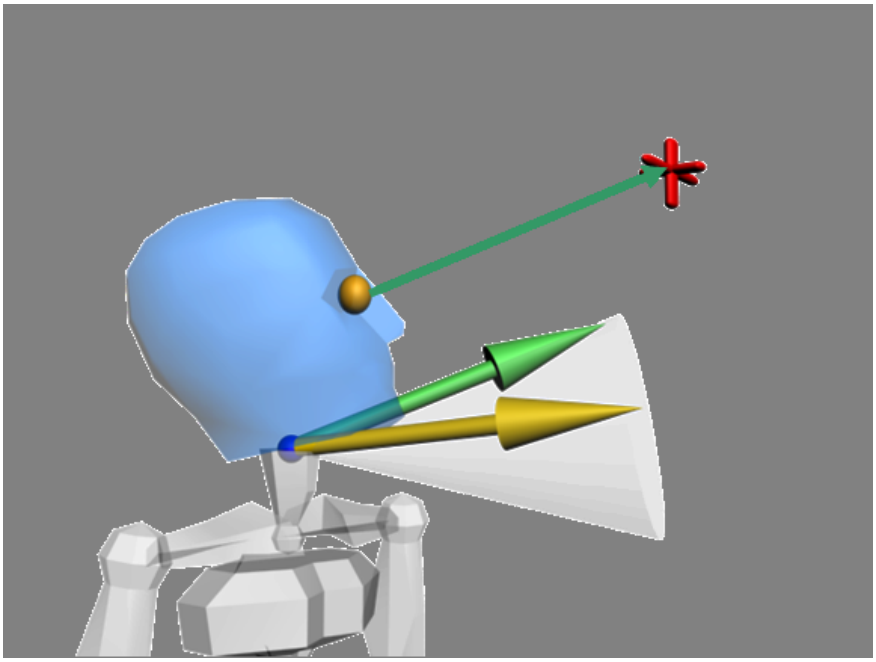


Figure 4.45: The output of the solver (target is outside the limiting cone)

Note:

Notice that the Look At IK Solver differs from other solvers since it operates on a single bone transform rather than a full pose. Thus, if your character has any children bones attached to the bone used for look at, you may want to modify them accordingly. For example, if you are modifying the head with the Look At solver, you probably want any eyes, antennae or ponytails to be modified accordingly. You can use an **hkaPose** object to do so, as shown below:

```
// "pose" is an hkaPose object that contains the current pose of our character
hkaLookAtIkSolver::Setup setup;
setup.m_fwdLS.set(0,1,0);
setup.m_eyePositionLS(0.12f,0.1f,0.0f);
setup.m_limitAxisMS.setRotatedDir(pose.getBoneModelSpace(m_neckIndex).getRotation(), hkVector4(0,1,0));
setup.m_limitAngle = HK_REAL_PI / 5.0f;
// By using the PROPAGATE flag, all children of the head bone will be modified alongside the head
// (their local transforms will remain constant)
hkaLookAtIkSolver::solve (setup, m_targetPosition, 1.0f, pose.accessBoneModelSpace(m_headIndex, hkaPose::
  PROPAGATE));
```

4.3.15.5 The Foot Placement IK Solver : hkaFootPlacementIkSolver

The Foot Placement solver is a specialized solver that can be used to modify the extension of legs and orientation of feet according to changes in ground height and slope. This solver is a little more sophisticated than the rest of the IK solvers, particularly because it keeps a state between calls and because it actually interacts with the environment (through a raycast interface) in order to operate.

The solver operates on a single leg, and therefore you will need to instantiate one solver for each leg. The solver operates with the following data:

- Setup Data : Passed on construction, it defines the structure of the character : skeleton, joints involved, limits and axis.
- Input Data : Passed to the solver on each step, it remains unmodified. It contains information about gains, the location of the character, the original transform (position and orientation) of the foot (the one the solver will base its results on) and a raycast interface for the solver to use
- Pose : The solver modifies an instance of hkaPose
- Output Data : Apart from modifying the pose, the solver returns some other information, like the vertical error detected, that can be used by the application (for example, to update the pelvis of the character).

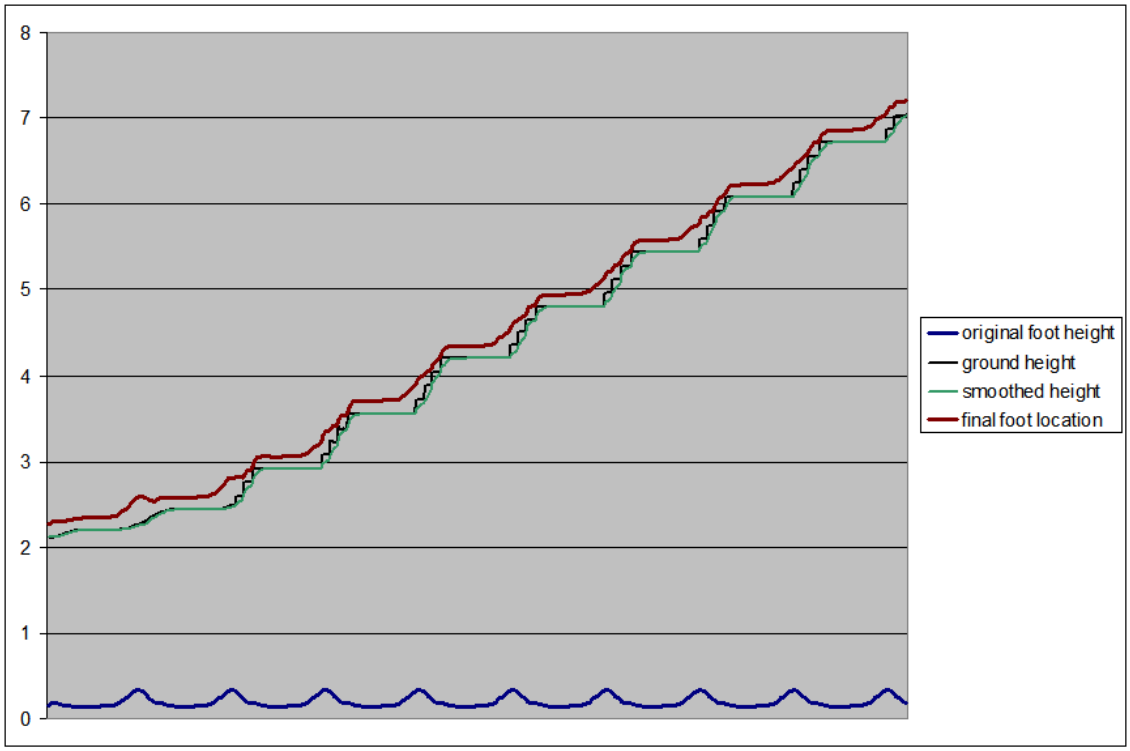
Overview

In simplified terms, the solver works in the following manner:

1. Using raycast(s), it cast the current position of the foot towards the ground, from where it gets a height and an incline
2. From the original transform of the foot, it calculates the height above ground the foot is supposed to be on (if the surface was flat).
3. Based on that and other gain values, it smooths out sudden changes in the height detected by the raycast.
4. Based on this smoothed height of the ground, and the original foot height above the ground, it places the foot at the right location (using a Two-Joints IK solver)
5. Similarly, based on the incline of the ground, and the original orientation of the foot, it reorients the foot accordingly.

The following graph, extracted from an actual application using the Foot Placement solver, should illustrate the above.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



The graph was taken from a character walking a set of stairs. In blue, at the bottom, you can see the original height (in model space) of the foot as driven by the animation system. In black, you can see the height (in world space) detected by raycasting the foot against the landscape. In green, you can see this height smoothed out; notice how it matches the ground height at the times where the foot is planted. Finally, in brown, you can see the resulting height of the foot, which is based on both the original foot height above the ground, and the smoothed ground height.

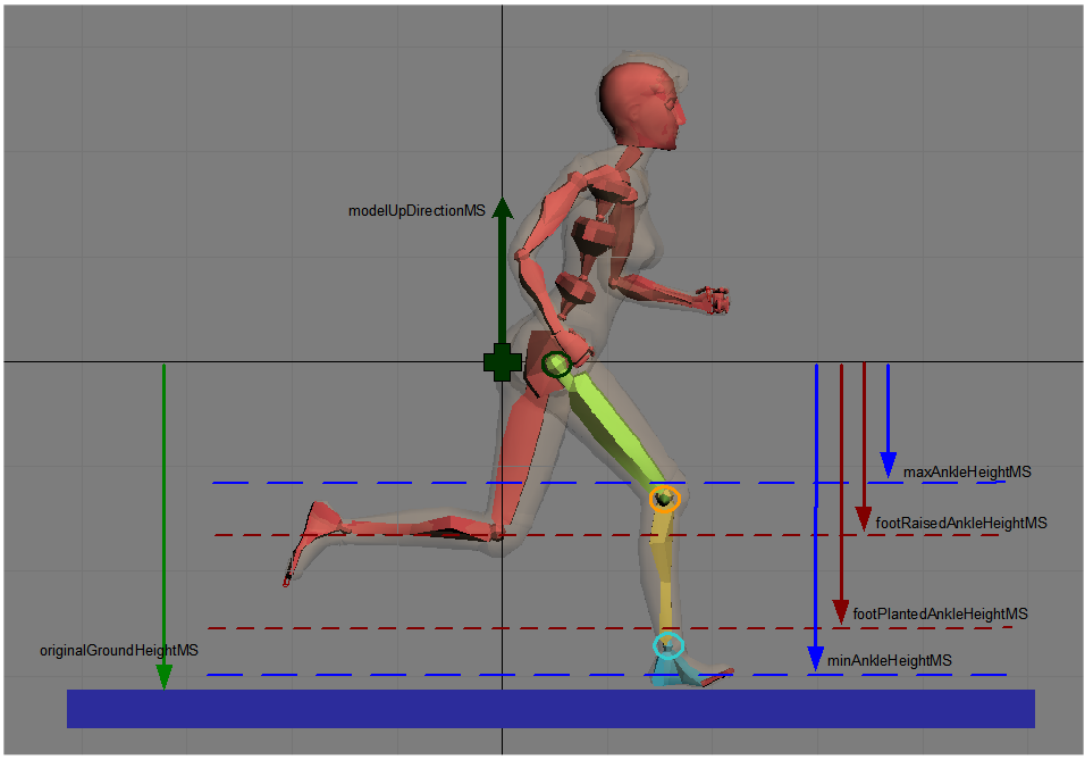
Setup Data

The Foot Placement solver takes some setup data on construction. This setup data contains information that usually doesn't change at run time, like limits and axis and bone configurations. This information is:

- The skeleton of the character to which we are going to apply this solver
- The indices of the three joints involved : hip, knee and ankle (in the diagram below, circled in green, orange and blue)
- The rotation axis of the hinge in the knee joint, specified in local space of the knee (in the diagram below, the green axis pointing outwards from the knee.
- The location of the end of the foot, specified in the local space of the ankle. This is used for raycast, and is described in more detail later on (it's the light green circle in the second diagram below). If this location is (0,0,0), the length of the foot will be ignored (a single ray cast will be used).

(Notice that the parameters above, with the exception of the "end of foot" parameter, are the same used by the Two-Joints IK solver)

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

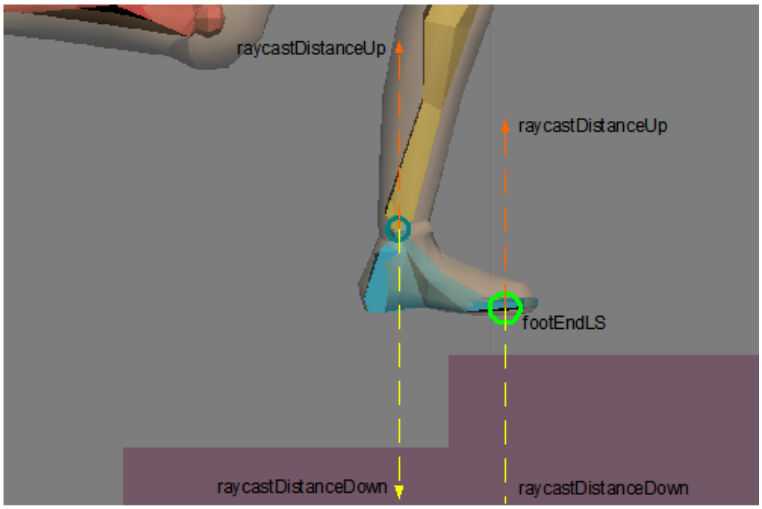


- The UP direction of the world (the application), specified in world space. This is the direction of UP in the game environment (pointing from the ground to the sky). It is used for the raycasts.
- The UP direction of the model, specified in model space (and shown in green in the diagram above). This is the direction of UP in the space where the model (character) was defined. It may not match the UP direction of the application : for example, a character may be setup with Z as the up axis (maybe setup in 3ds max), while it may walk around an environment where Y is up (maybe created in Maya).
- The original ground height in model space. This is the height of the (imaginary) ground used to author the model, i.e, the vertical distance between the origin (0,0,0) in model space (a.k.a. the reference frame) and this imaginary ground.

  Note:

  The location of this origin (and hence, the value of this parameter) will be different depending on the way your character is setup (root in the ground, root at the pelvis) and how motion is extracted. In the example above, this distance (and the ankle heights described later on) would be a negative value since the ground is located below the origin in our model space (shown as a green cross). If your character and animation place the model space origin (the reference frame) at the ground, then this distance is 0 (and the ankle heights will be positive).

- The ankle height (specified from the model space origin) below which we consider the foot is planted. This is used in order to calculate gains used for smoothing the foot placement (as described in the overview)
- Similarly, the ankle height above which we consider the foot is raised.
- A maximum and minimum height that the ankle cannot go above / below. The foot placement will ensure that the height of the ankle never exceeds this range.
- Maximum and minimum angles allowed for the knee (specified with their cosine). These are the same parameters used in the Two-Joints IK solver, and can be used to avoid knee popping artifacts.



- Two distances, raycastDistanceUp and raycastDistanceDown, that define the start and end of the rays that will be cast against the environment in order to place the foot. These distances are both positive and will be applied to both the start of the foot (the location of the ankle) and the end of the foot (as specified by footEndLS)

Note:

The reason two raycasts can be used instead of one is in order to take the length of the foot into account when doing foot placement over discontinuous terrain (like stairs), as in the example above. Casting a single ray would leave the foot going through the steps. But by using a second ray near the end of the foot the foot placement solver will detect this and the foot will then be placed on the step above.
If the end of the foot is not specified (i.e., if footEndLS is left as 0,0,0), then a single raycast will be used.

Input Data

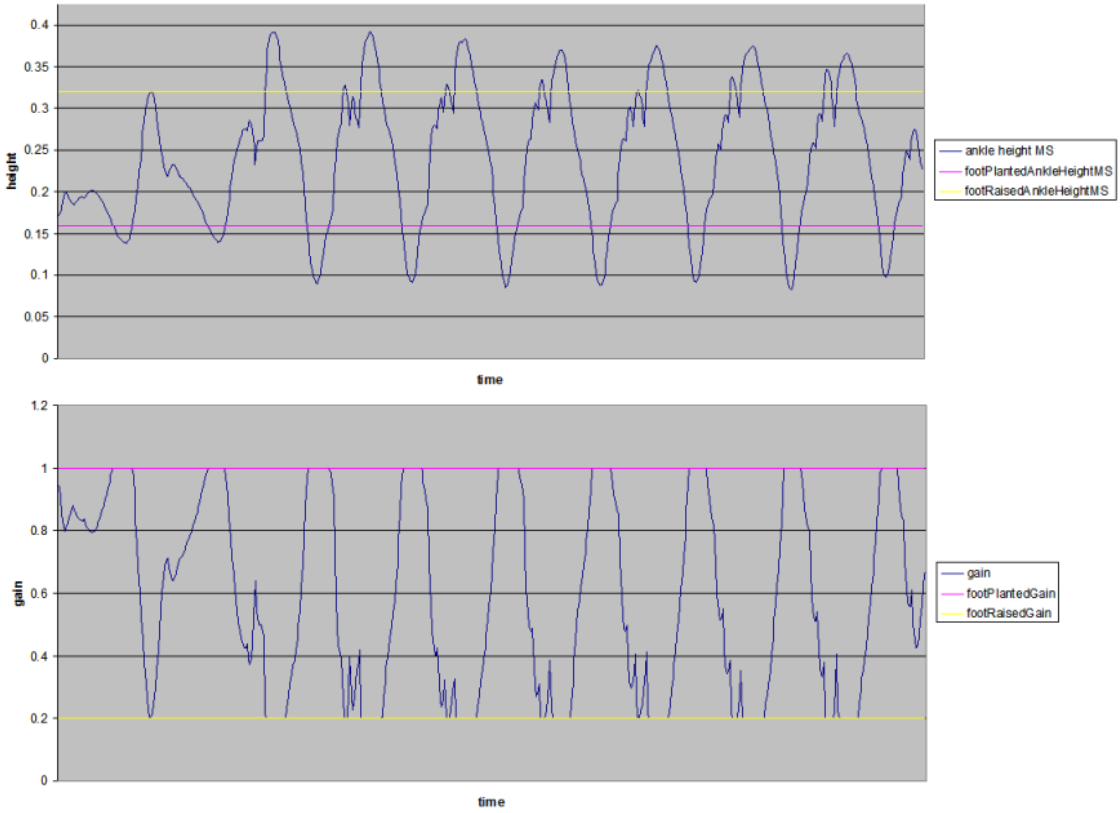
Alongside the pose to modify (passed as an hkaPose object, `poseInOut`), the following information is passed every time you want the Foot Placement solver to operate:

- The original transform of the ankle (in model space). The foot placement will base its results on this transform, rather than the transform of the ankle in the pose (`poseInOut`) passed to the solver. This is because, in some cases, you may want to do the fixing up based on an "original pose" different than the actual pose passed as a parameter. In the most common case, though, this transform will be the same as the model space transform of the ankle in poseInOut.
- The `worldFromModel` (model to world) transform. This transform describes the position and orientation of the character (the model) in the world and it's usually the same transform used throughout the application (for skinning, game logic, rag doll, etc).
- A flag (`footPlacementOn`) indicating whether we want the foot placement to switch itself on or off. The foot placement solver will ease itself in or ease itself out smoothly, and will keep itself on or off, based on this parameter.
- A raycast interface, defined by the abstract class `hkaRaycastInterface`. The implementation should be able to cast rays of limited length into the environment.
- An on/off gain, used by the solver to transition smoothly when switched on or off (based on changed in the footPlacementOn flag)

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- The gain used to smooth positive changes in ground height (i.e. when the ground is ascending)
- The gain used to smooth negative changes in ground height (i.e. when the ground is descending). The reason for having two different gains is that you may want to have higher gains when going up since penetration artifacts are more likely to happen when going up than when going down.

The final gain used to smooth changes in ground height is a combination (a multiplication) of the above gains (either the ascending or descending, depending on the landscape) and another gain value calculated based on how "planted" the foot is. This value is calculated based on the desired ankle height (as defined by the original transform of the ankle), the `footPlanted` and `footRaised` ankle heights (passed at setup time) and two other gain values, `footPlantedGain` and `footRaisedGain`, passed as part of the input structure.



As the graphics above show, this gain will be calculated as follows:

- For heights above the `footRaisedAnkleHeightMS` setup value, the `footRaisedGain` will be used
- For heights below the `footPlantedAnkleHeightMS` setup value, the `footPlantedGain` will be used
- For values in between, an interpolation between `footRaisedGain` and `footPlantedGain` will be used

Note:

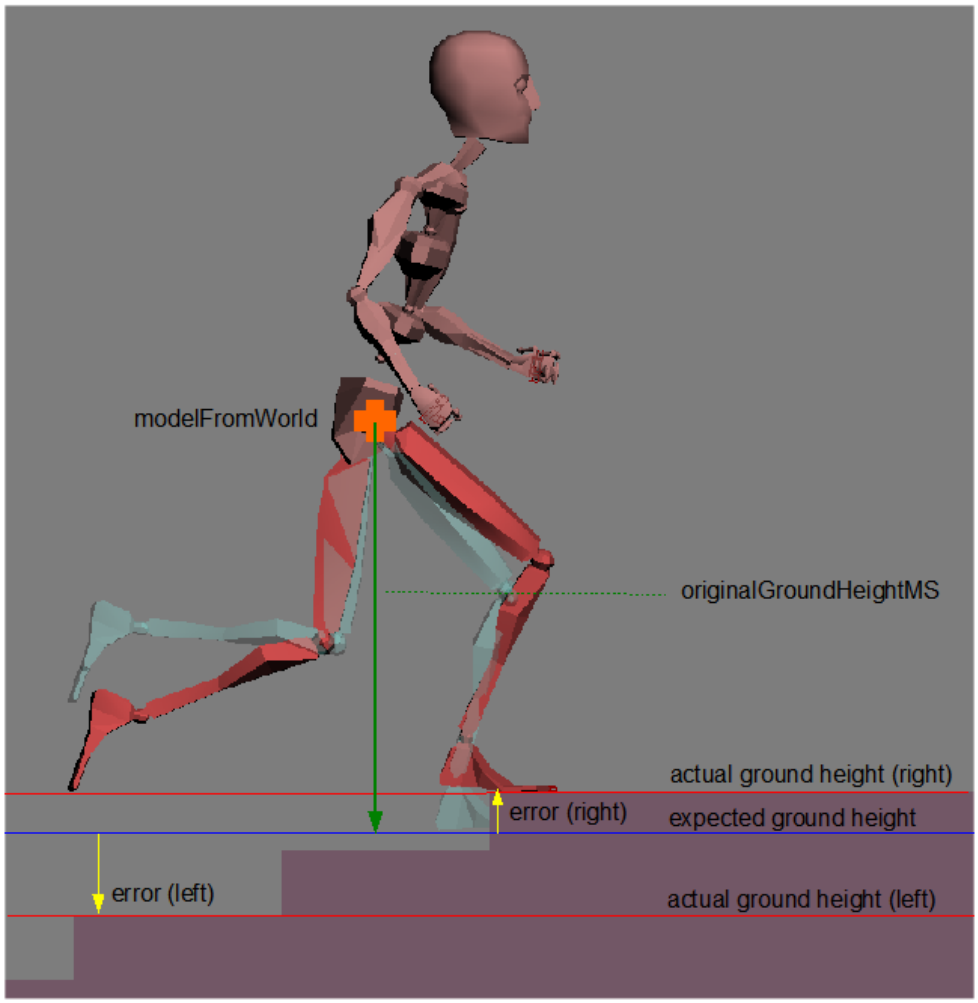
Again, remember the final gain used to smooth landscape changes is the product of this gain and the ascending/descending gain.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Output Data

The Foot Placement solver also returns some extra information that can be used by the application:

- A *vertical error*. This is the difference between the actual height of the ground as detected by the raycast and the height of the ground as expected by the animation (calculated by using the worldFromModel transform and the setup value `originalGroundHeightMS`). This is illustrated below



The bones in blue represent the pose before the foot placement is applied (twice, once for each leg), and the bones in red represent the pose after the foot placement is applied. You can see the vertical error returned for each leg in yellow. Notice that, in the example, the right leg is returning a positive error (since the ground is higher than expected) while the left leg is returning a negative error (the ground is lower than expected).

You can use these error values in order to update the vertical position of your character over time, so it reacts to the environment according to changes in ground height (i.e, the error gets reduced).

Tip:

When doing so, you may want to favour negative errors (ground lower than expected) over positive errors. This is because negative errors will try to extend the leg (to reach down), while positive errors will try to contract the leg. It is easier to correct negative errors by updating the position of the character (i.e, lowering the pelvis) than by using IK to extend the leg, since in most poses the leg is already extended and thus the available range is quite limited (while the range to contract the leg is far larger).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- A boolean, `hitSomething`. This will be set to true if the foot placement was able to detect ground below the feet (through ray casting). Otherwise, false is set. This can used by application logic: for example, to detect when the character is not supported by its feet. Note that when detecting the ground the variable also checks that the ground is actually close enough to touch the foot (the raycasts can extend beyond the foot). The distance check is controlled by the minAnkleHeightMS setup variable.

4.3.16 Using hkaPose

4.3.16.1 Introduction

When writing application code, you may find that very often you have to switch between the local and model space representation of the bones of a character[15]Some operations, like blending, work better in local space, while some others, like mapping from one skeleton to another, or IK, work in model space. Also, sometimes you may want to modify the model space transform of a bone while still keeping the local space transform of its children, hence working in both spaces at the same time. The hkaPose class helps you working with this dual (local+model space) representation of poses seamlessly and efficiently. Consider the following example:

An Example

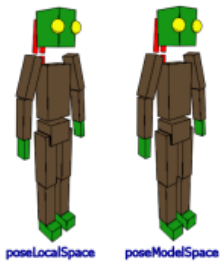
In this example, we'll want to apply Look At IK to a pose taken from an animated skeleton (hkaAnimatedSkeleton). We start by sampling the pose from that animated skeleton:

```
// We start with a pose in local space
hkLocalArray<hkQsTransform> poseLocalSpace(skeleton->m_numBones);
poseLocalSpace.setSizeUnchecked(skeleton->m_numBones);
animatedSkeleton->sampleAndCombineAnimations(poseLocalSpace.begin());
```

Now, we have checked the documentation of the Look At IK solver and noticed that it works in model space (it modifies a bone in model space). Thus, we need to convert the pose to model space:

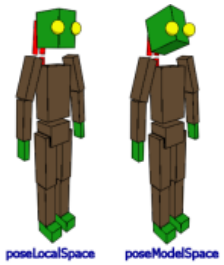
[15] For more information about these spaces, please check the World, Model and Local Spaces section at the beginning of the Animation Runtime chapter

```
// We convert it into model space
hkLocalArray<hkQsTransform> poseModelSpace(skeleton->m_numBones);
poseModelSpace.setSizeUnchecked(skeleton->m_numBones);
hkaSkeletonUtils::transformLocalPoseToModelPose(skeleton->m_numBones, skeleton->m_parentIndices,
  poseLocalSpace.begin(), poseModelSpace.begin());
```



Good. We can now call the IK solver with the bones in the model space pose:

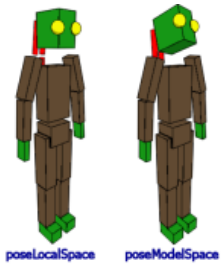
```
// We do IK on the head to look at the enemy
hkaLookAtIkSolver::Setup ikSetup;
ikSetup.m_fwdLS.set(0,1,0);
ikSetup.m_limitAxisMS.setRotatedDir(poseModelSpace[neckIndex].getRotation(), hkVector4(0,1,0));
ikSetup.m_limitAngle = HK_REAL_PI / 5.0f
hkaLookAtIkSolver::solve(ikSetup, enemyPositionMS, 1.0, poseModelSpace[headIndex]);
```



Well, the IK did his job and modified the head to point in the right direction. Notice, however, that we have only modified the model space transform of the head. What happens to children bones of the head, like the eyes or the pony tail? We want those to rotate with the head. At the moment, we only updated the model space transform of the head; we'll need to update the children accordingly:

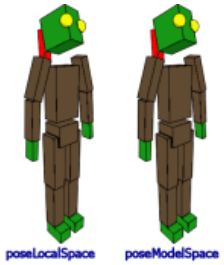
```
// Ok, that moved the head, but what about the eyes and the ponytail?
// We want to keep the children local transform respect to their parent....
hkArray<hkInt16> childrenOfHead;
hkaSkeletonUtils::getDescendants(*skeleton, headIndex, childrenOfHead);
for (int i=0; i<childrenOfHead.getSize(); i++)
{
    hkInt16 parentIdx = skeleton->m_parentIndices[childrenOfHead[i]];
    poseModelSpace[i].setMul(poseModelSpace[parentIdx], childrenOfHead[i])
}
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Good. This has now updated the model space transform of those bones. However, the local space transform array hasn't been updated yet, so at the moment `poseLocalSpace` and `poseModelSpace` don't describe the same pose. If we need the new pose in local space, we'll have to update it too:

```
// Update the local space pose
hkaSkeletonUtils::transformModelPoseToLocalPose(skeleton->m_numBones, skeleton->m_parentIndices,
  poseModelSpace.begin(), poseLocalSpace.begin());
```



Fine, this will work; it will recalculate all the local space transforms according to the new model space pose. However, thinking about it, it would actually have been enough to do this:

```
// This would have been enough. Neck is the parent of Head
// neckFromHead = neckFromModel * modelFromHead = inverse(modelFromNeck) * modelFromHead
poseLocalSpace[headIndex].setMulInverseMul(poseModelSpace[neckIndex], poseModelSpace[headIndex]);
```

Since when we updated the children of the head, we already made sure they kept their original local transform. So the head's local transform was the only local transform to update. By noticing this, we only used one transform multiply instead of n (where n is the number of bones in our skeleton).

If you imagine now the case were we applied LookAt IK to not only the head but also both eyes, followed by other IK operations, mapping and blending, you will notice that tracking the transforms that have been modified and those that need to be updated may become a complicated task which requires a fair amount of bookkeeping and where conceptually simple changes, like changing the order of two IK operations, will easily break the code.

In order to facilitate operations with poses, Havok Animation provides the hkaPose class. This class has the following properties:

- It represents a pose for a skeleton

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- It operates with both local and model space representations of this pose
- These representations are stored but only updated when required (lazy update)
- Changes on individual transforms are tracked and only the minimum number of transforms are flagged for update
- Operations (like hierarchy traversing and updates) have been optimized and do not require allocations (temporary arrays, etc)
- Direct access to both model and local space pose representations is also available
- All data can be stored using stack-based allocation to speed up construction and destruction.

The example above, if using hkaPose, would look like this:

```
// We start by filling a pose with an array of transforms in local space
hkaPose thePose(skeleton);
animatedSkeleton->sampleAndCombineAnimations(thePose.writeAccessPoseLocalSpace().begin());
```

```
// We do IK on the head to look at the enemy
hkaLookAtIkSolver::Setup ikSetup;
ikSetup.m_fwdLS.set(0,1,0);
ikSetup.m_limitAxisMS.setRotatedDir(thePose.getBoneModelSpace(neckIndex).getRotation(), hkVector4(0,1,0));
ikSetup.m_limitAngle = HK_REAL_PI / 5.0f
hkaLookAtIkSolver::solve(ikSetup, enemyPositionMS, 1.0, thePose.accessBoneModelSpace(headIndex, hkaPose::
  PROPAGATE));
```

And that's it. Notice the following things:

- We don't need to convert the pose to model space at the beginning. When accessing the model space transform of the head (to pass it to the IK solver), the model space transform of the head (and its parents) gets calculated and updated in the pose. The model space transforms of other bones (like the children of the head) is not calculated; there is no need for it yet (and as we know they will change once the head is rotated, so it would be wasted effort).
- After the IK, we don't need to update the model space transform of the children of the head yet. By passing the PROPAGATE flag, we told the hkaPose that modifications on the head should propagate to the children. The hkaPose then knows that the local space transforms of those children are still valid and hence don't need to be modified. Their model space transformation won't be calculated until somebody asks for it (so, if we now do further IK, we haven't wasted the effort)
- Similarly, we don't need to update any local space transforms yet. If in the future we ask for any them, the hkaPose instance will know that only the local space transform for the head is not up-to-date, but all the rest are.
- Apart from the performance gain caused by the reduced number of operations, the code is now much simpler. There is no bookkeeping required since, from the application point of view, hkaPose is always up-to-date, both in model and local space.

The following sections will give you more detail on how hkaPose works and how to make the most of it.

4.3.16.2 Overview

hkaPose objects keep a dual (local space and model space) representation of a pose (a set of n transforms for the n bones of a skeleton). Both representations are kept in sync on demand (lazily), so only the minimum amount of calculations are done whenever the pose is modified or accessed. The figure below shows a conceptual representation of an hkaPose object:

	0	1	2	3	4	5	6	7	8	9
Parent		0	0	2	3	4	5	6	1	8
LocalPose	L	L	L	L		L	L	L	L	L
LocalDirty					x					
ModelPose	M		M	M	M					
ModelDirty		x				x	x	x	x	x

In this example, most transforms in the local space representation of the pose are stored up-to-date (except for bone 4), while only a handful of model space transforms are (bones 0, 2, 3 and 4). But notice that both the local and model space transformation of any bone can be calculated at any time based on the information stored.

Note:
For ease-of-use the hkaPose class also contains a 'Float Slot Values' array into which float tracks may be sampled. Since all float slots are treated completely indpendently (there is no concept of 'local/model' space, and no hierarchy for float slots), the management of this part of the hkaPose class is trivial, and we will not go into it in detail here.

4.3.16.3 Creating an hkaPose object

There are three ways to create an instance of hkaPose. All constructors require a pointer to the skeleton associated with the pose.

- Create an uninitialized pose:

```
hkaPose (const hkaSkeleton* skeleton);
```

This constructor allocates the memory required to store the pose using calls to hkArray.setSize() - which in turn will call hkAllocateChunk().

This is the simplest constructor. Since the memory is allocated using normal allocation routines, it may be slower than the stack-based allocation constructor (described next)

Warning:
This and the next constructor create an uninitialized hkaPose instance. This means that none of the stored transforms in the pose are valid, and henceforth, the pose needs to be initialized before it's used. You should always call either `setToReferencePose()`, `setPoseLocalSpace()`, `setPoseModelSpace()` or use the write-only methods `writeAccessPoseLocalSpace()` or `writeAccessPoseModelSpace()` to initialize the pose after using these constructors. All these methods are explained later in this chapter.

- Create an uninitialized pose in preallocated memory:

```
hkaPose (const hkaSkeleton* skeleton, void* preallocatedMemory);
```

This constructor creates an hkaPose instance and uses a preallocated memory buffer for storage. You can get the size of the required buffer by using the static method `getRequiredMemorySize (hkaSkeleton*)`. This is particularly useful if you want to use a fast memory allocation routine, like stack-based memory allocation (check the Havok Base Library documentation for more details). The most common usage is presented below:

```
// We pre-allocate the memory in an hkLocalBuffer
hkLocalBuffer<char> localBuffer( hkaPose::getRequiredMemorySize( skeleton ) );
hkaPose thePose(skeleton, localBuffer.begin());
```

- Create an initialized pose:

```
// enum PoseSpace { MODEL_SPACE, LOCAL_SPACE };
hkaPose (PoseSpace space, const hkaSkeleton* skeleton, const hkArray<hkQsTransform>& pose);
```

This constructor initializes an hkaPose instance with a pose defined either in local or model space (as defined by the `space` parameter). It is equivalent to calling the `hkaPose(skeleton)` constructor followed by either `setPoseLocalSpace(pose)` or `setPoseModelSpace(pose)`.

4.3.16.4 Working with Single Bones

getBone() methods (direct read only)

You can retrieve the model or local space transform of any bone by using the methods

```
const hkQsTransform& getBoneLocalSpace (int boneIdx) const;

const hkQsTransform& getBoneModelSpace (int boneIdx) const;
```

If the transform in question was up-to-date, its value will be returned; otherwise it will be calculated (and stored, alongside any other bone transforms that had to be calculated).

Tip:
If your algorithm is going to be asking for the local or model transform of all the bones in the pose, consider the alternative of using the `getPose()` methods for optimized read-only access to a single pose representation. These methods are described in the Working with Multiple Bones section.

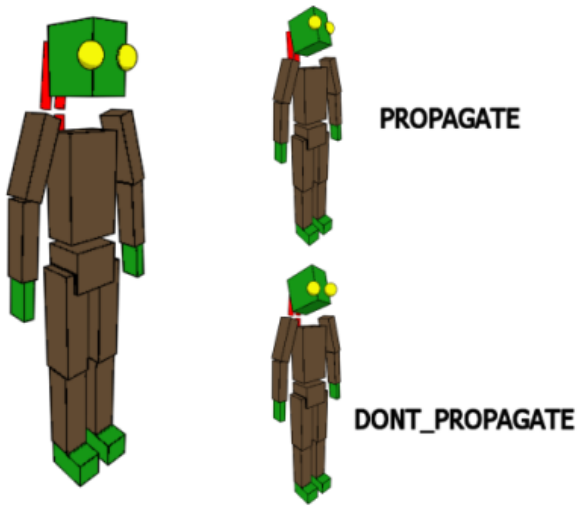
setBone() methods (write only)

You can also modify the local or model space transform of any bone using the methods:

```
void setBoneLocalSpace (int boneIdx, const hkQsTransform& boneLocal);

//enum PropagateOrNot { DONT_PROPAGATE, PROPAGATE};
void setBoneModelSpace (int boneIdx, const hkQsTransform& boneModel, PropagateOrNot propagateOrNot);
```

Setting the transform of a bone will store that transform in the hkaPose object, clean its "dirty" flag and possibly update or dirty the transforms of other bones affected by the change.

When setting the transform of a bone in model space, you can choose whether the model space transforms of its descendants are affected as well (`PROPAGATE`) or not (`DONT_PROPAGATE`), using the `propagate` parameter.



In the example above, we have set the transform of the head in model space. Notice how using the `PROPAGATE` flag modifies (flags as "dirty") the model space transforms of the descendants of the head (eyes and pony tail), while using the `DONT_PROPAGATE` flag keeps the model space transform of those bones (but, in turn, invalidates the local space transform of the children of the head).

When using `setBoneLocalSpace`, transformations are always propagated (i.e., the local space transforms of the children are always kept).

Tip:
If your algorithm is going to set the model or local space transforms of all bones in the pose, consider the alternative of using `setPose()` and `writeAccessPose()` methods for optimized write-only access (or the `accessPose()` methods for optimized read+write access) to a single pose representation. These methods are described in the Working with Multiple Bones section.

accessBone() methods (direct read+write)

You can also get direct access (a non-const reference) to a bone transform using the methods:

```
hkQsTransform& accessBoneLocalSpace (int boneIdx);
hkQsTransform& accessBoneModelSpace (int boneIdx, PropagateOrNot propagateOrNot = DONT_PROPAGATE);
```

Before returning, these methods will ensure that the transform in particular is up-to-date (read access). They will also assume that you are going to *immediately* change the value of the transform (write access), and so they will also go ahead and invalidate other transforms as required (the `propagateOrNot` flag works as it did in the `setBoneModelSpace()` method).

These methods can be used to perform some operations without requiring temporary variables, for example the following code:

```
hkQsTransform tempTransform = thePose.getBoneModelSpace(handIdx);
tempTransform.m_rotation.mul(extraRot);
thePose.setBoneModelSpace(handIdx, tempTransform, PROPAGATE);
```

can be written as:

```
thePose.accessBoneModelSpace(handIdx, PROPAGATE).m_rotation.mul(extraRot);
```

The other common reason for using the `accessBone()` methods is to modify a pose through a method that operates on a transform. For example, the Look At IK solver modifies a transform rather than a full pose (as we saw in the introductory example):

```
// hkaLookAtIkSolver::solve() takes a model space transform as an in+out (read+write) parameter
hkaLookAtIkSolver::Setup ikSetup;
ikSetup.m_fwdLS.set(0,1,0);
ikSetup.m_limitAxisMS.setRotatedDir(thePose.getBoneModelSpace(neckIndex).getRotation(), hkVector4(0,1,0));
ikSetup.m_limitAngle = HK_REAL_PI / 5.0f
hkaLookAtIkSolver::solve(ikSetup, enemyPositionMS, 1.0, thePose.accessBoneModelSpace(headIndex, hkaPose::
  PROPAGATE));
```

Warning:
When working with `accessBone()` methods, it is important to ensure that the non-const reference to the bone transform is used inside a limited scope (like in the examples above). The golden rule is: *references to bone transforms inside the hkaPose object should only be used for a short scope, and should never be used after another hkaPose method is called.*

4.3.16.5 Working with Multiple Bones

In some occasions, you may find that you want to operate on all bones in a single space: for example, when skinning you may require the model space transforms of all bones; or, when blending, you may need to access all the local space transforms. Also, you may need to interface to some Havok Animation (or your custom) components that take poses as arrays of transforms in a single space, rather than hkaPose objects.

In order to facilitate all this, there are different methods in hkaPose that allow you to get read, write, and read+write access to a single representation of all bones in the pose.

getPose() methods (direct read only)

You can get a const reference to full local space or model space representation of a pose by using the methods:

```
const hkArray<hkQsTransform>& getPoseLocalSpace() const;
const hkArray<hkQsTransform>& getPoseModelSpace() const;
```

These methods will ensure the internal representation of the local or model space pose is up-to-date, and will return a reference to that representation (an array of transforms). This is particularly useful if you need to interface with an object or method that takes a pose specified as an array of transforms rather than an hkaPose object.

Notice that since these methods return a reference to internal data, this data is only valid as long as no non-const methods are called on the hkaPose object.

setPose() methods (write only)

You can set the model or local space transform of all bones in a pose by using the methods:

```
void setPoseLocalSpace (const hkArray<hkQsTransform>& poseLocal);
void setPoseModelSpace (const hkArray<hkQsTransform>& poseModel);
```

These methods will store the given pose (in local or model space) and invalidate the alternative representation.

accessPose() methods (direct read+write)

You can get a non-const reference to the whole pose in a single space (local or model) by using one of these methods:

```
hkArray<hkQsTransform>& accessPoseLocalSpace();
hkArray<hkQsTransform>& accessPoseModelSpace();
```

The `accessPose()` methods will ensure all transforms are up-to-date for that particular representation, and will return a non-const reference to that representation (as an array of transforms). Since the reference is non-const, it will assume that you are going to modify those transforms and, therefore, it will also invalidate the internal transforms stored for the alternative representation (so, for example, calling `accessPoseLocalSpace()` will update all local space transform and invalidate all model space transforms).

Warning:
As with the single-bone version, `accessPose()` expects you to use the non-const reference returned for a limited scope, and never after further calls to other methods in the hkaPose object are made.

These methods are useful if a section of your algorithm is going to work and modify all (or most of the) bones in the pose using a single space, or when interfacing to a function/object that does so (takes an input+output pose as an array of transforms in a single space).

writeAccessPose() methods (direct write only)

For the times you want write-only access to a full pose in a single space (local or model), you can use one of these methods:

```
hkArray<hkQsTransform>& writeAccessPoseLocalSpace();
hkArray<hkQsTransform>& writeAccessPoseModelSpace();
```

These methods differ from the `accessPose()` methods described above by the fact that they do not ensure that the particular representation requested is up-to-date before returning a reference to it. The value of those transforms is undefined and therefore *you must set all the transforms in the array before*

using the hkaPose object again. These methods assume that you will initialize *all* transforms to the right value, and therefore will clear the dirty flag for all transforms of the requested space while setting the dirty flag for the transforms of the alternative space.

The most common usage of these methods is for the initialization of hkaPose objects, as well as to pass poses to methods or functions that fill a full pose, given as an *out* parameter, for example:

```
// hkaAnimatedSkeleton::sampleAndCombineAnimations takes an array of local space transforms as an out (
  write only) parameter
hkaPose thePose(skeleton);
animatedSkeleton->sampleAndCombineAnimations(thePose.writeAccessPoseLocalSpace().begin(), HK_NULL);
// Or, more correctly if the skeleton contains float slots also
animatedSkeleton->sampleAndCombineAnimations(thePose.writeAccessPoseLocalSpace().begin(),
  thePose.writeAccessFloatSlotValues().begin());
```

4.3.16.6 Other Methods

sync() methods

You can enforce the synchronization of the local or model space internal representations, or both of them, by using the methods:

```
void syncLocalSpace () const;
void syncModelSpace () const;
void syncAll () const;
```

setToReferencePose()

You can set or initialize a pose to the reference pose of its associated skeleton (`hkaSkeleton`) by using the method:

```
void setToReferencePose ();
```

enforceSkeletonConstraints()

You can ensure that all skeletal constraints in the pose are enforced by calling one of these methods:

```
void enforceSkeletonConstraintsLocalSpace ();
void enforceSkeletonConstraintsModelSpace ();
```

Currently, the only constraint that can be defined on a skeleton is whether particular bones can translate or not (`hkaBone::m_lockTranslation`). The methods above will check and ensure that no translations have been applied to bones with that flag.

The two methods perform the same operation, but they operate on different spaces. These methods are mostly used internally.

4.3.16.7 Caveats

When using hkaPose be careful when using the accessPose methods in parameter lists. For example:

```
hkArray<hkQsTransform>::const_iterator rPose = ragdollPose.getPoseModelSpace().begin();
 hkArray<hkQsTransform>::const_iterator aPose = animPose.getPoseLocalSpace().begin();
 hkArray<hkQsTransform>::iterator       aPoseM= animPose.accessPoseModelSpace().begin();
 mapper->mapPose( rPose, aPose, aPoseM );
```

will behave as expected, whereas

```
mapper->mapPose(ragdollPose.getPoseModelSpace().begin(), animPose.getPoseLocalSpace().begin(),
  animPose.accessPoseModelSpace().begin());
```

may behave unexpectedly if the compiler evaluates the third parameter before the second parameter. The call to accessPoseModelSpace invalidates all the local space flags and the call to getPoseLocalSpace will reevaluate the local pose using the out of date model pose.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.4 Animation Examples and Use Cases

4.4.1 GDC 2005 demo

The Havok GDC 2005 demo shows several features of the Havok Complete package. These include:

- Animation blending and playback.
- Integration with the Havok character controller.
- Foot, hand and look at IK.
- Local hit reactions using a ragdoll controller that works in model space.
- Getting back up using pose matching.

After a brief introduction on the demo setup, the following chapter describes each of these in detail.

4.4.1.1 Demo Setup

The GDC demo uses different representations for our main character. When running around and colliding with the enviroment we use the Havok character controller. We use a capsule proxy shape which is shown in the image below in yellow. You can display this in the demo from the Options menu [Display->ShowProxy]. This proxy is responsible for resolving the motion of the characters reference frame. During blending, animations that contain movement the animation system output a change in reference frame. This is used to drive the motion of this proxy. This process is described in detail in the section on Motion Extraction.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

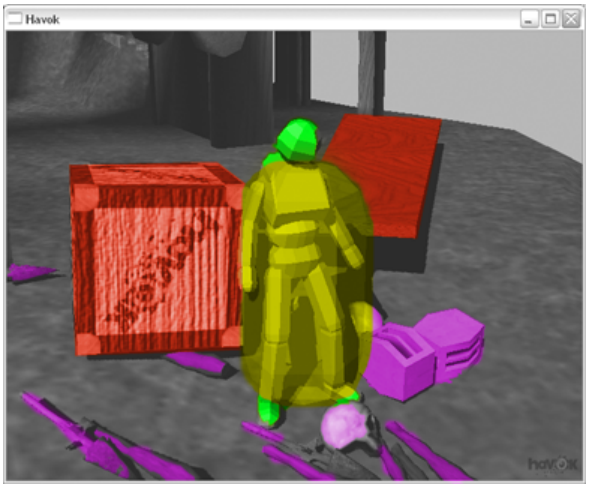


Figure 4.46: Collision layers in the demo

As well as resolving collision with the static landscape the proxy is responsible for interaction with dynamic game objects. In our demo these objects are the seesaw and large crates shown below in red. For these objects their interaction needs to be well defined we restrict the movement of the proxy by controlling the force that it applies to objects. This gives the character a sense of strength.

The other dynamic objects in the scene marked in purple below are flagged as debris items. These do not interact with, or affect the movement of, the character proxy. They do interact with the ragdoll representation shown in green.

The ragdoll representation is a constrained rigid body system. Under normal operation the pose from the animation system is used to drive the positions of the rigid bodies using a ragdoll rigid body controller (`hkaRagdollRigidBodyController`). Since these bodies are driven by the animation they collide realistically with the debris. In particular their velocity comes directly from the animation system. By default the legs are keyframed and the upper body is dynamic. This means the upper body will react to collision from both the static environemnt and the dynamic objects. If you run against the wall of pick up a dynamic object and hit the character with it you will see this reaction.

When the character falls and turns into a powered ragdoll it changes the motion state of the bodies and their collision filter information to allow it to interact with everything (including debris). In total we use eight distinct layers in the demo to control interaction. These are

- Landscape Layer : containing the static geometry for the level. In this demo we have both a visual geometry and a simplified collision geometry
- Proxy Layer : contains the proxy for the character controller.
- Debris Layer : containing the small dynamic objects in the scene and the bricks in the wall on the top level. Each of these have different masses
- Moveable Environment : contains the see saw and the large boxes / crates.
- Ragdoll Keyframed : contains the bodies of the ragdoll that are currently keyframed (usually the lower body)
- Raycast : this layer is used to control which items to ray cast against when performing foot and hand IK.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- Ragdoll Dynamic : contains the rigid bodies for the ragdoll which are not keyframed. When the ragdoll falls all ragdoll bodies are switched into this layer.
- Picking : A Layer to control which items are effected by our mouse pointer.

The interactions are shown here. Layers enabled are shown in green. Layers with interactions disabled are shown in red.

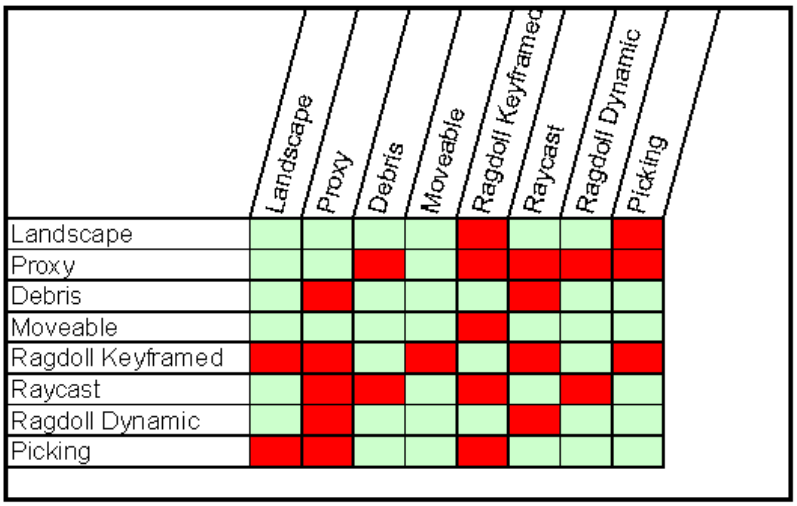


Figure 4.47: Layers enabled in the demo

Note that some layers are disabled for game play and control reasons. In particular the moveable objects, like the see saw and the large boxes, only interact with the proxy. The proxy pushes and interacts with objects in a more controlled way than the driven ragdoll. These larger objects would typically be involved in game play and changes in animation content should not change their response.

4.4.1.2 Animation playback and blending

Animations in the demo are played back using the low level mixing and blending functionality of hkAnimation. These low level calls are made through 2 higher level interfaces. The first is an animation command processor. (GdcAnimationMachine) This defines a very simple data protocol which translates messages into calls to animation controls. Another interface, the animation state machine (GdcStateContext, GdcStateManager), generates lists of these commands as it transitions between states.

4.4.1.3 Integration with the Havok character controller.

The character proxy is driven by animation at each frame to produce a new position. This process is described in Motion Extraction. Since the landscape has significant discontinuities, especially on the stairs, it's movement contains lots of high frequency noise. If we simply take the position of the proxy and use this as the reference frame for the model space of our animation then all this noise will be seen in the final animation. To combat this we decouple the proxy slightly from the animation. We allow it to drift (up to a limit) in both the vertical and horizontal direction. An error is fed back into the proxy on the next step. The decoupling is best viewed by turning on the character proxy in the display options and running up and down the stairs.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

4.4.1.4 Foot, hand and look at IK.

The parameter and setup for the foot, hand and look at IK are detailed in the Inverse Kinematics section. One useful trick shown in the demo feeds back the error from the IK solver directly into the pelvis' vertical position. In doing so the character automatically adjusts the pelvis according to the results of the IK. The feedback loop is very straighforward. The IK returns an positive error if the leg is compressed and a negative error if it does no reach its target. We compute a delta movement based on the maximum compression and stretch. Experimentally we have found that biasing the delta based more toward the stretch gives a more pleasing result (w is close to 1.0 in the code below). This delta is damped and then applied to the pelvis.

```
const hkReal weightedError = w * hkMath::min2(leftError, rightError) + (1.0f - w) * hkMath::max2(leftError
  , rightError);
hkReal pelvisDelta = weightedError * m_options.m_IK.m_pelvisFeedback; // damp it
verticalDisplacementInOut += pelvisDelta;
```

4.4.1.5 Local hit reactions

We drive the ragdoll representation of the character to match the current animation. If dynamic objects collide with the ragdoll the controller will briefly diverge from its desired location, the collision will be resolved and the controller returns the ragdoll to the desired pose. This can be seen by picking one of the debris objects (bricks, pulleys, skulls) and using the mouse spring to make it colide with the character while he is standing idle. The idle animation is playing which would normally keep the character standing upright. Normal collision detection and resolution takes place and the ragdoll bones move. The controller attempts to correct the movement and drive the ragdoll back to the idle pose. Obviously the key here is in choosing parameters for the controller strength. If the controller is too strong then we will see little movement in the character when objects collide. If it is too weak then the controller will not follow the animation properly.

In the demo the bodies of the ragdoll are automatically coloured to show the stress on each of the bones. This is a measure of how far the ragdoll has diverged from the animation. It can be used to drive sound, damage or trigger a full ragdoll sequence. For details on how this controller works and parameter choices please see `hkaRagdollRigidBodyController` and the utility class it relies on `hkaKeyFrameHierarchyUtility`

4.4.1.6 Animation driven ragdoll deaths

When the character is triggered into full ragdoll we switch all the bodies to dynamic and allow the physics to drive the animation. Using physics only gives a relatively lifeless performance. The ragdolls appear limp and floppy as if they are totally unconscious. Instead we use the controller, the `hkaRagdollPoweredConstraintController`, to layer an animation onto the local joints of the ragdoll, Even though the ragdoll is tumbling and falling through the air the controller drives the joints to match the currently playing animation. In the demo if you enable GetUp->DieAnimation it plays a death animation (which brings the character into a ball) as soon as we transition to ragdoll. As can be seen the general gait of the character matches the animation even though his overall orientation and position are different.

4.4.1.7 Getting back up using pose matching.

We also use this controller to help us make the ragdoll seemlessly get back up again. As the character is in the air tumbling through the scene we are constantly examing its pose and attempting to find a pose it can get back up from. The controller drives to this desired pose so when the character has settled after the fall they are usually in a configuration close to the initial pose in a get up animation.

In this demo we have just 3 get up animations. In each of these an artist has marked frames in the animation using annotations. When these animations are loaded we scan the annotations an search for marked frames. For each frame that is marked we sample the animation at that point and add it to a list of valid poses to get up from see GdcDemo::initPoseMatching. As the character tumbles through the air a utility function `hkaPoseMatchingUtility` find the closes poses that matchs the current ragdoll orientation and drives to that.

An artist controls the get up point by placing annotations in the modeller. These annotations are set in Max using note tracks and in Maya by creating a custom attribute.

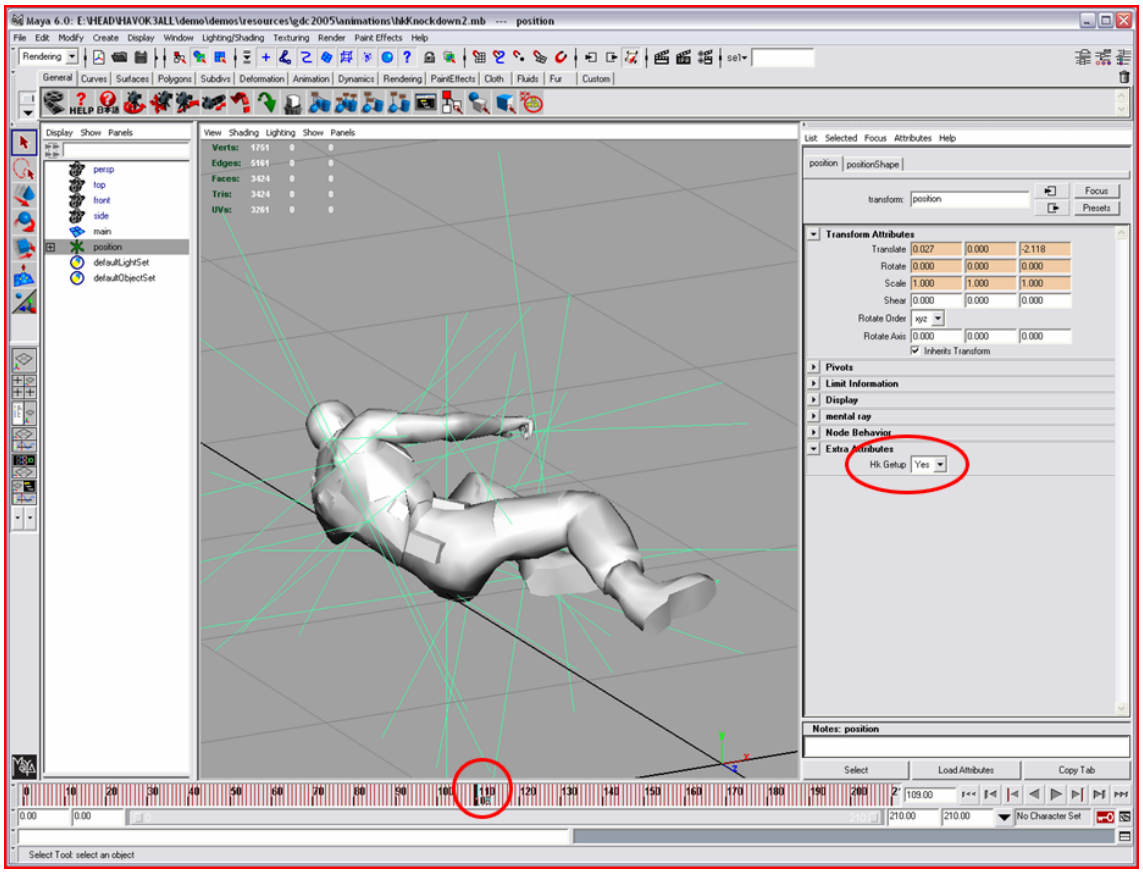


Figure 4.48: Annotating poses in the modeller

Our method for searching the pose database and matching against the current ragdoll configuration is simple and efficient. We choose three reference points on the body and use these to build a quaternion representing the orientation of the body. The reference points we choose include the main upper body axis (from pelvis to neck) which is usually the most discriminating axis. Before adding this reference pose we remove all twist about a specified up axis since we know we can always get up facing any direction.

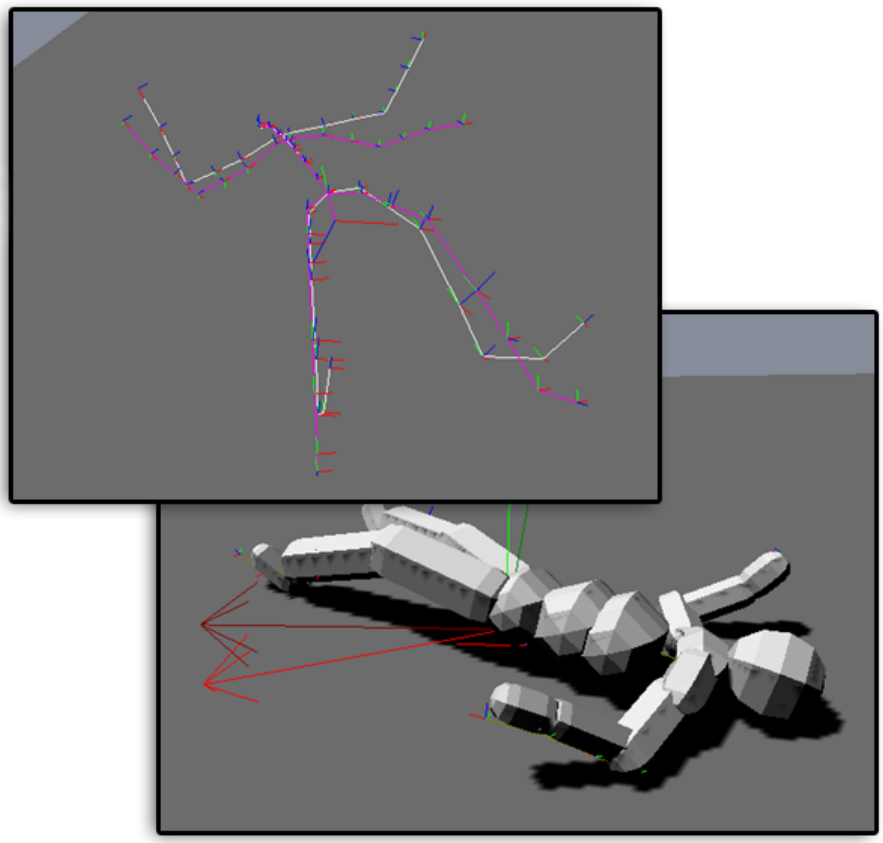


Figure 4.49: Pose Matching reference frame

One of the advantages of this scheme is that it is very efficient to search for a given pose. It requires we build a frame from the reference points in model space and compare this with the other frames held in the database, this reduces to constructing a quaternion and performing a quaternion dot product with the precomputed frames.

The diagram above shows a choice of reference frame and screen shots from a demonstration of the technique. The actual animation pose we are attempting to drive to is shown in white. The current pose of the ragdoll is overlaid in purple. As can be seen the match is close but not exact. It does not need to be exact because during get up we blend between the current ragdoll pose and our animation making the transition perfectly smooth.

Controlling pose switching

As the ragdoll falls we are constantly matching new poses to get up from. If our resting pose is very close to the boundary between two matching poses in the database then the ragdoll can toggle between them continuously. This usually occurs when the ragdoll does not settle quickly in one of the desired states. To eliminate this we must either change the pose matching function to a more discriminating one or add hysteresis to the code. In the demo we add hysteresis. When we find a new pose we do not make the switch until that pose remains the best match for a fixed time period. This prevents the toggling between similar states since it allows the ragdoll to settle. In addition we also stop looking for better poses once the ragdoll has settled. We simply check the velocity of the pelvis to make this decision. See the method GdcDemo::matchPose for details.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Chapter 5

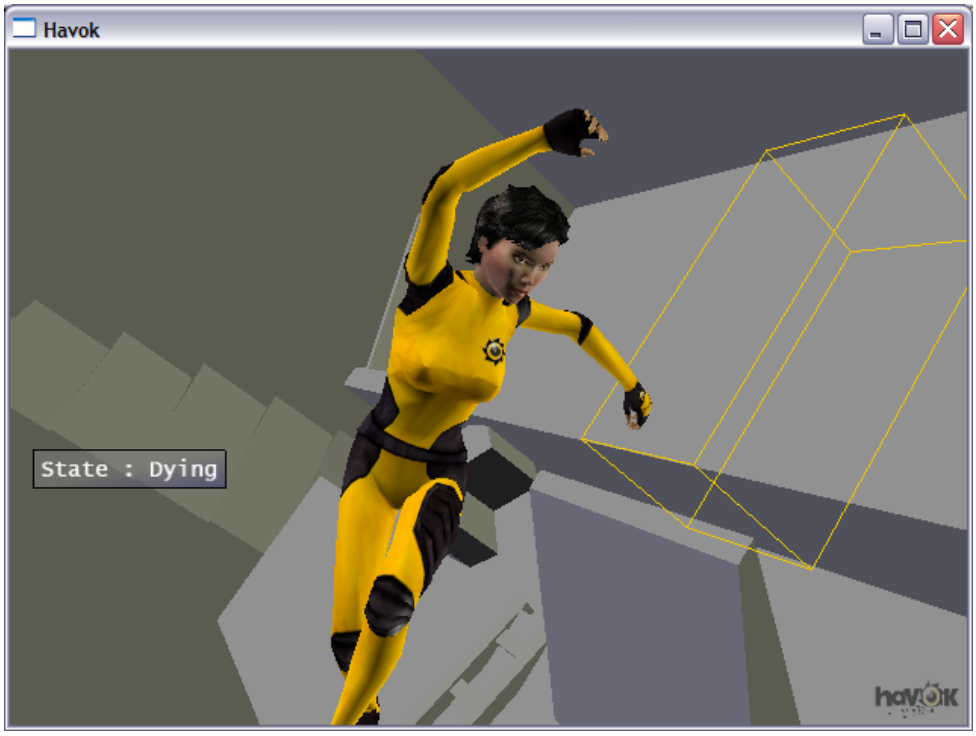
Physics and Animation Integration

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

5.1 Integrating Havok Animation with Havok Ragdolls

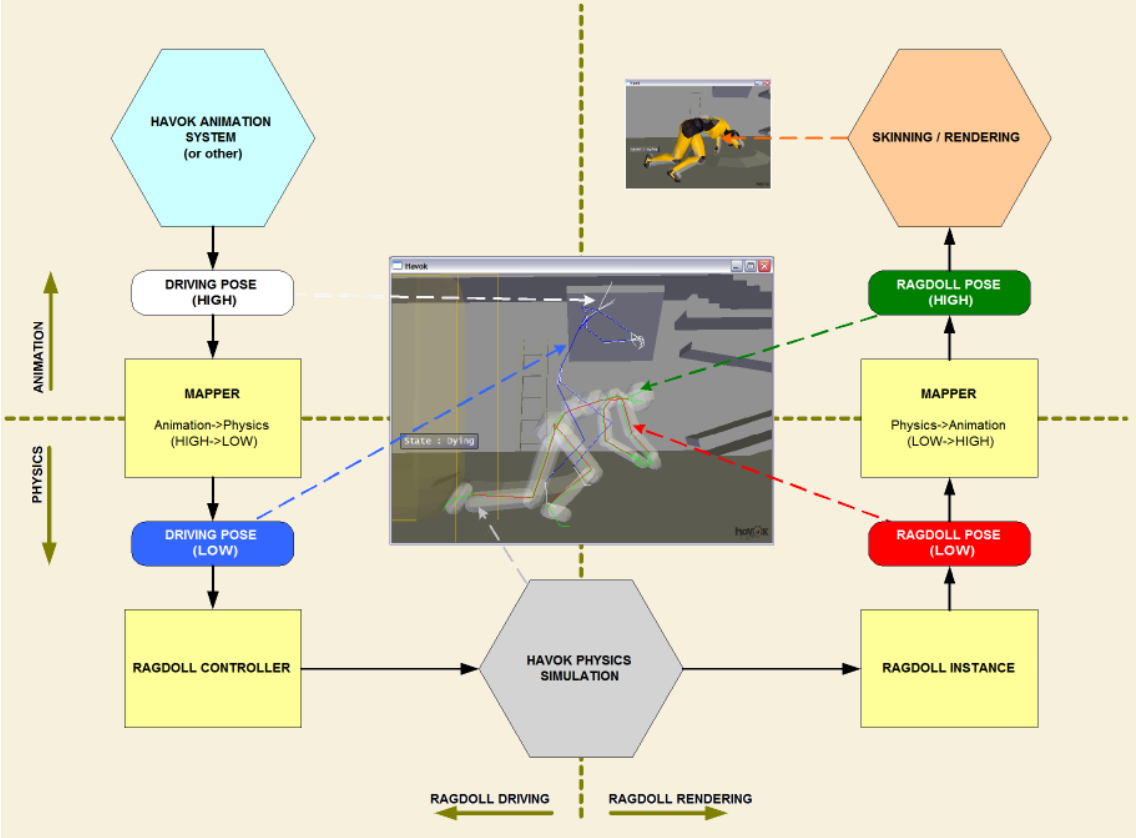
5.1.1 Introduction

This chapter describes the different tools provided by the Havok Animation and Havok Physics SDK in order to integrate physically-simulated characters (ragdolls) into an animation pipeline.



5.1.2 A Typical Scenario

The following diagram shows the typical flow of information in a system where ragdolls have been integrated with the animation system.



In the application above, the animation system is used to drive physical ragdolls, i.e., make the physical representation of the character achieve a desired pose. Also, the current physical pose of the physical ragdoll is then used to render the visual representation of the character (skin it).

Therefore, we have two quite distinct processes here:

- *Ragdoll Driving* (left side of the diagram) : Given a "desired" pose taken from the animation system, feed the physics simulation with some input that will drive the physical ragdoll towards that desired pose.
- *Ragdoll Rendering* (right side of the diagram) : Given the "current" pose taken from the physics simulation, feed the animation system with a pose that can then be rendered / skinned.

Notice that, for many reasons (mostly performance), the physical representation and the animation representation of a character are usually different. While an animation skeleton for a character may have 50 bones or more, we will usually represent this character with a ragdoll skeleton of less than 15 bones.

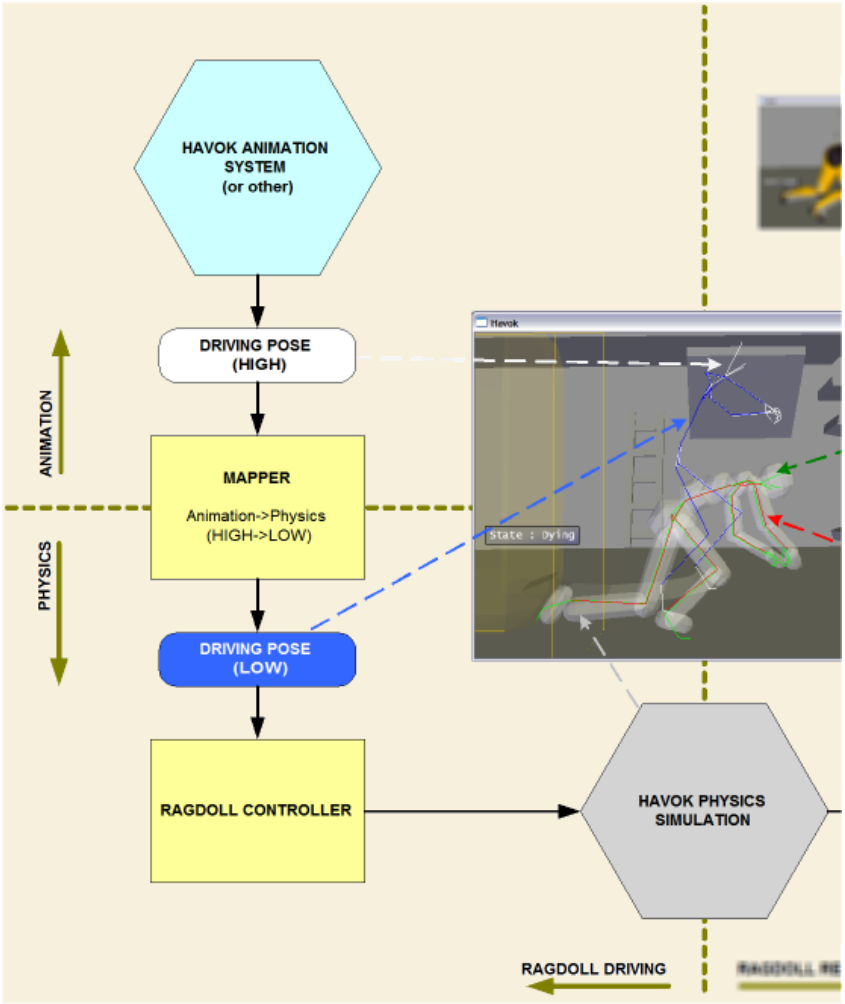
Therefore, we can distinguish as well between two different spaces, depending on the kind of poses we work with:

- *Animation Space* (top part of the diagram) : Poses have a higher-bone count
- *Physics Space* (bottom part of the diagram) : Poses have a lower bone count (a bone for each rigid body)

A lot of the work involved on ragdoll integration is associated with moving between these two spaces. Mapper (**hkaSkeletonMapper**) objects are used in order to do so.

Let's take a look at how the information flows both when driving and when rendering ragdolls.

5.1.2.1 Driving Ragdolls



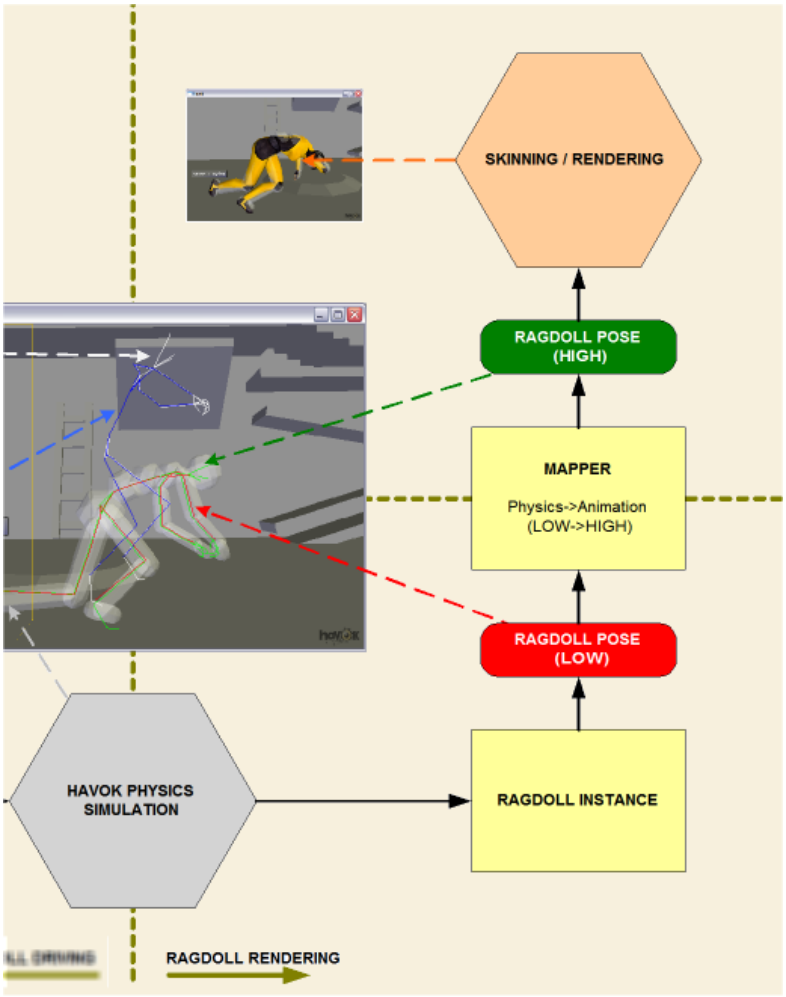
We start with a high bone count (animation) pose given to us by the animation system. This pose is depicted in white in the diagram, and we call it the "driving" pose.

When driving ragdolls, we want to convert this pose to a low bone count (physics) pose. We do this using a mapper (**hkaSkeletonMapper**). The mapper here will map from "high" to "low" bone count poses. The output is, then, a low bone count pose (depicted in blue). This pose is also a "driving" pose, but now using a low bone count representation.

Given this pose, we now want the physical representation of the character (the ragdoll) to try to match the pose. We use a ragdoll controller for that. The output of this object is a set of forces / torques / other input to the physics system.

We don't always need to pass the driving pose through a ragdoll controller. For example, we may want to force the rigid bodies to match the given pose (rather than driving them towards it). In this case, we can pass this pose straight to the rigid bodies positions and orientations by calling **hkaRagdollInstance::setPose()** .

5.1.2.2 Rendering Ragdolls



When rendering ragdolls, we start with a set of rigid bodies simulated by the physics. We can get a pose based on those rigid bodies by using a ragdoll instance, and calling **hkaRagdollInstance::getPose()** . This ragdoll pose has a low-bone count, and it's depicted in red in the diagram above.

Since skins are usually bound to the original high bone count skeleton, we need to convert this low bone count pose to a high bone count pose. Again, we use a mapper to do so. The output of this mapper is a high bone count pose, depicted in green.

Finally, we can use this pose to render and skin the character, or possibly modify it even further (blend other animations, apply IK).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

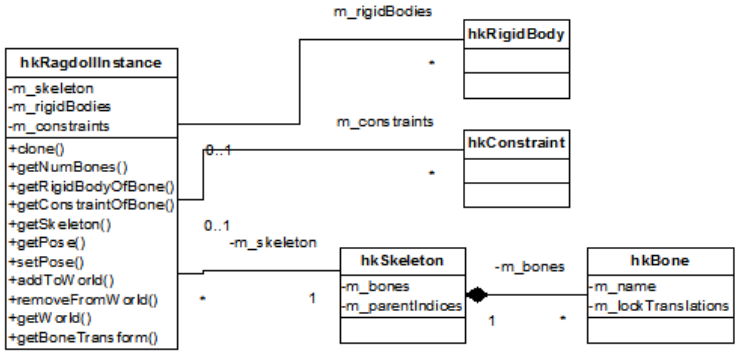
5.1.3 Ragdoll Instances

A lot of functionality relating to the manipulation of ragdolls is now encapsulated in the **hkaRagdollInstance** class.

A ragdoll instance keeps pointers to the rigid bodies and constraintsthat make up a ragdoll; it also keeps a pointer to an **hkaSkeleton** that describes the structure of this ragdoll.

Having all this information allows the ragdoll instance to provide the following functionality:

- Access to rigid bodies and constraints representing each bone / joint
- Access to the associated skeleton and its hierarchy
- Retrieving the current pose of the ragdoll in the physical world
- Setting the pose of the ragdoll in the physical world
- Clone the ragdoll
- Add and remove the ragdoll from the physical world



5.1.3.1 Creating a Ragdoll Instance

You can create ragdoll instance during tool chain operations using the Create Ragdoll filter. In order to setup a ragdoll instance at run-time, you need the following components:

- A set of rigid bodies and constraints, representing the bones and the joints of the ragdoll. As we'll see now, these rigid bodies and constraints have to be properly set up.
- A skeleton (**hkaSkeleton**), containing as many bones as rigid bodies in the ragdoll

These information (an array of rigid bodies, an array of constraints and a pointer to a skeleton) is passed to constructor of an `hkaRagdollInstance`.

```
hkaRagdollInstance ( const hkArray<hkpRigidBody*>& rigidBodies, const hkArray<hkpConstraintInstance*>&
  constraints, const hkaSkeleton* skeleton );
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Rigid Bodies and Constraints

In order for the hkaRagdollInstance object to optimally work with the rigid bodies and constraints at runtime, it requires that the array of rigid bodies and constraints have the following properties:

- The rigid bodies need to be properly ordered, parent first. That means that the first rigid body has to be the root rigid body (pelvis usually).
- If there is n rigid bodies, there has to be n-1 constraints (connecting those rigid bodies). Also, constraints have to be ordered so the child rigid body (rigid body A) of constraint i is rigid body $i+1$.
- The parentage of constraints need to be correct. Rigid body A in a constraint must always be the "child" rigid body, while rigid body B must always be the "parent" rigid body.
- The local transformation of the rigid bodies must match the pivot point of the bone they represent.
- Since the constraints represent joints, the pivot point of a constraint should match the pivot point of the joint (the pivot point of the child bone / child rigid body).

These requirements ensure that operations on ragdolls can be done, and can be done efficiently. Fortunately, you can use **hkaRagdollUtils::reorderAndAlignForRagdoll()** in order to ensure all the above.

```
static hkResult hkaRagdollUtils::reorderAndAlignForRagdoll (hkArray<hkpRigidBody*> &rigidBodiesInOut,
  hkArray<hkpConstraintInstance*> &constraintsInOut);
```

The only requirements on the input physics system in **reorderAndAlignForRagdoll()** are:

- There must be n rigid bodies and n-1 constraints
- Constraints must be properly parented
- Constraints' pivots must be properly placed (at the joint)

If the above is satisfied, the **reorderAndAlignForRagdoll()** method will reorder the rigid bodies and constraints and will also modify the rigid bodies so their local transform matches the pivot of the bones (the pivot of the associated constraint).

The Ragdoll Skeleton

In order to create a ragdoll instance, you also need to provide a pointer to an **hkaSkeleton** associated with the ragdoll. This skeleton is used at run time in order to access hierarchy information efficiently, as well as to store some per-bone information (name, initial pose and skeletal constraints).

It is easy to create an skeleton from the rigid bodies and constraints. First of all, you need to ensure that the arrays of rigid bodies and constraints satisfy the requirements we described in the previous section. If you have that, you can then use the utility **hkaRagdollUtils::constructSkeletonForRagdoll()** to create an appropriate skeleton.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
static hkaSkeleton* hkaRagdollUtils::constructSkeletonForRagdoll (const hkArray<hkpRigidBody*> &
  rigidBodies, const hkArray<hkpConstraintInstance*> &constraints);
static void hkaRagdollUtils::destroySkeleton (hkaSkeleton* skeleton);
```

The names of the bones stored in this new skeleton will be copied from the names of the rigid bodies; so it is important that your rigid bodies are properly named before calling this method.

Also, the initial pose of the skeleton is taken from the current position of the rigid bodies at the time of the call. So, in general, you should create this skeleton at setup time, with the rigid bodies aligned in a T-Pose.

Finally, it is important to notice that the **constructSkeletonForRagdoll()** method will allocate some memory in order to hold the array of bones as well as the bone names. You can destroy this skeleton as well as this associated memory by calling the method **hkaRagdollUtils::destroySkeleton()** .

5.1.3.2 Getting and Setting a pose

The hkaRagdollInstance class also allows you to set and retrieve the current pose of a ragdoll, using the following methods:

```
void getPoseModelSpace (hkQsTransform* poseModelSpaceOut, const hkQsTransform& worldFromModel) const;
void setPoseModelSpace (const hkQsTransform* poseModelSpaceIn, const hkQsTransform& worldFromModel);
```

The poses are specified in model space, i.e., as a set of *model-from-bone* transforms. You also need to specify how this model space is related to the world space (since that's the space used by Havok Physics to place the rigid bodies), passing an extra `worldFromModel` transform. For more information about these spaces and what they represent, check the chapter in **Havok Animation** > **Animation Runtime** > **World, Model and Local Spaces** .

Note that setting the pose using the setPoseModelSpace call is intended to be used for initialization, but not to be called during simulation as performance can be impacted. To influence the pose of an hkaRagdollInstance during simulation, it is preferable to use the driveToPose() functionality of the hkaRagdollRigidBodyContoller, or call hkpKeyFrameUtility::applyHardKeyFrame on each rigid body, instead.

5.1.4 Mappers

A mapper is responsible for converting a pose from one skeleton to another. The run-time class that does this in particular is the **hkaSkeletonMapper** class. Mappers are described in detail in the Animation chapter in this manual. They play a really important role in ragdoll integration, so please read the documentation about Skeleton Mappers.

5.1.5 Ragdoll Controllers

Ragdoll controllers are responsible for driving a ragdoll instance to a desired pose. This can be achieved by different means (powered constraints, applying velocities to rigid bodies), so multiple implementations of ragdoll controllers are possible.

Note:
In Havok 3.1, the *joint* and *bone* controllers have been renamed to *powered constraint* and *rigid body* controllers. They also no longer inherit from a common class, so the term "ragdoll controller" specifies the common application of both controllers, rather than refer to an actual abstract class.

There are two ragdoll controllers provided with the Havok SDK:

- *The Ragdoll Rigid Body Controller* : Operates on the rigid bodies (bones)
- *The Ragdoll Powered Constraint Controller* : Operates on powered constraints (joints)

5.1.5.1 The Ragdoll Rigid Body Controller

The ragdoll rigid body controller (**hkaRagdollRigidBodyController**) tries to drive the rigid bodies of the ragdoll to the desired pose by setting their linear and angular velocities.

This controller wraps the functionality in the `hkKeyframeHierarchyUtility`, and specializes it for use with ragdolls. You can create a ragdoll rigid body controller by either associating it with a ragdoll instance, or by explicity passing the rigid bodies and hierarchy involved:

```
hkaRagdollRigidBodyController ( hkaRagdollInstance* ragdoll );
hkaRagdollRigidBodyController ( int numBodies, hkpRigidBody*const* bodies, hkInt16* parentIndices );
```

The setup for the controller is available through a public member, `m_controlData`. This is an instance of `hkaKeyFrameHierarchyUtility::ControlData`, and stores information about gains and ranges used by the controller (follow the link to the class for details on its members)

```
hkaKeyFrameHierarchyUtility::ControlData m_controlData;
```

While driving the ragdoll using this controller, you should use the driveToPose() method:

```
void driveToPose ( hkReal deltaTime, const hkQsTransform* poseLocalSpace, const hkQsTransform&
  worldFromModel, hkaKeyFrameHierarchyUtility::Output* stressOut );
```

The call requires a timestep since the forces and velocities applied to the rigid bodies are time-step dependant. `driveToPose()` returns information about the error (stress) on the controller for each rigid body through the output parameter `stressOut`.

You will need to call `driveToPose()` every step of the simulation where you want the ragdoll to be driven.

You can stop driving the ragdoll by just not calling `driveToPose()`. If, after that, you want to start driving the ragdoll again (or if you move ragdoll's position or velocity outside the controller), you'll need to call `reinitialize()` to let the controller now (since it keeps state based on the last step).

```
void reinitialize();
```

5.1.5.2 The Ragdoll Powered Constraint Controller

This controller is implemented through a static utility class (**hkaRagdollPoweredConstraintController**). The powered constraint controller tries to drive the rigid bodies to the desired pose by using the motors of the powered constraints that attach those rigid bodies.

Therefore, this controller it's only useful if the constraints in the ragdoll are powered constraints (powered hinges or powered ragdoll constraints). Fortunately, you can convert a hinge or ragdoll constraint to its powered counterpart by using the utility method **hkpConstraintUtils::convertToPowered()** (in the `hkutilities` library).

```
static hkpConstraintInstance* hkpConstraintUtils::convertToPowered (const hkpConstraintInstance*
  originalConstraint, hkpConstraintMotor* constraintMotor);
```

The `hkaRagdollPoweredConstraintController` class provides the following static methods:

```
static void driveToPose( hkaRagdollInstance* ragdoll, const hkQsTransform* poseLocalSpace );
static void startMotors( hkaRagdollInstance* ragdoll );
static void stopMotors( hkaRagdollInstance* ragdoll );
```

The main method is `driveToPose()`. It takes a ragdoll instance and a pose, specified in local space. This method will go through the different joints and set the target of each motor to the desired pose.

The other two utility methods, `startMotors()` and `stopMotors()`, will go through the different motors associated with the joints (through the powered constraints) and enable / disable them accordingly.

The ragdoll powered constraint controller will generally work with all kinds of motors, yet the `hkpPositionConstraintMotor` is usually preferred because of its increased stiffness. Because each motor implementation has different values that control strength, damping, etc.. if you want to change the behaviour of the motors (for example, weaken or strengthen them) you will have to do so in your application code (where you can correctly cast each motor to its proper type).

5.1.5.3 What Controller to Use

As always, the choice of the ragdoll controller depends on the actual application. In many cases, you may want to use both types at different stages of the simulation. The following table gives a comparison of both controllers in different areas.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Rigid Body Controller	Powered Constraint Controller
Works on any ragdoll setup, regardless of the constraints used.	Requires the use of powered constraints. Powered constraints are slower to simulate than normal constraints.
Keeps state between calls	Stateless (access through static methods)
Time-step dependant	Independant of time step
Needs to be applied explicitely every step (through `driveToPose()`)	Can be switched on/off, but it's implicitely applied every step by the physics. So, once on, it's always applied. `driveToPose()` just changes the current target pose
Works in world space (bone transforms)	Works in local space (joints)
Drives all bones, including the root.	It doesn't drive the root
Since it works by setting velocities in world space, it will drive the ragdoll to the desired world space pose - forces applied to it will have little effect.	Since it works in local space (and doesn't drive the root), it gives more realistic results in many situations.For example, you can apply impulses and forces to the ragdoll (caused by, let's say, an explosion) - the ragdoll joint controller will still drive the joints while the ragdoll flies through the air.

Table 5.1: Using Bone and Joint Controllers

To see these controllers used in context see the Animation Examples and Use Cases section

5.1.6 The Detect Ragdoll Penetration Utility

Havok integration of physics and animation into the animation pipeline allows a continous change between an animation-driven character to a fully dynamic character (ragdoll). In many of these "state-change" situations it's very useful to know the penetration status of rigid bodies/bones. The detect ragdoll penetration utility (**hkaDetectRagdollPenetration**) checks for these penetrations and returns the current status of all ragdoll's bones for an specific input pose.

5.1.6.1 Overview

There are situations where you may want to exactly detect the penetration of ragdoll into the landscape. For example, your character is walking against the wall and suddenly he is shot by enemy. In the walking state character ragdoll is driven by an animation and the motion of each rigid bodies is keyframed, but after the shot the character state is swapped to dead (ragdoll) and the character is fully driven by dynamics.

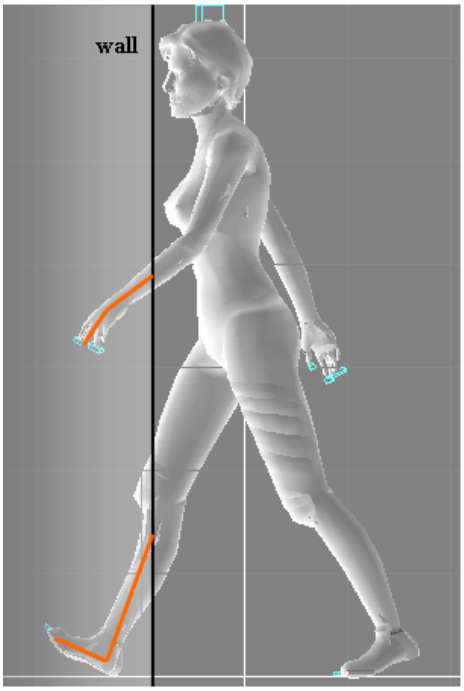


Figure 5.1: Character's limbs penetrate to landscape during keyframed to dynamics state change

If some of the rigid bodies (usually limbs) are in the "dark side" of the landscape at this moment, problematic situations can occur, which if they are not correctly handled may cause improper behaviour as collision detection and constraints may "fight" each other - causing the ragdoll to get "stuck" on the landscape, possibly jittering. A recovery method for ragdoll penetration is shown in *section 6.3*.

Given the fact that continuous physics guarantees, that if the ragdoll starts in a correct (not going through the landscape) pose, it should remain in a correct pose over time, i.e., it should not go through the landscape anytime, a good way to ensure proper behaviour is to follow the following process:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

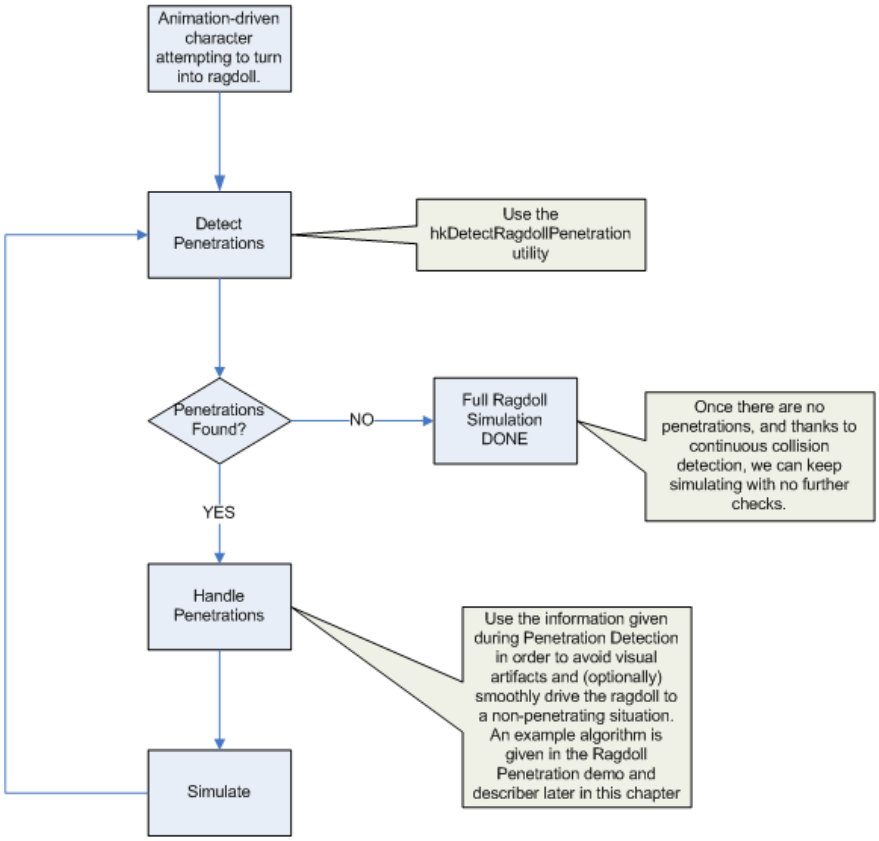


Figure 5.2: A process to handle ragdoll penetrations

Notice that the detection can also be used preemptively - for example, you can decide not to turn a character into a ragdoll if the penetration check fails.

5.1.6.2 The hkaDetectRagdollPenetration utility

hkaDetectRagdollPenetration utility (available in the `hkragdoll` library) detects the penetration of all bones of the ragdoll's skeleton with the landscape. The idea of detection is very simple. The detection routine performes a raycasting of all bones against the landscape until all bones are no penetrated. Since only the bone positions in space are required, this utility can be applied to an arbitrary skeletal pose, and not just to ragdolls. Each ray is defined by start and end point of the bone in world coordinates. The utility will associate one of three states to each bone: *non-penetrating*, *penetrating* or *descendant* of a penetrating bone. These three states describe the penetration of the actual pose much better as simple yes/no penetration state per bone.

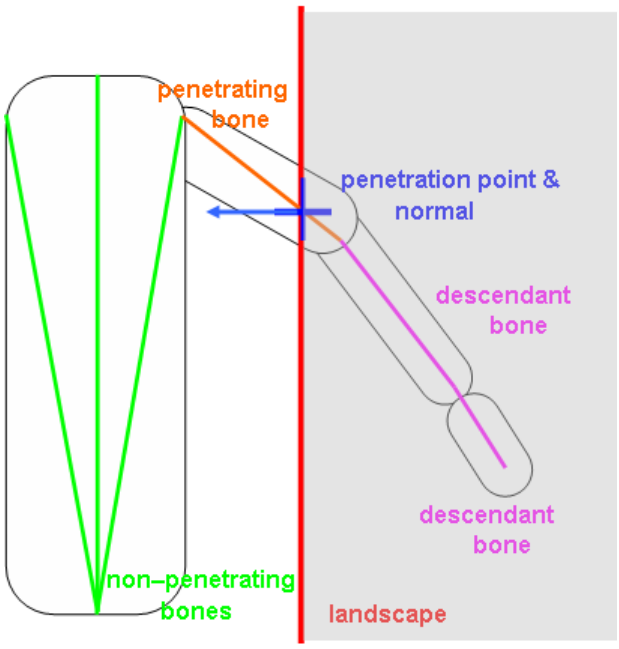


Figure 5.3: Status of particular bones during penetration

The utility is optimized for performance and stores an internal state - after the firtst check, it will only check (raycast) for bones that were previously penetrating. Since a state is kept, the utility needs to be instanciated through a constructor. The setup structure passed to the constuctor contains:

- A pointer to an skeleton (hkaSkeleton) : this would usually be the skeleton of the hkaRagdollInstance we are working with. But the utility will also work with any non-ragdoll skeleton (so you can use it to detect penetrations of any characters)
- A pointer to raycast interface (hkaRagdollRaycastInterface) : this interface will be used to perform the raycasts - you can use Havok's optimized raycasting system or your custom raycasting as well. The interface also gives you a chance to perform any setup on the raycast (for example, assigning particular collision filter information in order to select what objects should we raycast against).
- A pointer to an AABB phantom (hkpAabbPhantom), which should encapsulate the whole ragdoll. This phantom is used in order to optimize the raycasting when using Havok, and is passed to the raycast interface. The utility will also gradually modify the phantom as the pose changes and bones stop penetrating.

```
// setup DetectRagdollPenetration object
hkaDetectRagdollPenetration::Setup setup;

setup.m_ragdollSkeleton = m_ragdollInstance->getSkeleton();
setup.m_raycastInterface = m_ragdollRaycastInterface;
setup.m_ragdollPhantom = m_ragdollPhantom;

m_detectRagdollPenetration = new hkaDetectRagdollPenetration(setup);
```

The principal `detectPenetration()` method takes as the input an actual hkaPose *in world space* and performs raycasting of all bones with the landscape, using the phantom Aabb for optimization. The

detection routine fills an output structure, containing information for each penetrating bone as an array of `BonePenetration` structures. Each structure for the contains the begin and end indices of the bone, the penetration point in world coordinates and the normal of the penetrated surface.

```
// ModelPose is set directly to WorldPose used for penetration detection
hkaPose ragdollPose( m_ragdollInstance->getSkeleton() );
m_ragdollInstance->getPoseWorldSpace( ragdollPose.writeAccessPoseModelSpace().begin() );

m_detectRagdollPenetration->detectPenetration( ragdollPose, m_penetratedBones );
```

The penetration state of each bone is internally stored in the utility. As we mentioned before, each bonee can have three states: no penetrating (HK_NOP), penetrating (HK_YESP) or descendant of the penetrating bone (HK_YESP_DESCENDANT). The class provides the method `getBoneStatusArray()` to get a reference to the bone state array.

```
// get boneStatusArray from hkaDetectRagdollPenetration object
const hkArray<hkaDetectRagdollPenetration::BonePenetrationStatus>& boneStatusArray =
  m_detectRagdollPenetration->getBoneStatusArray();
```

5.1.6.3 An example of Ragdoll Penetration Recovery

As described above in overview, once you detect some bone penetrations, you need an algorithm to handle them in order to avoid/reduce any visual artifacts. The Ragdoll Penetration demo shows a very effective algorithm to recover from ragdoll penetrations and smoothly drive the ragdoll "out" of the landscape.

The algorithm operates based on the results of the hkaDetectRagdollPenetration utility, acting in different ways depending on the state of each bone:

- if HK_NOP (not penetrating bone/rigid body): enable collision with the landscape (if it was disabled)
- if HK_YESP_DESCENDANT (descendat of the penetrating bone/rigid body): dissable collision with the landscape.
- if HK_YESP (penetrating bone/rigid body) : dissable collision with the landscape and apply correction impulse. The direction of the impulse is the same as a penetration point normal. The magnitude of the impulse is calculated by the following formula:

  *Impulse = m * (vd - vc*cos(alpha))*

  where m is the mass of penetrated rigid body, vd is the desired velocity (constant for tuning), vc is the current velocity magnitude in penetration point and *alpha* is the angle between the current velocity and the normal of penetration point.

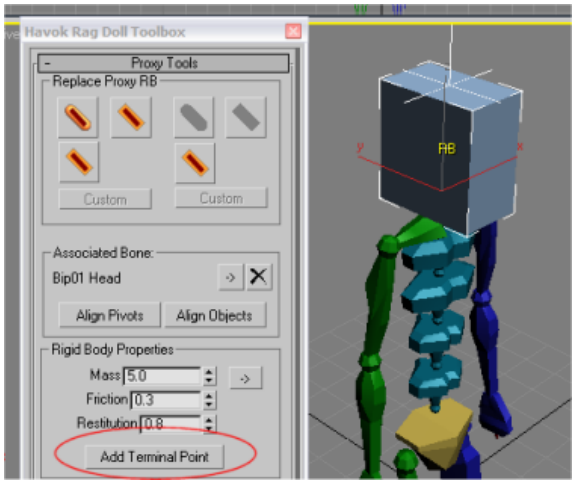
To see these utility used in real application see the ragdoll penetration demo in Animation demos.

5.1.6.4 Toolchain implications

Since penetration detection is done through raycasting from each bone to their children, one of the consequences is that leaf bones of a skeleton cannot be checked for penetrations. This raises a problems with ragdoll skeletons, where leaf bones were commonly "normal" bones (rigid bodies) with volume and could therefore intersect the landscape.

Since Havok 4.6 the `hkaRagdollInstance` class allows for its associated `hkaSkeleton` to have more bones than rigid bodies, and therefore allows for extra terminal bones to be added to the ragdoll skeleton, specifying the dimension of the leaf rigid bodies (for example, the tip of the feet, hands and head). This means that an `hkaRagdollInstance` may have n rigid bodies, but the associated skeleton may have m (where $m>=n$) bones.

The Havok Content Tools have been adapted to facilitate the creation of ragdoll terminal bones:



Caution:

When creating ragdolls with terminal bones, make sure you create the ragdoll skeleton using the Create Skeleton filter and then use it for the **Use Skeleton** option in the Create Ragdoll filter. Using the **Automatic** mode in the Create Ragdoll filter would ignore any terminal bones (as it builds the skeleton from the rigid bodies, and not vice-versa).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Chapter 6

Havok Content Tools

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.1 Introduction and Architecture

6.1.1 Introduction

A critical component of any game production pipeline involves extracting information from the content creation tools, serializing this to an extensible stream format and efficiently loading, processing, storing and finally accessing these assets at runtime. The Havok Content Tools and Havok's serialization framework provide an extensive and extensible set of tools for asset creation, export, processing and loading. This document focuses on asset creation, export and processing. For details on asset loading (within a game), please consult the documentation for the Havok Serialization Framework.

This document first presents an introduction to the core elements of the Havok Content Tools:

- Introduction and Architecture : This section introduces the Havok Content Tools and gives an overview of how they operate and fit into an asset pipeline.
- The Havok Filter Pipeline : Presents the tools available for asset processing and serialization and how they can be combined to achieve powerful results.

Several sections are then presented covering the content creation and export tools for each supported modeler:

- 3ds Max Tools : Presents the set of Havok Content Tools for 3ds Max, in particular the 3ds Max scene exporter and the 3ds Max tools for physics content creation. Includes tutorials for 3ds Max users.
- Maya Tools : Presents the set of Havok Content Tools for Maya, in particular the Maya scene exporter and the Maya tools for physics content creation. Includes tutorials for Maya users.
- XSI Tools : Presents the set of Havok Content Tools for XSI, in particular the XSI scene exporter and the XSI tools for physics content creation. Includes tutorials for XSI users.
- Common Concepts : All supported modeling packages share some concepts related to physics and animation setup, which are described in this section.

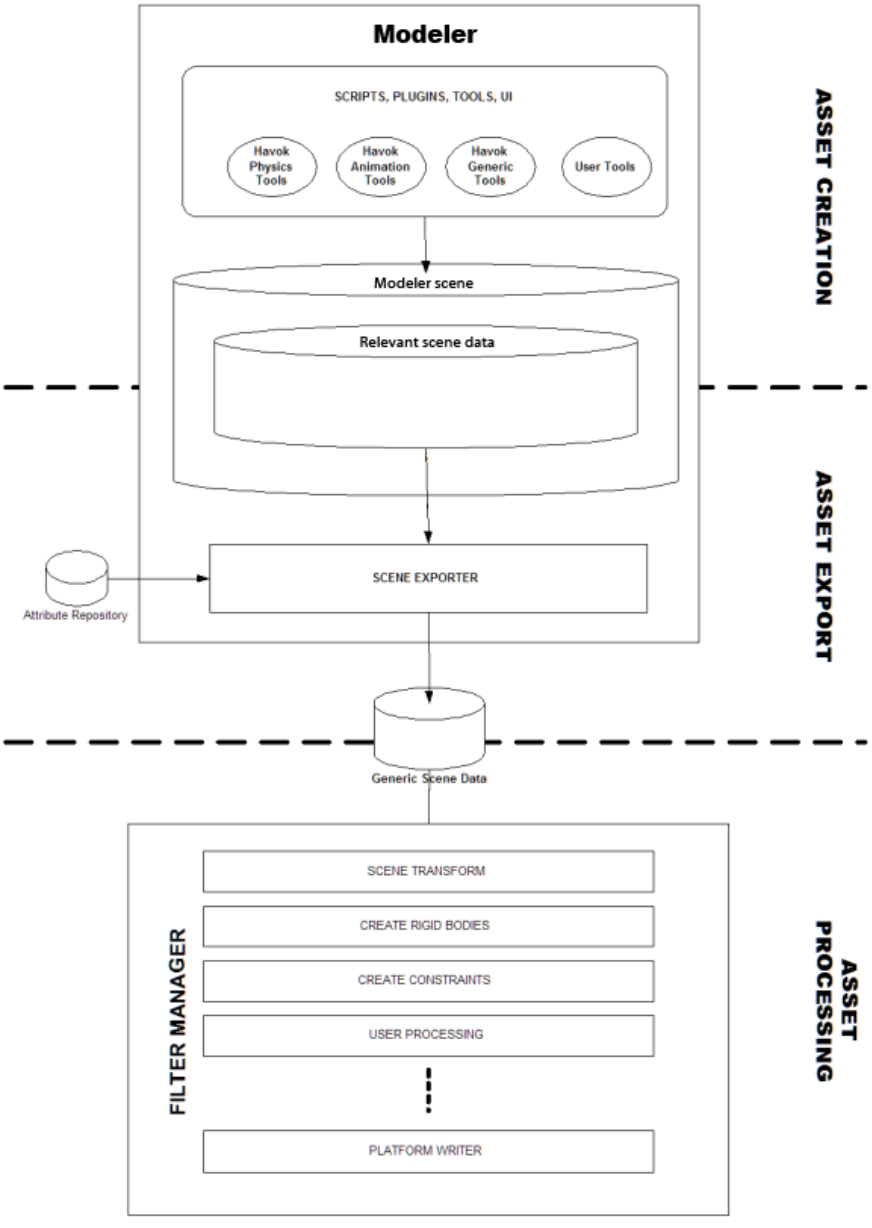
Finally, advanced topics are presented covering customization and troubleshooting of the Havok Content Tools:

- Integrating with the Havok Content Tools : A detailed guide on the interfaces exposed by each component of the Havok Content Tools and how best to extend and integrate with them.
- Troubleshooting : A list of common problems and gotchas and how to solve them.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.1.2 Architecture

The following diagram outlines the architecture behind the Havok Content Tools:



The tools can be divided into three functional categories: Asset Creation/Setup, Asset Export and Asset Processing.

6.1.2.1 Asset Creation Tools

These are tools created specifically for each modeler and are typically used by artists, animators or designers. Tools provided in this area therefore focus on usability, flexibility and having a good workflow

integrated with the rest of the modeler's tools. These tools do not have any dependencies with Havok or external libraries, they simply set up data within the modeler in such a way as it will be interpreted in a useful way by a Havok Scene Exporter.

Examples of Asset Creation Tools are:

- Havok Physics Tools
  - Custom nodes, modifiers, parameter sets and attributes that can be added to the scene or to existing nodes in order to assign physical properties to objects.
  - Tools to manipulate and visualize otherwise complex concepts like constraint limits and spaces.
  - Higher level tools such as those to create rigid bodies for animation bones, associate them and align them.
- Havok Animation Tools
  - Currently, all supported modelers offer enough built-in support for animation concepts so we do not ship any custom animation extensions. Future examples could be tools to add: annotations to tracks; custom IK information to bones; etc...
- Havok Generic Tools
  - Tools which provide an interface to the other Havok tools, by providing toolbars, menus, etc...
- User Tools
  - Since asset creation tools only interface with the scene and do not have any dependencies with Havok or external libraries, it is very easy for users to create custom tools that sit in this area. For example, users could provide their own scripts to: automate repetitive tasks; add game-specific data; etc..

The various content creation tools provided by Havok are described in detail in the 3ds Max Tools, Maya Tools and XSI Tools sections of this manual.

6.1.2.2 Asset Export Tools

The responsibility of the Havok Scene Exporters (one provided for each modeler) is to navigate the scene and extract relevant scene data, which is then converted into a generic (modeler-independant) Havok scene data format. This scene data (and how it relates to the actual modeler scene) is the key link between the asset creation, export and processing tools. In general, the scene data contains:

- The scene hierarchy (of nodes)
- The animation of each node's transform
- The meshes, lights, cameras that may be associated with each node
- The materials and textures that may be associated with each mesh
- Extra attributes that may be associated with each node (possibly animated)

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The Havok Scene Exporters are described in detail in the 3ds Max Tools, Maya Tools and XSI Tools sections of this manual.

The Integrating with the Havok Content Tools section of this manual gives more details about the format of this generic scene data.

6.1.2.3 Asset Processing Tools

The output of a Havok Scene Exporter is passed as the input to the asset processing tools. Since this input is generic scene data, the asset processing can operate in a totally modeler-independant fashion. Once an asset processing tool (a *filter*, as described later on) has been written, it can operate regardless of whichever modeler used to generate the original content.

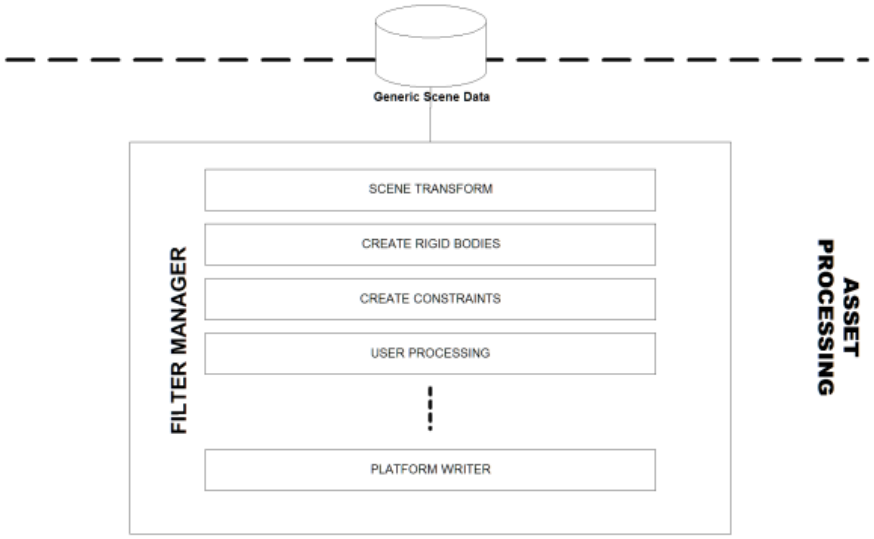
By combining these asset processing tools in various ways, powerful and varied results can be achieved.

The following section, The Havok Filter Pipeline, describes the individual processing tools in detail.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.2 The Havok Filter Pipeline

6.2.1 The Filter Manager



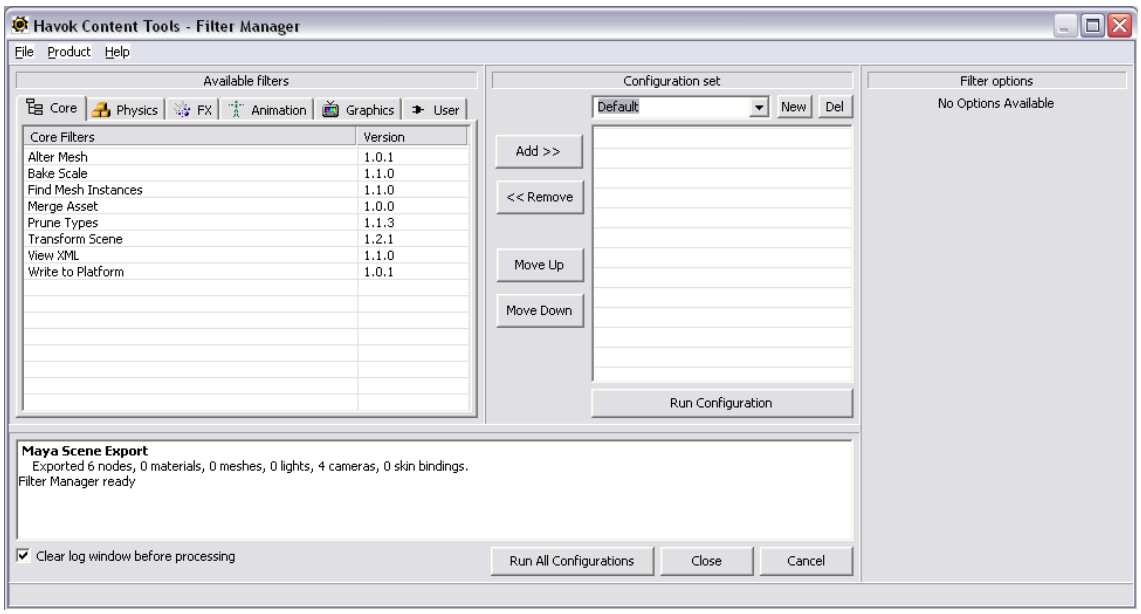
Asset processing in the Havok Content Tools is organized as a sequential combination of operators, where the output of each operator is taken as the input of the next one. These individual operators are known as *filter*'s in the Havok Filter Pipeline. There are multiple types of filter: some modify objects; some create new objects based on existing ones; some delete objects. Filters can also perform operations which have no affect on the content, such as previewing the scene or writing it to a file. Havok provides a large set of filters, which are individually described later in this chapter. One of the advantages of this scheme is that the small granularity and the loose coupling between filters makes them particularly easy to write and extend. The Writing Your Own Filter section at the end of this document gives more details on how to do so.

Before looking at the individual filters, lets take a look at the *filter manager*. This is the application which allows the user to select, organize and modify the options of one or more sets of filters. The filter manager is therefore the "control center" from which asset processing is managed.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.2.1.1 The Filter Manager Interface

The following interface is presented to the user after running any of the Havok Scene Exporters (or from the standalone filter manager, described in the following section):



The interface is divided into several panes:

- **Available Filters** : Shows a list of the available filters, grouped into the following categories :
  - **Core** : Filters that operate on generic scene objects (are not related to either physics or animation).
  - **Physics** : Filters that create / operate on physics objects (rigid bodies, constraints, etc).
  - **Animation** : Filters that create / operate on animation objects (skeletons, skin, motion, etc).
  - **Graphics** : Filters related to graphics elements (materials, etc). Also contains the scene previewer.
  - **User** : This category is left open for custom filters written by users.
- **Configuration Set** : Lists the set of configurations and the filter setup for the currently selected configuration. Filters are executed in the order that they appear in the list.
- **Filter Options** : Many filters have options whch affect the filter's behaviour. Whenever a filter is selected in the filter configuration list, any options it may have shown in this pane.
- *Log Window*: This is used to report messages, warnings, etc. while setting up the filter configurations and while processing. Any serious messages are reported in an orange/red color - if any such messages occur they should be investigated further.

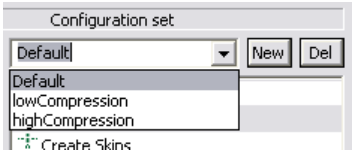
There are also some buttons in the filter manager UI:

- **Add >>** : Adds the filter currently selected in the *Available Filters* list to the current filters list. Double clicking on a filter performs the same operation.

- << **Remove** : Removes the current filter selected in the list. Pressing **DEL** performs the same operation.
- **Move Up** : Moves the currently selected filter in the list one slot up. Pressing the " **-** " key performs the same operation.
- **Move Down** : Moves the currently selected filter in the list one slot down. Presssing the " + " key performs the same operation.
- **Run Configuration** : Executes the current configuration.
- **Run All Configurations** : Executes all configurations consecutively.
- **Close** : Saves all changes to the configurations and closes the filter manager.
- **Cancel** : Closes the filter manager, prompting for confirmation to save any changes made to the configuration(s). Pressing **ESCAPE** , or closing the window performs the same operation.

Configuration Sets

A set of filters in a specific order and with specific options is known as a *configuration* in the Havok Filter Pipeline. It is possible to have multiple filter configurations associated with a single asset. The *Configuration Set* pane contains a dropdown list showing the active configuration, of which there is always at least one by default:



Each configuration contains its own set of filters, and operates on its own copy of the asset data. This allows the same original asset data to be processed in different ways without having to repeatedly change the filters being used. For example, multiple configurations can be used to process animation data using different amounts or types of compression.

To change the active configuration, simply select another from the dropdown list. To rename a configuration, edit it's name using the box. To delete a configuration, press the *Del* button. To add a new configuration, press the *New* button.

Whenever a new configuration is added, it becomes the active configuration, and begins as a copy the previous configuration.

Saving & Loading Configuration Sets

When invoked from a modeler, the filter manager configurations are automatically saved as part of the modeler asset file. In addition, configuration sets can be saved and loaded into a standalone file (also called an "options file", or .hko file), using the **File** > **Save Configuration Set** and **File** > **Load Configuration Set** menu options.

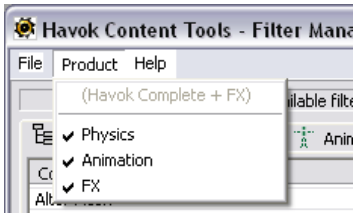
This allows the same filter setups to be used again and again, or tweaked as desired and resaved. They can even be used to override or upgrade the filters saved with individual assets. See the section on Overriding and Upgrading Filter Settings for more information on how to do this.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

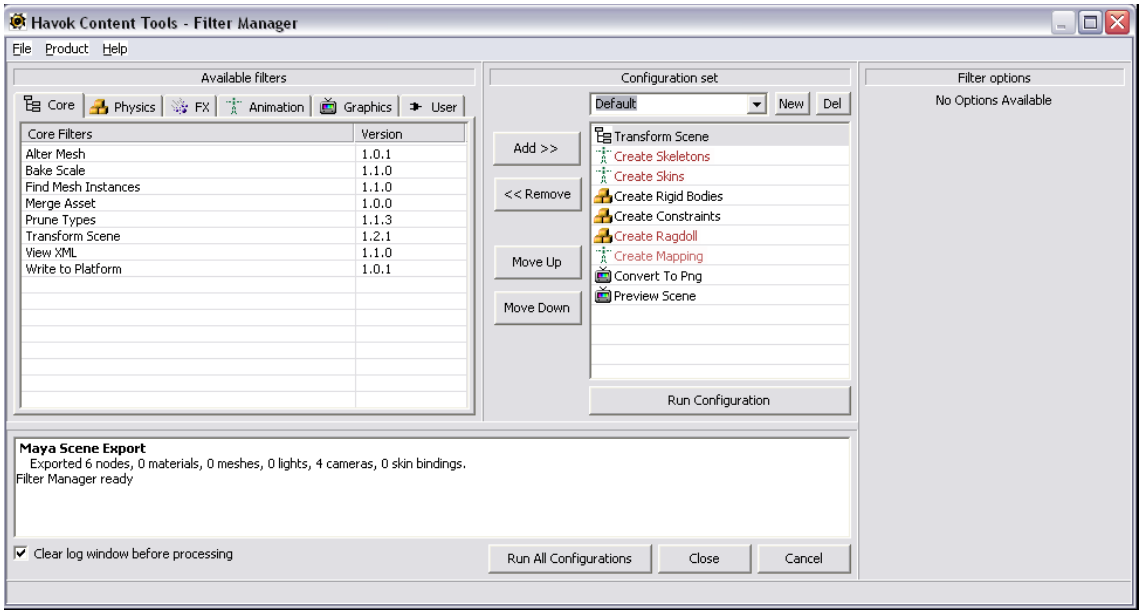
Product Selection

Filters produce output that may depend on different Havok SDK products to allow them to be loaded at run-time. For example: Scene Transform works for all Havok products; Create Rigid Bodies generates objects which require the Havok Physics/Complete SDK for loading at run-time; Create Rag Doll filter generates objects which require the Havok Complete (Physics+Animation) SDK for loading at run-time.

All filters are always available for use within the filter pipeline, regardless of the run-time product which your company has licensed. Therefore, it is useful to be aware of which filters may be unsupported by your Havok product. You can select your specific Havok product by using the **Product** menu in the filter manager:



This product selection is stored in the registry and therefore is a user-setting - its value remains between each session of the filter manager. Filters which are not supported at run-time by the selected product will appear in red in the filter manager, indicating that they should not be used:



On execution of any setup which includes run-time unsupported filters, a warning will appear in the log window for each unsupported filter.

6.2.1.2 The Standalone Filter Manager

The filter manager is implemented as a DLL - each modeler invokes the same filter manager through the DLL interface. It is also possible to invoke the filter manager through a standalone Havok application, known as the the *Standalone Filter Manager*.

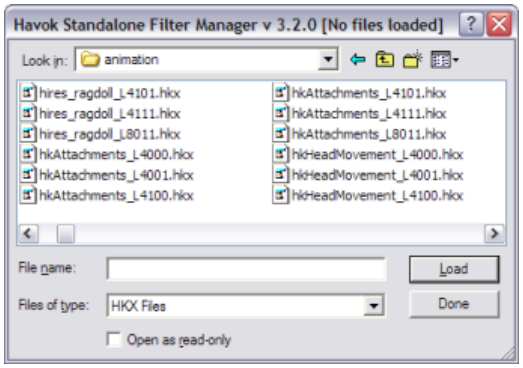
The input data for this application is taken from one or multiple havok serialized (`.hkx`) files. These files are usually the output of some processing previously performed from a filter manager invoked from a modeler.

The standalone filter manager has many possible uses:

- It can be used as a "preview" tool (by loading an asset and just using the Preview Scene filter).
- It can be used to batch process assets without the need to use a modeler.

Using the Standalone Filter Manager in Interactive Mode

To use the Standalone Filter Manager in interactive mode simply launch the associated executable (a shortcut is placed in the Start menu by the Havok Content Tools installer). A splash screen will appear for a short period and you will be presented with a file open dialog box.



Select a single file or multiple files to load and press the **Load** button. Each time you press the **Load** button the title in the dialog is updated to reflect the number of files selected for loading. Once you have successfully selected one or more files press the **Done** button to complete the loading process.

The files are loaded in turn using our standard serialization. If any file fails to load a warning is printed in the console window.

Successfully loaded files are then merged together into a single root level container and this is passed to the filter manager. The filter manager should now appear and interactively allow you to adjust the current filter stack.

Default Options

When `.hkx` files are loaded into the Standalone Filter Manager there are no filter configurations available, so a default configuration set is used instead. This configuration, called 'HKX Preview', contains a single Preview Scene filter, allowing the user to view to `.hkx` file contents.

This default setup is stored in an XML file called `"defaults.hko"`, placed in the same folder as the Standalone Filter Manager. This allows the default options to be modified or replaced as necessary. See the HKO Files section for more information on creating or modifying options.

Once loaded by the Standalone Filter Manager, the filter configurations can be edited and/or processed as per usual.

Command Line Arguments

The Standalone Filter Manager can also be launched in command line mode. It's usage is as follows:

```
USAGE: hkStandaloneFilterManager.exe [-p assetPath -s settingFilename.hko] file1.hkx [file2.hkx ..
  fileN.hkx]
```

This launches the Standalone Filter Manager in command line mode. The specified files (`file1..N`) are loaded and merged before being passed to the filter manager. The filter manager is automatically invoked and processing begins in batch mode. If you wish to specify files on the command line and still run the previewer in interactive mode, you can use the **-i** flag to force interactive mode.

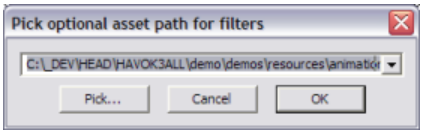
The optional **-s** flag specifies a filter set (i.e. a `.hko` file) to use. `.hko` files are usually created by launching the tool interactively, creating one or more filter stacks and choosing the 'save filter set' option. If this flag is omitted the default options file will be used.

The optional **-p** flag specifies the asset path, described further in the following section.

The main reason for using the command line mode is to efficiently process assets and create `.hkx` files. To this end, the filter set should contain a Platform Writer filter.

Asset Paths

When `.hkx` files are created using relative paths in the filters, they are relative to the asset path. Thus if you load several files you have several asset paths to choose from, or if you have an `.hkx` file that should not use its asset path for this run, then you can specify your path. The asset path dialog pops up after you have chosen your file set with the list of asset paths contained in those files. You can choose one of them, edit them, or just browse for one if you can't remember it. If you choose **Cancel** , the pipeline will run with a empty asset path, so any relative paths will be in relation to the current working directory.



Examples

The Standalone Filter Manager was originally designed to allow you to view `.hkx` files without having to resort to loading the original art asset in the modeler. However, since it uses the full power of the filter manager it can also be used directly in the tool chain as an asset processing tool. Some use cases would be:

- Use a View Xml filter to verify that the expected file elements are present in the asset.

- Load separate rigs, animation and skin data to verify they work together.
- Use any of the motion extraction or compression filters on raw animation to test their results in isolation.
- Batch process assets to transform a scene from a left handed to right handed system.
- Batch process XML production assets to produce final game assets in binary platform specific formats.

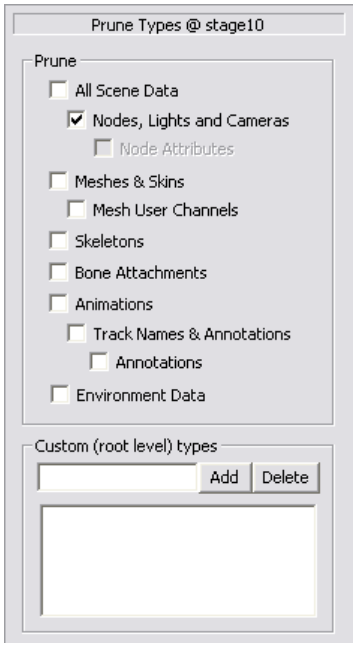
6.2.2 Core Filters

Core fiters are those which generally modify/examine basic scene data. They can always be used regardless of whichever Havok Product is licensed for use at run-time.

6.2.2.1 Prune Types

Components Required: Common

This filter is used to remove subsets of data by type, so they don't get exported, previewed, etc... It is useful to reduce the size of exported HKX files by removing unnecessary object types.



Here you can select some generic types of data to be removed from the scene:

- **All Scene Data** : Fully removes the `hkxScene` object
- **Nodes, Lights and Cameras** : Removes nodes, lights, cameras, materials and textures, retaining some basic scene information

- **Node Attributes** : It removes all attributes associated to nodes (hkxNode)
- **Meshes & Skins** : Removes mesh data (meshes and skin bindings)
- **Mesh User Channels** : Removes any vertex / triangle selection data associated to meshes.
- **Skeletons** : Removes any skeletons (`hkaSkeleton` objects)
- **Bone Attachments** : Remove any bone attachments (hkaBoneAttachment objects)
- **Animations** : Removes any animations (`hkaSkeletalAnimation` objects)
- **Track Names & Annotations** : Removes animation track data (`hkaAnnotationTrack` objects, containing track names and annotations) from animations.

  Warning:
  Do not check this option if you plan to bind animations to skeletons at run-time (since track names are required to do so).

- **Annotations** : Removes annotations (`hkaAnnotationTrack` objects) from animation tracks (but keeps the track names)
- *Environment Data*: Removes any `hkxEnvironment` object. Note that these objects are ignored by the Platform Writer filter, so it is usually not necessary to prune them.

The **Custom** (root level) types list allows you to add a list of custom type names (like `hkpPhysicsData` or `hkaRagdollInstance`). The filter will remove any objects of the given types found inside the named variants at root level (check Scene Data for details)

A common usage of the Prune Types filter is to separate out static data which is common to a single character or scene from corresponding dynamic data. For a single character a user may wish to export many animations separately, without duplicating the skeleton (rig) data in each exported file. This can be easily accomplished by first creating a Prune Types filter which prunes all data, *except* the skeleton information, and exporting just the skeleton information to a common file; next each animation may be exported in turn by pruning everything *except* the animation data. The resulting files will be smaller than if the skeleton data were included in each exported animation.

6.2.2.2 Merge Asset

Components Required: Common

This filter takes an HKX file, and merges its contents with the current asset. This is useful for example if you are exporting a rag doll and would like test it's behaviour in some already exported landscape.

The path to a HKX file which is to be merged should be specified in the filter options.

6.2.2.3 Bake Scale

Components Required: Common

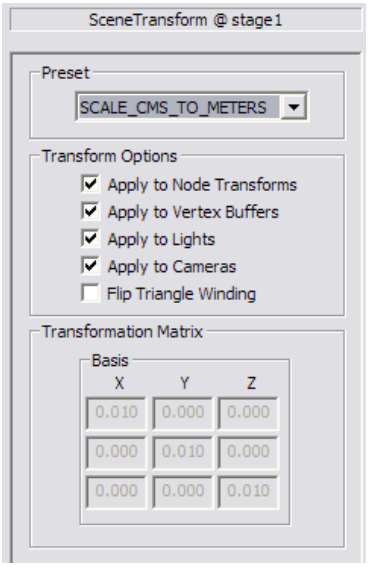
This filter iterates through each of the exported scene nodes, removing any scale found in the transform and applying it to the associated mesh/skin (if any). The scaling is removed by decomposing the original transform. This usually means that changes in scale are converted to changes in child node positions.

Use this filter when your exported assets should not include scaling. This filter can be useful in order to associate meshes to rigid bodies (since rigid bodies do not have scale). The animation system gracefully handles scaling.

6.2.2.4 Scene Transform

Components Required: Common

Often, the modeller and the runtime/renderer will use different conventions for scale and handedness. This filter transforms your asset to a different coordinate system, or to a different unit system, for example.



The filter allows you to choose the types of data you wish to operate on. Usually you will select all of the ' **Apply** ' options. In general when you perform a change in handedness (say using MIRRORING) you should also check the **Flip Triangle Winding** box.

The filter has a number of presets:

- MIRROR (X / Y / Z) Flips data
- SCALE_INCHES_TO_METRES (Usually used for Max, which models in inches by default)
- SCALE_CMS_TO_METRES (Usually used by Maya, which models in cms by default)

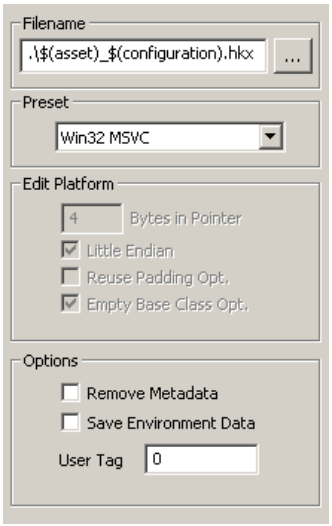
- SCALE_FEET_TO_METRES
- Custom: This option allows you to specify the complete transformation matrix to be applied to the scene. The matrix is pre-multiplied.

This filter will transform the scene and therefore affect the input of all filters proceeding filters in the pipeline, so it should generally appear as the first filter in a given configuration. For example, if this filter sits at the top of the stack then MIRROR_X will flip the nodes which will in turn ensure the Create Animations filter creates a mirrored animation.

6.2.2.5 Platform Writer

Components Required: Common

This filter allows you to write out the contents of the stream as either raw XML or a platform specific binary format. The target filename together with the target platform details are selectable in the options.



- *Filename* : The filename specifies where to dump the stream contents. Use the (...) button to open a standard file save dialog and specify the filename. NOTE: The path for the filename is always stored relative to the original asset. This means that if the asset is moved the output will be placed in a sensible relative location. This makes transferring assets between machine and batch processing far easier.

  You may use environment variables in this string. The strings '`\$(asset)`' and '`\$(configuration)`' are usually defined. Any instances of '`\$(asset)`' found in the filename will be replaced with the asset's name (the file extension is removed). Similarly, any instances of '`\$(configuration)`' will be replaced with the configuration's name. If an object was selected in the modeler before exporting, then a '`\$(selected)`' string is also available. Of course, you may define your own environment variables within the modeler. See the relevant modeler's export options for more.

  Environment variables are useful for creating HKX files when batch processing assets. Each platform writer belonging to a different asset / configuration can easily construct a write a unique filename using these variables.

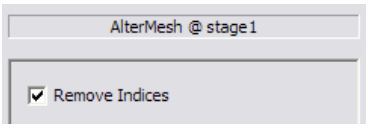
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- *Preset* : This allows you to specify either raw XML data or the binary format for a target platform. The binary target is both platform and compiler dependent (since some compilers do different types of optimization).
- *Remove Metadata* : This option will remove all hkClass data from the output file, reducing the file size. Notice that the type some variants will have to be interpreted by the user since the associated hkClass won't be available. Check the Serialization section in this manual for details.
- *Save Environment Data* : This option will include any defined environment variables in the output file. This is usually not required since the environment variables are generally used within the fitler pipeline only.
- *User Tag* : This value is an optional tag (an integer value) that can be associated with binary files (it is stored with the header of the binary file).

6.2.2.6 Alter Mesh

Components Required: Common

This filter removes triangle indices from meshes, and rewrites the vertex position data using a triplet for each triangle. This is useful for rendering engines and platforms that do not support triangle indices.

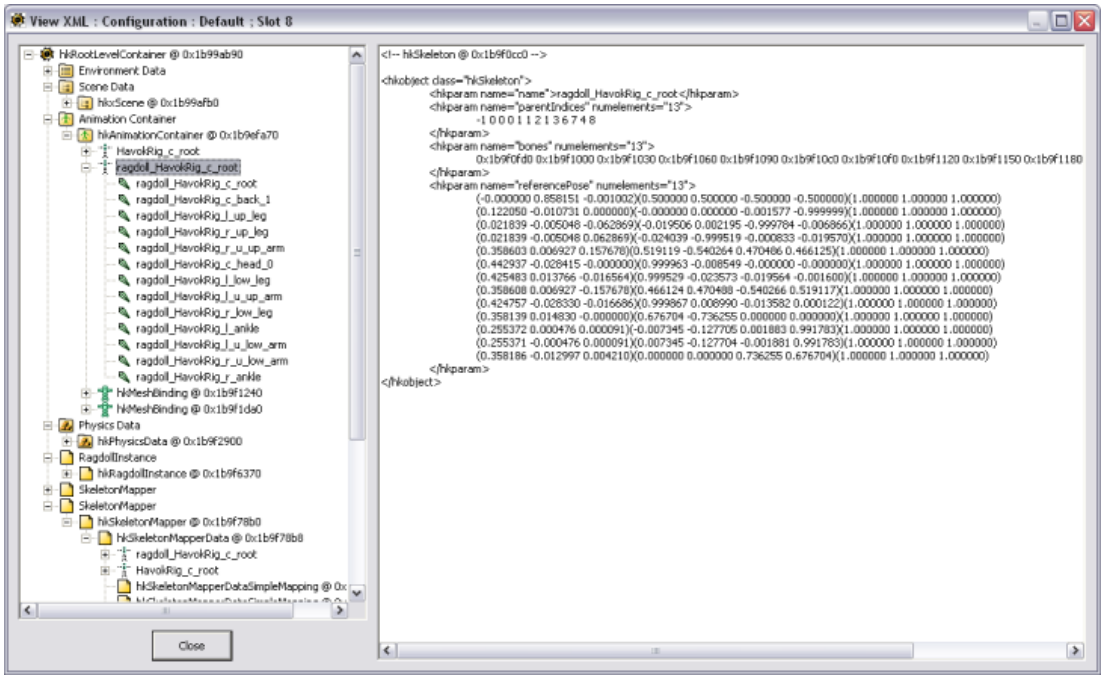


6.2.2.7 View XML

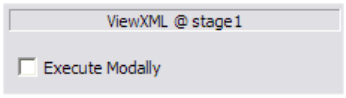
Components Required: Common

This filter, when executed, launches an interactive dialog which allows you to navigate the current data in the filter pipline. Selecting any item displays an XML description of its contents.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



This filter is very useful when you want to check the contents of a scene at any stage during processing. Since it uses Havok's serialization infrastructure, it can handle any data, even if it contains custom classes which have been created by custom filters.



By default the filter executes in a non-modal mode, that is, processing is not halted when the View XML window is opened. This allows you, for example, to have multiple View XML windows open at the same time.

Selecting the **Execute Modally** option will make the View XML operate modally; that is, processing will be halted until the View XML window is closed.

6.2.2.8 Find Mesh Instances

Components Required: Common

This filter examines all meshes in the scene data, looking for those which contain the same data. Any meshes which are considered to be duplicates of each other are replaced by a single instance of the mesh, and any pointers referencing the meshes are updated. By removing the duplicate mesh, the file size is kept down.

This is useful if you are using instancing/copying of objects to set up a scene in the modeller.

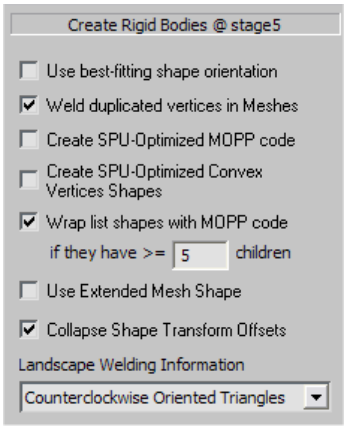
6.2.3 Physics Filters

Physics filters are those which create/modify Havok physics objects. Therefore these should generally only be used if your company has licensed a Havok Product which includes the physics component.

6.2.3.1 Create Rigid Bodies

Components Required: Havok Physics

This filter interprets rigid body data in the scene data (usually set up using the Havok Physics Tools within a modeler) and creates appropriate run-time rigid body and shape objects based on that data.



If the " **Use best-fitting shape orientation** " option is enabled, the filter will look for a best-fitting orientation for box, capsule and cylinder shapes rather than aligning those shapes to the transform of the scene node. Enable this option if you want the shapes to be based solely on the vertices of the mesh, regardless of the axes of the node transform. Normally this option is not required.

If the " **Weld duplicated vertices in Meshes** " option is enabled, the filter will look for duplicated vertices (and triangles) in meshes and ignore them when creating Havok shapes. This reduces the memory overhead and improves run-time performance of collision detection.

Using the " **Wrap list shapes with MOPP Code** " option will tell the filter to use Havok MOPP technology with list shapes with a high number of children (the threshold being specified by the user), which in many cases may speed up simulation.

When the " **Use Extended Mesh Shape** " option is on, the filter will use this mesh type (`hkpStorageExtendedMeshShape`) instead of the default (`hkpSimpleMeshShape`) mesh type. Extended mesh shapes are a bit more memory-intensive than normal mesh shapes, but they can also hold convex shapes alongside the landscape information. So, for compound rigid bodies, the filter will add any convex shapes belonging to the rigid body to the extended mesh shape. For more information about extended mesh shapes, please check the *Havok Physics SDK documentation*.

When the " **Collapse Shape Transform Offsets** " option is enabled, the filter will attempt to collapse intermediate transforms into leaf shapes (for the case of convex shapes and mesh shapes). If the option is disabled, the filter will keep shapes in their local space (the space of the node representing the shape). This facilitates instancing (sharing) of shapes. However, if the shape and its rigid body are in different

location in space, extra transform shapes are required to represent that offset. Enabling this option will remove those extra transform shapes by baking them into the convex / mesh shape. The option has no effect on bounding shapes (box, sphere, capsule and cylinder).

The " **Landscape Welding Information** " option will add extra information to Mesh Shapes regarding welding, improving collision detection and resolution accross triangle boundaries. In order to calculate this information, the filter needs to know what side of the triangles is considered to be *UP*. It will use the ordering of the triangle vertices in order to define this *UP* side. By default, most modellers use counter-clockwise orientation (where vertices are ordered in counter-clockwise order when viewed from the *UP* side).

The Create Rigid Bodies filter will also search for attribute groups named "`hkProperties`" associated with nodes flagged as rigid bodies, and convert any attributes in them to run-time `hkpProperty` objects associated with the run-time `hkpRigidBody` object. Since run-time properties are pairs of (ID, value) where both ID and value are integers, the filter will convert attributes (name, value) by hashing the name of the attribute (the hashing method is exposed in the `hkpProperty` class).
Check the Integration section for details on how to setup and export custom attributes associated with scene nodes.

6.2.3.2 Create Constraints

Components Required: Havok Physics

This filter interprets physical constraint data in the scene (usually set up using the Havok Physics Tools within a modeler) and creates the appropriate run-time constraint objects based on that data.

The rigid bodies connected by constraints must be created by the time this filter is processed, and therefore this filter needs to placed after a Create Rigid Bodies filter.

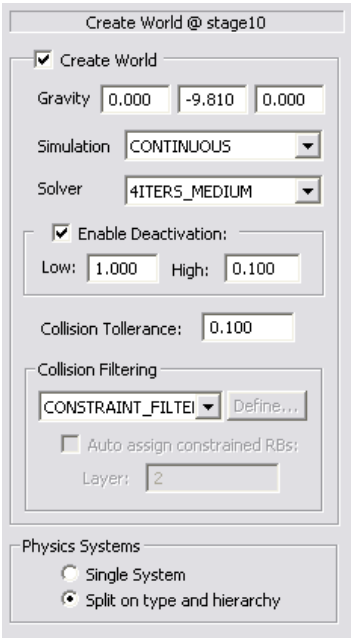
6.2.3.3 Create World

Components Required: Havok Physics

This filter can be used to set up information about the world (gravity, etc). It can also be used to group rigid bodies into physical systems and assign collision groups to them.

This filter is not essential when creating physical assets (a default world will be created in the Preview Scene filter if necessary). If this filter is used, it must be placed after any Create Rigid Bodies or Create Constraints filters in order to operate properly.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



- *Create World* : If this box is checked, a world setup will be added to the asset. Properties like gravity and other simulation settings can be specified in the options dialog. Please consult the Havok Dynamics chapter for details on these parameters and how they may affect the simulation.

  Among the world options that can be set up, the **Collision Filtering** section should be noted. Collision filters are used in order to determine which collisions between rigid bodies should be ignored. Many algorithms (collision filters) are provided in the Havok Physics SDK to do so. The options here allow you to choose between three options:

  – NO_FILTER: No collisions will be ignored
  – CONSTRAINT_FILTER: Collisions between constrained rigid bodies will be ignored
  – GROUP_FILTER: Collisions between rigid bodies with matching IDs will be be ignored

  The Group Filter implementation is thoroughly described in the Havok Physics SDK. It offers a good compromise between flexibility and efficiency (the CONSTRAINT_FILTER option, for example, can be quite slow). By cleverly assigning proper IDs to rigid bodies, multiple behaviours can be achieved.

  If the **Auto assign constrained RBs** box is checked, IDs will be assigned to rigid bodies so the GROUP_FILTER will disable collisions between constrained rigid bodies, as the CONSTRAINT_FILTER does (but at a smaller cost). You can also define an specific layer ID for all the rigid bodies in the current asset, as well as defining (through the **Define..** button) which layers should collide with which other layer. Again, check the Physics SDK documentation on more details about how this filter works and how it interprets the different rigid body IDs.

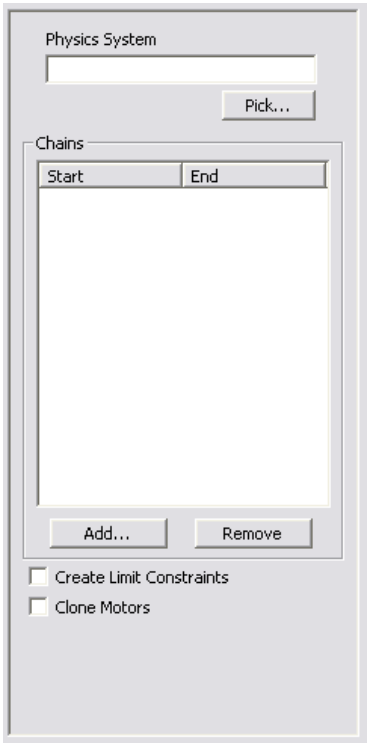
- This filter also allows you to split the rigid bodies and constraints currently in the scene into multiple Physics Systems (collections of constraints and rigid bodies). Selecting the " **Split on type and hierarchy** " options will split the scene by looking at how rigid bodies are connected between them and whether they are fixed or not. Selecting the " **Single System** " option will put all rigid bodies and constraints in a single system.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.2.3.4 Create Constraint Chains

Components Required: Havok Physics

This filter allows you to create constraint chains within the filter pipeline. To create a chain constraint chain you must specify the start and end bodies in the chain. See the Physics SDK documentation for more on constraint chains.

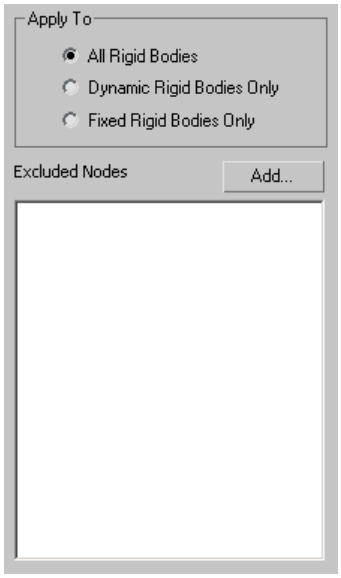


Note:
Currently this filter creates chains for powered constraint only. In the filter piepline only the constraints constructed by a Create Ragdoll filter are powered.

6.2.3.5 Shrink Shapes

Components Required: Havok Physics

This filter goes through each convex shape in the scene and shrinks the original shape, so that when this convex radius is added the total size of the shape matches the original size of the model.

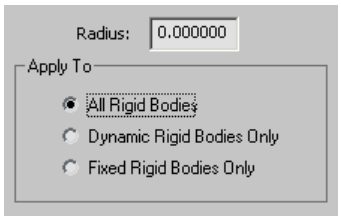


The filter allows you to specify which types of rigid body should be affected. In addition, you can exclude individual objects by using the **Excluded Nodes** list box - those objects will not be shrunk.

6.2.3.6 Set Shape Radius

Components Required: Havok Physics

This filter allows you to override the convex radius values of any rigid bodies in a scene.



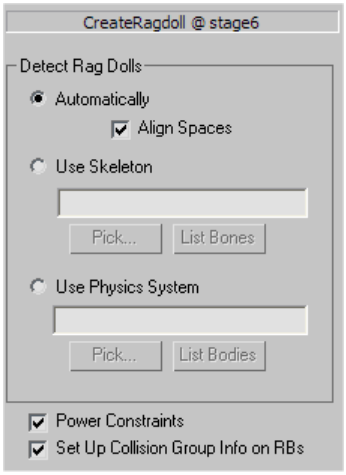
The filter goes through each convex shape in the scene and sets the convex radius to the value specified in the filter options. The filter allows you to specify which types of rigid body should be affected.

6.2.3.7 Create Rag Doll

Components Required: Havok Physics + Havok Animation (Havok Complete)

Havok Complete provides the ability of treating rag dolls as special objects (`hkaRagdollInstance`), tracking the rigid bodies and constraints involved, as well as a skeletal representation of the ragdoll.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



The filter can operate in three modes, which determine how a ragdoll is "detected":

- **Automatic** : Provided mostly for compatibility with older assets, this mode will assume that the first physics system in the scene contains the rigid bodies and constraints that make up the ragdoll. It will create a custom skeleton based on the hierarchy of rigid bodies and constraints.
  - **Align Spaces** : When using automatic mode, you can switch on this option to move the pivot of each rigid body so it matches the location of the child space of the constraint. It will also align the parent space to the child space. Usually this is not necessary if you have created the asset using the Havok Content Tools (as they'll usually be aligned already) but it is useful if you are processing assets created with older tools (the legacy physics exporters for example).
- **Use Skeleton** : A ragdoll will be created by finding rigid bodies with the same names as the bones of the selected skeleton. If the rigid bodies were originally created from a hierarchy of nodes (as they usually are) this is the most useful option, as it allows full control over the components of the ragdoll. It requires that a Create Skeleton filter is used beforehand in order to define the ragdoll skeleton.
- **Use Physics System** : Allows you to specify the rigid bodies and constraints that make up the rag doll by selecting a physics system in the scene. The skeleton will be automatically created based on the hierarchy of rigid bodies and constraints in that system.

The filter also provides further functionality to improve the ragdoll behaviour:

- **Power Constraints** : Constraints in a ragdoll are often powered so they can be driven (to do physics+animation blending for example). If this option is enabled, constraints in the ragdoll will be powered.

  Note:
  Only limited hinges and ragdoll constraints can be powered.
  If any constraint is already powered, this checkbox won't have any effect on it.

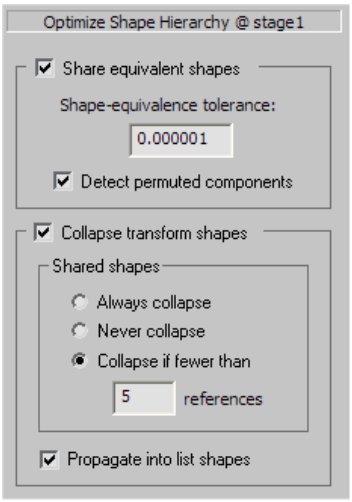
- **Set Up Collision Group Info on RBs** : Assigns collision IDs to the rigid bodies in the ragdoll so, if an `hkpGroupFilter` is used (check the Havok Physics SDK for details), constrained rigid bodies won't collide with each other. (Similar to the 'Auto assign constrained RBs' option in the Create World filter.)

Note that this filter operates on rigid bodies and constraints and therefore needs to be added after the Create Rigid Bodies and Create Constraints Filters. When using the **Use Skeleton** option, it also needs to be added after a Create Skeleton filter which sets up the skeleton of the ragdoll.

6.2.3.8 Optimize Shape Hierarchy

Components Required: Havok Physics

This filter attempts to optimize the shape hierarchies of all the rigid bodies in the scene, reducing the number of shapes used and/or the depth of shape hierarchies. Please check the Havok Dynamics chapter for details on shapes hierarchies and collilsion detection.



There are two (optional) steps, "Share equivalent shapes" and "Collapse transforms".

- **Share equivalent shapes** : If this box is checked, the filter identifies identical (or near-identical) shape structures in one or multiple rigid bodies and changes the hierarchy so such shapes are shared. It replaces "cloned" shapes or shapes structures with "instanced" ones.
  - **Shape-equivalence tolerance** : specifies a tolerance used to decide whether two shapes should be considered identical (and thus replaced with one, shared shape)
  - **Detect permuted components** : When this option is selected the filter will check for permutations of vertices/triangles/planes when comparing hkpSimpleMeshShape or hkpConvexVerticesShape objects. This will possibly detect more instanced shapes but due to the increased amount of comparisons it will take longer time. When the option is off vertices, triangles and planes are compared using the same order.
- **Collapse transforms shapes** : If this box is checked, the filter attempts to collapse transform (and convex transform or convex translate) shapes into their child shape. For example, a transform shape on top of a capsule shape can be collapsed into a new capsule shape with transformed vertices.
  - Shared shapes : When a shape is shared by multiple, different, transform shapes, collapsing each transform shape into it will create a different shape for each transform. Although this reduces the shape hierarchy depth, if the original shared shape was large, having multiple instances of it can increase memory usage. Therefore, a trade off between reduced hierarchy depth and reduced memory usage is required. The options are:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

    * **Always collapse** : Transform shapes will be collapsed even if they point to a shared shape.
    * **Never collapse** : Transform shapes won't be collapsed into shared shapes.
    * **Collapse if fewer than** X r **eferences** : Transform shapes will not be collapsed if they point to a shape shared by X or more shapes/rigid bodies. Specifying a number <=2 is equivalent to **Never Collapse** .
  - **Propagate into lists shapes** : When this option is selected, the filter will propagate any transform shapes applied to list shapes into the children of the list. This may reduce the depth on some of the branches if the transform can be collapsed in any of the children - however if the transform cannot be collapsed into the children this may actually increase the total number of shapes.

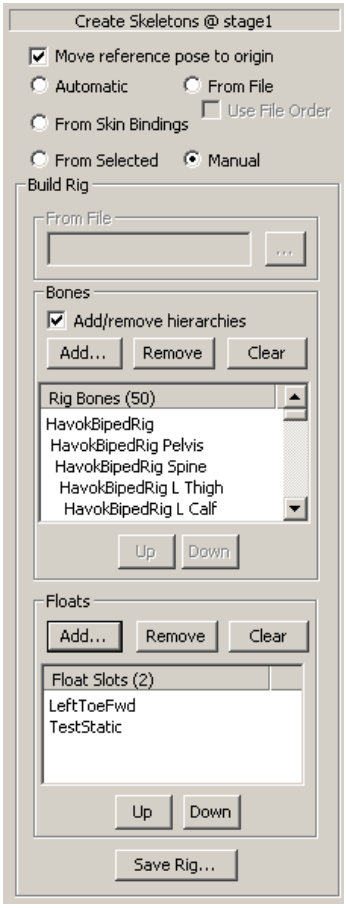
6.2.4 Animation Filters

Animation filters are those which create/modify Havok animation objects. Therefore these should generally only be used if your company has licensed a Havok Product which includes the animation component.

6.2.4.1 Create Skeletons

Components Required: Havok Animation

This filter is fundamental to the animation filter pipline, as skeleton objects are involved in all animation tasks. This filter constructs a skeleton (an `hkaSkeleton` object) from a set of nodes in the scene.



If *Move Reference Pose to Origin* is checked, the translation of the root bone for the created skeleton will be set to zero (as if it whole skeleton was placed at the origin).

The rest of the options determine how the rig is constructed, i.e. which nodes in the scene should be used to create the skeleton. See the Note below for details on 'Float' node addition.

- *Automatic* : This constructs the rig by examining the scene for any skinned meshes. It takes any node associated with the first skin found and travels up the scene graph until it reaches the root node. From this node (just under the root node) it walks back down the scene graph and recursively adds ALL child nodes. Note that no Floats (Float Slots) are added automatically from this option.
- *From Skin Bindings* : This method selects all nodes referenced by the skin bindings in the scene, then searches for and adds any nodes in the scene that connect the referenced nodes together. This usually creates less bones than the " **Automatic** " mode. Note that no loats (Float Slots) are added automatically from this option.
- *From Selection* : This constructs the rig from the nodes selected in the modeler when the filter manager was run.
- *From File* : The file loaded is a text file containing the names of the bones and float slots that should be contained in the rig. Each line of the file corresponds to a different rig bone or float slot. *The path for the filename is always stored relative to the original asset.* With this option selected, the control allowing you to specify the filename becomes enabled. To create a new rig file from a skeleton which has set up, use the *Save Rig...* button.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Use File Order : If this option is checked and the 'From File' mode is used, then the bone order will not be automatically reordered by the filter to match the scene hierachy - the bone order will exactly match that specified in the file.

Warning:

Use File Order allows the user complete control of the bone order. Bone orders which are not *topologically sorted* (every bone's parent occurs before the child in the list) are not allowed as they will cause proplems with partial sampling, the hkaPose class and ragdoll mapping. It is up to the user to ensure that the bone order they specify in the file is topologically sorted, otherwise an animation will not be created.

- *Manually* : This allows you to construct the rig by directly specifying nodes in the scene. With this option selected, the controls allowing manual selection become enabled. To add a node click on the corresponding **Add...** button. This will bring up the *Select Object* dialog which allows you to choose a transform node (for Bones) or an animated float node (for Floats) - For bones, if the **Hierarchy** check box is checked then all children of this node will be added to the list of bones. To remove a bone/float simply highlight it in the list at any time and click *Remove* - if the **Hierarchy** check box is checked then all children are also removed. To empty the list, press *Clear*.

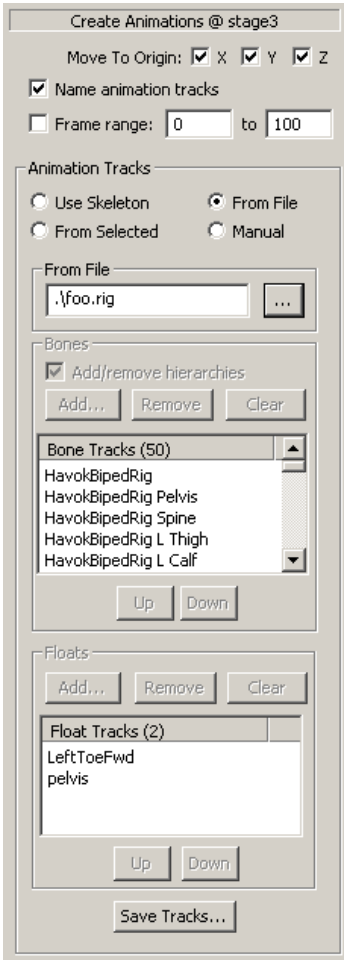
Note:

The float slots for the skeleton specified in the CreateSkeletons filter by the 'Floats' list will normally correspond to the names of (some subset of) the animated float data present in the scene. While the animated bone data (as transforms associated with scene nodes) is always exported out of the modeller into the hkxScene object parsed by the filters, the user must explicitly request animated float data to be exported via one of two methods: Custom Attributes (see appropriate Max, Maya or XSI descriptions) or using Attribute Selection.
Using either of these two methods, the float data will be pulled out as Attributes of type hkxAnimatedFloat inside Attribute Groups associated with the corresponding scene nodes.

6.2.4.2 Create Animations

Components Required: Havok Animation

Given a skeleton (created by the Create Skeleton filter), this filter extracts the keyframes stored in the nodes of the scene associated with the skeleton bones, and creates a skeletal animation object from them (an `hkaInterleavedSkeletalAnimation` to be exact, which contains uncompressed animation data).



If the X,Y or Z *Move To Origin* boxes are checked, the animation will be translated so that the corresponding root bone component starts at the origin.

To use a subset of the valid frame range click the **Frame Range** check box and specify a valid start and end frame number. Both the start and end frame will be included in the animation.

If the **Name Animation Tracks** box is checked, the name of each bone track will be exported with the animation. This information can be used, for example, in order to bind animations to skeletons at run-time. If you don't need this information, you can reduce the size of the exported animation by unchecking this box.

By default, animations are created from all of the bones and float slots in the skeleton, but it is also possible to animate a subset of the skeleton's bones or float slots, such as the legs or the upper body.

There are four available options when choosing what to animate:

- *Use Rig*: This is the default option. Animations are created from all the bones and float slots in the skeleton.
- *From Selected*: Only those bones that are selected at export time are animated. The list of selected bones is displayed in the list at the bottom.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- *From File*: The file loaded is a text file containing the names of the bones that should be contained in the rig. Each line of the file corresponds to a different rig bone or float slot. *The path for the filename is always stored relative to the original asset.* With this option selected, the control allowing you to specify the filename becomes enabled. To create a new rig file from a skeleton which has set up, use the *Save Rig...* button.
- *Manually*: This allows you to construct the rig by directly specifying nodes in the scene. With this option selected, the controls allowing manual selection become enabled. To add a node click on the **Add...** button. This will bring up the *Select Object* dialog which allows you to choose a transform node (for Bones) or an animated float node (for Floats) - For bones, if the **Hierarchy** check box is checked then all children of this node will be added to the list of bones. To remove a bone/float simply highlight it in the list at any time and click *Remove* - if the **Hierarchy** check box is checked then all children are also removed. To empty the list, press *Clear*.

Also, when using the **Manual** option, you can change the order of the tracks inside the animation by selecting a bone or float track and using the **Up** and **Down** buttons.

Note that no animations will be created unless a skeleton is present in the stream (a Create Skeleton filter has been added).

Note:

The float tracks for the animation specified in the CreateAnimations filter by the 'Floats' list must correspond to the names of (some subset of) the animated float data present in the scene. While the animated bone data (as transforms associated with scene nodes) is always exported out of the modeller into the hkxScene object parsed by the filters, the user must explicitly request animated float data to be exported via one of two methods: Custom Attributes (see appropriate Max, Maya or XSI descriptions) or using Attribute Selection.
Using either of these two methods, the float data will be pulled out as Attributes of type hkxAnimatedFloat inside Attribute Groups associated with the corresponding scene nodes.

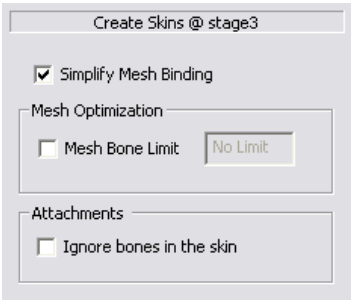
Note:

Preview of float tracks (using the Preview Scene Filter) can be done by enabling the 'Float Tracks' display under the 'View' menu. Float slot names and the current slot value wil be displayed as text in the viewport.

6.2.4.3 Create Skins

Components Required: Havok Animation

The Create Skins filter examines the skinned meshes in the scene (`hkxSkinBinding` objects) and creates a Havok mesh binding (`hkaMeshBinding`) object for each one of them. These mesh bindings are used to link the bones in the rig with the bone indices stored in the mesh. See the section on skinned meshes in the Animation chapter in this manual for more details on how this transformation is done.



Note that the mesh binding will be based on the `hkxSkinBinding` object exported from the modeler. You can control the number of bones associated with the skin by modifying the scene in the modeler or, in some cases, by changing related export options (for example, the XSI exporter's option, **Ignore Unused Bones in Skin Bindings**).

The **Simplify Mesh Binding** option creates a trivial binding, whereby the mappings in the mesh binding are set to NULL. The bone indices in the original mesh are altered to map directly to bones in the skeleton on a one-to-one basis.

You can limit the maximum number of bone influences a mesh primitive can have to facilitate hardware skinning. The mesh will be broken up into sections such that each set of triangles has at most the specified limit of bones affecting it, and the remapping from the original skeleton bones to the reduced set of bones in each section will be stored and used at runtime.

Note:
When using the bone limit feature, DirectX must be selected and hardware skinning enabled in the Preview Scene filter in order to correctly display skinned meshes.

The filter will also create Attachment objects based on other meshes parented to the bones in the skeleton. Check the Animation chapter in this manual for details on attachments.

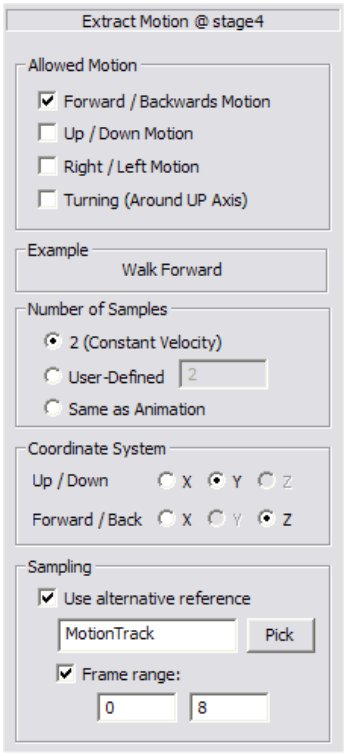
The **Ignore bones in the skin** option, when selected, will tell the filter not to create attachment objects from mesh nodes which are also bones of the skeleton.

Note that no skin will be created unless a skeleton is present in the stream (a Create Skeleton filter has been added).

6.2.4.4 Extract Motion

Components Required: Havok Animation

The Extract Motion filter is used to sample the displacement (relative to the environment) from the animation. This movement can be reconstructed later and used to drive the world position of the animation during playback.



The **Allowed Motion** section specifies the type of movement to be extracted from the animation. Select any combination of these check boxes and the ' **Example** ' section displays the type of motion that will be extracted.

Use the **Number of Samples** section to specify how many samples to take when extracting the motion. This can be any number from 2 to n, where n is the number of frames in the animation.

Finally select the appropriate **Coordinate System** for performing the motion extraction. For most locomotions this corresponds to Z up in Max and Y up in Maya.

The **Sampling -Use Alternative Reference** option allows you to select an animation track (or potentially an entirely different object) other than the root animation track as a frame of reference for motion extraction. This option is switched off by default. When enabled the name of the node to use should be entered in the edit box. Alternatively the ' **Pick** ' button can be used to select a node.

A **Frame Range** can be specified for the alternative reference. This option should be used if the range of keyframes exported from the modeller doesn't match the lenght of keyframes in the animation (that is, if the **Frame Range** option was used in the Create Animations filter). In that case, select the start and end frames of the alternative reference frame so they match the start and end of the animation.

Float tracks will be ignored. See the section on Motion Extraction in the *Havok Animation* chapter for specific details on our motion extraction process.

Note that this filter expects to find at least one Interleaved Animation in the data stream (it usually follows a Create Animations filter).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.2.4.5 Spline Compression

Components Required: Havok Animation

The Spline Compression filter provides an interactive interface for creating spline compressed animation data. The spline compression filter interface is show in . This section of the manual describes the Spline Compression filter interface. For detailed information about Havok's Spline Compression technology, please refer to the *Spline Compression Overview* section of the Animation chapter.

Note that this filter expects to find at least one Interleaved Animation in the data stream (it usually follows a Create Animations filter).

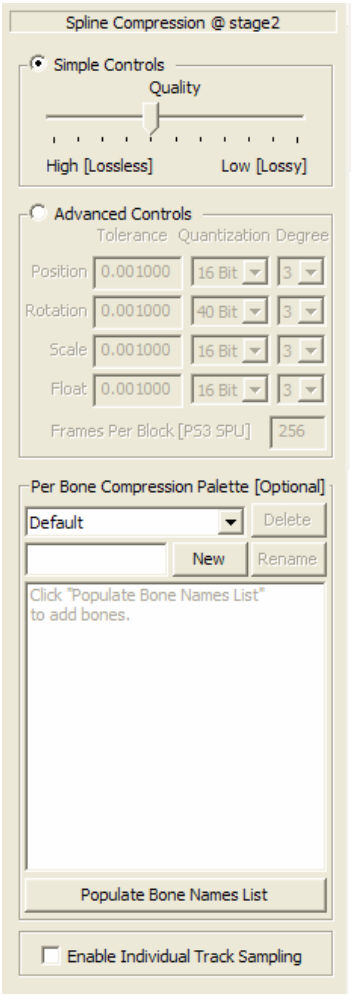


Figure 6.1: Spline Compression Filter Interface

The spline compression interface consists of four panes which are described in , ,, and respectively. Each time the spline compression filter is run the size of the resulting compressed animation and the effective compression ratio will be displayed in the **Log Window** . Users of the PLAYSTATION®3 SPU technology will additionally want to note the "Size of largest block" information.

Simple Controls

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The simple controls pane is shown in . The slider bar provides a single control to increase or decrease the quality of the compressed animation. Users may try various quality settings and preview the results by using the Preview Scene Filter and clicking the **Run Configuration** button. The size of the resulting compressed animation is displayed in the **Log Window** .

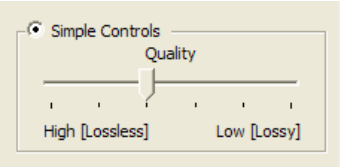


Figure 6.2: Simple Controls Pane

The simple control slider modifies the rotation tolerance of the compression settings. Each tick mark on the quality control slider corresponds to a 10x change in the rotation tolerance. The simple controls work best with animations which consist primarily of rotations. Users desiring more control over specific compression features may activate the Advanced Controls pane described in .

Advanced Controls

The advanced controls pane is shown in . This pane allows users full control over all spline compression settings. The Tolerance value typically has the greatest impact on the quality of the animation and may be fine-tuned to the user's preference. The Tolerance setting should be the primary value used to govern compression settings. The Quantization settings may be changed in situations where higher compression ratios are desired and quality may be sacrificed. The Degree setting controls the smoothness of the result; degree 3 curves will tend to be more visually appealing than degree 1 curves at a modest cost in output file size. Each of these settings is described in more detail in the *Spline Compression Overview* section of the Animation chapter.

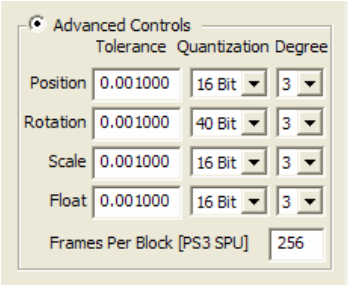


Figure 6.3: Advanced Controls Pane

Users of the PLAYSTATION®3 SPU technology will additionally want to note the "Size of largest block" information printed to the **Log Window** . A warning will be issued if the animation is in danger of being too large to fit within the SPU memory requirements. If a warning is found, the size may be reduced by lowering the "Frames Per Block" setting at the bottom of the Advanced Controls pane.

Users desiring a simplified settings interface may activate the Simple Controls pane described in .

Per Bone Compression Settings

The Per Bone Compression Palette pane, shown in , allows users to tailor the compression settings individually for each bone in the skeleton. Per bone compression settings may be used any time the required accuracy of bones varies throughout the skeleton. The default setting is to apply compression settings uniformly for each bone.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Judiciously setting compression parameters per bone can result in a significant reduction in output data size. One such use case is to create level of detail compression by grouping bones roughly by depth in the hierarchy; tight tolerances could be user for "major bones" such as spine and arm bones while more relaxed tolerances for "minor bones" such as hands and fingers could be used to gain additional savings. A similar use case might involve grouping bones as "upper body" and "lower body". Extremities such as hands often contain a significant number of bones though their visual impact may be limited.

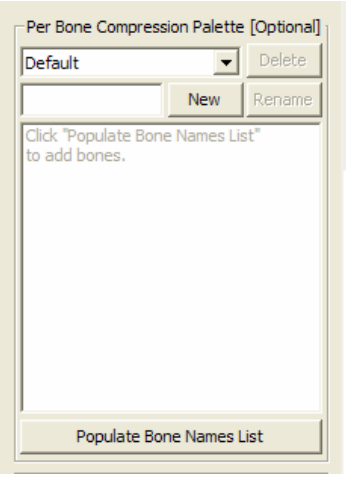


Figure 6.4: Per Bone Compression Palette Pane

The use cases described in the previous paragraph motivated the design of a compression settings palette. A small number of *per bone compression sets* may be created, and multiple bones may be added to each set. Any bones which are not explicitly added to a set are assumed to belong to the Default set.

Creating a Per Bone Compression Set

The first step to create a new per bone compression set is to populate the list of bone names by clicking the **Populate Bone Names List** button. Note that this step expects at least one skeleton to be present in the data stream (typically created by an earlier Create Skeletons filter). Once the list of bone names has been populated, a per bone compression set may be created. Typing a name in the text box to the left of the **New** button followed by clicking the **New** button will create a new per bone compression set. If the text box has been left empty, the per-bone compression set will be created with a default name. A per bone compression set may be renamed at any time by typing a new name and clicking the **Rename** button. A per bone compression set may be removed at any time by clicking the **Delete** button.

Once a per bone compression set has been created users may add bones by clicking on the bone names in the list. Pressing the Control key while clicking allows multiple bones to be added to the set as shown in . In this example, bones belonging to a character's hand have been added to a per bone compression set named "Minor Bones".

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

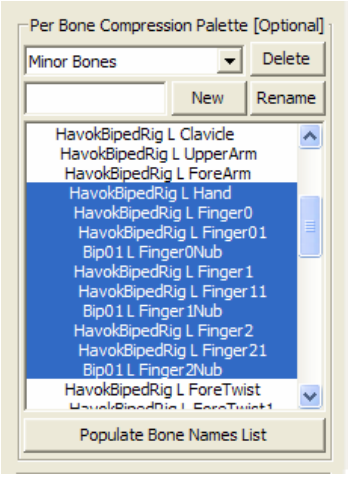


Figure 6.5: Per Bone Compression Palette Showing Added Bone Names

The compression settings for the current set may be adjusted using the controls described in and . These compression settings will be applied to each bone in the current set. Multiple per bone compression sets may be created, each with unique compression settings. Bones may belong to only one set. If a bone name is added to a set, it is automatically removed from all others. Bones which do not belong to any set are considered to belong to the Default set, and the compression settings of the Default set are applied. The Default set compression settings can be changed like any other set. The Default set cannot be renamed or deleted.

Individual Track Sampling

The check box shown in allows individual tracks (and bones) of the spline compressed animation to be sampled at runtime. This feature is disabled by default as it requires a modest amount of data to be stored in the compressed animation which indicates the start location in memory for each track. This check box must be checked at the time of content creation in order to use the per bone or per track sampling features of the animation SDK. The affect of choosing this option on the size of the output may be determined by running the filter and checking the **Log Window** .

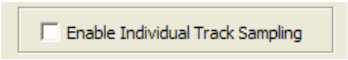
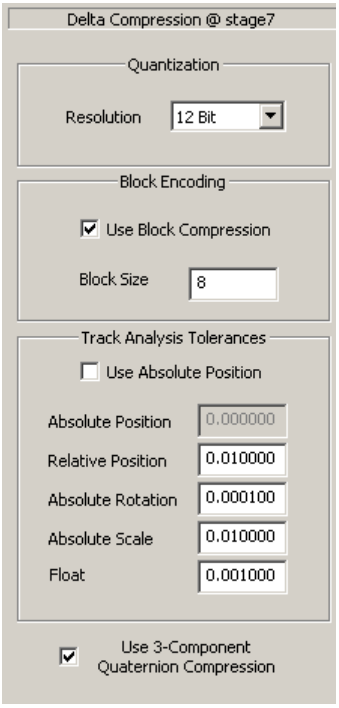


Figure 6.6: Individual Track Sampling Pane

6.2.4.6 Delta Compression

Components Required: Havok Animation

With the Delta Compression filter you can convert an uncompressed Interleaved Animation to a compressed animation using the Delta Compression algorithm.



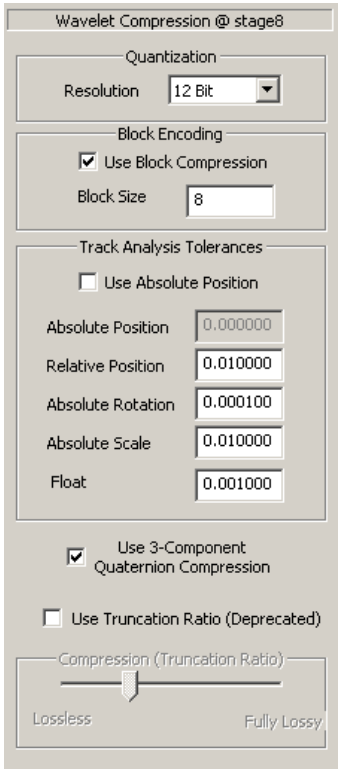
Please check the *Compression Overview* section in the Animation chapter in this manual for details on how Delta Compression works, and the Controlling Compression section of this manual for details on how to best to choose and tweak these parameters.

Note that this filter expects to find at least one Interleaved Animation in the data stream (it usually follows a Create Animations filter).

6.2.4.7 Wavelet Compression

Components Required: Havok Animation

With the Wavelet Compression filter you can convert an uncompressed Interleaved Animation to a compressed animation using the Wavelet Compression algorithm.



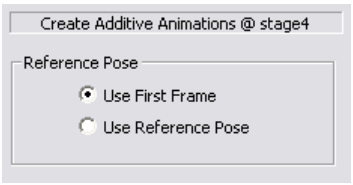
Please check the *Compression Overview* section in the Animation chapter in this manual for details on how Wavelet Compression works, and the Controlling Compression section for details on how to best to choose and tweak these parameters.

Note that this filter expects to find at least one Interleaved Animation in the data stream (it usually follows a Create Animations filter).

6.2.4.8 Create Additive Animations

Components Required: Havok Animation

Additive animations allow you to create animation which represent fine character detail. These animation can be reused in many situations and layered on existing animations. The Create Additive Animations filter creates additive animations from all raw interleaved animation found in the data stream. To create this additive animation you can specify some options for which base pose to use when determining the detail in the animation.



There are two available options:

- **First Frame** : This is the default. The first frame of the animation is subtracted from each frame of the entire animation.
- **Reference Pose** : The reference pose of the animation (of the first rig in the scene) is subtracted from each frame of the entire animation.

Note that no additive animations will be created unless an interleaved animation has already been added to the stream (using the Create Animation filter). If you are using the **Reference Pose** option you must also have at least one skeleton there too (using the Create Skeleton filter).

6.2.4.9 Loop Animations

Components Required: Havok Animation

This filter forces an animation to loop by crossfading the start and the end of the animation over a number of frames. This number is specifed by the filter options.

The cross fade happens for the specified number of frames at both the start and the end of the animation.

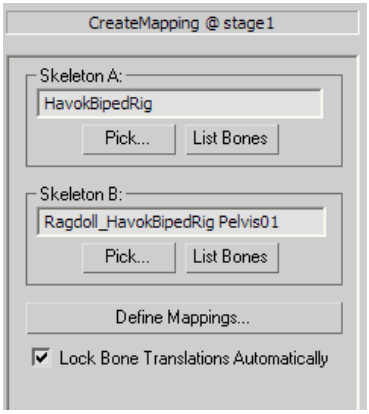
Usually this filter is applied after a Create Animations filter.

6.2.4.10 Create Mapping Filter

Components Required: Havok Animation

Sometimes we want to play an animation (or, in more general terms, convert a pose) on different skeletal representations of the same character. The most common use case is the transformation between a high-res (high bone count) representation of a character to a low-res (low bone count) version (a ragdoll for example), and vice versa. In the first case, we may want to transform a high-res pose into a ragdoll pose in order to drive the ragdoll; in the second case, we may want to convert a ragdoll pose into a high-res pose in order to skin the ragdoll (or blend its behaviour). In most cases, we will want to do both conversions multiple times.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



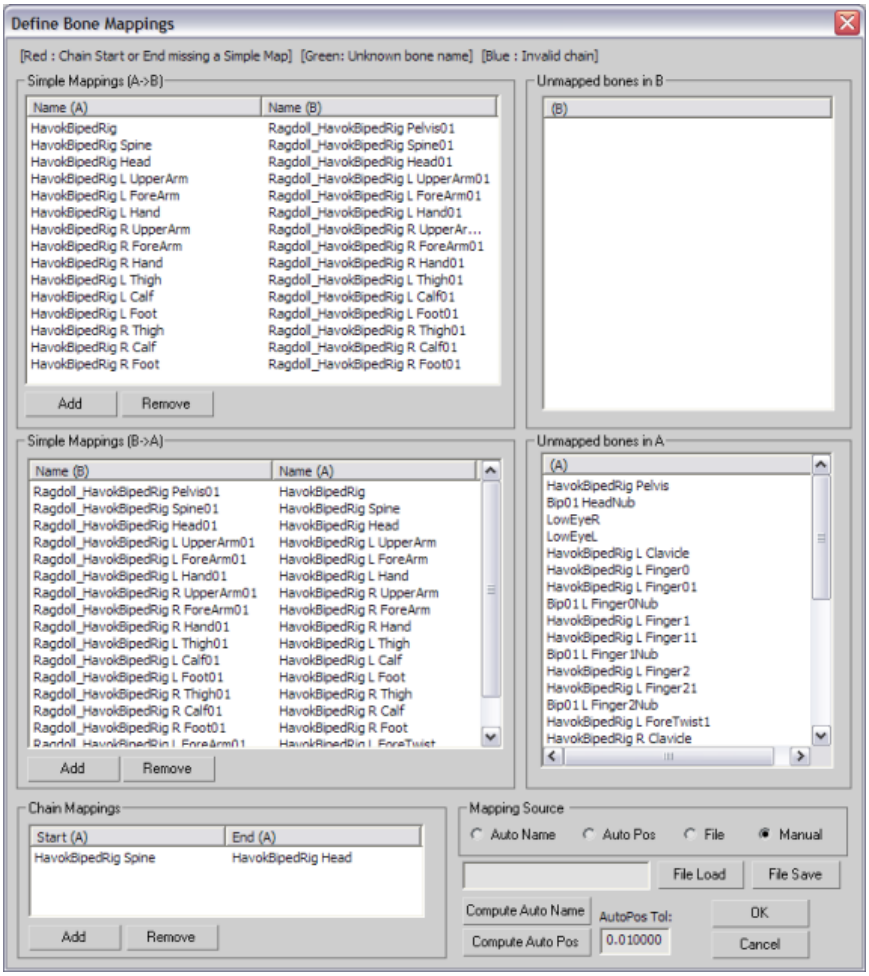
A skeleton mapper allows us to do these conversions. The Create Mapping filter sets up all the information required by the skeleton mapper (hkaSkeletonMapper) at run time by comparing and matching two skeletons (in their reference pose), finding pairs of bones and pairs of chain of bones.

Thus, the minimal information a skeleton mapper requires is two skeletons, A and B. If not specified, the filter will pick A as the first skeleton, and B as the first skeleton associated with a ragdoll that are found. It is always best to manually specify both skeletons using the **Pick** buttons. Also, you won't be able to see/customize the mapping (use the **Define Mappings..** button) unless both skeletons have been specified.

In order for a skeleton mapper to operate properly at run-time, it requires knowledge about which bones/joints can translate and which can not. By selecting the **Lock Bone Translations Automatically** option, the filter will also flag all bones in both skeletons (except the roots) as rotation-only.

The **Define Mappings...** button opens a new window, which allows you to view and edit the mapping information, which is used at run-time in order to convert poses from A to B and from B to A.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Simple Mappings

A "Simple Mapping" is a one-to-one association from a bone in one skeleton to a bone in the other skeleton. For example, a typical simple mapping from HighRes to RagDoll would be "HighRes Head" is mapped to "RagDoll Head". This means that, at runtime, the position and orientation of "HighRes Head" will be used to place the "RagDoll Head".

Usually, for every Simple Mapping A->B there is also the symmetrical mapping B->A (in the example, "RagDoll Head"->"HighRes Head").

Simple mappings are listed in the first and second windows on the right side of the Define Mappings window. In *Manual Mode*, you can add and remove simple mappings to either A->B or B->A by using the **Add** and **Remove** buttons in each list.

Chain Mappings

Chain mappings are atomic associations between 2 or more bones in one skeleton to 2 or more bones in the other skeleton. The best example (and the most common usage) is a spine: while a highres skeleton may have 5 or more spine links, a ragdoll version of the same character may have just 1 or two links (rigid bodies).

A chain mapping drives a chain of bones by trying to match, as well as possible, the start and end

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

bones of the chain to another chain. In the example of the spine, it may try to match the chain "spine1"-"spine2"-"spine3"-"spine4"-"spine5"-"head" ("spine1-head") to the chain "ragdollspine1"-"ragdollspine2"-"ragdollhead" ("ragdollspine1-ragdoll head").

A condition for the creation of a chain mapping is that both the start and the end of the chains have to be simple mappings; continuing with the spine example, spine1/ragdollspine1 and head/ragdoll head need to be simple mappings.

Chain Mappings are listed at the bottom of the Define Mappings dialog. They are listed with the names of the bones in skeleton A (the associated chain in B can be deducted by applying the A->B simple mappings). In *Manual Mode*, you can add and remove Chain Mappings by using the **Add** and **Remove** buttons.

Unmapped Bones

After applying all Simple and Chain Mappings from one skeleton to another, it is possible for many of the bones in the destination skeleton not to have been modified. This is usually the case for the skeleton of higher resolution, and is uncommon for the low-res skeleton (ragdoll skeletons, for example).

When a bone is unmapped, its position and orientation is kept relative to (it moves with) its parent (i.e., its local transform is not modified).

Unmapped bones are listed on the left hand side of the Define Mappings dialog. Any bones that are not part of a simple mapping or a chain mapping are listed there.

Mapping Sources

There are different ways this mapping information can be set up: they are reflected in the "Mapping Sources" radio buttons:

- **Auto Name** : Simple mappings A->B and B->A will be created by detecting bones in both skeletons which have the same name (removing any prefixes). This option allows more control from the user (as you can carefully choose the names according to your desired mapping) but it involves extra work in order to name all the bones, and therefore is error prone. This mode is provided for backwards compatibility with older assets.

  In *Manual Mode*, you can still fill the mapping information in the dialog with the results of Auto Name by clicking on the **Compute Auto Name** button.

- **Auto Pos** : The most accurate and simple method to use is Auto Pos. This method will compare the position of the bones in each skeleton and create simple mappings A->B and B->A for each pair of bones whose position match within a tolerance (specified by the **AutoPos Tolerance** edit box).

  Note:
  Using this method requires that both skeletons (and their bones) are properly aligned (this alignment is also important for the proper behaviour of the skeleton mapper).

  In *Manual Mode*, you can fill the mapping information in the dialog with the results of the Auto Pos calculations by clicking on the **Compute Auto Pos** button.

  Both Auto Name and Auto Pos methods first discover simple mappings. Chain Mappings are then constructed by detecting pairs of simple mappings where the bones in-between haven't been mapped (think of the "spine1" - "head" example).

- **File** : Mapping information can also be stored in a file and reused. This can be useful in order to protect the mapping information from being modified. The file in particular is specified in the

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

edit box below, and its path is relative to the location of the asset. The **File Load** and **File Save** buttons allow you to save and load the current mapping information.

- **Manual** : In this mode you are able to modify the mapping information manually. This option is useful in the cases where mapping information calculated automatically (by the **Auto Pos** or **Auto Name** options, for example) may not be enough to provide a perfect mapping.

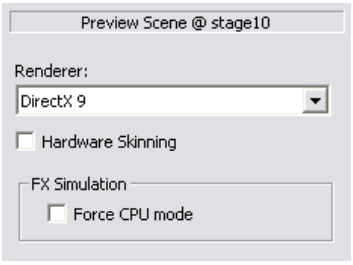
6.2.5 Graphics Filters

Graphics filters are those which modify/visualize graphical elements of the asset. These can always be used regardless of whichever Havok Product is licensed for use at run-time.

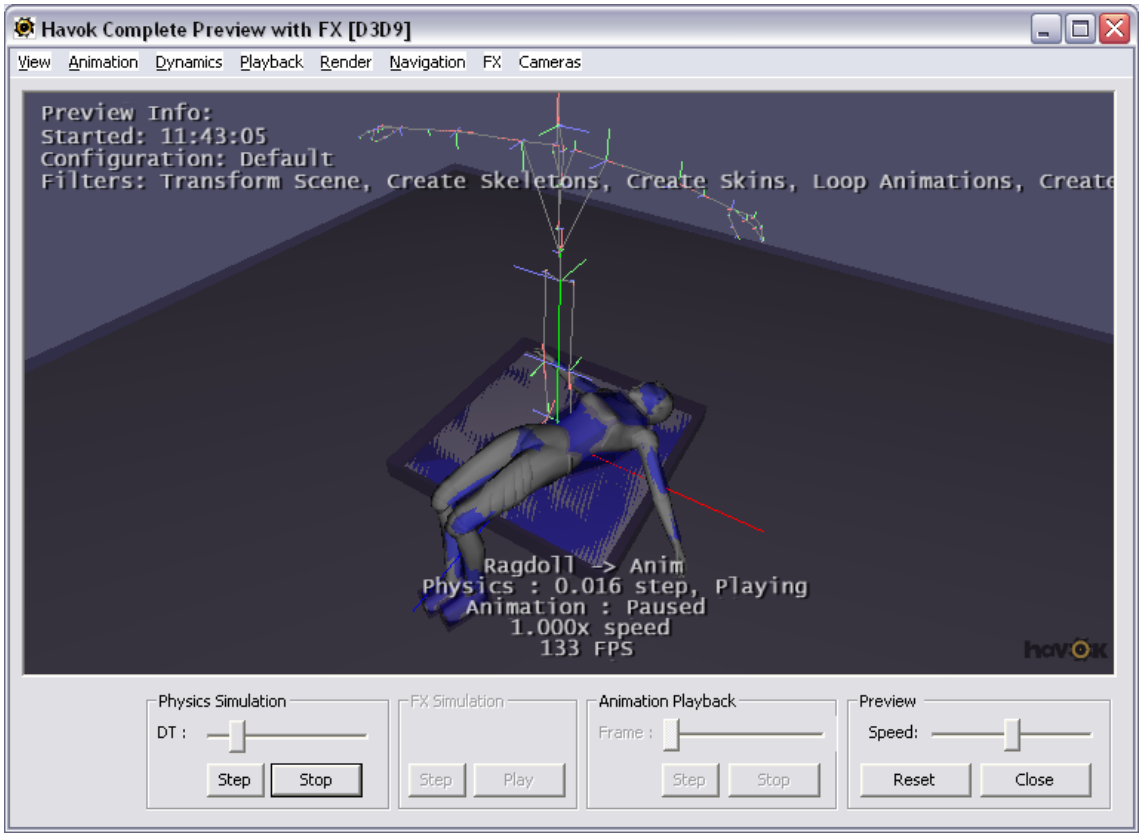
6.2.5.1 Preview Scene

Components Required: Common

The Preview Scene filter allows you to visually examine the contents of your asset (the result of the filters preceding it), without modify anything. This filter is modeless and can be added to the filter stack several times (for example, in order to compare different compression results).



Different modelers and tools use different graphics contexts, so the filter allows you to specify which graphics system to use in the drop list at the top. If you are trying to view an asset that has limited bone sections (see the Create Skin filter above) you must use a DirectX renderer and have **Hardware Skinning** on.



When executed, the filter displays both physics and animation information, if they are present, and provides a set of simulation controls:

The **Physics DT** slider is enabled when physics information (rigid bodies) are present. It allows you to specify the size of the physical timestep (displayed at the bottom of the 3d window). The Physics Simulation **Play/Stop** button can be used to start and stop the physical simulation. You can also use the Step button in order to advance the simulation a single step.

The **Animation Frame** slider is enabled when there are animations present in the scene. It displays the location of the current frame played, and allows you to scrub the animation forwards and backwards.

The Animation **Play/Stop** button can be used in order to start and stop the playback of animations. You can also use the Single Step (>—) button in order to advance the animation a single frame.

The **Speed** slider can be used in order to slow down the playback of both physics and animation. Note that this speed doesn't affect the actual time step taken by the physics (it just plays in slow motion). This can be very useful in order to examine animations and physical behaviours in detail.

The **Reset** button will restore both the physics and animation content to their original starting state.

The **OK** button will close the preview.

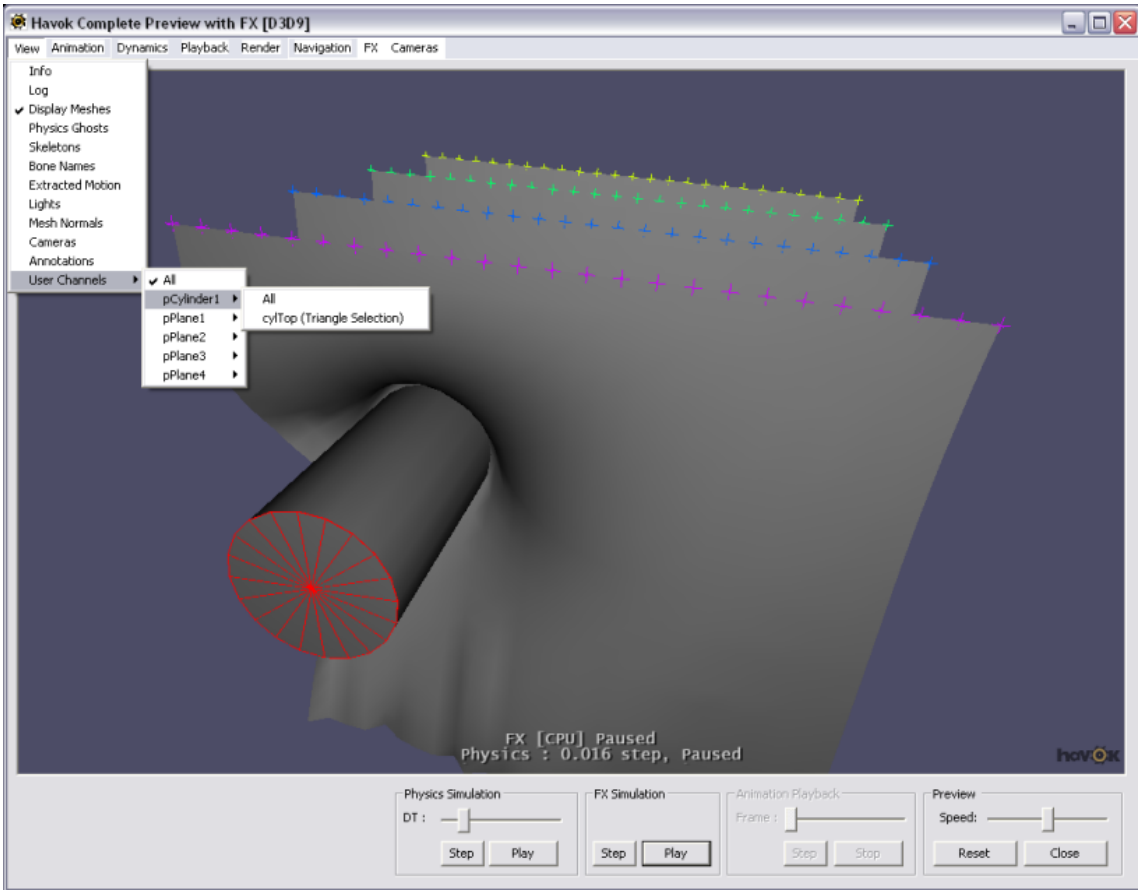
Menus

The menus in the Preview Dialog give you control over many aspects of the preview.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- **View** : Gives you control over what different elements are displayed in the preview
  - **Info** : Extra information about filters, speed etc, is displayed in the preview.
  - **Display Meshes** : On by default, if switched off meshes (other than ghost meshes associated with rigid bodies) won't be displayed. Useful when you just want to see the exact physical world being simulated.
  - **Physics Ghosts** : Alpha blended representation of the rigid bodies in the scene will be displayed in the viewport. Useful in order to compare the physical objects with their display representation.
  - **Skeletons** : Any skeleton found in the scene will be rendered (as a set of lines connecting the bone positions).
  - **Bone Names** : Opens a dialog where the list of bone names is displayed - the names of those bones selected will be displayed in the viewport.
  - **Extracted Motion** : If there are animation present, and those animations had motion extracted from them (see the Extract Motion filter for details) a visual representation of the motion will be displayed as a path in front of the character.
  - **Lights** : Light objects in the scene will be displayed as tripods in the viewport.
  - **Mesh Normals** : Mesh normals will be displayed.
  - **Cameras** : Camera objects in the scene will be displayed as tripods in the viewport.
  - **Annotations** : If an animation has annotations, those will be displayed in the viewport as text that temporarily appears at the annotation time.
  - **Float Tracks** : If an animation has float tracks, those will be displayed in the viewport as text in the format [slot name] [current slot value].
  - **User Channels** : If any meshes in the scene have associated vertex or triangle selections, these selections can be examined in the preview. Under the User Channels menu, a submenu appears for each mesh containing a selection. Each submenu lists the selections associated with that mesh, with the selection type indicated in brackets. Both vertex and triangle selections are supported. Choosing an item toggles the display of that selection. Vertex selections are depicted as crosshairs on the model, and triangle selections as lines along the triangle edges. Each separate selection which is currently visible is drawn in a different color to distinguish them. Note that all of the selections in the scene can be viewed by selecting the "All" menu item in the submenu containing the list of meshes. Additionally, all of the selections associated with a particular mesh can be viewed by selecting the "All" menu item in the submenu for each mesh.

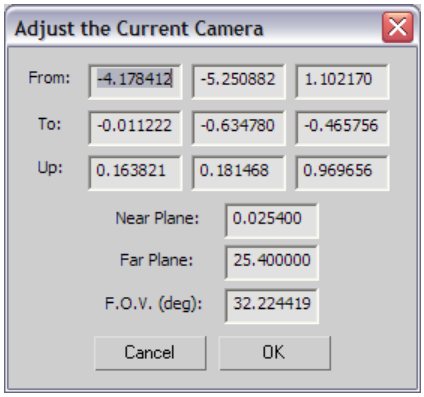
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



- **Animation** : Options related to animations
  - **Accumulate Motion** : If motion has been extracted from the animation, it will now be accumulated during playback. This is very useful in order to examine the proper behaviour of extracted motion. If motion has been extracted properly, the character should displace continuosly accross animation loops (i.e., it should continue moving forward while running).
  - **Reference Pose** : When switched on, the Reference Pose of the skeleton will be displayed (instead of the animation).
  - **Rag Doll->Animation Map** : If there is a Ragdoll Instance (see Create Ragdoll filter), and a skeleton mapper from the rag doll skeleton and a high-res skeleton (see Create Mapping filter), switching this option on will automatically sample poses from the ragdoll (using the ragdoll instance) and convert them to a pose for the high-res skeleton (using the skeleton mapper), overriding any animation that may be present. If there is skin or attachments associated with the high-res skeleton, they will reflect this pose. This option is very useful in order to test Ragdoll Mappings (from the rag doll towards the high res pose).
  - **Animation -> Rag Doll Map** : If there is a Ragdoll Instance (see Create Ragdoll filter), and a skeleton mapper from a high-res skeleton to the skeleton of the ragdoll (see Create Mapping filter), switching this option on will automatically convert poses from the high-res skeleton (or any animation running on it) into poses for the ragdoll (using the skeleton mapper) and set the ragdoll rigid bodies position and orientation (using the ragdoll instance) to that pose. This option is very useful in order to test Ragdoll Mappings (from the high-res pose towards the rag doll).
  - **No Mappings** : If you want to ignore any mapping between rag dolls and high-res skeleton, select this option.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- **Dynamics** : Options related to the physical simulation
  - **Gravity X, Y, Z, Negate** : Allows you to modify the direction of gravity
  - **Simulation Frequency (Hz)** : You can also select the simulation frequency through this menu - this has the same effect as modifying the **Physics DT** slider.
- **Playback** : In this menu you can select different playback speeds (for both physics and animation). This has the same effect as modifying the **Speed** slider.
- **Render** : Different options concerning how things are rendered
  - **Lighting** : Switches on/off lightning in the scene
  - **Shadows** : Switches on/off the rendering of shadows
  - **Textures** : Switches on/off the use of textures
  - **Cull backface** : Switches on/off the culling of back faces (triangles not facing the camera)
  - **Wireframe** : If on, it displays all meshes using wireframes
  - **Color Primitives** : If on, it displays each mesh with a different, randomly picked color
  - **Flashlight** : If on, all lights are switched off and replaced by a light placed at the camera position (a flash light).
- **Navigation** : Allows you to switch between different navigation modes (different mappings between mouse buttons and movement and camera operations)
- **Cameras** : Lists the available cameras in the scene and allows you to pick one for use in the preview.
  - **Adjust Camera...** : For the currently selected camera, it opens a dialog where you can adjust all the camera parameters. Particularly useful if you run intro problems with clipping planes.



Navigation and Interaction

You can move the camera position, orientation and target by clicking and dragging with the mouse in the 3d viewport. Different behaviours are specified by the **Navigation** menu - some of them replicate the behaviour in modelers like 3ds Max and Maya.

Mouse Picking

When physical objects are present, you can also use the *mouse picking* feature by placing the mouse over a rigid body and then pressing and holding **SPACE** . This attaches a virtual spring between the mouse

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

cursor and the selected object. Moving the mouse while **SPACE** is pressed then drags the object around. FX rigid bodies (although not FX cloth) can also be picked and dragged.

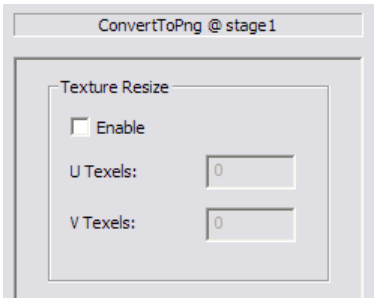
Resetting the Camera

While the *Reset* button restores the physics and animation to their original starting states, it does not alter the current camera settings. To reset the camera to (one of) the initial state(s), re-select the appropriate camera from the *Cameras* menu.

6.2.5.2 Convert to PNG

Components Required: Common

This filter will convert any texture in the scene pointing to an external file into an "inplace" texture, stored in PNG format.



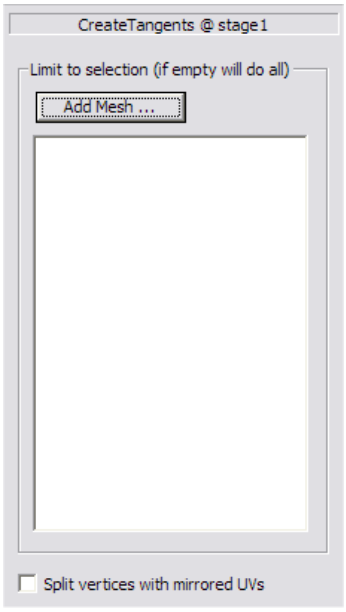
If **Texture Resize** is on, the filter will also resize any texture to the given size.

6.2.5.3 Create Tangents

Components Required: Common

This filter will calculate tangents and binormals of the meshes in the scene by examining normals and texture UV coordinates.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



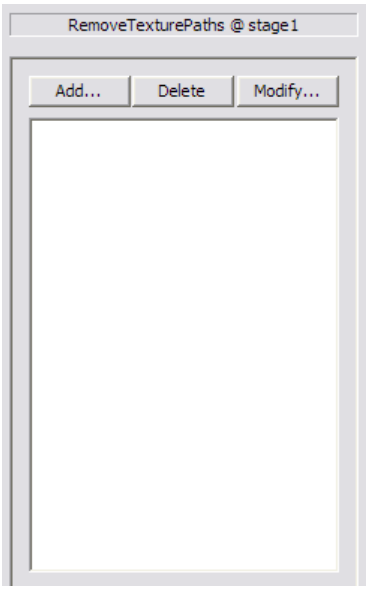
By default, the filter will operate on all meshes. Alternatively, you can use the filter options to specify which individual meshes to process.

When the **Split vertices with mirrored UVs** check box is selected, the filter will also search for vertices shared by triangles with mirrored UVs, and duplicate them accordingly (since tangent information will be different depending on the triangle).

6.2.5.4 Remove Texture Paths

Components Required: Common

This filter can be used to convert absolute texture paths into relative ones.



The filter options allows you to define a list of texture folders. When the filter is executed, it will search through textures, and for any texture in the scene that points to one of the specified folders, it will convert the absolute path into a relative one (relative to that folder in particular).

6.3 3ds Max Tools

6.3.1 Introduction

In this chapter we present the 3ds Max specific components of the Havok Content Tools:

- The 3ds Max Scene Exporter
- The 3ds Max Physics Tools
- The 3ds Max Animation Tools

We also present several tutorials:

- Export and Animation Basics: How to set up, export and process an animation.
- Physics Basics: How to set up, export and process a physics scene.
- More On Rigid Bodies: A few more concepts about rigid body setup.
- Rag Doll Setup: How to set up, export and process a rag doll and its association with a high-res skeleton.

Check also:

- Common Concepts: For common concepts useful for all Havok Tools (regardless of modeler).
- The Havok Filter Pipeline: For details on how the content created by the 3ds Max tools can be processed.
- Extending the 3ds Max Tools: For information about exporting custom data and MAXScript \ C++ access.

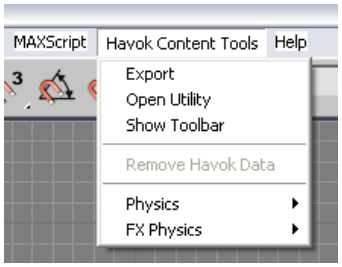
6.3.2 3ds Max: Scene Exporter

Havok's 3ds Max Scene Exporter is responsible for exploring 3ds Max's scene graph, collecting scene information, and passing it to the filter pipeline for processing.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.3.2.1 Accessing the Scene Exporter

The Scene Exporter is implemented as a 3ds Max Utility object. The best way to access this utility and many of it's options is through the **Havok Content Tools** menu and toolbar. If the toolbar is not visible then you should select the *Havok Content Tools > Show Toolbar* menu option.

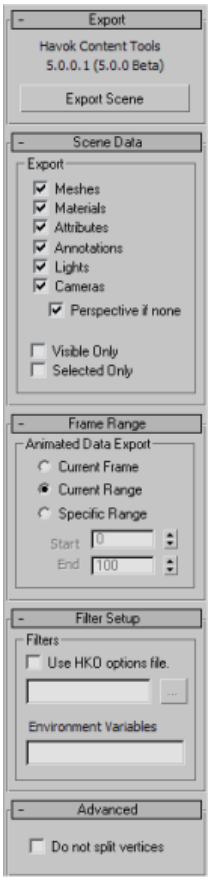


Most of the items in the menu and toolbar relate to the Physics Tools and are described in their own section. Let's focus only on the options related to the Scene Exporter:

- **Export** : This invokes the Scene Exporter, which will navigate the current scene graph extracting information about meshes, lights, nodes, attributes, etc (according to the current export options). After this is done, the Havok Filter Manager will be invoked in order to process this data.
- **Open Utility** : This command will open the Havok Scene Exporter Utility, where various export options can be defined. These options are described in the following section.

6.3.2.2 Export Options

The export options for the 3ds Max Scene Exporter are exposed through the utility interface. The export options UI is displayed by pressing the button or selecting ' **Havok Content Tools -> Open Utility** '.



- **Export Scene** : This button performs the same operation as the **Export** option in the **Havok Content Tools** menu or the button in the toolbar.

The **Scene Data** rollout controls which parts of the scene will be exported or not:

- **Meshes** : Controls whether meshes are exported or not. Enabled by default.
- **Materials** : Controls whether materials and textures are exported or not. Enabled by default.
- **Attributes** : Controls whether custom Havok attributes are exported or not. Enabled by default.
- **Annotations** : Controls whether annotations are exported or not. Enabled by default.
- **Lights** : Controls whether lights are exported or not. Enabled by default.
- **Cameras** : Controls whether cameras are exported or not. Enabled by default.
  - **Perspective if none** : If this is enabled and no cameras are available to export then a default camera, based on the perspective viewport, will be exported with the scene. Enabled by default.
- **Visible Only** : Only objects which are currently visible in 3ds Max will be exported. Disabled by default.
- **Selected Only** : Only objects which are currently selected in 3ds Max will be exported. Disabled by default.

The **Frame Range** rollout controls how animated data is handled by the exporter:

- **Current Frame** : The exporter will only sample the scene at the current frame (so no animation data will be exported). This option can considerably speed up the export process.
- **Current Range** : Animated data (node transforms, annotations, attributes) will be exported - the animation will be sampled through the current animation range in 3ds Max. This is the default option.
- **Specific Range** : This allows you to specify an animation range (through the **Start** and **End** edit boxes) for the scene exporter to sample animation. This option is useful if you want to control the range of the animation export regardless of the current range in 3ds Max.

The *Filter Setup* rollout provides the ability to customize some elements of the filter pipeline:

- **Use HKO Options File** : Enabling this check box allows you to choose an HKO file that contains the desired filter configurations to use for processing the asset. Options (filter configurations) are saved with the asset (the max scene), but sometimes it is useful to specify a custom set of options explicitely (for example, during batch processing, or to avoid having to build a filter set up from scratch in new assets).
- **Environment Variables** : Here you can specify pairs of (variable, value) by concatenating strings of the form : **VARIABLE1=VALUE1; VARIABLE2=VALUE2;** etc.. This data is not usually stored with the final asset but can be used by a (custom) filter during process. For example, the Platform Writer filter will replace any environment variables specified in its **Filename** field by their values. For more, see the section on environment variables.

The **Advanced** rollout contains other miscellaneous, less-used options:

- **Do Not Split Vertices** : By default, the 3ds Max Scene Exporter will export as many mesh vertices as required in order to store texture and normals. So, for example, while for a cube only 8 geometric vertices would be required, in order to store per-face normal and texture coordinates, 24 vertices need to be exported.

  Selecting this option will override the default behaviour, and multiple vertices that have the same position will be exported as one, regardless of their normal or UV coordinates. This may reduce the size of the exported mesh - but due to the lack of accurate normal and texture coordinates, the mesh may not display as it originally did in the modeller.

6.3.3 3ds Max: Physics Tools

6.3.3.1 Introduction

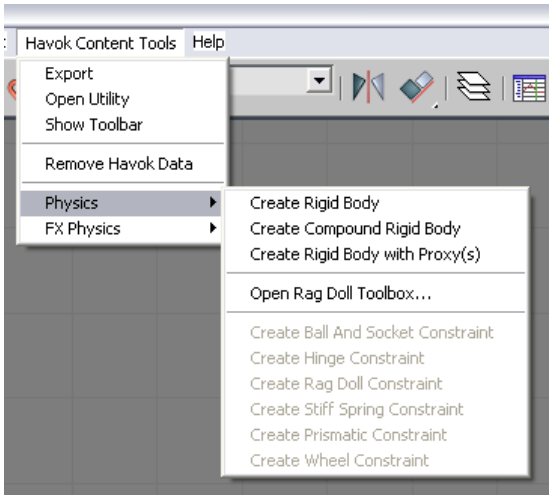
While all modelers, including 3ds Max, have the ability to specify scene and animation data that has a close relationship to the equivalent run-time objects in Havok (and provide many sophisticated tools to manipulate that data), the tools available for physical set up in the modeler is limited and do not necessarily match the features in the run-time. Moreover, since constructing complex sets of physical systems (such as ragdolls) can be a daunting task, providing tools that automate and facilitate these and another operations becomes more and more important.

It is for those reasons that the Havok Content Tools include special 3ds Max tools (modifiers, scripts, etc) for setting up physical information in the 3ds Max scene. In the following sections we will explore those tools.

Advanced users should first read the Common Concepts chapter for in-depth details on some of the rigid body and constraint concepts.

Accessing the Physics Tools

The Havok Physics Tools for 3ds Max can be accessed through the **Havok Content Tools** menu and toolbar. If the toolbar is not visible then you should select the *Havok Content Tools* > *Show Toolbar* menu option.



Havok Modifiers in 3ds Max

Most of the Havok physics data added to a scene is done so through the use of special modifiers. While it is possible to add those modifiers to an object by hand (through the modify panel, via scripting, etc.), it is highly recommended that you use the **Havok Content Tools** menus, scripts and toolbar to do so as doing otherwise may leave the scene in an inconsistent state.

For reference, the modifiers used by the Havok Physics Tools for 3ds Max are:

- Rigid Body Modifier: Adds rigid body (dynamics) information to an object.
- Shape Modifier: Adds shape (collision detection) information to an object.
- Constraint Modifiers (Ball and Socket, Hinge, Ragdoll): Add constraint information to a rigid body.

You can remove all Havok modifiers from selected objects at any time by choosing the **Physics** > **Remove Havok Modifiers** option in the **Havok Content Tools** menu.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.3.3.2 Rigid Bodies

In this section we will describe how to create and manipulate rigid bodies and shapes in 3ds Max. For information about concepts and properties associated with rigid bodies and shapes, please check the Rigid Body Concepts section.

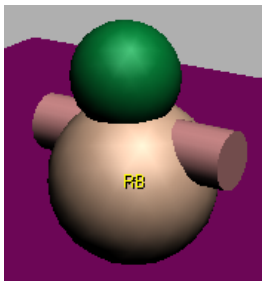
The Rigid Body Modifier

A Havok rigid body modifer defines the root of a rigid body and contains information on the dynamics of the body. Any object which contains a rigid body modifier will be converted to a runtime rigid body by the Create Rigid Bodies filter (provided there is at least one associated shape modifier).

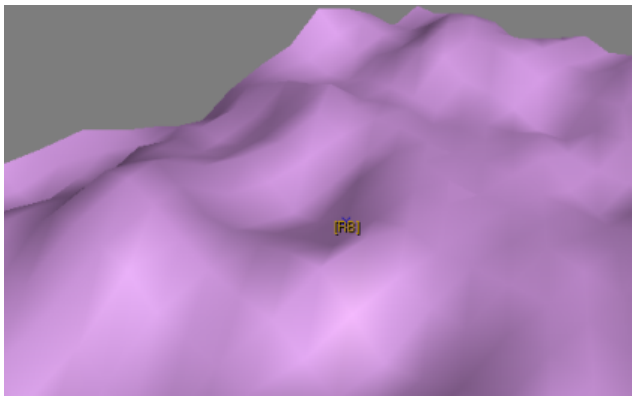
Class Properties

Mark Rigid Bodies is a class property (it applies to all rigid body modifiers). When enabled, rigid bodies (objects with the rigid body modifier) will display a "*RB*" label in the viewports (so they are easy to identify).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



If the rigid body is fixed (ie. it's mass is zero), the label will surrounded by square brackets ("*[RB]*") and will be displayed in a different color:



Note:
The colors used for these labels, and many other Havok elements, are customizable through 3ds Max's 'Customize User Interface...' window.

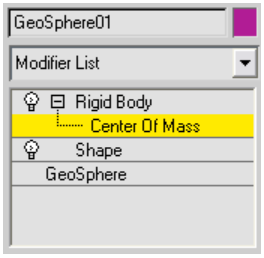
General Properties

Check the common Rigid Body Properties section for details on the properties listed here.

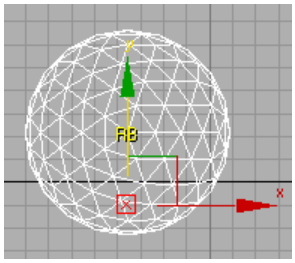
C.O.M. & Inertia Tensor

By default, the center of mass and inertia tensor of the rigid body will be calculated by the Create Rigid Bodies filter based on the rigid body geometry and mass distribution. You can however override this calculation and specify the values within the modeler instead: either by typing the values directly; or by manipulating them in the viewports.

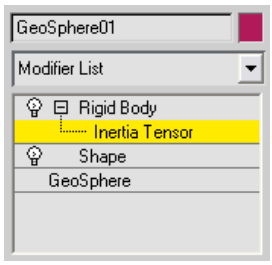
Selecting the **Override C.O.M.** check box enables manual specification of the center of mass. It also adds a new " **Center of Mass** " subobject to the Rigid Body Modifier:



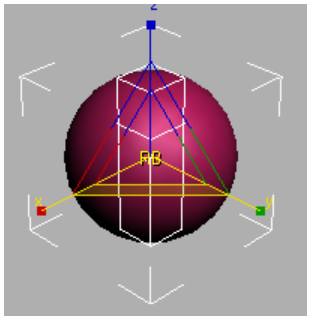
You can select this subobject mode and use the Move tool in order to place the center of mass of the rigid body (displayed as a little crossed square):



Similarly, selecting the **Override Inertia Tensor Proportions** enables manual specification of the inertia tensor proportions. It also adds an " **Inertia Tensor** " subobject to the Rigid Body Modifier:



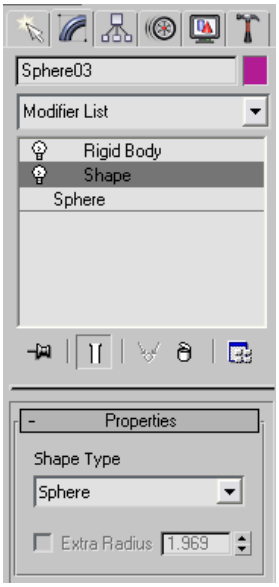
You can select this subobject mode and use the Scale Tool in order to change the proportions of the inertia tensor (displayed as a 3D box):



The Shape Modifier

A Havok shape modifier contains information about how an object's mesh will contribute to the collision detection setup of a rigid body. One or more shape modifiers must be associated with a rigid body modifier in order for the Create Rigid Bodies filter to produce a runtime rigid body.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Check the common Shape Properties section for details on the properties listed here.

Shape modifiers currently have no representation in the viewports.

Creating a Simple Rigid Body

Most of the rigid bodies you will ever want to create will be simple rigid bodies - where both the rigid body and shape information is associated with the same object.

You can create a simple rigid body by selecting an object in 3ds Max and clicking on the **Create Rigid Body** menu option or toolbar button .

This will add both a rigid body modifier and a shape modifier to the selected object.

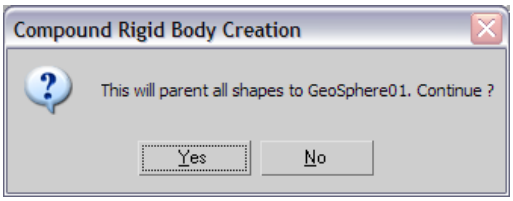
Creating a Compound Rigid Body

A compound rigid body is a rigid body with more than one shape (more than one collision detection primitive). In a compound rigid body one object has both rigid body and shape information while one or more other child objects add extra shape information.

You can create a compound rigid body by selecting two or more objects in 3ds Max and clicking on the **Create Compound Rigid Body** menu option or toolbar button .

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body and shape modifiers while the rest of objects will each hold a shape modifier. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new hierarchy:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

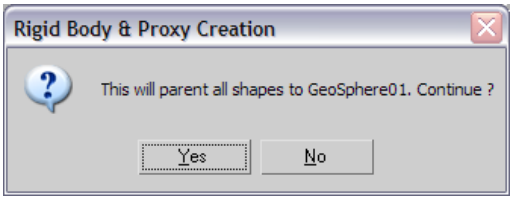


Creating Rigid Body with Proxy(s)

A rigid body with proxy(s) is a rigid body where the node that contains the rigid body modifier doesn't contain any shape modifier - the shape modifiers are stored in one or more child nodes (the proxies). In that sense, a rigid body with proxy(s) is similar to a compound rigid body, with the difference being that the object which has the rigid body modifier has no shape modifier in this case.

You can create a rigid body with proxy(s) by selecting two or more objects in 3ds Max and clicking on the **Create Rigid Body with Proxy(s)** menu option.

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body modifier while the rest of objects will become the proxies and just hold shape modifiers. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new hierarchy:



6.3.3.3 Constraints

In this section we will present how constraints are created and manipulated in 3ds Max.

Constraint Modifiers

A Havok constraint modifier contains information which will be used to create a runtime constraint on export. Since a constraint must act on a rigid body, a constraint modifier is useful only if the object also contains a rigid body modifier. If a pair of rigid bodies are to be constrained, a constraint modifier is attached to only one of the bodies (the *child*), and it maintains a reference to the other body (the *parent*). Again, please consult the common Constraint Concepts section of this manual for more details on child and parent spaces.

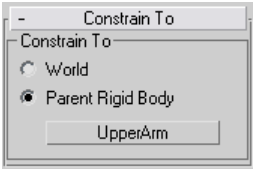
There are currently three types of Havok constraint parameter set - the specifics of which are described in the common Constraint Types section of this manual. Each of these types provides the same basic constraint parameters, which allow you to specify the child and parent spaces. For example, a hinge constraint is shown:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Constrain To Rollout

In this rollout you can choose which other rigid body (if any) the constraint is attached to:

- **World** : If this option is selected, the constraint will attach the child rigid body to a fixed location in space.
- **Parent Rigid Body** : If this option is selected, the constraint will attach the child rigid body to the rigid body specified in the pick node button below (that rigid body will become the parent rigid body).

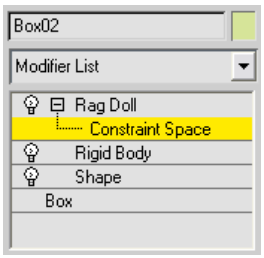


Constraint Spaces Rollout

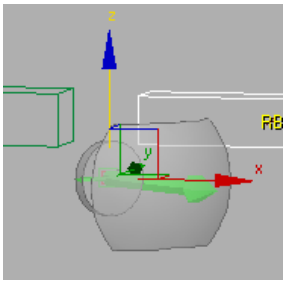
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Constraints can be considered as having two distinct spaces: the *child* (in the local space) and the *parent* (in the parent object's space). See the common Constraint Spaces section for more details.

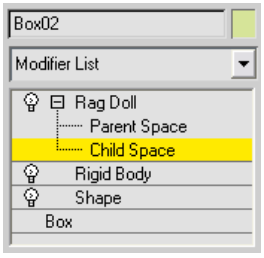
Depending on the state of the **Lock** buttons in the **Constraint Spaces** rollout, various subobjects may be available in the constraint modifier for manipulation. If the translation and rotation of the parent space are locked, then a single subobject appears:



- **Constraint Space Subobject** : In this subobject you can manipulate both child and parent spaces as one, using the move or rotate tool (depending on the type of constraint and the state of the **Lock** buttons).

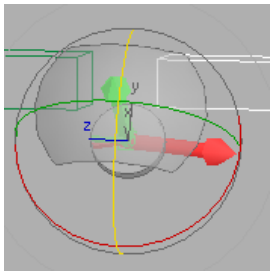


If the parent space translation or rotation is unlocked, two subobjects may appear:



- **Parent Space Subobject** : In this subobject mode you can manipulate the position and/or orientation of the parent space of the constraint.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



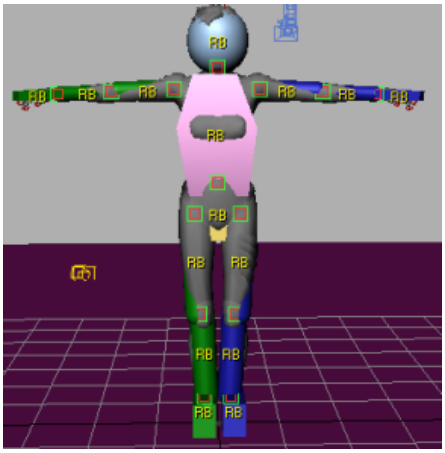
- **Child Space Subobject** : In this subobject mode you can manipulate the position and/or orientation of the child space of the constraint.

Properties Rollout

The properties here are specific to each type of constraint. Check the common Constraint Types sections for details on each constraint type.

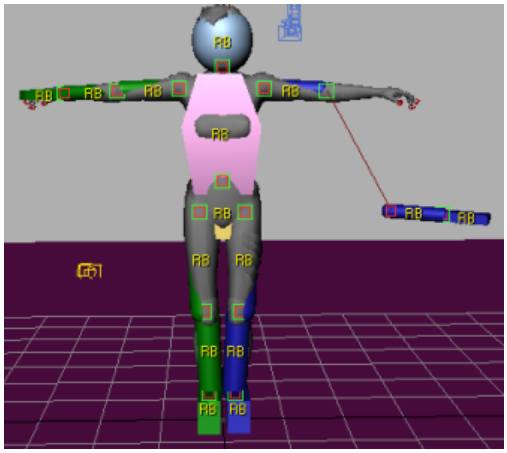
Display

Constraints are displayed in the viewports as two markers, representing the location of the child and parent spaces of the constraint. In most cases, those two spaces should be aligned, so both markers are drawn at the same location:

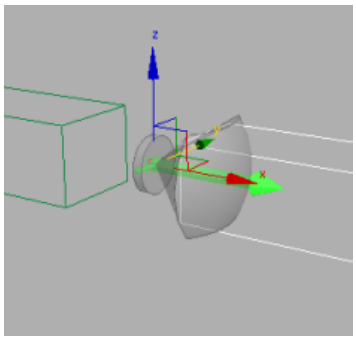


If the spaces are misaligned, a dotted line is also drawn between them to display the offset:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

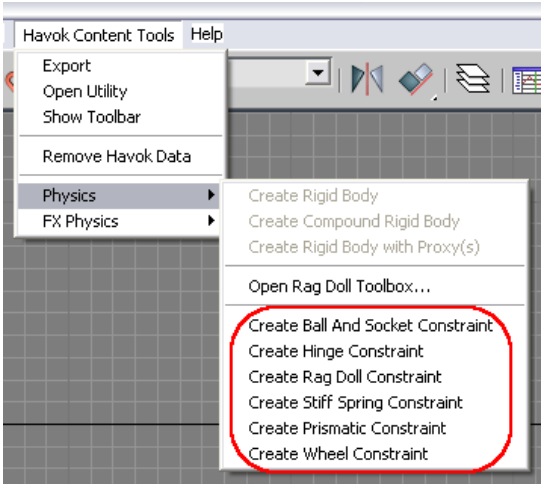


Whenever a constraint is selected in the Modify panel , any 3D representation it may have will be displayed in the viewports, based on it's constraint space and parameter values:



Creating a Constraint

You can create a constraint by using either the **Havok Content Tools** menu or toolbar:



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

You can create a constraint by either having one or two rigid bodies selected:

- If one rigid body is selected, this rigid body will be the child rigid body of the constraint (and the constraint modifier will be applied to it). If it has a parent node and that parent node is also a rigid body, the constraint's parent rigid body will be set to that node; otherwise the rigid body will be constrained to the world.
- If two rigid bodies are selected, and one is a descendant of the other, the descendant will be the child rigid body of the constraint (the modifier will be applied to it) and the ancestor will be the parent rigid body. If they are not explicitly parented, the last rigid body selected will be the parent and the first rigid body will be the child.

A single rigid body can have any number of constraints applied to it.

6.3.3.4 The Rag Doll Toolbox

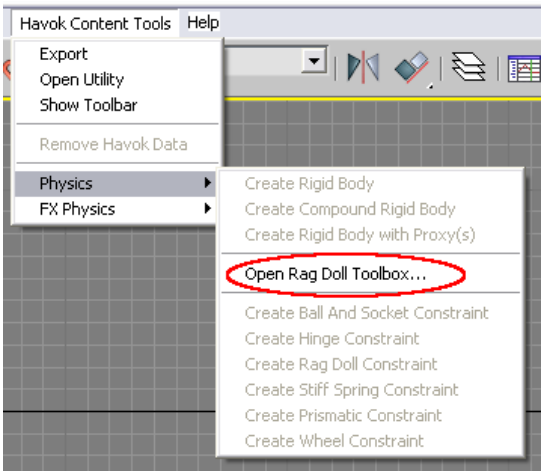
Introduction

One of the most common usages of rigid body simulation is the creation of articulated systems of rigid bodies representing characters (rag dolls). This usually involves the creation and alignment of rigid body proxies for the bones, as well as the setup and alignment of constraints that represent the joints in the character. The Rag Doll Toolbox provides a set of tools to facilitate those operations. It is built as a MAXScript tool that works on top of the lower-level physics tools.

See also: Customizing the 3ds Max Rag Doll Toolbox

Opening the Rag Doll Toolbox

You can open the Rag Doll Toolbox by using either the **Havok Content Tools** menu or toolbar:



The Rag Doll Toolbox is a MAXScript floating window. It can also be docked to either side of the 3ds Max UI:

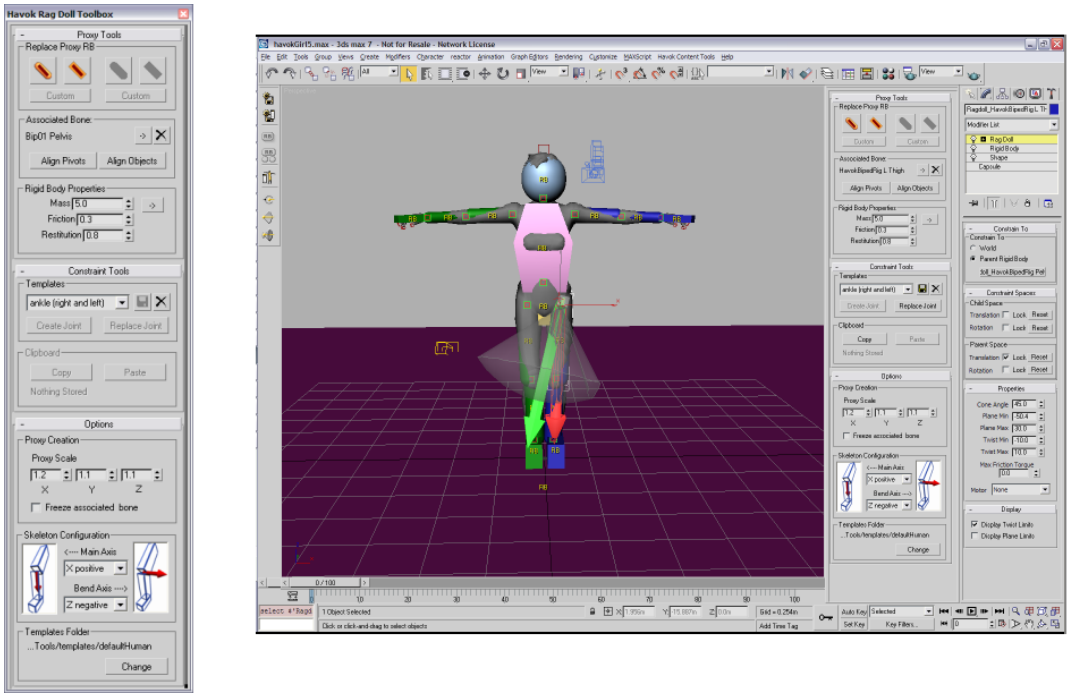


Figure 6.7: The 3ds Max Rag Doll Toolbox UI

Proxy Tools

The first task when creating a rag doll representation of a character is to construct a set of rigid bodies which represent a selection of the the character's bones. These rigid bodies are usually collision primitives like capsules and boxes, which are very efficient to simulate by the physics runtime (capsules in particular).

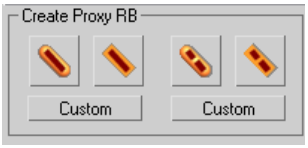
When creating these rigid bodies, it is very important that their pivots (the location of their transforms) are aligned to the bones they represent (since, at run-time, movements on the bones will have to be converted to movement on the rigid bodies and vice versa). Thus, it's very important to keep track of the association between bones and the proxies that represent them.

The **Proxy Tools** rollout allows you to:

- Given a bone, create a proxy representation (a box, a capsule, or a custom rigid body) of it and align and associate the proxy with the original bone.
- Given a chain of bones, create a single proxy representation of that chain and align and associate that proxy with the chain of bones.
- Given a proxy rigid body (associated with a bone or chain of bones), replace it with another proxy.
- Keep track of the association between rigid body proxies and bones, and easily realign them.

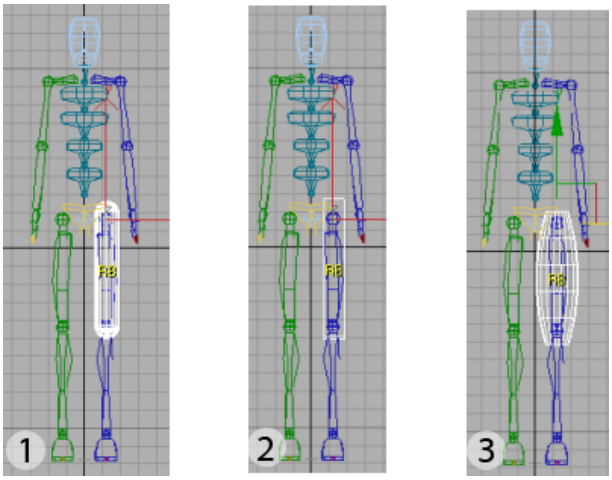
- Quickly access and modify the basic properties of one or multiple rigid body proxies (mass, restitution and friction).

Create/Replace Proxy RB



Buttons in this group box allow you to create proxies and associate them with bones.

The first set of buttons (on the left) create *simple proxies* - rigid body proxies that are associated with a single bone:



1. *Create Capsule Proxy(s)*

   With one or more bones selected, this will create a capsule rigid body bounding each of the selected bones. It will also associate and align each capsule with each bone.

2. *Create Box Proxy(s)*

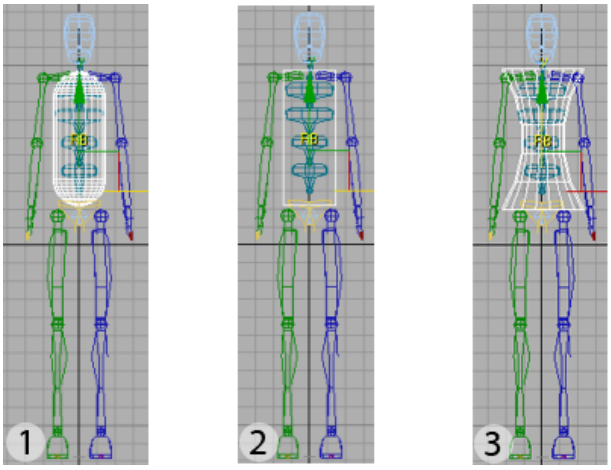
   With one or more bones selected, this will create a box rigid body bounding each of the selected bones. It will also associate and align each capsule with each bone.

3. *Associate Custom Proxy*

   With a single bone selected, this will open a " **Select by Name** " dialog where you can select any mesh object to become the proxy rigid body for the selected bone. It will then associate and align that rigid body with the selected bone.

The second set of buttons (on the right) create what we call *chain proxies* - rigid body proxies that are associated to a chain of bones (a set of bones in hierarchical order). Chain proxies are useful in order to create single proxies that represent multiple bones, like a rigid body bounding multiple spine links:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



1. *Create Capsule Chain Proxy*

   With a chain of bones selected, this will create a capsule rigid body bounding all of them. It will also associate and align the capsule with the chain of bones.

2. *Create Box Chain Proxy*

   With a chain of bones selected, this will create a box rigid body bounding all of them. It will also associate and align the box with the chain of bones.

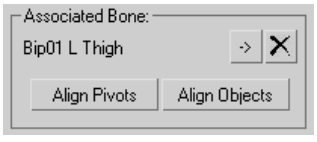
3. *Associate Custom Chain Proxy*

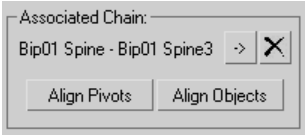
   With a chain of bones selected, this will open a "Select by Name" dialog where you can select any mesh object to become the proxy rigid body for the chain of bones. It will then associate and align that rigid body with the chain.

When a rigid body proxy (either single or chain proxy) is selected, these "Create Proxy" buttons become " **Replace Proxy** " buttons - the proxy they create will replace the selected proxy (and the association with the original bone will be updated).

Bone Associations

Controls in this box display and operate on associations between rigid body proxies and bones. Thus, they are only enabled when one or more rigid body proxies (rigid bodies created by one of the "Create Proxy" options above) are selected.





Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

If a single proxy is selected, the name of the associated bone (or chain of bones, if it's a chain proxy) will be displayed below the **Associated Bone / Associated Chain** box.

Pressing the **Select Associated Bone(s)** button will select the bone or chain of bones associated with the selected proxy.

The **Remove Association** button will remove the current association between the selected proxy and the bone or chain of bones. The proxy will remain as a rigid body, but it's association with the original bone(s) will be lost.

The **Align Pivots** button will ensure that the pivot of the proxy matches the pivot of the associated bone (or the first bone of the associated chain) by changing the local pivot of the proxy (without actually moving the proxy). Use this button whenever you have modified a proxy and want to make sure that the pivots are aligned (without displacing the proxy)

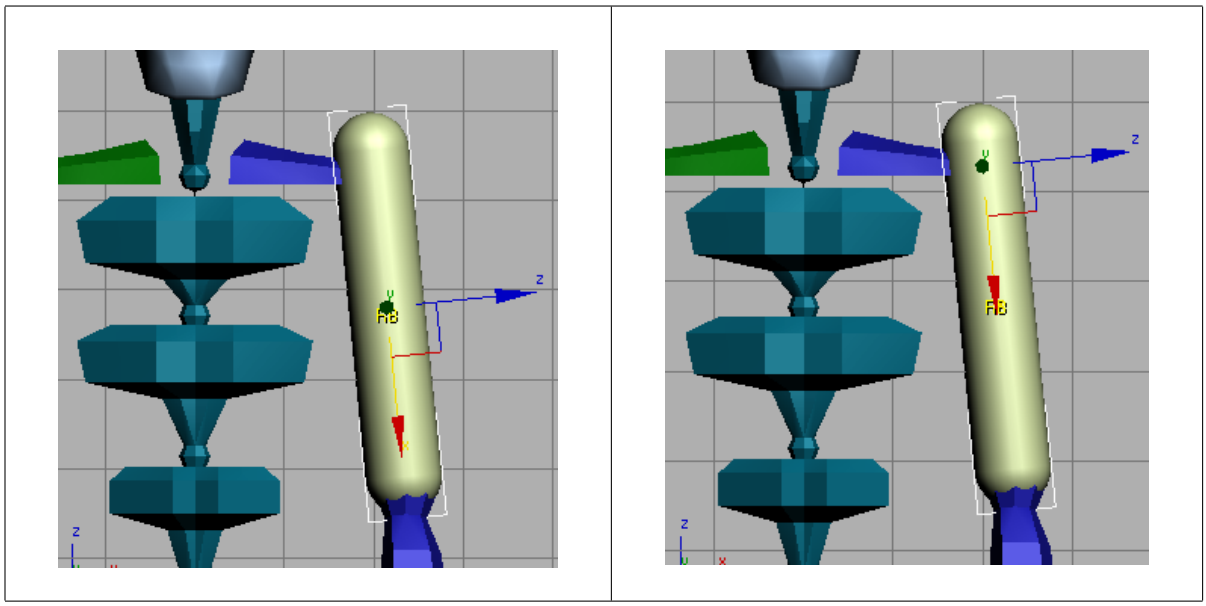
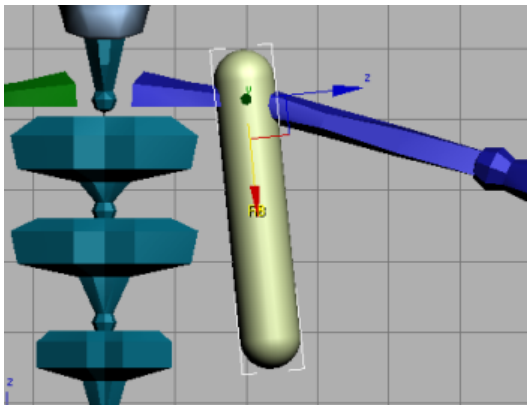


Table 6.1: Aligning Pivots

The **Align Objects** button will ensure that the pivot of the proxy matches the pivot of the associated bone by moving the proxy rigid body. Use this button whenever you move or rotate a bone and want to make sure that the associated proxy follows that movement.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



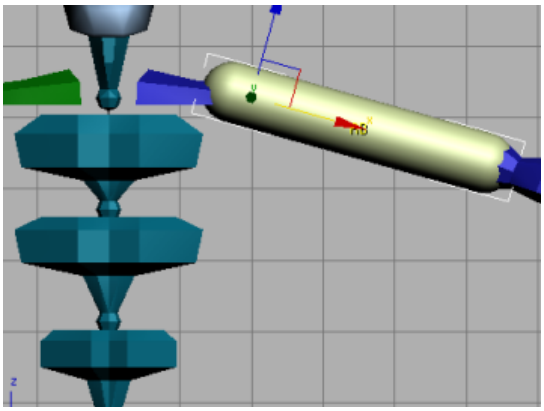
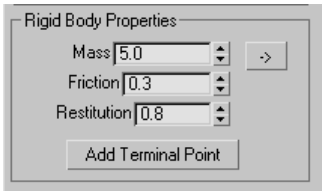


Table 6.2: Aligning Objects

Rigid Body Properties

The controls in the **Rigid Body Properties** box are enabled whenever one or multiple rigid bodies (proxies or not) are selected.



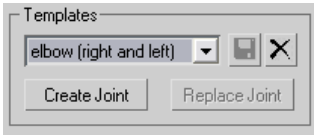
They allow you to modify the basic rigid body properties of the selected objects easily - they are particularly handy when you want to modify multiple rigid bodies at the same time.

The **Open RB Modifier** button will open the rigid body modifier associated with the selected rigid body in the Modify panel (so it can be edited there as well).

The **Add Terminal Point** button is used to add extra leaf bones to skeletons to be used for raycasting, specifically to prevent ragdoll penetration with landscapes using the hkaDetectRagdollPenetration utility. Selecting a ragdoll proxy and clicking the add terminal point button creates a point helper as the child of the proxy. The helper is located on the local bounding box of the proxy at the center of the face furthest from the proxy pivot.

Constraint Tools

The **Constraint Tools** section of the Rag Doll Toolbox gives you the ability of reusing the setup of constraints by using *templates*. A template is a representation of Havok constraint which is stored as a text file. The file includes the type of constraint, plus any parameter values which the original constraint had when the template file was created.



The **Templates** dropdown list show all the available templates. These list is built by looking at all templates stored in the current *templates folder* , the location of which can be changed in the Options rollout of the Rag Doll Toolbox.

The **Save Template As...** button is enabled only when a rigid body with a single constraint is selected. It allows you to save the parameters of that constraint as a template. Templates are saved as `*.txt` files.

The **Remove Template** button will permanently remove the selected template (i.e., it will delete the template file from the templates folder).

The **Create Joint** button is only enabled when two rigid bodies are selected. It will create a constraint based on the currently selected template. If the rigid bodies are parented, it will apply the constraint to the child rigid body. If they are not parented, but they are proxies of bones that are parented, the rigid bodies will be first parented accordingly, and then the constraint will be applied to the child. Otherwise, the user is required to manually parent the rigid bodies before creating the joint.

The **Replace Joint** button is only enabled when a rigid body with a single constraint is selected. It will replace that constraint with a new one based on the selected template. The rigid body will be constrained to the same parent it was originally constrained to.

Clipboard

The Constraint Tools also allow you to cut & paste constraints without explicitly storing them as templates:

The **Copy** button is enabled only when a rigid body with a single constraint is selected. It will store the parameteres of that constraint in memory (the type of constraint stored is displayed)

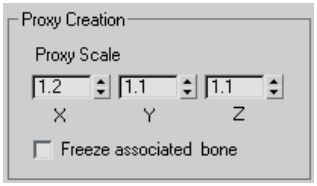
The **Paste** button operates as the **Create Joint** and **Replace Joint** buttons (depending on the selection), with the difference that it applies the template stored in the clipboard rather than the template selected in the drop down list.

Rag Doll Toolbox Options

This rollout contains general options that apply to both the Proxy Tools and Constraint Tools. These options are saved with the scene, and are presented in three sets:

Proxy Creation Options

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



The **Proxy Scale X, Y and Z** values are used during the creation of bounding proxies (capsule, box) for a bone. The size of the box/capsule will be calculated by taking the bounding box of the bone and then scaling it by the given values.

This is particularly useful when the size of the bones may not properly represent the full volume of the character part / limb they represent.

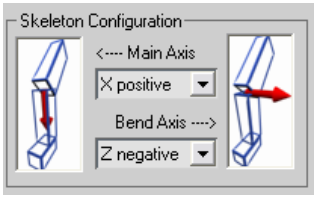
Also, it is usually desirable to have some overlap between bones, so there are no gaps at the joints. This is why it is useful to have some extra scale in the direction of the main axis of the bone (by default X).

When the **Freeze Associated Bone** option is on, creating a proxy from a bone will automatically freeze the original bone. This can be very useful in order to reduce the amount of selectable objects in the viewport during proxy creation operations (so bones that already have proxies are no longer selectable in the viewports).

Skeleton Configuration Options

When storing and applying constraint templates (check the Constraint Tools section for details), it is important to have a convention for the local axis of the bones.

Most 3ds Max tools, including Bones and Character Studio, follow some convention on the local orientation of their bones. The default convention used by the Rag Doll Toolbox is that of Character Studio, but should your tool of choice have a different convention, the options here allow you to define it:



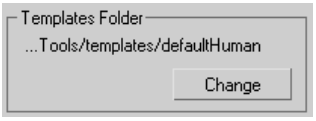
The **Main Axis** option should specify the name and sign of the axis (in the local space of the bone) that points towards the length of the bone (to the next bone), outwards. The default is X positive.

The **Bend Axis** option should specify the name and sign of the axis (in the local space of the bone) around which a positive (counter clockwise) rotation bends the joint inwards. The default is Z negative.

Templates Folder Options

Templates available to the Constraint Tools are collected from the list of available template files in an specific folder. This folder is defined in the **Templates Folder** options box.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



You can change the location of the current templates folder by clicking on the **Change** button and selecting a different folder. This allows you to have different templates folder for different types of characters.

6.3.3.5 The Convex Hull Tool

In this section we describe a simple tool for generation of convex hulls from selected meshes. This tool generates an approximate convex hull from the currently selected objects (either meshes or nurbs) with a user-specified maximum number of vertices **Max verts** . The hull is approximate in the sense that it does not necessarily enclose the selected meshes, but is a reasonably good approximation (using the metric of volume difference) which satisfies the constraint on the maximum number of vertices. This is useful in a number of contexts, such as generation of assets for platforms which require <64 vertices per convex body. If the specified maximum exceeds the number of vertices in the true hull, the tool just generates the true hull. Note that if multiple meshes/nurbs are selected, the hull tool generates a single hull from all of their vertices combined.

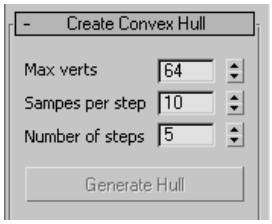
The approximate hull is found by first computing the original set of vertices in the true hull. If the number of vertices in the original hull is N, and the desired maximum number of vertices in the generated hull is M, the tool has to remove R=(N-M) vertices. In the case when the number of steps equals 1, a trial is done which consists of randomly choosing a sample of R of the true hull's vertices, and computing the hull obtained by removing that sample from the original set. A user-specified number **Samples per step** of such trials are done (with a default of 10), and the trial hull obtained which had the minimum difference in volume from the true hull is taken as the final choice. This constitutes one 'step'.

Rather than removing all R vertices in a single step, the user can specify that the the vertices be removed in a sequence of X steps, where X is the parameter **Number of steps** . On each step roughly R/X vertices are removed (if X exceeds R, each step removes only one vertex). Increasing the number of steps will clearly lead to hulls which better approximate the true hull, but will take longer to finish. The default is 5 steps.

The convex hull tool utility simply consists of fields for the three parameters **Max verts** , **Samples per step** , and **Number of steps** described above, and a **Generate Hull** button to generate the hull using these parameters. Generated hulls are placed in the scene root with the name "foo_Hull01" where "foo" is the name of the first object in the selection. Subsequently generated hulls accumulate in the scene root but are given unique names.

On clicking Create Hull, a progress bar appears which allows the hull generation to be aborted if it is taking too long. On aborting, the best hull at the current point in the computation (i.e. the hull from the current step with minimal volume difference so far) is generated.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



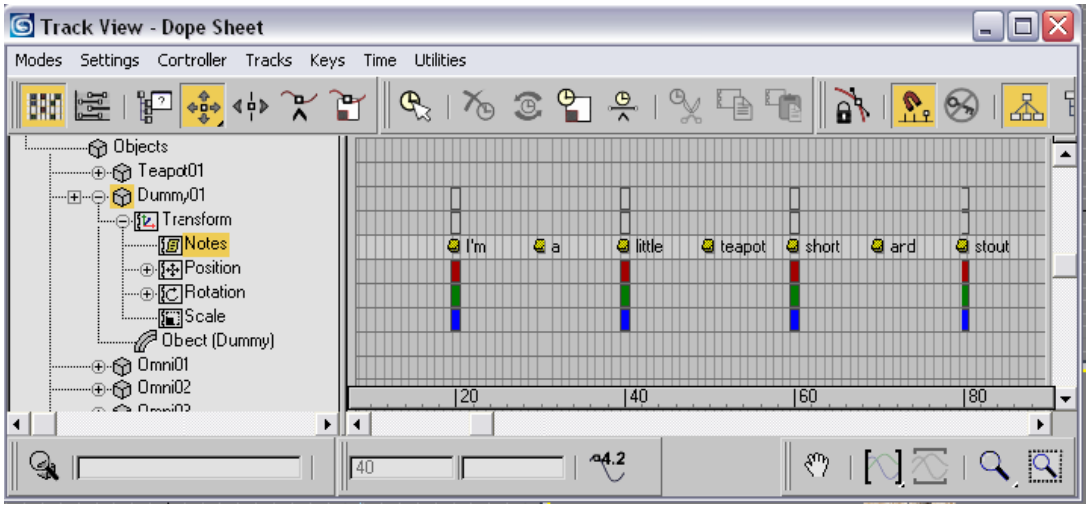
6.3.4 3ds Max: Animation Tools

3ds Max already provides a wealth of tools for creating content for animation playback. The Havok Content Tools use these features provided by 3ds Max without the need to provide custom extensions (as it does for the physics). Thus concepts like skeletons, bones, skins and animations in 3ds Max transfer to the same concepts/objects in the Havok SDK. The following section outlines one of the less obvious connections between 3ds Max and the Havok Animation SDK.

6.3.4.1 Annotations in 3ds Max

Please check the common Annotations section for details on annotations in the Havok Animation SDK.

3ds Max has built-in support for what is called "note tracks", which work in very similar fashion to annotations. Therefore, in order to create annotations in 3ds Max, simply attach note tracks to the desired bone - the string values will be exported as annotations. The 3ds Max user reference manual contains a detailed section on setting up note tracks.

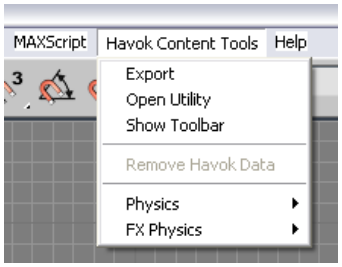


6.3.5 Tutorial: Export and Animation Basics

In this tutorial we are going to explore the basic operations involved in exporting and processing an asset. We are going to use the example of exporting skeletons, animations, etc.. but the principles regarding how assets are exported, the filter pipeline and the use of the filter manager and individual filters for processing applies to all Havok products.

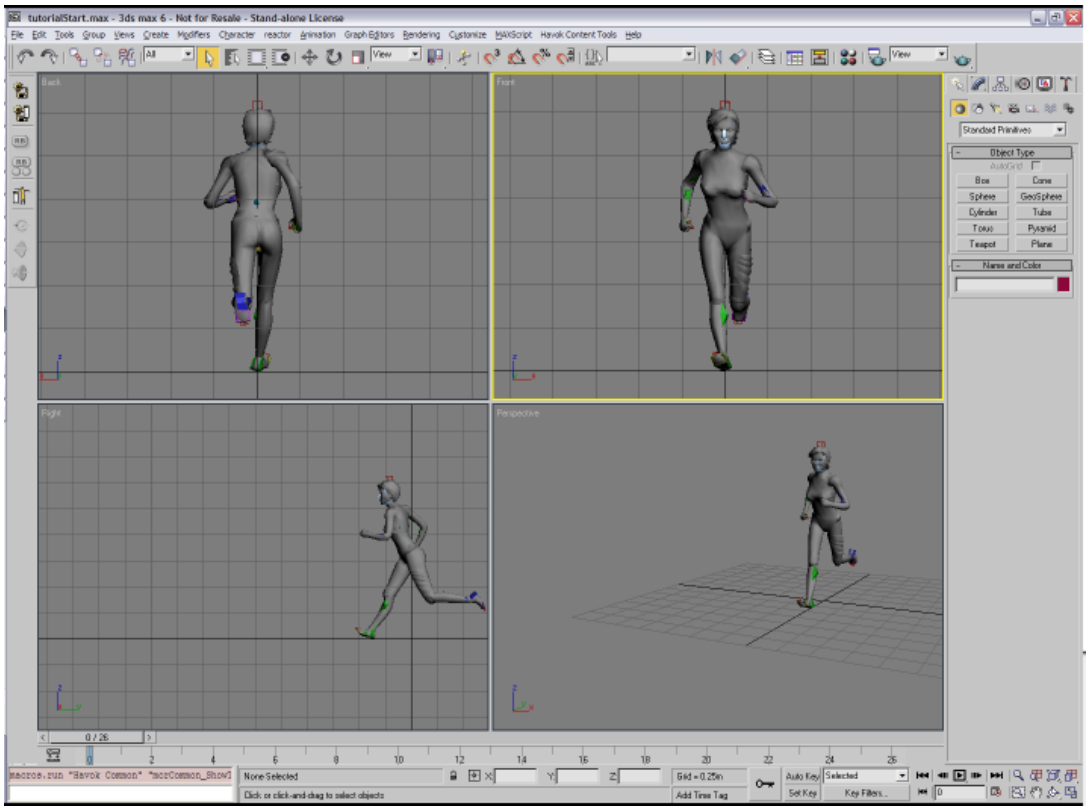
6.3.5.1 Getting Started

Ensure that you have installed the Havok Content Tools for your version of 3ds Max. You should see a **Havok Content Tools** menu in the 3ds Max menu bar:



The Havok Content Tools toolbar will be useful for this and the next tutorials, so it is useful to have it in the UI. If it is not present, click on **Show Toolbar** in the **Havok Content Tools** menu. You can dock the toolbar to one of the sides, or leave it floating.

Open the file `"scenes/havokContentTools/tutorials/exportBasics/tutorialStart.max"`:



The scene contains the animation of a girl running (a single cycle). If you examine the scene, you will notice that the animation is constructed by using a set of animated Character Studio bones which modify a mesh skinned with the Physique modifier.

We want to export this information (bones, animation, skin weights, etc) from 3ds Max into objects that the Havok Animation/Complete SDK can interpret.

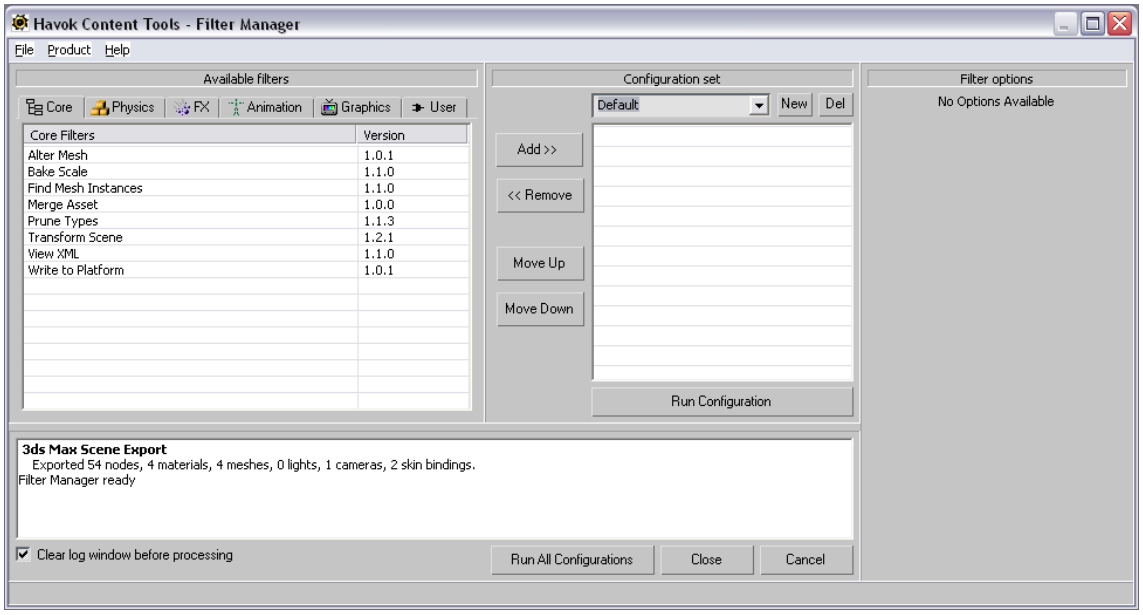
Note:

Even if you are not a user of Havok Animation or Havok Complete, you can still follow this tutorial as the Havok Content Tools packages contain filters for all products. This tutorial focuses on introducting the basic concepts behind asset export and processing so it is relevant for all Havok products.

6.3.5.2 Exporting the Scene

Exporting the current scene is as simple as to click on the **Export** button in the **Havok Content Tools** menu or toolbar.

By doing this the 3ds Max Scene Exporter is invoked - the scene is navigated and relevant information is pulled from the 3ds Max nodes, modifiers, meshes, etc.. and converted into a modeler-independant format. After that, that content is passed to the filter manager - it's UI should appear after a short time:

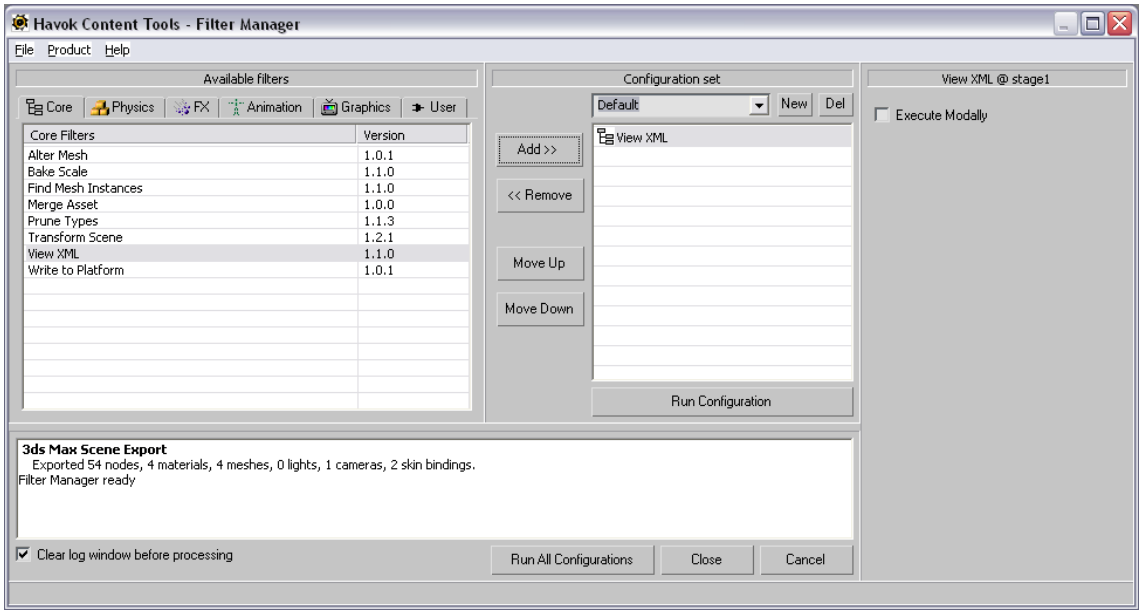


The **Configuration Set** window (in the middle of the dialog) lists the filters that will be applied to the content we just exported. It is now empty, which means that nothing will happen to the content. You can add filters by selecting them from the **Available Filters** tabs and clicking on the **Add** >> button.

Let's start by adding a very useful filter for debugging purposes: the View XML filter.

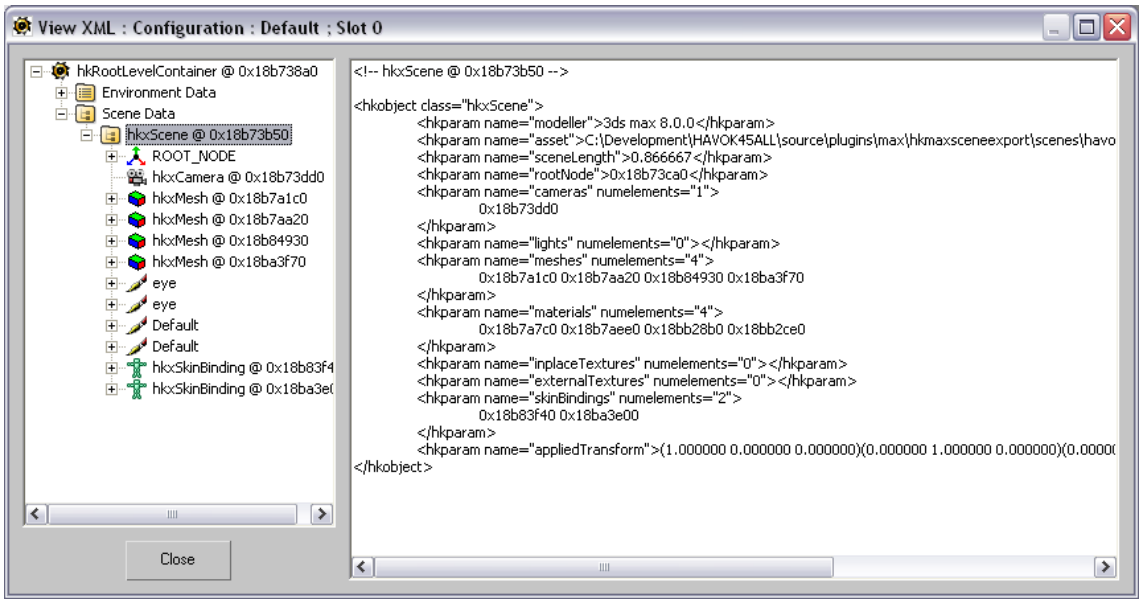
- On the **Available Filters** window, click on the **Core** tab (if it's not selected already)
- Double-click on the **ViewXml** filter (or select it and press **Add** > >)

- That filter should now appear at the top of the **Current Filters** list



Filters are executed in the order they appear in the **Current Filters** list; if a filter modifies the content, that modified content is what is used by the next filter (check the filter pipeline documentation for details).

The View XML filter doesn't modify any content, though. It just opens a window that presents the current content in a human-readable XML format. Let's check what we exported from 3ds Max by clicking on the **Run Configuration** button.



Notice that what was exported is an "`hkxScene`" object, which contains information about nodes, meshes, materials and skin bindings. This is what the 3ds Max Scene Exporter extracted from our current scene,

and is the raw information that we are going to process further until we can create a useful set of objects ready to be used by our run-time.

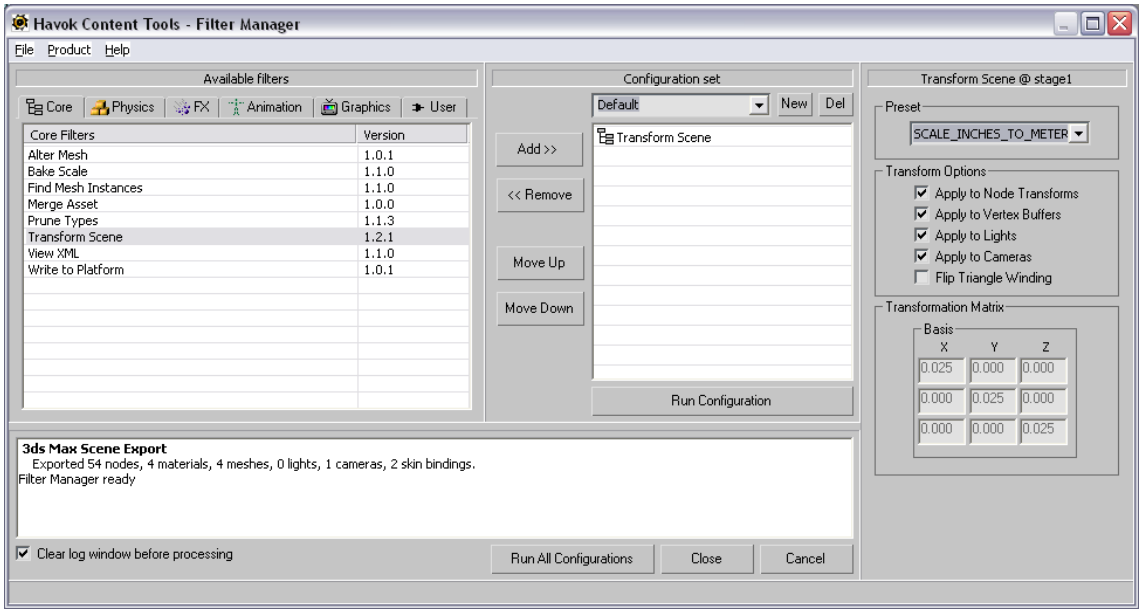
Close the XML window and get back to the filter manager. Remove the View XML filter by selecting it and clicking on << **Remove** or hitting **DEL** .

Tip:

Feel free to add the View XML filter at different locations during this and other tutorials in order to examine how processing is done by different filters. This filter is a very useful debugging tool.

6.3.5.3 Processing the Scene

Let's now add some useful filters for processing the scene. The first filter that we'll add will be the Scene Transform filter (from the **Core** tab):



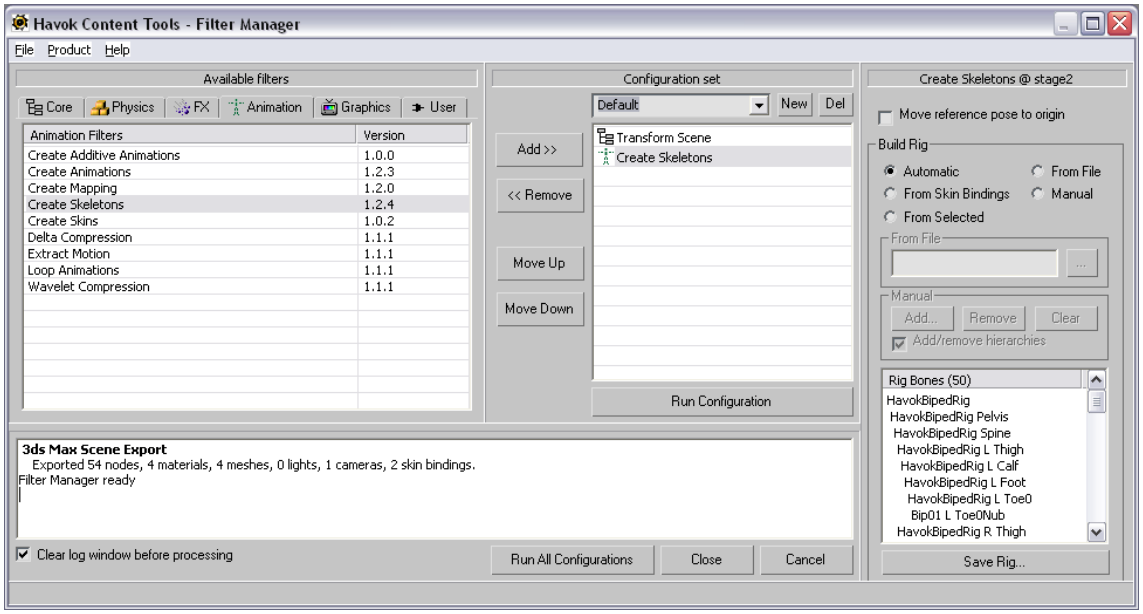
This filter is very useful and, in the majority of cases, it will be the first filter that you'll add to any filter configuration. This filter applies an arbitrary transform to all the objects (nodes, meshes, etc) in the scene. In this case, we want to apply it because in 3ds Max the default units are inches, but in our run-time units are meters. Hence we use the **SCALE_INCHES_TO_METERS** preset in the options of the filter (the right panel in the filter manager).

The next filter we will apply is the Create Skeleton filter (from the **Animation** category). This filter takes a look at the nodes in the scene and creates an hkaSkeleton object (used by the Havok Animation/Complete SDK).

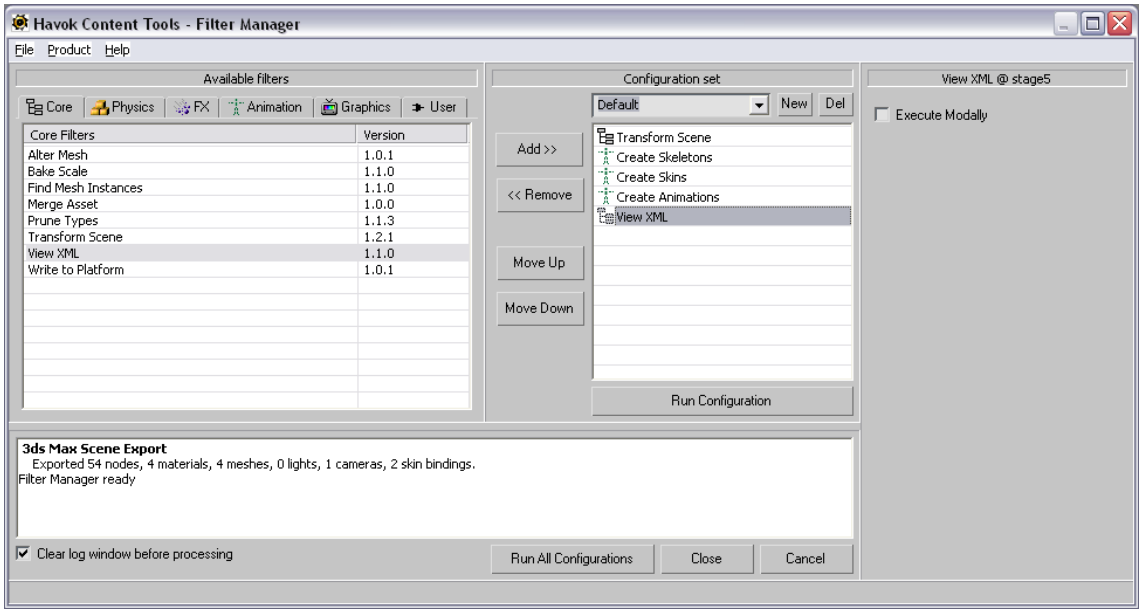
Note:

If your current Product Selection is set to Havok Physics, some of the filters will appear in red, and when run, warnings will be generated. You can either ignore those warnings, or change your Product Selection temporarily to Havok Complete.

As you can see, this filter also has options associated with it (they appear in the right panel).

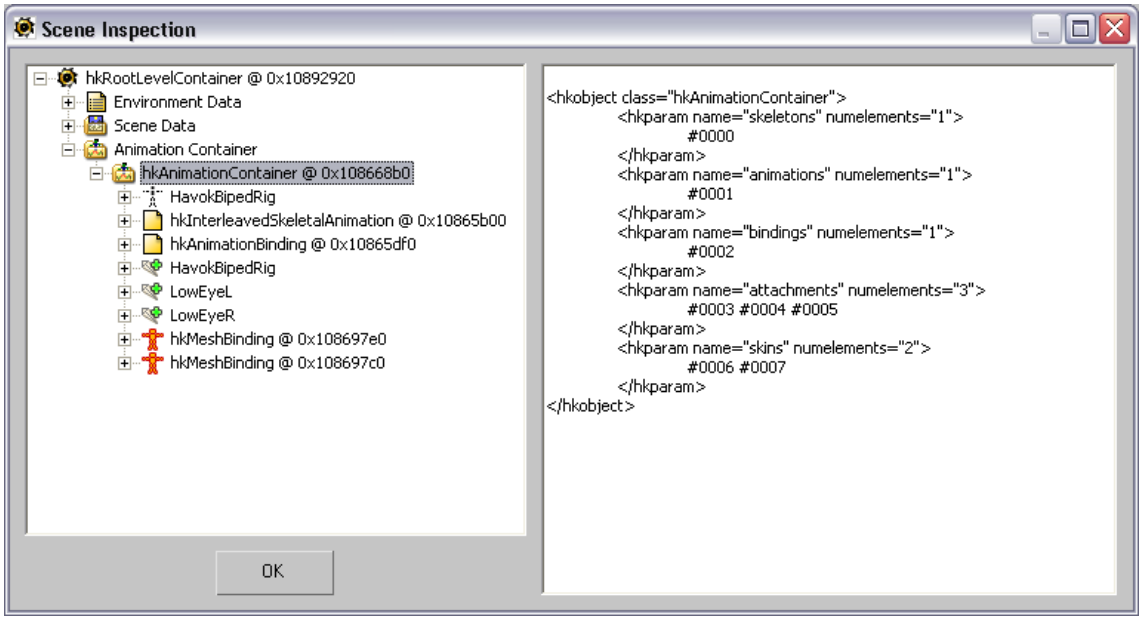


We will leave the default options as they suit us fine. You can check the documentation for this and other filters if you want to learn more about how they work. After this filter, add the Create Skin and Create Animations filters (also in the **Animation** category) - leave the default options as well. Finally, place the View XML filter again at the bottom of the filter list.



If you run (press the **Run Configuration** button) the filter processing, the View XML filter will show you the effect of all the processing:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

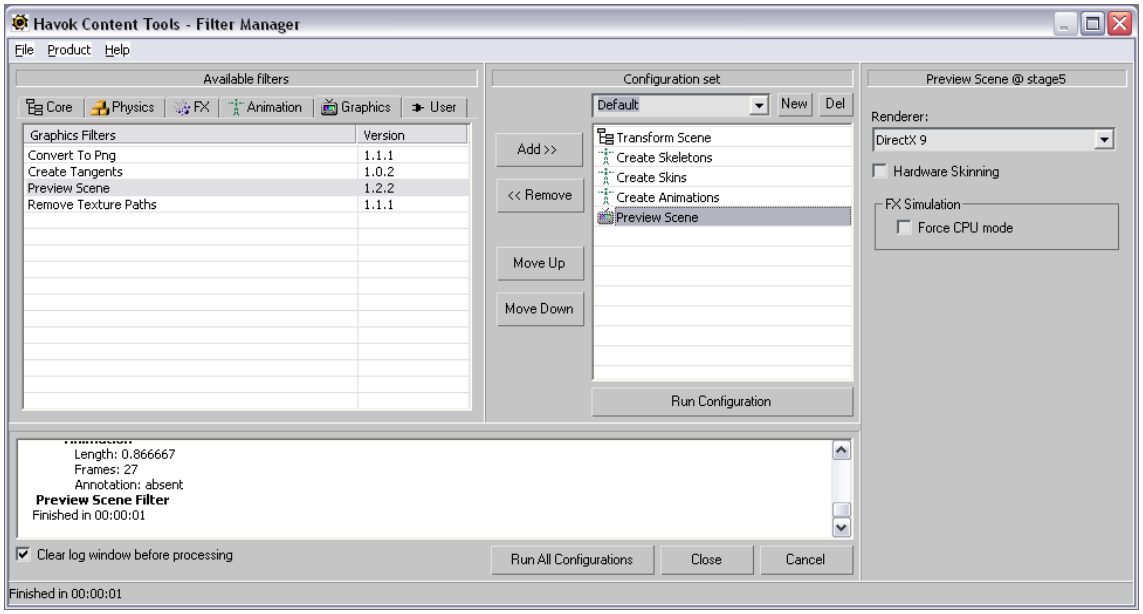


Notice how a new root-level object (an animation container) is now part of the content, and how it contains Havok Animation/Complete objects like `hkaInterleavedSkeletalAnimaton`, `hkaSkeleton` (HavokBipedRig), `hkaMeshBinding`, `hkaBoneAttachment`, etc.. that we should be able to use at run-time.

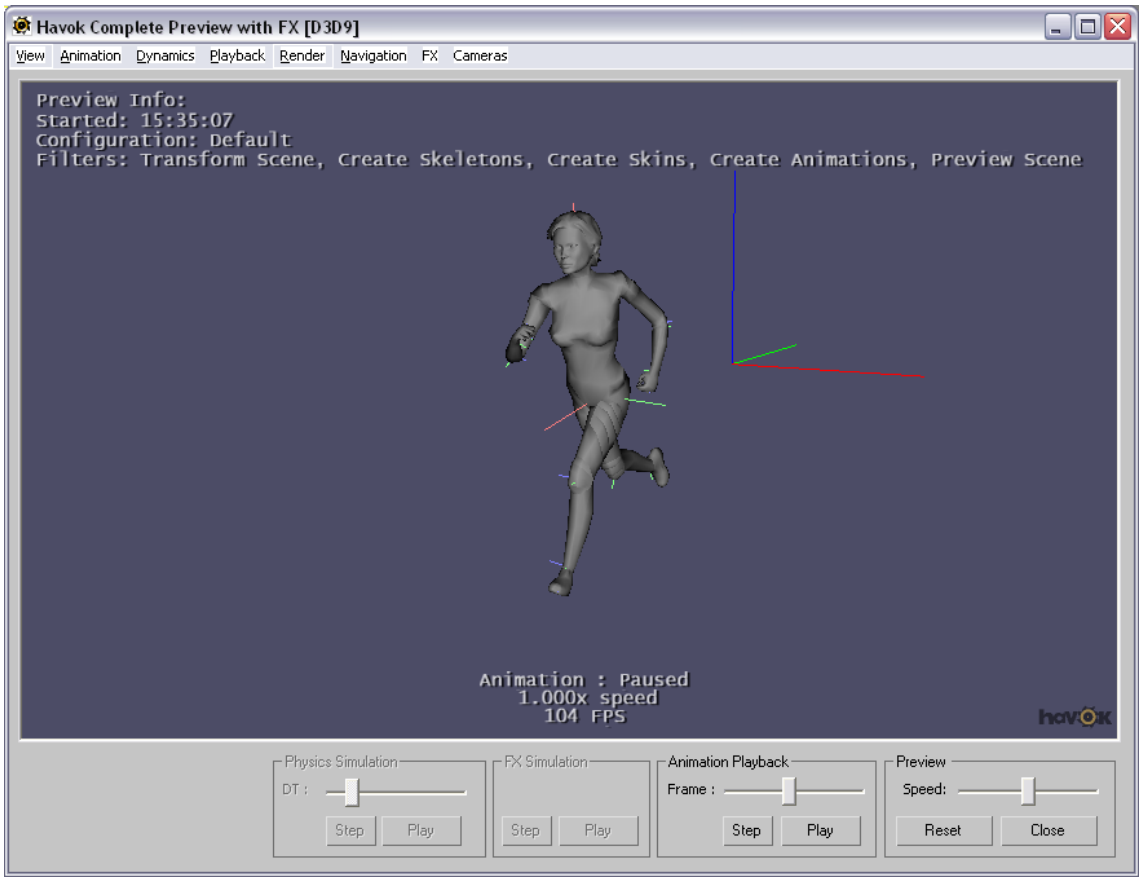
6.3.5.4 Previewing the Scene

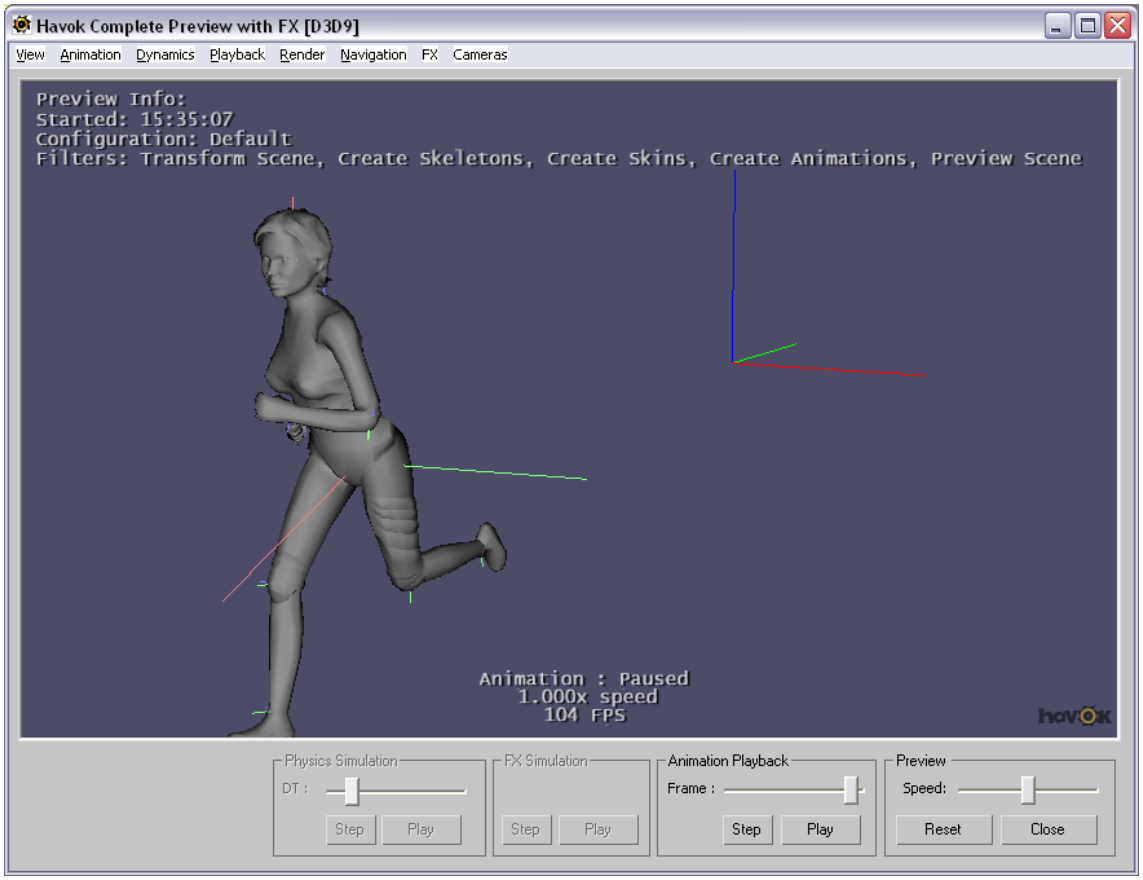
Although examining the contents by viewing the XML output give us an indication on how our processing went, in most cases visual inspection is the only way to ensure that the processed content matches our expectations. Havok provides the Preview Scene filter (in the **Graphics** category) just for that - this filter is built with the Havok run-time, including some display libraries, and will play back any Havok content it finds.

Replace the View XML filter at the end of the filter list with a Preview Scene filter, and click again on **Run Configuration** :



A modeless window will appear with our animation and skinning running in real-time:



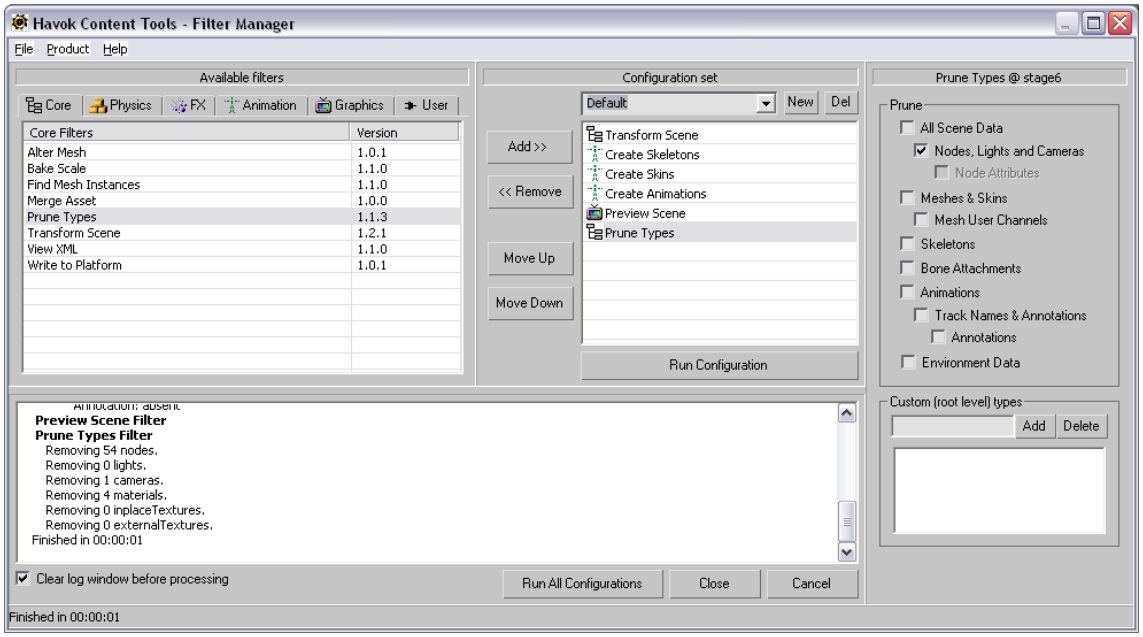


The Preview window gives you multiple options to change the speed of playback, the current frame in the animation, etc.. Check the Preview Scene filter documentation for more details.

6.3.5.5 Pruning Some Data

Most of the filters either modify (eg. the Scene Transform filter) or add (eg. the Create Skeleton filter) information to the asset. Sometimes it is useful to remove some of the data in the assets that is no longer relevant (particularly at the end of the processing). For example, once we've created our skeletons and animations, the original data regarding nodes and their keyframes is probably no longer needed (since it has been transformed into other classes).

We can add a Prune Types filter (in the **Core** category) after the Preview Scene filter. Leave the default options as they are - this will remove nodes (and their keyframes), lights and camera information



Note:

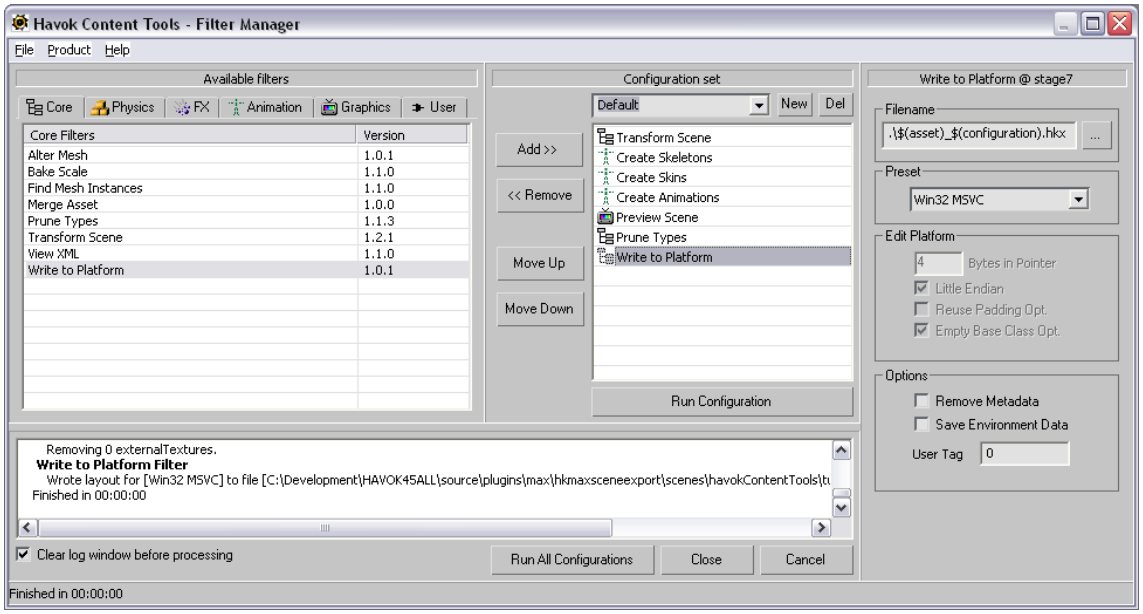
You could also add this filter before the Preview Scene filter - but since the Prune Types filter will remove camera information, the preview will have to use a default camera (rather than reflect the camera in the original scene).

6.3.5.6 Writing the Processed Contents

So far, all the processing done by the filters has happened in memory, and hasn't been written to any file.

The Platform Writer filter (in the **Core** category) will serialize the processed content to a file. It can write either XML format or binary format for many different platforms. Add the Platform Writer filter at the end of the filter configuration. By default it will save a binary .hkx file for the Win32 platform in the same location as the asset.

If you now process the asset again (click on **Run Configuration**), you will observe in the log window that a file has been written:



The processed asset is now saved alongside our 3ds Max file, ready to be loaded into the Havok SDK.

This finalizes this first tutorial: you can find the final 3ds Max scene in the "`tutorialEnd.max`" file.

We'd like to encourage you to experiment with different filters and filter options. By checking the results in the filter processing log, as well as the output of the View XML filter, you should be able to get a good idea of what each filter does.

If you are a Havok Animation/Complete user, the following sections go a little further in processing this scene by doing some motion extraction. If you are a Havok Physics user, skip the following section and go to the Physics Basics Tutorial.

6.3.5.7 Appendix (Animation/Complete Only): Extracting Motion

The animation we just exported has some *motion* on it - in between the first and last frame of the animation there has been some displacement on the character (it has moved forward).

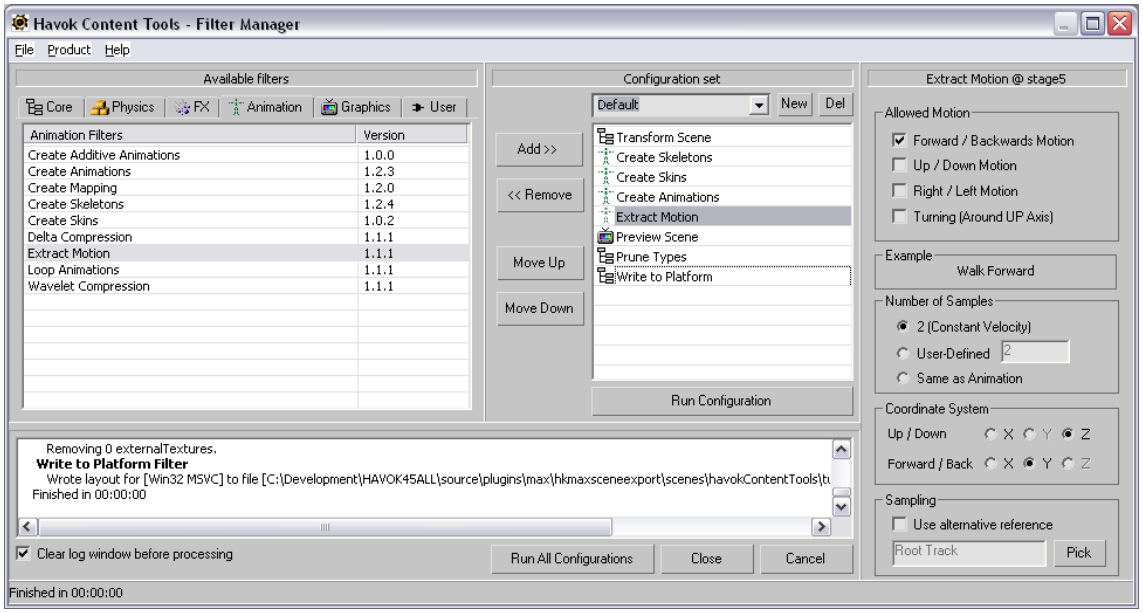
Motion extraction, explained in detail in the *Havok Animation* section of this manual, is the process on which we examine an animation and divide it into two - a motion/displacement component (the moving forward during running) and a local animation component (the movement of arms, legs and body during the run cycle).

After motion extraction, thus, our running animation should become a run-on-the-spot animation, complemented by some information on how the character should displace when that animation is applied (the extracted motion).

In this appendix to the tutorial we are going to do motion extraction to our run animation.

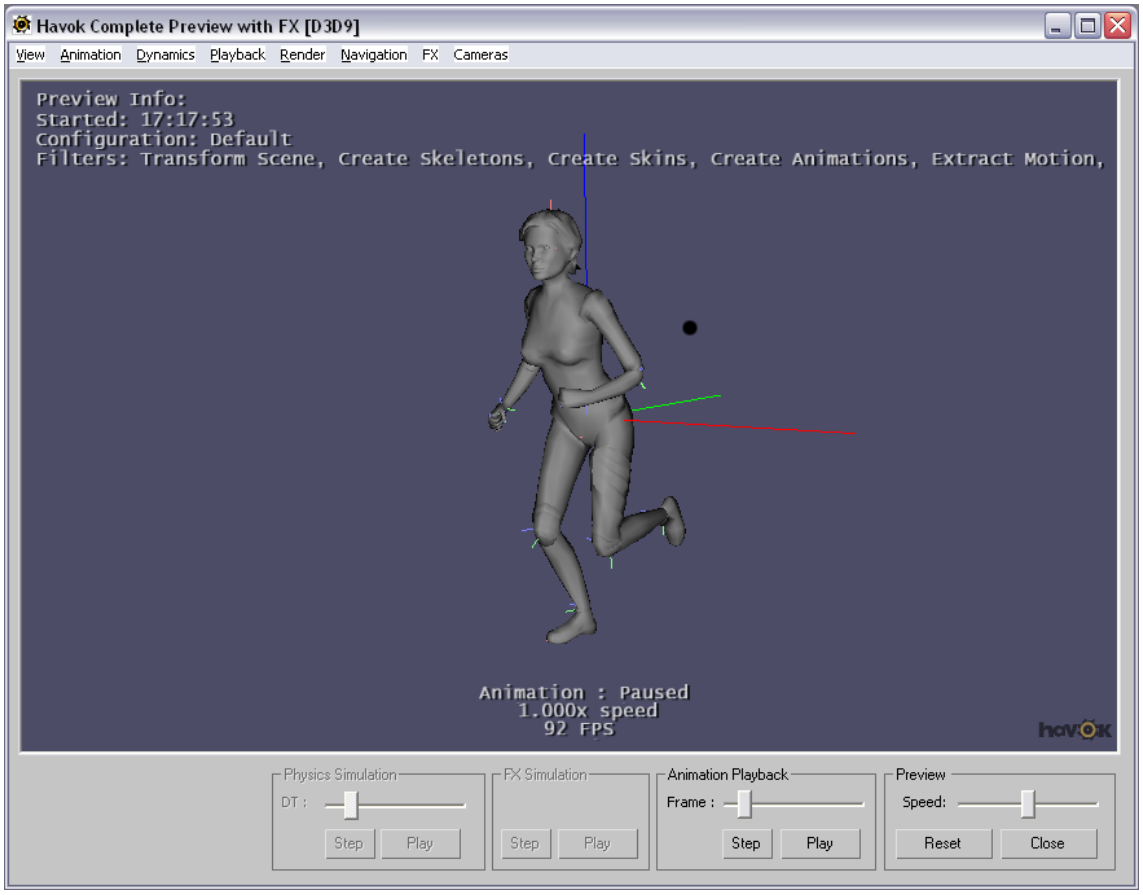
- Open the filter manager again (if you have closed it) by clicking on **Export**

- Add the Extract Motion filter (in the **Animation** category) to the filter list, just after the Create Animations filter (use the **Move Up** and **Move Down** buttons to place the filter in the right place).

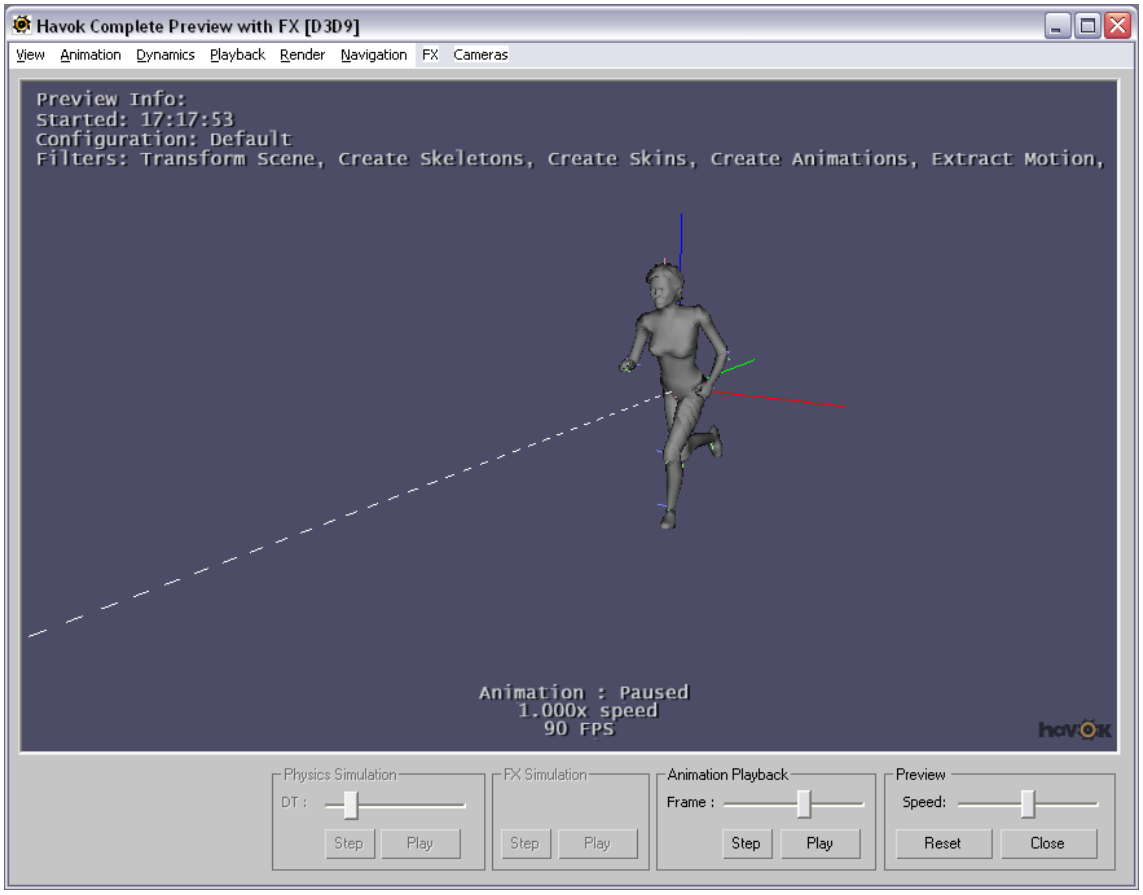


The filter options allow you to specify what kind of motion you want to extract from the animation. In our case the motion is a forward motion so we'll leave the **Forward/Backwards** check box on. But, since in our animation the forward direction is Y (the character is running in that direction) we need to specify that in the **Coordinate System** options (the default was X).

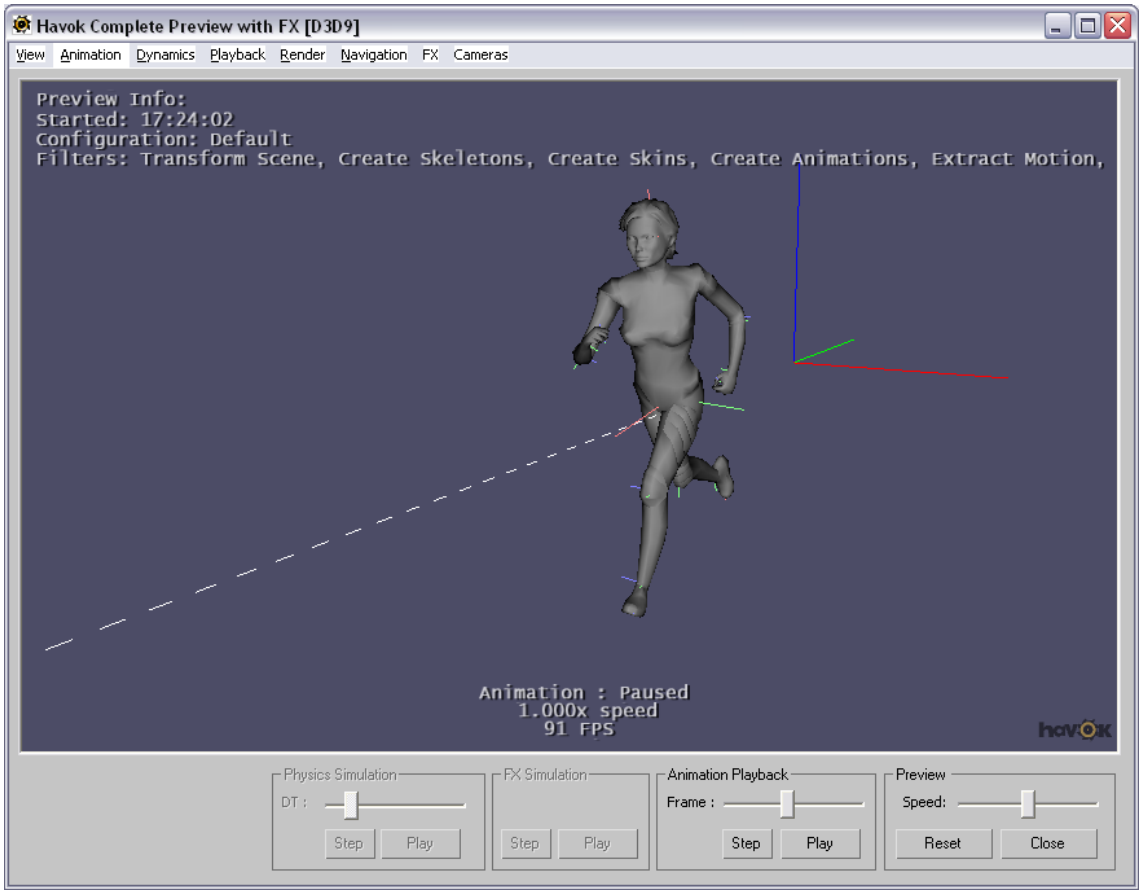
If we now run our filters, it will open the Preview dialog - you can see how now our animation is now a run-on-the-spot animation:



The Preview Scene filter knows about motion and it allows you to visualize it. In the preview dialog, click on **View** > **Extracted Motion** :



A dashed line is displayed showing the motion extracted from the animation. The preview can also apply and accumulate the extracted motion. In the **Animation** menu, click on **Accumulate Motion** :



Notice how the character now moves according to the motion that was extracted - notice also how that motion accumulates: the character moves forward continuosly over animation loops (instead of warping back to the starting position at the beginning of each loop).

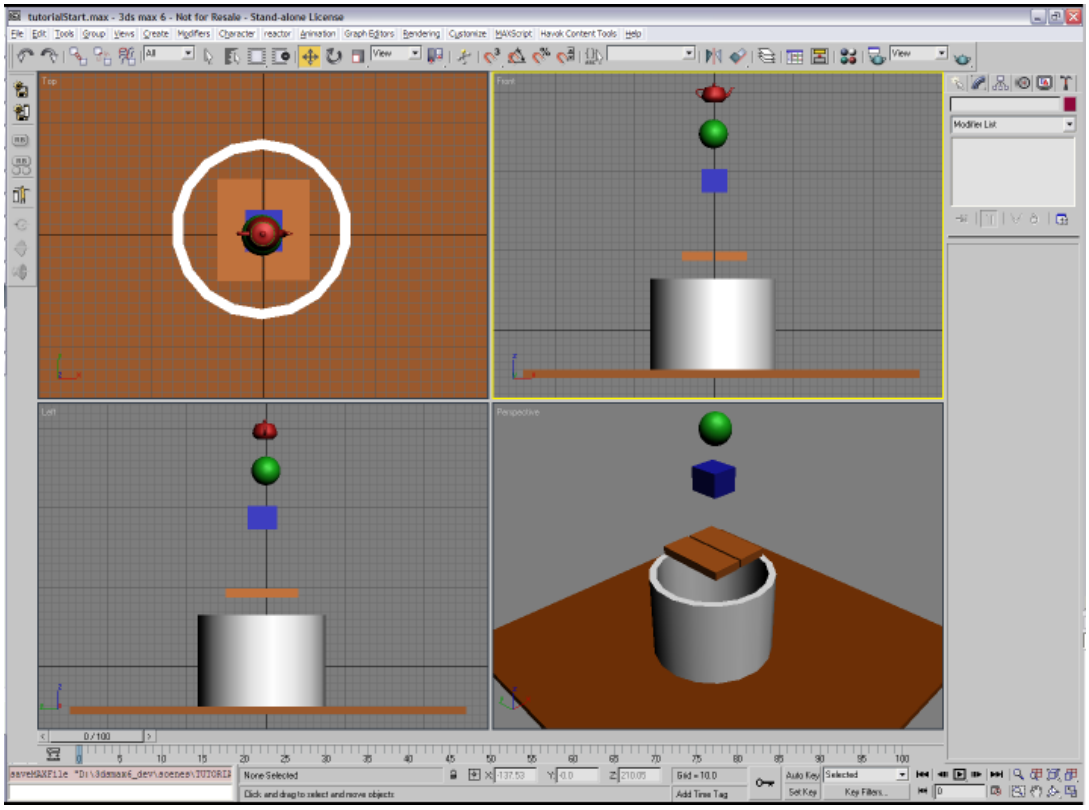
You can find the tutorial scene with motion extraction applied in the file "`tutorialEnd_MotionExtracted.max`".

6.3.6 Tutorial: Physics Basics

In this tutorial we are going to explore the basic operations for seting up rigid bodies and constraints in 3ds Max, and processing them through the filter pipeline. This tutorial assumes that you are familliar with the basics of exporting and processing scenes, covered by the Export And Animation Tutorial. We also recommend that you take a look at the rigid body and constraint concepts sections of this documentation.

6.3.6.1 Getting Started

Start by loading the scene "`tutorialStart.max`", located in the "`scenes/havokContentTools/tutorials/physicsBasics`" folder.

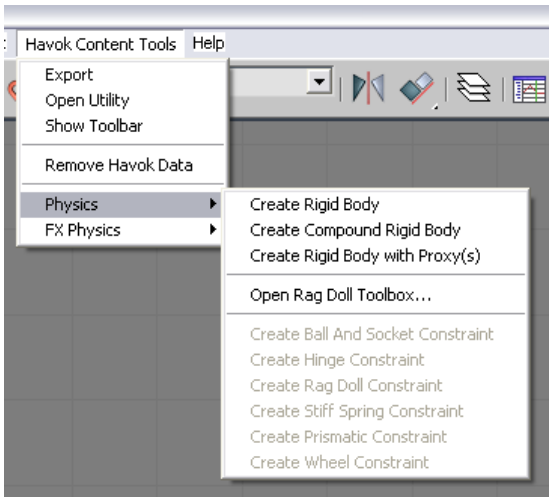


In this scene we have a set of objects: a teapot, a sphere, a box, a trap door, a can and a ground. We will attach rigid body properties to them so they behave as they would in real life.

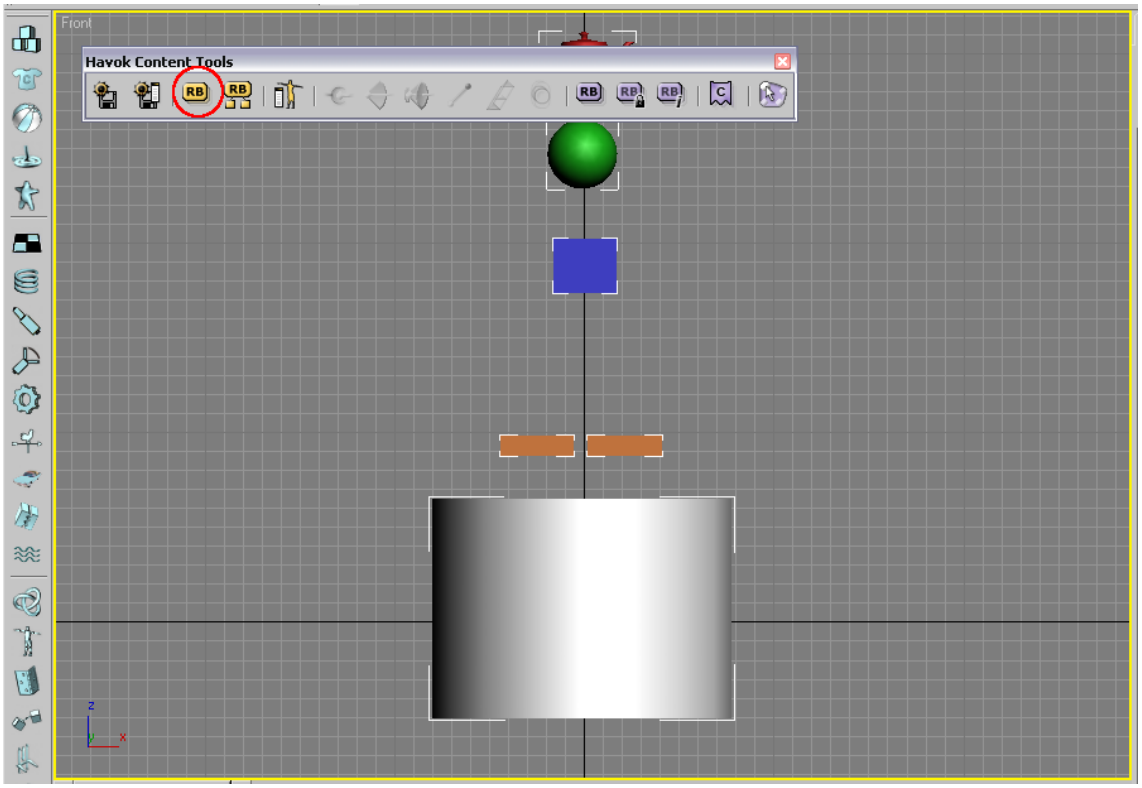
6.3.6.2 Setting Up Rigid Bodies

To start with, we need to make these objects become rigid bodies, i.e., attach rigid body information to them so we can later on create rigid bodies and simulate them using the Havok SDK.

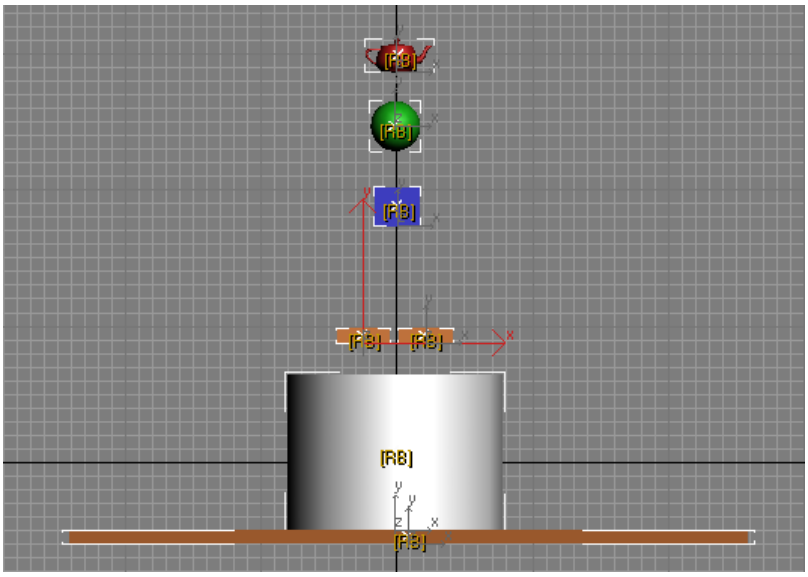
The Havok Content Tools provide a set of physics tools in 3ds Max in order to do so easily. These tools are available through the **Havok Content Tools** menu and toolbar:



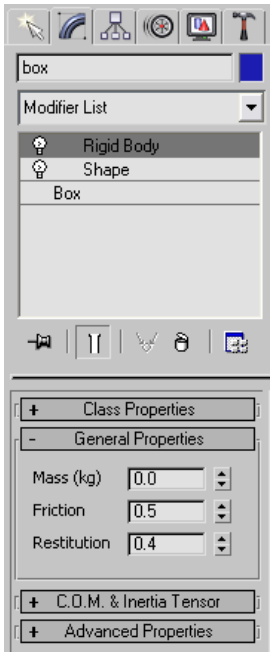
In our case, we want to create a rigid body out of every single object, so let's start by doing so. Select all the objects in the scene, and click on the **Create Rigid Body** button or menu option:



Notice how all the objects in the scene become labeled with the text " **[RB]** ":



This means that those objects have now rigid body properties (and are currently fixed bodies). In 3ds Max, rigid body properties are stored in modifiers. For example, if you select the blue box and then go to 3ds Max's Modify panel , you should see two modifiers applied to the box:



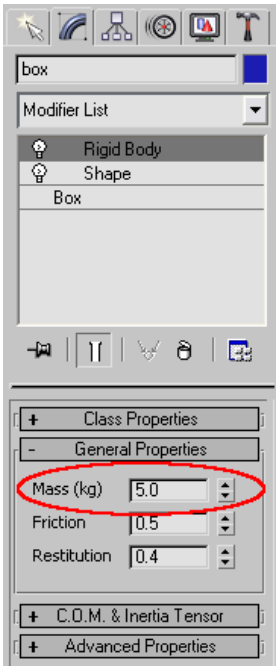
The rigid body modifier contains information about the dynamic behaviour of the object - mass, friction and restitution are specified here. The shape modifier contains information about the representation of the object for collision detection.

We will be exploring these modifiers a little bit more later on. For now, notice how the mass of the box is currently zero - this is the default mass for new rigid bodies. Since Havok will consider a mass of zero to be an indication of the rigid body being fixed in space (not moving), we want to make sure that we

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

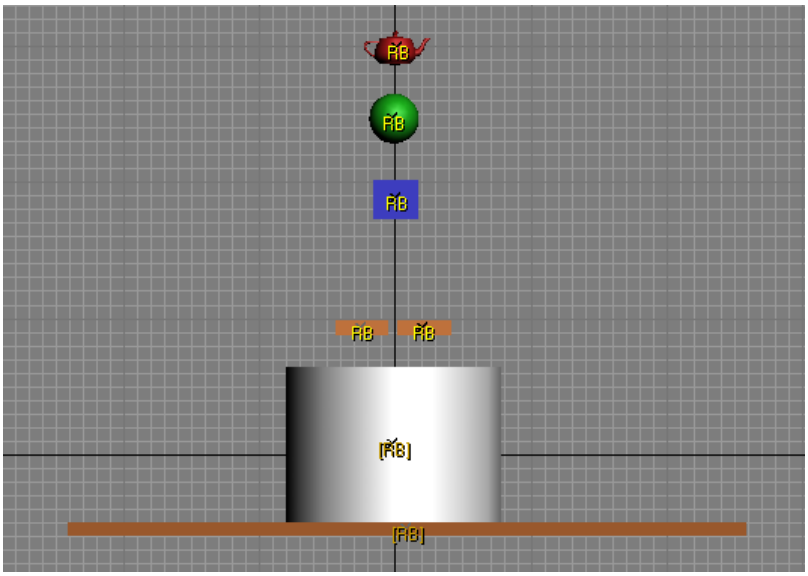
specify a mass for all the dynamic objects in our scene - that would be the teapot, the sphere, the box and the trap doors.

Set the **Mass** property to a value different than zero (5Kg for example):



Then, without leaving the Modify panel , do the same for the sphere, the teapot and each trap door (select each one of them individually and set the mass to 5).

After doing this, we have a scene setup with 7 rigid bodies, 5 of them dynamic/movable and 2 of them (the can and the ground) fixed:

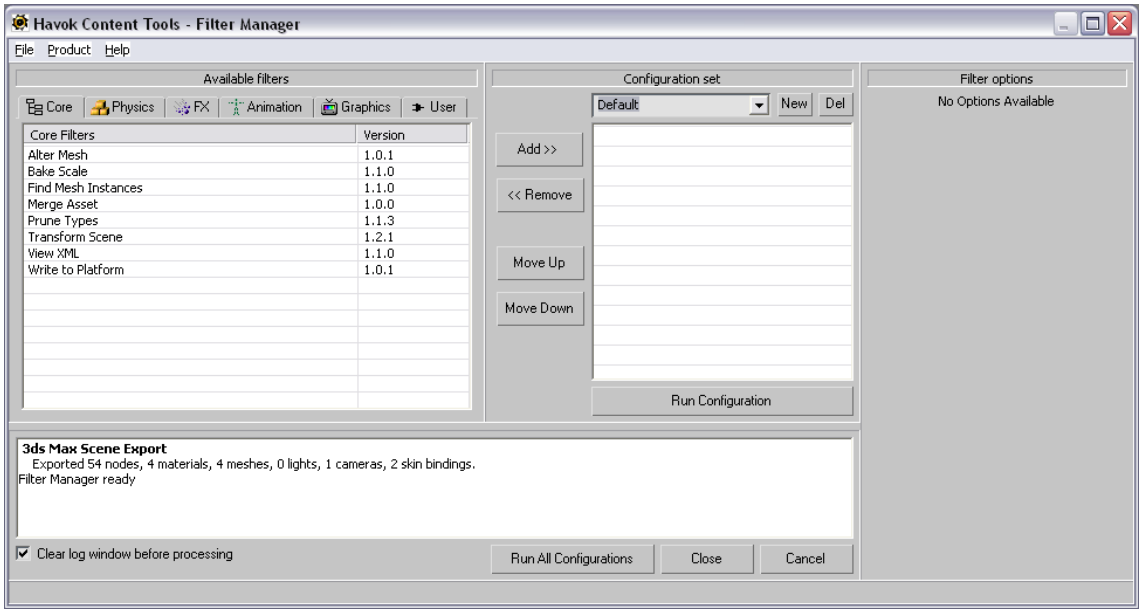


Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.3.6.3 Previewing the Scene

Now that we have some content set up, we'd like to preview it and check that everything behaves as expected. In the previous tutorial we saw how to export, process and preview assets using the 3ds Max scene exporter and the filter pipeline. We are now going to do the same, but this time using some processing specific to physics assets.

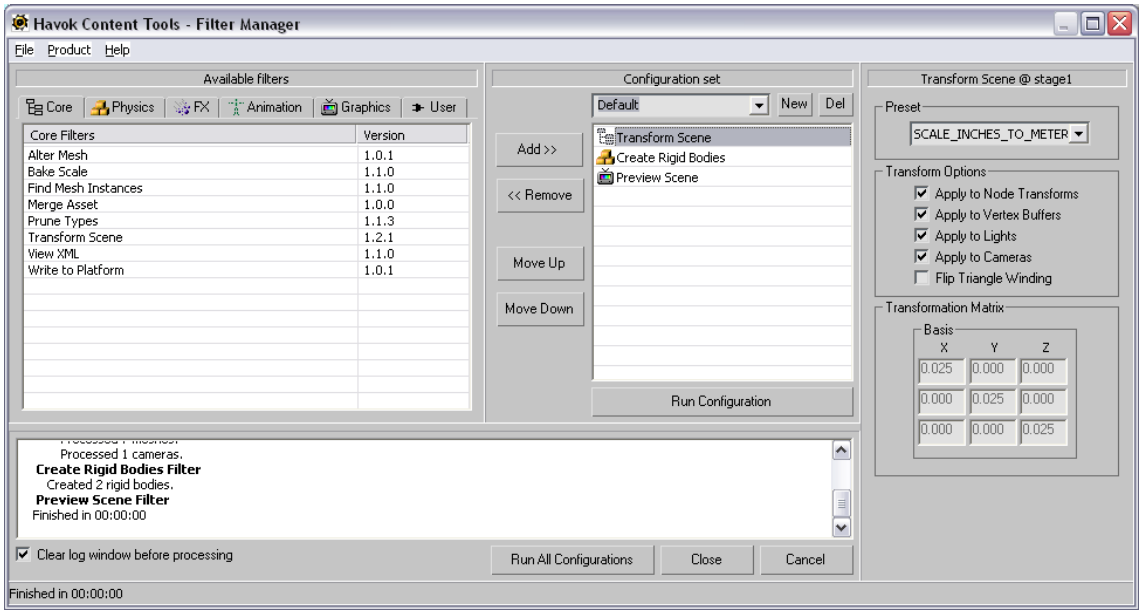
Start the export and processing by clicking on the **Export** button or menu option. This should open the filter manager, with an empty filter setup:



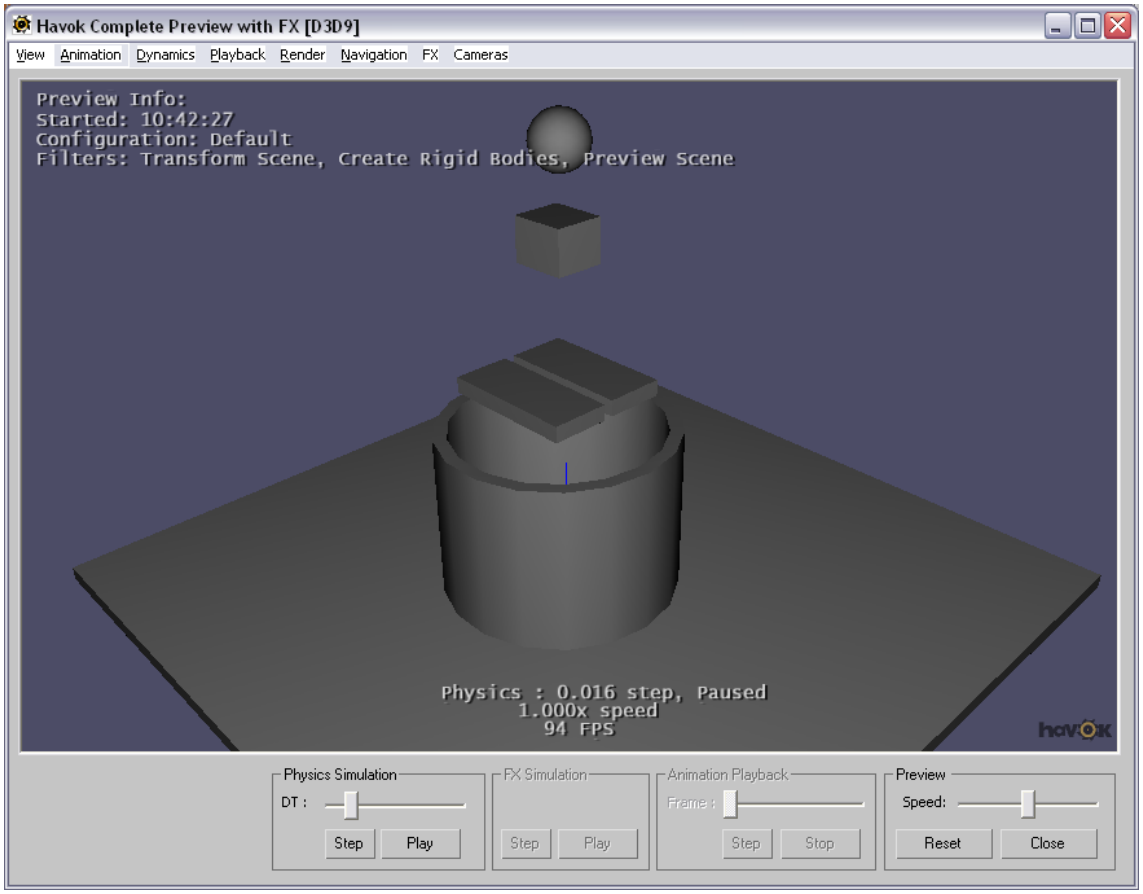
Insert, in this order, the following filters:

1. Scene Transform (**Core** category): Since 3ds Max works in inches and we work in meters, set the **Preset** parameter to **SCALE_INCHES_TO_METERS**
2. Create Rigid Bodies (**Physics** category): This filter will detect the rigid body information we assigned to our objects and will create rigid bodies based on it.
3. Preview Scene (**Graphics** category): We want to visually inspect the behaviour of our objects, so we will use this filter to render and simulate them.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



If we now process our asset (press the **Run Configuration** button), a preview window should appear with our scene

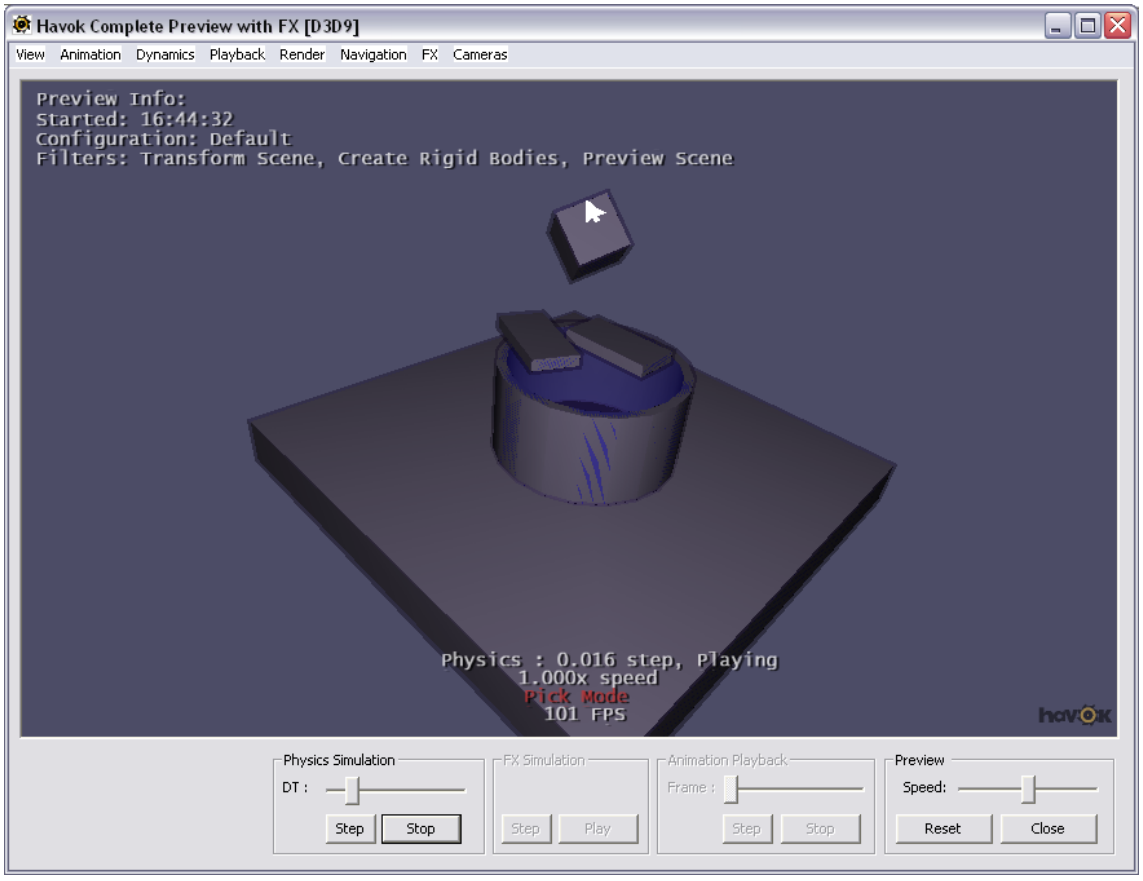


Press the **Play/Stop** button to start the simulation.

You will notice that all the objects fall as expected, but there are a couple of things that are not quite right:

- None of the objects fall inside the can - it is like there is an invisible lid on it. If you enable the menu option **View** > **Physics Ghosts** you should see a ghost representation of the rigid bodies displayed in the preview (in alpha blended blue). That shows that there is indeed a lid on the can. It looks like the can is being simulated as a closed volume!
- The trap doors do not behave as such, and are just falling as any other object. This is because we haven't setup any constraint to specify that they should behave as hinged doors. We will take a look at this later in this tutorial.

Mouse Picking
You can interact with the objects in the scene by using *mouse picking*: with the mouse cursor over one of the movable objects, press the **SPACE** bar and, without releasing it, move the cursor around - the rigid body will behave as if a spring was attached between itself and the mouse pointer. Release the **SPACE** bar to release the object.

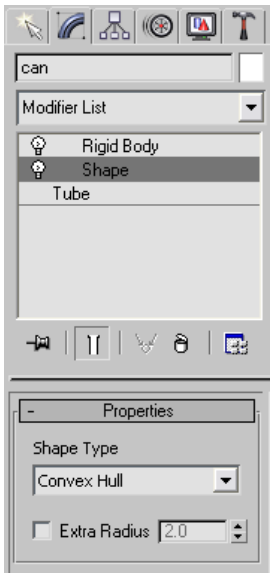


Close the preview and the filter manager and return to our 3ds Max scene.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.3.6.4 Shape Properties

Let's take a look at the can. Since our problems are related to collision detection, we'll take a look at the shape information associated with it. Select the can object in the viewports and, in the Modify panel, select the Shape modifier:



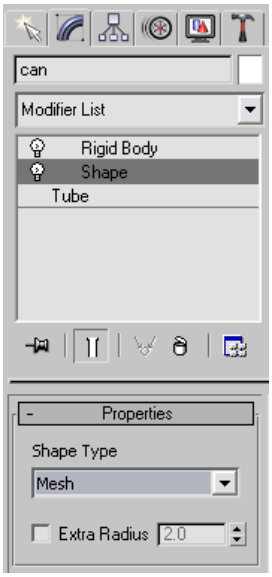
Notice how the **Shape Type** parameter is set to Convex Hull.

When creating a rigid body from a mesh, the default value for this Shape Type parameter is taken by examining the object type in 3ds Max. Thus, the default Shape Type for boxes is (Bounding) Box, the default Shape Type for spheres and geospheres is (Bounding) Sphere. For other arbitrary objects (like our can or the teapot), the default Shape Type is Convex Hull[1]. The Convex Hull of an object is the tightest convex volume that includes all the vertices of the object. The best way to visualize a convex hull is by imagining shrink-wrapping the object. In our case, using the convex hull of the can results on simulating a closed cylinder, rather than an open one, leading to the behaviour we saw in the preview.

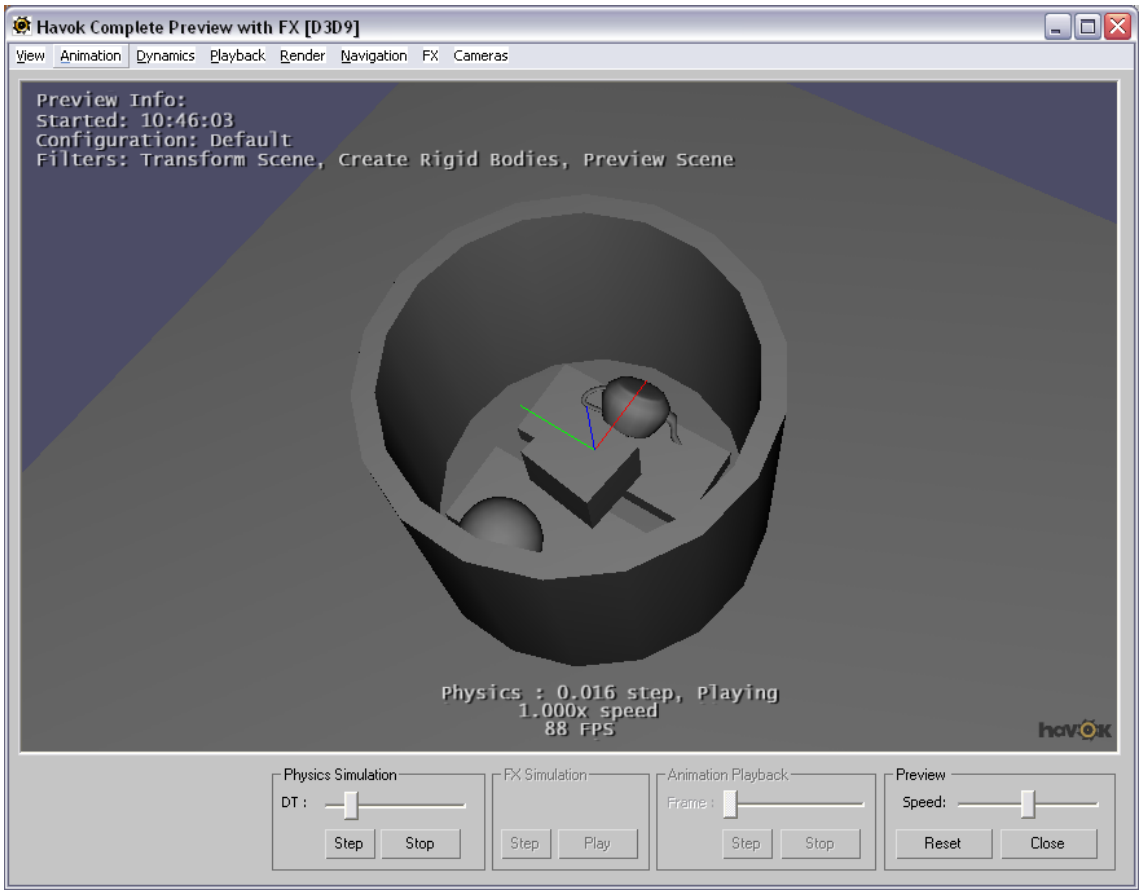
In our case we want to use the exact mesh of the object. Change the **Shape Type** parameter to **Mesh** :

[1] The reason why the default Shape Type is Convex Hull and not Mesh is that convex objects are much faster to simulate than concave meshes and, in many cases (the teapot for example), simulating a mesh as its convex hull gives very good results. Also, movable (not fixed) concave objects (objects with Shape Type set as Mesh) are not really suitable for interactive simulations (a warning will be generated if used).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



If you now **Export** the scene again and process the asset (press **Run Configuration**), you will see that the objects now properly fall inside the can during the simulation:



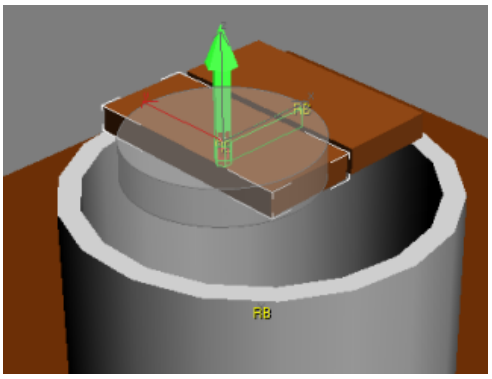
Close the preview and the filter manager and return to our 3ds Max scene.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.3.6.5 Setting Up Constraints

The last thing we want to improve in our scene is the behaviour of the trap doors - currently they behave as the other objects (they just fall). We want them to behave as if they were constrained to their location by hinges. The Havok Content Tools for 3ds Max give you the ability to set up and visualize constraints in 3ds Max. In our case, we want to create a hinge constraint - a constraint that will limit the movement of the trap doors to a rotation around a single axis.

Select one of the trap doors and click on the **Create Hinge Constraint** button or menu option:



This adds a **Hinge** constraint modifier to the object. First of all, notice how the **Constrain To** parameter is set to **World** - this means that the constraint will only affect our rigid body, and its position, orientation and limits will be absolute rather than relative to another (parent) rigid body. This is what we want.

This modifier, when selected in the Modify panel, displays a representation of the hinge - an axis and a volume of allowed rotation (with another axis specifying where zero-rotation is).

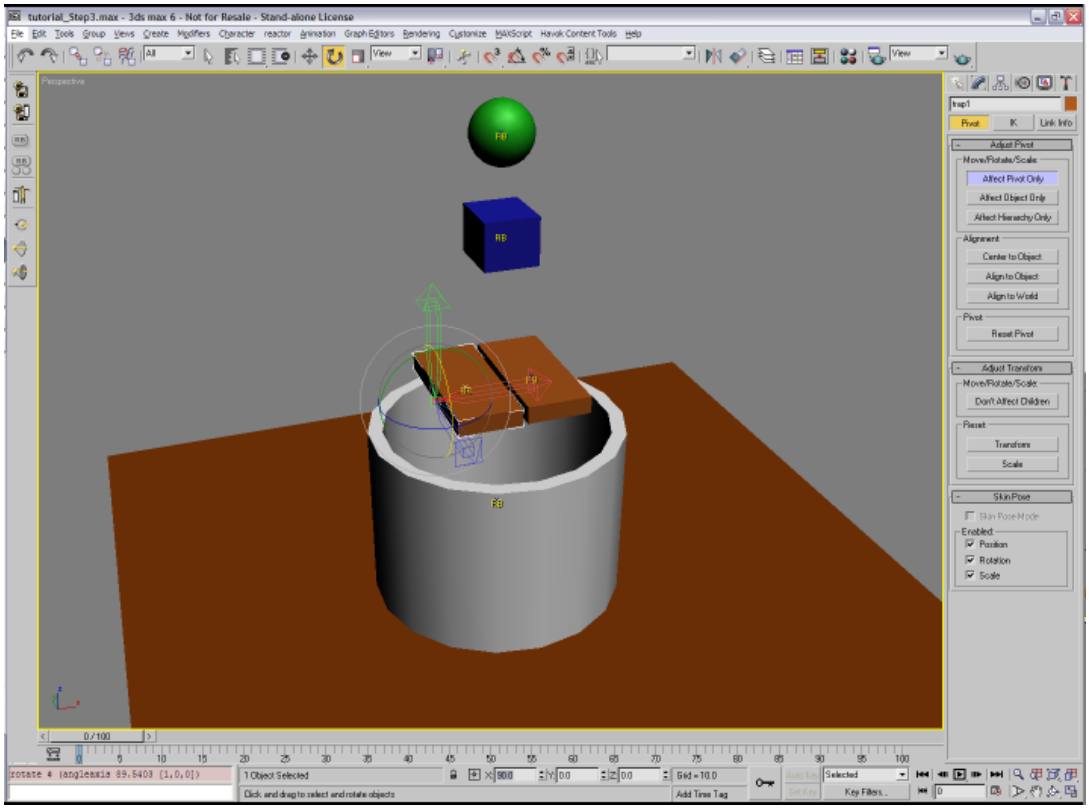
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

By default these axes are aligned to the object's Z and X axis, and placed at the object's pivot point. In our case, however, we'd like to place the constraint on the side of the trapdoor and orientate it so that the trap door would move up and down rather than sideways.

We can change the orientation of the constraint limits and axes in two different ways: by modifying the object's pivot or by manipulating constraint spaces.

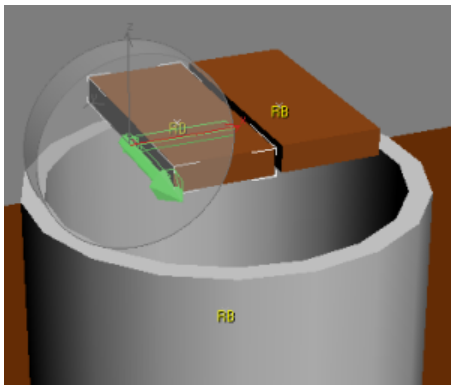
Orientating the constraint by modifying the pivot

With the trap door selected, go to the **Hierarchy** panel and click on the **Affect Pivot Only** button. Then, with standard Move and Rotate tools, move the pivot of the trap door to the side, and orientate the pivot so the Z axis (in blue) points alongside the length of the door:



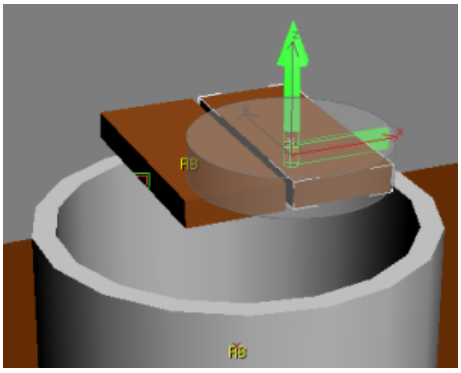
If you now go back to the Modify panel, you will see how the constraint is placed and orientated following the new setup of the pivot:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



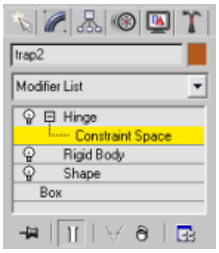
Orientating the constraint by manipulating constraint spaces

Let's now set up a hinge constraint for the other trap door. Select the other trap door and click on **Create Hinge Constraint** :

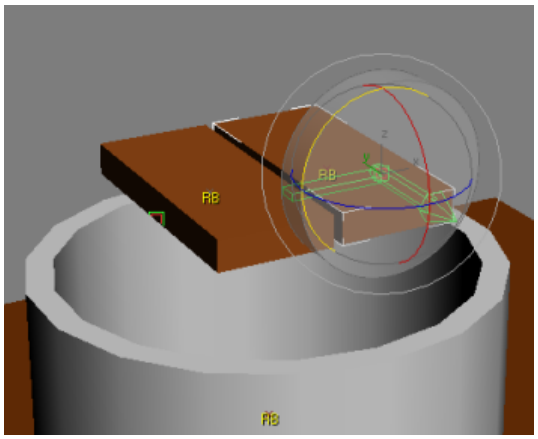


Again, the constraint's default orientation is not what we want. This time, rather than modifying the pivot of the object, let's manipulate the constraint spaces associated with the hinge.

In the **Modify** panel, open the subobjects in the **Hinge** modifier and select the **Constraint Space** subobject:



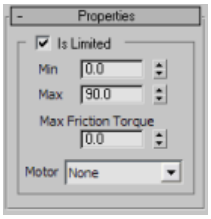
In this subobject mode you can use the standard move and rotate 3ds Max tools to manipulate the location of the constraint. Move the constraint so that it is placed on the side of the trap door, and rotate it so the main axis points alongside the length of the door and the zero-rotation axis (the one inside the volume) points towards the other trap door:

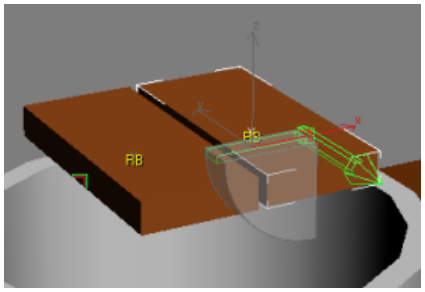


Limiting the hinges

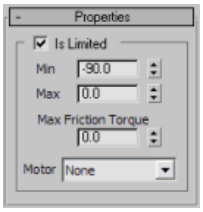
By default, hinge constraints allow full freedom of rotation around the main axis. It is possible, however, to limit that amount of rotation to a specific range.

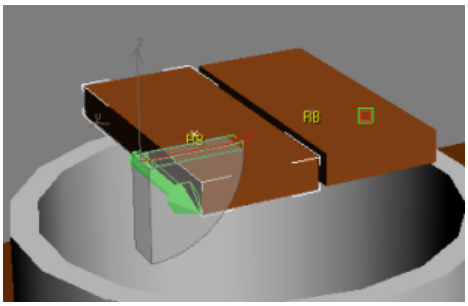
Select the right trap door, and in the **Modify** panel, check the **Is Limited** box. Notice how the **Min** and **Max** parameters are now enabled, and in the viewports the volume (displayed in transparent grey) reflects those two values (the limits are specified counterclockwise from the zero-rotation axis). Change the **Min** limit to 0. This, as reflected in the viewport, should enforce that the trapdoor doesn't rotate upwards:





Now do the same steps for the left trap door; in this case, change the **Max** limit to 0:



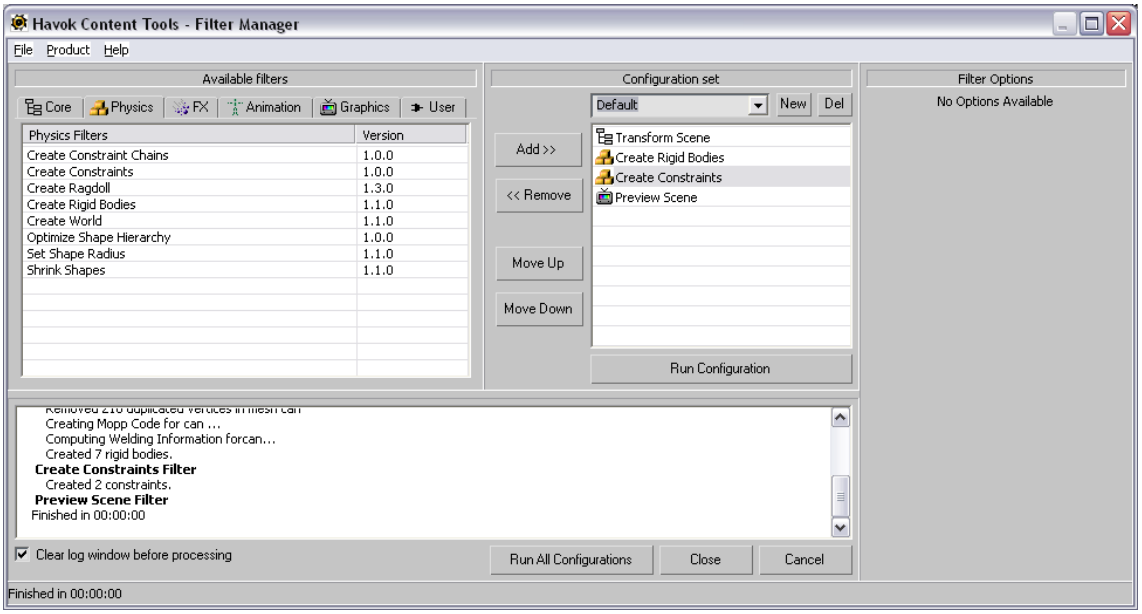


6.3.6.6 Previewing the Scene

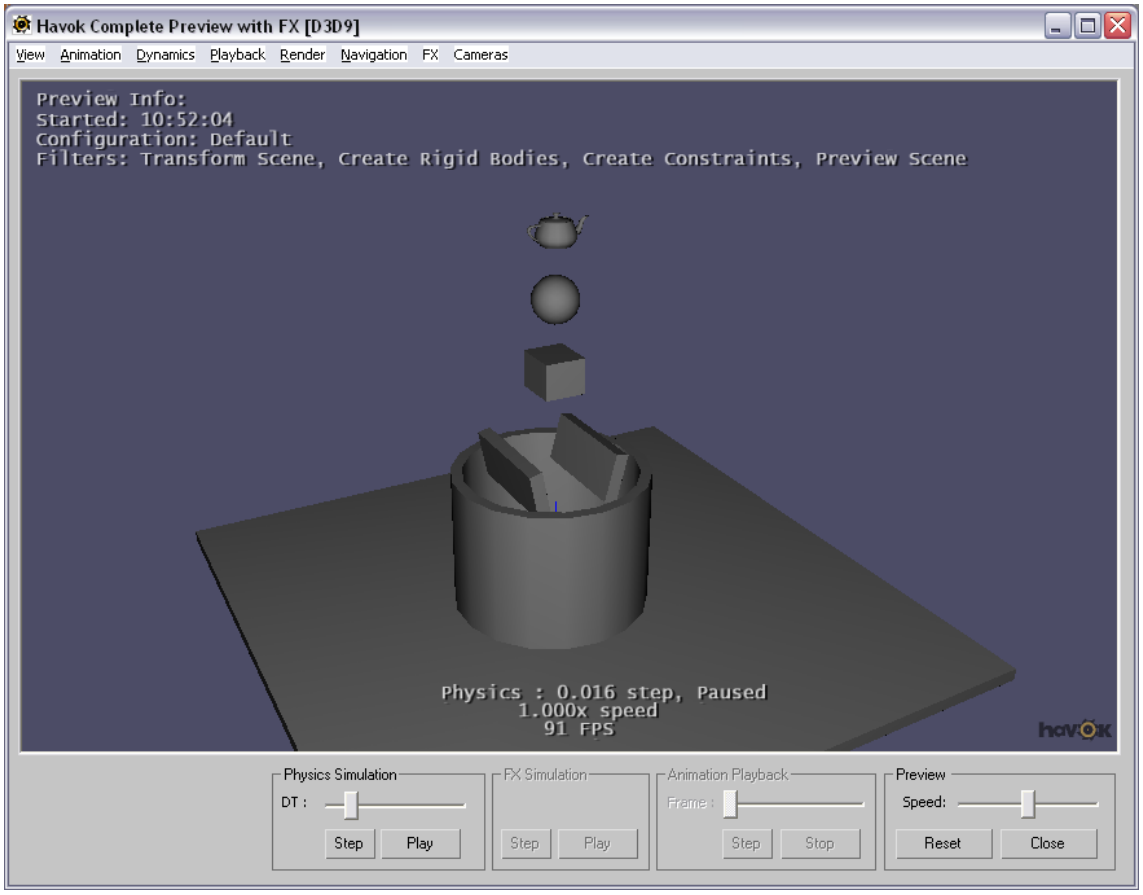
Now that we have set up our constraints, let's see what effect they have in the simulation.

Open the filter manager again by exporting the scene .

We now need to add a Create Constraints filter (in the **Physics** category), in order to create constraint objects from the data we set up in 3ds Max. Place it after the Create Rigid Bodies filter, and before the Preview Scene filter.



If you now execute the processing by pressing **Run Configuration** , you will see the constraints in action:



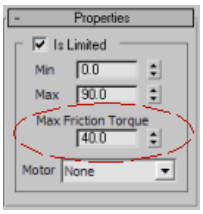
You can check that the limits we applied to the hinges are working by using mouse picking - you will notice that the doors won't move outside the limits we specified.

Close the preview window and the filter manager, and return to 3ds Max.

6.3.6.7 Adding friction to constraints

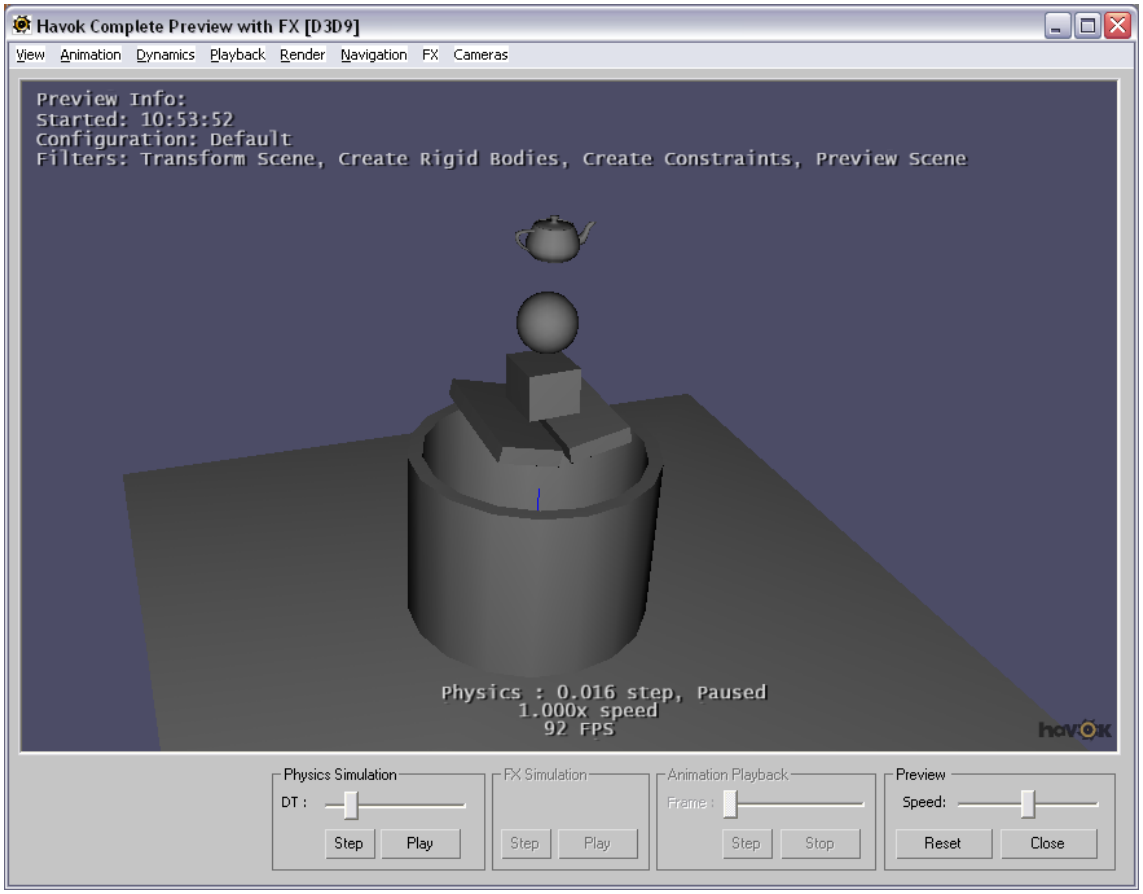
Our setup is almost finished. The last thing we'd like to fix is the fact that the trap doors open even before the objects hit them (due to gravity). We'd like to add some friction to the hinges so the doors only move when pushed by the objects falling on them.

Select on the trap doors, and in the **Modify** panel, change the **Max Friction Torque** parameter to 40. This parameter specifies how much torque (angular force) is absorbed by the hinge's friction - only torques above this value will cause the hinge to move.



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

If you now export and preview the scene, you will notice how the trap doors remain closed until the objects fall on top of them.



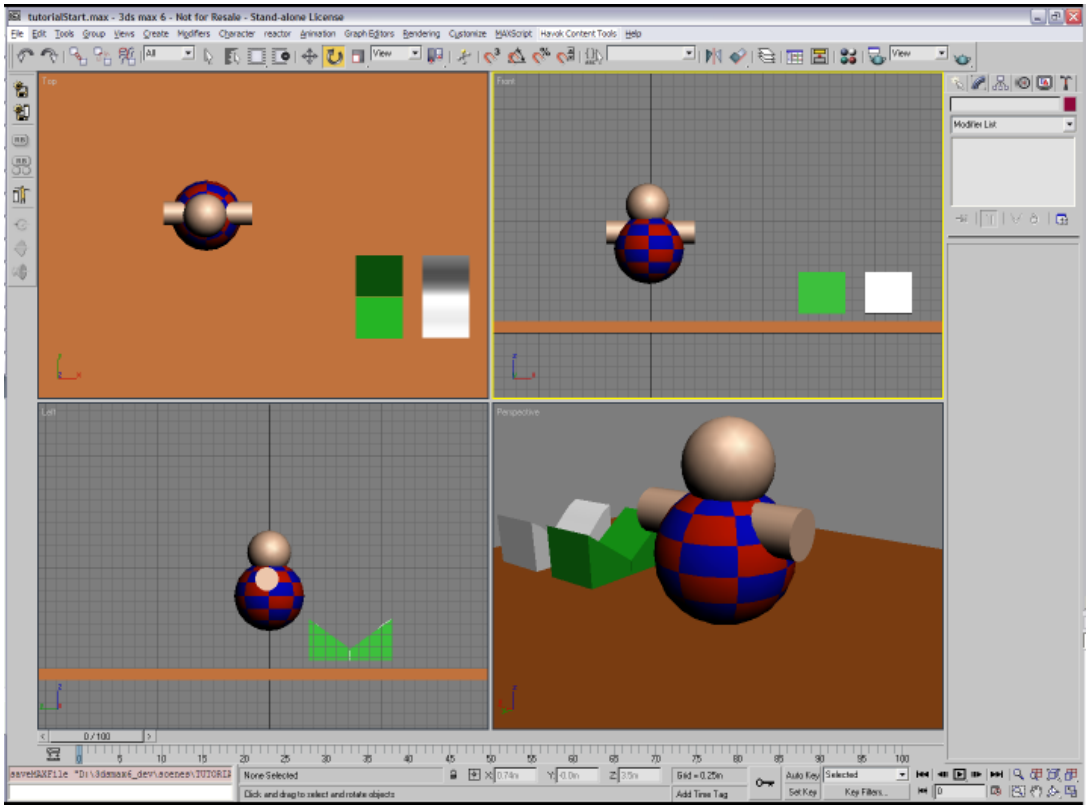
And this finishes our tutorial. You can find the fully completed tutorial in the scene file "`tutorialEnd.max`".

6.3.7 Tutorial: More on Rigid Bodies

In this tutorial we will explore some more concepts regarding rigid body setup, like creating compound rigid bodies and changing the center of mass. The tutorial assumes that you are acquainted with the basics of exporting and processing assets and working with rigid bodies. We recommend you follow the first two tutorials (Export and Animation Basics and Physics Basics) before proceeding with this one.

6.3.7.1 Getting Started

Start by loading the scene "`tutorialStart.max`", located in the `"scenes/havokContentTools/tutorials/moreOnRigidBodies"` folder.

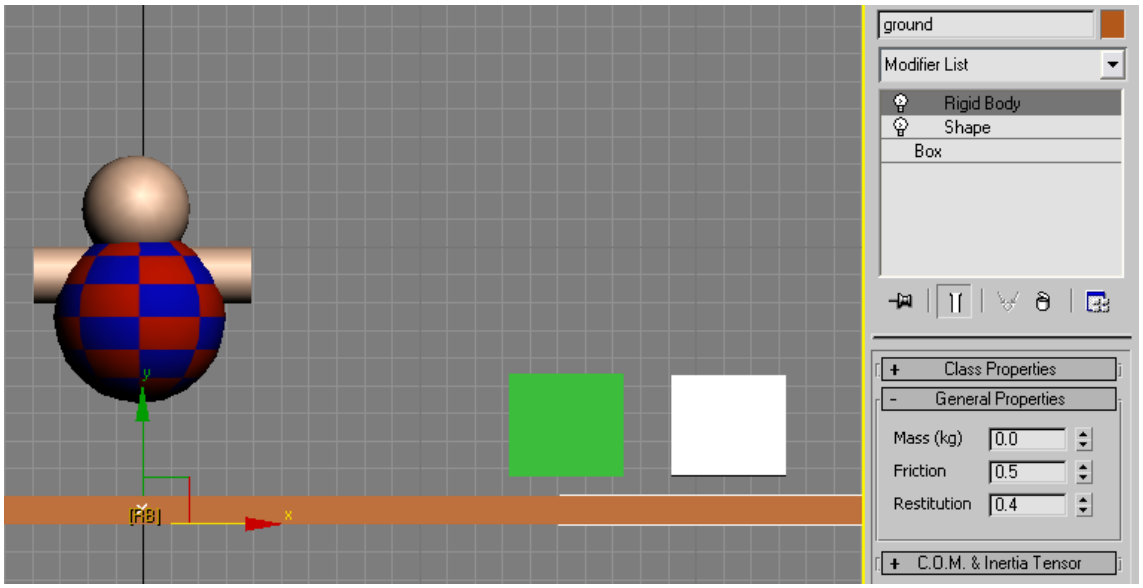


In the scene you should find the following:

- Two spheres and a cylinder, making up the shape of a little toy man
- A ground box
- A concave object (a little ramp)
- Two convex objects that together make a shape similar to the ramp

6.3.7.2 Creating a Fixed Rigid Body

Let's start by making a rigid body out of the ground box. Select it and press the **Create Rigid Body** button or menu option. Since the default mass is zero, the ground box will be fixed in space. That's exactly what we want.

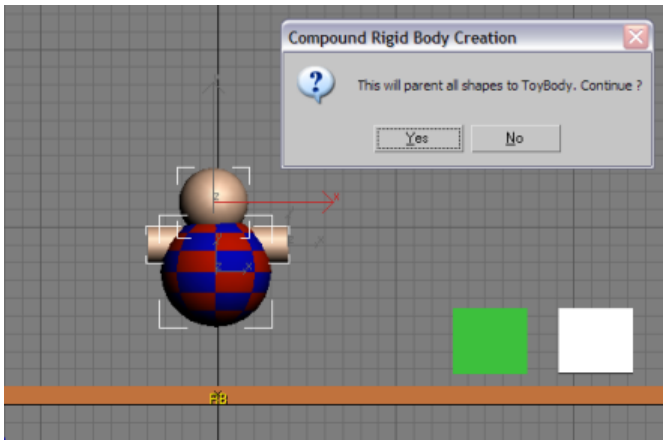


6.3.7.3 Creating a Compound Rigid Body

For the toy man we want to create a rigid body from the two spheres and the cylinder. When we create a rigid body out of multiple objects, we are creating a compound rigid body.

When creating a compound rigid body, one of the objects in it needs to take the role of rigid body - this object will hold the rigid body information, and the rigid body body will be named after it. The other objects need to be parented to that rigid body object. If the objects are not parented already, the last object selected will be considered to be the rigid body object.

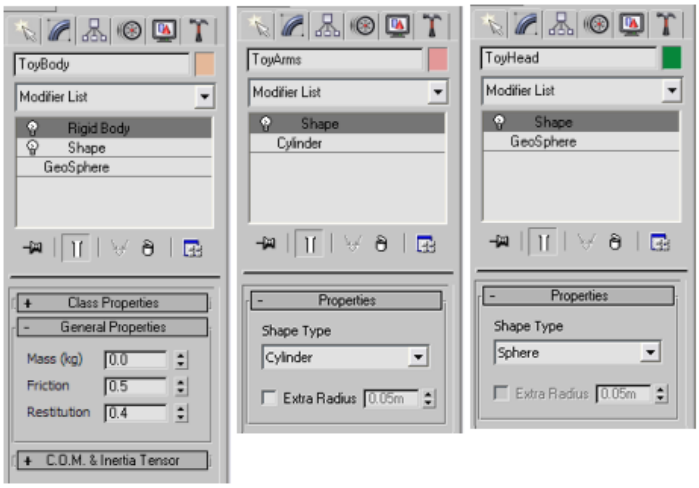
Holding 'ctrl', select in turn the head, the arms and finally the body of the wobbly man. Then click on the **Create Compound Rigid Body** button or menu option :



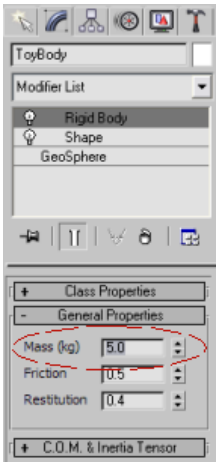
A confirmation dialog appears. This is because, as we mentioned, compound rigid bodies need to form a hiearchy. The Havok Physics Tools have assumed that the last one selected (*ToyBody*) is the parent (the one that will hold the rigid body information) - but is asking us for confirmation before proceeding to reparent the arms and the head. Click **Yes** to continue.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

After this, *ToyArms* and *ToyHead* will be parented to *ToyBody*. Shape information (a Havok Shape Modifier) will be added to each of the three objects (as the three of them are used for collision detection). Rigid Body information (the Rigid Body modifier) will only be added to *ToyBody*. You can verify this by looking at the modifier stack of each object:



Notice that the rigid body has a default mass of zero. Since we want it to move, we need to set the mass to a different value, let's say 5 Kg

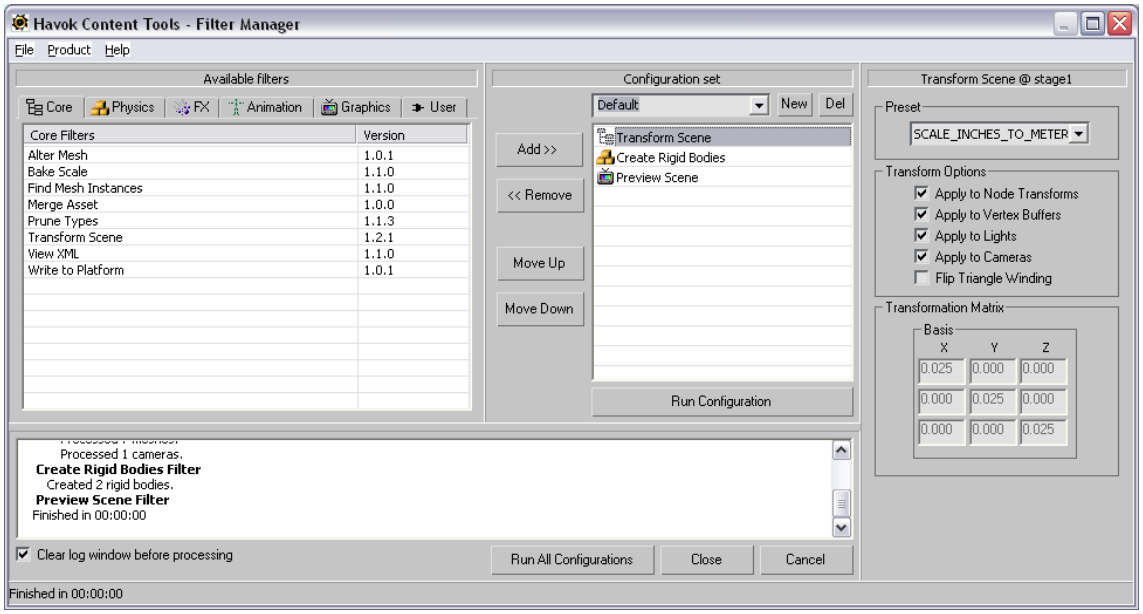


Also notice how the arms of the toy have the Shape Type parameter already set to " **Cylinder** " and the head to " **Sphere** " - the Havok Content Tools will automatically detect some 3ds Max primitives and set the right **Shape Type** for them when creating rigid bodies. It is still useful always to verify the shape type.

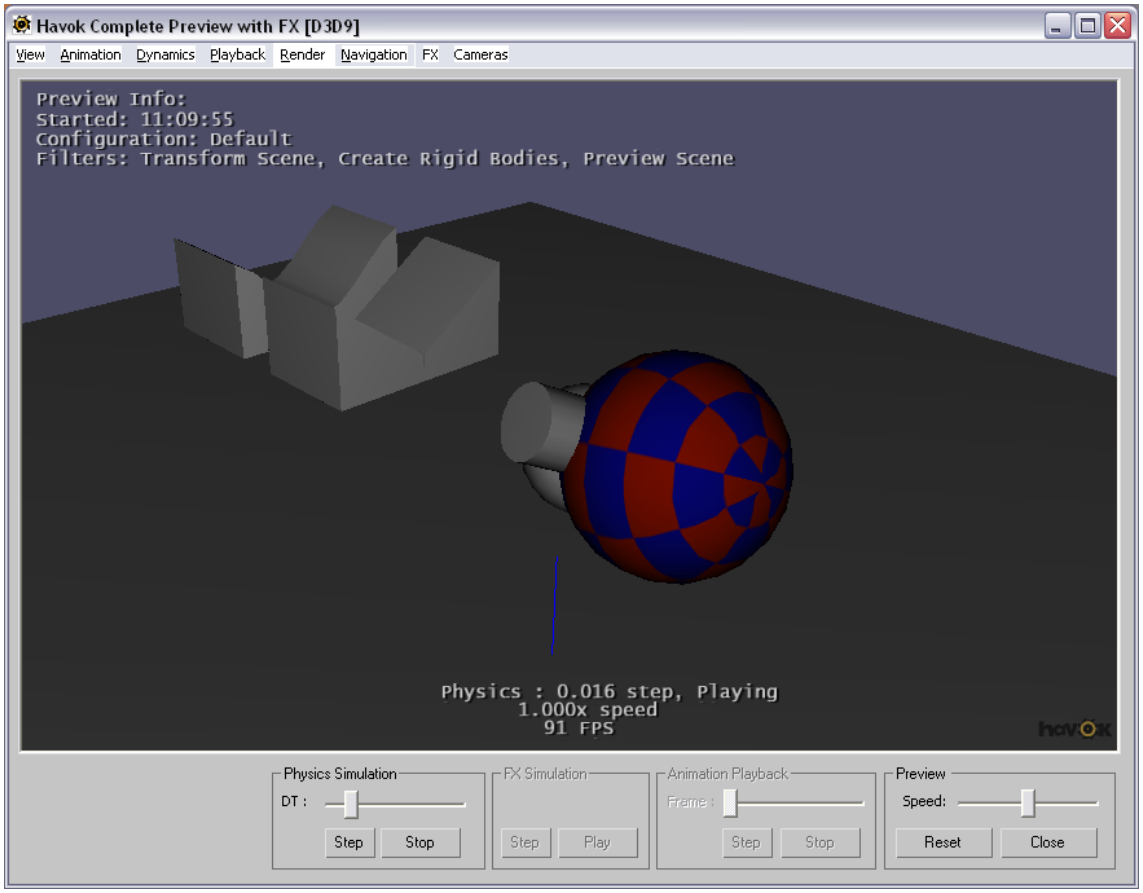
Previewing the Scene

Let's now preview the behaviour of our compound rigid body. Click on **Export** and, in the filter manager, add the Scene Transform (using the **SCALE_INCHES_TO_METERS** preset), the Create Rigid Bodies and the Preview Scene filters (as we did for the previous tutorial):

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Click the *Run Configuration* button - this will process the asset and open a preview window:



As you can see (using mouse picking to interact with the toy man) the three objects behave as a single rigid body.

Close the preview and the filter manager and return to 3ds Max.

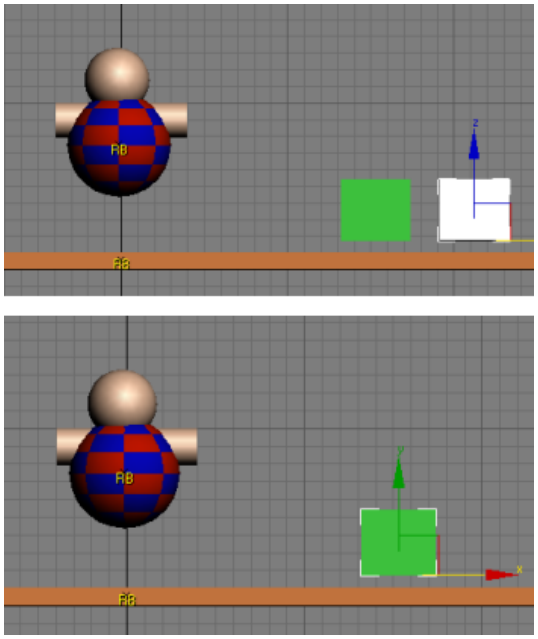
6.3.7.4 Creating a Rigid Body with Proxy Shapes

Sometimes we may want to simulate an object as a rigid body using another object or set of objects for the collision detection rather than the mesh of the object itself. For example, we may want to simulate a concave object as the combination of multiple convex pieces.

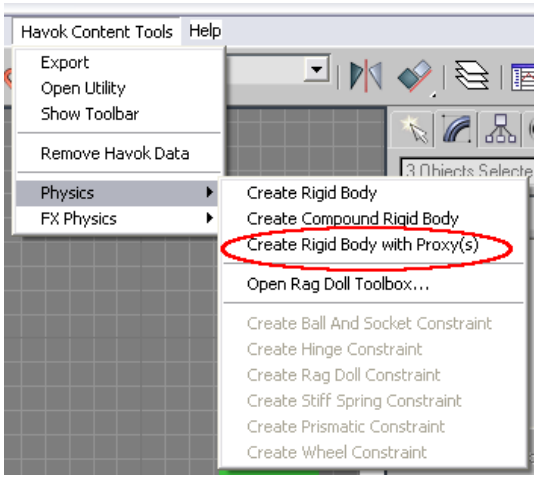
Creating a rigid body with proxy shapes is very similar to creating a compound rigid body. The only difference is that, in this case, the object containing the rigid body information (the rigid body modifier) does not contain shape information (the shape modifier) since its mesh it's not used for collision detection. Shape information is instead taken only from its descendants.

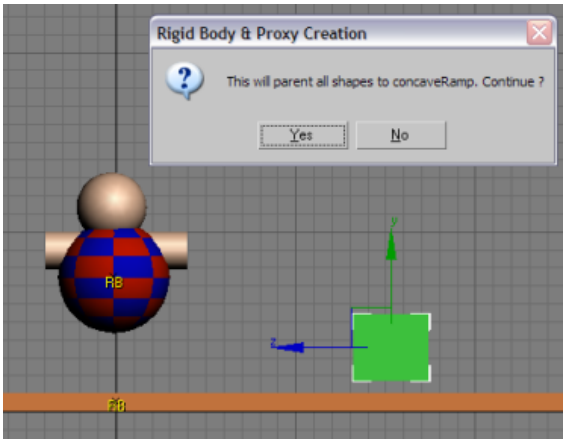
Let's see this in action in this scene. Notice how the white ramp is a concave object. Ideally, we'd like to simulate it using two convex objects (since simulating convex objects is far more efficient than simulating concave objects).

We will use the two green pieces to create a proxy shape representation of the ramp. Let's start by moving the objects so the two pieces and the concave ramp match their locations in the scene:



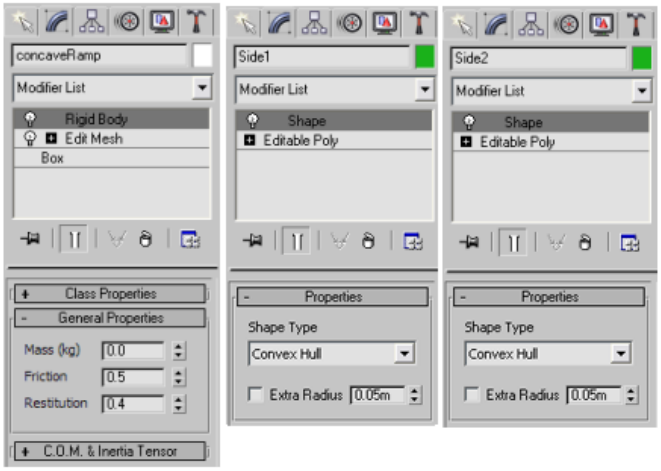
We now want to convert the white ramp to a rigid body, but using the two convex pieces as the shapes. Hence, select the two green convex shapes first and then the white concave ramp, and click on the **Physics > Create Rigid Body with Proxy(s)** menu option.



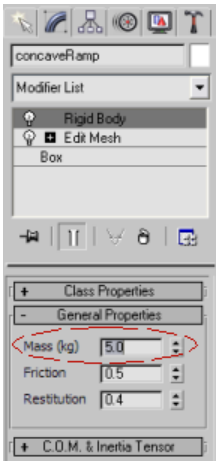


If you selected the objects in the right order, the confirmation dialog above will appear. Click on **Yes** to confirm. Alternatively, start by first parenting the two convex pieces to the ramp manually, and then with the three objects selected click on **Create Rigid Body with Proxy(s)** .

Notice how the *concaveRamp* object now has only a rigid body modifier added, while the two convex pieces (which are now parented to it) each have only a shape modifier added:



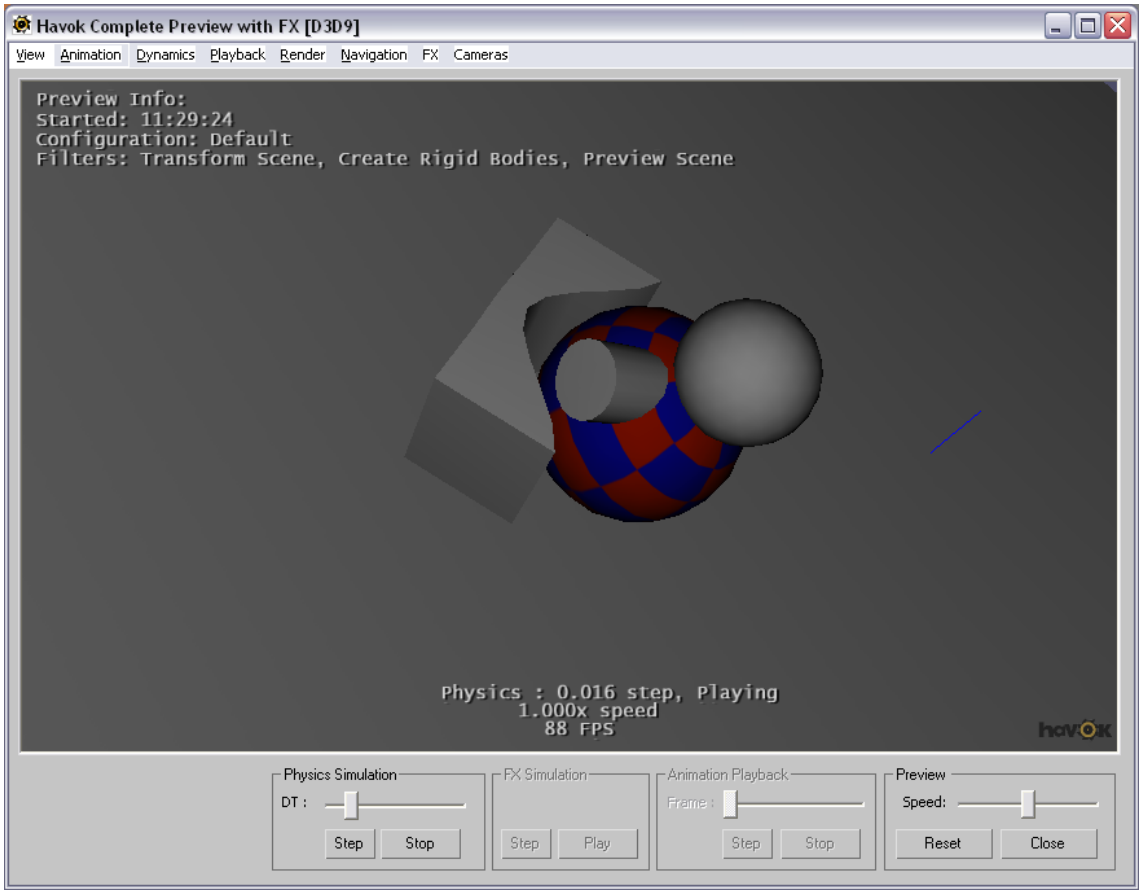
Notice how since the two convex pieces are not boxes, spheres, capsules or cylinders, their **Shape Type** is set by default to be " **Convex Hull** ". Also notice how, again, the default mass of our rigid body is zero. Since we want it to be dynamic (movable), we'll set it to something different (5Kg for example):



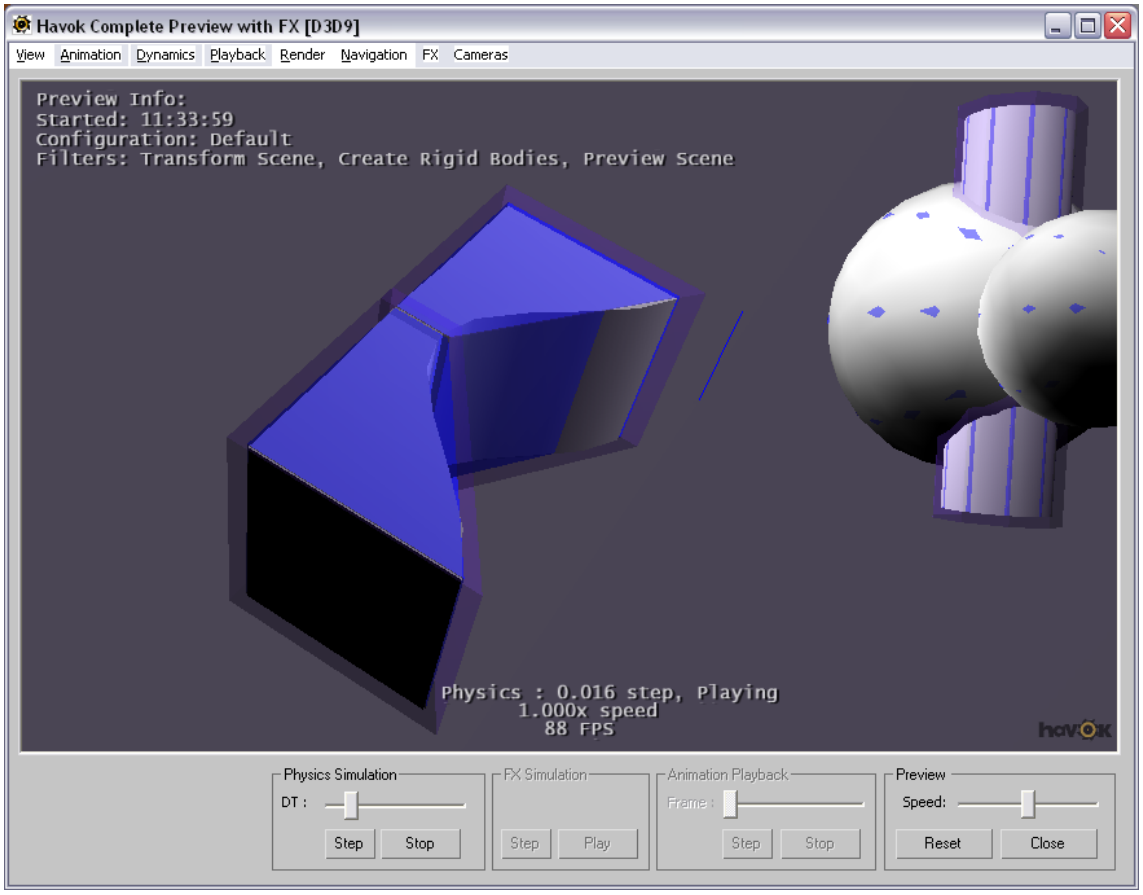
Previewing the Scene

Let's see how the ramp behaves in our preview. Export the scene and press on **Run Configuration** to execute the preview. You should see how the ramp is simulated using the convex pieces:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



To view exactly what is simulated, switch off **Display Meshes** and switch on **Physics Ghosts** in the **View** menu of the preview:



Notice how the two convex pieces, and not the concave mesh, is being used to simulate the ramp.

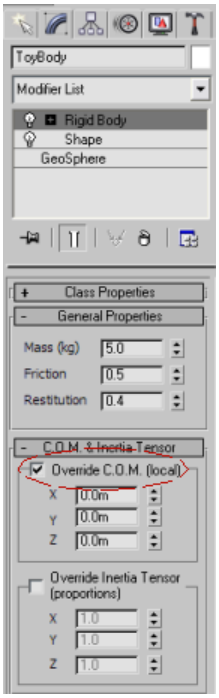
Note:

Unfortunately the Preview Scene filter currently displays the proxies along with the actual rigid body mesh. For this reason it may be difficult to make out the curved ramp mesh, with the proxy meshes obscuring it. Enabling wireframe mode may help here.

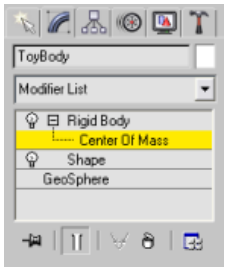
6.3.7.5 Changing the Center of Mass

We'd like out little man to behave like a tumbler, one of those toys that keep themselves straight. Those toys work by having a very low center of mass (usually by having a lead weight inside them). The Havok Content Tools allow you to change the center of mass of rigid bodies easily.

Back in 3ds Max, select the *ToyBody* rigid body modifier, and in the **COM and Inertia Tensor** rollout, check the **Override COM (local)** check box.

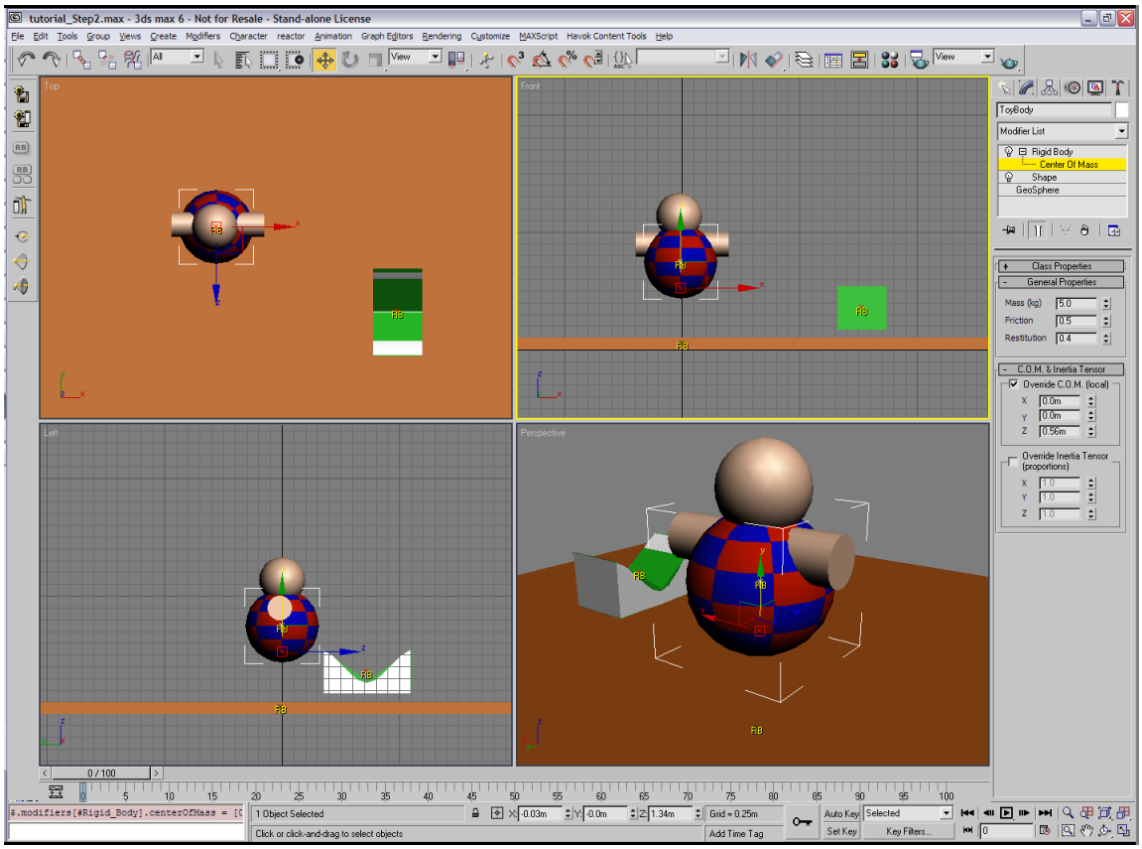


A new " **Center of Mass** " subobject appears in the rigid body modifier:



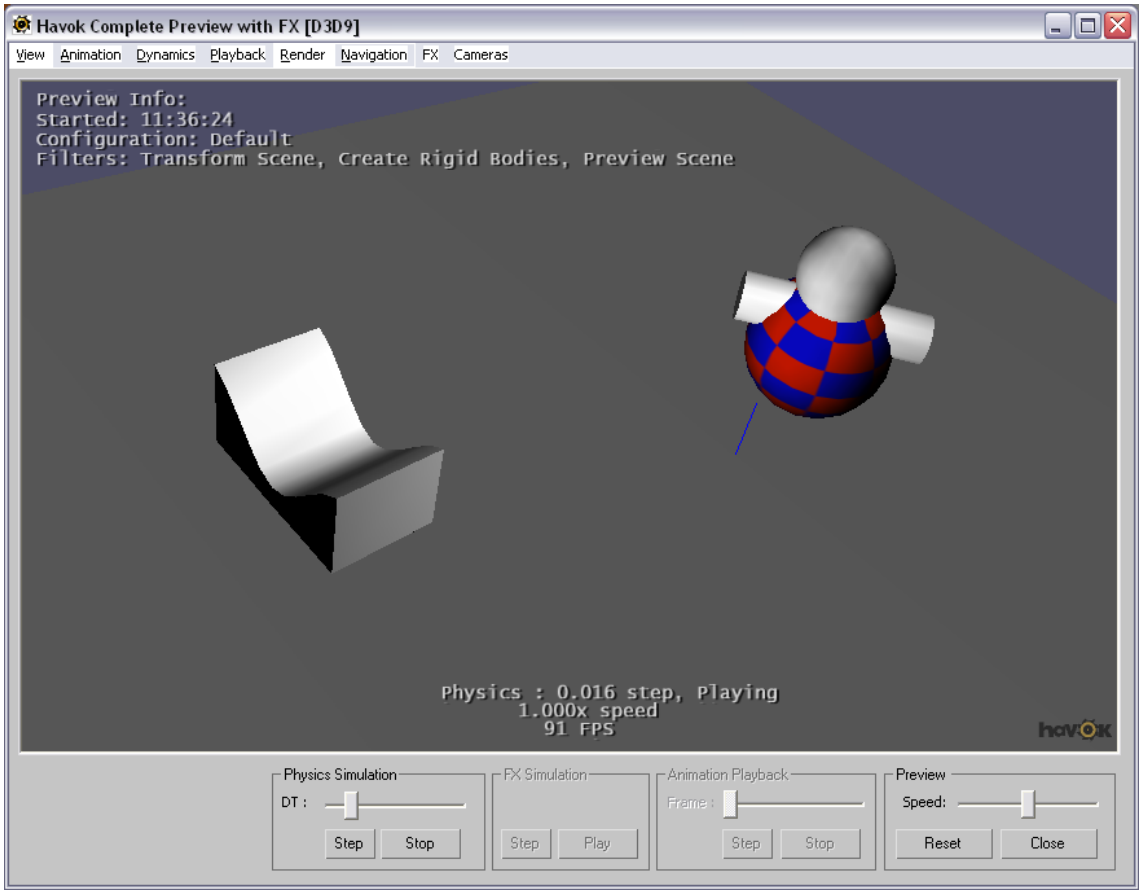
This subobject allows you to change the location of the Center of Mass (displayed as a red crossed box) visually in the viewports (using the standard move tool). In our case we want to lower it:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



If you execute the preview again (export and press **Run Configuration**), you see how now our toy keeps himself straight. Use mouse picking to interact with the toy:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

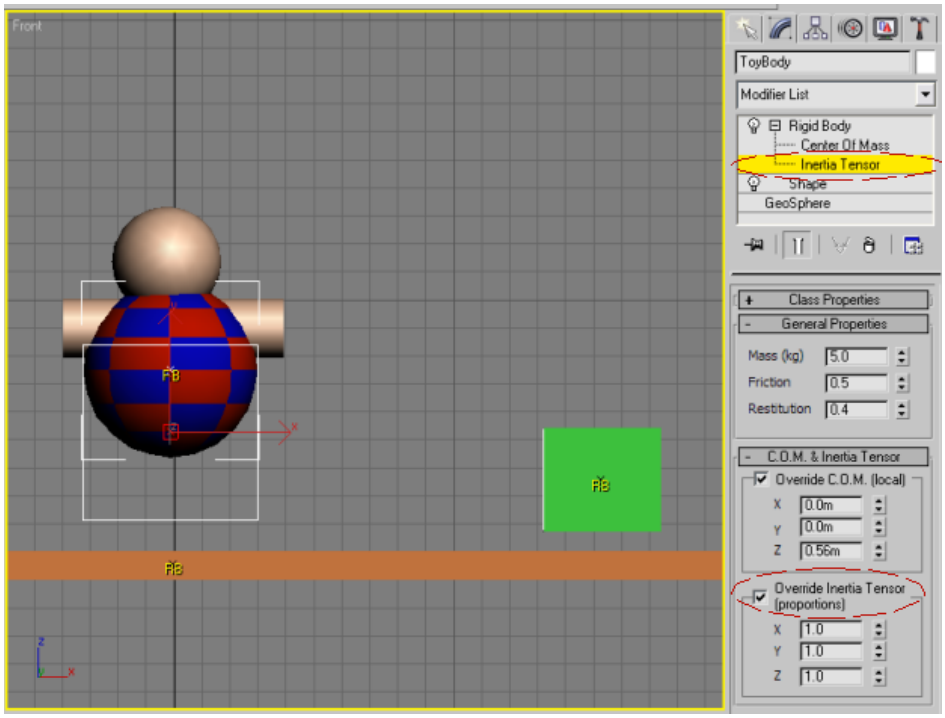


6.3.7.6 Changing the Inertia Tensor

We'd now like to customize the motion of our little man even more by changing the proportions of its inertia tensor.

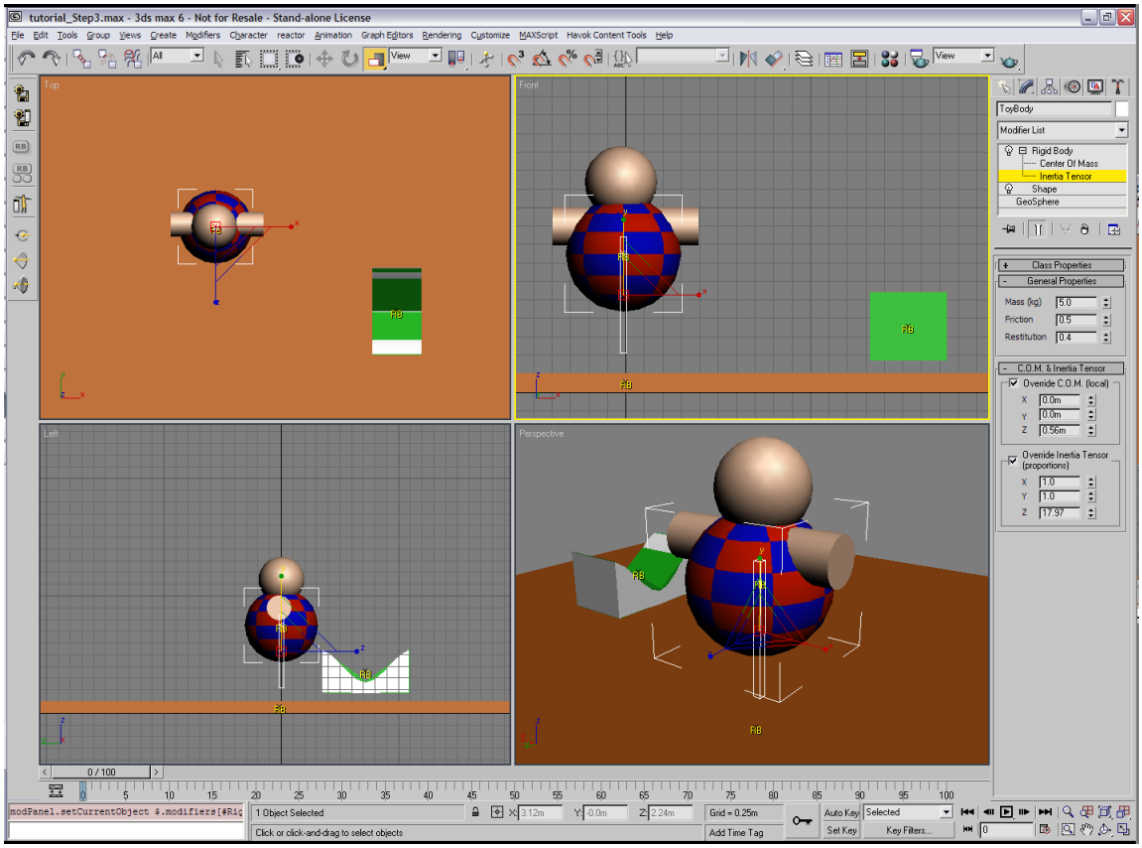
Select its rigid body modifier again, and check the box **Override Inertia Tensor (proportions)** . This will add another subobject, **Inertia Tensor** , to the rigid body modifier, and will also show a visual representation of the inertia tensor in the viewports (a unit cube in this case):

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



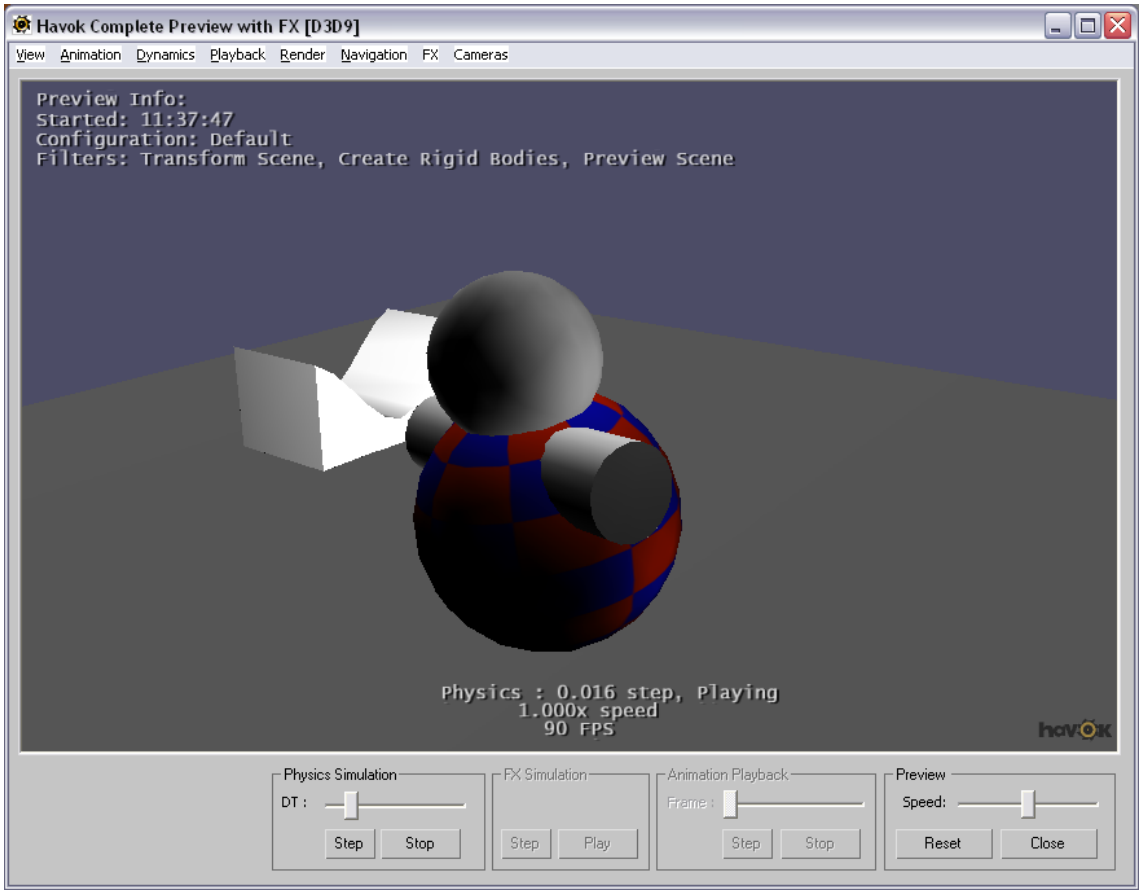
You can visually change the proportions of the inertia tensor by going into the **Inertia Tensor** subobject and using 3ds Max Scale tool. In our case (but feel free to experiment with different settings) we'd like an inertia tensor which is biased in the vertical direction. So we scale that direction up:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



If you now preview again, you will see that our little man will tend to wobble more around this vertical direction, while it won't lean as much as before:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



And this finishes our tutorial. You can find the fully completed tutorial in the scene file `"tutorialEnd.max"`.

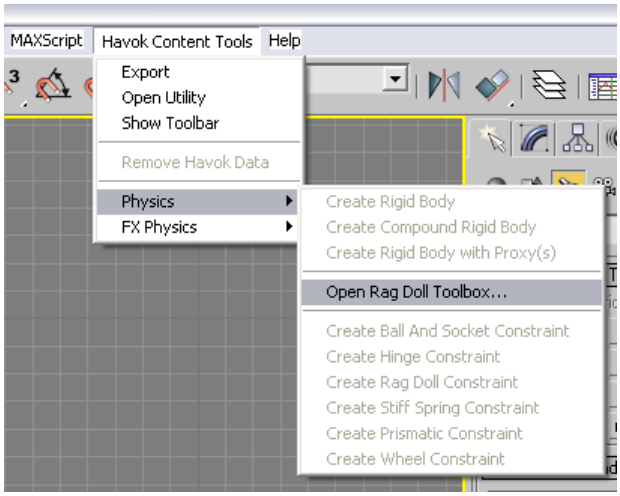
6.3.8 Tutorial: Rag Doll Toolbox

The "Rag Doll Toolbox" is a set of tools which facilitates the creation of proxies and the reuse of constraint setups. In this tutorial we will use the Rag Doll Toolbox to set up a (low res) hierarchy of constrained proxy rigid bodies to represent a (high res) hierarchy of bones.

This tutorial consists of three distinct parts, in which we will:

1. Create and associate rigid bodies with an existing skeleton using the Rag Doll Toolbox.
2. Create appropriate constraints between the rigid bodies using the Rag Doll Toolbox.
3. Setup a mapping between the rigid body skeleton (ragdoll) and the original high-res skeleton

To begin, open and examine the Rag Doll Toolbox, via the main **Havok Content Tools** menu or the toolbar icon:



The rag doll toolbox appears as a MAXScript window consisting of three rollouts:

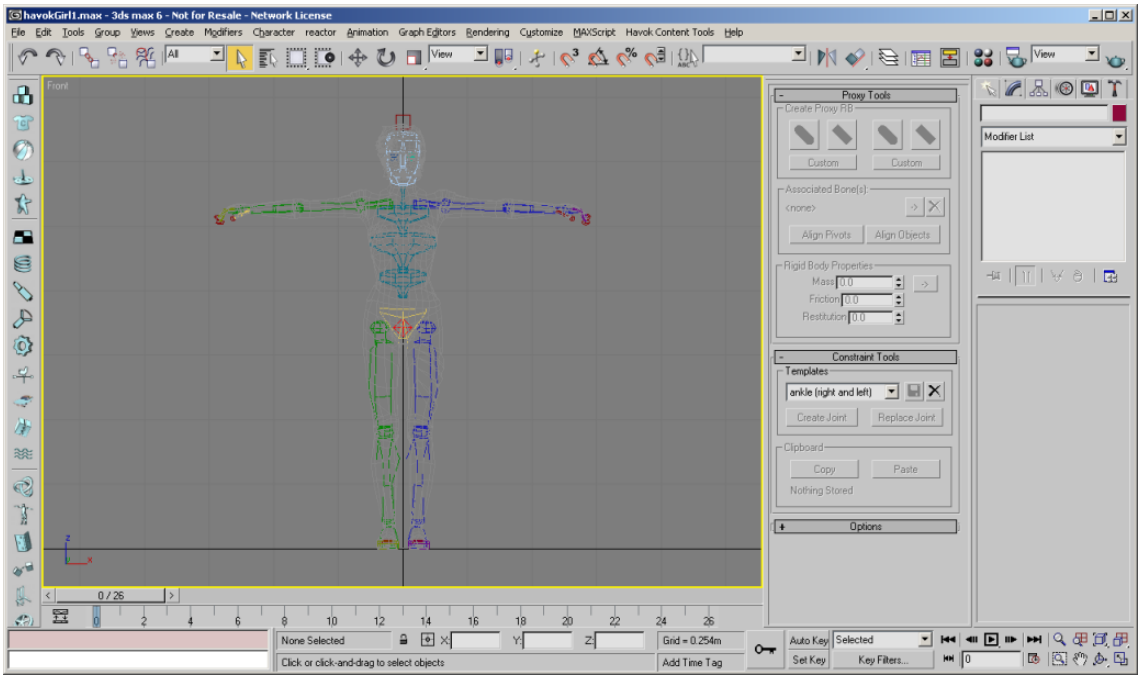
- *Proxy Tools*: Utilities to create proxy (rigid body) representation of animation bones, and associate one to each other.
- *Constraint Tools*: Utilities to create constraints between proxies and reuse constraint data between characters.
- *Options*: General options regarding both Proxy and Constraint Tools.

As well as being a floating window, the Rag Doll Toolbox can also be docked to the sides of the 3ds Max interface. Since you will usually work with one rollout at time (you will usually start by creating all of your proxies, then you will create your constraints), you can also close any rollout to reduce the size of the window.

6.3.8.1 Creating Rigid Bodies

Sample scenes are provided with the Havok Content Tools for this tutorial. These files can be found in "`scenes/havokContentTools/tutorials/ragDollToolbox/`".

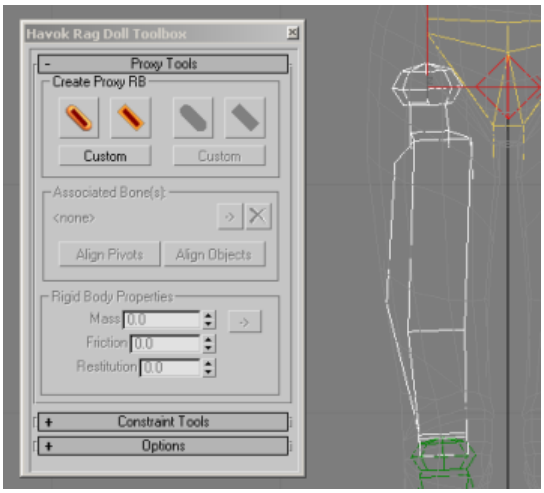
Let's start by loading the scene "*havokGirl1.max*" this scene contains a biped structure and a mesh which is attached to the biped through the use of a *Physique* modifier:



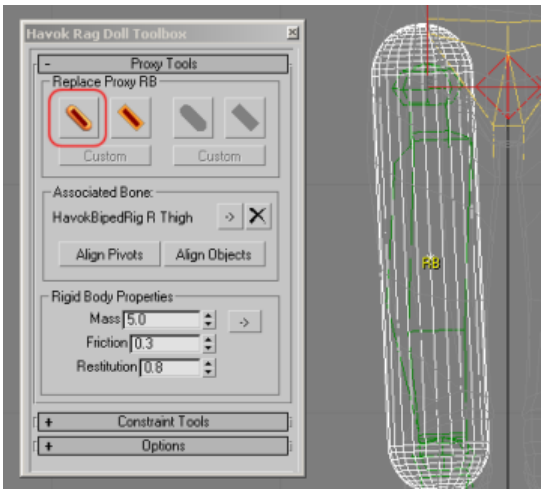
In this section we will focus on creating a rigid body representation of this character (a ragdoll) by creating and associating proxies to the bones of the biped. The *Proxy Tools* rollout of the Rag Doll Toolbox provides the tools to achieve this.

Simple Proxies

Start, for example, by selecting one of the thigh bones of the character. Notice how two of the buttons, **Create Bounding Capsule** and **Create Bounding Box** , now become enabled:

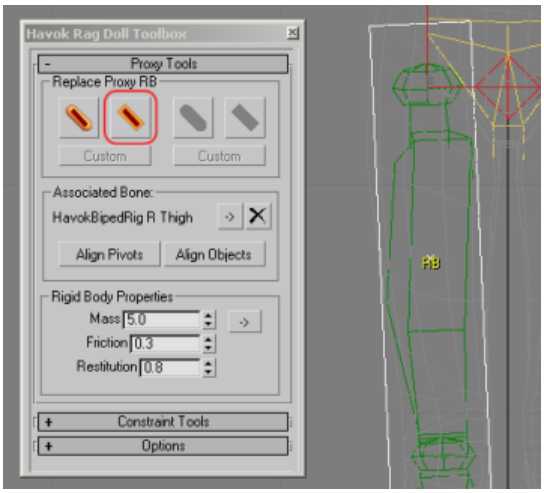


Click the first button to create a new capsule proxy for the thigh bone. A new capsule shape will be created to fit around the bone, with Havok rigid body and shape modifiers already attached, and initialized to a set of default values:

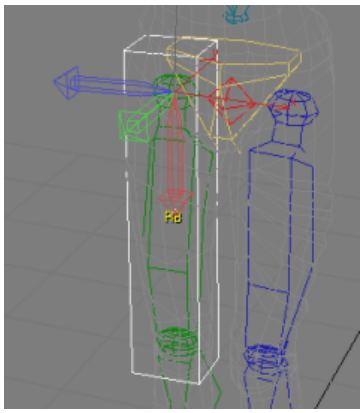


If you would prefer to use a box proxy instead, simply select the newly created capsule proxy and click the second of the two enabled buttons. This will replace the proxy with a box version, while retaining any rigid body properties you may have changed:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Observe that in each case above, the new proxy is created unparented (so as not to pollute the skeleton hierarchy). Observe also that the rotate pivot point of a new proxy automatically becomes aligned to that of the bone:

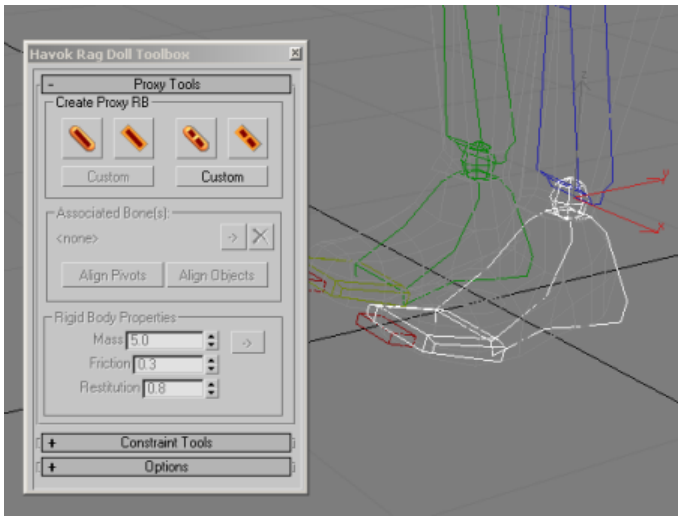


Continue creating simple proxies for each simple bone in the biped: thighs, calves, upper arm, hand, pelvis, head.

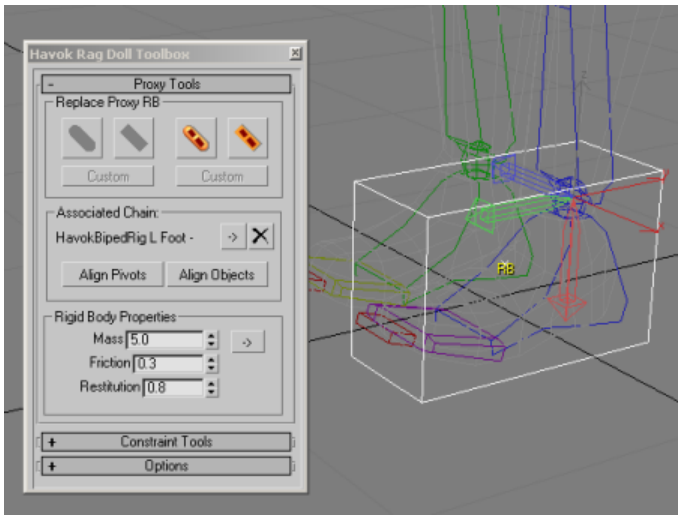
Chain Proxies

Suppose that you would like to create a single proxy to represent a chain of bones, for example the feet in this example. Select the two bones representing a foot (holding **CTRL** for multiple selection). When a valid chain of bones is selected, the second set of buttons in the **Create Proxy RB** section of the toolbox becomes enabled, allowing chain proxies to be created:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Click one of the buttons to create a capsule/box proxy for the selected bones:



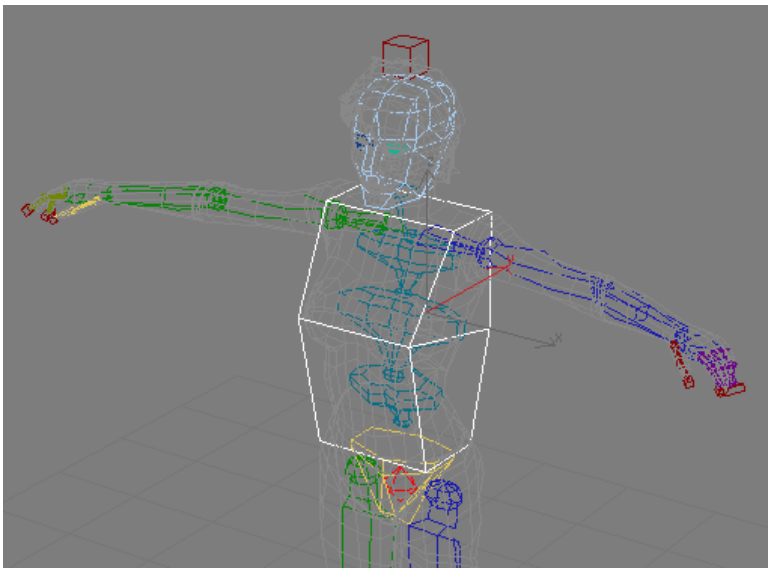
The only different between proxies created from a chain and proxies created from a single bone is that chain proxies keep a reference to both the start and end of the chain (while single proxies keep a single reference to a bone). Pivots of chain proxies are aligned to the start of the chain (ie. the ankle in this case). For any other purposes, chain proxies operate the same way as single proxies.

Continue creating chain proxies for the other foot, and for the two forearm bones in each arm.

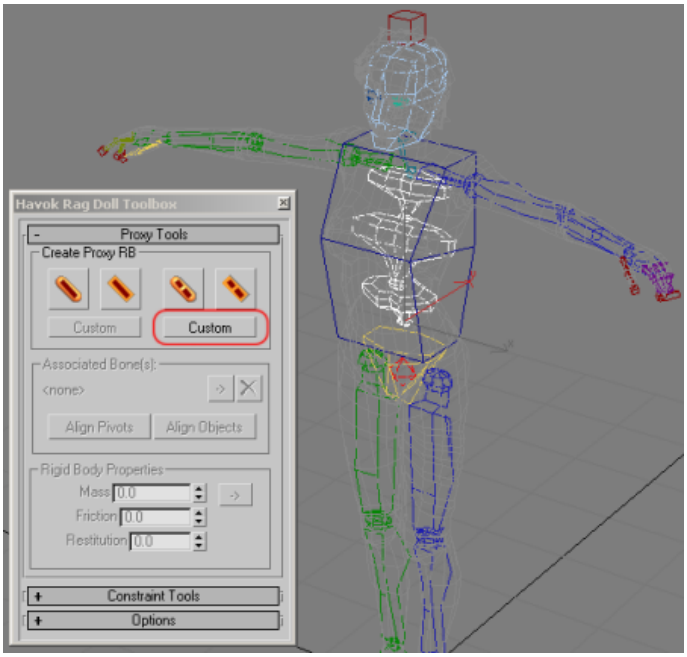
Custom Proxies

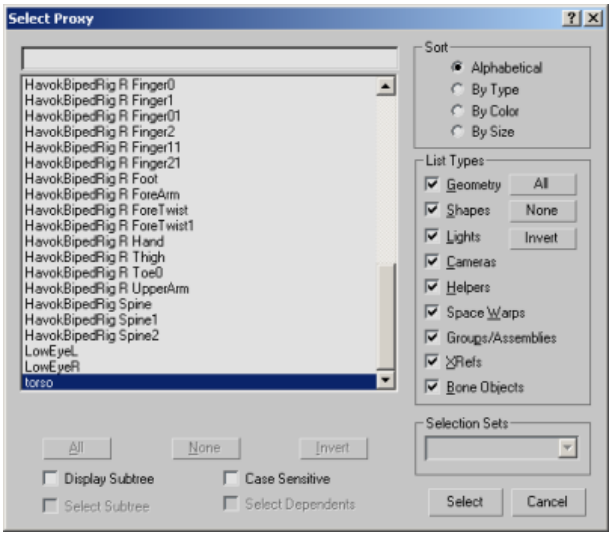
Sometimes a box or a capsule is not enough to accurately represent the volume around a bone or a chain of bones. For example, take the torso region of this model. In this case, instead of creating a box/capsule using the toolbox, we will first create a custom mesh and then associate it with the spine bones.

Create a new (convex) polygon mesh in world space to represent the characters torso, as shown, using any of the standard 3ds Max tools:



To associate this mesh as a chain proxy for the spine bones, select the three spine bones (holding 'ctrl' for multiple selection), then click the **Custom** button in the toolbox. This will open a **Select By Name** dialog where you can pick the existing mesh to use as a proxy. Select your custom mesh and click **Select** :





Once the custom mesh has been selected, it becomes associated as a proxy for the set of spine bones. From this point it behaves like any other chain proxy - i.e: it inherits default Havok properties; it can be replaced by clicking the other chain proxy buttons; it's pivot is aligned to the most senior of the associated bones, etc.

Note that the "Shape Type" property of the new proxy's shape modifier should be set to '*Convex Hull*', to accurately represent a mesh such as this. Verify that this is the case.

Reshaping/resizing Proxies

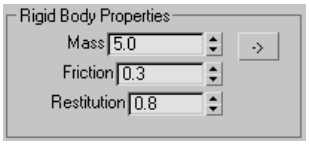
After a proxy has been created, it may not accurately represent the skin volume. To correct this, the proxy can be moved/rotated/etc using the standard tools to achieve the desired fit, with one caveat: **non-uniform scale should be avoided** . The reason for this is that the proxy meshes used in the modeller act as guides for creating the rigid body shapes (based on the 'Shape Type' attributes). If any non-uniform scale is introduced into the proxy meshes then the exported rigid bodies may not accurately match the proxies displayed in the modeller. This is true for capsules in particular.

Therefore the preferred method of editing a proxy is to use the modify panel to change the construction attributes (width/height/etc.) and use the translate/rotate tools to position the proxy.

After moving or rotating a proxy, the **Align Pivots** button (in the 'Associated bone' section of the toolbox) *should always be pressed* to ensure that the proxy pivot matches the joint position. This helps to create accurately mapped rag dolls during the filter process.

Setting Rigid Body Properties

The **Rigid Body Properties** section of the toolbox allows quick access to rigid body properties of selected proxies. As you create proxies, think about these properties and try to set appropriate values for each proxy:



These properties can also be applied in bulk, for example to set the friction property for all of your proxies you can select them all together and input some value into the **Friction** field above.

Finishing with rigid bodies

Using the techniques above, you should have created *simple* proxies for the following bones:

- Head
- Upper Arms
- Hands
- Pelvis
- Thighs
- Calves

and you should have created *chain* proxies for the following bones:

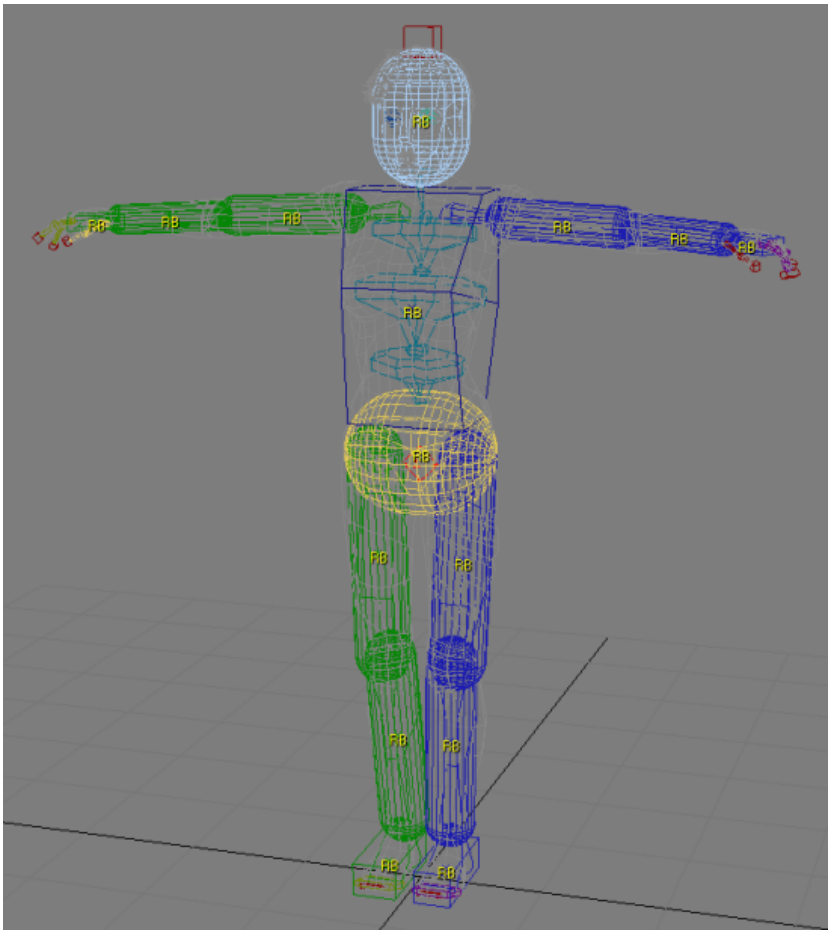
- Feet
- Forearms

as well as a *custom* proxy for the torso.

The pivot of each proxy should be aligned to the associated bone(s), and appropriate rigid body properties should be specified.

You may wish to open the sample " **havokGirl2.max** " file - this contains the original skinned biped with a suitable set of rigid bodies created and associated to the bones:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



6.3.8.2 Creating Constraints

After creating the proxies, we are left with a flat collection of rigid bodies (they are not parented to each other, or connected in any way). To finish setting up the ragdoll, we need to organize the rigid bodies into a constrained hierarchy. The *Constraint Tools* rollout in the Rag Doll Toolbox gives us the tools to achieve this.

Templates

Templates are representations of Havok constraints, saved as text files. Each represents a particular type of constraint, plus information about its spaces, limits, etc.. so that it can be reused between different characters. For example, an "elbow" template might describe a hinge constraint with certain limits and spatial orientation.

Templates are stored together in an specific directory. Although a default set of templates (`defaultHuman`) is provided, in most cases you will be creating your own templates to match the structure of your characters.

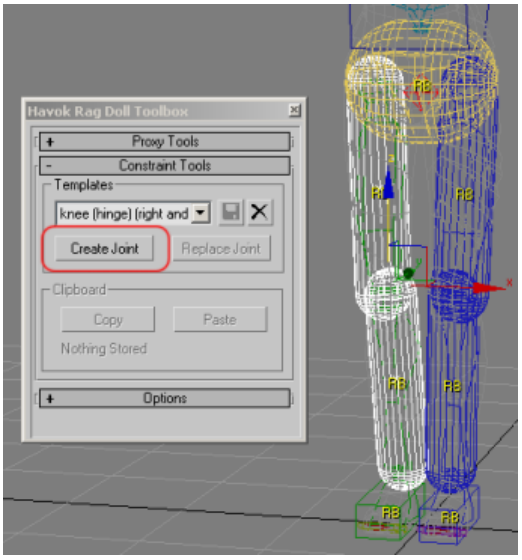
You can change the current location of the templates folder at any time from within the **Options** rollout of the Rag Doll Toolbox.

When working with templates, some assumptions need to be made regarding the local axis of the bones.

In particular, the main axis (an axis pointing towards the length of the bone, in the "outside" direction) and the bend axis (an axis around which a positive rotation (using the right hand rule) causes a bending of the joint) need to be defined. By default these axis are defined as +X and Z, since this is the convention used by Character Studio. You can change the convention used by your characters in the **Options** rollout if necessary.

Applying Templates

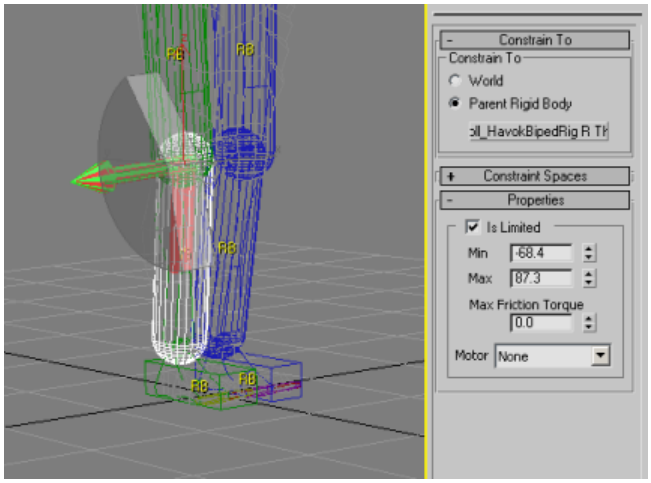
To apply a template, start by selecting two proxies in the viewports. For example, select the left thigh and calf. In the templates combo box, select "`Knee (Hinge) (Right and Left)`" and then click on "*Create Joint*". (You may prefer to use a 'Ragdoll' version of the constraint, which is also provided):



Several things happen at this stage:

- The two proxies become parented to each other - in this case the calf is parented to the thigh
- A constraint (a hinge constraint modifier) is added to the child proxy (the calf)
- The properties of the constraint are set from those stored in the template

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Modifing Constraints

Once a template is applied, you can always modify its properties by changing the constraint parameters through the Modify panel (as shown above). You can then save (and possibly replace) the constraint as a new template by clicking on the " **Save** " icon beside the template combo box.

Replacing Constraints

After a template has been applied, it is possible to replace the applied constraint with a new template. In this case however, you must only select the child object (the one with the constraint modifier), instead of the pair, and then press the " **Replace Joint** " button. No re-parenting of proxies is done in this case.

Copy and Paste

Symmetric characters usually have similar constraint properties on their left and right side. The copy and paste mechanism of the toolbox is useful in this case.

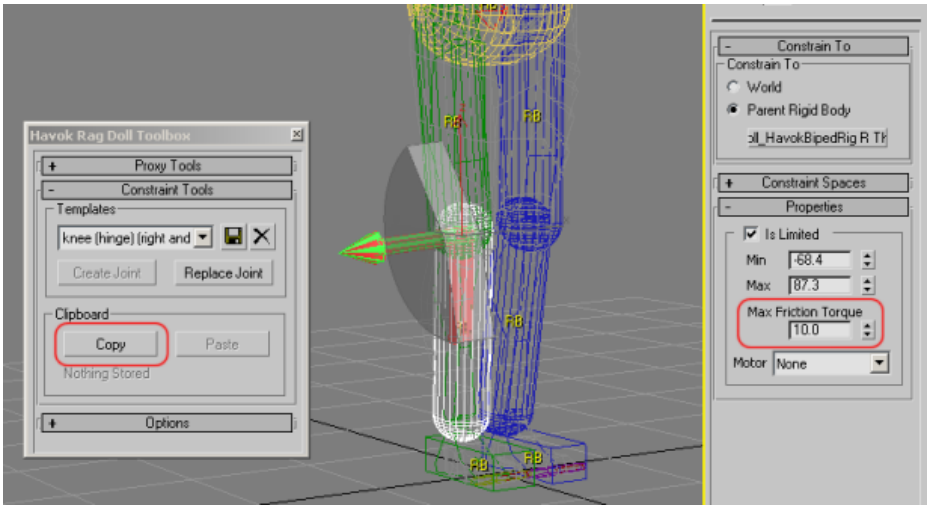
With an already constrained proxy selected, press the "copy" button to store its constraint as a template in memory. The constraint can then be applied to:

- A pair of unconstrained proxies, by selecting the pair and pasting. This behaves in the same way as creating a new joint.
- An already constrained proxy, by selecting the proxy and pasting. This behaves in the same way as replacing an existing joint.

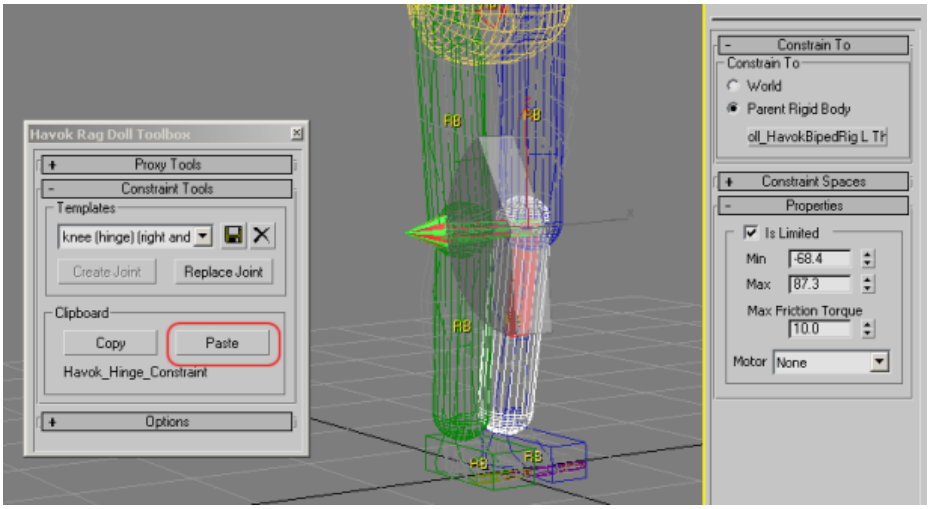
For example, let's say we have modified the left knee friction torque, and we now want to apply the same constraint with the same values to the other knee:

- Select the calf, and change the **Max Friction Torque** value of the constraint to 10
- Click on **Copy** in the Constraint Tools Clipboard

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



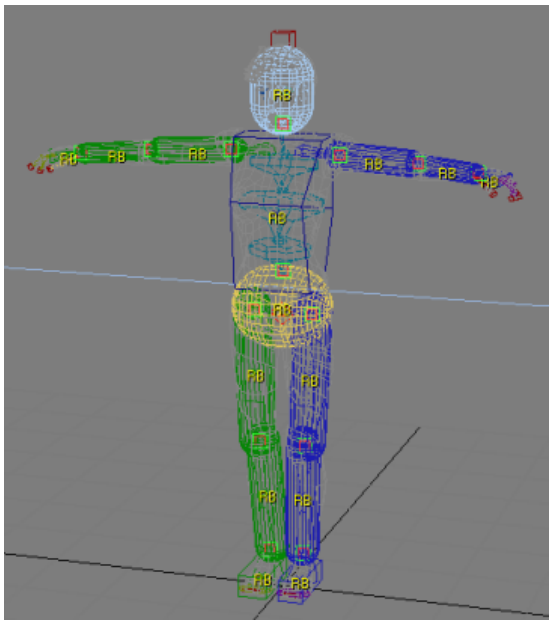
- Select the opposite thigh and calf together in the viewport
- Click on **Paste** in the Constraint Tools Clipboard



Finishing with Constraints

Continue applying templates to each pair of proxies, using the default templates provided (feel free to modify any parameters). You may wish to open the sample " **havokGirl3.max** " file - this contains the final skin, skeleton, and the constrained proxy hierarchy.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

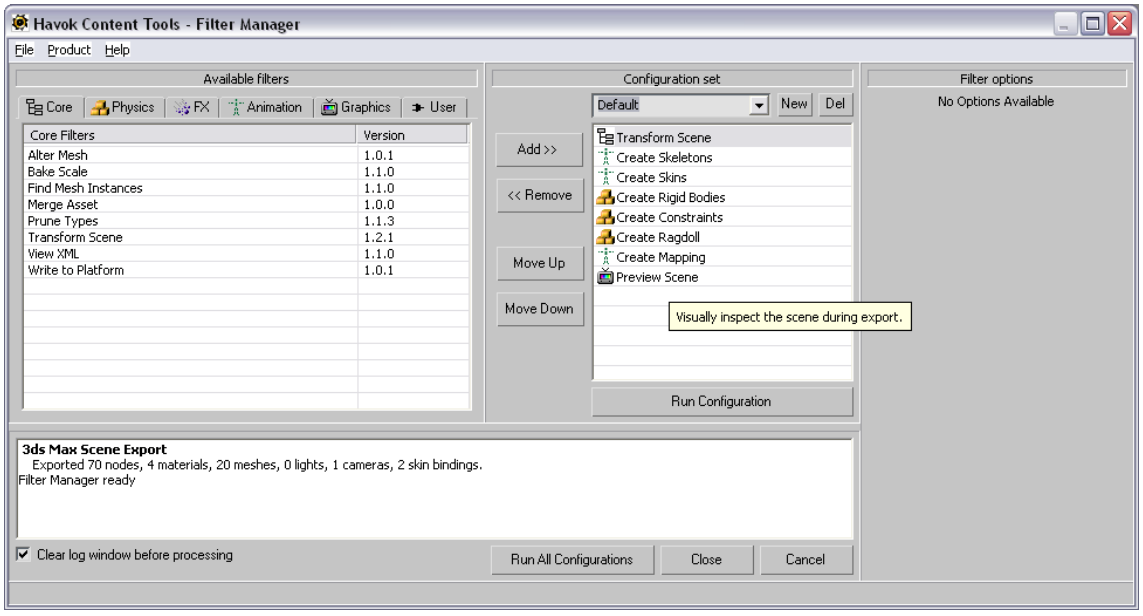


6.3.8.3 Processing the Scene

Now that we have the rag doll set up to match the biped in the scene, we want to process so that:

- The original skeleton and skin are generated
- The rigid bodies and constraints for the ragdoll are generated
- A "ragdoll instance" and skeleton for the ragdoll are generated
- A pose mapper between ragdoll skeleton and original skeleton is generated (so the ragdoll can eventually drive the skin)
- We can preview it

Export the scene to the filter manager using the Havok exporter. The following filter setup is required to process the ragdoll (this setup is included with the " **havokGirl3.max** " file):



- **Scene Transform**

  3ds Max stores the scene internally in inches (by default) and Havok works in meters. Using the **SCALE_INCHES_TO_METERS** preset performs the necessary scaling.

- **Create Skeleton**

  We want to create a skeleton for the bones of the character (so we can skin it). The Create Skeleton filter will detect the skinned skeleton and, with the "From Skin Bindings" option, will create a (36 bone) skeleton for us. Switch off the "Move Reference pose to Origin" as we want the skeleton to remain where it is in the scene.

- **Create Skin**

  This simple filter will create a skin object and associate it with the skeleton we just created.

- **Create Rigid Bodies**

  This filter has no options it will detect the rigid bodies and shapes in the scene and create the actual rigid body objects for the simulation.

- **Create Constraints**

  As with the Create Rigid Bodies filter this has no options - it will create actual physical constraints based on the setup we defined in the scene.

- **Create Rag Doll**

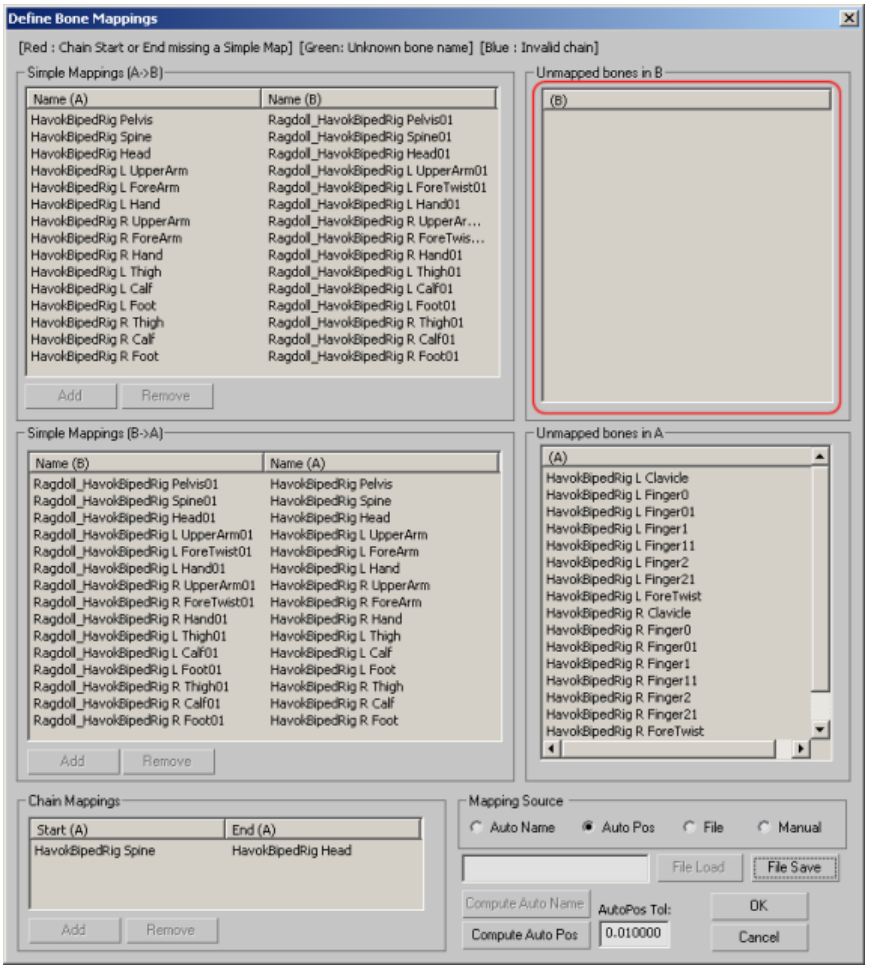
  This filter creates a Ragdoll run-time object (an `hkRagdoll` instance), which contains the rigid bodies, constraints and skeleton associated with a rag doll. Choose ' **Automatically** ' in the ' **Detect Rag Dolls** ' section, which works fine in this case as there is only one rag doll in the scene.

- **Create Mapping**

  This filters sets up skeleton mapper information that allows the transformation of poses between two different skeletons (in our case, the original biped skeleton and the proxy rigid body skeleton).

  For ' **Skeleton A** ', pick 'HavokBipedRig Pelvis' (the skeleton root) For ' **Skeleton B** ', pick 'Ragdoll_HavokBipedRig Pelvis01' (the rag doll root)

  Click on ' **Define Mappings** ' to verify the mapping:



For a successful mapping, all of the bones in one skeleton should be mapped to (some of) the bones in the the other skeleton. In this case we can see that there are no unmapped bones in the 'B' skeleton (the rag doll) so this is a successful mapping.

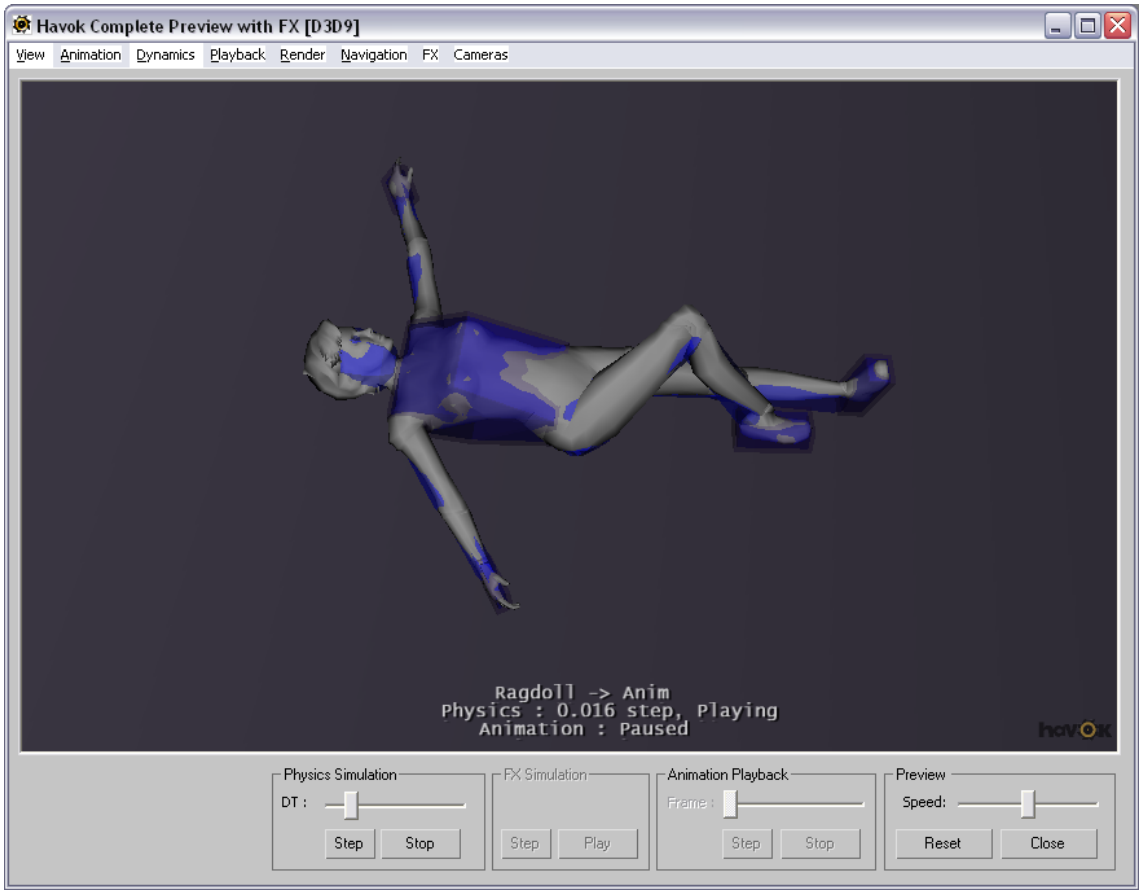
The default behaviour of the mapping filter is to match the skeletons by comparing the positions of the bones in each. This is why it was important to ensure that the pivot points of the rigid bodies were aligned to the pivot points of the joints. If you have any unmapped bones in your skeleton, return to the modeler and ensure that each proxy's pivot is aligned to that of its associated bone(s). Also ensure that ' **Auto Pos** ' is selected as the **Mapping Source** in the **Define Mappings** dialog.

Click ' **OK** '.

- **Preview Scene**

  So that we can interactively test the rag doll behaviour.

Run the filter setup above. When the preview window appears, play the physics and interact with the rag doll using the space bar to pick and drag the rigid body shapes:



If the rag doll does not behave as expected, close the filter manager and review the rigid bodies and constraints. Tweak the attributes if necessary and continue to test using the filter manager. Try appling ragdoll constraints instead of hinge constraints for the elbows/knees. Also try varying the rigid body masses, the constraint's **Max Friction Torque** parameter and the angular limits of the constraints and observe the results in the previewer.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.4 Maya Tools

6.4.1 Introduction

In this chapter we are going to present the Maya specific components of the Havok Content Tools. These include:

- The Maya Scene Exporter
- The Maya Physics Tools
- The Maya Animation Tools

We also present several tutorials:

- Export and Animation Basics: How to set up, export and process an animation.
- Physics Basics: How to set up, export and process a physics scene.
- More On Rigid Bodies : A few more concepts about rigid body setup.
- Rag Doll Setup: How to set up, export and process a rag doll and its association with a high-res skeleton.

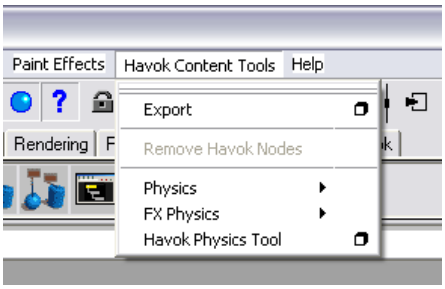
Check also:

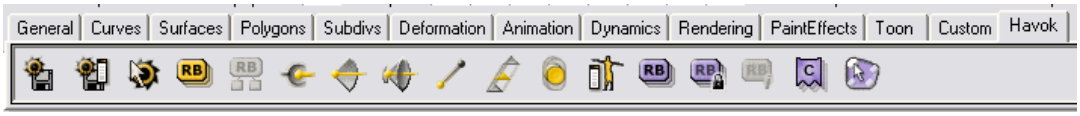
- Extending the Maya Tools: For information about exporting custom data and MEL/C++ access.
- Common Concepts: For common concepts useful for all Havok Tools (regardless of modeler).
- The Havok Filter Pipeline: For common concepts used in processing exported data (regardless of modeler).

6.4.1.1 Loading the Plugins

Currently, the Havok Content Tools for Maya are provided in the form of two plugins - one for the Scene Exporter and one for the Physics Tools. Both of these plugins need to be loaded for the Havok Tools to be useful. If the plugins are loaded then both a **Havok Content Tools** menu and a Havok shelf should be present:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



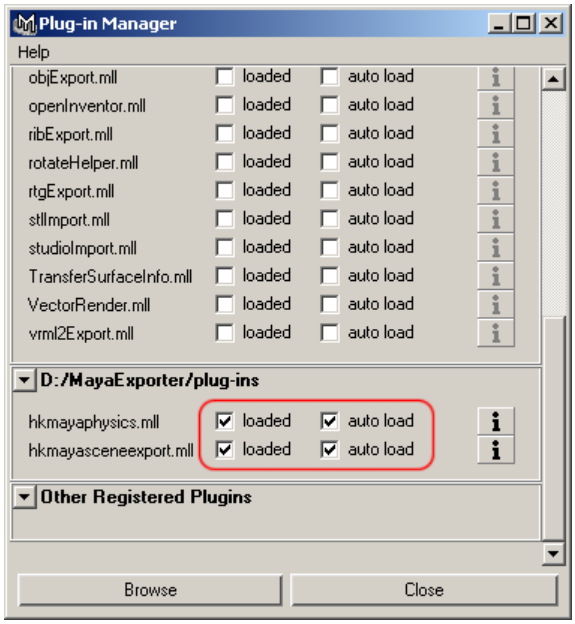


If any of these elements are missing from the Maya UI then it is likely that either one or both of the plugins are not loaded correctly. If they are present then you may skip the rest of this section.

Note:

Depending on the order in which the exporter plugin and the physics plugin are loaded, the order of items in the main menu and the Havok shelf tab may appear different to the images above and throughout this documentation. There is no requirement for either plugin to be loaded before the other, just that both are loaded.

The Havok Content Tools installer installs the Maya plugins, scripts and other files to a 'module' folder, normally under "`%ProgramFiles%\Havok\HavokMayaModules\<x.x>`". It also instructs Maya to look for plugins under this path, by creating a special 'pointer' file at "`%CommonProgramFiles%\Alias Shared\modules\maya\<x.x>\HavokContentTools.txt`": this is a simple text file which specifies the path to the module folder. If these files are set up correctly, then the two plugins ("`hkmayasceneexport.mll`" and "`hkmayaphysics.mll`") will appear in Maya's **Windows > Setting/Preferences > Plugin Manager...** :



If the plugins appear in the plugin mananger window, ensure that the ' **loaded** ' checkbox is checked for each Havok plugin, and also that the ' **auto load'** checkbox is checked to avoid having to load the plugins manually in the future.

If the plugins do not appear in this window then exit Maya, re-install the Havok Content Tools, and re-open Maya.

If they still do not appear, Maya is probably not finding the installed module 'pointer' file. Run the following MEL command within Maya: "`getenv MAYA_MODULE_PATH`" - this gives a list of paths where Maya is looking for module files. Locate the "`HavokContentTools.txt`" file, and copy it to the one of these module paths. Restart Maya again and look for the plugins.

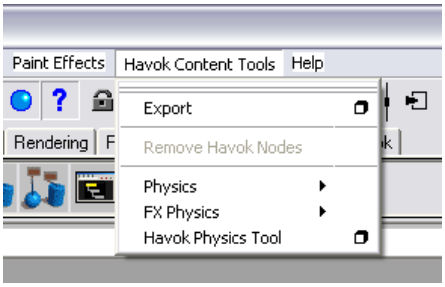
If still not found, then refer to your Maya documentation and environment variables to figure out where the module should be placed.

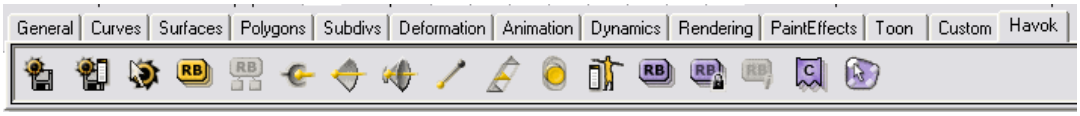
6.4.2 Maya: Scene Exporter

The Maya Scene Exporter is responsible for collecting scene information from Maya's scene graph and passing it to the filter pipeline for processing.

6.4.2.1 Accessing the Scene Exporter

The Scene Exporter is implemented as a custom Maya command, with a set of optional arguments to configure it's behaviour. The best way to access this command and it's options is through the *Havok Content Tools* menu and shelf:





Most of the items in the menu and shelf relate to the Physics Tools and are described in their own section. Let's focus only on the options related to the Scene Exporter:

- *Export*

  When this command is invoked the Scene Exporter will navigate the current scene graph, extracting information about meshes, lights, nodes, attributes, etc (according to the current export options).

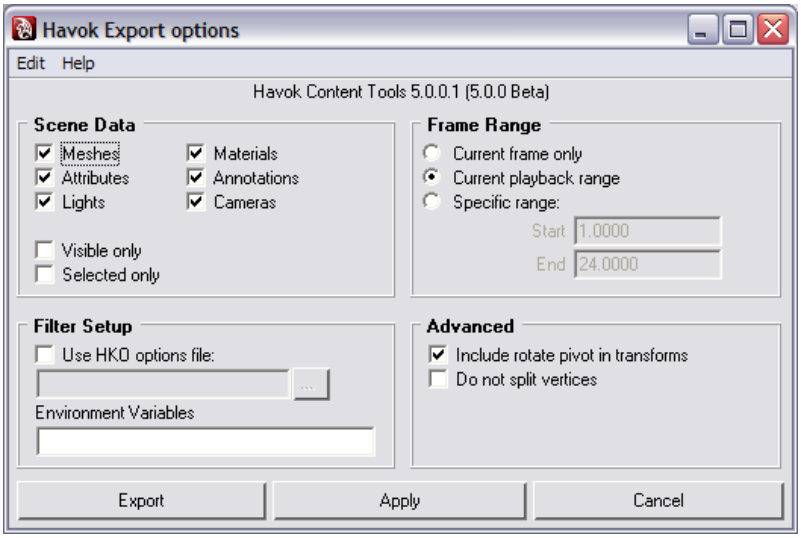
Once this is complete, the Havok Filter Manager will be invoked so that this data can be processed further.

- *Export Options* (the little box beside the 'Export' menu item)

  This command will open the Havok Export Options window, where various export options can be defined. These options are described next.

6.4.2.2 Export Options

The export options UI is displayed by pressing the little box to the right of the **Havok Content Tools -> Export** menu item, or the toolbar button.



The export options are divided into several sections (with default values indicated in the screenshot above):

- **Scene Data**

  This section defines which nodes in the Maya scene should be considered in the export process:

  - **Meshes** : Controls whether meshes are exported
  - **Materials** : Controls whether materials and textures are exported
  - **Attributes** : Controls whether custom Havok attributes are exported - note that some Physics Filters depend on these attributes
  - **Annotations** : Controls whether annotations are exported
  - **Lights** : Controls whether lights are exported
  - **Cameras** : Controls whether cameras are exported
  - **Visible only** : If enabled, only nodes which are currently visible in Maya are exported
  - **Selected only** : If enabled, only nodes which are currently selected in Maya are exported

- **Frame Range**

  This section defines the range over which animated data (node transforms, attributes, annotations) is sampled:

  - **Current Frame Only** : The exporter will sample the scene at the current frame only (so no animation data will be exported). This option can considerably speed up the export process if no animation is required.
  - **Current Playback Range** : Animated data will be exported, with the data sampled over the current playback range in Maya. This is the default option.
  - **Specific Range** : Allows you to specify an frame range (with the **Start** and **End** edit boxes) over which the exporter should sample animated data. This is useful if you want to control the range of the animation export regardless of the current range in Maya.

- **Filter Set**

  This section provides the ability to customize some elements of the filter manager:

  - *Use HKO file* : It is possible to specify an options (HKO) file for the filter manager to use (check the Configurations section for details). Filter configurations are always saved with the asset (the Maya scene), but sometimes it is useful to specify a custom set of options explicitly (for example, during batch processing, or to avoid having to build a filter set up from scratch in new assets). Enabling the check box allows you to choose an HKO file which contains your desired filter configuration(s).
  - *Environment Variables* : Here you can specify pairs of (variable, value) by concatenating strings of the form : **VARIABLE1=VALUE1; VARIABLE2=VALUE2;** etc.. This data is not usually stored with the final asset but can be used by a (custom) filter during process. For example, the Platform Writer filter will replace any environment variables specified in its **Filename** field by their values. For more, see the section on environment variables.

- **Advanced**

  This section provides options which should not normally be changed:

  - **Include Rotate Pivot in Transforms**

    Transforms in Maya do not include the pivot point as part of the transform. However it is desirable for the Havok tools and exporter to do so. With this option enabled, exported nodes will use transforms which include the *rotate pivot*[2] of the Maya nodes.

    The Havok Physics Tools for Maya also use this option. In particular: with this option enabled constraint spaces are considered to be relative to the rotate pivot points of their transform nodes; with this option disabled, constraint spaces are considered relative to the usual Maya transform (normally the center an object).

    This option is enabled by default - this is the recommended setting.

  - **Do Not Split Vertices**

    By default, the Maya Scene Exporter will export as many mesh vertices as required in order to store texture and normals. So, for example, while for a cube only 8 geometric vertices would be required, in order to store per-face normal and texture coordinates, 24 vertices need to be exported.

    Selecting this option will override the default behaviour, and multiple vertices that have the same position will be exported as one, regardless of their normal or UV coordinates. This may reduce the size of the exported mesh - but due to the lack of accurate normal and texture coordinates, the mesh may not display as it originally did in the modeller.

[2] Objects in Maya have both a scale pivot and a rotate pivot, which are noramlly located in the same place. We only consider the rotate pivot.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The export options window also provides several buttons. ' **Export** ' begins the export process using the current options. ' **Apply** ' closes the options window and saves any changes made. ' **Cancel** ' closes the options window, discarding any changes.

6.4.3 Maya: Physics Tools

6.4.3.1 Introduction

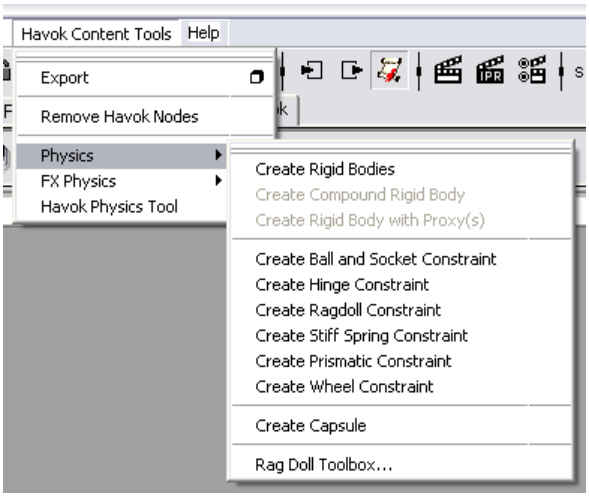
All modeling packages, including Maya, have the ability to specify scene and animation data that has a close relationship to the equivalent run-time objects in Havok (and provide many sophisticated tools to manipulate that data). However, for physical set up the tools available in the modeler are limited and do not necessarily match the Havok run-time features. Moreover, as the construction of complex physical systems (such as rag dolls) can be a complex task, providing tools that automate and facilitate these and another operations becomes more and more important.

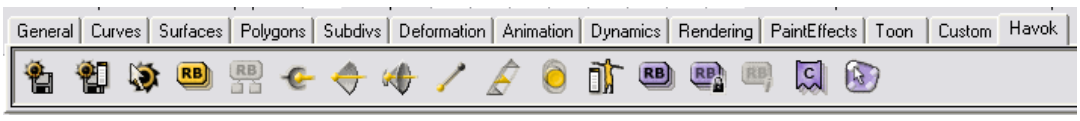
It is for these reasons that the Havok Content Tools include special Maya tools (nodes, scripts, etc) for setting up physical information in the Maya scene. In the following sections we will explore those tools.

Advanced users should first read the Common Concepts chapter for in-depth details on some of the rigid body and constraint concepts.

Accessing the Physics Tools

The physics tools for Maya can be accessed through the **Havok Content Tools** > **Physics** menu or the Havok shelf:





Havok Nodes

Most of the Havok physics data in a scene is added through the use of custom nodes, attached as children of an object's transform node. These custom node types are registered with Maya by the Havok Physics plugin. For reference, the main nodes used by the Havok Physics Tools for Maya are:

- Rigid Body Node: Adds rigid body (dynamics) information to an object
- Shape Node: Adds shape (collision detection) information to an object
- FX Rigid Body Node: Adds FX rigid body (dynamics) information to an object
- Constraint Nodes (Ball and Socket, Hinge, Ragdoll): Add constraint information to an object

The Havok Content Tools also provide some other custom nodes, generally for storing options and some internal data. These node types are recognizable by the common '`hkNode`' prefix.

While it is possible to manually add those nodes to an object (through scripting, via the outliner panel, etc.), it is highly recommended that you use the Havok Content Tools menus, scripts and toolbar to do so as doing otherwise may leave your scene in an inconsistent state.

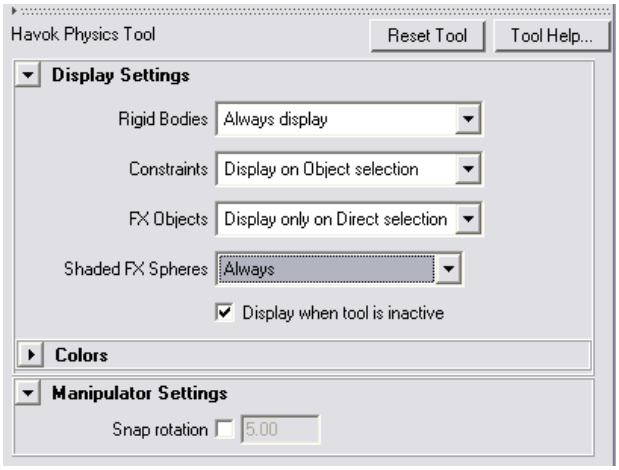
You may remove any Havok Physics nodes from selected objects at any time by choosing the **Remove Havok Nodes** menu option.

The Havok Physics Tool

Some Havok nodes represent themselves in the viewports in various ways. The Havok Physics Tool provides the ability to view and manipulate these Havok nodes.

The tool can be activated by clicking on the **Havok Physics Tool** menu option or shelf button . It is activated automatically whenever a new Havok rigid body / constraint is created, so that the new object appears in the viewports.

While the tool is active, it's icon will appear in Maya's toolbox, alongside the usual (select, move, rotate, scale, manipulate) tools. The tool also becomes mapped to a shortcut key ('*Y*' by default). You can access the tool properties by double-clicking on the Havok Physics Tool icon, or by clicking the little box beside the Havok Physics Tool menu item. This shows the following UI:



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Rigid bodies and constraints are usually only drawn while this tool is active[3]. The *Display Settings* rollout provides further display options for any Havok rigid bodies and constraints in the scene:

- **None** : Do not represent or manipulate such objects in the viewports.
- **Selection** : Represent all such objects in a basic way, and fully represent and allow manipulation of selected objects only (default).
- **All** : Represent all such objects fully in the viewports, and allow manipulation of selected objects.

The *Colors* rollout allows the user to customize the colors used to draw the various elements of rigid bodies and constraint. These settings are stored with the application.

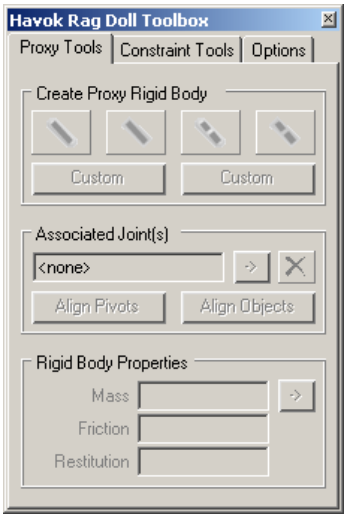
The *Manipulator Settings* rollout allows the user to enable rotation snapping. This setting affects any rotation manipulators in the viewports while this tool is active. This may be useful if you want to set constraint limits to specific angular increments for example.

Constraint display size

Havok constraints in a scene are drawn in the same way, and to a similar scale, as Maya's manipulators - they have a *screen size* as opposed to a world size. This means that Havok constraints will always appear the same size on-screen regardless of the camera viewpoint. *The global display size (for both manipulators and constraints) can easily be changed at any time using the +/- keys on the keyboard.*

MEL scripts/procedures

Maya's scripting language, MEL, is used for several components of the Havok Content Tools for Maya. Firstly, it handles much of the UI (menus, toolbars, attribute layouts, etc.). Secondly, it provides the Rag Doll Toolbox. This is a set of scripted controls which is layered upon the core physics tools. The purpose of the toolbox is to simplify the creation of rag doll representations of skeletal structures - it takes care of common tasks such a fitting rigid bodies to bones, aligning pivots, setting up heirarchies, creating constraints, etc.



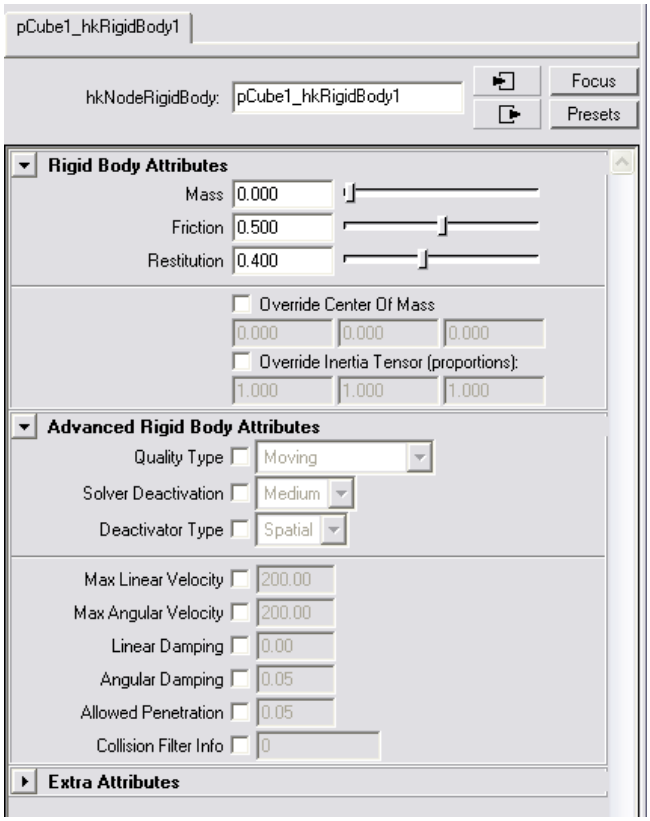
[3] The exception being that Havok nodes are always drawn when directly selected, regardless of whether the Havok Physics Tool is active or not.

6.4.3.2 Rigid Bodies

In this section we are going to describe how to create and manipulate rigid bodies and shapes in Maya. For information about concepts and properties associated with rigid bodies and shapes, please check the Rigid Body Concepts section.

Rigid Body Node

A Havok rigid body node defines the root of a rigid body and contains information on the dynamics of the body. Any object which contains a rigid body node will be converted to a runtime rigid body by the Create Rigid Bodies filter (provided there is at least one associated shape node).



Check the common Rigid Body Properties section for specific details on the attributes listed here.

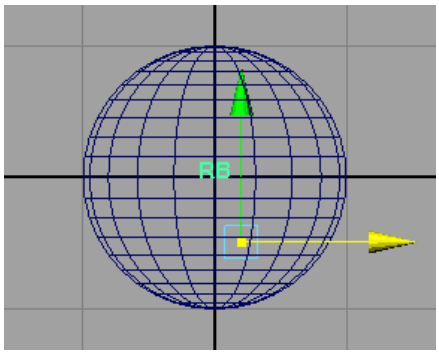
Note that rigid bodies are represent by '*RB*' labels in the viewport whenever the Havok Physics Tool is active. If a rigid body has zero mass, it's label changes to '*[RB]*' to indicate that the body is fixed.

Overriding the Center of Mass and Inertia Tensor

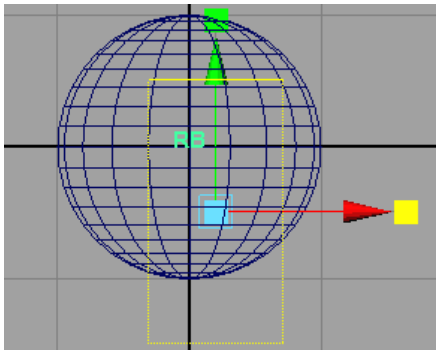
By default, the center of mass and inertia tensor of the rigid body will be automatically calculated by the Create Rigid Bodies filter based on the rigid body geometry and mass distribution. You can, however, override those values and specify them, either by inputting values directly into the attribute editor, or by manipulating them in the viewports.

Selecting the **Override Center Of Mass** check box will display a point in the viewport, which can be

manipulated by activating the Havok Physics Tool and dragging the triad manipulator handles:

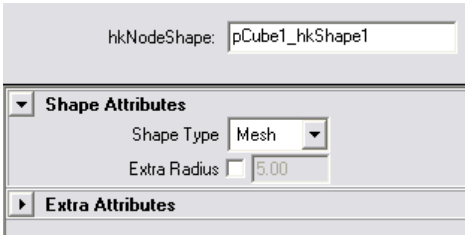


Similarly, selecting the **Override Inertia Tensor Proportions** check box will display a box in the viewport, which can also be manipulated by activating the Havok Physics Tool, in this case by dragging the scale manipulator handles:



Shape Node

A Havok shape node contains information about how an object's mesh will contribute to the collision detection setup of a rigid body. One or more shape nodes must be associated with a rigid body node in order for the Create Rigid Bodies filter to produce a runtime rigid body.



Check the common Shape Properties section for specific details on the attributes listed here.

Whenever a new Havok shape node is created by the Havok Content Tools, its *Shape Type* attribute is automatically determined based on the mesh which it applies to. However it is always useful to verify each shape type attribute to ensure that it matches your requirements.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Shape nodes currently have no representation in the viewports.

Creating a Simple Rigid Body

Most of the rigid bodies you will ever want to create will be simple rigid bodies - where both the rigid body and shape information is associated with the same object.

You can create a simple rigid body by selecting an object in Maya and clicking on the **Create Rigid Body** menu option or shelf button.

This will add both a rigid body node and a shape node to the selected object.

Creating a Compound Rigid Body

A compound rigid body is a rigid body which has more than one shape (more than one collision detection primitive). In a compound rigid body one object has both rigid body and shape information while one or more other child objects add additional shape information.

You can create a compound rigid body by selecting two or more objects in Maya and clicking on the **Create Compound Rigid Body** menu option or shelf button.

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body and shape nodes while each of the other objects will hold a shape node. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new heirarchy.

Creating a Rigid Body with Proxy(s)

A rigid body with proxy(s) is a rigid body where the object which contains the rigid body node doesn't contain any shape node - the shape nodes are stored in one or more child nodes (the proxies). In that sense, a rigid body with proxy(s) is similar to a compound rigid body, with the difference being that the object which has the rigid body node has no shape node in this case.

You can create a rigid body with proxy(s) by selecting two or more objects in Maya and clicking on the **Create Rigid Body with Proxy(s)** menu option.

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body node while the rest of the objects will become the proxies and just hold shape nodes. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new herarchy.

6.4.3.3 Constraints

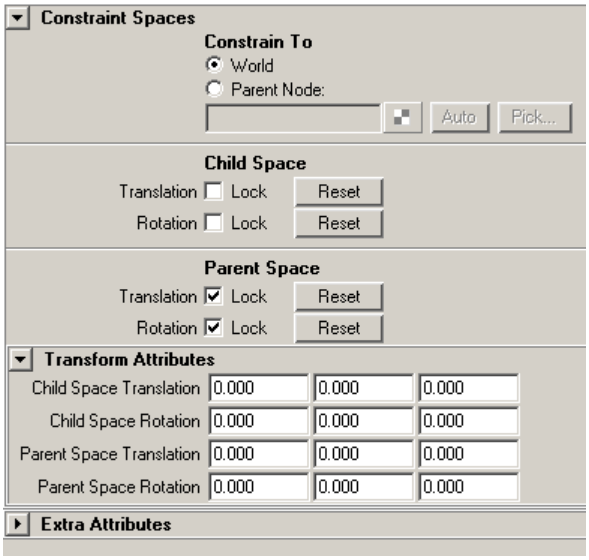
In this section we will present how Havok constraints are created and manipulated in Maya.

Constraint Nodes

A Havok constraint node contains information which will be used to create a runtime constraint on export. Since a constraint must act on a rigid body, a constraint node is useful only the object which contains it also contains a rigid body node. If a pair of rigid bodies are to be constrained, a constraint

node is attached to only one of the bodies (the *child*), and it maintains a reference to the other body (the *parent*). Again, please consult the common Constraint Concepts section of this manual for more details on the child and parent spaces.

There are currently three types of Havok constraint node - the specifics of which are described in the common Constraint Types section of this manual. Each of these types provides the same basic constraint attributes, which allow you to specify the child and parent spaces (each constraint type also provides further attributes relevant to that type only):



'Constraint To' controls

These controls allow you to choose which other rigid body (if any) is the constraint is attached to:

- **World** : The constraint will attach the child rigid body to a fixed location in space.
- **Parent Rigid Body** : The constraint will attach this (child) rigid body to the (parent) rigid body specified in the text field. The parent can be chosen using either of the following methods:
  - **Auto** : This will set the hierarchical parent of the constraint's transform node to be the constraint parent. This is usually the desired setup.
  - **Pick** : This allows you to choose any suitable rigid body in the scene as the constraint parent, via a popup window.

Note:
The parent rigid body specified here is in fact the transform node of a rigid nody node, not the rigid body node itslef. This is to be consistent with Maya's representation of objects as a transform node with of a set of child leaf nodes (mesh / rigid body / shape / etc).

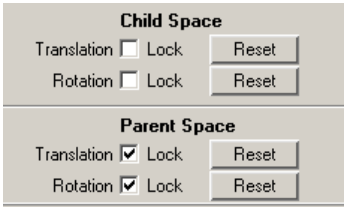
'Constraint Space' controls

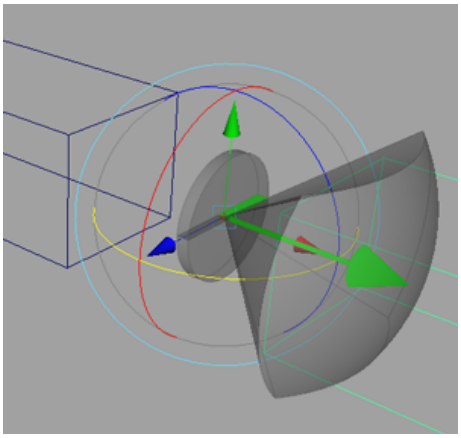
Constraints can be considered as having two distinct spaces: the child (in the local space) and the parent (in the parent object's space). See the common Constraint Spaces section for more details.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

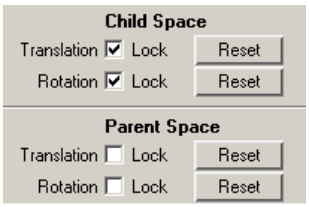
The individual [X,Y,Z] components of each constraint space are always editable, via the 'Transform Attributes' rollout. However it is often easier and more intuitive to manipulate the spaces in the viewport using the Havok Physics Tool.

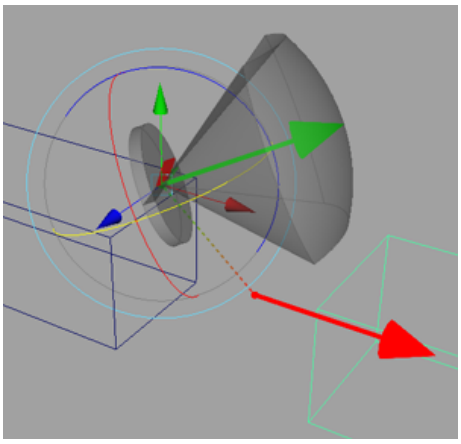
Depending on the state of the **Lock** buttons in the **Constraint Spaces** rollout, various methods of viewport manipulation become available. If the parent space attributes are locked and either of the child space translation or rotation attributes is unlocked, the constraint spaces can be manipulated as a single entity - both the child and parent spaces move together:





If the child space attributes are locked and the either of the parent space translation or rotation attributes is unlocked, the constraint parent space can be manipulated independantly of the child space:





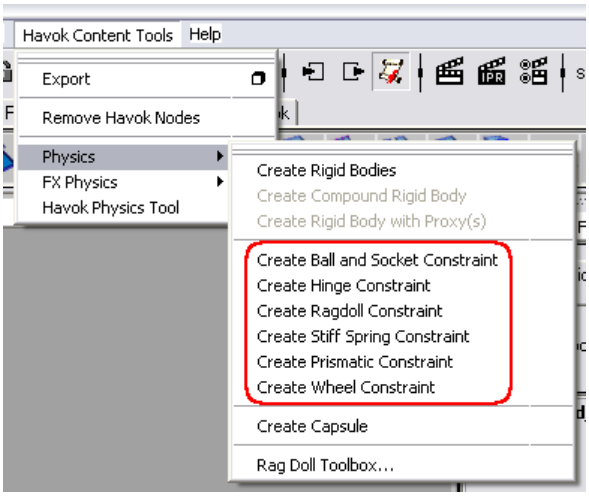
In all cases, the **Reset** buttons return a constraint space attribute to its initial setting, moving the constraint space to the pivot point of the appropriate object.

Note:

If all of the child and parent space attributes are unlocked, then it is possible to fully manipulate both spaces independantly and at the same time. However this can become difficult since you now have four manipulators which be be difficult to distinguish from each other. It is best to unlock a space only when you wish to alter it, and lock it again once you are satisfied with the change. Of course, if you want the parent space to move with the parent object in the modeler, then the parent space must always remain unlocked.

Creating a Constraint

You can create a constraint by using either the **Havok Content Tools** menu or shelf buttons:



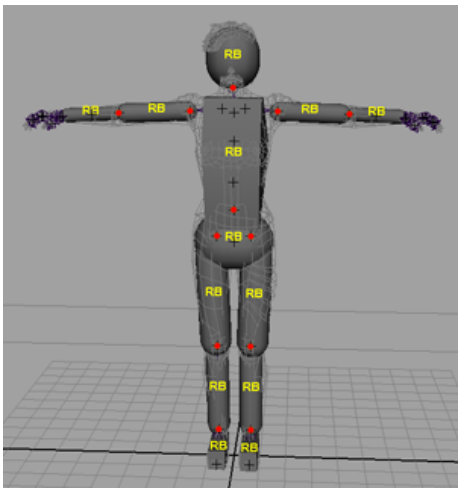
Either one or two rigid body objects must be selected before creating a constraint. Then, depending on the number of objects selected:

- If one object is selected, a constraint node will be attached to that object. If the object's heirarchical parent is also a rigid body then that object will be set as the constraint parent; otherwise the rigid body will be constrained to the world.
- If two objects are selected and one is a descendant of the other, a constraint node will be attached to the descendant and the ancestor will set as the constraint parent. If the objects are not explicitely parented, a constraint node will be attached to the first rigid body and the second will be set as the constraint parent.

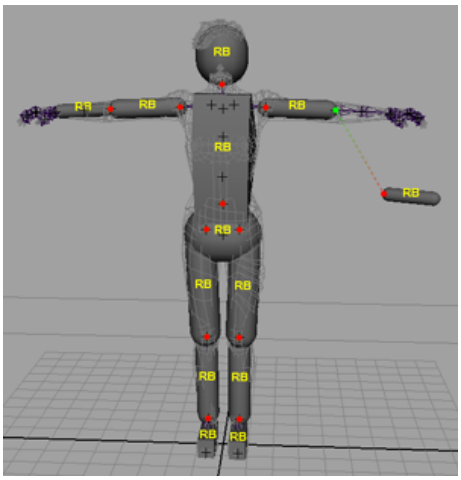
A single rigid body object can have any number of constraints applied to it.

Constraint Display

With the Havok Physics Tool active, constraints are displayed in the viewports as two dots representing the location of the child and parent spaces of the constraint. In most cases, these two spaces should be aligned, so both dots are drawn at the same location:

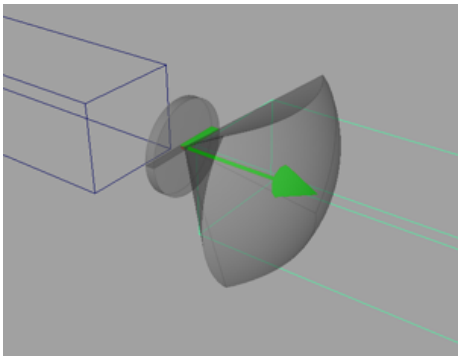


If the parent and child spaces are misaligned, a dotted line is drawn between them to indicate the offset:



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

When a constraint node is directly selected or the Havok Physics Tool is active with appropriate settings, a 3D representation of a constraint will be displayed to represent the constraint spaces and whatever other properties may be present (such as angular limits):



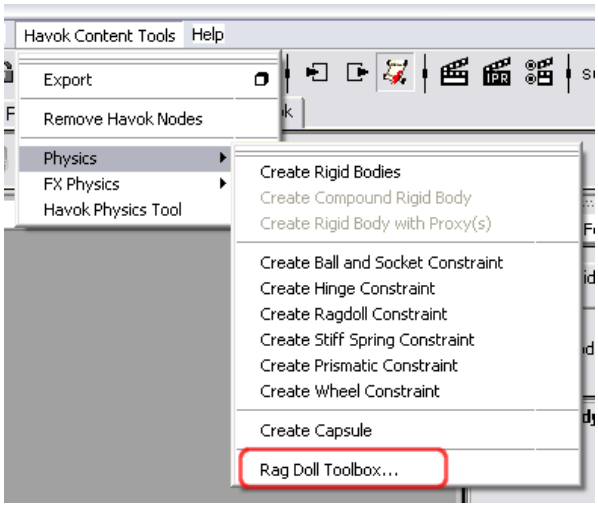
Constraints are displayed in the same way as manipulators, in that they have a *screen size* as opposed to a world size - constraints will always appear the same size on-screen regardless of the camera viewpoint or properties. *The global display size (for manipulators and constraints) can easily be changed at any time using the +/- keys on the keyboard.*

6.4.3.4 The Rag Doll Toolbox

Introduction

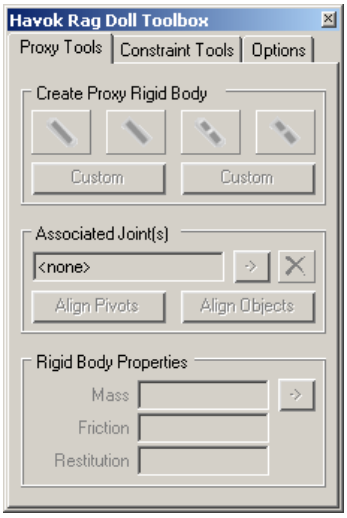
One of the most common applications of rigid body simulation is the creation of articulated systems of rigid bodies representing characters (rag dolls). This usually involves the creation and alignment of rigid body proxies for the character's bones, as well as the setup and alignment of constraints that represent which character's joints. The Rag Doll Toolbox provides a set of tools to facilitate those operations. It is built entirely as a MEL tool that works on top of the lower-level physics tools.

You can open the Rag Doll Toolbox using either the **Havok Content Tools** menu or shelf:



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The toolbox appears as a floating window with several tabs: the *Proxy Tools* are used to create and associate rigid bodies with bones; the *Constraint Tools* are used to organize and constrain these rigid bodies together:

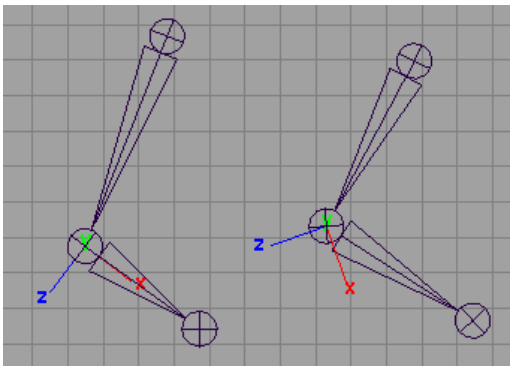


Several default properties of the Rag Doll Toolbox can be customized to suit your implementation and to ensure consistency across different installations of the Havok Content Tools - see Customizing the Maya Rag Doll Toolbox for more.

Issues with Joint Spaces in Maya

Before describing the various Rag Doll tools, it is worth mentioning a problematic issue which is specific to Maya.

Maya works in terms of *joint's* rather than *bone's*. These joints may be assigned any orientation. For example, both of the following scenarios are valid within Maya:



In the first case, the X axis of the central joint points along the smaller bone. In the second case, there is no axis pointing along this bone. Therefore the local space of a joint in Maya does not necessarily

correspond to the space of any bone which connects from the joint to a child joint. This indeterminism causes several problems for the Rag Doll Toolbox:

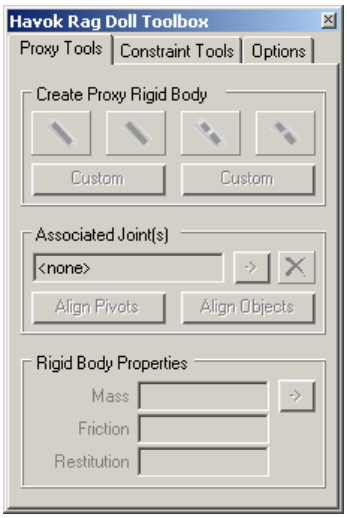
- When we create a new proxy associated to a bone, we need a 'bone basis' in which the proxy should be constructed. If we do not have a deteminate bone basis then it becomes difficult to create a proxy with the same basis as the bone.
- When we apply a templated constraint between proxies, that template would have been generated relative to some proxy/bone basis. Since proxy bases are indeterminate, the orientation of any applied template will often not end up as expected.

If the joint spaces in your rig do in fact consistently match those of the bones in you rig, then any such problems can be avoided. However, this is usually not the case. An '*Override joint spaces*' option is provided by the Rag Doll Toolbox which attempts to provide a workaround for this problem. If this checkbox is disabled, the Rag Doll Toolbox always takes the joint space to be that of the bone. If the checkbox is enabled, the toolbox attempts to determine a suitable bone space. To determine a bone space, the Rag Doll Toolbox looks for a single child joint of the joint we are interested in. If one exists, then we rotate the joint space so that it's X axis points toward the child. If none exists, then we just use the joint space as-is.

If you are sure that your joint spaces are set up correctly, then it is safe to disable this option before working with the rag doll toolbox. Otherwise it is recommended that you leave this option enabled.

Proxy Tools

The first task when creating a rag doll representation of a character is to construct a set of rigid bodies to represent a selection of the bones of the character. These rigid bodies are usually collision primities such as capsules and boxes, which are very efficient to simulate by the physics runtime (capsules in particular).



The Proxy Tools provide the following functionality:

- Given a bone, create a proxy representation (a box, a capsule, or a custom rigid body) and align and associate the proxy with the original bone.

- Given a chain of bones, create a single proxy representation of that chain and align and associate that proxy with the chain of bones.
- Given a proxy rigid body (associated with a bone or chain of bones), replace it with another proxy.
- When creating proxies, it is very important that their pivots are aligned to those of the bones they represent, since, at runtime, movements on the bones will be converted to movement on the rigid bodies and vice versa. The proxy tools track this association and provide the ability to realign the proxy pivot/object at any time.
- Quickly access and modify the basic properties of one or multiple rigid bodies (mass, restitution and friction).

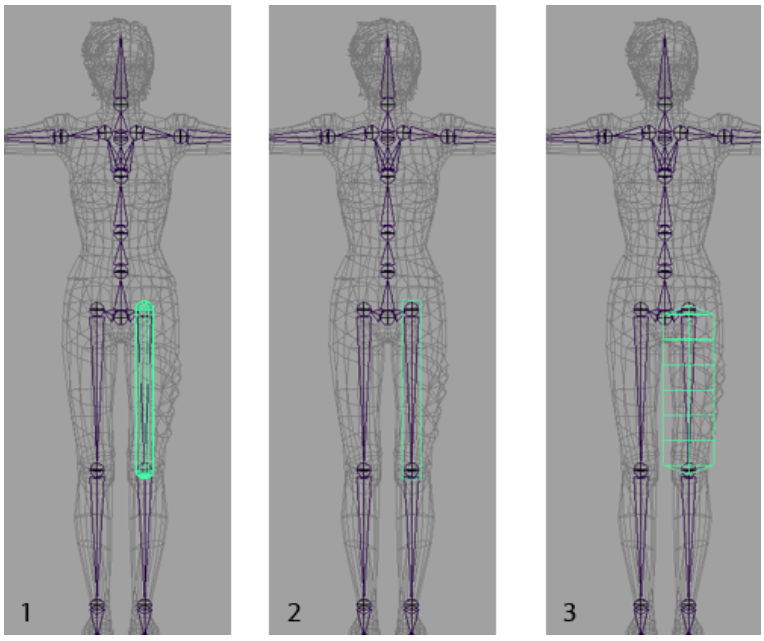
Create/Replace Proxy Rigid Body

These buttons allow you to create proxies and associate them with bones:

When a rigid body proxy (either single or chain proxy) is selected, these "Create Proxy" buttons become " **Replace Proxy** " buttons - the proxy they create will replace the selected proxy, and the association with the original bone will be updated.

Note:
When proxies are created, they should have the same space as the bone with which they are associated. However this is not always possible in Maya. See Issues with Joint Spaces for more.

The first set of buttons (on the left) create *simple proxies* - where each rigid body is associated with a single bone.



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

1. **Create Capsule Proxy(s)**

   With one or more bones selected, this will create a capsule rigid body bounding each bone selected. It will also associate and align each capsule with each bone.

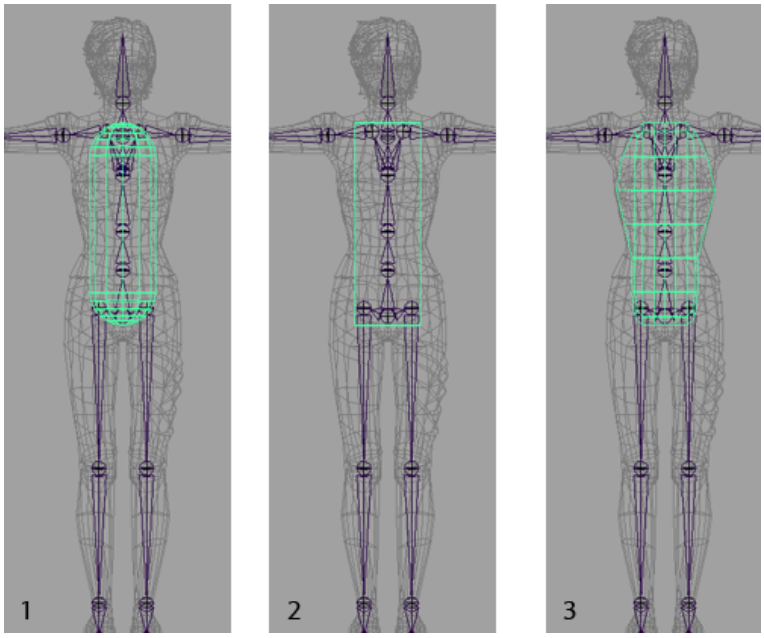
2. **Create Box Proxy(s)**

   With one or more bones selected, this will create a box rigid body bounding each bone selected. It will also associate and align each box with each bone.

3. **Associate Custom Proxy**

   With a single bone selected, this will open a popup window where you can select any mesh object to become the proxy rigid body for the selected bone. It will then associate and align that rigid body with the selected bone.

The second set of buttons (on the right) create what we call *chain proxies* - rigid bodies which are associated to a set of bones in hierarchical order. Chain proxies are used in order to create single rigid bodies to represent multiple bones, such as a rigid body bounding multiple spine links.



1. **Create Capsule Proxy(s)**

   With a chain of bones selected, this will create a capsule rigid body bounding all of them. It will also associate and align the capsule with the chain of bones.

2. **Create Box Proxy(s)**

   With a chain of bones selected, this will create a box rigid body bounding all of them. It will also associate and align the box with the chain of bones.

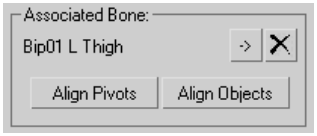
3. **Associate Custom Proxy**

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

With a chain of bones selected, this will open a popup window where you can select any mesh object to become the proxy rigid body for the chain of bones. It will then associate and align that rigid body with the chain.

Bone Associations

These controls are used to display and edit the associations between rigid body proxies and bones. Thus, they are only enabled when one or more rigid body proxies (rigid bodies created by one if the "Create Proxy" options above) are selected:



When a single proxy is selected, the name of the associated bone(s) is displayed below the **Associated Bone / Associated Chain** box.

The **Select Associated Bone(s)** button will select the bone(s) associated with the selected proxy.

The **Remove Association** button will remove the current association between the selected proxy and the bone or chain of bones. The proxy will remain as a rigid body, but the association with the original bone(s) will be lost.

The **Align Pivots** button will ensure that the rotate pivot of the proxy matches the pivot of the associated bone (or the first bone of the associated chain), without actually moving the proxy. Use this button whenever you have modified a proxy and want to ensure that the pivot is aligned:

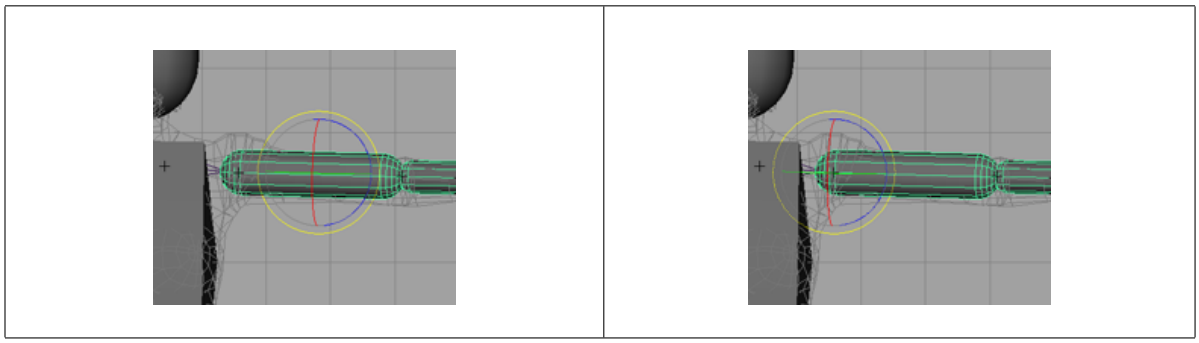


Table 6.3: Aligning Pivots

The **Align Objects** button will ensure that the pivot of the proxy matches the pivot of the associated bone by moving and/or rotating the proxy rigid body. Use this button whenever you move or rotate a bone and want to update the associated proxy accordingly:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

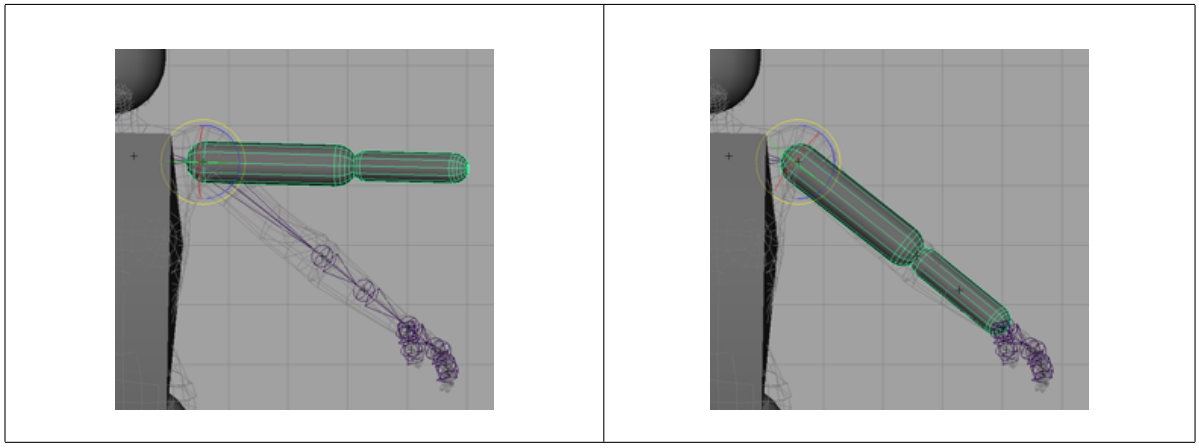
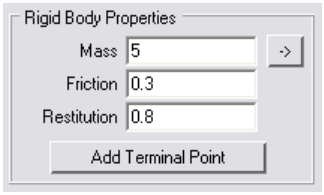


Table 6.4: Aligning Objects

Rigid Body Properties

These controls are enabled whenever one or more rigid bodies (bone proxies or not) are selected. They allow you to modify the basic rigid body properties of the selected objects easily, and are particularly useful when you want to modify multiple rigid bodies at the same time.



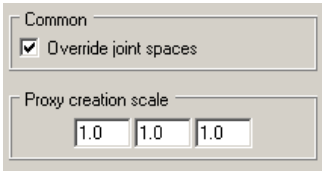
The **Open RB Modifier** button will open the rigid body node associated with the selected rigid body in the attribute editor (eg. so that further properties can be edited).

The **Add Terminal Point** button is used to add extra leaf bones to skeletons to be used for raycasting, specifically to prevent ragdoll penetration with landscapes using the hkaDetectRagdollPenetration utility.

Selecting a ragdoll proxy and clicking the add terminal point button creates a locator node as the child of the proxy. The locator is positioned on the local bounding box of the proxy at the center of the face furthest from the proxy pivot.

Proxy Creation Options

The 'options' tab of the toolbox provides some options which relate to proxy creation:



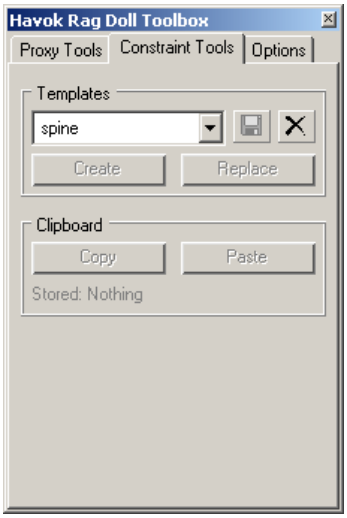
The '*Override joint spaces*' option has been explained already - see Issues with Joint Spaces for more

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Whenever new proxies are created, they are automatically sized to fit the bones which they associated to. However, bones are often much narrower than the skin volume and so the proxies may not be wide enough. A '*Proxy Creation Scale*' option is provided for this reason. This is an additional scaling factor which is used during the creation of new capsule/box rigid body proxies - it can be though of as a local [X,Y,Z] scale applied to a bone space before a new proxy is created to fit around it. Normally, X points along the bone axis.

Constraint Tools

Once the rigid body proxies have all been created, the next step in setting up a rag doll is to organize them into a constrained heirarchy to match that of the original skeleton.

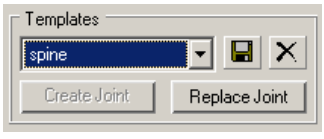


The Constraint Tools provide the following functionality:

- Given a pair of proxies, reparent them in a way which reflects the original skeleton, and apply a constraint between them.
- Save and reuse constraint setups between characters, as *template* files.
- Easily replace a constraint with another templated version.
- Quickly copy and paste constraints within the same scene.

Templates

The Rag Doll Toolbox provides the ability to reuse constraint setups through the use of *templates*. These are text files which contain a type of constraint and it's attribute values.



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The dropdown list show all the available templates - this list is built by looking at all templates stored in the current *templates folder*, the location of which can be changed in the options section of the Rag Doll Toolbox.

The **Save Template As...** button is enabled only when a rigid body with a single constraint is selected. It allows you to save that constraint and its attribute values as a new template. Templates are saved as `*.txt` files.

The **Remove Template** button will permanently remove the selected template (i.e., it will delete the template file from the templates folder).

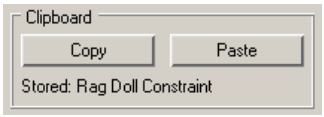
The **Create/Replace** buttons are those which do the main work, and are enabled only when two rag doll proxies are selected / an already constrained proxy is selected. When either of these buttons is pressed the following steps take place:

1. Ensure that heirarchy of the selected proxies is correct, by looking at the heirararchy of the associated bones. If not correct then reparent the proxies accordingly.
2. Create/replace a constraint node on the 'child' object - the one which is lower down in the heirarchy. The type of constraint which should be created is stored in the template file.
3. Set the constraint's parent object as the one higher up in the hierarchy.
4. Set the constraint's attribute values based on those stored in the template.

Note:
When working with templates, the orientation of applied templates is based on the relative space in which the template was created. Due to Issues with Maya's Joints, it is often necessary to rotate new constraints into the correct orientation after they have been applied from a template.

Clipboard

While working on a symmetric character's constraints, it is often desirable to replicate constraints on each side of the character. The clipboard provides a quick way to create a temporary template which can then be applied throughout the rest of the character.



The **Copy** button is enabled only when a rigid body with a single constraint is selected. It will create a template of that constraint in memory, and display the type of the stored constraint for reference.

The **Paste** button operates as the **Create/ Replace** template buttons (depending on the selection), with the difference that it applies the template stored in the memory rather than the template selected in the drop down list.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.4.3.5 The Convex Hull Tool

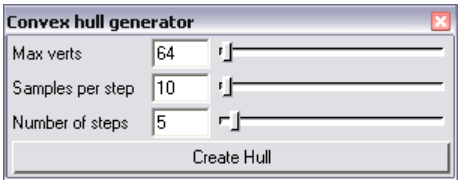
In this section we describe a simple tool for generation of convex hulls from selected meshes. This tool generates an approximate convex hull from the currently selected objects (either meshes or nurbs) with a user-specified maximum number of vertices **Max verts** . The hull is approximate in the sense that it does not necessarily enclose the selected meshes, but is a reasonably good approximation (using the metric of volume difference) which satisfies the constraint on the maximum number of vertices. This is useful in a number of contexts, such as generation of assets for platforms which require <64 vertices per convex body. If the specified maximum exceeds the number of vertices in the true hull, the tool just generates the true hull. Note that if multiple meshes/nurbs are selected, the hull tool generates a single hull from all of their vertices combined.

The approximate hull is found by first computing the original set of vertices in the true hull. If the number of vertices in the original hull is N, and the desired maximum number of vertices in the generated hull is M, the tool has to remove R=(N-M) vertices. In the case when the number of steps equals 1, a trial is done which consists of randomly choosing a sample of R of the true hull's vertices, and computing the hull obtained by removing that sample from the original set. A user-specified number **Samples per step** of such trials are done (with a default of 10), and the trial hull obtained which had the minimum difference in volume from the true hull is taken as the final choice. This constitutes one 'step'.

Rather than removing all R vertices in a single step, the user can specify that the the vertices be removed in a sequence of X steps, where X is the parameter **Number of steps** . On each step roughly R/X vertices are removed (if X exceeds R, each step removes only one vertex). Increasing the number of steps will clearly lead to hulls which better approximate the true hull, but will take longer to finish. The default is 5 steps.

The convex hull tool window simply consists of fields for the three parameters **Max verts** , **Samples per step** , and **Number of steps** described above, and a **Create Hull** button to generate the hull using these parameters. Generated hulls are placed in a transform node in the scene root with the name "foo_hull" where "foo" is the name of the first object in the selection. Subsequently generated hulls accumulate in the scene root but are given unique names. The convex hull tool window remains open after hull generation, so that different parameters can be tried by simply undoing after clicking Create Hull and changing the parameters.

On clicking Create Hull, a progress bar appears which allows the hull generation to be aborted if it is taking too long. On aborting, the best hull at the current point in the computation (i.e. the hull from the current step with minimal volume difference so far) is generated.



6.4.4 Maya: Animation Tools

Since Maya already provides a wealth of tools for creating content for animation playback, the Havok Content Tools use the features provided by Maya without the need of providing custom extensions (as it does for the physics). Concepts like skeletons, bones, skins and animations in Maya transfer to the same concepts/objects in the Havok SDK. The following section outlines one of the less obvious connections

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

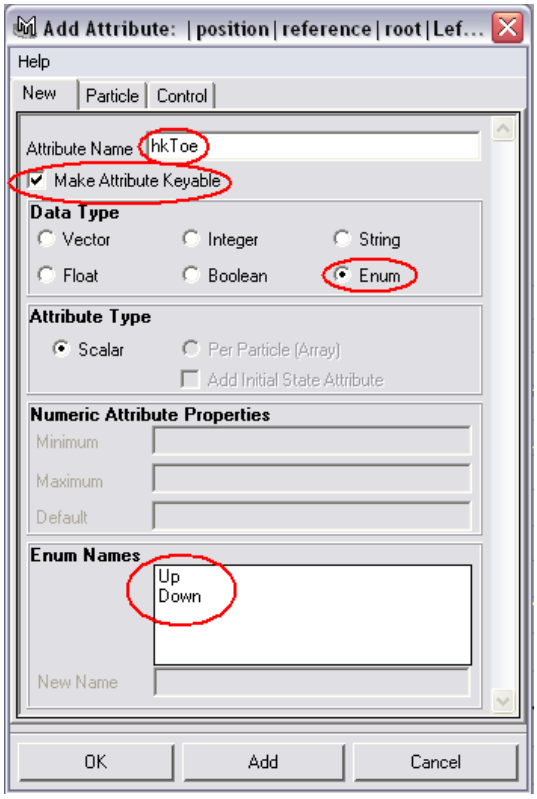
between Maya and the Havok Animation SDK.

6.4.4.1 Annotations in Maya

Please check the common Annotations section for details on annotations in the Havok Animation SDK.

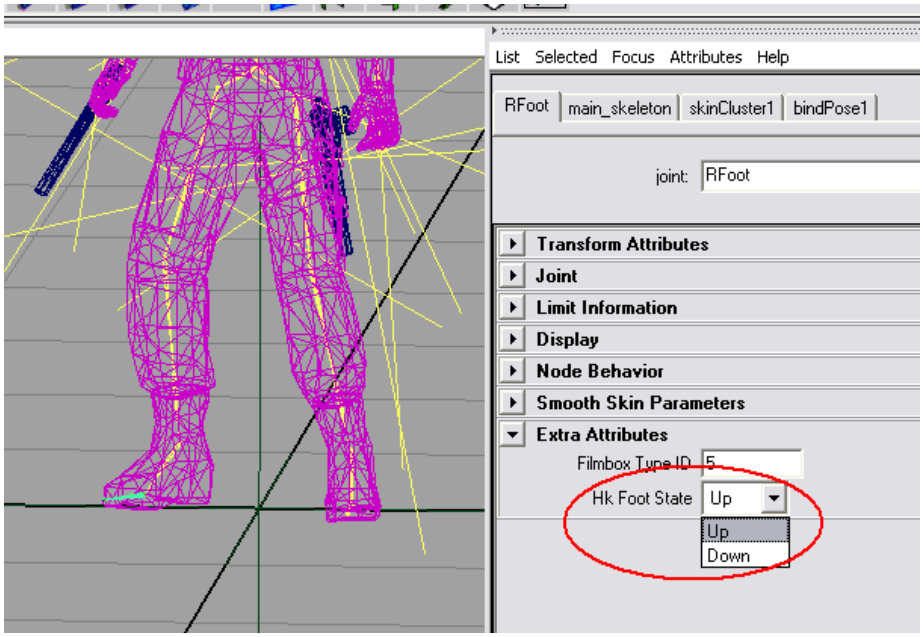
Maya does not explicitly support annotations, so dynamic attributes are used instead. Any compatible dynamic attributes owned by bones in the scene are converted to annotations at export time. The following steps explain how to create such attributes:

1. Select the bone to which you wish to add the annotation. Select *Modify* > *Add Attribute...* in the main Maya menu or *Attributes* > *Add Attribute...* in the Maya Attribute Editor to raise the following dialog:



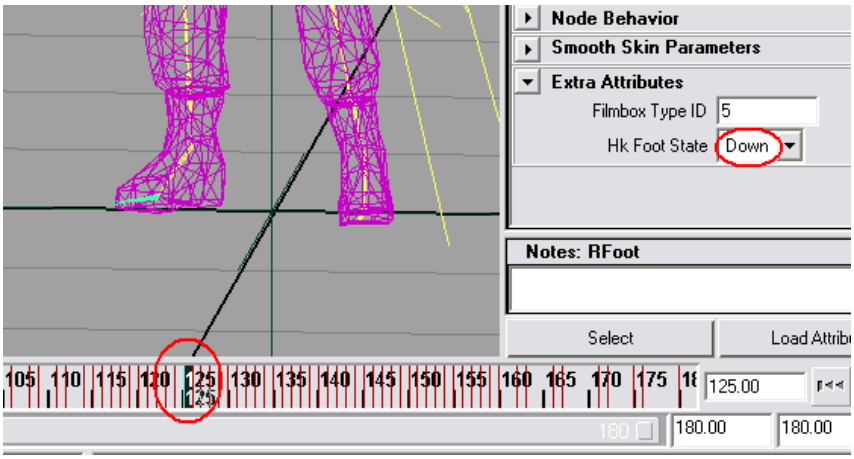
   Create an attribute whose name begins with 'hk', and ensure it is both keyable and enumerated. Populate the `Enum Names` dialog with the string values you wish to annotate the bone with, and click the `Add` button.

2. Once you have added an attribute, it will be displayed in the attribute editor window, under `Extra Attributes`:



The dropdown displays the value of the new attribute at the current frame. Expanded, it displays the list of annotation values to choose from that you set up earlier.

3. To create an annotation, you need to create a keyframe at the desired animation frame. To do this, first set the time slider to the desired frame and choose the desired annotation value:

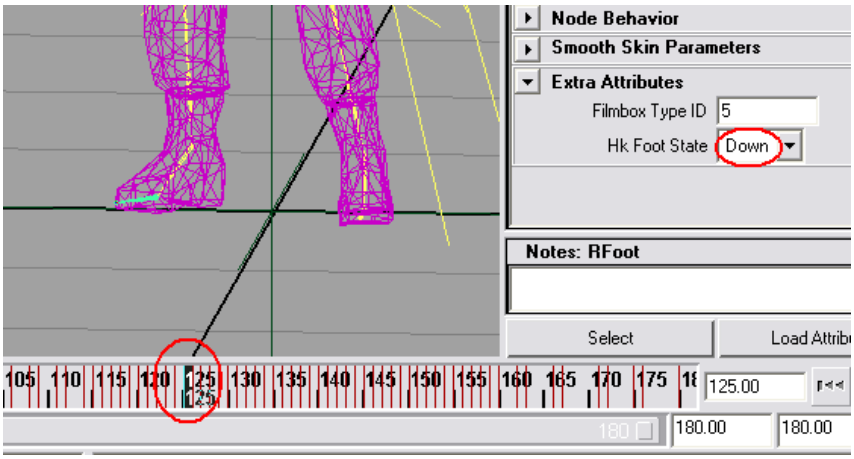


To create a keyframe, reselect the bone in the model view and press the ' **S** ' key. A vertical red line should appear in the time slider where the key has been created. Note that if you've changed the value of the enumerated attribute just before you press the ' **S** ' key, the keyframe won't be created because the attribute dropdown will have keyboard input focus. This is why you should reselect the bone in the model view before creating the keyframe.

4. Create as many keyframes as you like, using changing the value of the enumerated custom attribute as you go. Note that a keyframe with the same value as the previous keyframe is unnecessary, and won't be exported as an annotation. The value of the custom attribute at the current frame is always displayed in the attribute editor, but it doesn't change during animation playback or when moving the frame slider by hand, only when animation is stopped.

   You can view the annotations you've created in the Persp/Graph view. Simply click the Persp/Graph button on the left hand side of the screen, and select the custom attribute to see a graph of enumeration value against time.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



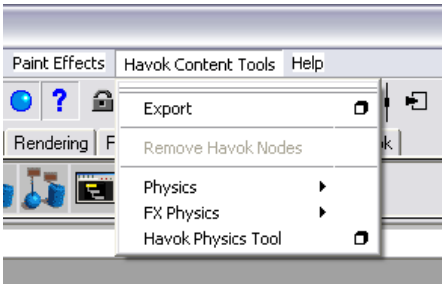
5. At export time, annotation strings are created by appending the name of the enumerated value to the name of the custom attribute. For example, a custom keyable enumerated attribute called '`hkFootState`' with enumerated values '`up`' and '`down`' will yield annotations called '`hkFootStateUp`' and '`hkFootStateDown`'.

6.4.5 Tutorial: Export and Animation Basics

In this tutorial we are going to explore the basic operations involved in exporting and processing an asset. We are going to use the example of exporting skeletons, animations, etc.. but the principles regarding how assets are exported, the filter pipeline and the use of the filter manager and individual filters for processing applies to all Havok products.

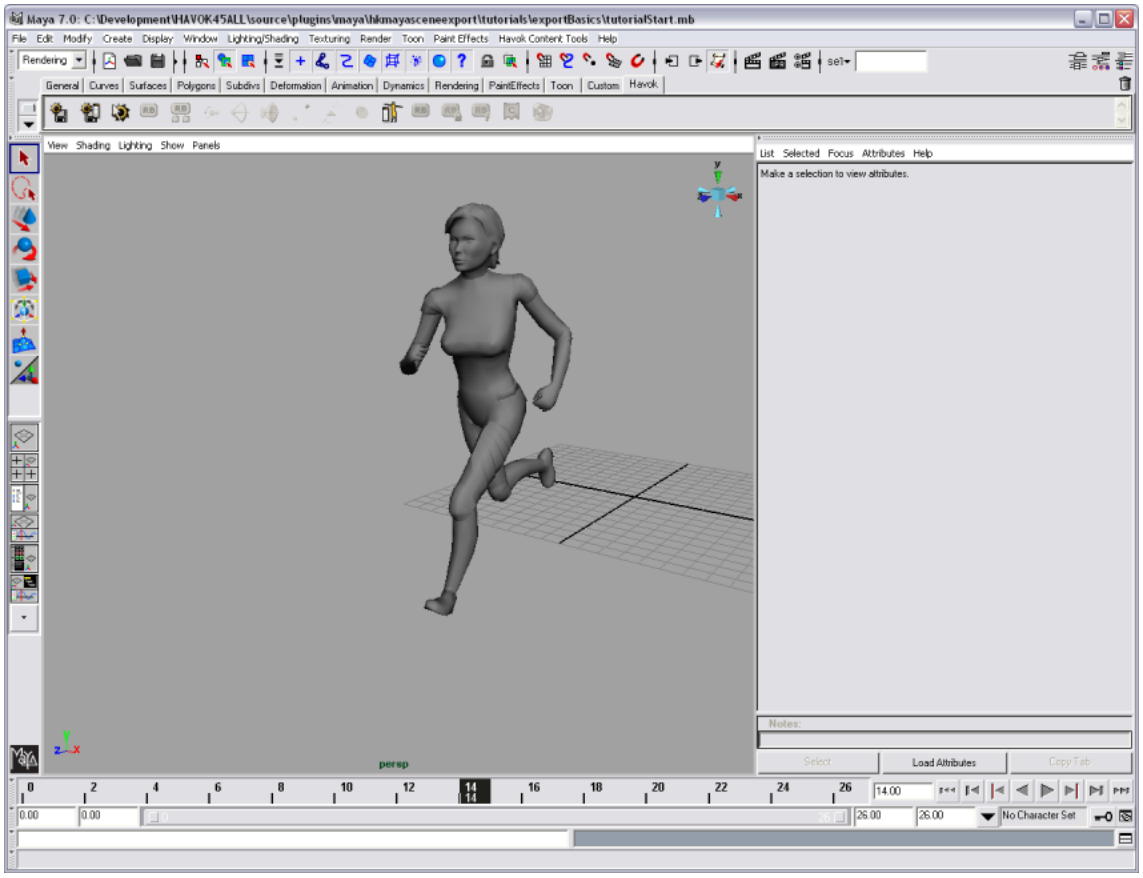
6.4.5.1 Getting Started

Ensure that you have installed and loaded the Havok Content Tools for your version of Maya. You should see a **Havok Content Tools** menu in the Maya menu bar:



If the menu is not present then refer to the section on loading the plugins.

Open the file "`tutorials/exportBasics/tutorialStart.mb`" (installed with the Maya Havok Content Tools, usually in "`Program Files/Havok/HavokMayaModules`"):



The scene contains the animation of a girl running (a single cycle). If you examine the scene, you will notice that the animation is constructed using a set of bones which modify a mesh through the use of Maya skinning algorithms.

In this tutorial we will export this information (bones, animation, skin weights, etc) from Maya into a format that the Havok Animation/Complete SDK can interpret.

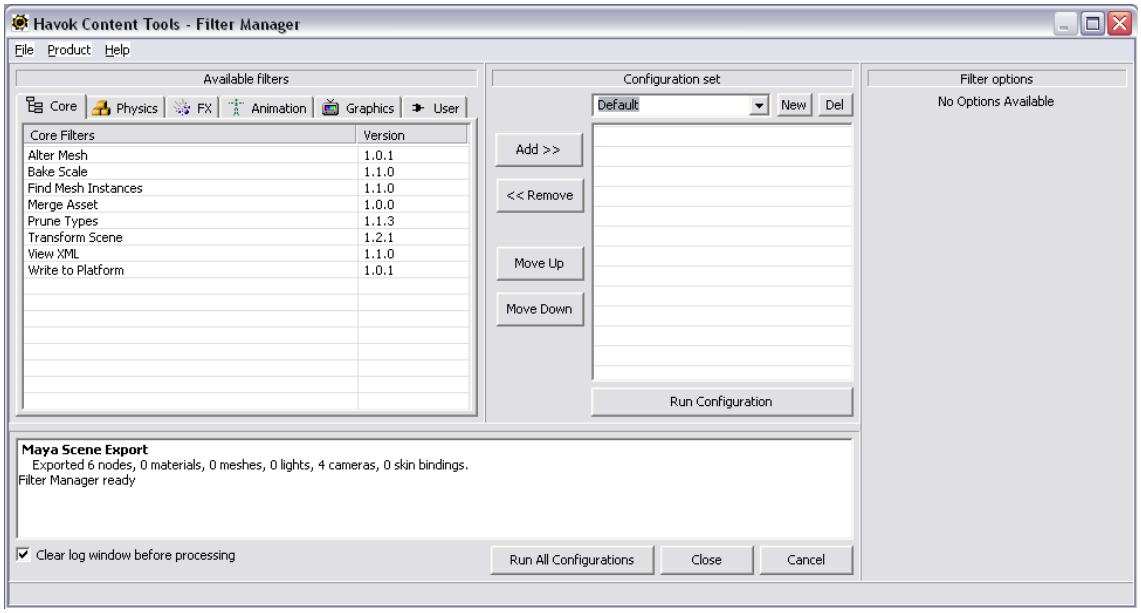
Note:

Even if you are not a user of Havok Animation or Havok Complete, you can still follow this tutorial as the Havok Content Tools packages contain filters for all products. This tutorial focuses on introducting the basic concepts behind asset export and processing so it is relevant for all Havok products.

6.4.5.2 Exporting the Scene

To export the current scene, simply click on the **Export** button in the **Havok Content Tools** menu or shelf .

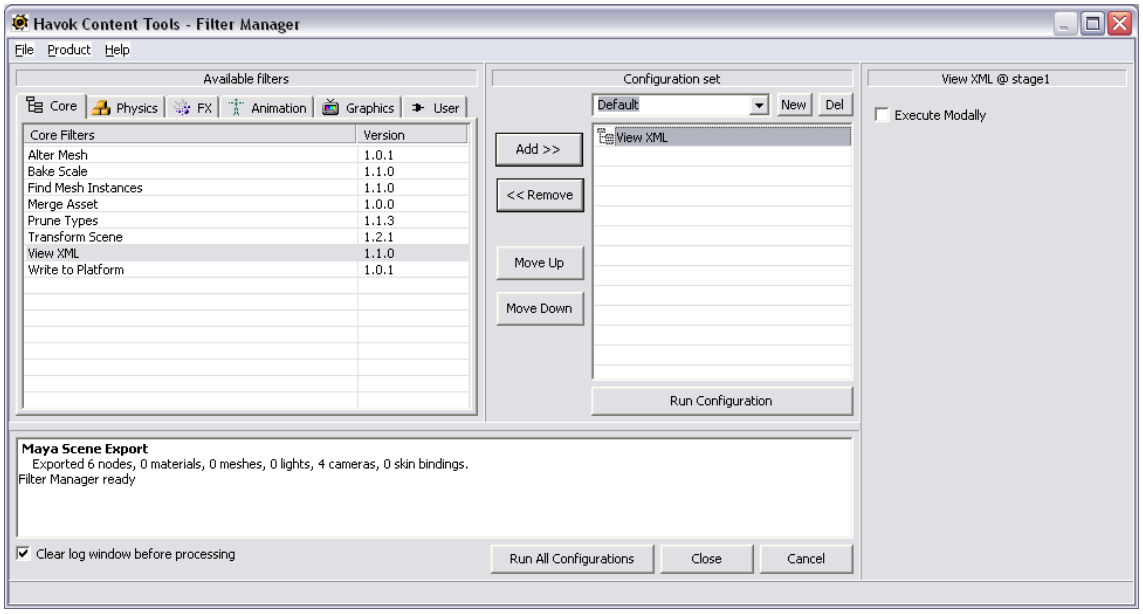
This invokes the Maya Scene Exporter - the scene is navigated and relevant information is pulled from the Maya nodes, attributes, meshes, etc. and converted into a modeler-independant format. After that, the content is passed to the filter manager - it's UI should appear after a short time:



The **Configuration Set** pane (in the middle of the window) lists the filters that will be applied to the content we just exported. It is now empty, which means that nothing will happen to the content. You can add filters by selecting them from the **Available Filters** tabs and clicking on the **Add** >> button.

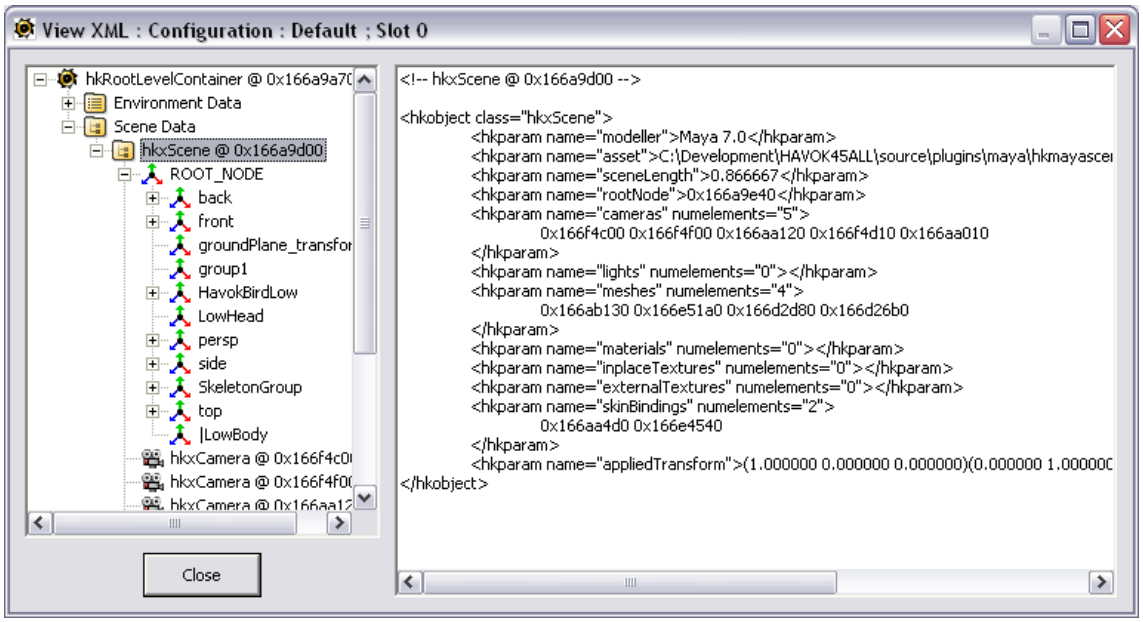
Let's start by adding a very useful filter for debugging purposes: the View XML filter.

- On the **Available Filters** window, click on the **Graphics** tab
- Double-click on the **ViewXml** filter (or select it and press **Add** > >)
- That filter should now appear at the top of the **Current Filters** list



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Filters are executed in the order they appear in the **Current Filters** list; if a filter modifies the content, that modified content is what is used by the next filter (check the filter pipeline documentation for details). The View Xml filter is special in that it does not modify any content - it just opens a window that presents the current content in a human-readable XML format. Let's check what we exported from Maya by clicking on the **Run Configuration** button:



Notice that filter manager is processing an "`hkxScene`" object (scene data), which contains information about nodes, meshes, materials and skin bindings. This is what the Maya Exporter extracted from our current scene, and is the raw information that we are going to process further until we have an appropriate set of objects to be used by our run-time.

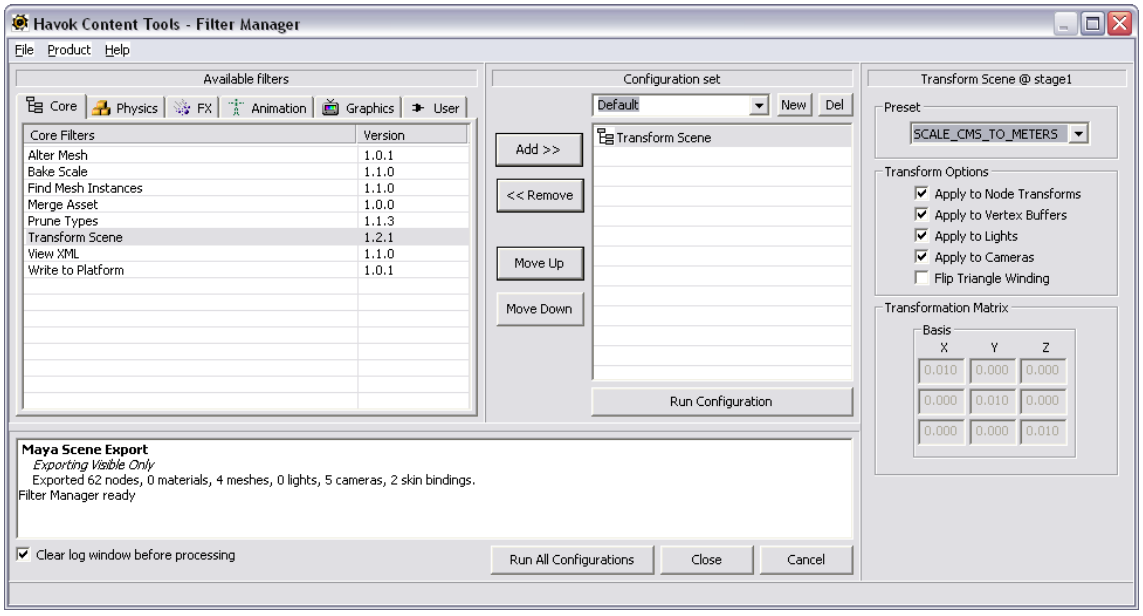
Close the XML window and return to the filter manager. Remove the View Xml filter by selecting it and clicking on << **Remove** or hitting **DEL** .

Tip:

Feel free to add the View Xml filter at different locations during this and other tutorials in order to examine how processing is done by different filters. This filter is a very useful debugging tool.

6.4.5.3 Processing the Scene

Let's now add some useful filters for processing the scene. The first filter that we'll add will be the Scene Transform filter (from the **Core** tab):

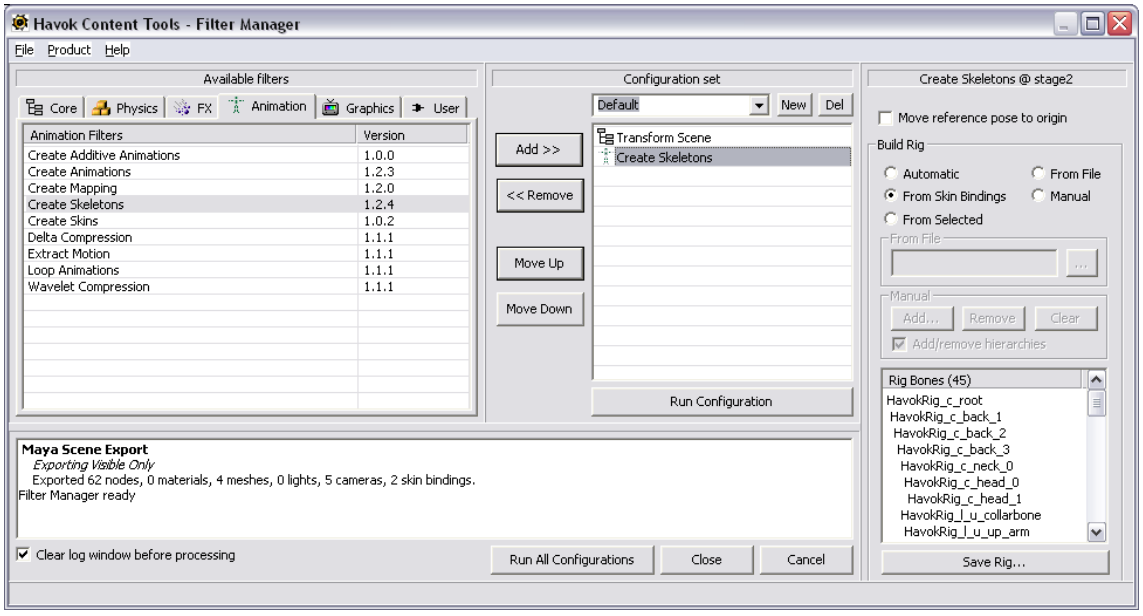


This filter is very useful and, in the majority of cases, it will be the first filter that you will need to add to any filter configuration. This filter applies an arbitrary transform to all the objects (nodes, meshes, etc) in the scene. In this case, we want to apply it here because the internal units in Maya are centimeters, whereas the Havok run-time units are meters. Hence we use the **SCALE_CMS_TO_METERS** preset in the options of the filter (the right panel in the filter manager).

The next filter we will apply is the Create Skeleton filter (from the **Animation** category). This takes a look at the nodes in the scene and creates an "`hkaSkeleton`" object (used by the Havok Animation/Complete SDK).

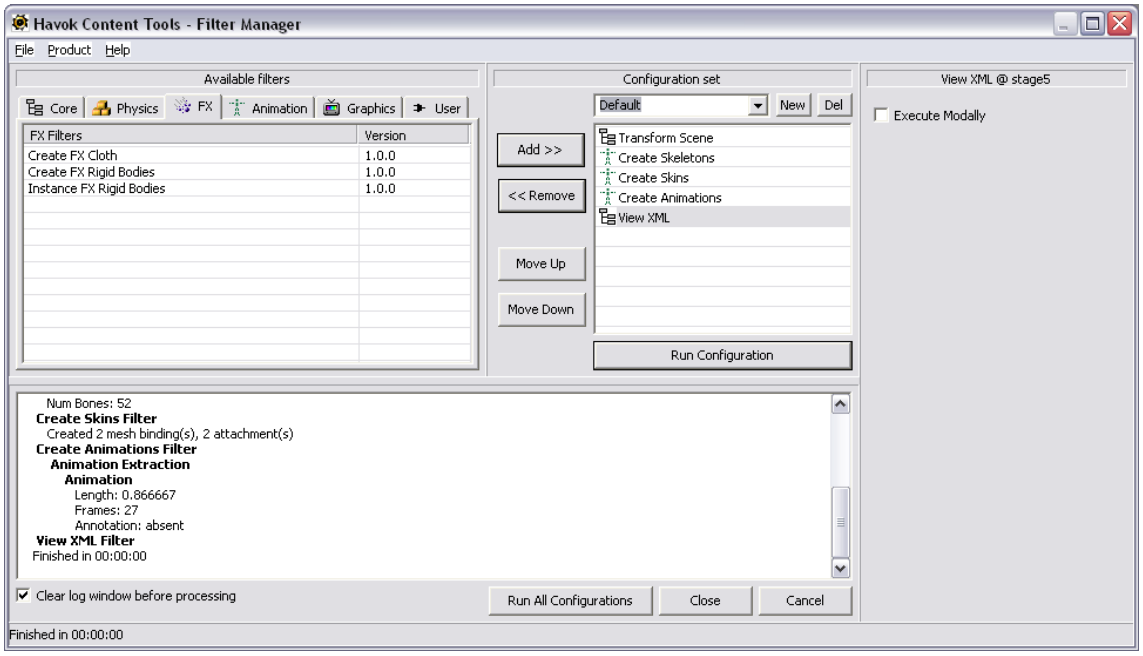
Note:

If your current Product Selection is set to Havok Physics, some of the filters will appear in red, and when run, warnings will be generated. You can either ignore those warnings, or change your product selection temporarily to Havok Complete.

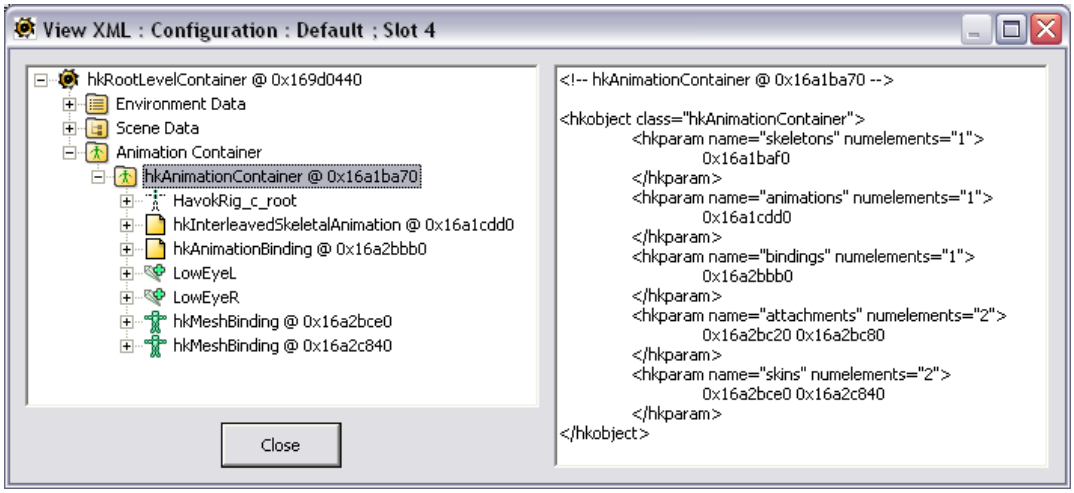


As you can see, this filter also has options associated with it (these appear in the right hand panel). In this case, switch from " **Automatic** " to " **From Skin Bindings** " - this will reduce the number of bones in the skeleton to just those associated with the skin. You can check the documentation for this and other filters if you want to learn more about how they work:

After this filter, add a Create Skin and a Create Animations filter (also in the **Animation** category) - using the default options for each. Finally, place the View Xml filter again at the bottom of the filter list:



When you run (press the **Run Configuration** button) the filter processing, the View XML filter will show you the effect of all the processing:



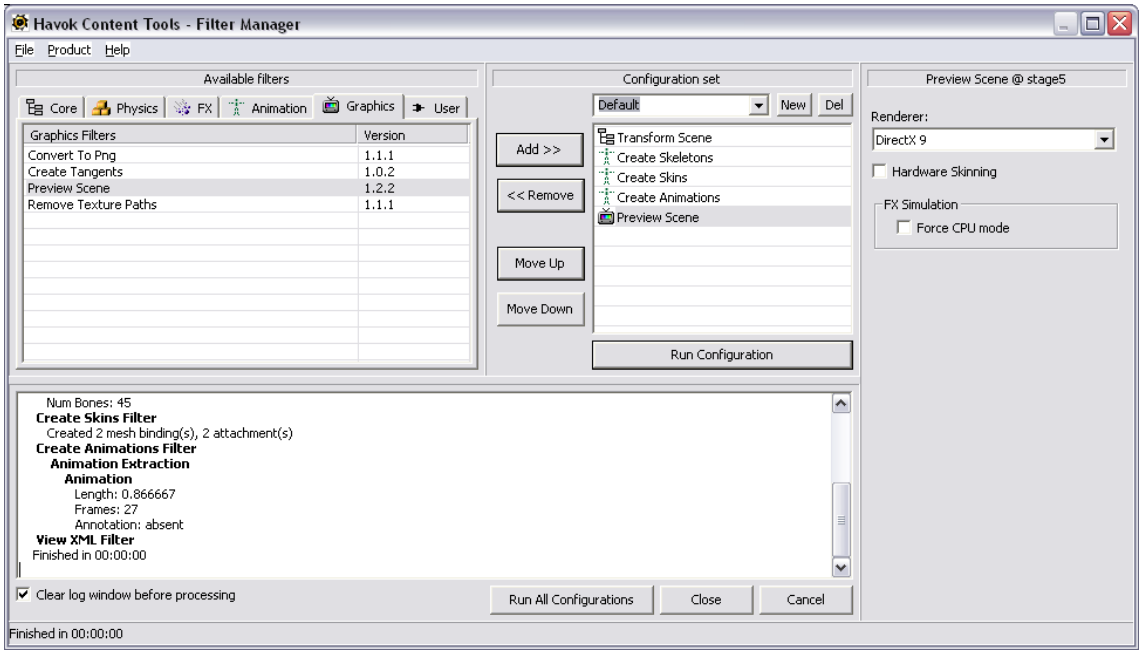
Notice how a new root-level object, an *Animation Container*, is now part of the content. This contains Havok Animation/Complete objects such as `hkaInterleavedSkeletalAnimaton`, `hkaSkeleton`, `hkaMeshBinding`, `hkaBoneAttachment`, etc.. that we should be able to use at run-time.

6.4.5.4 Previewing the Scene

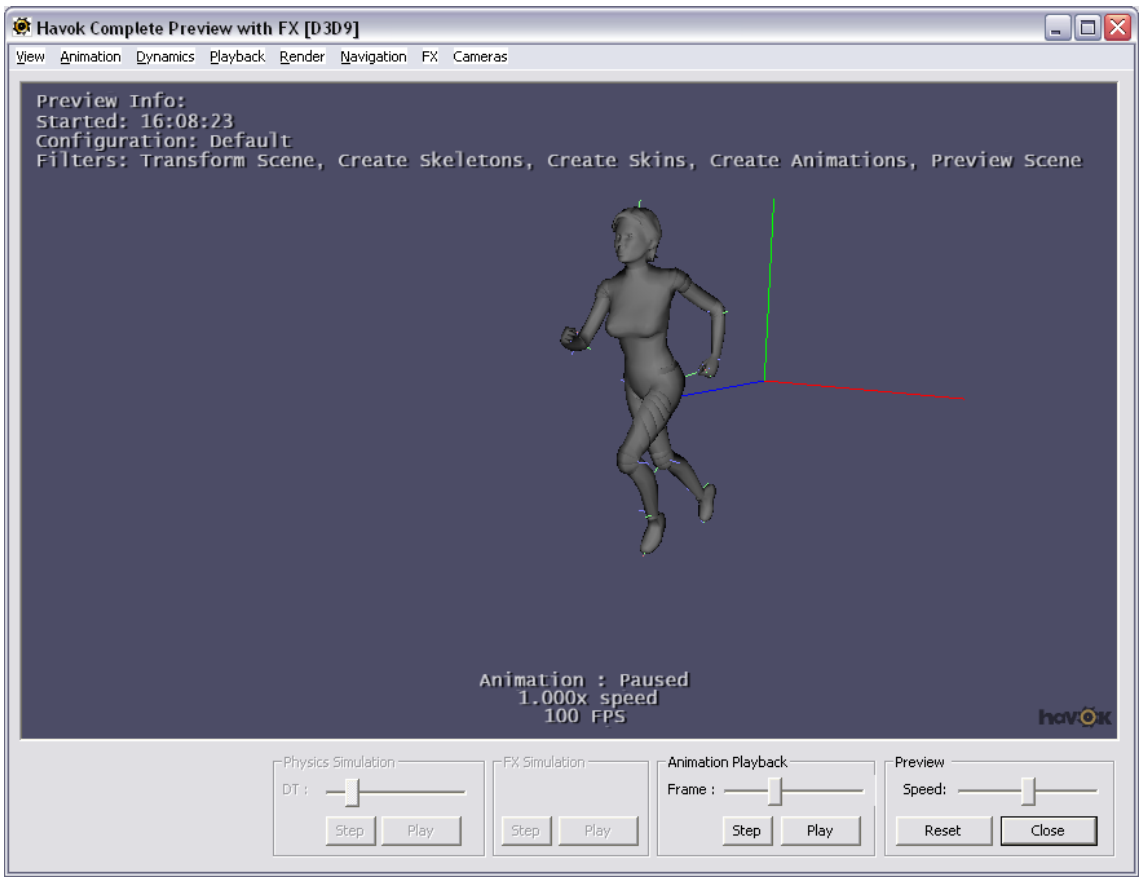
Although examining the contents by viewing the XML output give us an indication on how our processing went, in most cases visual inspection is the best way to ensure that the processed content matches our expectations.

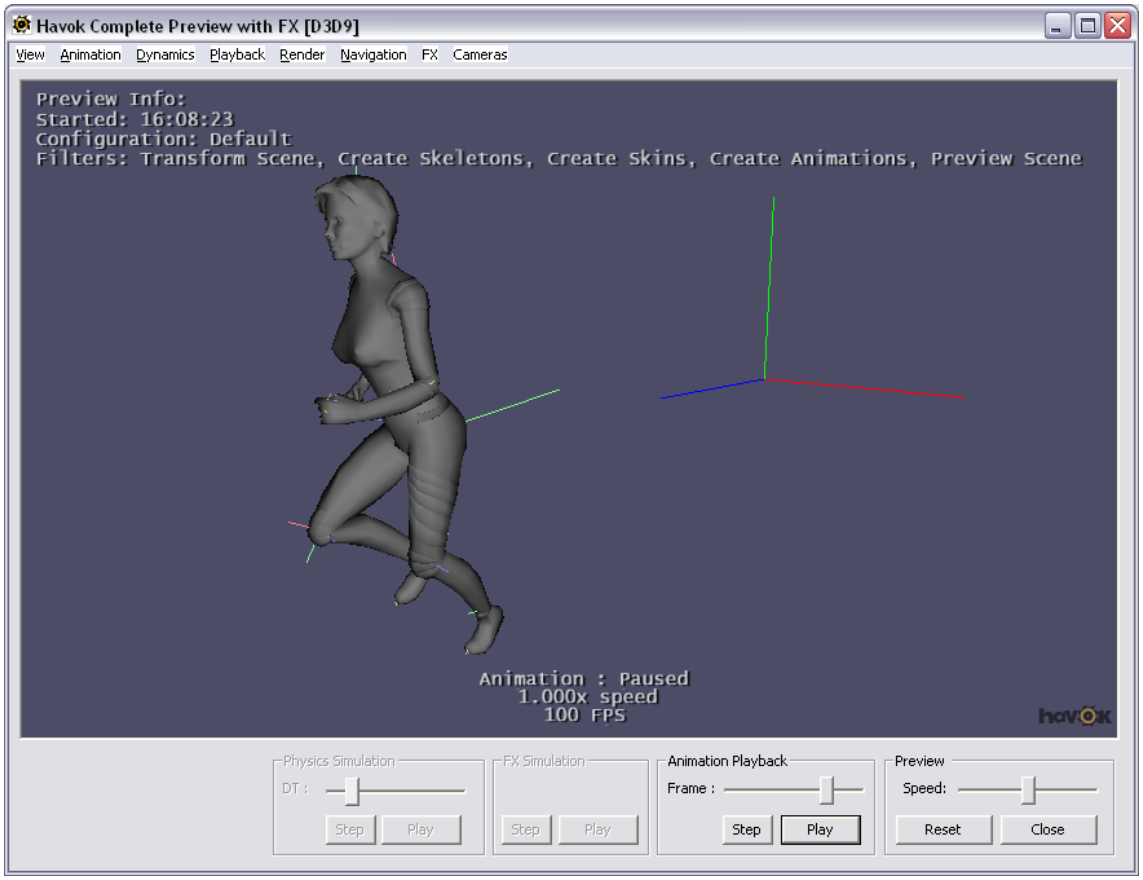
Havok provides the Preview Scene filter (in the **Graphics** category) just for this purpose - this filter is built using the Havok run-time, including some display libraries, and will play back any Havok content it finds.

Remove the View XML filter, add a Preview Scene filter at the end of the filter list, and click again on **Run Configuration** :



A modeless window will appear with our animation and skinning running in real-time:





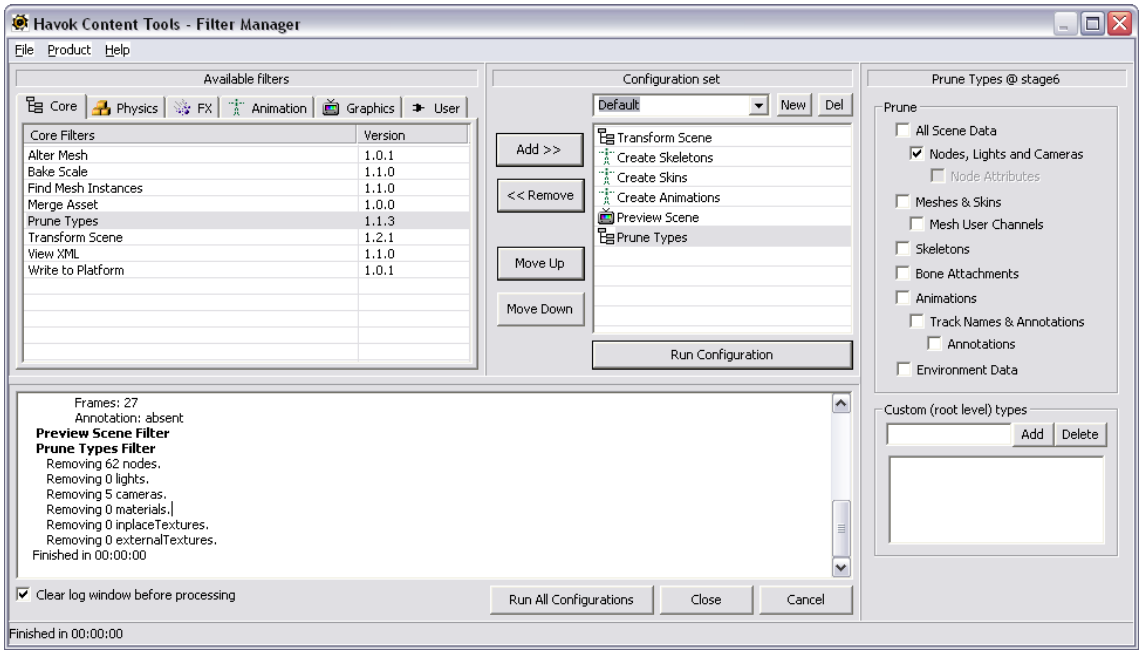
The preview window gives you a set of options to change the speed of playback, the current frame in the animation, etc.. Check the Preview Scene filter documentation for more details.

6.4.5.5 Pruning Some Data

Most of the filters either modify (eg. the Scene Transform filter) or add (eg. the Create Skeleton filter) information to the scene. Often it is useful to remove some of the data in an asset which is no longer relevant (particularly at the end of the processing). For example, once we have created our skeletons and animations, the original data regarding nodes and their keyframes is probably no longer needed (since it has been transformed into other classes).

We can add a Prune Types filter (in the **Core** category) after the Preview Scene filter. Leave the default options as they are: this will remove nodes (and their keyframes), lights and camera information:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



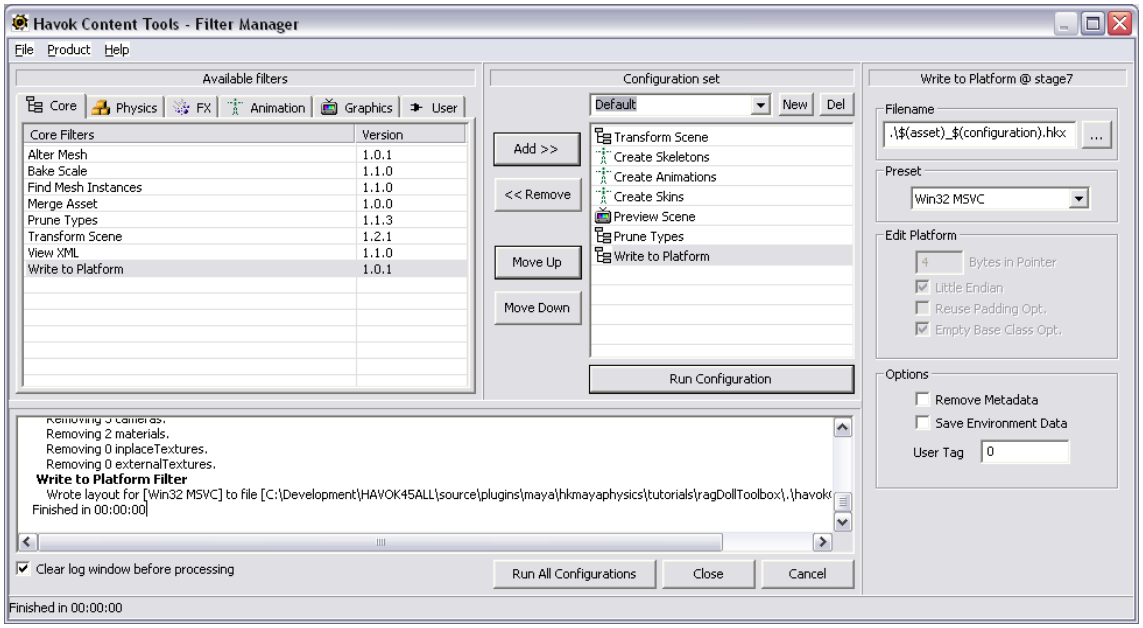
Note:

You could also add this filter before the Preview Scene filter - but since the Prune Types filter will remove camera information, the preview will have to use a default camera (rather than reflect the camera in the original scene).

6.4.5.6 Writing the Processed Contents

So far, all of the processing done by the filters has happened in memory, and has not been written to any file.

The Platform Writer filter (in the **Core** category) will serialize the processed content to a file. It can write either XML format or binary format for many different platforms. Add the Platform Writer filter at the end of the filter configuration - by default it will save a binary .hkx file for the Win32 platform in the same location as the asset. Process the asset once again (click on **Run Configuration**):



Observe in the log window that the processed asset is now saved alongside our Maya file, ready to be loaded into the Havok SDK.

This finalizes this first tutorial: you can find the final maya scene in the `"tutorialEnd.mb"` file.

We'd like to encourage you to experiment further with different filters and filter options. Check the results in the filter processing log, as well as the output of the View Xml filter, you should be able to get a good idea of what each filter does.

If you are a Havok Animation/Complete user, the following sections go a little further in processing this scene by doing some motion extraction. If you are a Havok Physics user, skip the following section and go to the Physics Basics Tutorial.

6.4.5.7 Appendix (Animation/Complete Only): Extracting Motion

The animation we just exported has some *motion* on it: in between the first and last frame of the animation there has been some displacement on the character (it has moved forward).

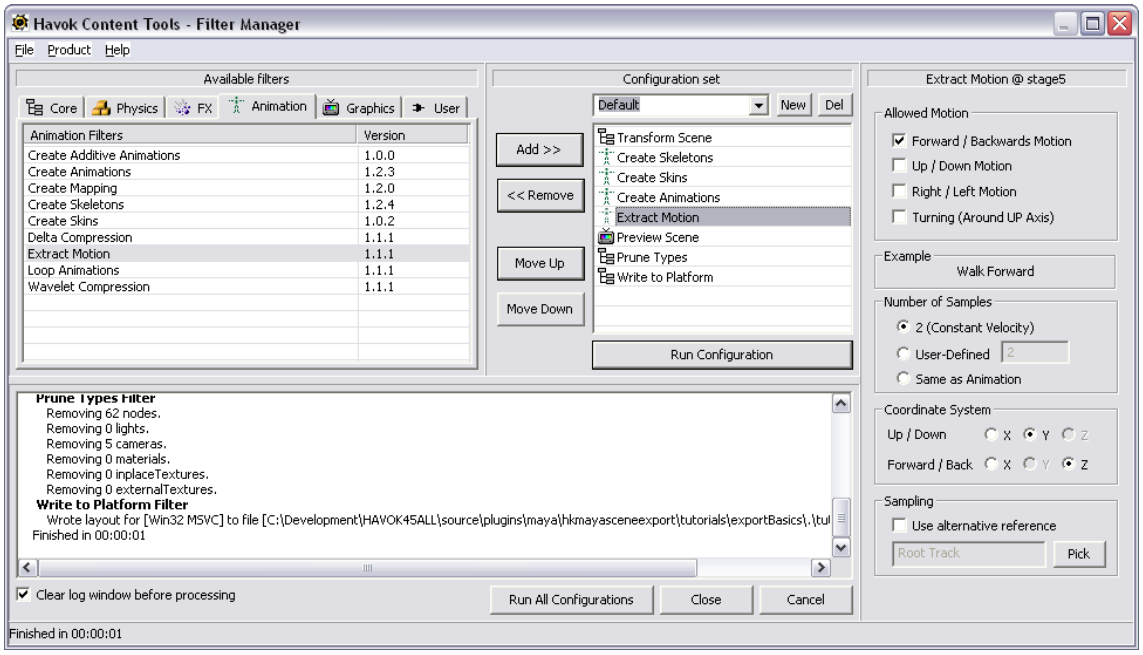
Motion extraction, explained in detail in the *Havok Animation* section of this manual, is the process on which we examine an animation and divide it into two: an motion/displacement component (the moving forward during running) and a local animation component (the movement of arms, legs and body during the run cycle).

So, after motion extraction our running animation should become a run-on-the-spot animation, complemented by some information on how the character should displace when that animation is applied (the extracted motion).

In this appendix to the tutorial we are going to do motion extraction to our run animation.

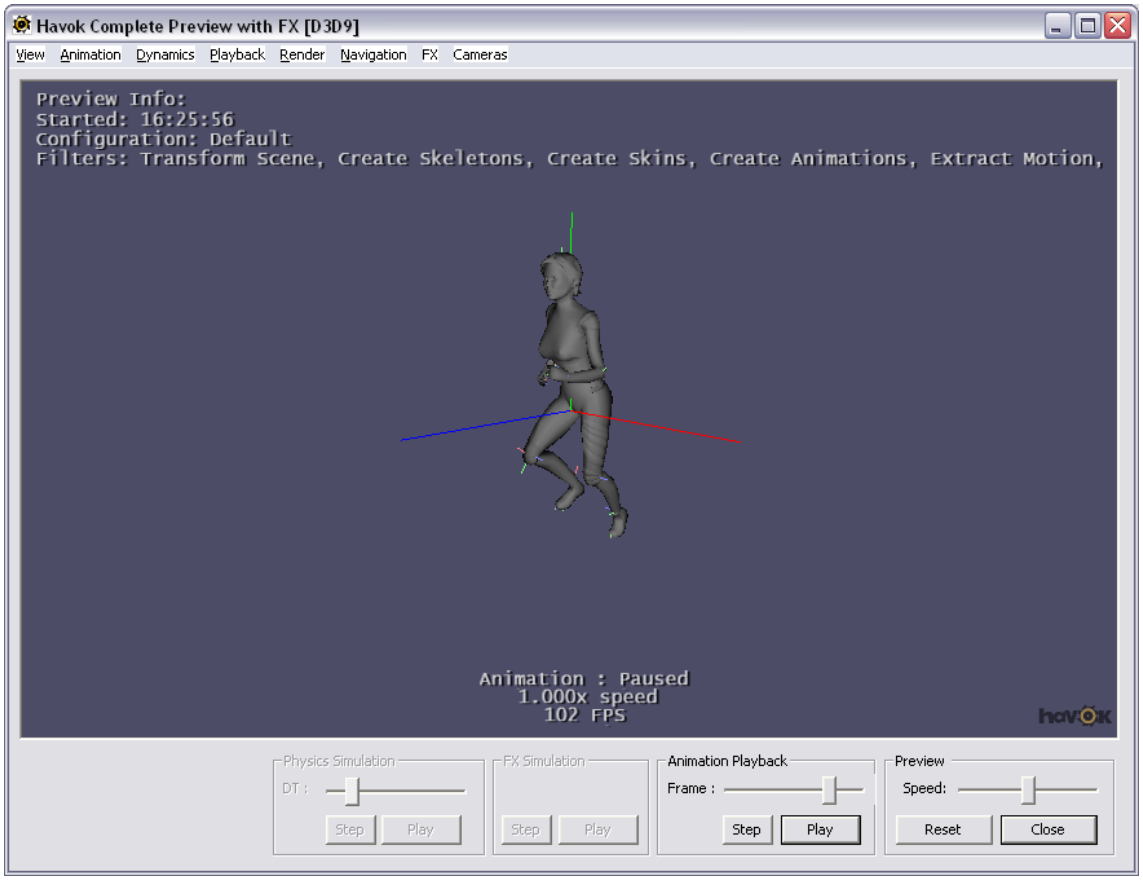
- Open the filter manager again (if necessary) by clicking on **Export**

- Add the Extract Motion filter (in the **Animation** category) to the filter list, just after the Create Animations filter (use the **Move Up** and **Move Down** buttons to position the filter in the right order):



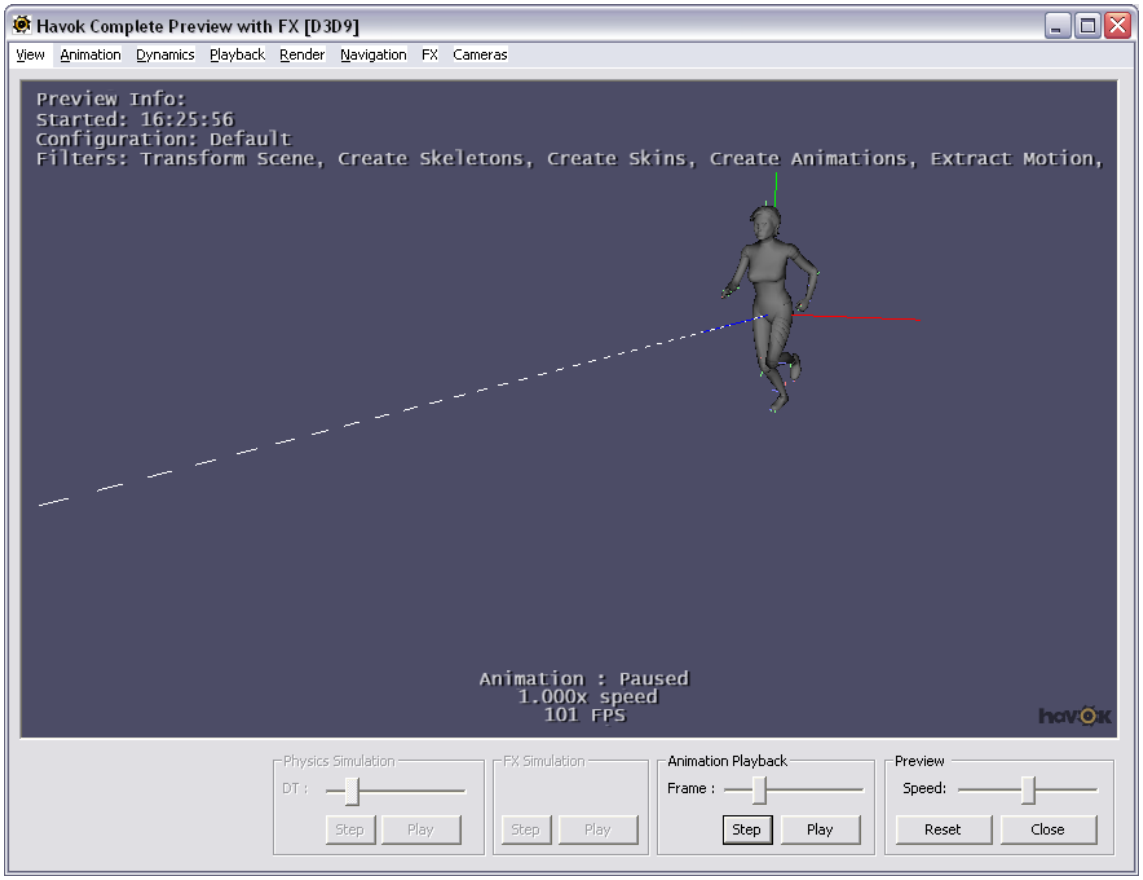
The filter options allow you to specify what kind of motion do you want to extract from the animation. In our case the motion is a forward motion so we'll leave the **Forward/Backwards** check box on. But, since in our animation the up direction is Y and forward direction is Z we need to specify that in the **Coordinate System** options (the defaults are Z, X).

If we now run our filters, it will open the Preview dialog: you can see how now our animation is now a run-on-the-spot animation:

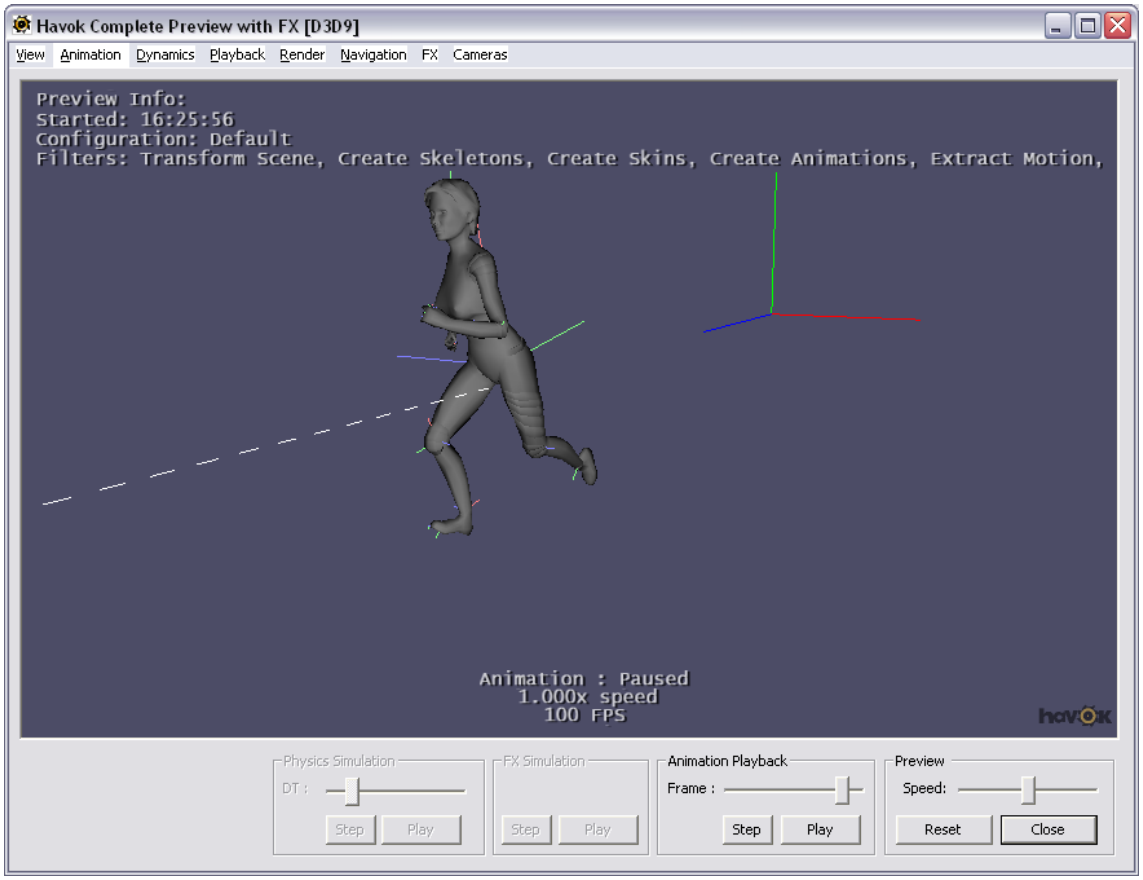


The Preview Scene filter knows about motion and it allows you to visualize it. In the preview dialog, click on **View** > **Extracted Motion** :

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



A dashed line is display showing the motion extracted from the animation. The preview can also apply and accumulate the extracted motion - on the **Animation** menu, click on **Accumulate Motion** :



Notice how the character now moves according to the motion that was extracted - notice also how that motion accumulates : the character moves forward continuously over animation loops (instead of warping back to the starting position at the beginning of each loop).

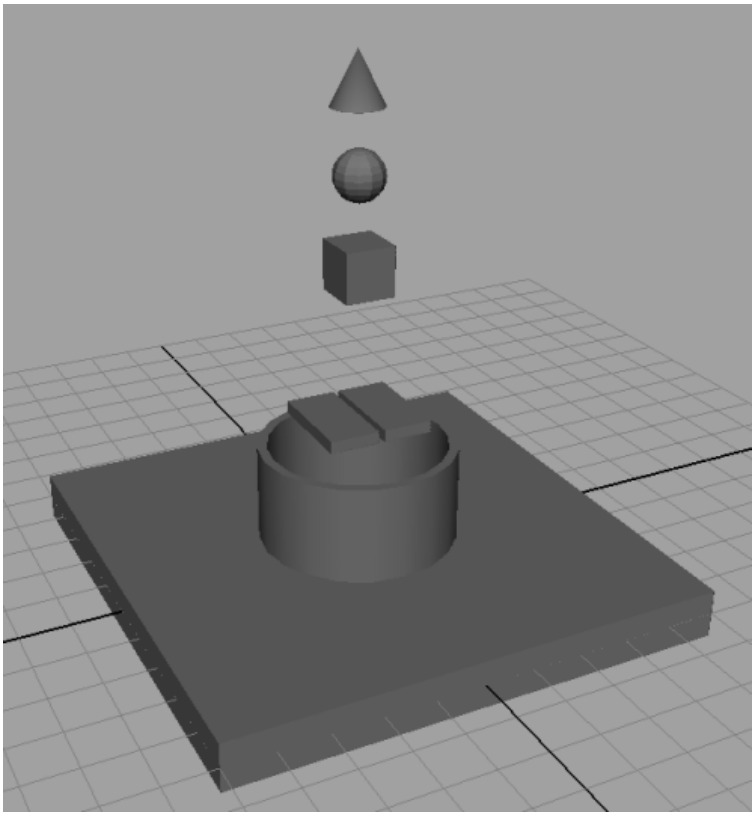
You can find the tutorial scene with motion extraction applied in the file `"tutorialEnd_MotionExtracted.mb"`.

6.4.6 Tutorial: Physics Basics

In this tutorial we are going to explore the basic operations for setting up rigid bodies and constraints in Maya, and processing them through the filter pipeline. This tutorial assumes that you are familliar with the basics of exporting and processing scenes, covered by the Export And Animation Tutorial. We also recommend that you take a look at the rigid body and constraint concepts sections of this documentation.

6.4.6.1 Getting Started

Start by loading the scene "`tutorialStart.mb`", located in the "`tutorials/physicsBasics`" folder of your Maya module ("`Program Files/Havok/HavokMayaModules/...`" by default):

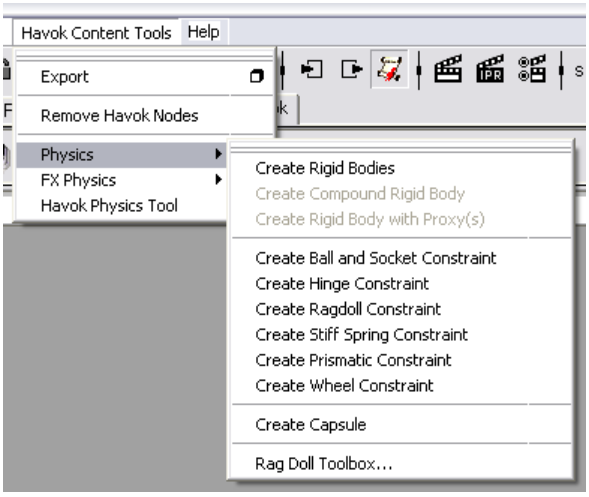


In this scene, we have a set of objects: a cone a sphere, a box, a trap door, a can and a ground. We want to attach rigid body properties to them so they behave as they would in real life.

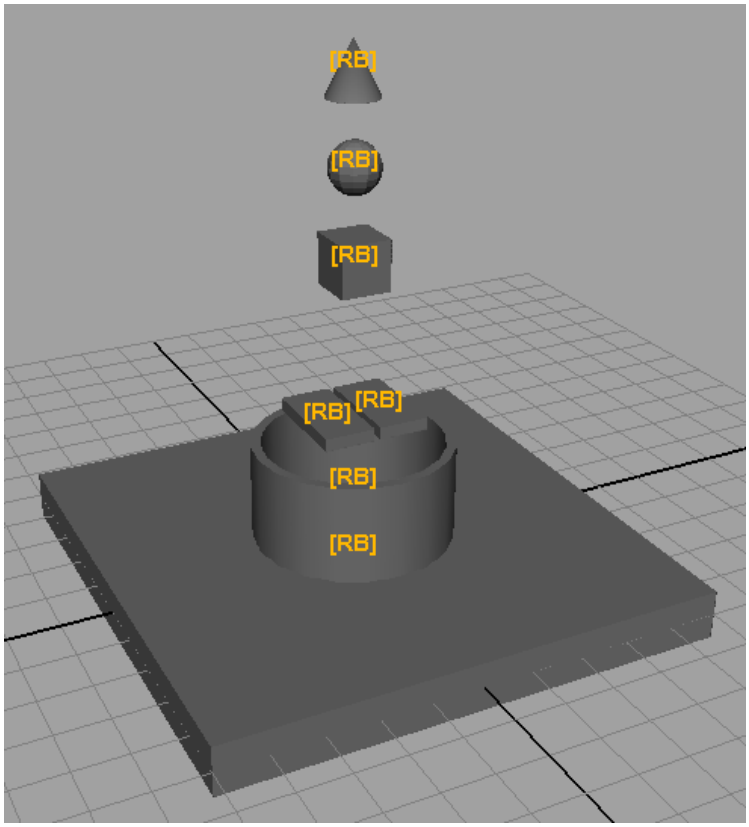
6.4.6.2 Setting Up Rigid Bodies

To start with, we need to make these objects become rigid bodies, i.e., attach rigid body information to them so we can later on create rigid bodies and simulate them using the Havok SDK.

The Havok Content Tools provide a set of physics tools in Maya in order to do so easily. These tools are available through the **Havok Content Tools** menu and toolbar:



In our case, we want to create a rigid body out of every single object, so let's start by doing so. Select all the objects in the scene, and click on the **Create Rigid Body** button or menu option. Notice how all the objects in the scene are now are labeled with the text " **[RB]** ", indicating that these objects have now rigid body properties (the square brackets indicate that they are fixed bodies):



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Note also that the Havok Physics Tool has been actived. This enables the mode in which Havok properties are represented in the viewport. The tool is customisable - you can access the tool settings at any time by double-clicking its icon in Maya's toolbox.

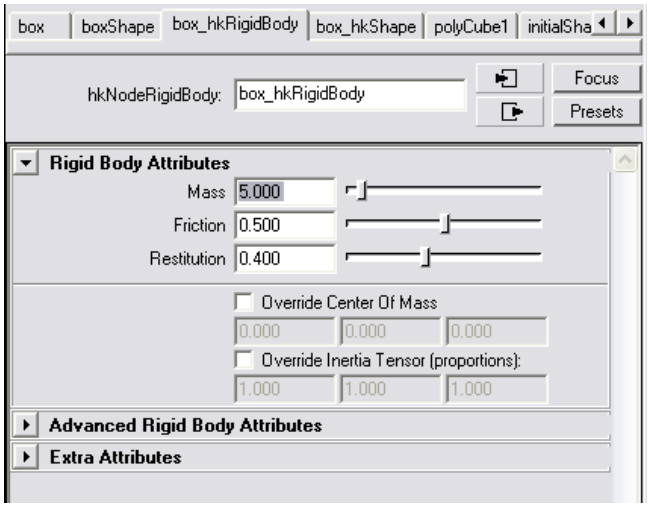
In Maya, rigid body properties are stored in nodes. For example, if you select the suspended box and then open Maya's outliner panel, you should see that two nodes have been connected to the box:

The rigid body node contains information about the dynamic behaviour of the object: mass, friction and restitution are specified here. The shape node contains information about the representation of the object for collision detection.

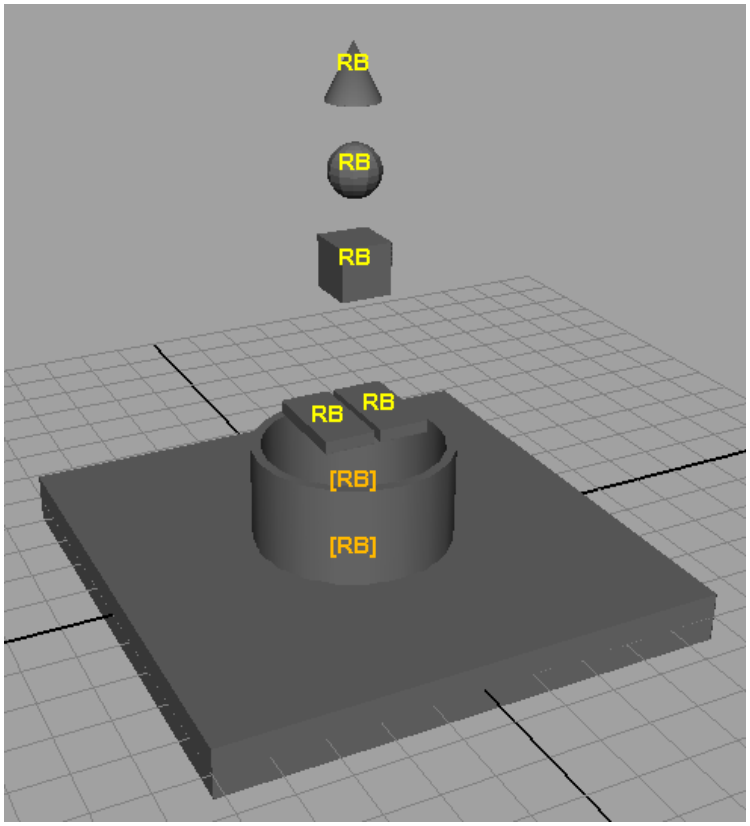
We will be exploring these modifiers a little bit more later on. For now, select the rigid body node and open the attribute editor. Notice how the mass of the box is currently zero - that is the default mass for new rigid bodies. Since Havok will consider a mass of zero to be an indication of the rigid body being fixed in space (not moving), we want to make sure that we specify a mass for all the dynamic objects in our scene - that would be the cone, the sphere, the box and the trap doors.

Set the **Mass** attribute of each dynamic to a value different than zero (5kg for example). Do this by opening the attribute editor and either: selecting each rigid body node directly in turn; selecting each object and locating the rigid body node tab:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



After doing this, we have a scene set up with 7 rigid bodies, 5 of them dynamic/movable and 2 of them (the ground and the can) fixed. We can easily identify which are fixed and which are not:



Note:
The 'RB' labels, like all other Havok content, is only visible while the Havok Physics Tool is active. Remember to activate this tool whenever you wish to view or edit any Havok content in the scene.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

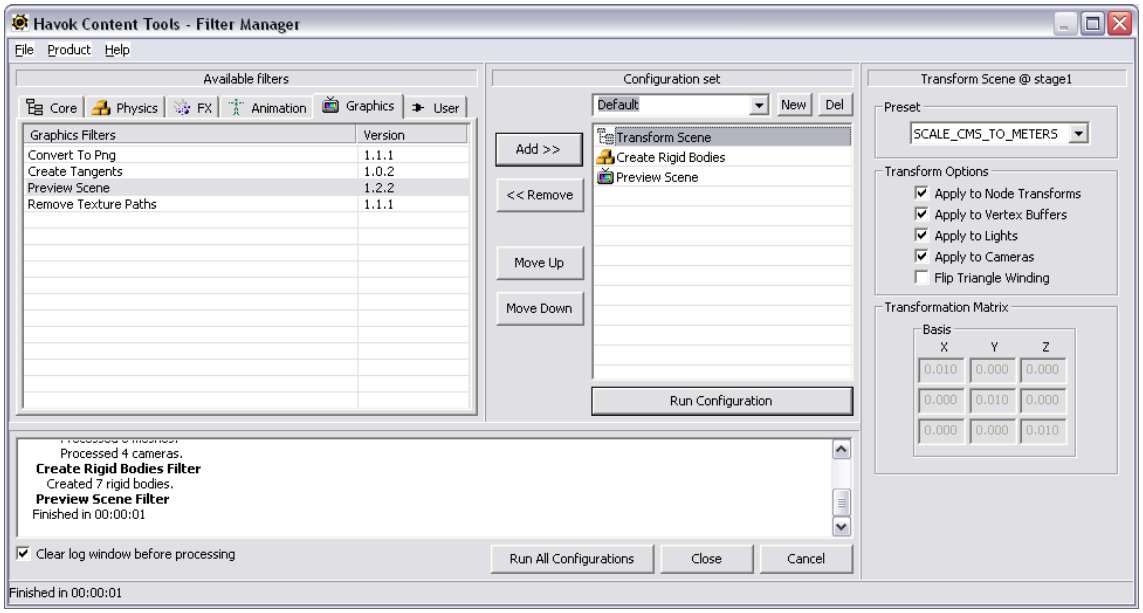
6.4.6.3 Previewing the Scene

Now that we have some content set up, we'd like to preview it and check that everything behaves as expected. In the previous tutorial we saw how to export, process and preview assets using the Maya scene exporter and the filter pipeline. We are now going to do the same, but this time using some processing specific to physics assets.

Start the export and processing by clicking on the **Export** button or menu option. This should open the filter manager, with an empty filter setup.

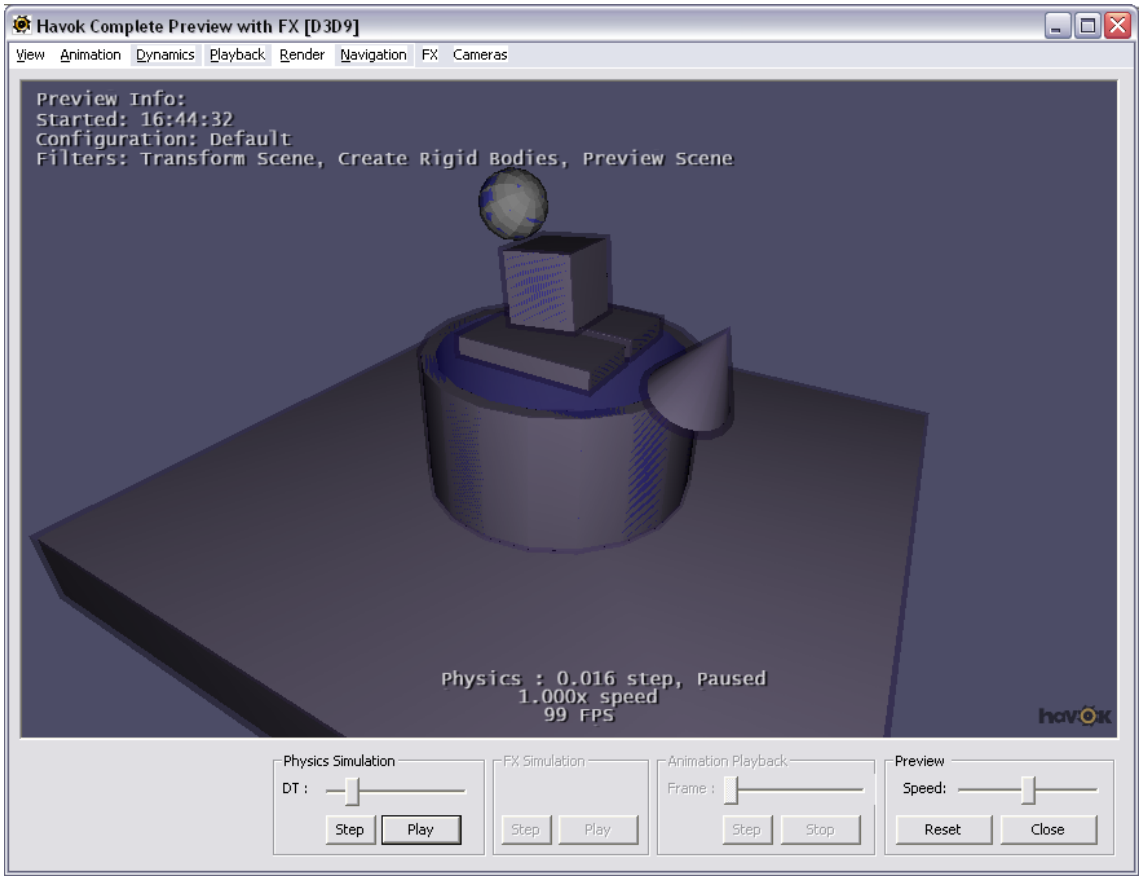
Insert, in this order the following filters:

1. Scene Transform (**Core** category): Since Maya works internally with centimeters and we work in meters, set the **Preset** parameter to **SCALE_CMS_TO_METERS**
2. Create Rigid Bodies (**Physics** category): This filter will detect the rigid body information we assigned to our objects and will create rigid bodies based on it. Use the default options.
3. Preview Scene (**Graphics** category): We want to visually inspect the behaviour of our objects, so we will use this filter to render and simulate them.



If we now process our asset (press the **Run Configuration** button), a preview window will appear with our scene:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



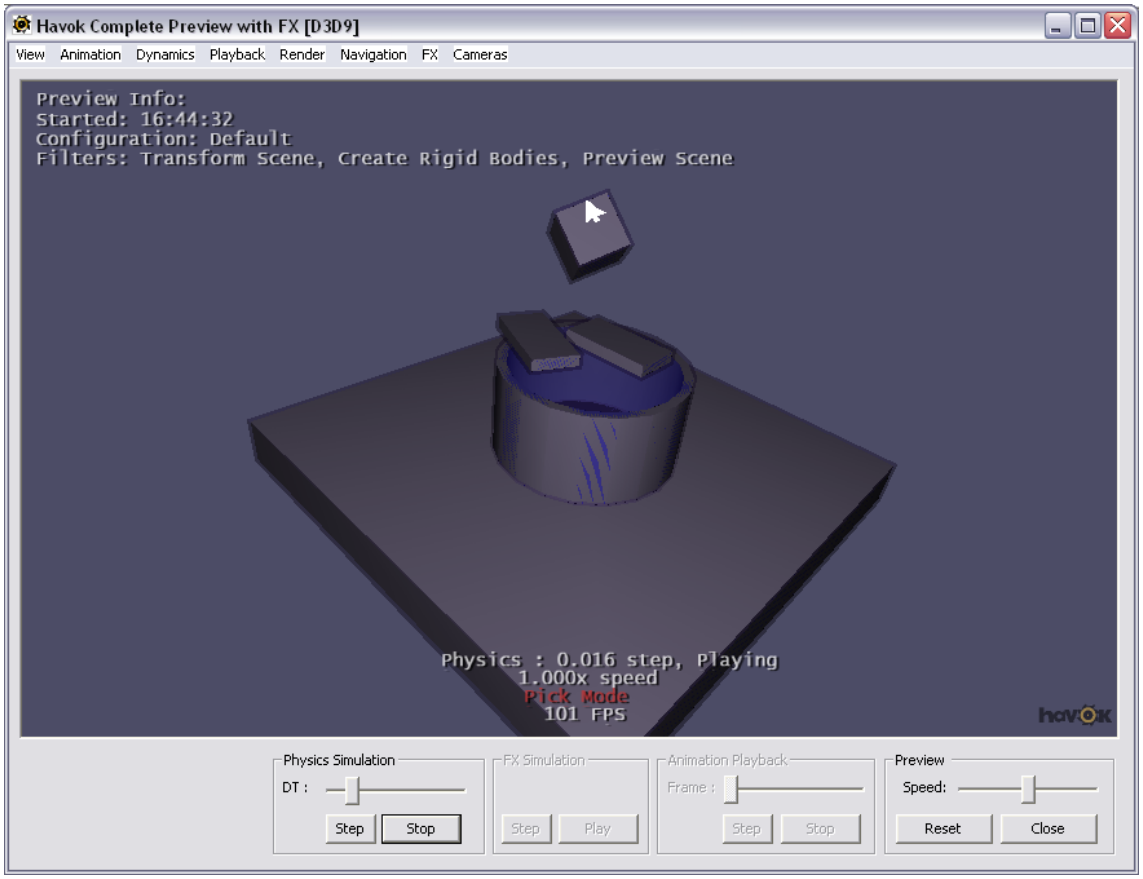
Press the **Play/Stop** button to start the simulation.

You will notice that all the objects fall as expected, but there is couple of things which are not quite right:

- None of the objects fall inside the can - it is like there is an invisible lid on it. If you switch on the menu option **View** > **Physics Ghosts** you should see a ghost representation of the rigid bodies displayed in the preview (in alpha blended blue). That shows that there is indeed a lid on the can. It looks like the can is being simulated as a closed volume!
- The trap doors do not behave as such, and are just falling as any other object. This is because we have not set up any constraint to specify that they should behave as hinged doors. We will take a look at this later on this tutorial.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

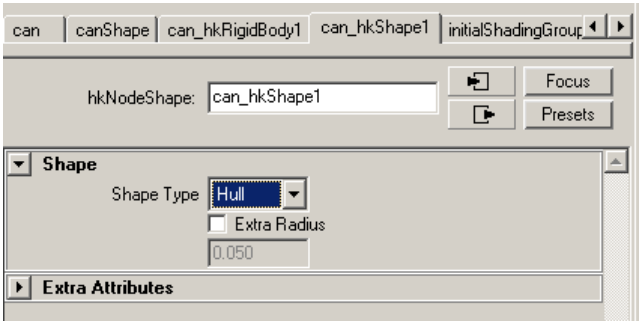
Mouse Picking
You can interact with the objects in the scene by using *mouse picking*: with the mouse cursor over one of the movable objects, press the **SPACE** bar and, without releasing it, move the cursor around - the rigid body will behave as if a spring was attached between itself and the mouse pointer. Release the **SPACE** bar to release the object.



When you are ready, close the preview and the filter manager and return to our Maya scene.

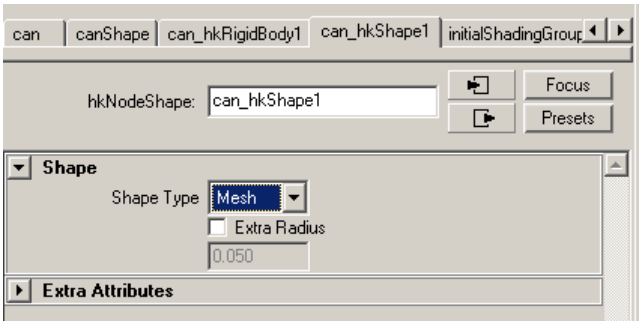
6.4.6.4 Shape Properties

Let's take a look at the can. Since our problems are related to collision detection, we'll take a look at the shape information associated with it. Select the shape node attached to the can object - by selecting the can in the viewport and finding the appropriate attribute editor tab:



Notice how the Shape Type parameter is set to "Convex Hull". When creating a rigid body from a mesh, the default value for this Shape Type parameter is taken by examining the object type in Maya. Thus, the default Shape Type for boxes is (Bounding) Box, the default Shape Type for spheres is (Bounding) Sphere, etc. For other arbitrary objects the default Shape Type is Convex Hull[4]. The Convex Hull of an object is the tightest convex volume that includes all the vertices of the object. The best way to visualize a convex hull is by imagining shrink-wrapping the object. In our case, using the convex hull of the can results on simulating a closed cylinder, rather than an open one, leading to the behaviour we observed in the preview.

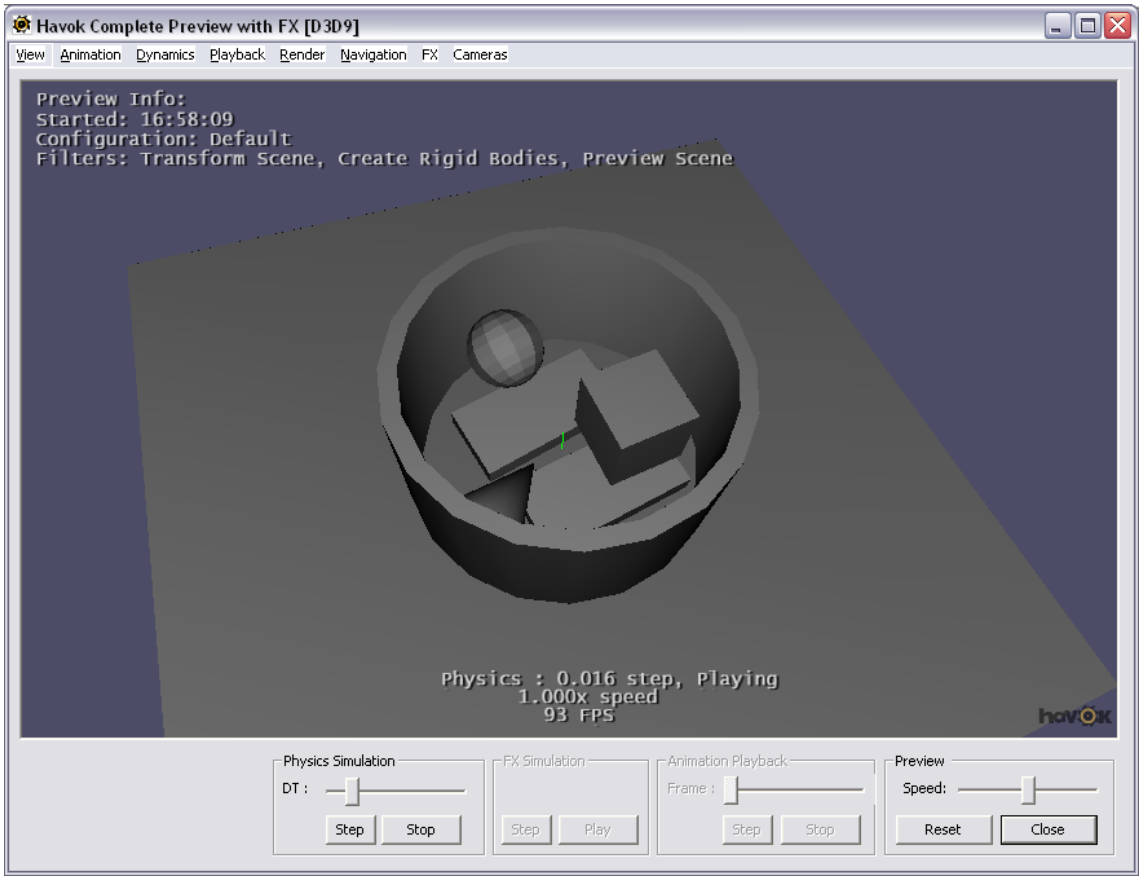
In our case we want to use the exact mesh of the object, so we need to change the Shape Type parameter to **Mesh** :



If you now **Export** the scene again and process the asset (press **Run Configuration**), you will see that the objects now properly fall inside the can during the simulation:

[4] The reason why the default Shape Type is Convex Hull and not Mesh is that convex objects are much faster to simulate than concave meshes and, in many cases, simulating a mesh as its convex hull gives very good results. Also, movable (not fixed) concave objects (objects with Shape Type set as Mesh) are not really suitable for interactive simulations (a warning will be generated if used).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Close the preview and the filter manager and return to our Maya scene.

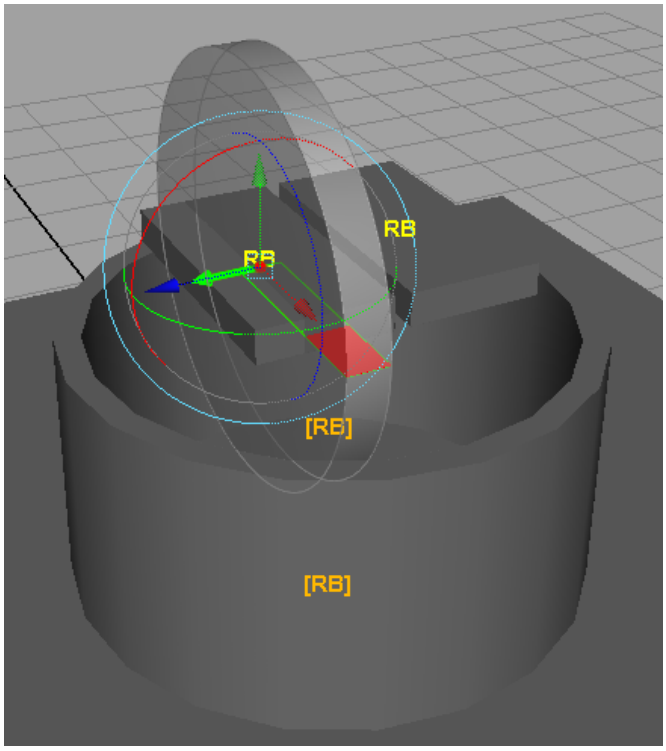
6.4.6.5 Setting Up Constraints

The last thing we want to improve in our scene is the behaviour of the trap doors: currently they behave as the other objects - they just fall. We want them to behave as if they were constrained to their location by hinges.

The Havok Content Tools for Maya give you the ability to set up and visualize constraints in Maya. In our case, we want to create a hinge constraint - a constraint that will limit the movement of the trap doors to a rotation around a single axis.

Select one of the trap doors and click on the **Create Hinge Constraint** button or menu option:

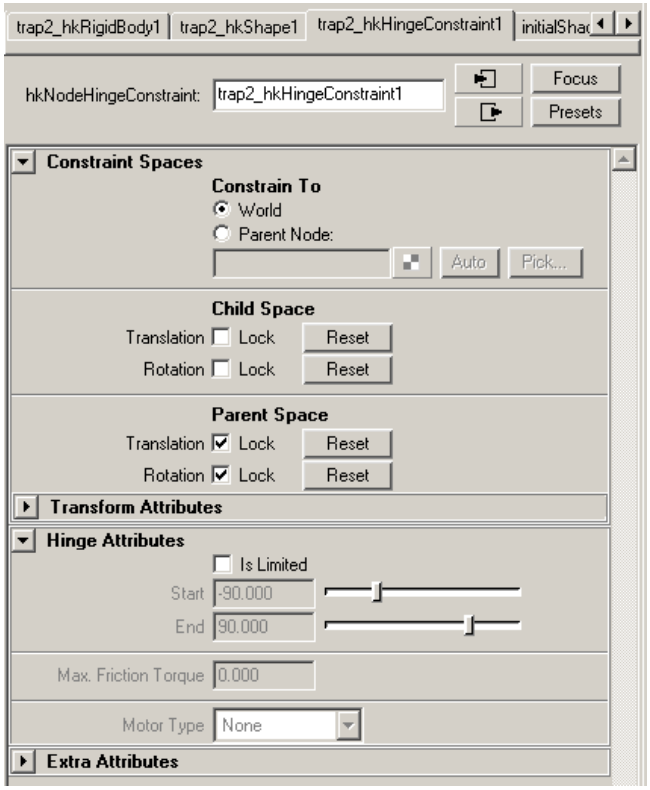
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



This adds a **Hinge** constraint node to the object, and activates the Havok Physics Tool so that we can visualize and manipulate it in the viewport. Note that the display size of the constraint is matched to the current size of Maya manipulators (which can be changed using the +/- keys).

Locate the attributes of the hinge constraint in the attribute editor for the trapdoor object:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



First of all, notice how the **Constrain To** parameter is set to **World** : this means that the constraint will only affect our rigid body, and its position, orientation and limits will be absolute (rather than relative to another (parent) rigid body). This is what we want.

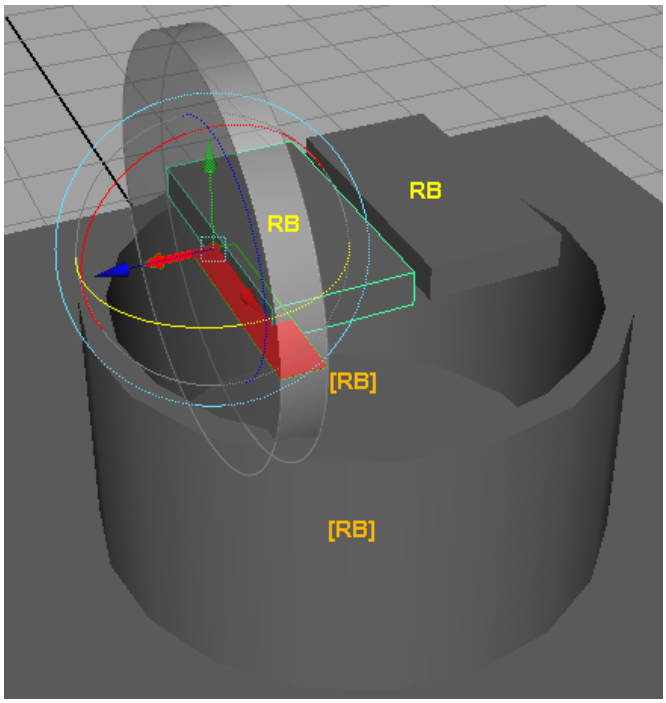
The constraint is represented in the viewport as an axis and a volume of allowed rotation (with another axis specifying where zero-rotation is). By default these axes are aligned to the object's Z and X axis, and is placed at the pivot point. In our case, however, we'd like to place them on the sides and orientate them so the trap door would move up and down rather than sideways. We can change the position of the constraint in two ways: by modifying the object's pivot or by manipulating the constraint spaces.

Positioning the constraint by modifying the rotate pivot

With the trap door selected, select Maya's translate tool and press the ' **Insert** ' key - this will put the tool into pivot mode. Then, drag the pivot of the trap door to the side which we want to act as the fixed point of the the door.

Press ' **Insert** ' again to put the tool back into normal mode, and active the Havok Physics Tool to visualize the constraint:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



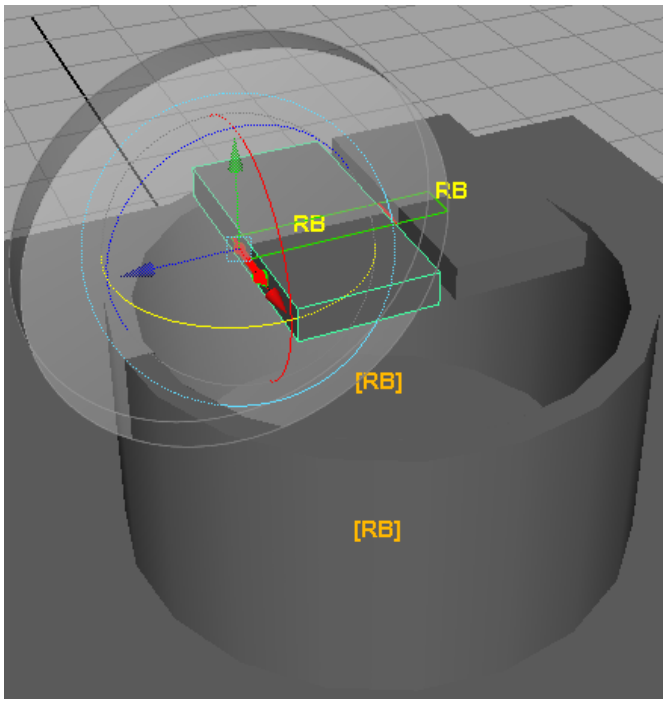
Observe that the constraint's position follows the the (rotate) pivot point of the object. However, we cannot change the orientation of the hinge by moving the pivot point - we can only do this by manipulating the constraint spaces.

Note:
An option is provided by the Havok Content Tools to 'include rotate pivot in transforms'. If this option is disabled then moving the rotate pivot of an object will have no effect whatsoever on the constraint. See the Export Options for more.

Positioning/orientating the constraint by manipulating it's constraint spaces

Let's now set up the orientation of the hinge constraint for the trap door. Activate the Havok Physics Tool if is not already active, so that we can manipulate the constraint spaces. This constraint provides manipulators in the viewport to change it's orientation. Drag the manipulators in the viewport to orient the hinge so that the main axis points along the length of the trap door and the zero-rotation axis (the one within the volume) points towards the other trap door:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Note that you can also change the position of the constraint by dragging the tripod manipulator. However this is not necessary in this case since the constraint is aligned to the pivot point of an object, which we have moved already.

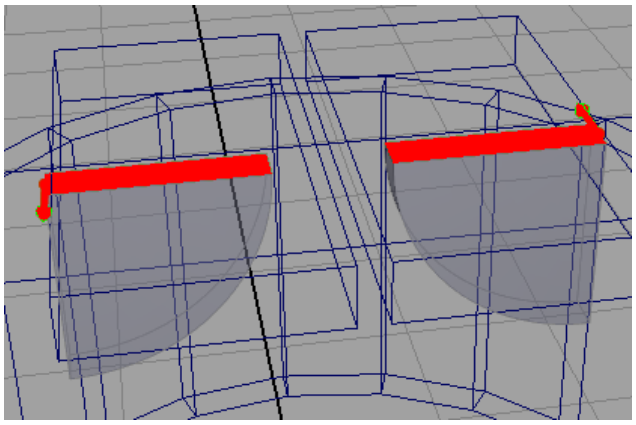
Repeat the process with the other trapdoor to create a hinge constraint with the appropriate position and rotation. In this case, try positioning the constraint by manipulating the constraint space only, instead of moving the pivot point.

Limiting the hinges

By default, hinge constraints allow full freedom of rotation around the main axis. It is often desirable to limit that amount of rotation to a specific range.

Select one of the trap doors, and locate its hinge constraint attributes in the attribute editor. Enable the "*Is Limited*" checkbox. Notice how the **Min** and **Max** parameters are now enabled, and how the limits are reflected in the hinge volume in the viewport (the limits are specified counterclockwise from the zero-rotation axis). Set the limits on each of the hinges to appear as follows:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



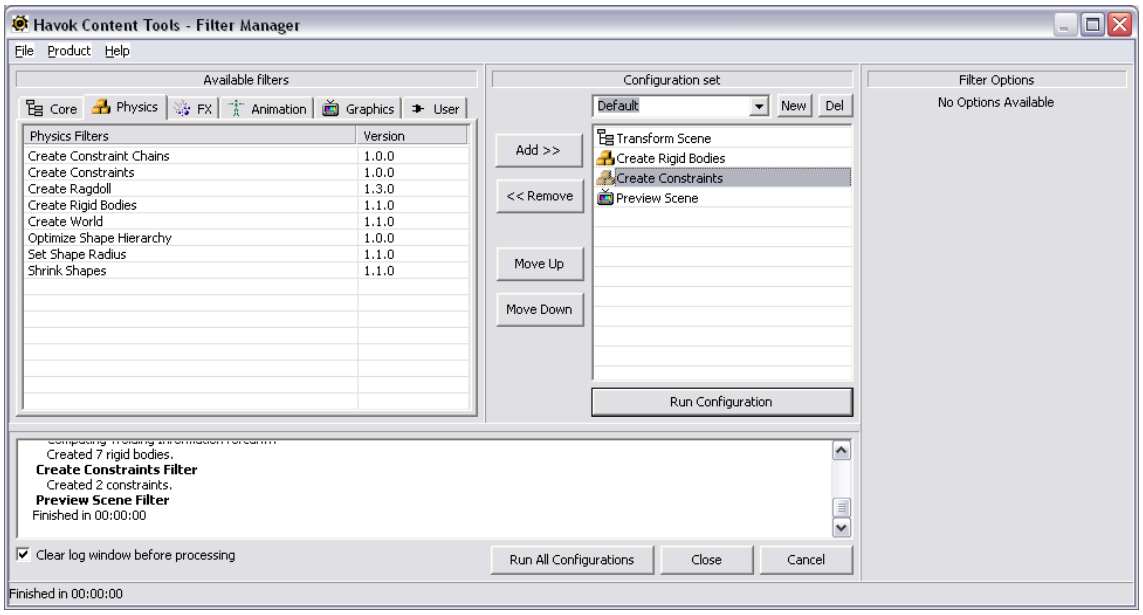
(Note that the Havok Physics Tool is set to **Show All Constraints** for this screenshot)

6.4.6.6 Previewing the Scene

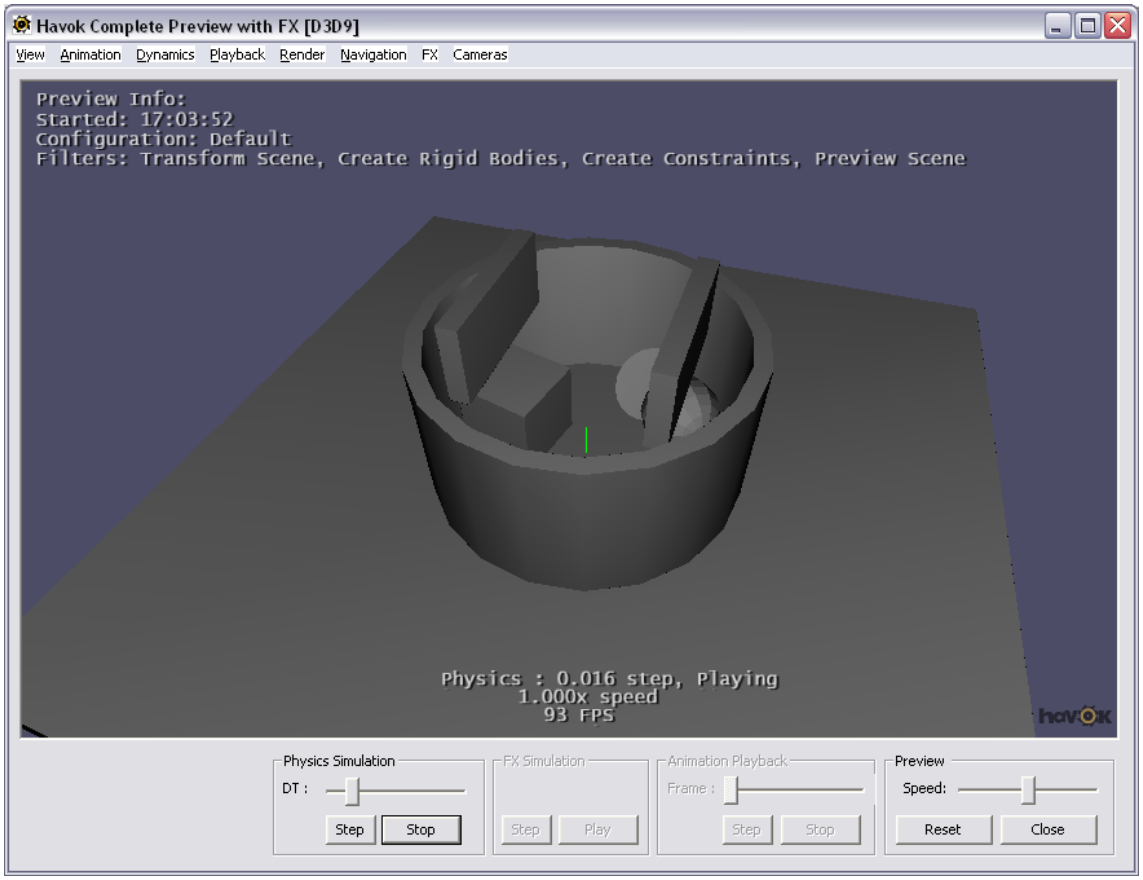
Now that we have set up our constraints, let's see what effect they have on the simulation.

Open the filter manager again by exporting the scene .

We now need to add a new filter, the Create Constraints filter (in the **Physics** category), in order to create constraint objects from the data we set up in Maya. Place it after the Create Rigid Bodies filter, and before the Preview Scene filter:



If you now execute the processing by pressing **Run Configuration** , you will see the constraints in action:



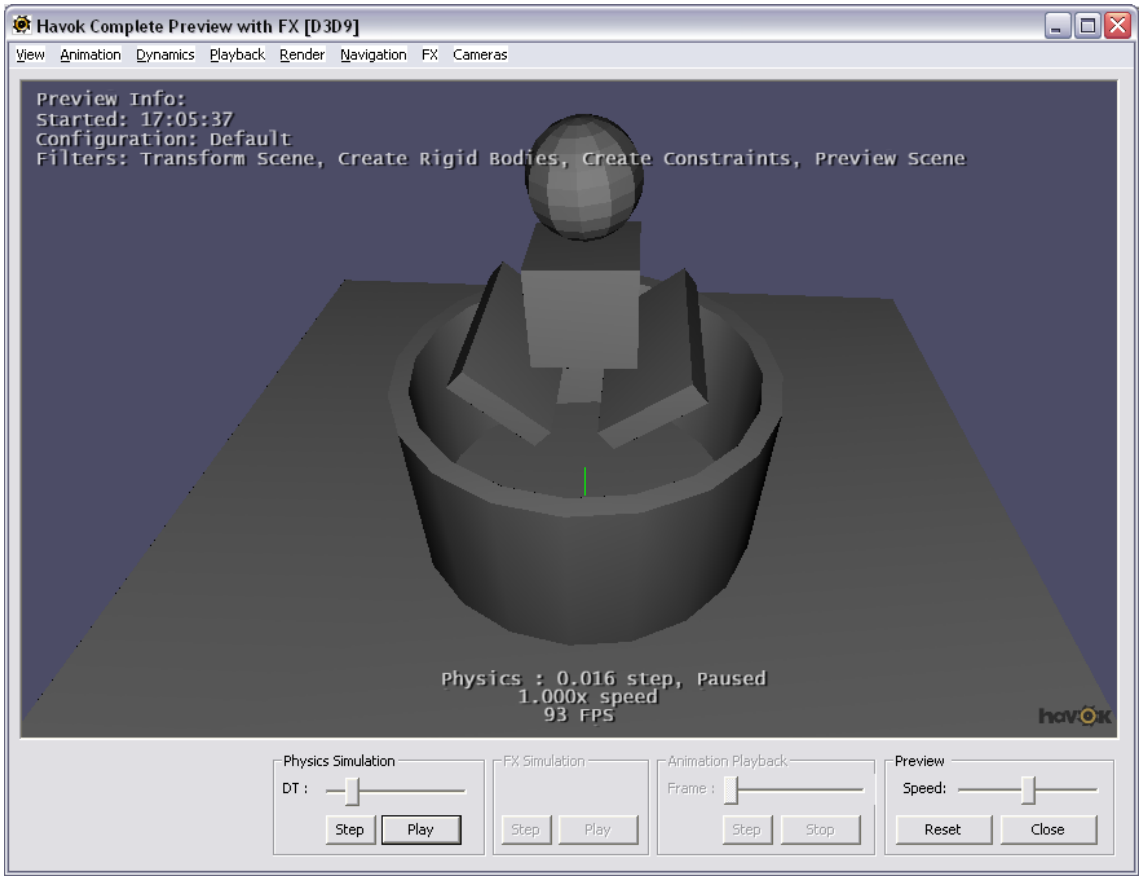
You can check that the limits we applied to the hinges are working by using mouse picking - you will notice that the doors won't move outside the limits we specified.

Close the preview window and the filter manager, and return to Maya.

6.4.6.7 Adding friction to constraints

Our setup is almost finished. The last thing we'd like to fix is the fact that the trap doors open even before the objects hit them (due to gravity). We'd like to add some friction to the hinges so the doors only move when pushed by the objects falling on them. Select each of the trap doors and locate the hinge constraint tab in the attribute editor. Change the **Max Friction Torque** parameter to 40. This parameter specifies how much torque (angular force) is absorbed by the hinge's friction - only torques above this value will cause the hinge to move.

If you now export and preview the scene, you will notice how the trap doors remain closed until the objects fall on top of them.



And this finishes our tutorial. You can find the fully completed tutorial in the scene file "`tutorialEnd.mb`". Try varying the properties of the rigid bodies and constraints, and observing the effects by running the filter setup.

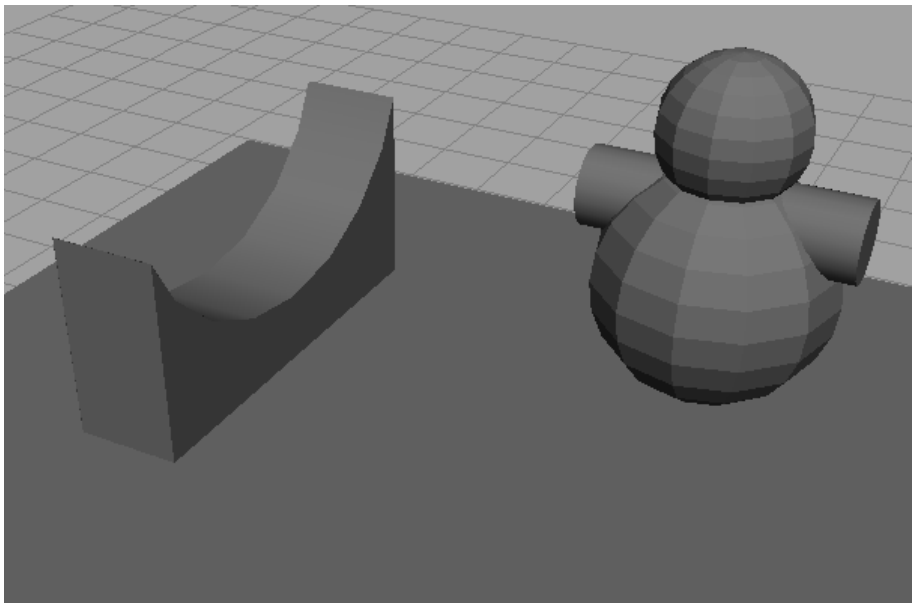
6.4.7 Tutorial: More on Rigid Bodies

In this tutorial we will explore some more concepts regarding rigid body setup, such as creating compound rigid bodies and changing the center of mass. This tutorial assumes that you are acquainted with the basics of exporting and processing assets and working with rigid bodies. We recommend you follow the first two tutorials (Export and Animation Basics and Physics Basics) before proceeding with this one.

6.4.7.1 Getting Started

Start by loading the scene "`tutorialStart.mb`", located in the "`tutorials/moreOnRigidBodies`" folder of your Maya module ("`Program Files/Havok/HavokMayaModules/...`" by default):

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

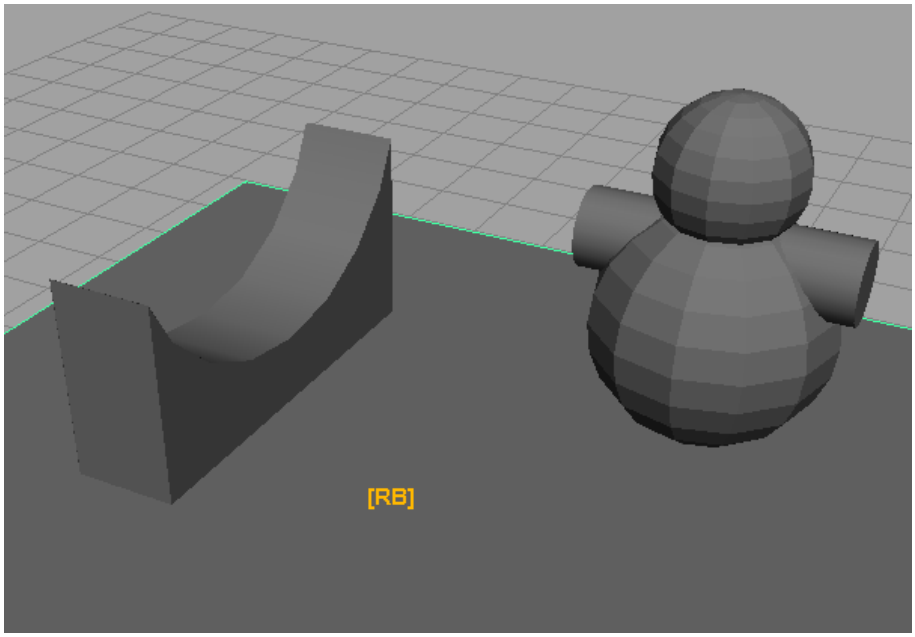


In the scene you will find the following:

- Two spheres and a cylinder, making up the shape of a little toy man
- A ground box
- A concave object (a little ramp)
- Two (hidden) convex objects that together make a shape similar to the ramp

6.4.7.2 Creating a Fixed Rigid Body

Let's start by making a rigid body out of the ground box. Select it and press the **Create Rigid Body** button or menu option.



Since the default mass is zero, the ground box will be fixed in space. That's exactly what we want.

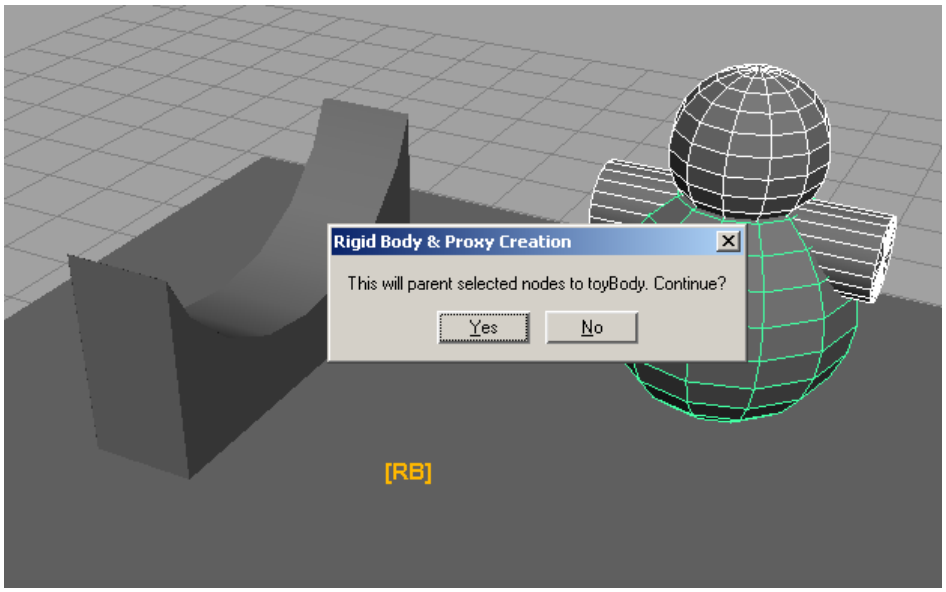
6.4.7.3 Creating a Compound Rigid Body

For the toy man we want to create a single rigid body from the two spheres and the cylinder. When we create a rigid body like this out of multiple objects, we are creating a compound rigid body.

When creating a compound rigid body, one of the objects needs to take the role of the rigid body - this object will hold the rigid body information, and the rigid body will be named after it. The other objects need to be children of that rigid body object. If the objects are not parented already, the last object selected will be considered to be the rigid body object and the other objects will be reparented accordingly.

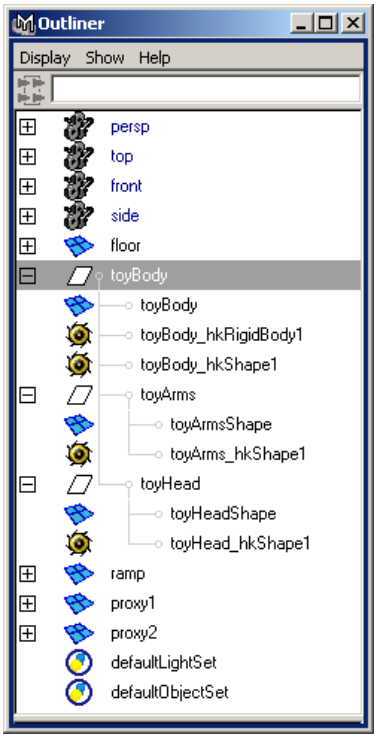
Select, in turn, the head, the arms and finally the body of the toy (holding SHIFT for multiple selection). Then click on the **Create Compound Rigid Body** button or menu option:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



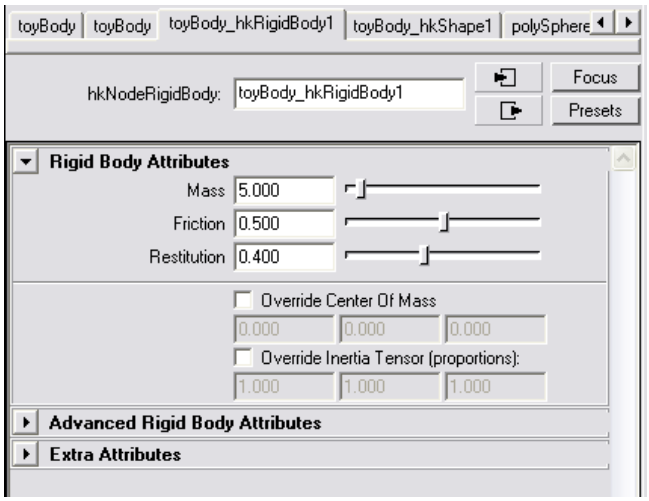
A confirmation dialog appears. This is because, as we mentioned, compound rigid bodies need to form a hiearchy. The Havok Content Tools have assumed that the last one selected (*ToyBody*) should be the parent (the one that will hold the rigid body information) - but is asking us for confirmation before proceeding to reparent the arms and the head. Click **Yes** to continue.

After this, *ToyArms* and *ToyHead* will be parented to *ToyBody*. Shape information (a Havok Shape Node) will be added to all three objects (as the three of them are used for collision detection). Rigid Body information (the Rigid Body Node) will only be added to *ToyBody*. You can verify this setup using the outliner:



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

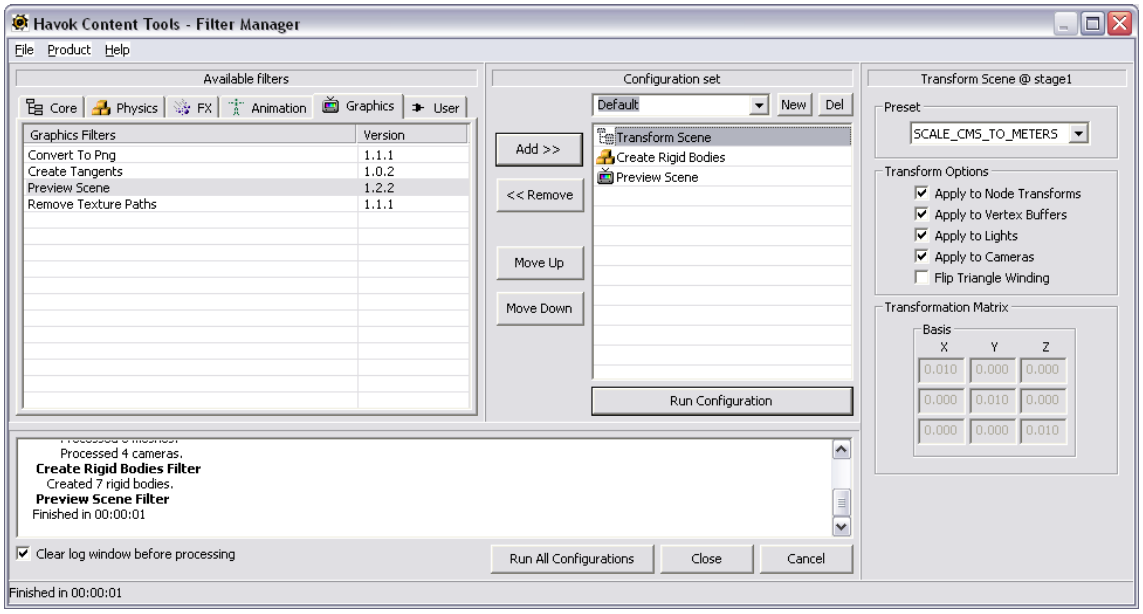
Notice that the rigid body has a default mass of zero. Since we want it to be dynamic, select the attribute editor tab for the the rigid body node and set the mass to a different value, let's say 5 Kg:



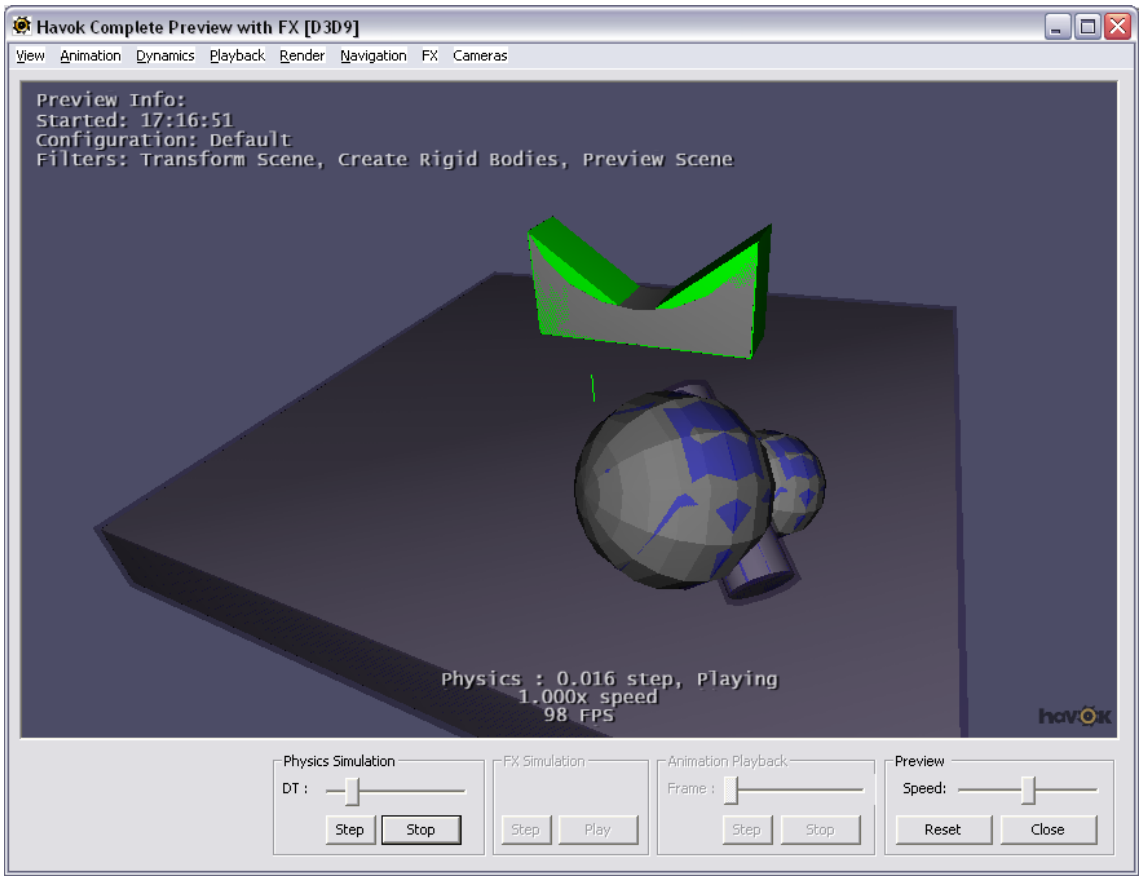
It is also useful to locate the attributes of the shape nodes: notice how the arms of the toy have the *Shape Type* parameter already set to " **Cylinder** " and the head to " **Sphere** ". The Havok Content Tools will automatically detect some Maya shapes and set the appropriate Havok Shape Type for them when creating rigid bodies - however it is still useful to always check the shape type value to ensure that it is as expected.

Previewing the Scene

Let's now preview the behaviour of our compound rigid body. Click on **Export** and, in the filter manager, add the Scene Transform (using the **SCALE_CMS_TO_METERS** preset), the Create Rigid Bodies and the Preview Scene filters (as we did for the previous tutorial):



Click on *Run Configuration* - this will process the asset and open a preview window:



As you can see (use mouse picking to interact with the toy man) the three objects behave as a single rigid body.

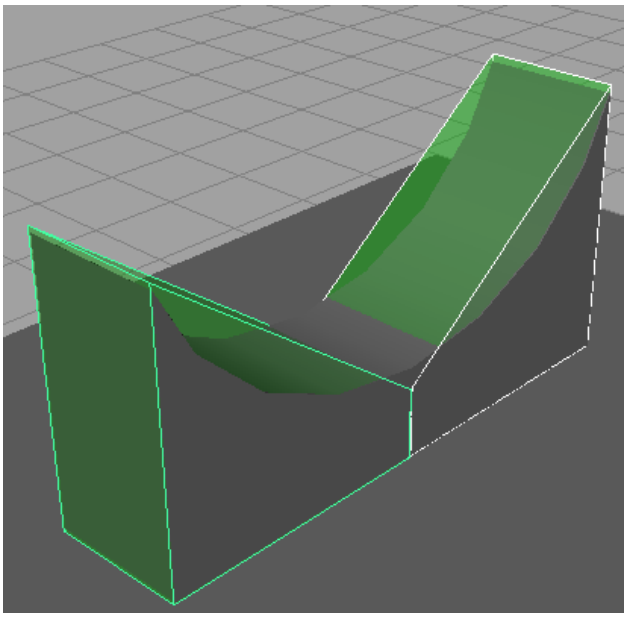
Close the preview and the filter manager and return to Maya.

6.4.7.4 Creating a Rigid Body with Proxy Shapes

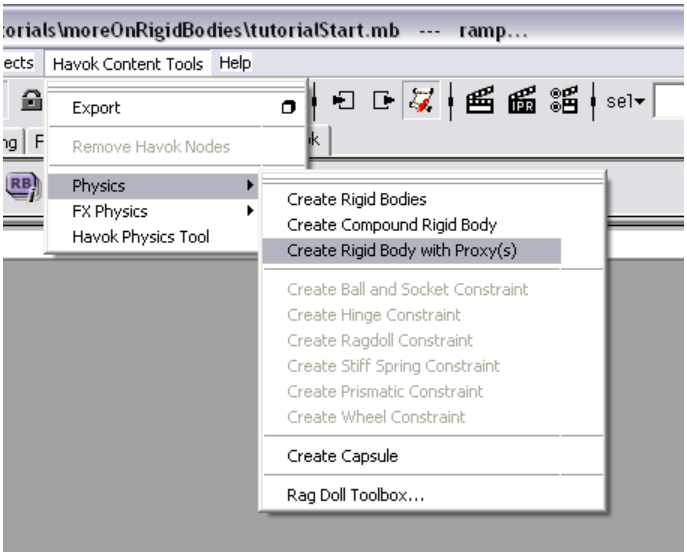
Sometimes we may want to simulate an object as a rigid body, while using another object or set of objects for the collision detection rather than the mesh of the object itself. The usual example of this case is when we may want to simulate a concave object as the combination of multiple convex pieces. Creating a rigid body with proxy shapes is very similar to creating a compound rigid body. The only difference is that, in this case, the object containing the rigid body information (the rigid body node) does not contain shape information (the shape node) since its mesh is not used for collision detection. Shape information is instead taken only from its descendants.

Let's see this in action in this scene. Notice how the white ramp is a concave object. Ideally, we'd like to approximate its rigid body shape it using several convex objects (since simulating convex objects is far more efficient than simulating concave objects). We will use two convex pieces provided to create a proxy shape representation of the ramp. Locate and unhide the two proxy objects (*proxy1*, *proxy2*), then move the objects so the two pieces and the concave ramp match their locations in the scene. We have coloured the proxy obejcts transparent green to make it easier to distinguish them:

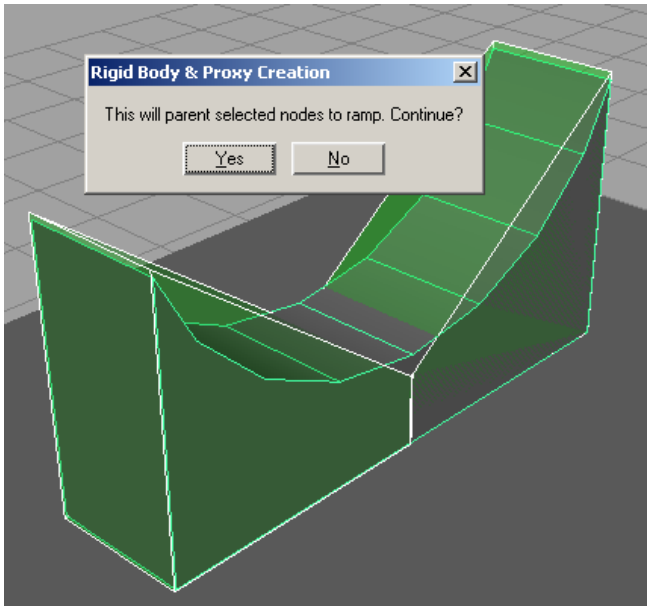
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



We now want to convert the ramp to a rigid body while using the two convex pieces as the shapes. So, select the two green convex shapes first and then the concave ramp, then click on the **Physics** > **Create Rigid Body with Proxy(s)** menu option.

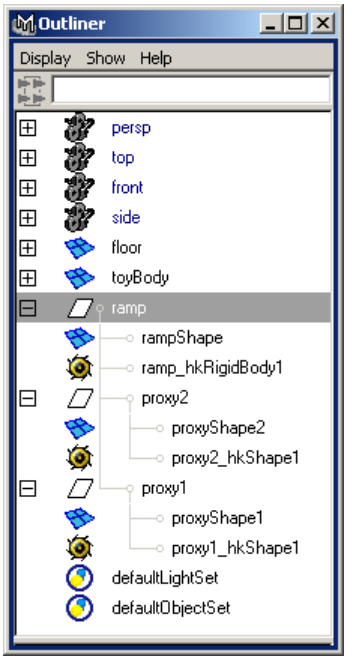


Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



In the same way as for compound bodies, the rigid body object must be the parent of the shape objects - the dialog above notifies that the proxies must be reparented. Click on **Yes** to confirm. (Alternatively, start by first parenting the two convex pieces to the ramp manually, and then with the three objects selected click on **Create Rigid Body with Proxy(s)** .)

Notice how the *ramp* object now has a rigid body node only; and how the two convex pieces (which are now children of it) have a shape node only:



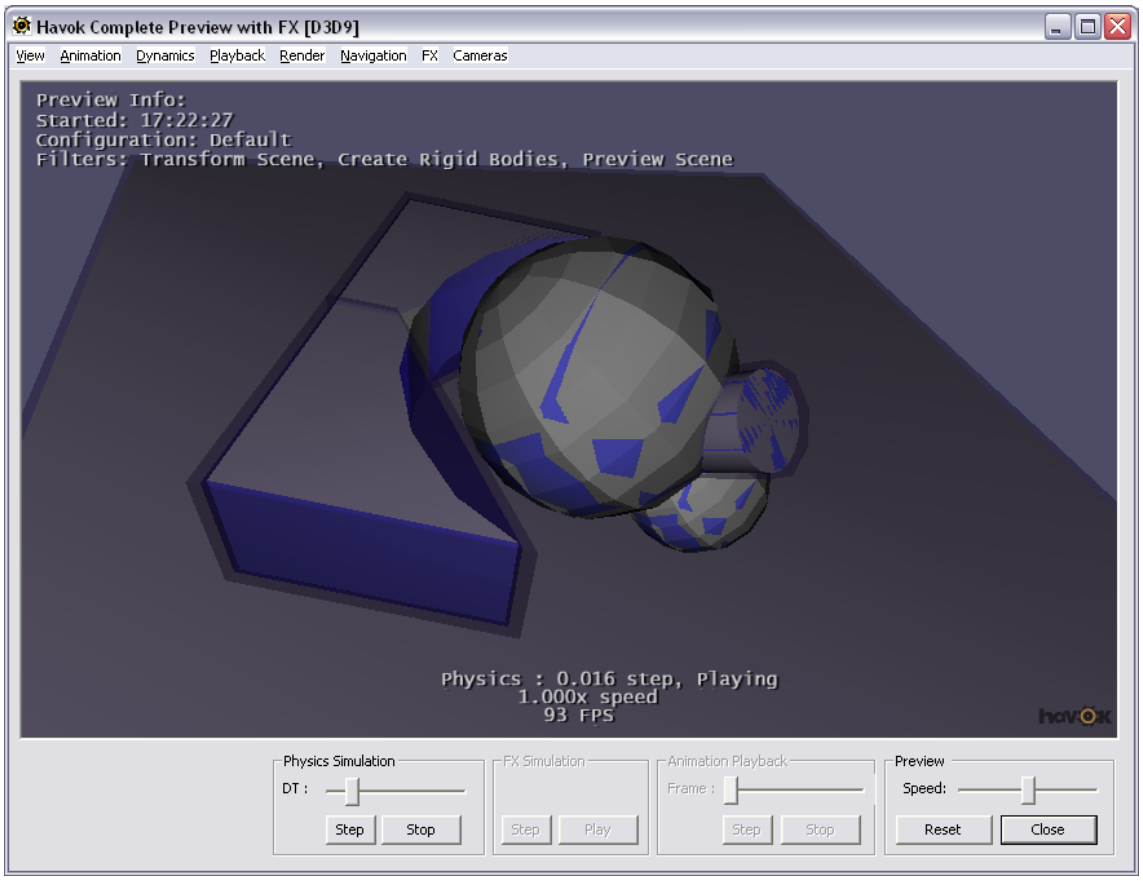
Since the two convex pieces are neither boxes, spheres, capsules or cylinders, their *Shape Type* attribute should be set to " **Convex Hull** ". Ensure that this is indeed the case. Also notice how, again, the

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

default mass of our rigid body is zero. Since we want it to be dynamic (movable), it should be set to something different (5Kg for example).

Previewing the Scene

Let's see how the ramp behaves in our preview. Export the scene and press on **Run Configuration** to execute the preview. You should be able to see how the ramp is simulated using the convex pieces. To view exactly what is simulated, enable **Physics Ghosts** in the **View** menu of the preview. Notice how the two convex pieces, and not the concave mesh, is being used to simulate the ramp:

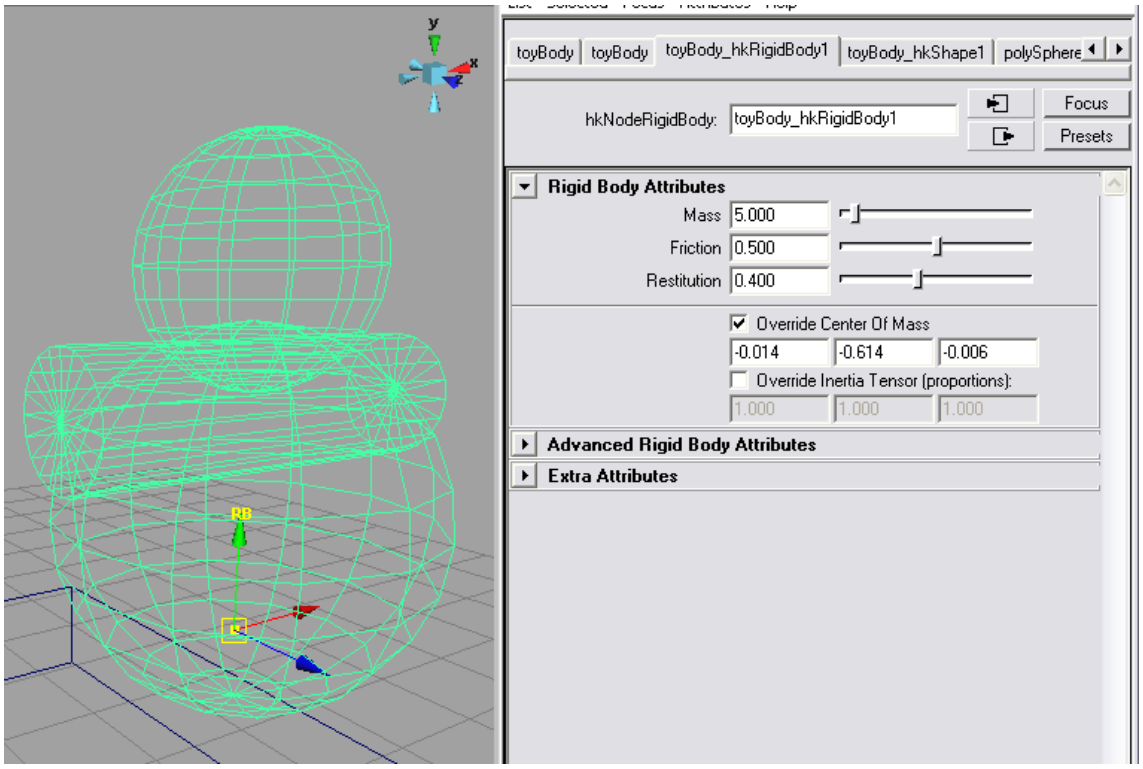


6.4.7.5 Changing the Center of Mass

We'd like out little man to behave like a tumbler - one of those toys that keep themselves straight. Those toys work by having a very low center of mass (usually by having a lead weight inside them). The Havok Content Tools allow you to easily change the center of mass of rigid bodies.

In Maya, select the *ToyBody*'s rigid body node, and in the **Mass** section of the attribute editor, check the **Override Center Of Mass** check box. This enables editing of the local center of mass coordinates. If the Havok Physics Tool is enabled, the center of mass is represented as a point in the viewport which can be manipulated by dragging it:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

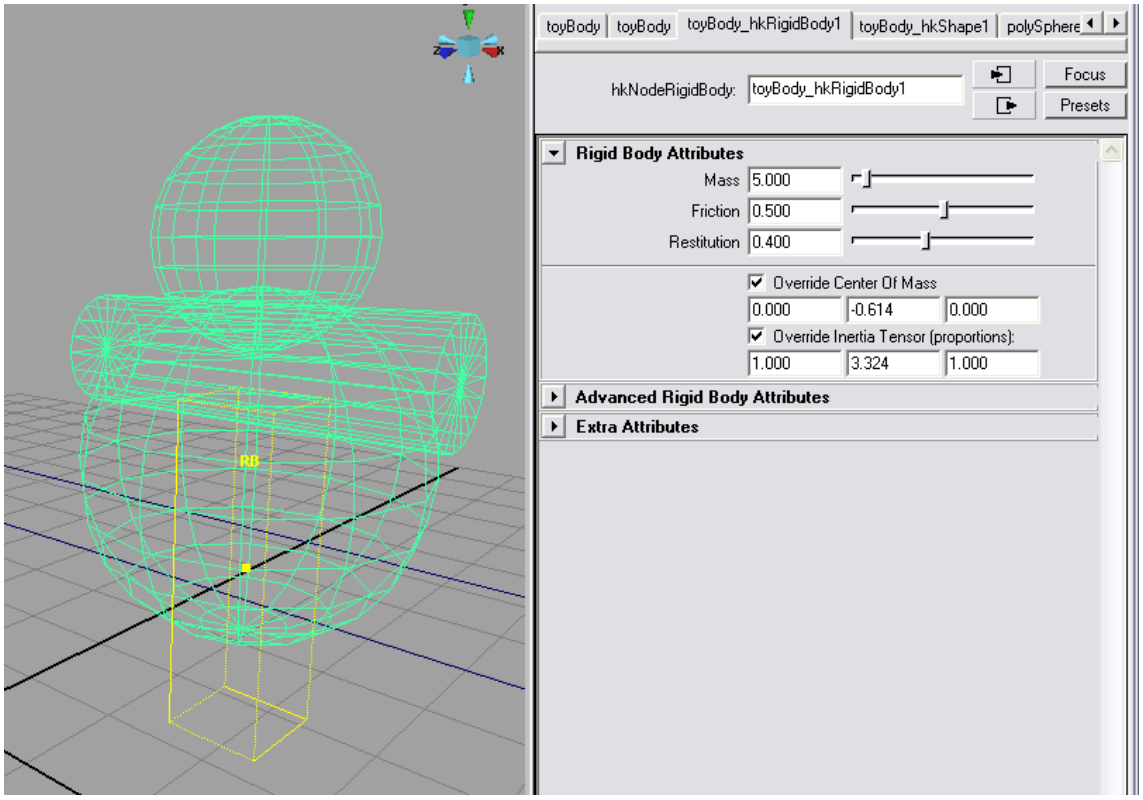


Postion the center of mass towards the base of the toy. If you execute the preview again (export and press **Run Configuration**), you see how our little man now keeps himself upright as he is dragged around the scene.

6.4.7.6 Changing the Inertia Tensor

We'd now like to customize the motion of our little man even more by changing the proportions of its inertia tensor.

Select its rigid body node again, and check the box **Override Inertia Tensor (proportions)** . This enables editing of the inertia tensor proportions of the rigid body. Again, the Havok Physics Tool provides a visual representation of the inertia tensor proportions - in this case as a scaleable unit cube which can be manipulated by dragging the scale handles:



If you now preview again, you will see that our little man will tend to wobble more around this vertical direction, while it won't lean as much as before. Experiment further with the center of mass and inertia tensor values, observing the results using the preview.

And this finishes our tutorial. You can find the fully completed tutorial in the scene file "`tutorialEnd.mb`".

6.4.8 Tutorial: Rag Doll Setup

The Rag Doll Toolbox allows the user to create and edit a new constrained rigid body hierarchy, to match to an existing bone hierarchy. In this walkthrough we will:

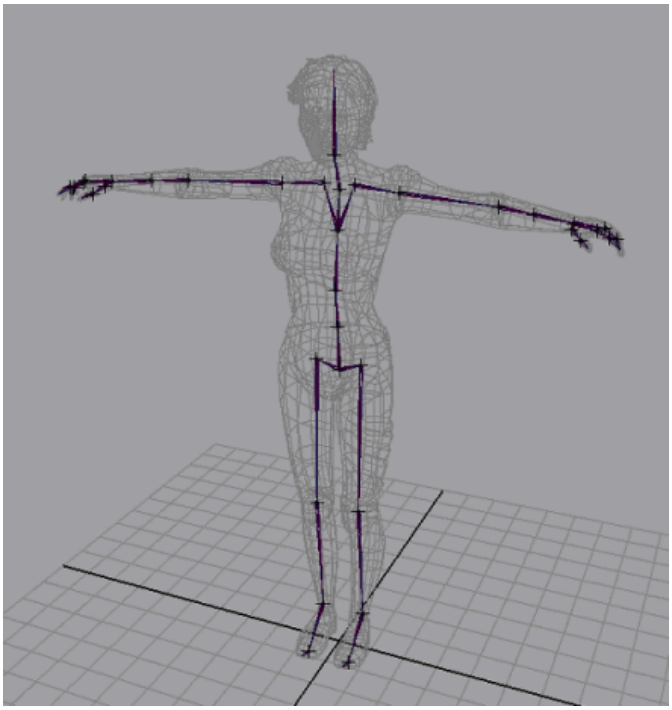
- Create and associate rigid bodies with an existing skeleton using the Rag Doll Toolbox
- Create appropriate constraints between the rigid bodies using the Rag Doll Toolbox
- Export and test the rag doll, using appropriate filters in the Havok Filter Manager

Note:
Maya tends to work in terms of 'joints' rather than 'bones' - in this tutorial the two terms may be used interchangably. Please read Issues with Maya Joints for more related information.

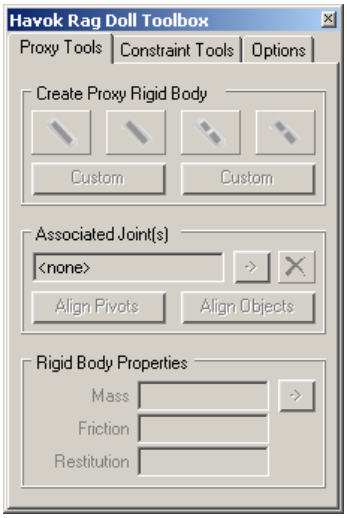
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.4.8.1 Getting Started

The sample files used in this tutorial may be found in "`tutorials\ragDollToolbox`", under the same location as your installed Havok Maya modules (normally "`Program Files\Havok\HavokMayaModules\...`"). Open the **"havokGirl1.mb"** sample file this contains a skeleton which has been skinned with a polygon mesh:



Open the Rag Doll Toolbox via the main **Havok Content Tools** menu or the toolbar icon :

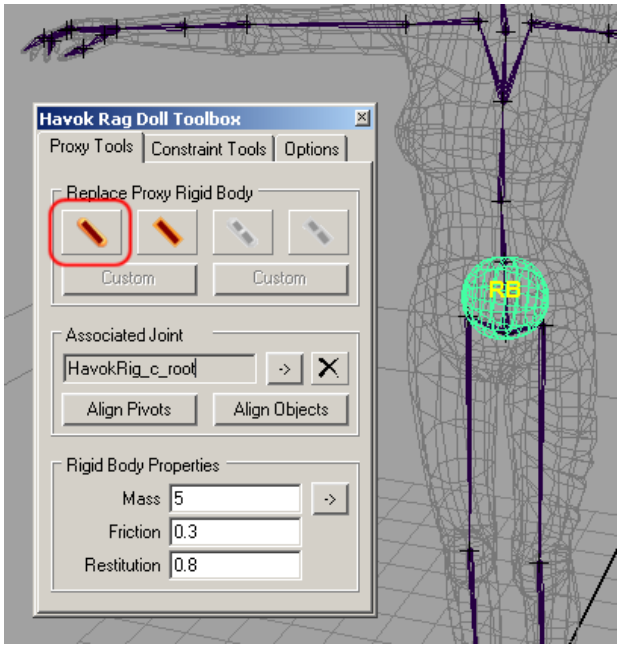


6.4.8.2 Creating Rigid Bodies

We will now create a set of rigid bodies to represent the physical rag doll for this character's bone structure.

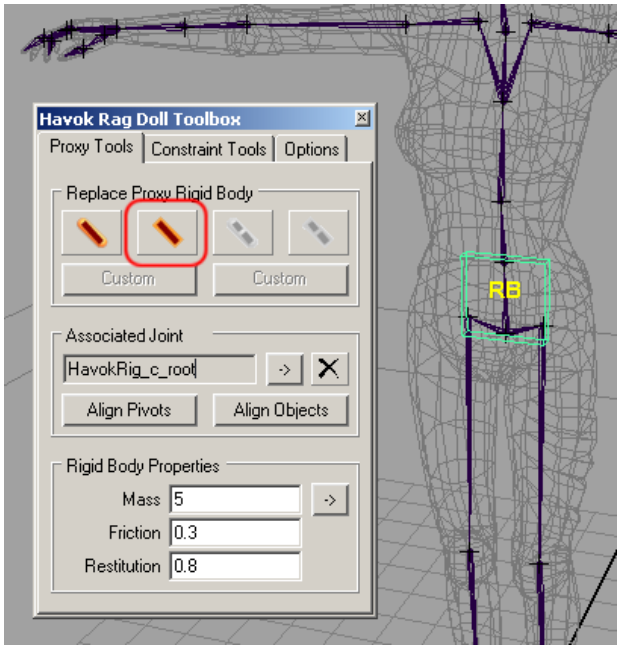
Simple Proxies

With the **Proxy Tools** tab of the Rag Doll Toolbox open, select the "*HavokRig_c_root*" pelvic joint. Since a single unassociated 'bone' is now selected, the first set of buttons in the "Create Proxy Rigid Body" section of the toolbox becomes enabled, allowing rigid body proxies to be created for the selected bone. Click the first button to create a new capsule proxy for the pelvic bone. A new capsule shape will be created to fit around the joint and its children, with Havok rigid body and shape nodes already attached, and initialized to a set of default values:

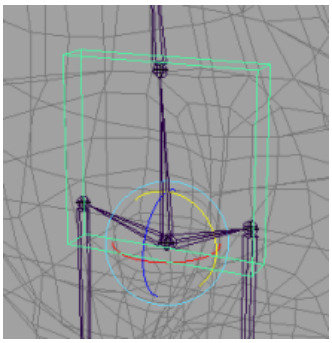


If you would prefer to use a box proxy instead, simply select the newly created capsule proxy and click the second of the two enabled buttons. This will replace the proxy with a box version, while retaining any rigid body properties you may have changed:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Observe that in each case above, the new proxy is created unparented (so as not to pollute the skeleton hierarchy). Observe also that the rotate pivot point of a new proxy automatically becomes aligned to that of the joint:



Note:

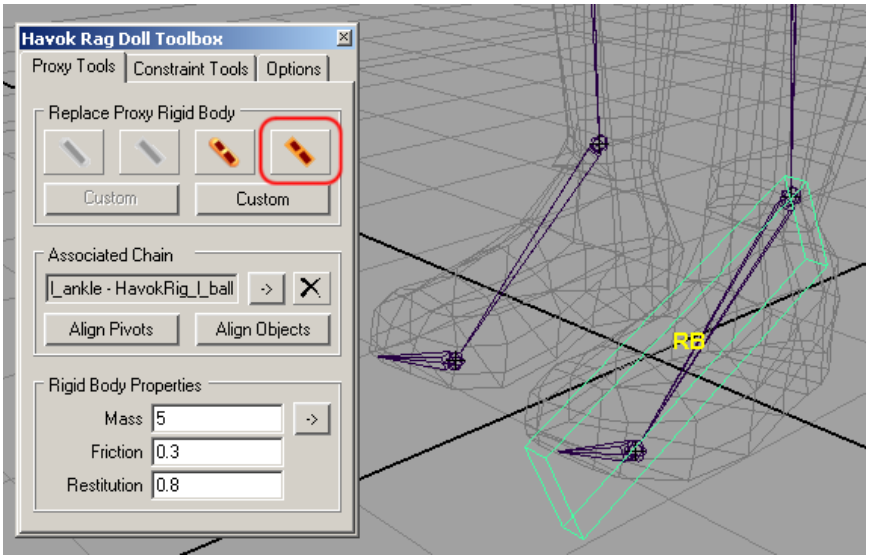
A bone can have at most one associated proxy. Therefore, if any selected bone already has an associated proxy then all buttons to create a new proxy will be disabled.

Chain Proxies

Suppose that you would like to create a single proxy to represent a chain of bones, for example a foot in this example. Select the two bones representing a foot (holding SHIFT for multiple selection)[5]. When a chain of bones is selected and none of them already has an associated proxy, the second set of buttons in the "Create Proxy Rigid Body" section of the toolbox becomes enabled. Click one of the 'chain' buttons to create a capsule/box proxy for the selected bones:

[5] Unfortunately Maya highlights all descendants of selected nodes in the viewport. This can make it difficult to see which bones are directly selected while creating chain proxies. Selecting nodes using the outliner panel helps if this is causes difficulty.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

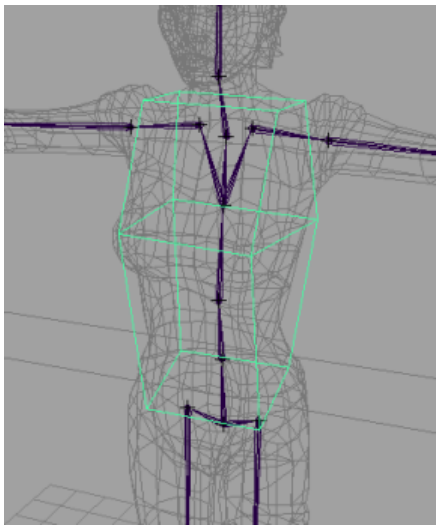


Observe that as before, the new proxy is created with the world as it's parent. Also observe that in this case the rotate pivot point of the new proxy automatically becomes aligned to that of the most senior joint in the selection.

Observe also that the new proxy is not yet oriented or positioned in a useful way. This will be fixed later in this tutorial.

Custom Proxies

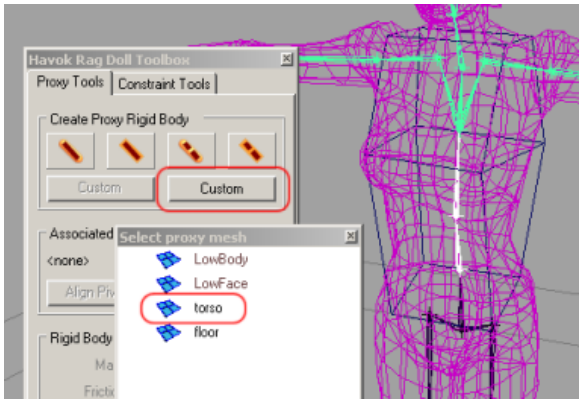
Sometimes a box or a capsule is not enough to accurately represent the volume around a bone or a chain of bones. For example, take the torso region of this model. In this case, instead of creating a box/capsule using the toolbox, we will first create a custom mesh and then associate it with the spine bones. Create a new polygon mesh in world space to represent the characters torso, as shown. Use any of Maya's standard tools to create the mesh:



To associate this mesh as a chain proxy for the spine bones, select the following three bones (holding SHIFT for multiple selection): "*HavokRig_c_back_1*", "*HavokRig_c_back_2*", and "*HavokRig_c_back_3*".

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Now click the "*custom*" button in the toolbox - this will open a selection window where you can pick the existing mesh to use as a proxy. Select your custom mesh and click "OK":



This mesh has now been associated as a proxy for the set of spine bones, and behaves like any other chain proxy - i.e: it inherits default Havok properties; it can be replaced by clicking the other chain proxy buttons; it's pivot is aligned to the most senior of the associated joints, etc.

Reshaping/resizing Proxies

After a proxy has been created, it probably won't accurately represent the skin volume. To correct this, the proxy can be moved/rotated/etc using the standard tools to achieve the desired fit, with one caveat: **non-uniform scaling should be avoided** . The reson for this is that the proxy meshes used in the modeller act as guides for creating the rigid body shapes (based on the the 'Shape Type' attributes). If any non-uniform scale is introduced into the proxy meshes then the exported rigid bodies may not accurately match the proxies displayed in the modeller. This is true for capsules in particular.

Therefore the preferred method of editing a proxy is to use the attribute editor to change the construction attributes (width/height/etc.) and use the translate/rotate tools to position the proxy.

As observed, whenever a new proxy becomes associated with a bone, it's rotate pivot point is automatically aligned to that of the bone. After moving or rotating a proxy, the *Align Pivots* button in the 'Associated Bone' section of the toolbox *should always be pressed* to ensure that the proxy pivot matches the joint position. This helps to create accurate rag dolls during the filter process.

Reshape and reposition any proxies created so far to achieve a good fit with the skin that they represent.

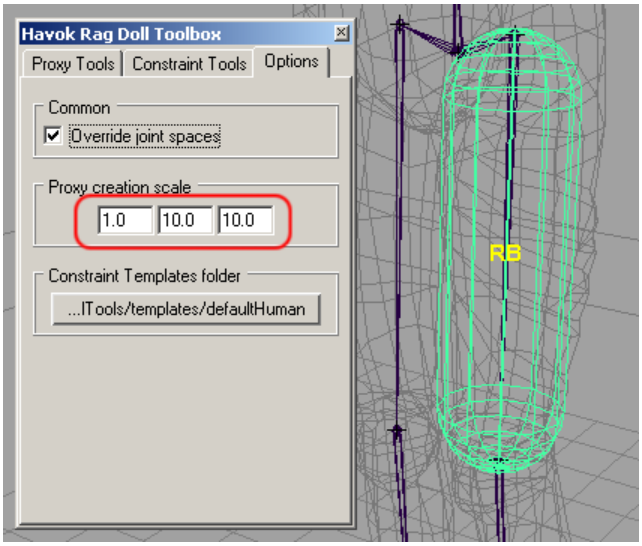
Using the 'Proxy Creation Scale' option

If your character's bones are much narrower than the volume of your character's skin, as in this case, then most new proxies which you create using the Rag Doll Toolbox will require reshaping to match the skin volume. The 'Proxy Creation Scale' option is provided to help this scenario.

This option sets a scaling factor which is used during the creation of new capsule/box proxies. This can be thought of as an additional local [X,Y,Z] scale applied to a bone before a new proxy is created to fit around it., where 'X' is **usually** the primary bone axis (see Maya Joint Issues).

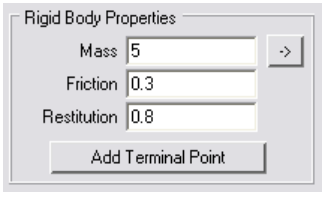
For example: in the *Options* tab set the values to [1,10,10], then in the *Proxy Tools* tab create an new capsule proxy for a thigh bone. This results in a proxy which is ten times wider than the bone but has the same length:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Rigid Body Properties

The "rigid body properties" section of the Proxy Tools tab allows quick access to rigid body properties of selected proxies. As you create proxies, think about these properties and try to set appropriate values for each proxy:



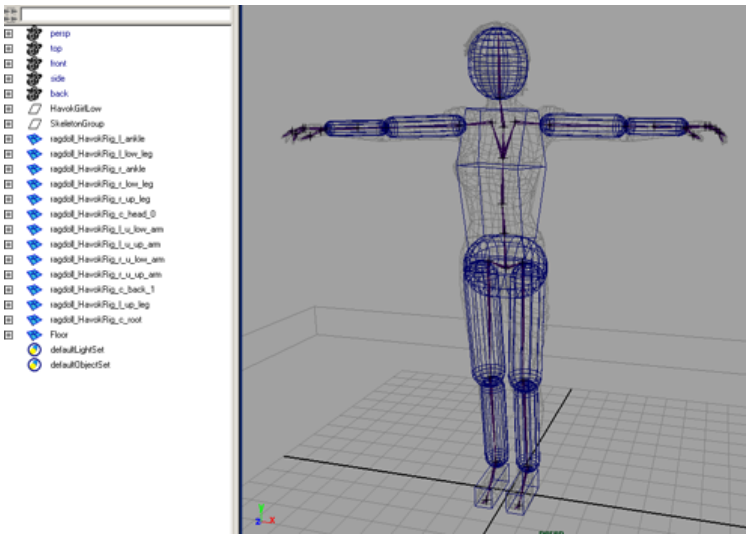
These properties can also be applied in bulk, for example to set the friction property for all of your proxies you can select them all together and input some value into the 'Friction' field above.

Apply suitable properties to the rigid bodies as you create them. For instance, the torso would be a lot heavier than a foot.

Finishing with rigid bodies

Continue creating, reshaping and positioning proxies for this character's skeleton. Remember to align the pivots once you have positioned your proxies, and to set appropriate rigid body properties. You may wish to open the sample " **havokGirl2.mb** " file - this contains the original skinned skeleton with rigid bodies created and associated to most of the bones:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Examine the proxies before continuing. Points of interest:

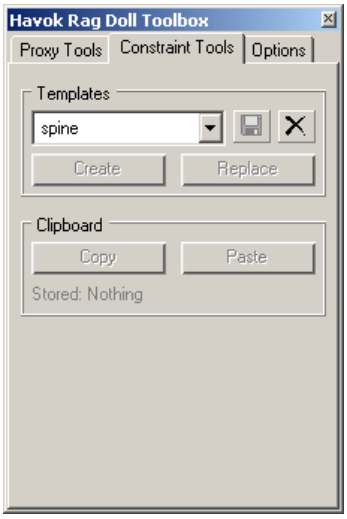
- All proxies are parented to the world the skeleton hierarchy is unchanged
- All proxies have their pivot points aligned to their associated bone(s)
- Some proxies are associated to bone chains
- The torso uses a custom (convex) mesh
- Masses differ for each proxy eg. the torso is heavier than the feet
- Some proxies overlap, others don't - there is no requirement either way
- Some bones are not associated (eg. hands). We don't require that amount of rigid body detail for this tutorial. Feel free to add additional proxies if desired.

You may export and preview the scene before continuing if desired. An appropriate filter setup is included with the scene. On previewing, the rigid bodies fall to the floor and are not connected to each other or to the skin, as expected.

6.4.8.3 Creating Constraints

At this point, you should have a scene with a skinned skeleton and a set of rigid proxies associated with the bones. To finish setting up our rag doll, we need to organize the proxies into a constrained hierarchy. The **Constraint Tools** tab of the Rag Doll Toolbox contains the tools to achieve this.

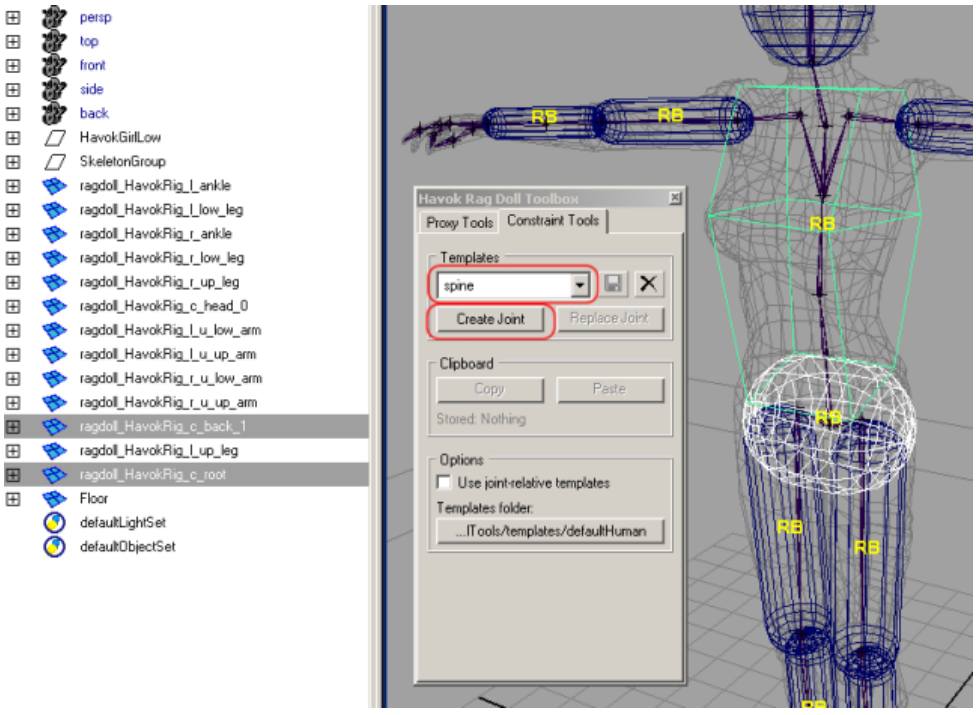
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Applying Constraints

Templates are representations of Havok constraints, saved as text files. For instance an "knee" template might describe a limited hinge constraint with specific limits and other attributes. They provide a useful way of wrapping a constraint type and attributes together in a re-usable way. Templates are stored together in a specific directory (which is customisable via the 'Options' tab). The Havok Content Tools installer installs a set of default templates for human characters, in the default template directory. We will use these templates.

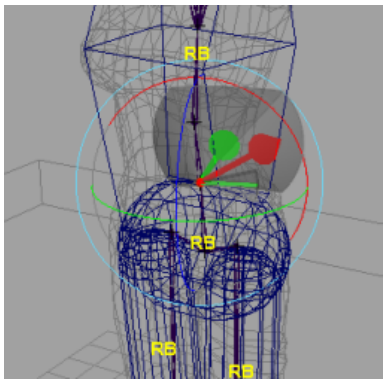
To apply a template, a pair of rag doll proxies must first be selected. Select the pelvis and torso proxies together, then select the "Spine" template from the drop-menu and click the *Create* button:



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Observe that several things happen at this stage:

1. The two proxies are automatically re-parented into the correct hierarchical order
2. A constraint node is added to the child proxy (the torso)
3. The constraint's properties are set from those stored in the template



The new constraint may not be oriented correctly initially. This is due to a known issue with Maya - see Maya Joint Issues for more.

Modifing Constraints

Once a template is applied, you can always modify its properties by changing the constraint parameters using the attribute editor panel and/or the Havok Physics Tool. You can then save (and possibly replace) the constraint as a new template by clicking on the " **Save** " icon beside the template combo box.

Replacing Constraints

To replace a constraint in the scene which was added using templates, we need to select only the child proxy which contains the constraint node (the torso in this scenario), select a new template from the drop-menu, and click the *Replace* button. In this case no re-parenting is performed, since the proxies should already be in their correct hierarchal order from the first application of a tempate.

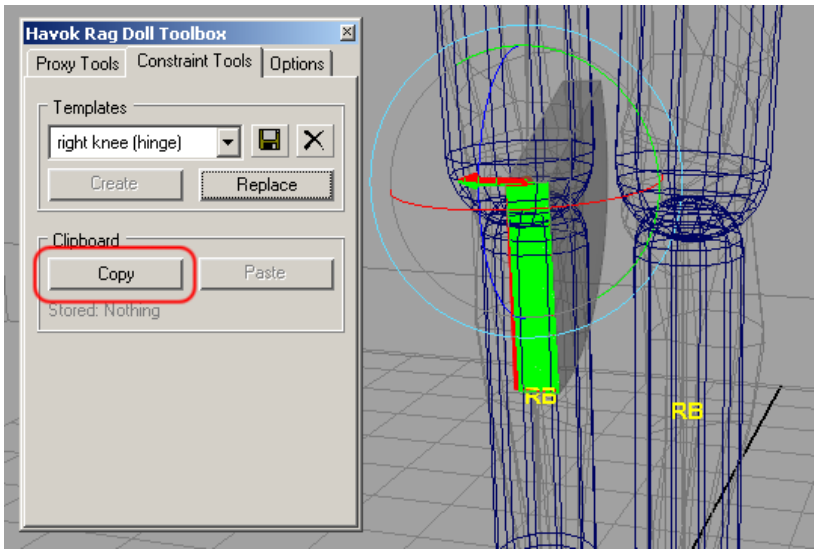
Try replacing the spine constraint with various other templates.

Copy and Paste

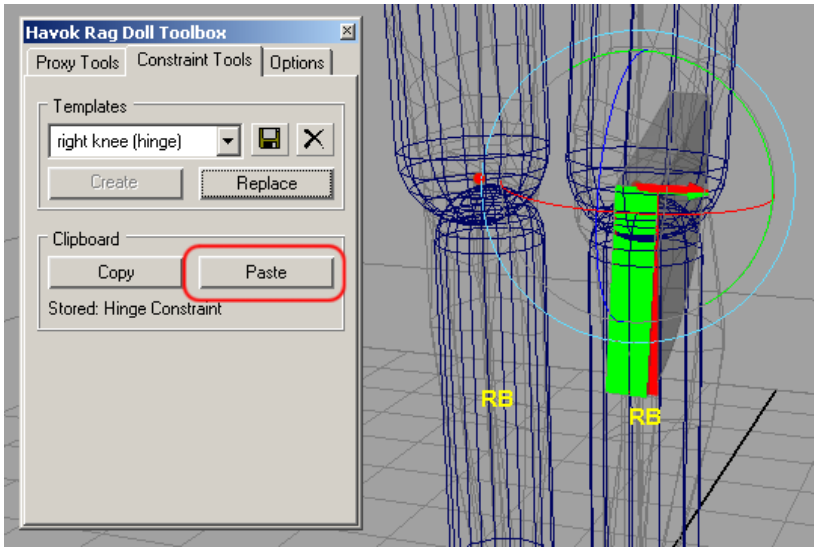
Symmetric characters usually have similar constraint properties on their left and right side. The copy and paste mechanism of the toolbox is useful in this case.

For example, create a "knee" constraint between one of the thighs and it's calf. Fix the orientation if necessary. Now, tweak some attributes of the constraint (limiting angles, max friction torque, etc.). When ready, press the **Copy** button. This generates a templated version of this constraint in memory.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



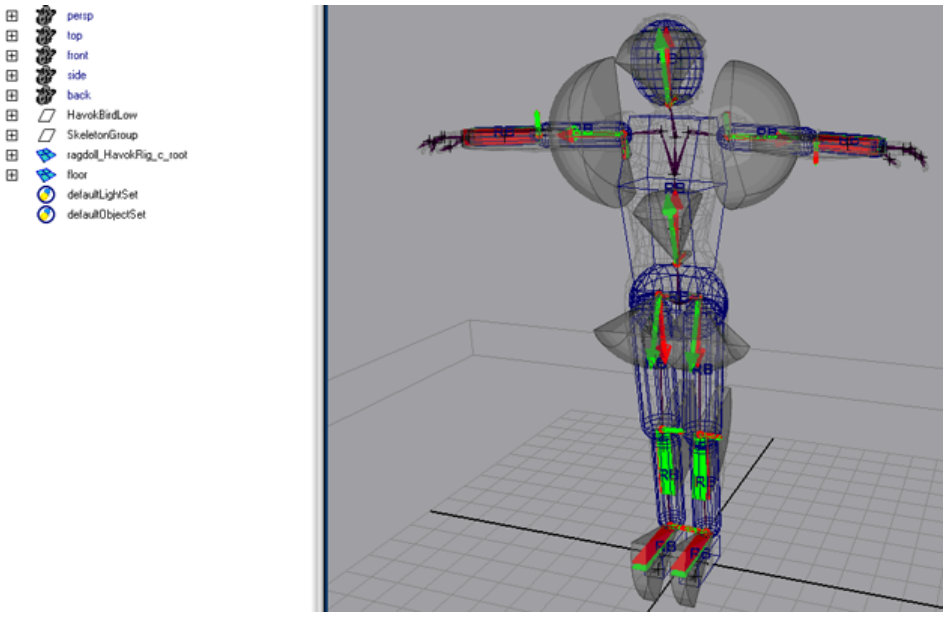
To apply this in-memory template to the other leg, simply select the other thigh and calf together and press the paste button. This acts in the same way as creating a new constraint using the 'Create' button, but uses the in-memory template instead. Unfortunately this may also require some manual fixup of the orientation.



Finishing with Constraints

Continue applying templates to each pair of proxies, until the rigid body heirarchy is complete. You may wish to open the sample " **havokGirl3.mb** " file - this contains the skin, skeleton, and the constrained proxy hierarchy.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



This screenshot is taken with the Havok Physics Tool active and set to show all constraints.

Examine the scene before continuing. Points of interest:

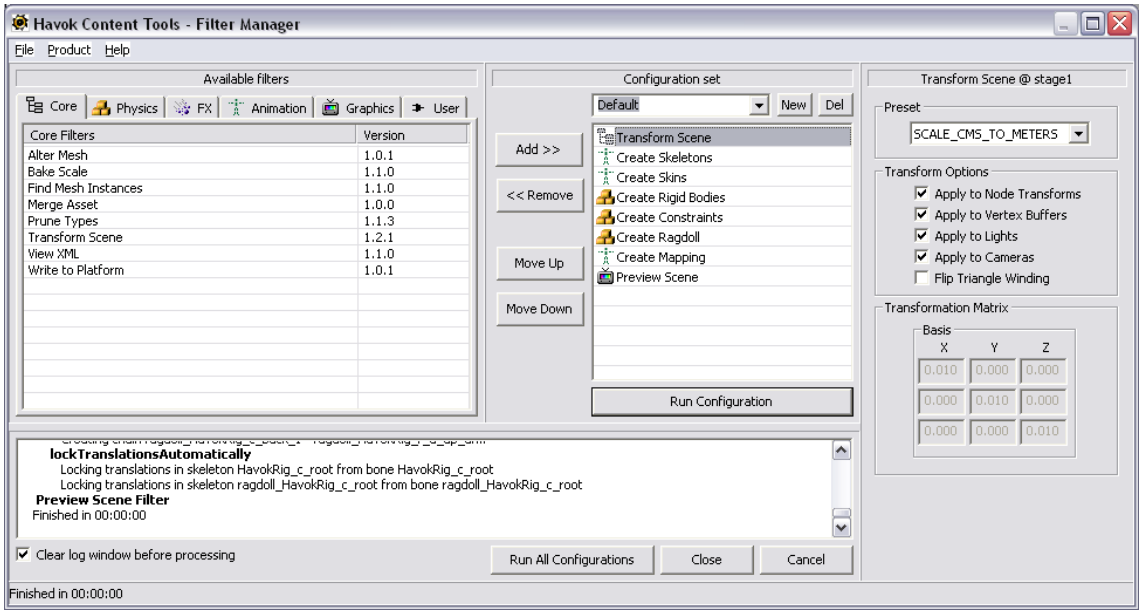
- The proxies now form a single hierarchy, matching that of their associated bones
- Each proxy has a single constraint (except for the root) , with it's hierarchical parent set as the constraint parent

6.4.8.4 Exporting the Rag Doll

At this stage your scene should contain at least:

- The original skeleton hierarchy
- The original skin
- The new constrained proxy hierarchy

Export the scene to the filter manager using the Havok exporter. The sample scene includes the appropriate set of filters to create the rag doll. These are:



- **Scene Transform**

  Maya exports everything in cm's and Havok works in meters, so for Maya we always need a SCALE_CMS_TO_METERS transformation.

- **Create Skeleton**

  Creates a skeleton object from the set of bones. Choose 'Skin bindings' in the 'Build Rig' section.

- **Create Skin**

  Creates a skin object for the skeleton.

- **Create Rigid Bodies**

  Creates the rigid body objects from the proxies we created (and the floor).

- **Create Constraints**

  Creates the constraint objects which constrain the rigid bodies together.

- **Create Rag Doll**

  Collects the rigid bodies and constraints into a rag doll object. Choose ' **Automatic** ' in the ' **Detect Rag Dolls** ' section.

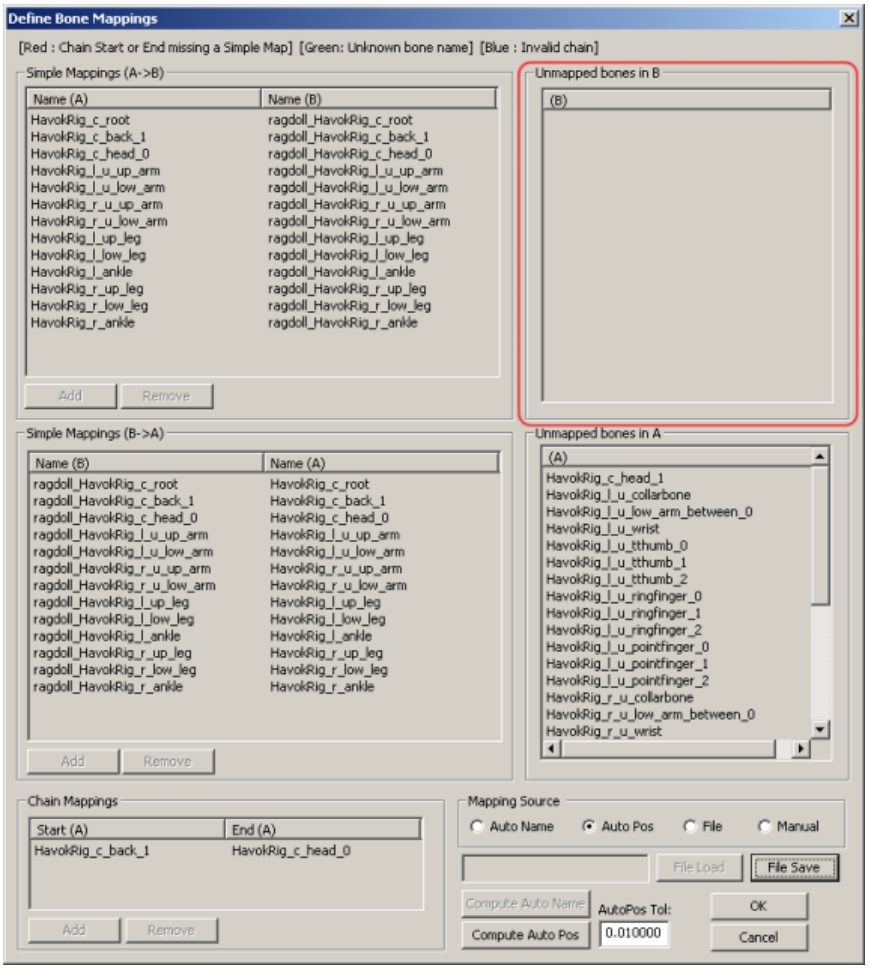
- **Create Mapping**

  Maps the skinned skeleton to the rag doll object, so that the rag doll can drive the bones (which in turn drives the skin).

  For ' **Skeleton A** ', pick '`HavokRig_c_root`' (the skeleton) For ' **Skeleton B** ', pick '`ragdoll_HavokRig_c_root`' (the rag doll)

  Click on ' **Define Mappings** ' to check the mapping:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



For a successful mapping, all of the bones in one skeleton should be mapped to (some of) the bones in the the other skeleton. In this case we can see that The are no unmapped bones in the 'B' skeleton (the rag doll) so this is a successful mapping.

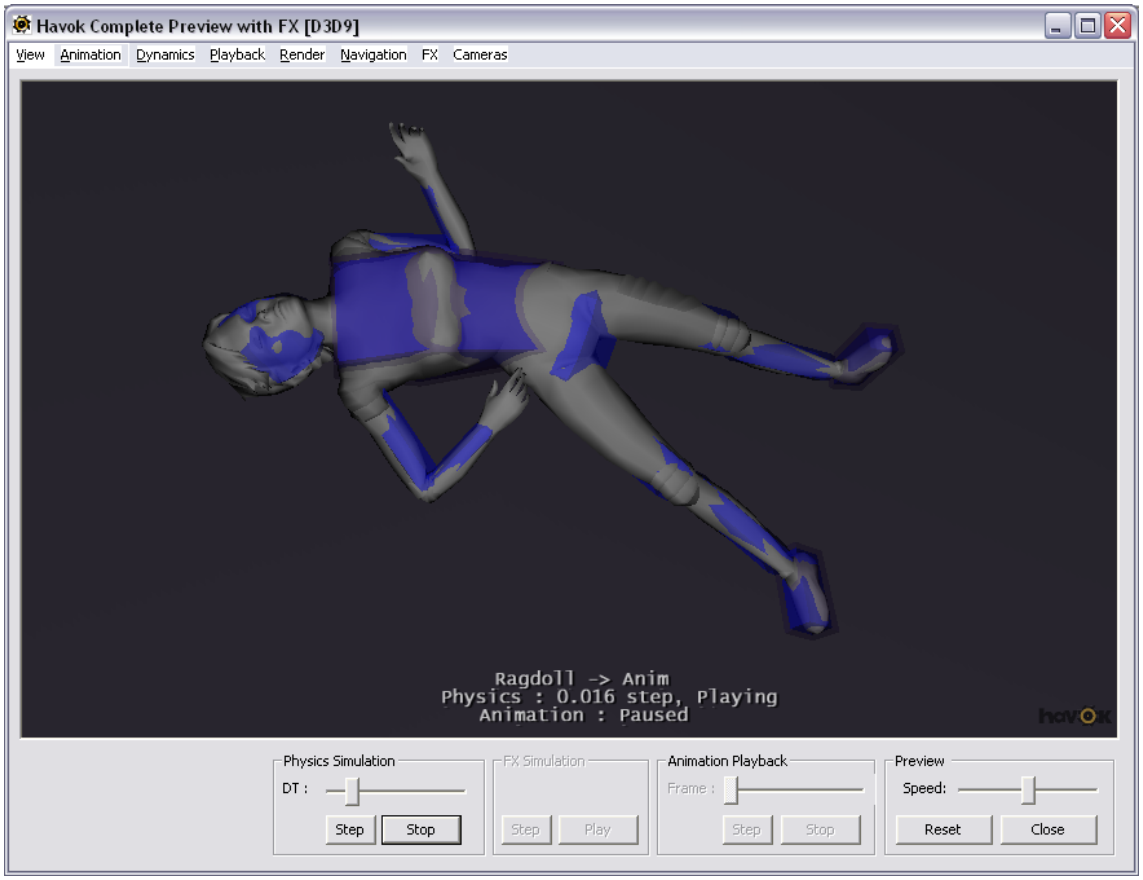
The default behaviour of the mapping filter is to match the skeletons by comparing the positions of the bones in each. This is why it was important to ensure that the pivot points of the rigid bodies were aligned to the pivot points of the joints.

Click ' **OK** '.

- **Preview Scene**

  So that we can interactively test the rag doll behaviour.

Run the filter manager. When the preview window appears, play the physics and interact with the rag doll using the space bar to pick and drag the rigid body shapes:



If the rag doll does not behave as expected, close the filter manager and review the rigid bodies and constraints in Maya. Tweak the attributes if necessary and continue to test using the filter manager until satisfied.

At this point you may want to serialize the completed rag doll to a file. See the export turorial for details on how to do this.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.5 XSI Tools

6.5.1 Introduction

In this chapter we are going to present the XSI specific components of the Havok Content Tools. These include:

- The XSI Scene Exporter
- The XSI Physics Tools
- The XSI Animation Tools

We also present several tutorials:

- Export and Animation Basics: How to set up, export and process an animation.
- Physics Basics: How to set up, export and process a physics scene.
- More On Rigid Bodies: A few more concepts about rigid body setup.
- Rag Doll Setup: How to set up, export and process a rag doll and its association with a high-res skeleton.

Check also:

- Extending the XSI Tools: For information about exporting custom data and scripting access to XSI.
- Common Concepts: For common concepts useful for all Havok Tools (regardless of modeler).
- The Havok Filter Pipeline: For details on how the content created by any modeler can be processed.

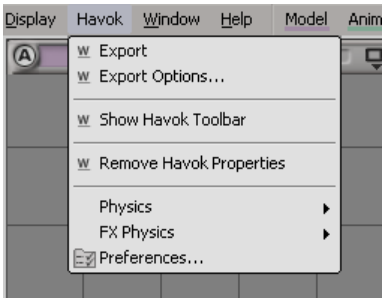
6.5.2 XSI: Scene Exporter

Havok's XSI Scene Exporter is responsible for exploring XSI's scene graph, collecting scene information, and passing it to the filter pipeline for processing.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.5.2.1 Accessing the Scene Exporter

The scene exporter is implemented as a custom XSI command. The best way to access the exporter functionality is to use the **Havok** menu in XSI:



You can also access functionality in the XSI Scene Exporter using the **Havok** toolbar. Use the **Havok > Show Havok Toolbar** or the **View > Toolbars > Havok** menu option ino order to show it. You can also switching to the **Havok** custom layout (through the **View > Layouts > Havok** menu option) - the toolbar should appear docked in the bottom left corner of the screen. You can also include this toolbar as an element of your own custom layouts.

Let's focus on the options related to the Scene Exporter:

- **Export**

  This invokes the Scene Exporter, which will navigate the current scene graph extracting information about meshes, lights, nodes, attributes, etc (according to the current export options). After this is done, the Havok Filter Manager will be invoked in order to process this data.

- **Options...**

  This command will open a Property Editor within XSI where variousexport options can be modified. These options are described next.

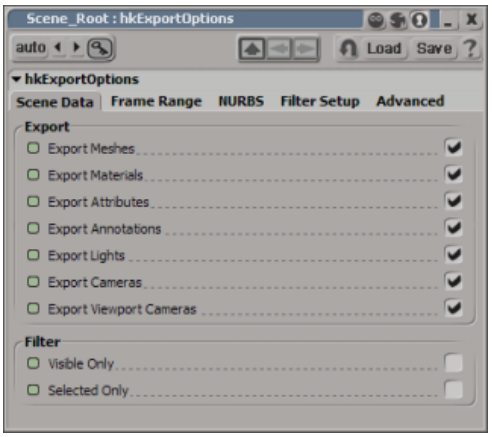
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.5.2.2 Export Options

The export options for the XSI Scene Exporter are stored as a *custom parameter set* in the scene. To view these options, click the **Havok > Options...** menu option or toolbar button.

The options are distributed among multiple tabs:

Scene Data

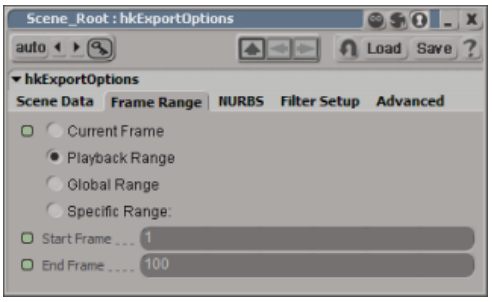


Options in this tab determine which types of objects are exported from the scene. The screenshot above shows the default settings.

- **Export Meshes** : Controls whether meshes should be exported.
- **Export Materials** : Controls whether materials and textures should be exported.
- **Export Attributes** : Controls whether custom Havok attributes should be exported. *Note that the Physics Tools depend on these attributes being exported.*
- **Export Annotations** : Controls whether annotations should be exported.
- **Export Lights** : Controls whether lights should be exported.
- **Export Cameras** : Controls whether cameras should be exported.
- **Export Viewport Cameras** : Controls whether any 'user' viewport cameras in use should be exported.
- **Visible Only** : If enabled, only objects which are currently visible in XSI will be exported.
- **Selected Only** : If enabled, only objects which are currently selected in XSI will be exported.

Frame Range

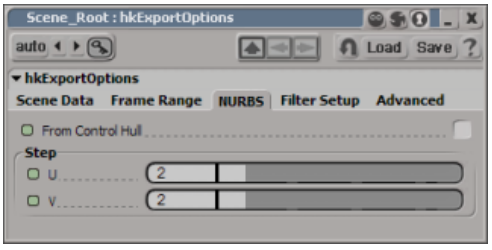
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Options in this tab specify how animated transforms are exported.

- **Current Frame** : Samples the scene only at the current frame. This option is useful if no animation data is required, as it speeds up the export process.
- **Playback Range** : Samples the scene using the current playback range.
- **Global Range** : Samples the scene using the global animation range.
- **Specific Range** : Allows you to specify a sampling range from **Start Frame** to **End Frame** .

NURBS



The Havok scene data format supports only polygonal meshes. Any NURB surfaces will be converted to a polygonal representation during export (the XSI scene won't be modified). These options control how this conversion is performed. Notice that these options reflect the parameters of the XSI **nurbsToMesh** operator - check the XSI documentation on this operator for more details.

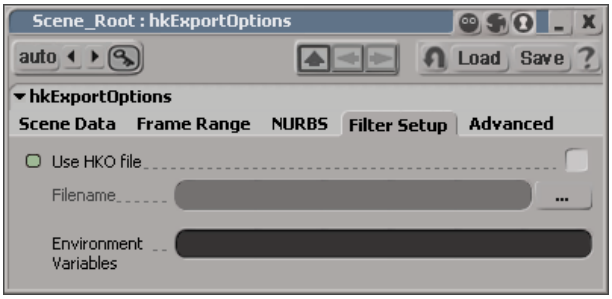
- **From Control Hull** : Controls the placement of the polygon mesh's vertices.
  - When this option is **off** , vertices will be located at the knot positions and along the interpolated NURBS surface. This is the most common option.
  - When this option is **on** , vertices will be placed at the positions of the surface's control points. This can be useful if the mesh is going to be used to create a subdivision surface.
- **Step U, V** : The number of edges on the polygon mesh for each span between successive knots on the original surface

Note:

Skinned NURB surfaces are not supported by the XSI Scene Exporter - the NURBS surface will be converted to a polygonal representation but the skins binding (bone weights) won't be exported (since they are based on the NURBS surface and not the new polygonal mesh).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

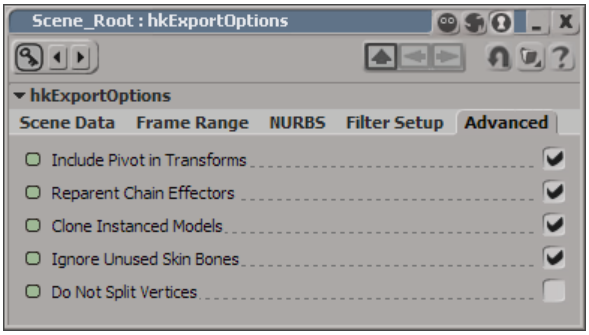
Filter Setup



These options allow customization of the filter manager and individual filter behaviour.

- **Use HKO file / Filename** : Filter configurations (set up in the Filter Manager) are stored as part of the the XSI scene, but sometimes it is useful to specify a custom set of options explicitly - for example, during batch processing, or to avoid having to build a filter set up from scratch in new assets. A set of filter configurations can be stored in an HKO file. Enabling this check box allows you to choose such an HKO file which contains the desired filter configurations to use for processing the asset. Check the Configurations section for more details.
- *Environment Variables*: Any variables specified here (through scripting for example) will be added to the exported scene under the "`hkxEnvironment`" root level variant. These variables may then be used by individual filters.

Advanced

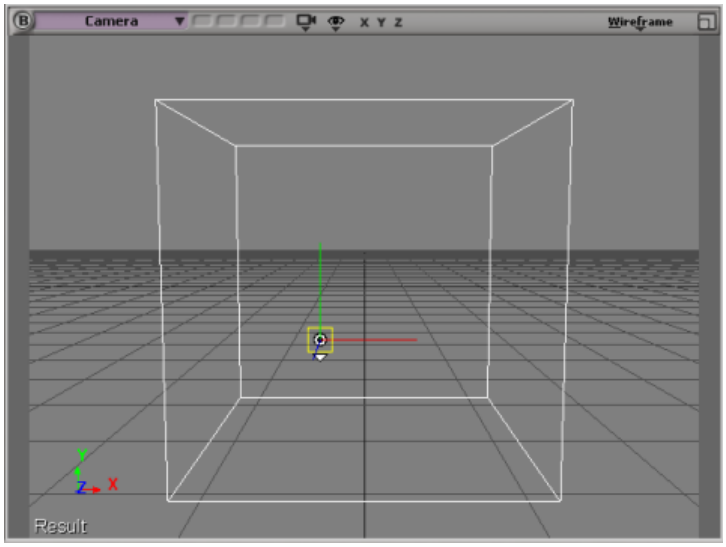


- **Include Pivot in Transform**

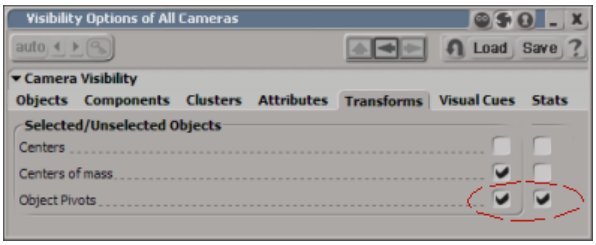
  It is often desirable to have control over the transform of an object without modifying the object itself. For example, the transform of a newly created box is usually specified at the center of the box. Changing the translation of this transform would move the box with it.

  XSI also defines the concept of *a pivot*, which is an extra transform associated with an object. Moving the pivot of an object doesn't move the object itself. By default the pivot is aligned with the transform of the object, but you can move it by holding **ALT** during a transform manipulation.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



You can also tell XSI to display a representation of the pivot in the viewports by changing the **Transform** settings in the **Visibility Options** menu:



The pivot is also used as the center of manipulation for the translation, rotation and scale tools. Thus, for hierarchical structures, it usually represents the location of the joint associated with the object.

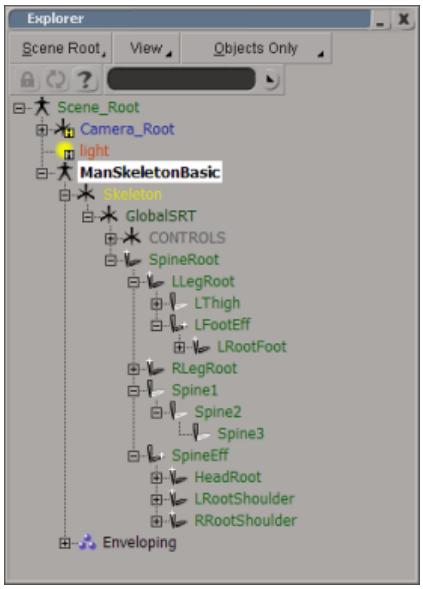
For all of these reasons, it can be very useful to export the transforms of objects in the scene as that of the pivot (which the user can fully control) instead of the object (which the user has limited control). The **Include Pivot in Transform** option enables this behavior. When this option is on, the exported transform of objects in the scene takes the pivot translation into consideration.

In particular, the Physics Tools extensively use the ability to manipulate the pivot in order to place constraints and align proxy rigid bodies to bones. Therefore in most cases you should leave this option enabled.

- **Reparent Chain Effectors**

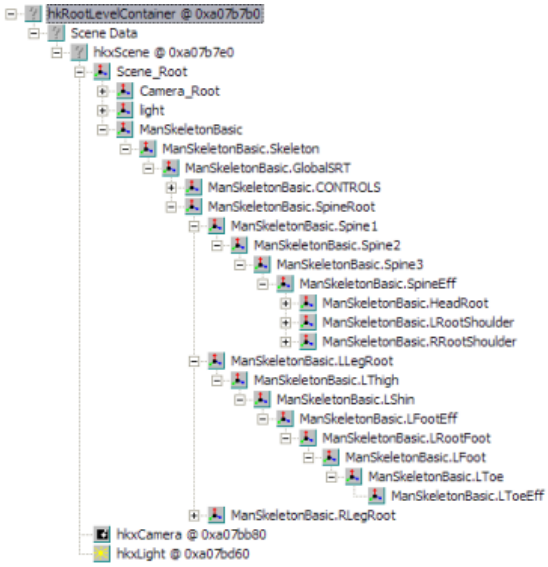
  It is quite common in XSI to model skeletons as multiple chains of bones not necessarily hierarchically connected in the same way they operate. For example, the node structure of the default biped (man skeleton) model in XSI is as follows:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

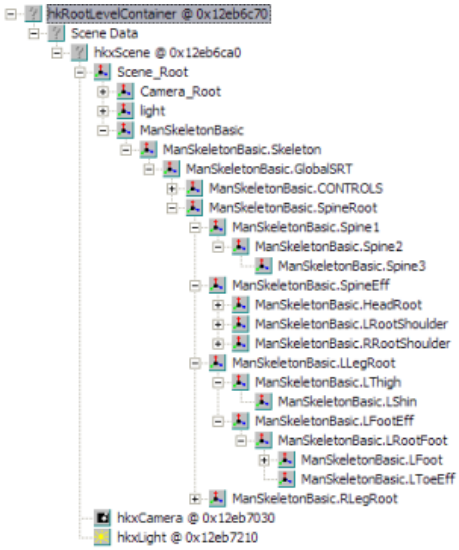


While you may expect the Head to be parented to the last link in the spine, notice how *HeadRoot* is parented to *SpineEff* instead of *Spine3*. Similar parenting can be found on the feet and arms.

In XSI the effector is treated as part of the chain, even if it's not a child of the last bone. In order to replicate this "implicit" parenting, the Havok XSI Scene Exporter can optionally detect chain effectors and parent them to the end bone of the associated chain. This is controlled by the **Reparent Chain Effectors** option. With that option *on*, the node hierarchy for the above skeleton is exported as a standard skeleton heirarchy:

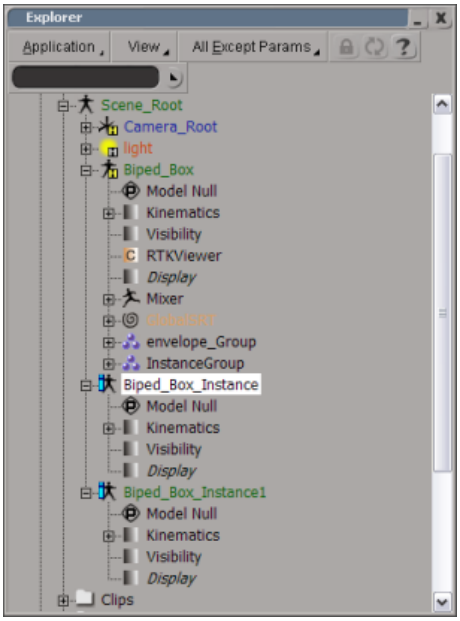


While switching the option *off* the node hierarchy exported will match that of XSI:

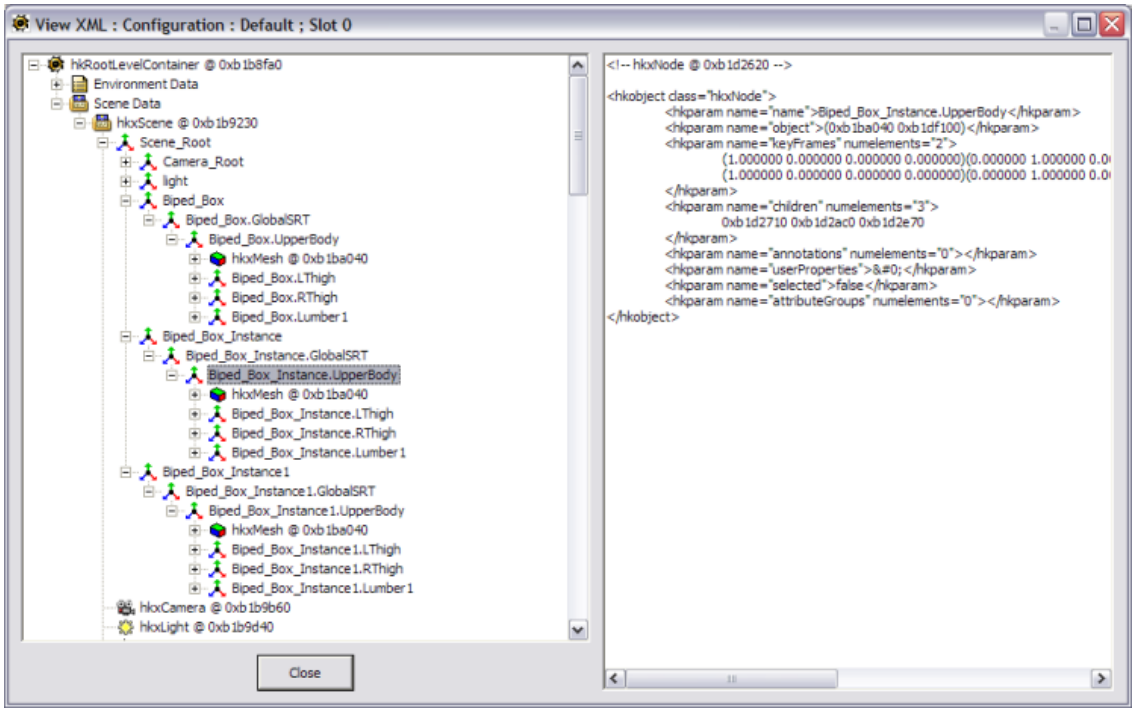


- **Clone Instanced Models**

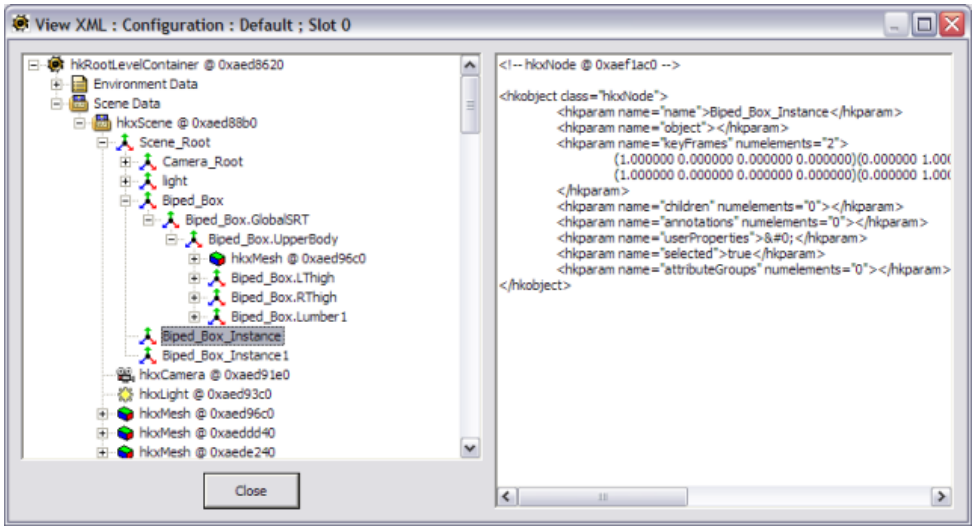
  XSI provides functionality for instancing models - it allows to replicate node hierarchies (full models) in the scene without any duplication of information - each instance has it's own root (instanced model) which is just a transform - the children of the model are implicitely duplicated (they don't appear under the instanced model).



  The Havok XSI Scene Exporter allows you to convert this implicit cloning into a more explicit one. If the **Clone Instance Models** option is selected, the node hierarchy under the original (master) model will be replicated under each of the instances in the exported scene.



If this option is off, the instanced models are exported as a single nodes with no extra children:



- **Ignore Unused Skin Bones**

  If this option is selected, the XSI Scene Exporter will perform an extra check for the bones reported by XSI as deformers of a given skin, and ignore them if they don't actually deform it (if all the associated weights are zero). This may reduce the size of the exported skin bindings and it may help create simpler rigs.

- **Do Not Split Vertices**

  By default, the XSI Scene Exporter will export as many mesh vertices as required in order to store texture and normals. So, for example, while for a cube only 8 geometric vertices would be required, in order to store per-face normal and texture coordinates, 24 vertices need to be exported.

Selecting this option will override the default behaviour, and multiple vertices that have the same position will be exported as one, regardless of their normal or UV coordinates. This may reduce the size of the exported mesh - but due to the lack of accurate normal and texture coordinates, the mesh may not display as it originally did in the modeller.

6.5.3 XSI: Physics Tools

6.5.3.1 Introduction

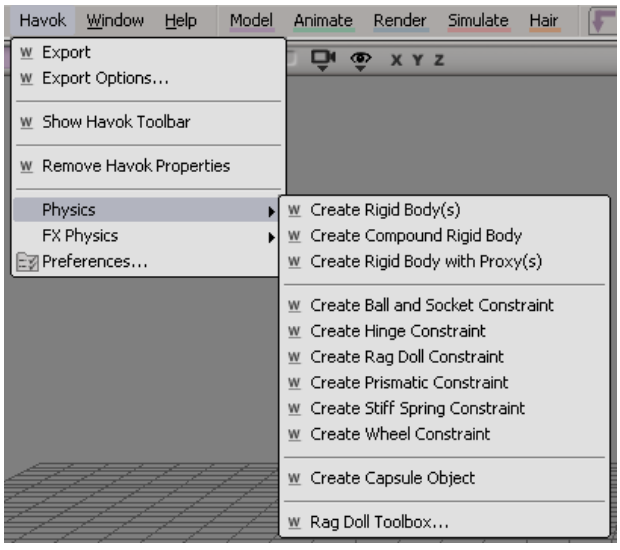
All modeling packages, including XSI, have the ability to specify scene and animation data that has a close relationship to the equivalent run-time objects in Havok (and provide many sophisticated tools to manipulate that data). However, for physical setup the tools available in the modeler are either limited or do not necessarily match the Havok run-time features. Moreover, as the construction of complex physical systems (such as rag dolls) can be a complicated task, providing tools to automate and facilitate these and another operations become important.

It is for these reasons that the Havok Content Tools include special XSI tools (parameter sets, operators, commands, views, etc) for setting up physical information in an XSI scene. In the following sections we will explore these tools.

Advanced users should first read the Common Concepts chapter for in-depth details on some of the rigid body and constraint concepts.

Accessing the Physics Tools

The physics tools for XSI can be accessed through the **Havok** > **Physics** menu or through the **Havok** toolbar:



Havok Parameter Sets

Custom parameter sets (also known as custom properties) are the main way in which additional (user) information can be attached to an object in XSI. Most of the Havok physics data in a scene is added through the use of such parameter sets, which are registered with XSI by the Havok Physics plugin.

For reference, the main parameter sets used by the Havok Physics Tools for XSI are:

- Rigid Body Parameter Set : Adds rigid body (dynamics) information to an object.
- Shape Parameter Set : Adds shape (collision detection) information to an object.
- Constraint Parameter Sets (Ball and Socket, Hinge, Rag doll) : Add constraint information to an object.

We will be examining these parameter sets closer in the following sections.

The Havok Content Tools also provide some other parameters sets, generally for storing options and some internal data. These parameter sets are recognizable by the common '`hk`' prefix.

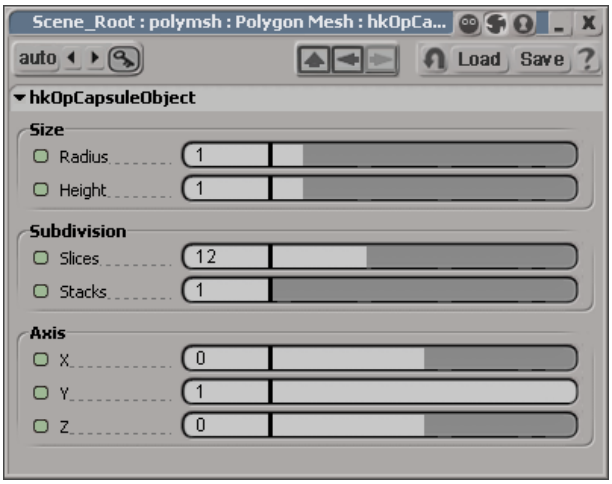
While it is possible to manually add these parameter sets to your scene (through scripting etc.), it is highly recommended that you use the Havok menu and toolbar items to do so as doing otherwise may leave your scene in an inconsistent state.

You may remove any Havok Physics properties from selected objects at any time by choosing the **Remove Havok Properties** menu option.

Havok Operators - Capsule

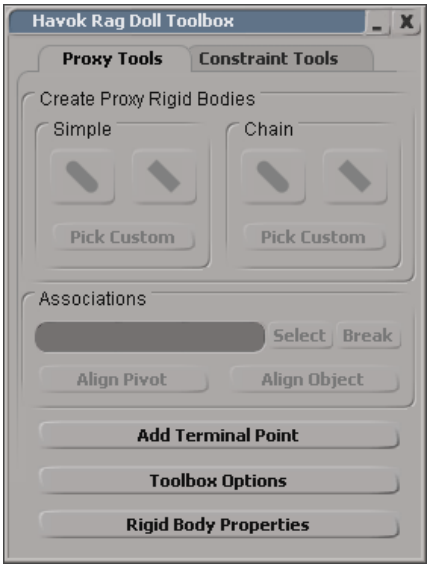
Unfortunately, XSI does not provide a capsule primitive. Capsules are a very useful primitve in Havok, so the Havok Content Tools provide an *hctOpCapsuleObject* operator. This allows the physics tools to create and modify capsule meshes easily. A polygon capsule can be created at any time by clicking on the **Create Capsule Object** menu option. The capsule operator provides several parameters to control the resulting polygon mesh:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Havok Views

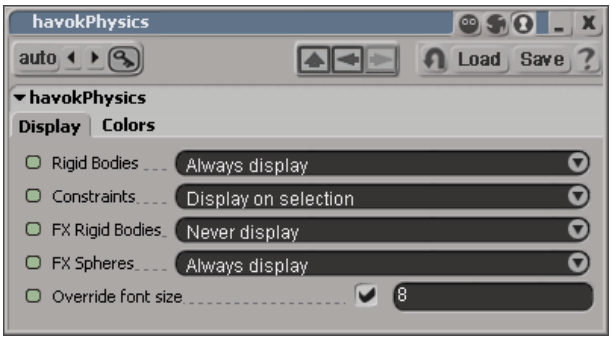
The Havok Content Tools also include the *Rag Doll Toolbox*, implemented as a custom view in XSI. This is a set of controls which are layered on top of the core physics tools. The purpose of the toolbox is to simplify the creation of rag doll representations of skeletal structures - it takes care of common tasks such a fitting rigid bodies to bones, aligning pivots, setting up heirarchies, creating constraints, etc.



Physics Preferences

Most Havok Physics parameter sets in a scene are represented in the viewports in some way. It is useful to have some control over this representation - the Havok Physics Preferences provide this control. You can open the preferences by clicking on the **Preferences...** menu option, or by locating the *havokPhysics* category in the application preferences:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



These options mainly control the display of Rigid Body and Constraint parameter sets in the viewports. These display settings are as follows:

- **Always Display** : Always show these items in the viewports.
- **Display on Selection** : Show these items whenever the parameter set or it's parent 3D object is selected.
- **Never Display** : Never show these items in the viewports.

There is also an option to specify the font size to use for Havok labels etc. in the viewports. If the *Override* option is disabled, the labels will be drawn at the same size as any other XSI text.

The second tab allows users to specifiy colors to use for various components which are drawn in the viewports, allowing you to easily adjust the display to suit your setup.

Note:
These preferences are normally saved with the application, not with the scene.
However, in some situations it may be necessary to save them with the scene - see the troubleshooting section for more.

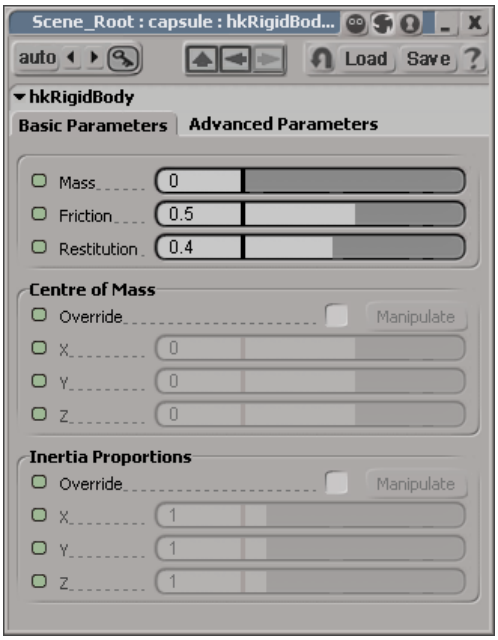
6.5.3.2 Rigid Bodies

In this section we will describe how to create and manipulate rigid bodies and shapes in XSI. For information about concepts and properties associated with rigid bodies and shapes, please check the Rigid Body Concepts section.

Rigid Body Parameter Set

A Havok rigid body parameter set defines the root of a rigid body and contains information on the dynamics of the body. Any object which contains a rigid body parameter set will be converted to a runtime rigid body by the Create Rigid Bodies filter (provided there is at least one associated shape parameter set).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Check the common Rigid Body Properties section for specific details on these parameters.

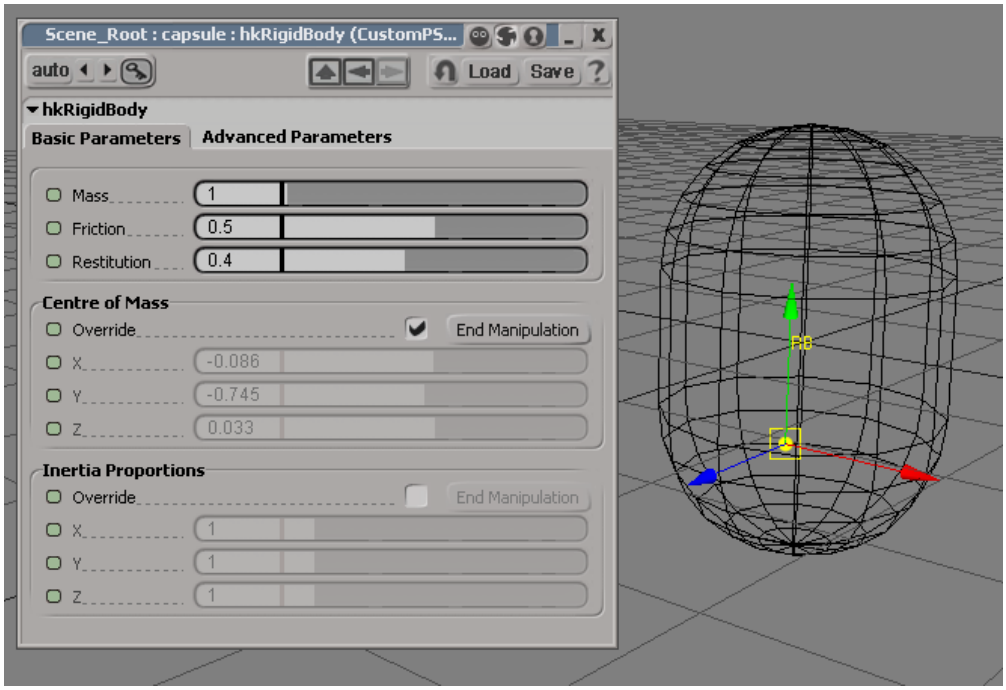
Rigid bodies are represented by '*RB*' labels in the viewport. If a rigid body has zero mass, it's label changes to '*[RB]*' to indicate that the body is fixed.

Overriding the Center of Mass and/or the Inertia Tensor

By default, the center of mass and inertia tensor of a rigid body will be automatically calculated during export, based on the rigid body geometry and mass distribution. You can, however, override these values and specify them within XSI, either by editing the parameter values directly, or by manipulating them in the viewports.

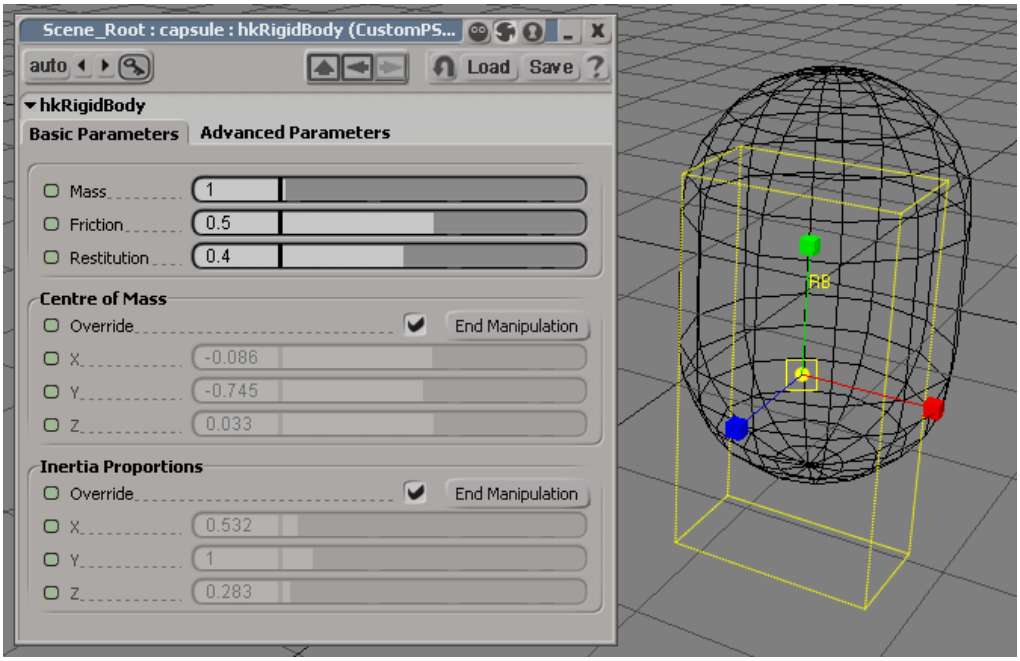
Selecting the **Override Center Of Mass** check box will display a point in the viewport, representing the center of mass of the object. To manipulate this point, click the *manipulate* button in the parameter set's property page. This sets up a temporary mechanism whereby we can drag a gizmo in the viewport to drive the parameter values:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



To exit this 'manipulation' mode, you may either press the *end manipulation* button, or change the selection in the viewport. This method of manipulating parameters is used by several of the Havok Physics tools.

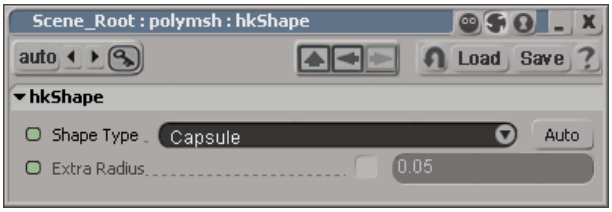
Similarly, selecting the **Override Inertia Tensor** check box will display a box in the viewport which represents the inertia distribution of the object. Again, to manipulate this tensor, you may click its *manipulate* button, which in this case provides a scaling gizmo:



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Shape Parameter Set

A Havok shape parameter set contains information about how an object's mesh will contribute to the collision detection setup of a rigid body. One or more shape parameter sets must be associated with a rigid body parameter set in order for the Create Rigid Bodies filter to produce a runtime rigid body.



Check the common Shape Properties section for specific details on these parameters.

Whenever a new shape parameter set is created by the Havok Content Tools, it's *Shape Type* parameter is automatically set based on the mesh which it applies to. However it is still useful to check each shape type parameter to ensure that it suits your requirements.

Shape parameter sets currently have no representation in the viewports.

Creating a Simple Rigid Body

Most of the rigid bodies you will ever want to create will be simple rigid bodies - where both the rigid body and shape information is associated with the same object.

You can create a simple rigid body by selecting an object in XSI and clicking on the **Create Rigid Body** menu option or toolbar button.

This will add both a rigid body parameter set and a shape parameter set to the selected object.

Creating a Compound Rigid Body

A compound rigid body is that which has more than one shape (ie. more than one collision detection primitive). In a compound rigid body one object has both rigid body and shape information while one or more of its children provide additional shape information.

You can create a compound rigid body by selecting two or more objects in XSI and clicking on the **Create Compound Rigid Body** menu option or toolbar button.

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body and shape parameter sets while each of the other objects will hold a shape parameter set. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new hierarchy.

Creating a Rigid Body with Proxy(s)

A rigid body with proxy(s) is one where the object which contains the rigid body parameter set doesn't contain any shape parameter set - the shape parameter sets are provided by one or more of the children (ie. the proxies). In that sense, a rigid body with proxy(s) is similar to a compound rigid body, the only difference being that the object which has the rigid body information has no shape information.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

You can create a rigid body with proxy(s) by selecting two or more objects in XSI and clicking on the **Create Rigid Body with Proxy(s)** menu option.

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body parameter set while the rest of the objects will become the proxies and just hold shape parameter sets. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new herarchy.

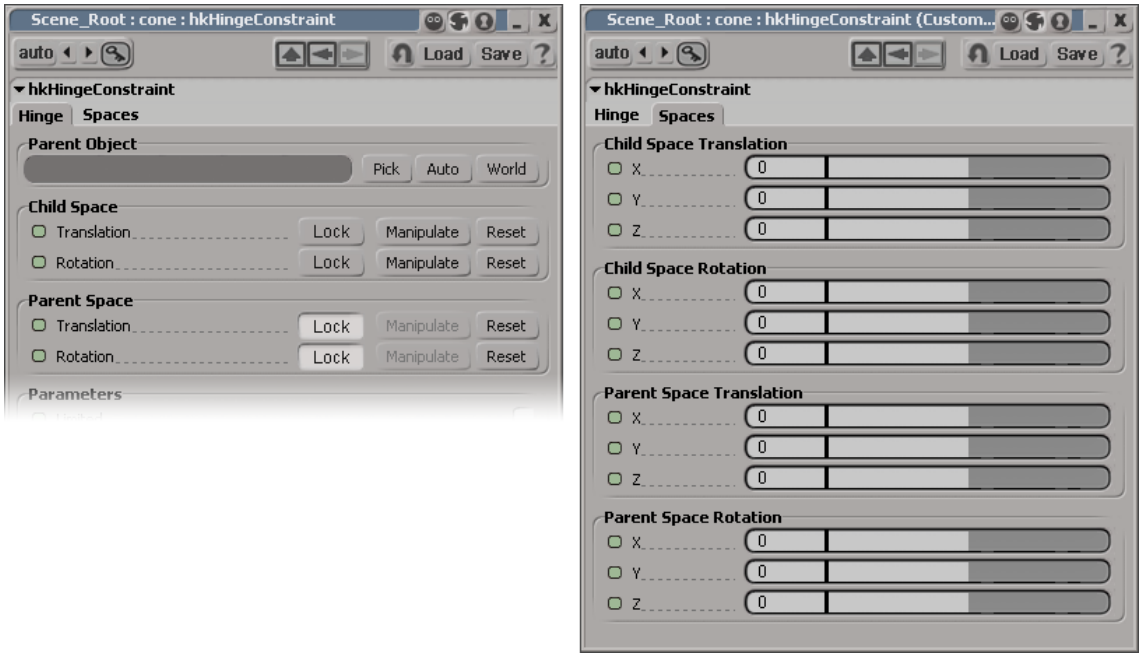
6.5.3.3 Constraints

In this section we will present how rigid body constraints are created and manipulated in XSI.

Constraint Parameter Sets

A Havok constraint parameter set contains information which will be used to create a runtime constraint on export. Since a constraint must act on a rigid body, a constraint parameter set is useful only if the parent object also has a rigid body parameter set. If a pair of rigid bodies are to be constrained, a constraint parameter set is attached to only one of the bodies (the *child*), and it maintains a reference to the other body (the *parent*). Again, please consult the common Constraint Concepts section of this manual for more details.

There are currently three types of Havok constraint parameter set - the specifics of which are described in the common Constraint Types section of this manual. Each of these types provides the same basic constraint parameters, which specify the child and parent spaces:



Each constraint type generally provides additional parameters which are relevant to that type only. Each type also represents itself in the viewport in a way which reflects the type and its parameter settings.

'Parent Object' controls

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

These controls allow you to choose which other rigid body object (if any) is the constraint attached to.

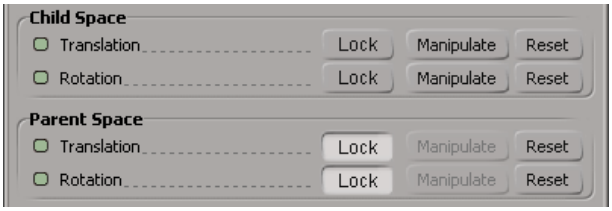
- **Pick** : Starts a picking session which allows you to specify any other object in the scene as the parent object. The picked object must contain a rigid body parameter set.
- *Auto*: Examines this object's hierarchical parent - if it contains a rigid body parameter set, then it will be set as the parent object. This is usually the desired setup.
- **World** : Clears the parent object. The constraint will attach the rigid body to a fixed location in space.

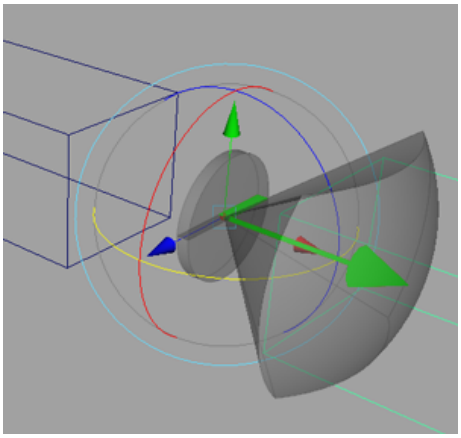
'Constraint Space' controls

Constraints can be considered as having two distinct spaces: the *child* (in the local space) and the *parent* (in the parent object's space). See the common Constraint Spaces section for more details.

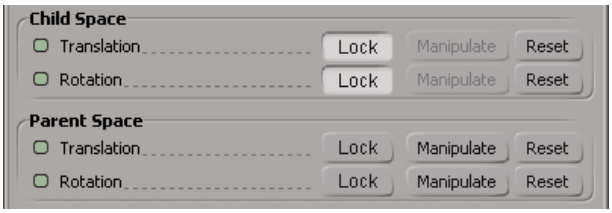
The individual [X,Y,Z] components of each space are editable via the controls in the 'spaces' tab. However it is often easier and more intuitive to manipulate the spaces in the viewport using the various *manipulate* buttons. Once the selection is changed or another button is pressed in the property panel, the manipulation mode will end.

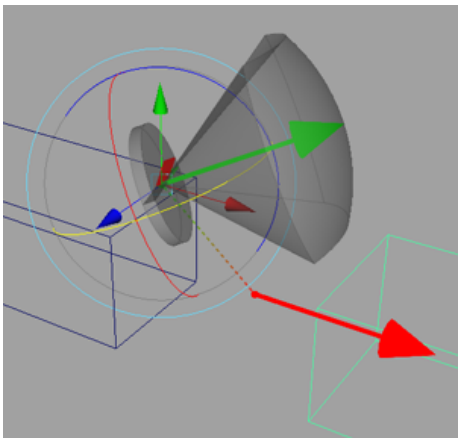
The state of the **Lock** buttons determine which spaces may be manipulated, and in the case of the parent space also determine how the spaces are interpreted. If the parent space is locked in both translation and rotation (the default setting) then the constraint space behaves as a single entity - both the child and parent spaces move together:





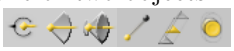
If the parent space is unlocked in either translation and rotation then the constraint parent space can be manipulated independantly of the child space:





In all cases, the **Reset** buttons return a constraint space parameter to its initial setting, effectively moving the space to the local origin of the relevant object.

Creating a Constraint

You can create a constraint by selecting either one or two objects in XSI and using the **Create <xxx> Constraint** menu options or toolbar buttons. 

Each selected object must already have a rigid body parameter set attached. Based on the number of selected objects, the constraint is created as follows:

- If one object is selected, the constraint parameter set will be attached to that object. If its parent object is also a rigid body then that object will be set as the constraint parent object, otherwise no parent object will be specified.
- If two objects are selected, and one is a descendant of the other, the constraint parameter set will be attached to the descendant and the ancestor will be set as the constraint parent. If the two objects are not explicitely parented, the constraint parameter set will be attached to the first object and the second will be set as the constraint parent.

A single rigid body object may have any number of constraints attached to it.

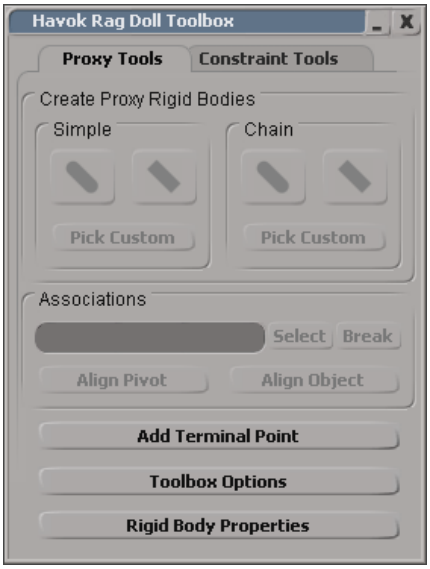
6.5.3.4 The Rag Doll Toolbox

Introduction

One of the most common applications of rigid body simulation is the creation of articulated systems of rigid bodies representing skeletal characters. This usually involves the creation and alignment of rigid

body proxies for the character's bones, as well as the setup and alignment of constraints which represent the character's joints. The *Havok Rag Doll Toolbox* provides a set of tools to facilitate these operations. It is built as a custom view in XSI that works on top of the lower-level physics tools.

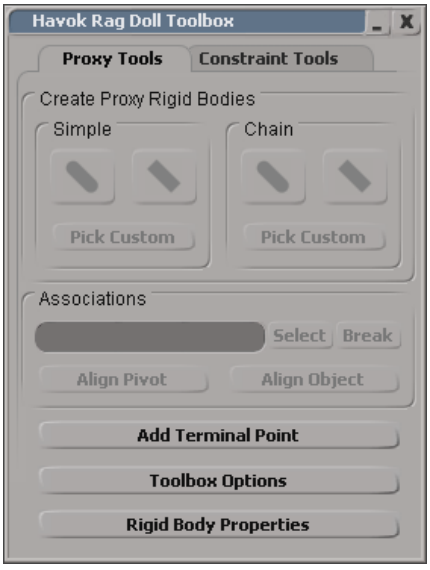
You can open the Rag Doll Toolbox using either the **Rag Doll Toolbox...** menu option or toolbar button.



The toolbox appears as a floating window with two tabs: the *Proxy Tools* are used to create and associate rigid bodies with bones; the *Constraint Tools* are used to organize and constrain these rigid bodies together:

The toolbox may also be opened in any viewport by selecting it from the ' **custom display host** ' submenu of the viewport's menu.

Proxy Tools



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

The first task when creating a rag doll representation of a character is to construct a set of rigid bodies to represent a selection of the character's bones. These rigid bodies are usually simple collision primitives such as capsules and boxes, which are very efficient to simulate by the physics runtime (capsules in particular).

The Proxy Tools provide the following functionality:

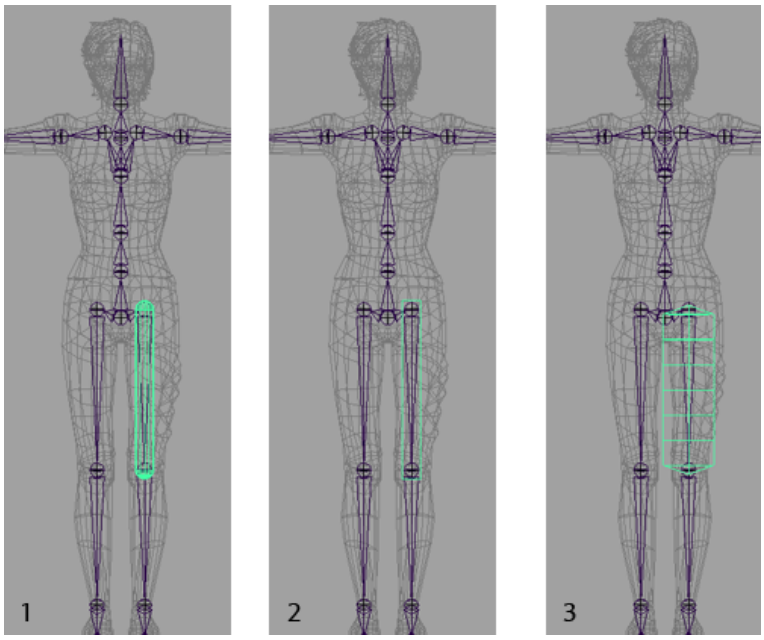
- Given a bone, create a proxy representation (a box, a capsule, or a custom mesh) and align and associate the proxy with the original bone.
- Given a chain of bones, create a single proxy representation of that chain and align and associate that proxy with the chain of bones.
- Given a proxy rigid body (associated with a bone or chain of bones), replace it with another proxy.
- When creating proxies, it is very important that their pivots are aligned to those of the bones they represent, since, at runtime, movements on the bones will be converted to movement on the rigid bodies and vice versa. The proxy tools track this association and provide the ability to realign the proxy pivot/object at any time.
- Quickly access and modify the parameters of one or multiple rigid bodies.

Create/Replace Proxy Rigid Bodies

These buttons allow you to create proxies and associate them with bones, and are enabled based on the current selection in XSI. Note that the Toolbox Options contains several settings which apply to proxy creation.

The first group of buttons create *simple proxies* - those which are associated with a single bone.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



1. **Create Capsule Proxy(s)**

   With one or more bones selected, this will create a capsule rigid body for each selected bone. It will also associate and align each capsule with each bone.

2. **Create Box Proxy(s)**

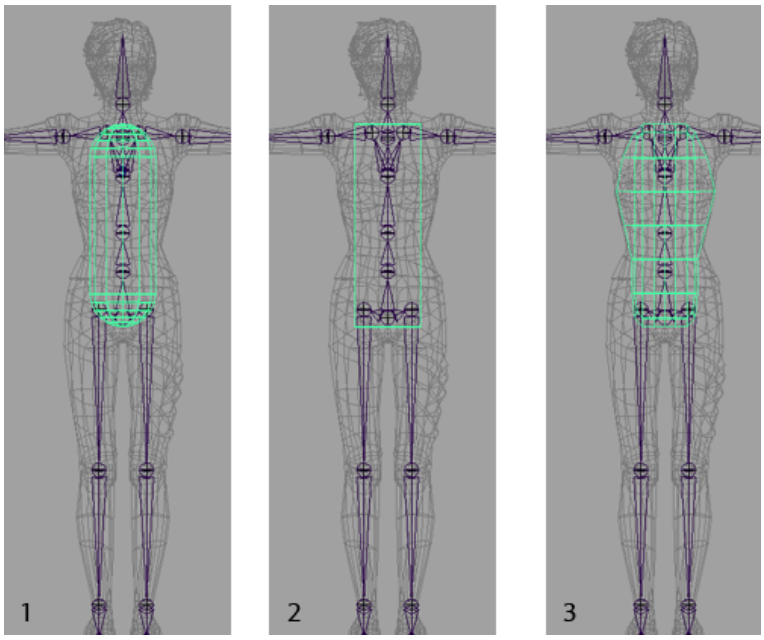
   With one or more bones selected, this will create a box rigid body for each selected bone. It will also associate and align each box with each bone.

3. **Pick Custom**

   With a single bone selected, this will start a picking session where you can select any mesh object in the scene to become the proxy rigid body for the selected bone. It will then associate and align that rigid body with the selected bone.

The second group of buttons create what we call *chain proxies* - those which are associated to a set of bones in hierarchical order. Chain proxies are useful in order to create single rigid bodies to represent multiple bones, such as a rigid body bounding multiple spine links.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



1. **Create Capsule Proxy(s)**

   With a chain of bones selected, this will create a capsule rigid body bounding all of them. It will also associate and align the capsule with the chain of bones.

2. **Create Box Proxy(s)**

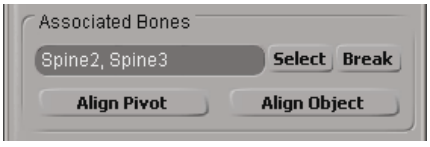
   With a chain of bones selected, this will create a box rigid body bounding all of them. It will also associate and align the box with the chain of bones.

3. **Pick Custom**

   With a chain of bones selected, this will start a picking session where you can select any mesh object in the scene to become the proxy rigid body for the chain of bones. It will then associate and align that rigid body with the chain.

Whenever a rigid body proxy (either single or chain proxy) is selected, these *Create Proxy* buttons become *Replace Proxy* buttons - the proxy they create will replace the selected proxy, and the association with the original bone will be updated accordingly.

Associations



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

These controls are used to display and edit the associations between selected rigid body proxies and bones.They are enabled whenever one or more rigid body proxies (rigid bodies created by one if the " **Create Proxy** " options above) are selected, or one or more associated bones are selected.

If a set of associated proxies are selected, the edit box displays the associated bones. If a set of associated bones are selected, the edit box displays the associated proxies. The **Select** button will change the current selection to that displayed in the edit box.

The **Break** button will remove the associations that are stored with the selected proxys/bones. The proxies will remain as rigid bodies in the scene, but their association with the original bones will be lost.

The **Align Pivots** button will ensure that the pivots of the each selected proxy match that the pivots of the associated bones (or the first bone of the associated chain), without transforming the proxy mesh. Use this button whenever you have modified a proxy's transform and want to ensure that the pivot is aligned correctly:

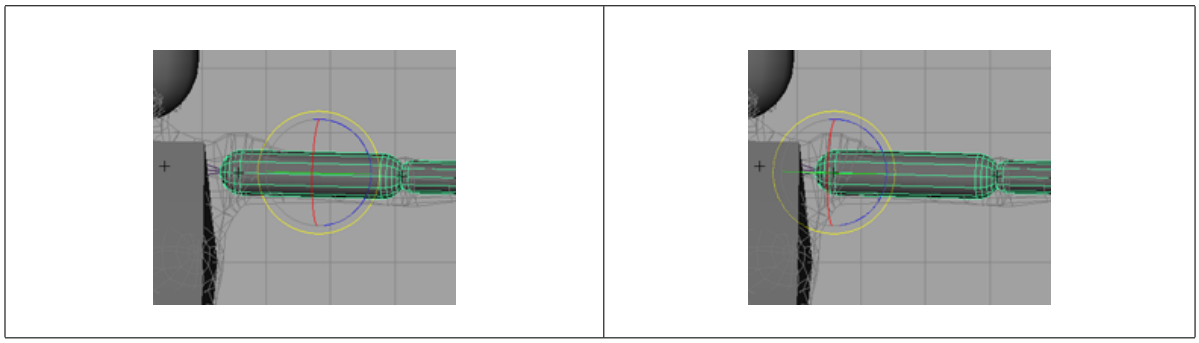


Table 6.5: Aligning Pivots

The **Align Objects** button will ensure that the pivot of each selected proxy matches the pivot of the associated bone by moving and/or rotating the proxy. Use this button whenever you move or rotate a bone and want to update the associated proxy accordingly:

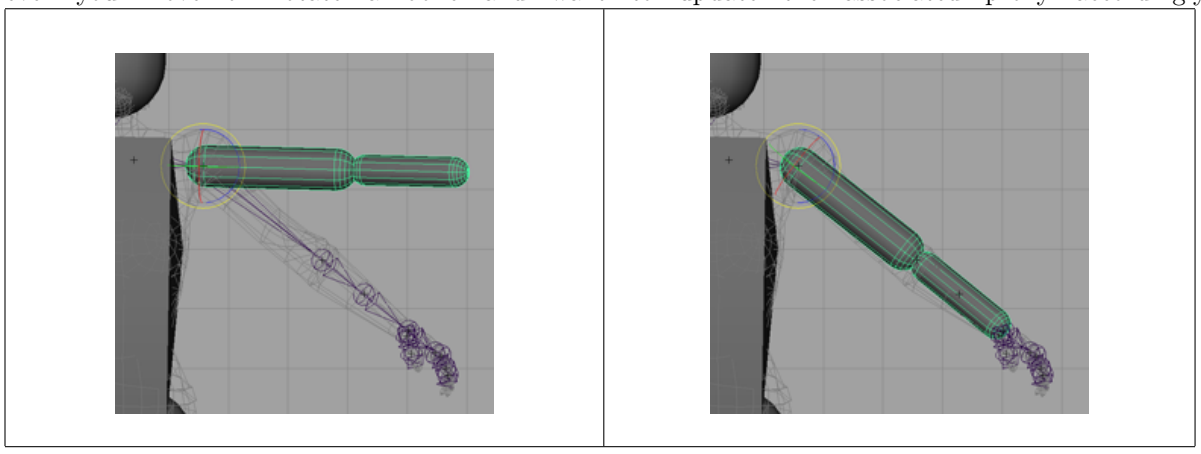


Table 6.6: Aligning Objects

Rigid Body Properties

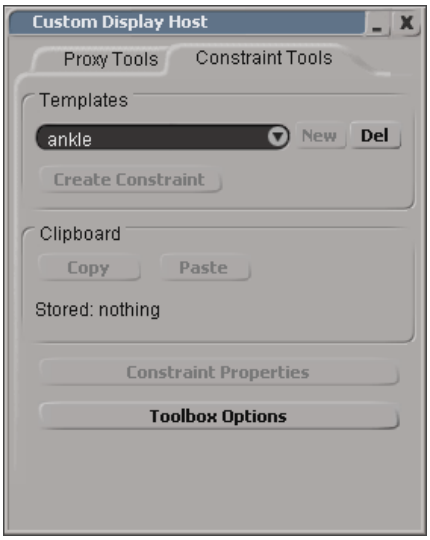
The **Rigid Body Properties** button allows quick access to the rigid body parameters of a set of selected rigid body objects. This is particularly useful when you want to modify multiple rigid bodies at the same time. Simply select a set of proxies and click this button to open a multi-edit property panel.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Add Terminal Point

The **Add Terminal Point** button is used to add extra leaf bones to skeletons to be used for raycasting, specifically to prevent ragdoll penetration with landscapes using the hkaDetectRagdollPenetration utility. Selecting a ragdoll proxy and clicking the add terminal point button creates a null transform as the child of the proxy. The null is positioned on the local bounding box of the proxy at the center of the face furthest from the proxy pivot.

Constraint Tools

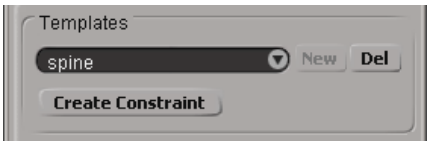


Once the rigid body proxies have all been created, the next step in setting up a rag doll is to organize them into a constrained hierarchy to match that of the original skeleton.

The Constraint Tools provide the following functionality:

- Given a pair of proxies, reparent them in a way which reflects the original skeleton, and apply a constraint between them.
- Save and reuse constraint setups between characters, as *template* files.
- Easily replace a constraint with another templated version.
- Quickly copy and paste constraints within the same scene.
- Quickly access and modify the parameters of a constraint.

Templates



These buttons allow you apply any of a set of saved constraint setups (templates), and allows you to create new templates or delete existing ones. The dropdown list show all the available templates. This list is built by looking at all templates stored in the current *templates folder*, the location of which can be changed from the Toolbox Options.

The **New** button is enabled whenever a constraint parameter set (or its parent 3D object) is selected. It allows you to save the parameters of that constraint as a new template file (templates are saved as `*.txt` files). The **Del** button allows you to permanently remove the selected template (deleting it from the templates folder).

The **Create/Replace Constraint** button is that which does the main work. This button is enabled only when two proxys are selected. When a constraint is created/replaced, the following steps take place:

1. Ensure that hierarchy of the selected proxies is correct, by looking at the hierarchy of the associated bones. If not correct, it reparents the proxies accordingly.
2. Create/replace a constraint parameter set on the 'child' object - the one which is lower down in the hierarchy. The type of constraint which should be created is stored in the template file.
3. Set the constraint's parent object as the one higher up in the hierarchy.
4. Set the constraint's parameter values based on those stored in the template.

Note:

When working with templates, some assumptions need to be made regarding the local axis of the bones. In particular, the main axis (an axis pointing towards the length of the bone, in the "outside" direction) and the bend axis (an axis around which a positive rotation (using the right hand rule) causes a bending of the joint) need to be defined. These axes are currently assumed to be +X and Z, since this is the convention used by XSI's default skeletons. The Havok Content Tools default templates were created using this converntion. If a proxy's transform does not follow this convention, then any applied templates will need to be rotated to the correct orientation.

Clipboard

While working on a symmetric character's constraints, it is often desirable to replicate constraints on each side of the character. The clipboard provides a quick way to create a temporary template which can then be applied throughout the rest of the character.

The **Copy** button is enabled only when a rigid body with a single constraint is selected. It will create a template of that constraint in memory, and display the stored constraint type for reference.

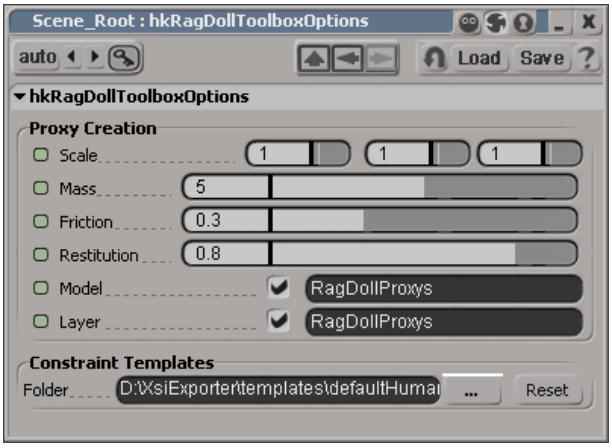
The **Paste** button operates in the same way the as the **Create/ Replace Constraint** button, with the difference being that it applies the template stored in memory rather than the template selected in the drop down list.

Constraint Properties

The **Constraint Properties** button allows quick access to a constraint's parameters. Simply select a constrained rag doll proxy to enable this button. When the button is clicked, a property panel will open for the relevant constraint parameter set.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Toolbox Options



The toolbox options are stored as a custom paramter set in the scene, and are accessible via the *Toolbox Options* button on each tab of the toolbox.

Whenever new proxies are created, they are automatically sized to fit the bones they are associated with. However, bones are often much narrower than the skin volume and so the resulting proxies are generally not wide enough. The *Scale* options the ability to set a scaling factor to be used during the creation of new capsule/box rigid body proxies. Once a bounding box has been created in the local space of a bone (or a chain of bones) this additional scale is applied to the bounding box before a new proxy is created to fit the bounding box.

The *Mass*, *Friction*, and *Restitution* values are those which are applied to any newly created proxy, and are self-explanatory.

The *Model* and *Layer* options determine how new proxies are placed in the scene. By default any new proxies are grouped under an XSI 'model', and placed in a unique layer. This makes it easy to hide/lock/etc the proxies as a whole without affecting the rest of your scene.

6.5.3.5 The Convex Hull Tool

In this section we describe a simple tool for generation of convex hulls from selected meshes. This tool generates an approximate convex hull from the currently selected objects (either meshes or nurbs) with a user-specified maximum number of vertices **Max verts** . The hull is approximate in the sense that it does not necessarily enclose the selected meshes, but is a reasonably good approximation (using the metric of volume difference) which satisfies the constraint on the maximum number of vertices. This is useful in a number of contexts, such as generation of assets for platforms which require <64 vertices per convex body. If the specified maximum exceeds the number of vertices in the true hull, the tool just generates the true hull. Note that if multiple meshes/nurbs are selected, the hull tool generates a single hull from all of their vertices combined.

The approximate hull is found by first computing the original set of vertices in the true hull. If the number of vertices in the original hull is N, and the desired maximum number of vertices in the generated hull is M, the tool has to remove R=(N-M) vertices. In the case when the number of steps equals 1, a trial is done which consists of randomly choosing a sample of R of the true hull's vertices, and computing the hull obtained by removing that sample from the original set. A user-specified number **Samples per**

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

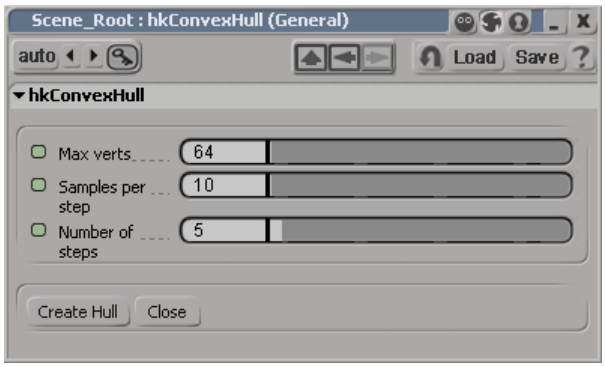
step of such trials are done(with a default of 10), and the trial hull obtained which had the minimum difference in volume from the true hull is taken as the final choice. This constitutes one 'step'.

Rather than removing all R vertices in a single step, the user can specify that the the vertices be removed in a sequence of X steps, where X is the parameter **Number of steps** . On each step roughly R/X vertices are removed (if X exceeds R, each step removes only one vertex). Increasing the number of steps will clearly lead to hulls which better approximate the true hull, but will take longer to finish. The default is 5 steps.

The convex hull tool utility simply consists of fields for the three parameters **Max verts** , **Samples per step** , and **Number of steps** described above, and a **Create Hull** button to generate the hull using these parameters. Generated hulls are placed in the scene root with the name "foo_hull" where "foo" is the name of the first object in the selection. Subsequently generated hulls accumulate in the scene root but are given unique names.

The convex hull tool window remains open after hull generation, so that different parameters can be tried by simply undoing after clicking Create Hull and changing the parameters. Convex hull parameters are kept temporarily in an hkConvexHull property in the scene root. This is deleted on clicking the **Close** button in the property window.

On clicking Create Hull, a progress bar appears which allows the hull generation to be aborted if it is taking too long. On aborting, the best hull at the current point in the computation (i.e. the hull from the current step with minimal volume difference so far) is generated.



6.5.4 XSI: Animation Tools

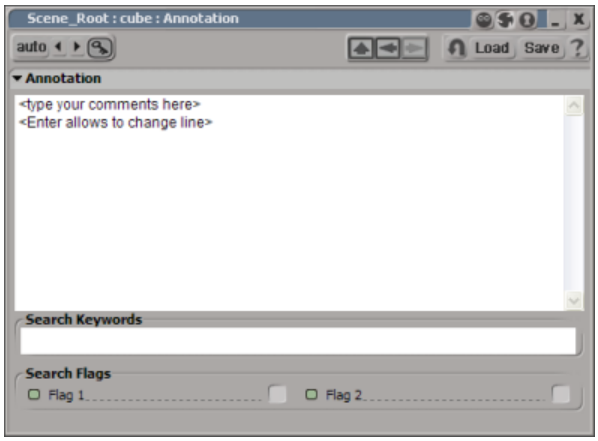
Since XSI already provides a wealth of tools for creating content for animation playback, the Havok Content Tools use these features already provided by XSI without the need of providing custom extensions (as it does for the physics). Thus concepts like skeletons, bones, skins and animations in XSI transfer to the same concepts/objects in the Havok SDK. The following section outlines one of the less obvious connections between XSI and the Havok Animation SDK.

6.5.4.1 Annotations in XSI

Please check the common Annotations section for details on annotations in the Havok Animation SDK.

XSI supports the concept of annotations as simple "extra information" associated with any object. An

XSI annotation consists of:



- An annotation text (non animatable).
- Search keywords (non animatable).
- Two boolean flags (animatable).

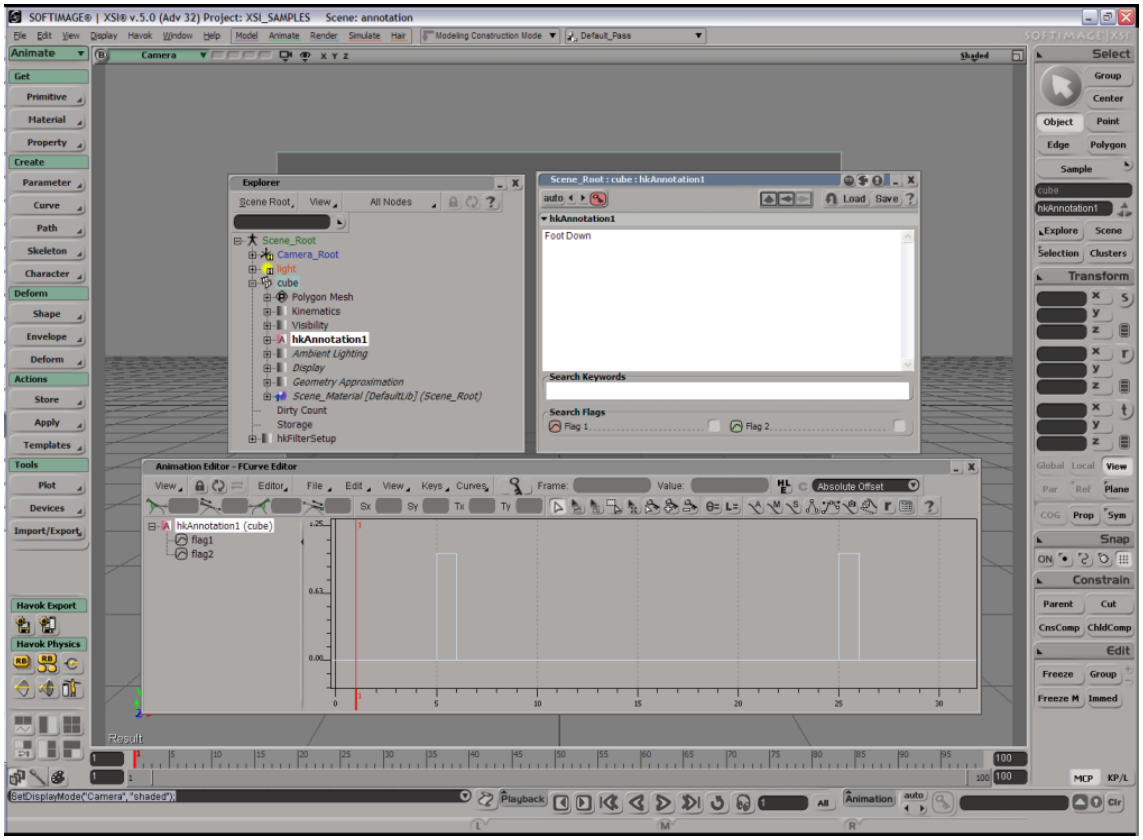
Notice that XSI annotation objects are associated to objects, but not to a particular time in the animation, while a Havok Annotation is associated both to a node and a time. In order to bridge the gap between the two types, the following logic is applied when creating Havok Annotations from XSI annotations:

- The XSI annotation has to be associated with a node (3D object) in the scene
  - This is because Havok annotations can only be associated to nodes
- The XSI annotation's name has to start with "*hkAnnotation*" (for example, "*hkAnnotation1*")
  - This is so the XSI Scene Exporter can decide which annotations are meant to be transformed into Havok Annotations
- The animation of the **flag1** boolean parameter is used to determine at which time(s) are the annotations added. Each time this flag transitions from *false* to *true*, a Havok annotation will be added.
- The text of the Havok annotation is taken from the text of the XSI annotation. If this text is empty, the name of the XSI annotation is used instead.

Example

The screenshot below show an XSI annotation created so it exports as a Havok annotation.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

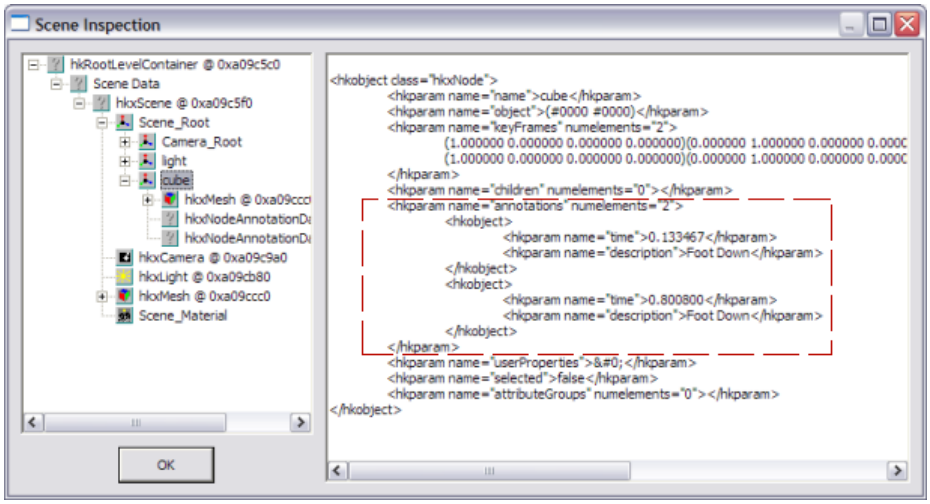


Notice the following:

- The XSI annotation is associated to the *cube* node
- The XSI annotation is named "*hkAnnotation1*"
- The **flag1** parameter is animated so it transitions to *true* at frame 5 and 25
- The animation text is set to "*Foot Down*"

Since this XSI annotation matches the criteria we described above, it will be exported as a Havok annotation. The following screenshot shows a section of the scene data exported by the XSI exporter:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

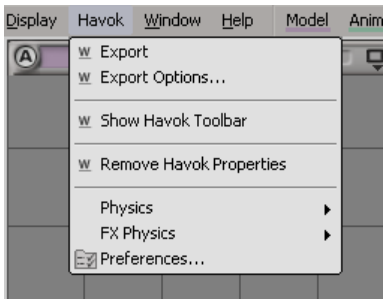


6.5.5 Tutorial: Export and Animation Basics

In this tutorial we are going to explore the basic operations involved in exporting and processing an asset. We are going to use the example of exporting skeletons, animations, etc.. but the principles regarding how assets are exported, the filter pipeline and the use of the filter manager and individual filters for processing applies to all Havok products.

6.5.5.1 Getting Started

Ensure that you have installed the Havok Content Tools for XSI. You should see a **Havok** menu in the XSI menu bar:

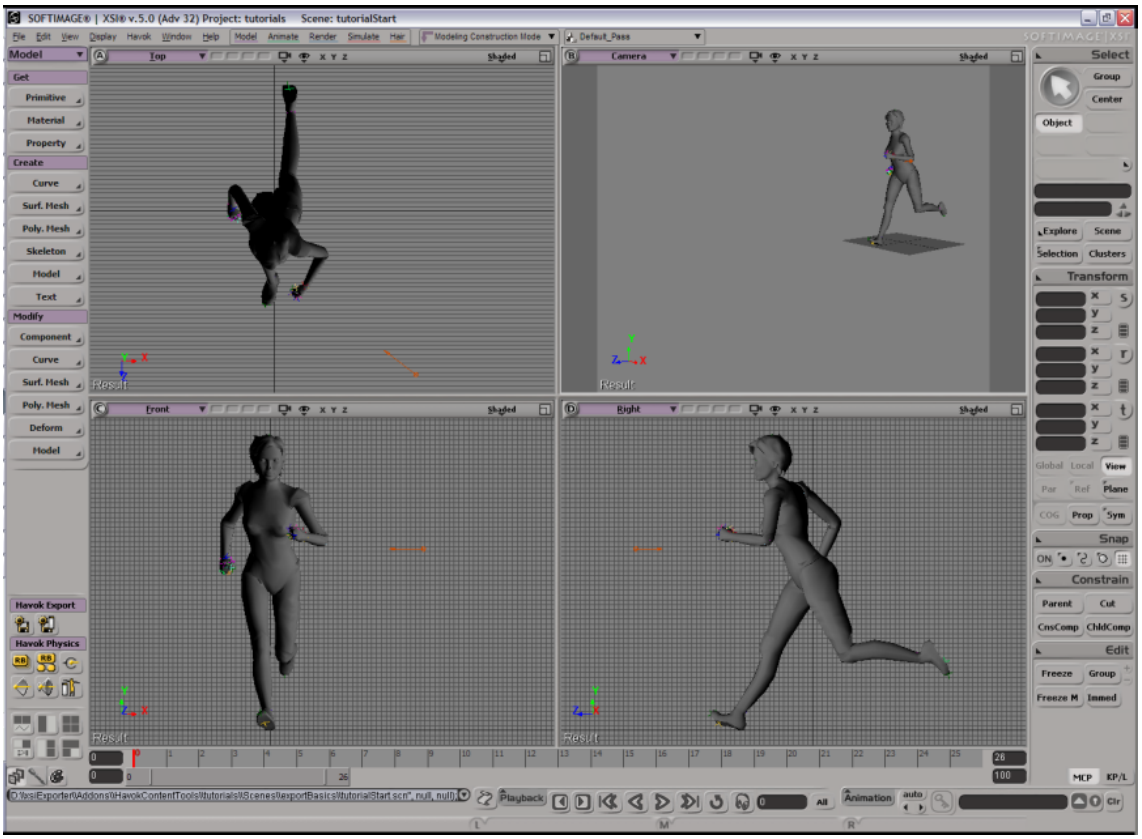


The **Havok** toolbar will be useful for this and the next tutorials, so it is useful to have it in the UI. You can switch to the Havok layout (**View** > **Layout** > **Havok** menu option) or open the toolbar as a floating window (**View** > **Toolbars** > **Havok** menu option)

Open the scene "`exportBasics/TutorialStart.scn`", in the *tutorials* project (which should be installed in `[INSTALLDIR]/Addons/HavokContentTools/tutorials`).

Tip:

If you haven't added this project to your project list we recommend you do so in order to quickly access the tutorial files.



The scene contains the animation of a girl running (a single cycle). If you examine the scene, you will notice that the animation is constructed by using a set of animated bones which modify a skinned mesh.

We want to export this information (bones, animation, skin weights, etc) from XSI into objects that the Havok Animation/Complete SDK can interpret.

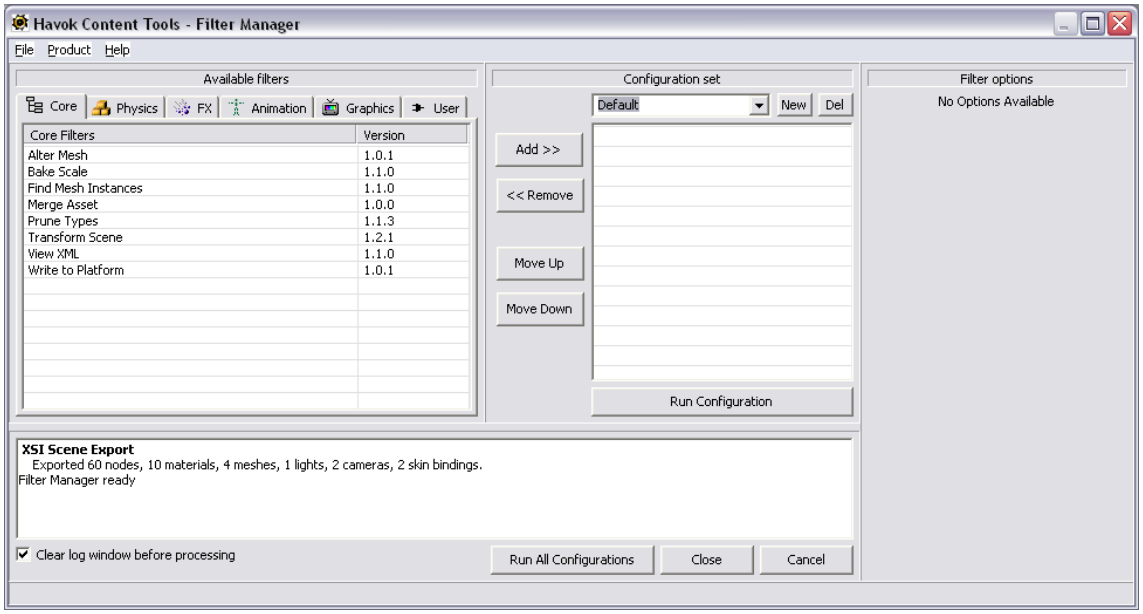
Note:

Even if you are not a user of Havok Animation or Havok Complete, you can still follow this tutorial as the Havok Content Tools packages contain filters for all products. This tutorial focuses on introducting the basic concepts behind asset export and processing so it is relevant for all Havok products.

6.5.5.2 Exporting the Scene

Exporting the current scene is as simple as to click on the **Export** button in the **Havok** menu or toolbar.

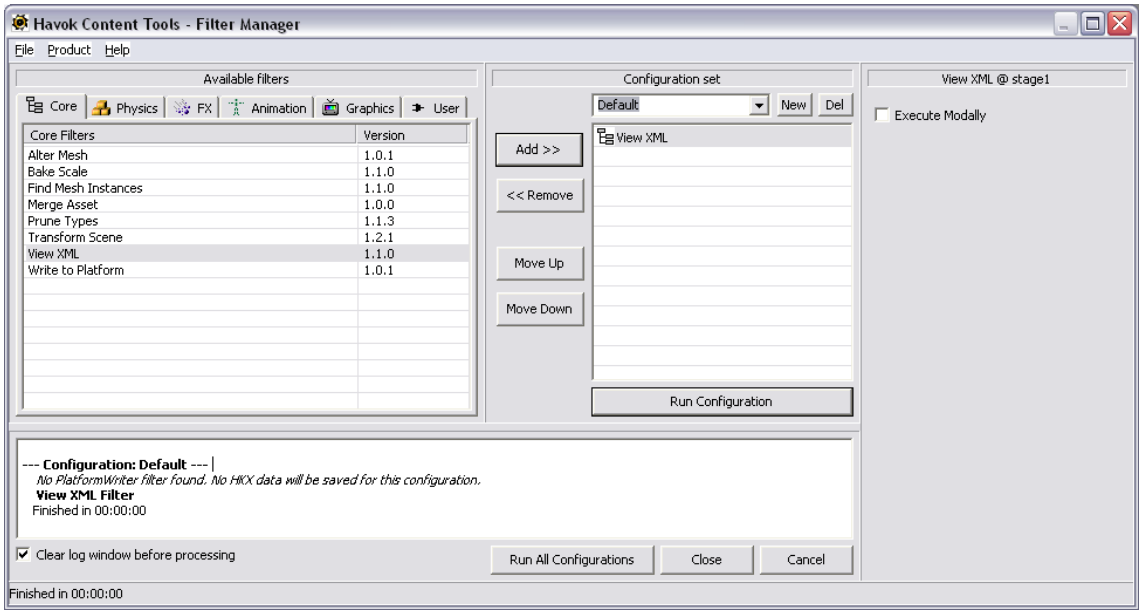
By doing this the XSI Scene Exporter is invoked - the scene is navigated and relevant information is pulled from the XSI nodes, properties, meshes, etc.. and converted into a modeler-independant format. After that, that content is passed to the filter manager; it's UI should appear after a short time:



The **Filter Order** window (in the middle of the window) lists the filters that will be applied to the content we just exported. It is now empty, which means that nothing will happen to the content. You can add filters by selecting them from the **Available Filters** tabs and clicking on the **Add**>> button.

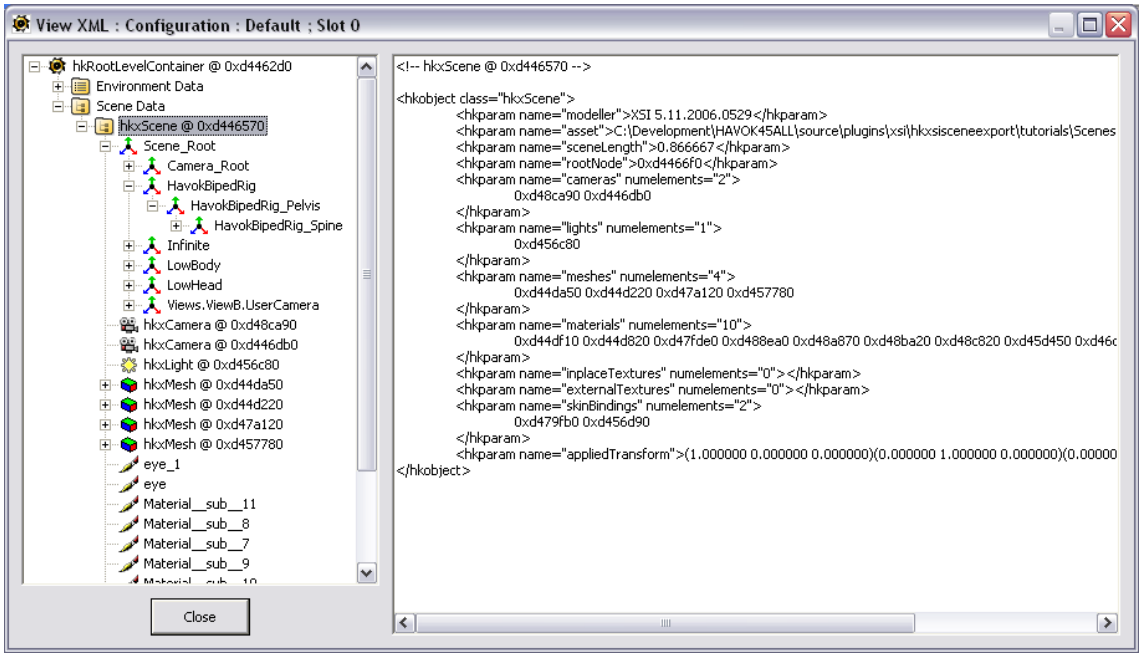
Let's start by adding a very useful filter for debugging purposes: the View XML filter.

- On the **Available Filters** window, click on the **Core** tab
- Double-click on the **ViewXml** filter (or select it and press **Add** > >)
- That filter should now appear at the top of the **Current Filters** list



Filters are executed in the order they appear in the **Current Filters** list; if a filter modifies the content, that modified content is what is used by the next filter (check the filter pipeline documentation for details).

The View XML filter doesn't modify any content, though. It just opens a modal window that presents the current content in a human-readable XML format. Let's check what we exported from XSI by clicking on the **Run Configuration** button.



Explore the contents of the exported data. Under the root level object, the main object that was exported is an `hkxScene` object, which contains information about nodes, meshes, materials and skin bindings.

This is what the XSI Scene Exporter extracted from our current scene, and is the raw information that we are going to process further until we can create a useful set of objects ready to be used by our runtime.

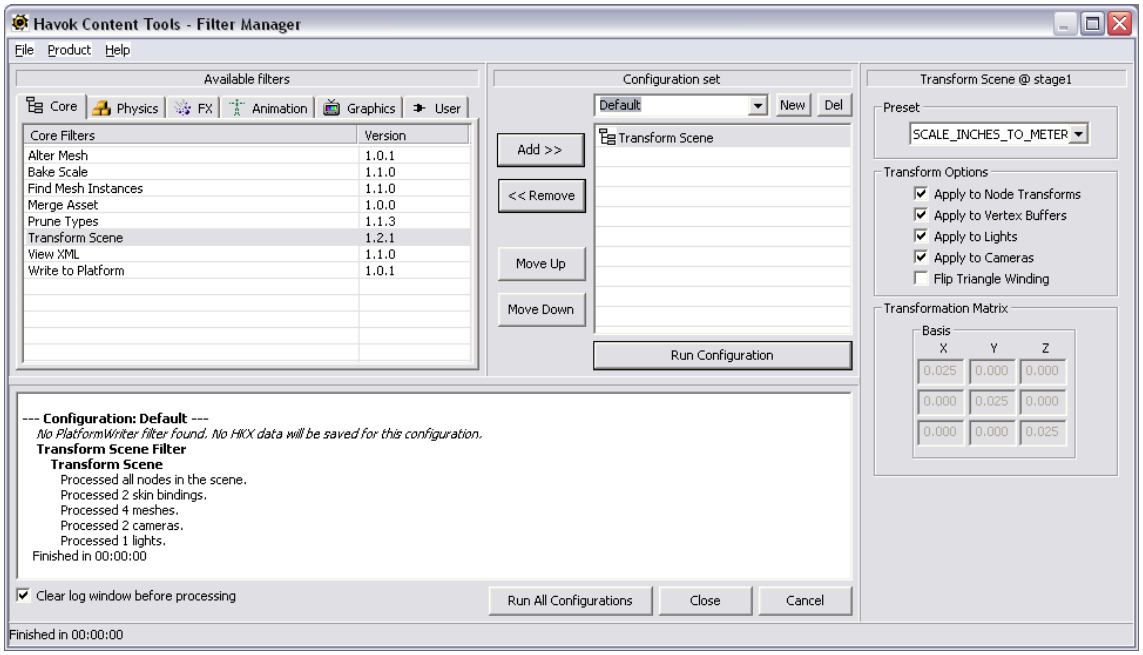
Close the XML window and get back to the filter manager. Remove the ViewXml filter by selecting it and clicking on << **Remove** or hitting **DEL** .

Tip:

Feel free to add the View XML filter at different locations during this and other tutorials in order to examine how processing is done by different filters. This filter is a very useful debugging tool.

6.5.5.3 Processing the Scene

Let's now add some useful filters for processing the scene. The first filter that we'll add will be the Scene Transform filter (also from the **Core** tab):



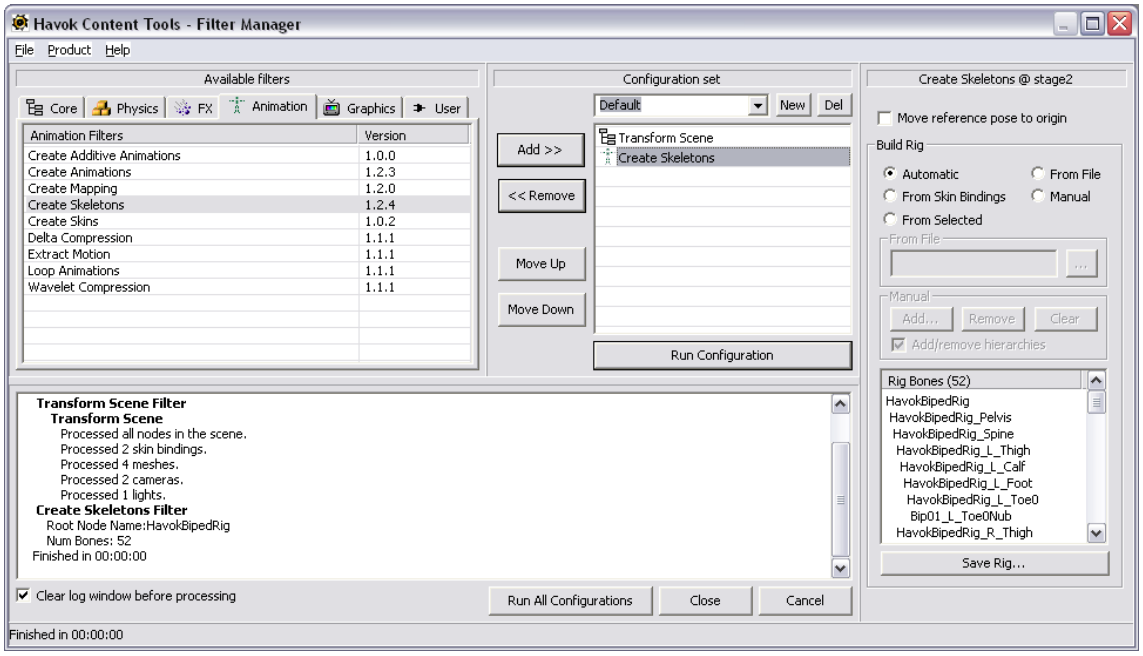
This filter is very useful and, in many cases, it will be the first filter that you'll add to any filter configuration. This filter applies an arbitrary transform to all the objects (nodes, meshes, etc) in the scene. Our scene has been modelled in inches. Assuming that our run-time is modelled in meters, we use the **SCALE_INCHES_TO_METERS** preset in the options of the filter (the right panel in the filter manager) to scale the scene.

The next filter we will apply is the Create Skeleton filter (from the **Animation** category). This filter takes a look at the nodes in the scene and creates an hkaSkeleton object (used by the Havok Animation/Complete SDK).

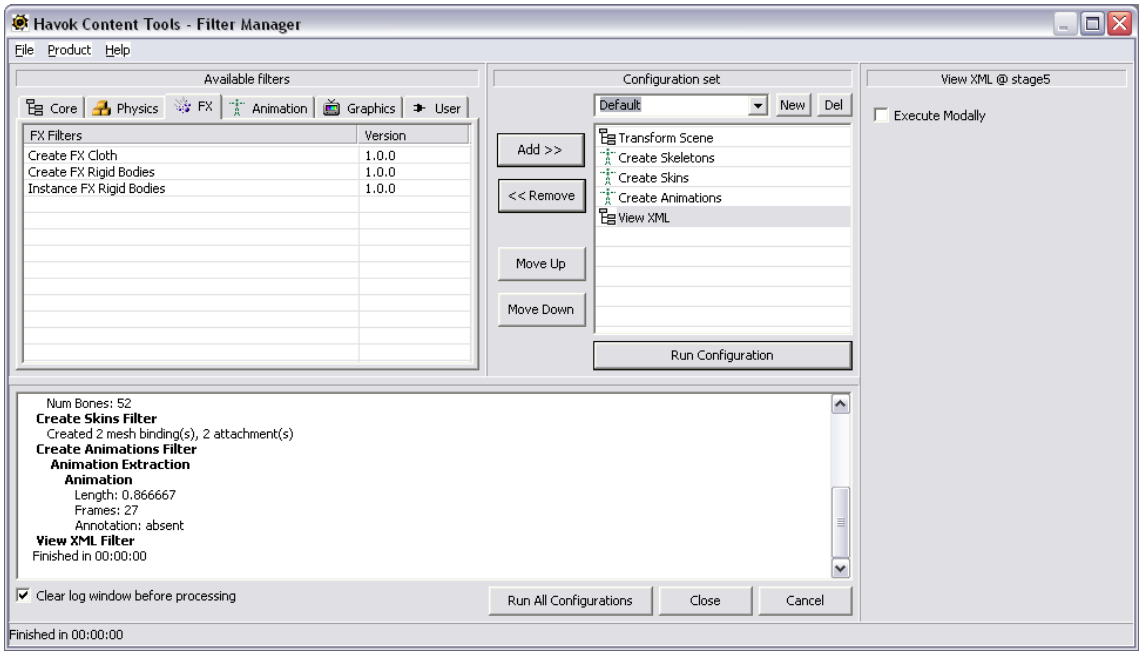
Note:

If your current Product Selection is set to Havok Physics, some of the filters will appear in red, and when run, warnings will be generated. You can either ignore those warnings, or change your product selection temporarily to Havok Complete.

As you can see, this filter also has options associated with it (they appear in the right panel).

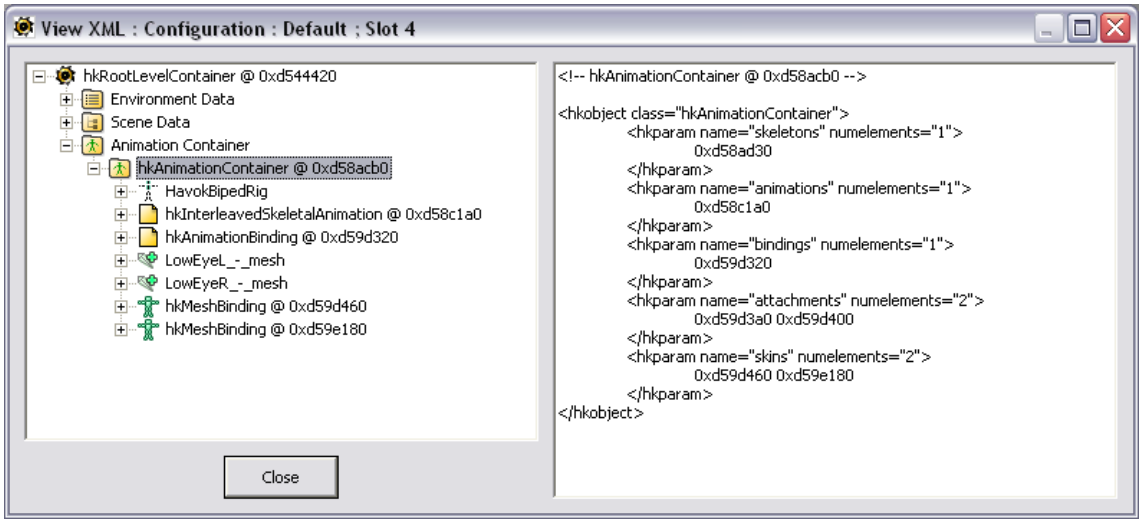


We will leave the default options as they suit us fine. You can check the documentation for this and other filters if you want to learn more about how they work. After this filter, add the Create Skin and Create Animations filters (also in the **Animation** category) - leave the default options as well. Finally, place the View XML filter again at the bottom of the filter list.



If you run (press the **Run Configuration** button) the filter processing, the View XML filter will show you the effect of all the processing:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



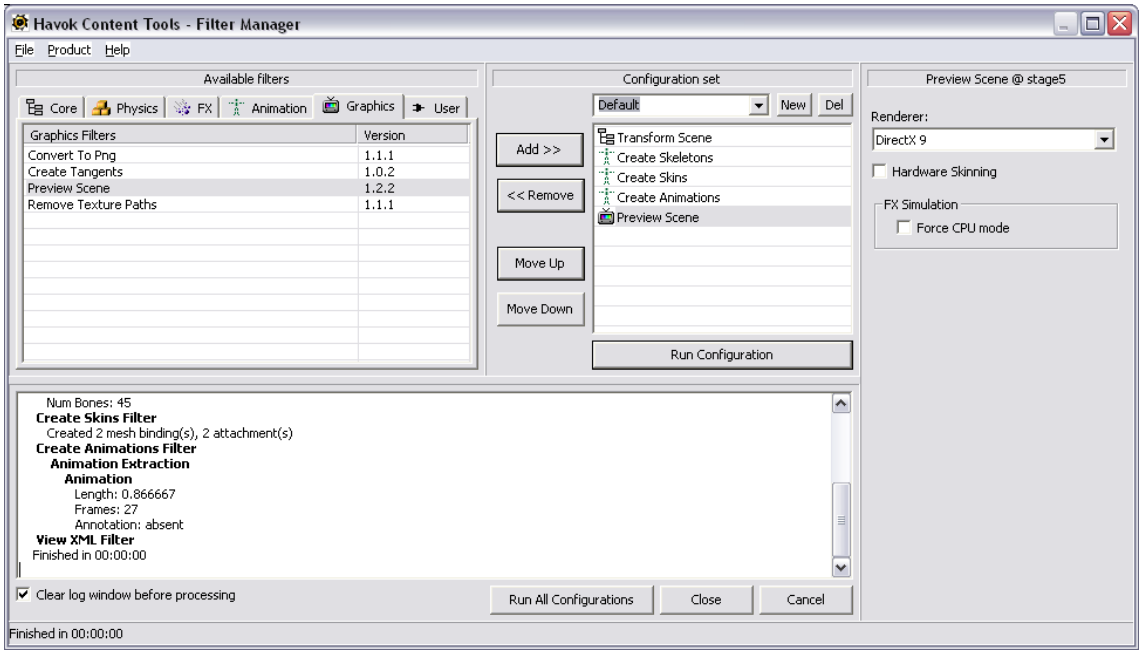
Notice how a new root-level object, an animation container, is now part of the content, and how it contains Havok Animation/Complete objects like `hkaInterleavedSkeletalAnimaton`, `hkaSkeleton` (HavokBipedRig), `hkaMeshBinding`, `hkaBoneAttachment`, etc.. that we should be able to use at run-time.

6.5.5.4 Previewing the Scene

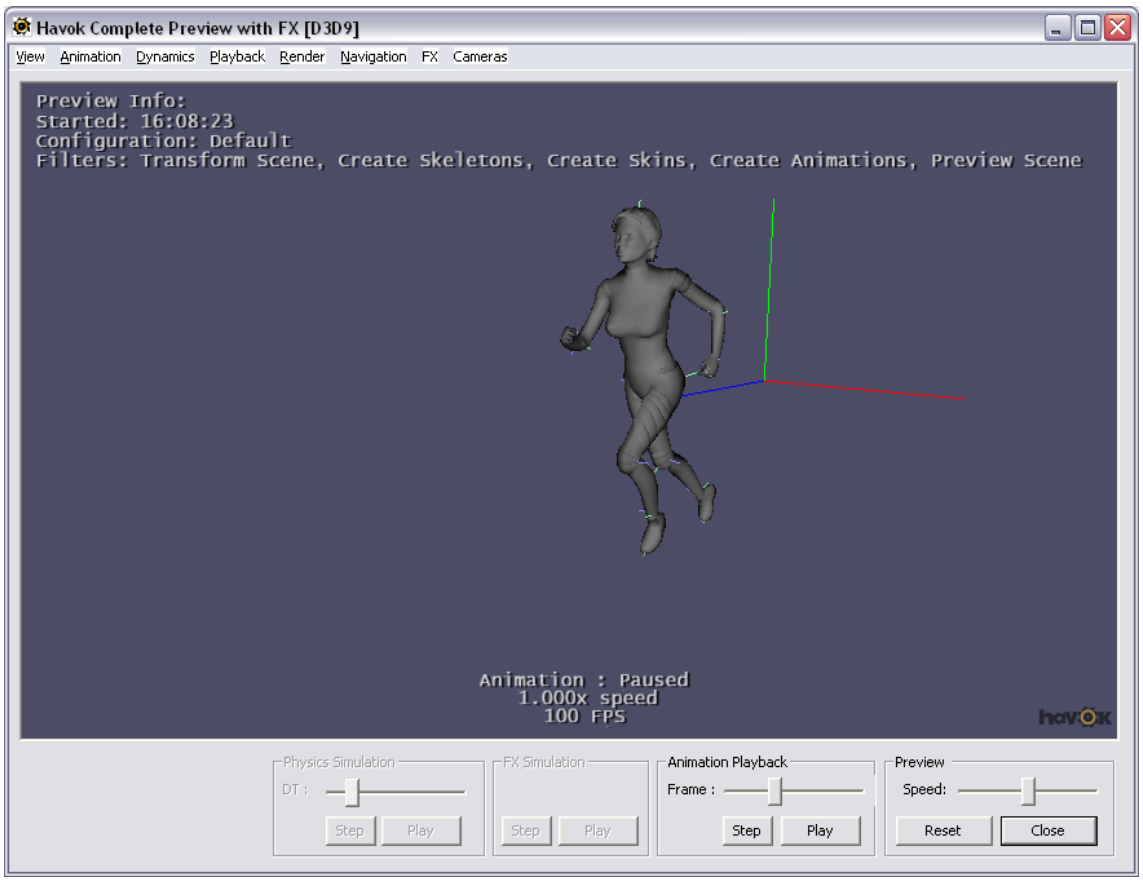
Although examining the contents by viewing the XML output give us an indication on how our processing went, in most cases visual inspection is the only way to ensure that the processed content matches our expectations.

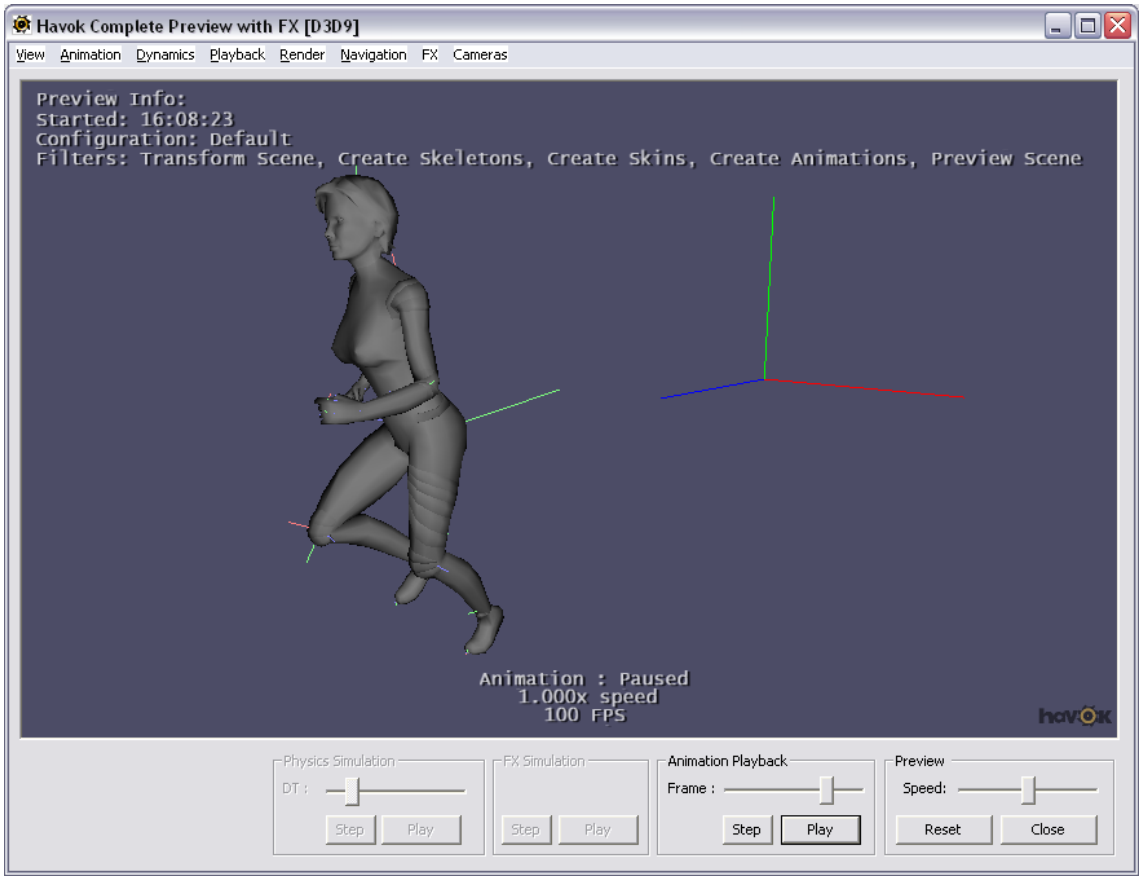
Havok provides the Preview Scene filter (in the **Graphics** category) just for that - this filter is built with the Havok run-time, including some display libraries, and will play back any Havok content it finds.

Replace the View XML filter at the end of the filter list with a Preview Scene filter, and click again on **Run Configuration** :



A modeless window will appear with our animation and skinning running in real-time:



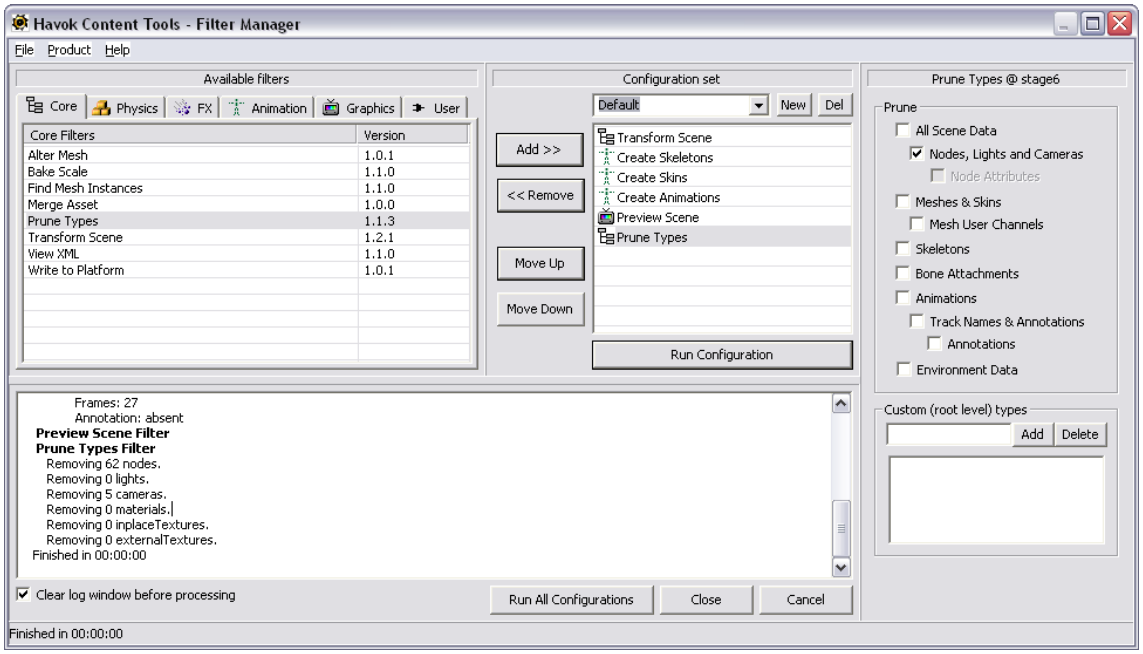


The Preview window gives you multiple options to change the speed of playback, the current frame in the animation, etc.. Check the Preview Scene filter documentation for more details.

6.5.5.5 Pruning Some Data

Most of the filters either modify (eg. the Scene Transform filter) or add (eg. the Create Skeleton filter) information to the asset. Sometimes it is useful to remove some of the data in the assets that is no longer relevant (particularly at the end of the processing). For example, once we've created our skeletons and animations, the original data regarding nodes and their keyframes is probably no longer needed (since it has been transformed into other classes).

We can add a Prune Types filter (in the **Core** category) after the Preview Scene filter. Leave the default options as they are - this will remove nodes (and their keyframes), lights and camera information:



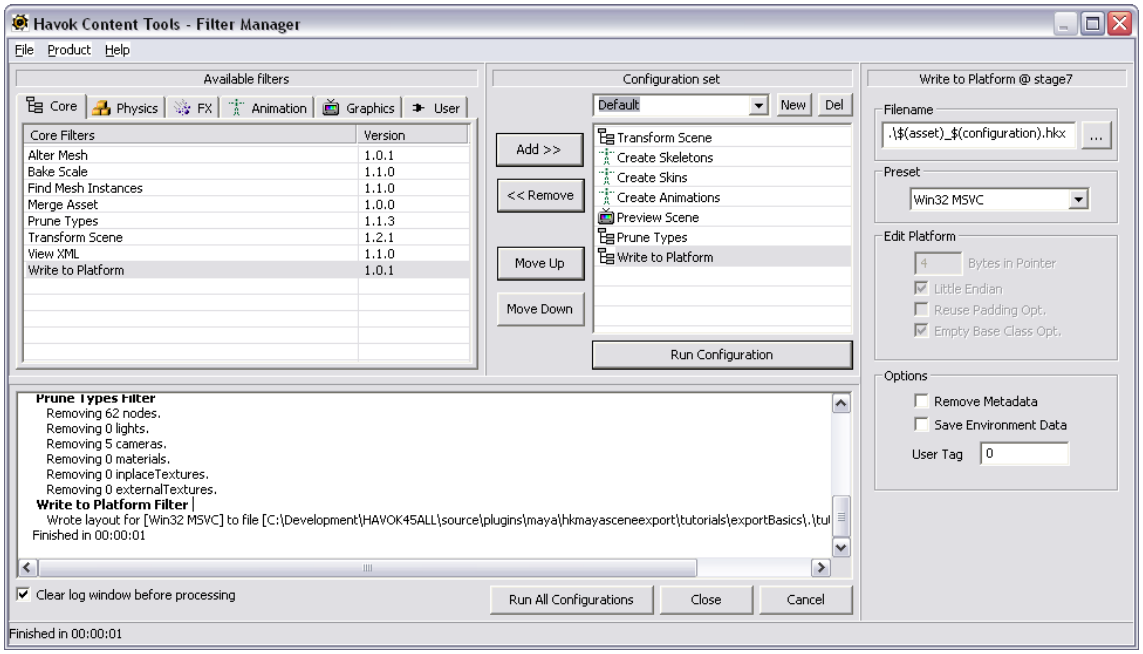
Note:

You could also add this filter before the Preview Scene filter - but since the Prune Types filter will remove camera information, the preview will have to use a default camera (rather than reflect the camera in the original scene).

6.5.5.6 Writing the Processed Contents

So far, all the processing done by the filters has happened in memory, and hasn't been written to any file.

The Platform Writer filter (in the **Core** category) will serialize the processed content to a file. It can write either XML format or binary format for many different platforms. Add the Platform Writer filter at the end of the filter configuration. By default it will save a binary .hkx file for the Win32 platform in the same location as the asset. If you now process the asset again (click on **Run Configuration**):



Observe the log at the bottom of the Filter Manager window. This reports that the file has been written alongside our XSI scene. This file is ready to be loaded into the Havok SDK.

This finalizes this first tutorial: you can find the final XSI scene in the `"exportBasics/tutorialEnd.scn"` file.

We'd like to encourage you to experiment with different filters and filter options. By checking the results in the filter processing log, as well as the output of the View XML filter, you should be able to get a good idea of what each filter does.

If you are a Havok Animation/Complete user, the following sections go a little further in processing this scene by doing some motion extraction. If you are a Havok Physics user, skip the following section and go to the Physics Basics Tutorial.

6.5.5.7 Appendix (Animation/Complete Only): Extracting Motion

The animation we just exported has some *motion* on it - in between the first and last frame of the animation there has been some displacement on the character (it has moved forward).

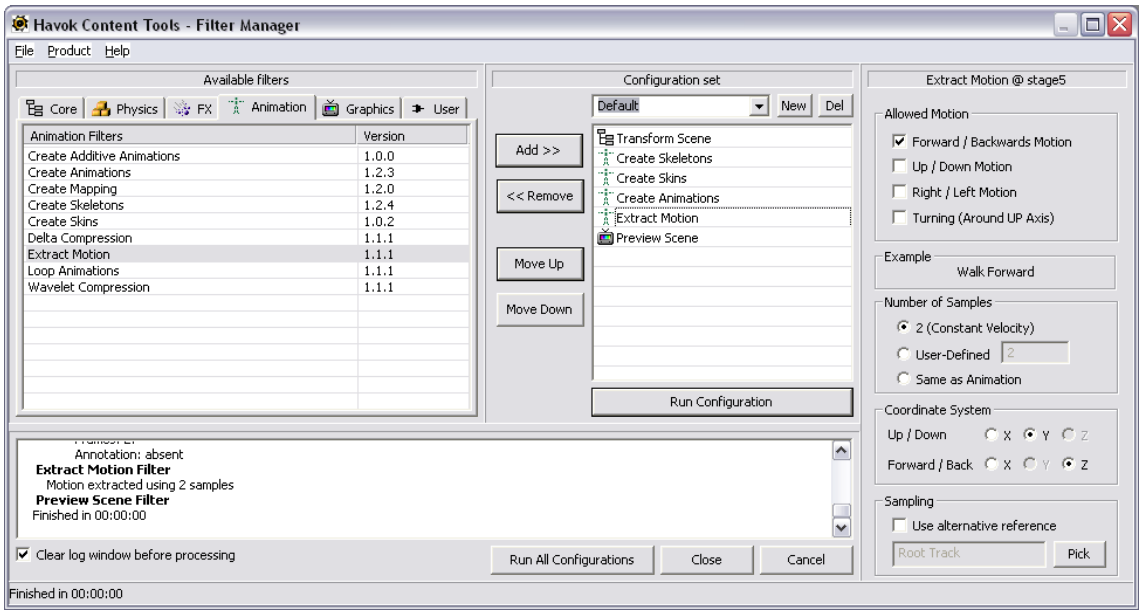
Motion extraction, explained in detail in the Animation chapter in this manual, is the process on which we examine an animation and divide it into two - a motion/displacement component (the moving forward during running) and a local animation component (the movement of arms, legs and body during the run cycle).

After motion extraction, thus, our running animation should become a run-on-the-spot animation, complemented by some information on how the character should displace when that animation is applied (the extracted motion).

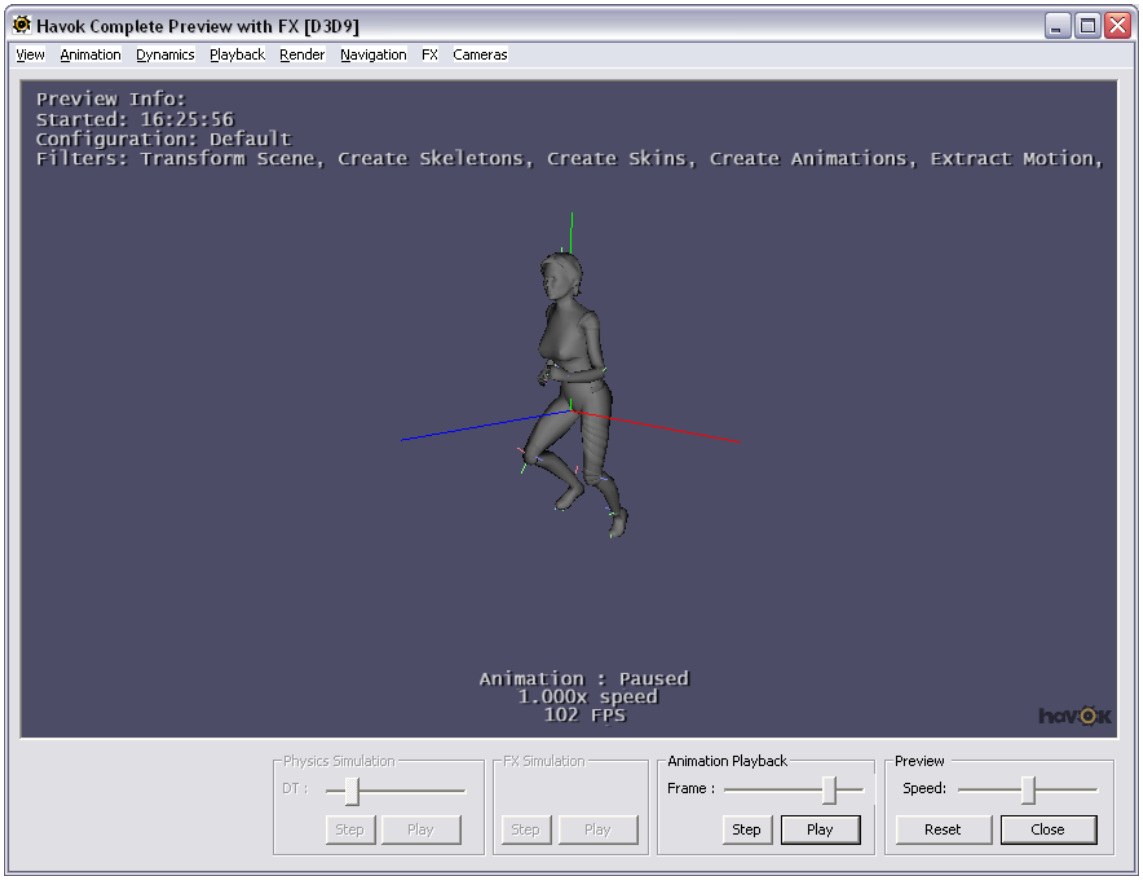
In this appendix to the tutorial we are going to do motion extraction to our run animation.

- Open the filter manager again (if you have closed it) by clicking on **Export**
- Add the Extract Motion filter (in the **Animation** category) to the filter list, just after the Create Animations filter (use the **Move Up** and **Move Down** buttons to place the filter in the right place).

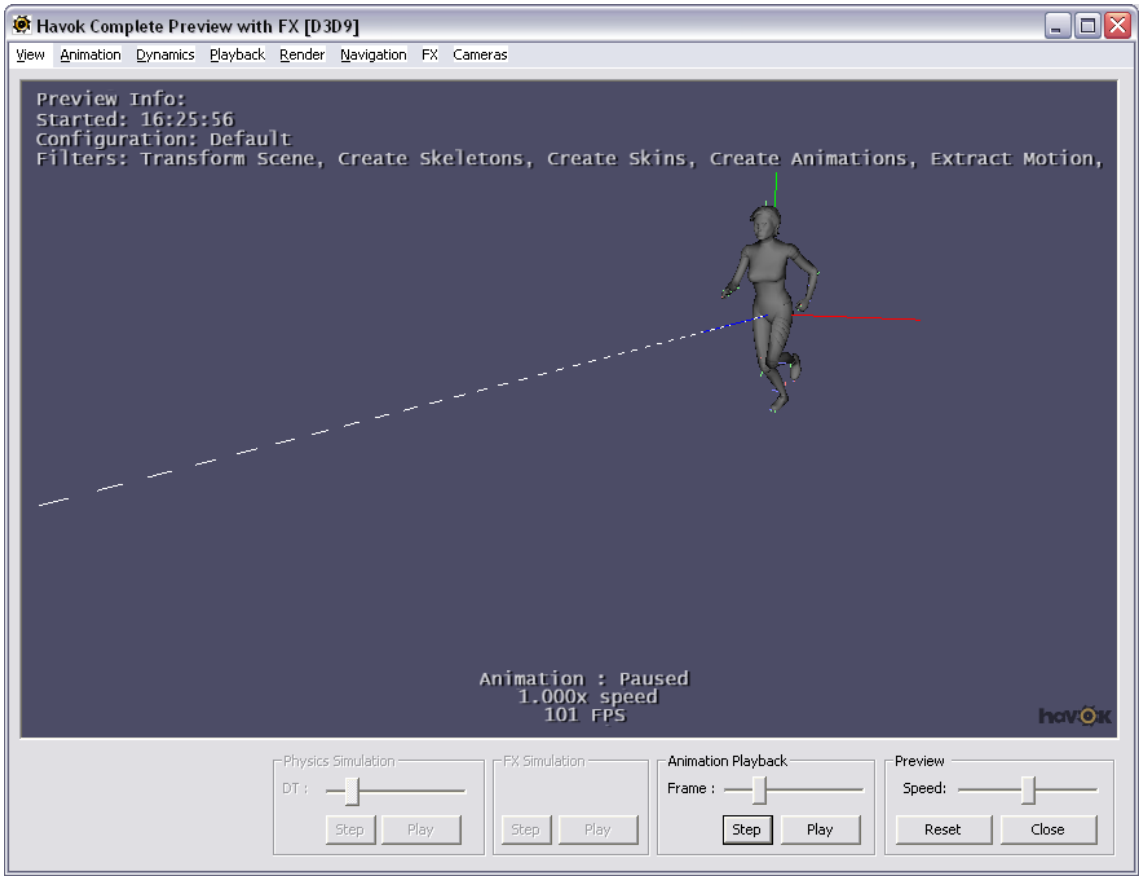
The filter options allow you to specify what kind of motion you want to extract from the animation. In our case the motion is a forward motion so we'll leave the **Forward/Backwards** check box on. But, since in our animation the up direction is Y (and not Z), change the **Up/Down** radio button in the **Coordinate System** box to **Y** . Similarly, since our character is moving forward alongisde the Z axis, change the **Forward/Back** radio button to **Z** (instead of X)



If we now run our filters, it will open the Preview dialog - you can see how now our animation is now a run-on-the-spot animation:

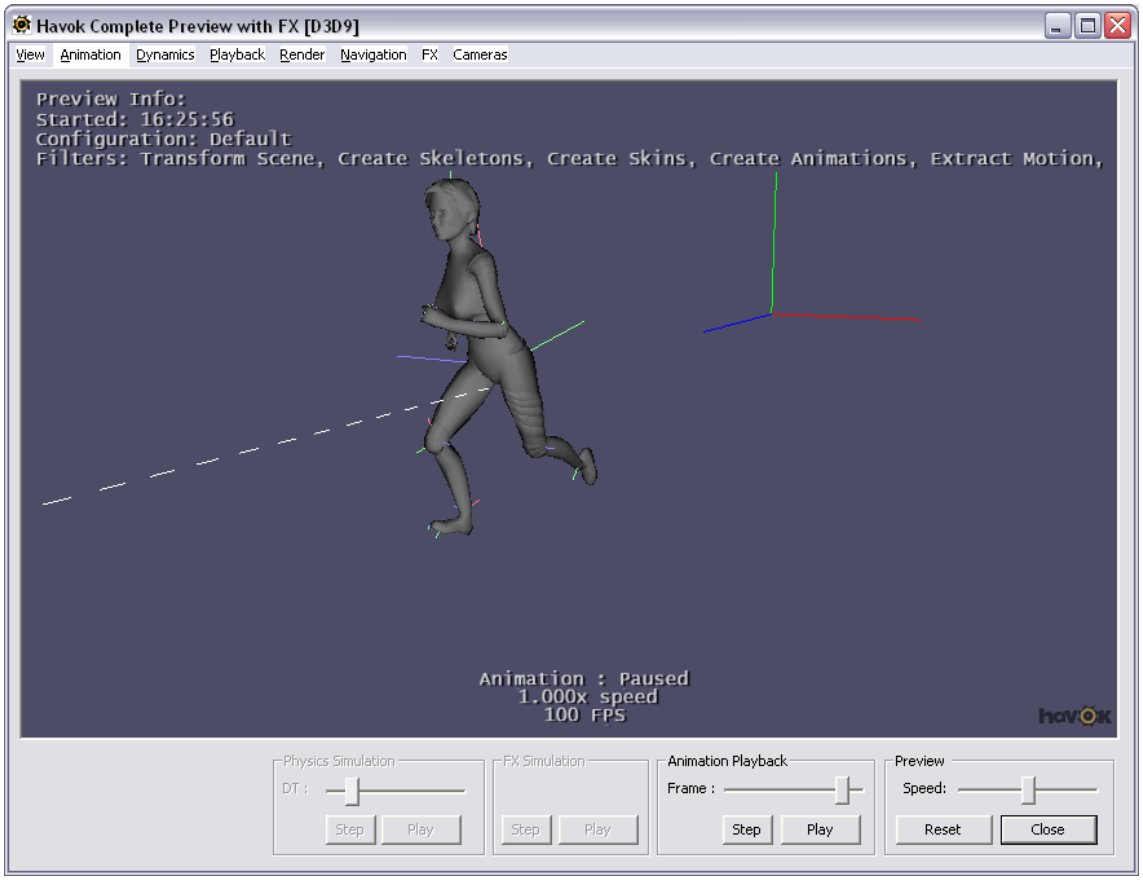


The Preview Scene filter knows about motion and it allows you to visualize it. In the preview dialog, click on **View** > **Extracted Motion** :



A dashed line is displayed showing the motion extracted from the animation. The preview can also apply and accumulate the extracted motion. In the **Animation** menu, click on **Accumulate Motion** :

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Notice how the character now moves according to the motion that was extracted - notice also how that motion accumulates: the character moves forward continuosly over animation loops (instead of warping back to the starting position at the beginning of each loop).

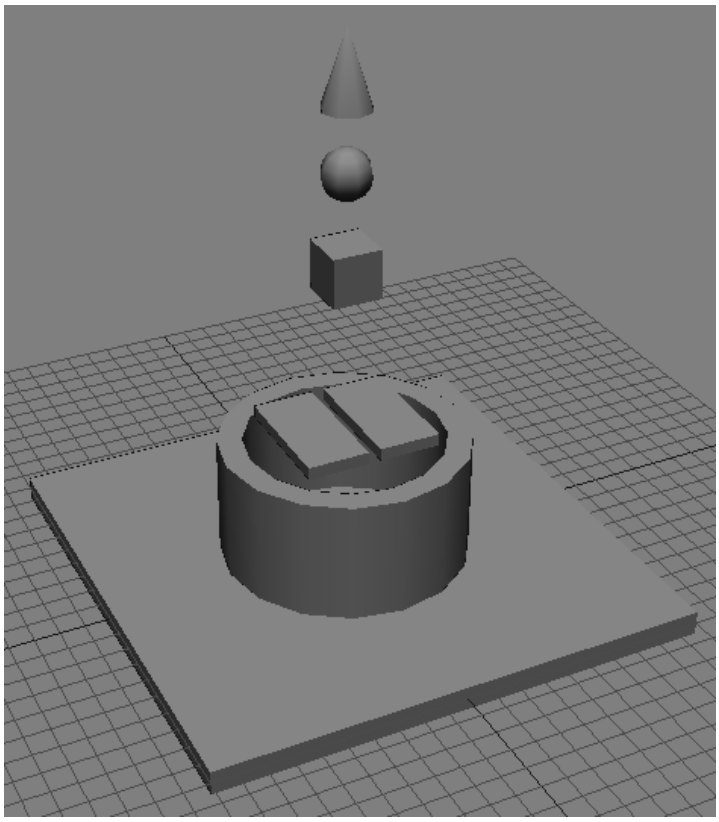
You can find the tutorial scene with motion extraction applied in the `"exportBasics/tutorialEnd_MotionExtracted.scn" file`.

6.5.6 Tutorial: Physics Basics

In this tutorial we are going to explore the basic operations for seting up rigid bodies and constraints in XSI, and processing them through the filter pipeline. This tutorial assumes that you are familliar with the basics of exporting and processing scenes, covered by the Export And Animation Tutorial. We also recommend that you take a look at the rigid body and constraint concepts sections of this documentation.

6.5.6.1 Getting Started

Start by loading the scene `"physicsBasics/TutorialStart.scn"`, in the *tutorials* project (which should be installed in `[INSTALLDIR]/Addons/HavokContentTools/tutorials`):

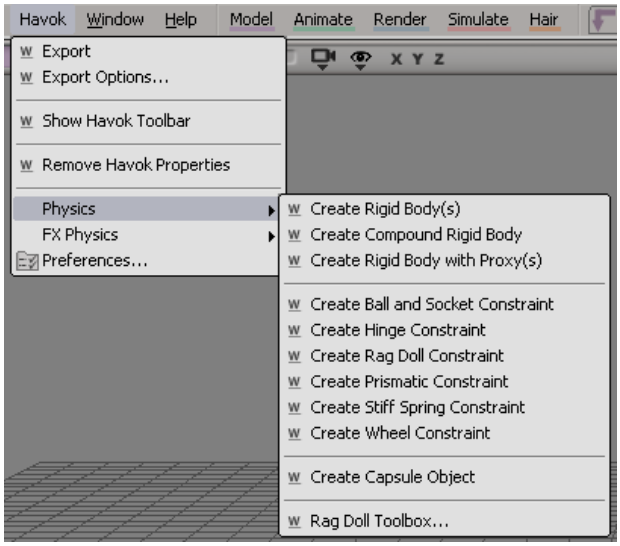


In this scene, we have a set of objects: a cone, a sphere, a box, a trap door, a can and a ground. We want to attach rigid body properties to them so they behave as they would in reality.

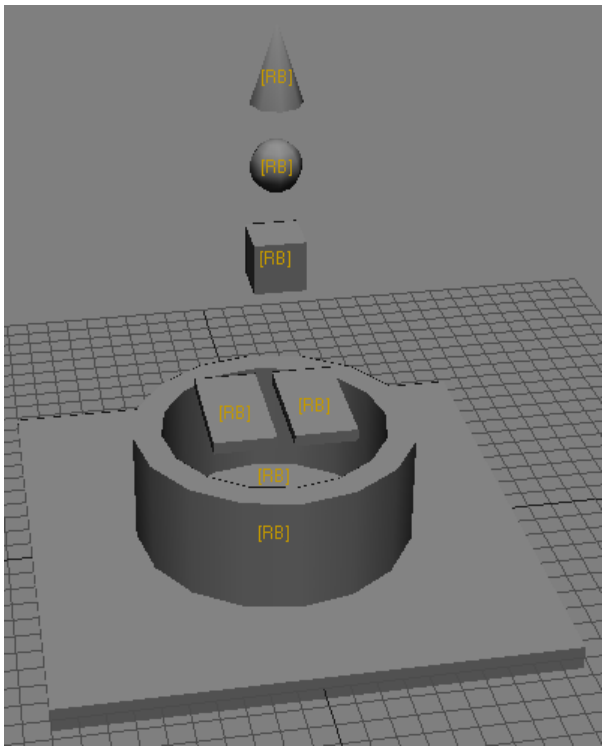
6.5.6.2 Setting Up Rigid Bodies

To start with, we need to make the objects in this scene become rigid bodies, i.e. attach rigid body information to them so we can later on create rigid bodies and simulate them using the Havok SDK.

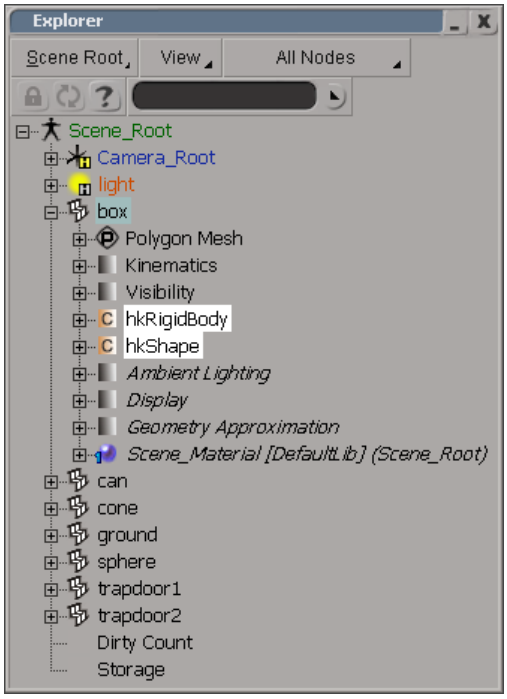
The Havok Content Tools provide a set of XSI physics tools in order to do so easily. These tools are available through the **Havok** menu and toolbar:



In this scene, we want to create a rigid body out of every single object, so let's start by doing so. Select all the objects in the scene, and click on the **Create Rigid Body** button or menu option. Notice how all the objects in the scene are now are labeled with the text " **[RB]** ", indicating that these objects have now rigid body properties. Note that the square brackets around the "*RB*" indicate that these are all currently fixed rigid bodies.



In XSI, rigid body properties are stored in custom parameter sets. For example, if you select the suspended box in XSI's explorer view, you should see that two parameter sets have been attached to the box:

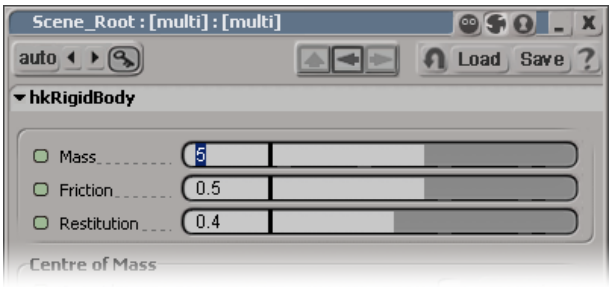


The Rigid Body Parameter Set (*hkpRigidBody*) contains information about the dynamic behaviour of the object: mass, friction and restitution are specified here. The Shape Parameter Set (*hkpShape*) contains information about the representation of the object for collision detection.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

We will be exploring these parameter sets a little bit more later on. For now, open the rigid body parameter set's property panel (by double-clicking on the *hkpRigidBody* parameter set). Notice how the mass of the box is currently zero - this is the default mass for all new rigid bodies. Since Havok will consider a mass of zero to be an indication of the rigid body being fixed in space (not moving), we want to make sure that we specify a mass for all the dynamic objects in our scene: the cone, the sphere, the box and the trap doors.

Set the **Mass** attribute of each dynamic object to a value different than zero (5Kg for example):



After doing this, we have a scene set up with 7 rigid bodies, 5 of them dynamic/movable and 2 of them (the ground and the can) fixed.

6.5.6.3 Previewing the Scene

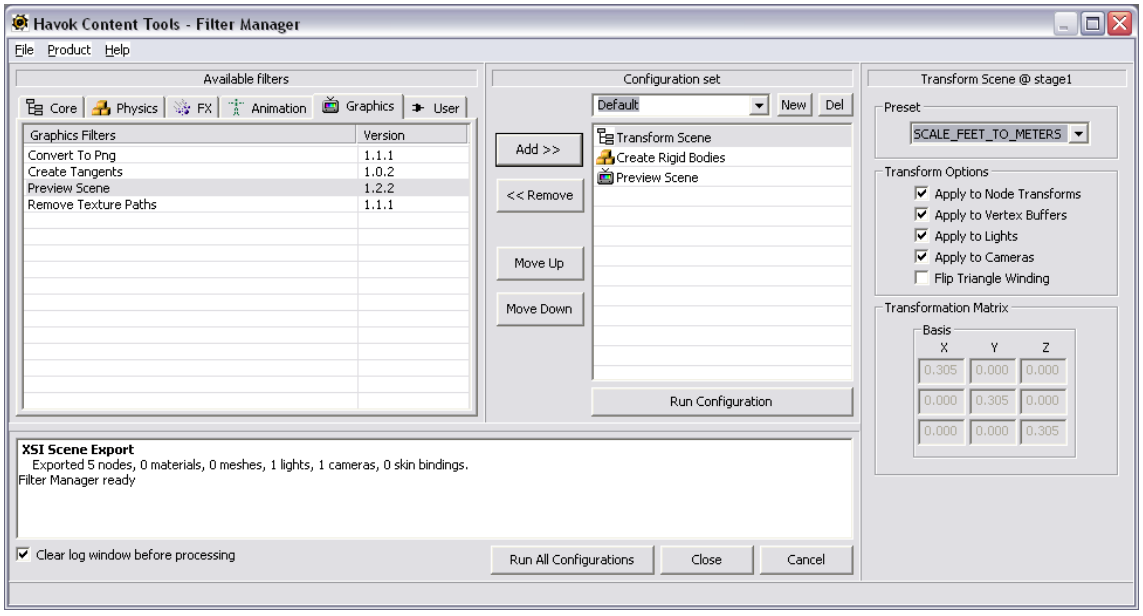
Now that we have some content set up, we'd like to preview it and check that everything behaves as expected. In the previous tutorial we saw how to export, process and preview assets using the XSI scene exporter and the filter pipeline. We are now going to do the same, but this time using some processing specific to physics assets.

Start the filter manager by clicking on the **Export** button or menu option. You should begin with an empty filter setup - remove any filters which may be present.

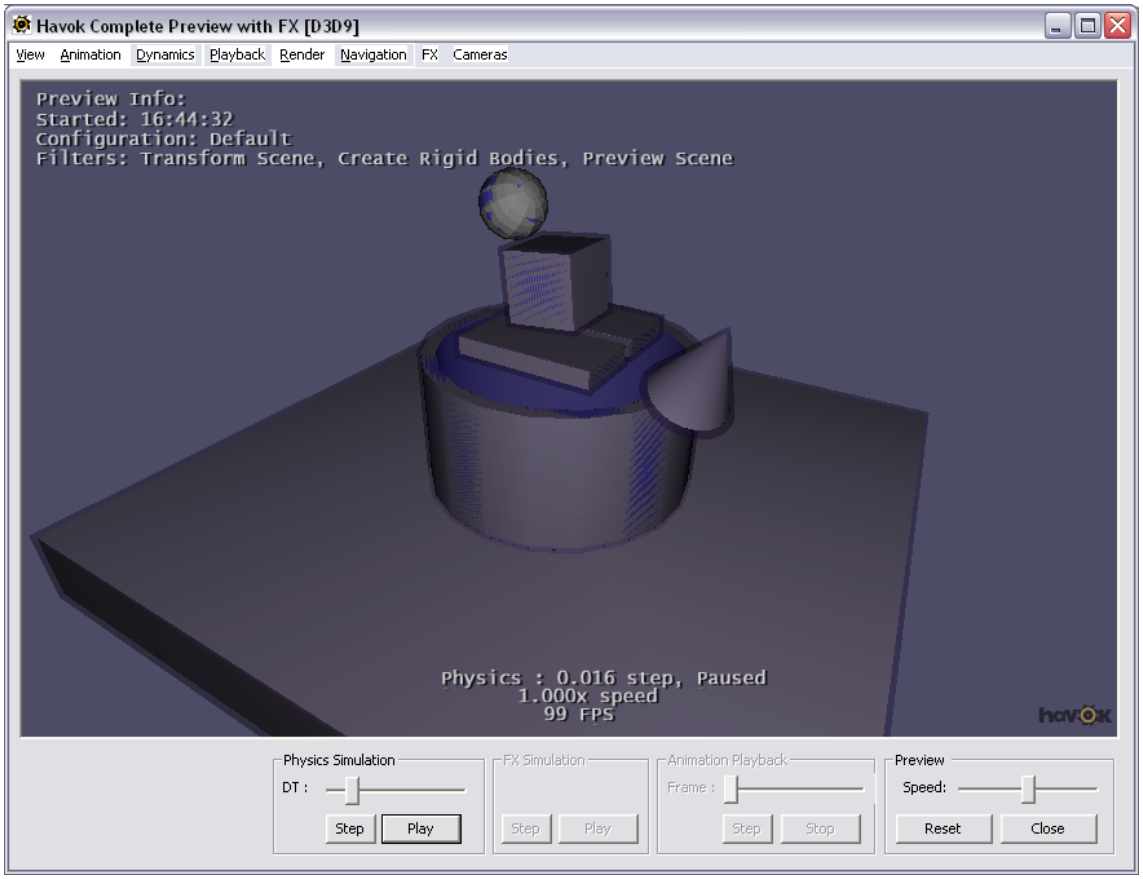
Insert, in this order, the following filters:

1. Scene Transform (**Core** category): Since the scene units are pretty big, and Havok works in meters, set the **Preset** parameter to **SCALE_FEET_TO_METERS** (for example).
2. Create Rigid Bodies (**Physics** category): This filter will detect the rigid body information we assigned to our objects and will create rigid bodies based on it.
3. Preview Scene (**Graphics** category): We want to visually inspect the behaviour of our objects, so we will use this filter to render and simulate them.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



If we now process our asset (press the **Run Configuration** button), a preview window appears with our scene. Press the **Play/Stop** button to start the simulation:

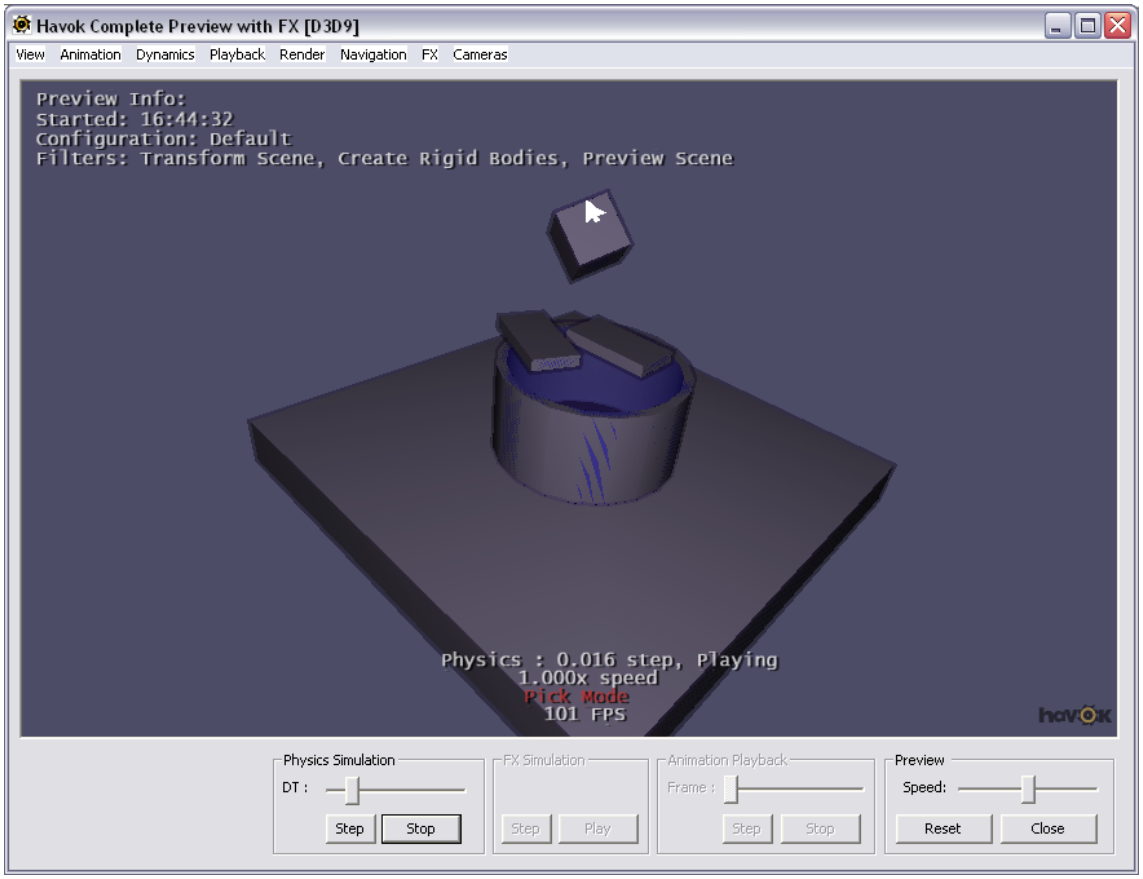


You will notice that all the objects fall as expected, but there are several things that are not quite right:

- The trap doors do not behave as such, and are just falling as any other object. This is because we have not set up any constraint to specify that they should behave as hinged doors. We will take a look at this later on this tutorial.
- None of the objects fall inside the can - it is as though there is an invisible lid on it.

If you switch on the menu option **View** > **Physics Ghosts** you should see a ghost representation of the rigid bodies displayed in the preview (in alpha blended blue). That shows that there is indeed a lid on the can. It looks like the can is being simulated as a closed volume!

Mouse Picking
You can interact with the objects in the scene by using *mouse picking*: with the mouse cursor over one of the movable objects, press the **SPACE** bar and, without releasing it, move the cursor around - the rigid body will behave as if a spring was attached between itself and the mouse pointer. Release the **SPACE** bar to release the object.

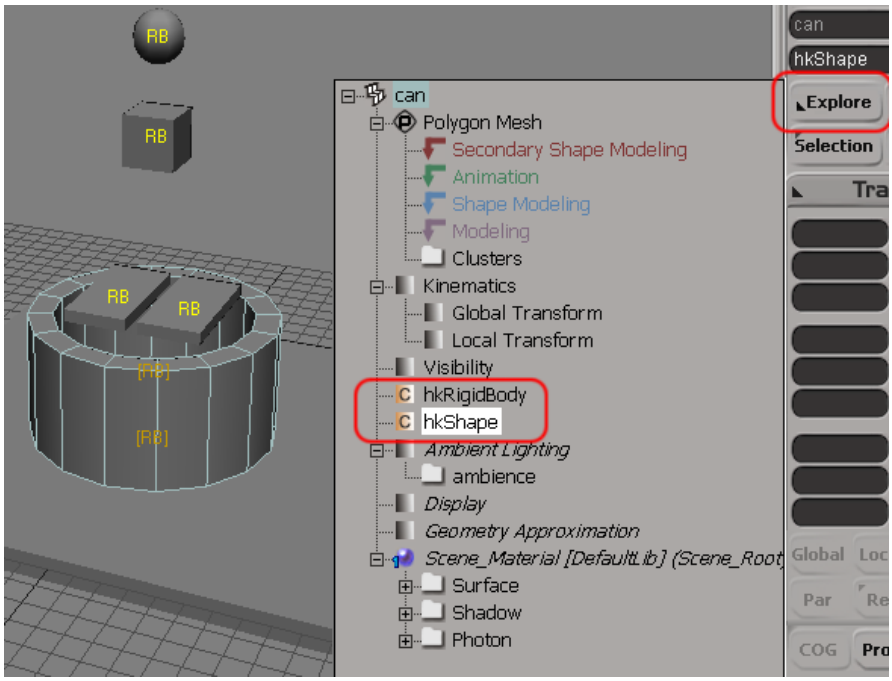


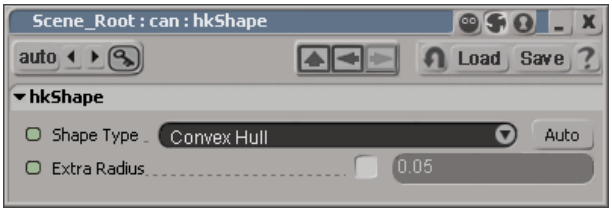
Close the preview and the filter manager and return to our XSI scene.

6.5.6.4 Shape Parameters

Let's take a look at the can. Since our problems are related to collision detection, we'll take a look at the shape information associated with it. Open the shape parameter set's property panel - do this by

selecting the can in the viewport, opening the 'selection' popup from the tools panel, and double-clicking on the *hkpShape* custom parameter set:





Notice how the Shape Type parameter is set to "Convex Hull".

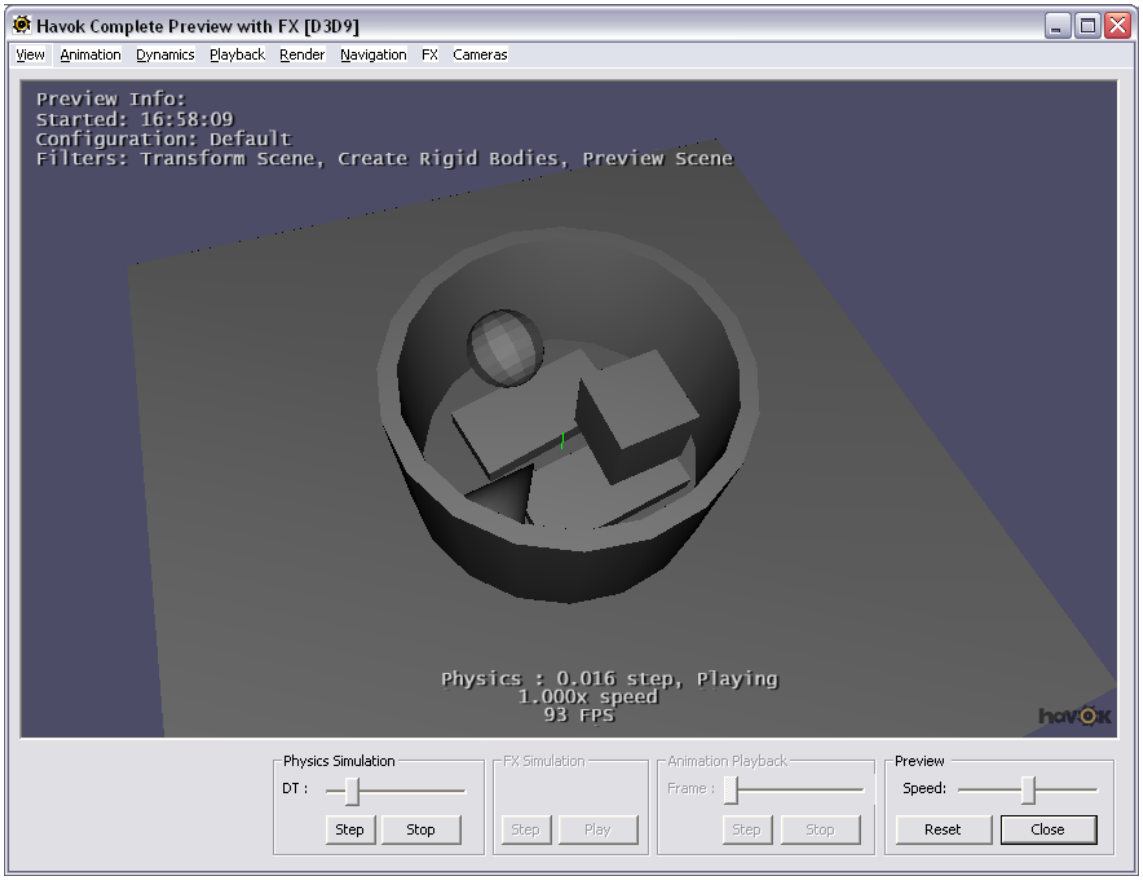
When creating a rigid body from a mesh, a value for this Shape Type parameter is guessed by examining the object's geometry in XSI. Thus, the default Shape Type for boxes is (Bounding) Box, the default Shape Type for spheres is (Bounding) Sphere, etc. For other arbitrary objects the default Shape Type is Convex Hull. The Convex Hull of an object is the tightest convex volume that includes all the vertices of the object. The best way to visualize a convex hull is by imagining shrink-wrapping the object. In our case, using the convex hull of the can results on simulating a closed cylinder, rather than an open one, leading to the behaviour we observed in the preview.

In our case we want to use the exact mesh of the object instead of the convex hull, so we need to change the Shape Type parameter to **Mesh** .

Note:
The reason why the default Shape Type is Convex Hull and not Mesh is that convex objects are much faster to simulate than concave meshes and, in many cases, simulating a mesh as its convex hull gives very good results. In addition, movable 'mesh' objects are not really suitable for interactive simulations (a warning will be generated if used).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

After changing the shape type to Mesh, if you now **Export** and process the scene again you will see that the objects now properly fall inside the can during the simulation:



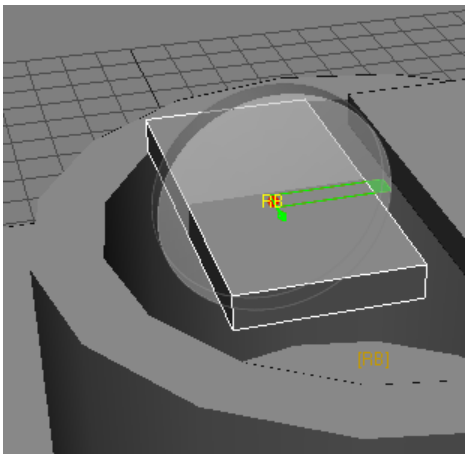
Close the preview and the filter manager and return to our XSI scene.

6.5.6.5 Setting Up Constraints

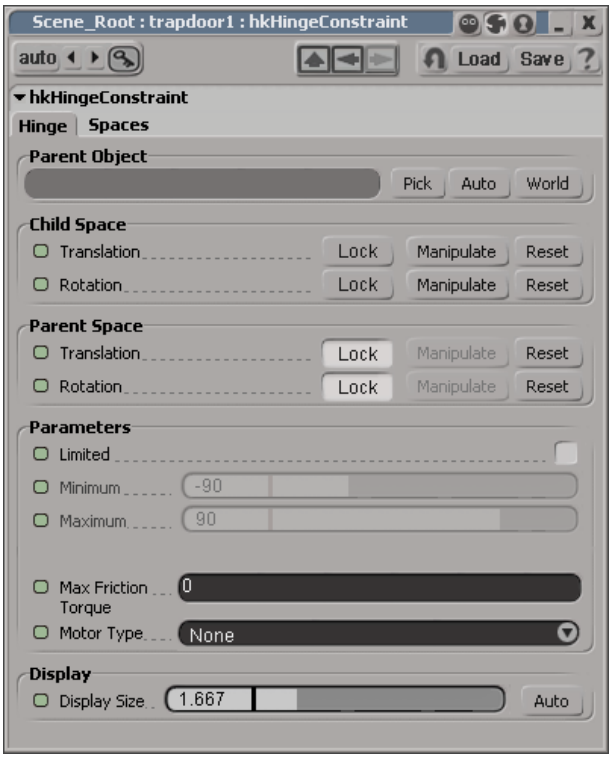
Another thing we want to improve in our scene is the behaviour of the trap doors: currently they behave just like the other objects (they just fall). We want them to behave as if they were constrained to their location by hinges.

The Havok Content Tools give you the ability to set up and visualize constraints in XSI. In our case, we want to create a hinge constraint - a constraint that will limit the movement of the trap doors to a rotation around a single axis.

Select one of the trap doors and click on the **Create Hinge Constraint** button or menu option. This adds a **Hinge** constraint parameter set to the object, which is represented in the viewport as shown:



Examine the parameters of the hinge constraint (ie. the *hkHingeConstraint* parameter set) in a property editor:



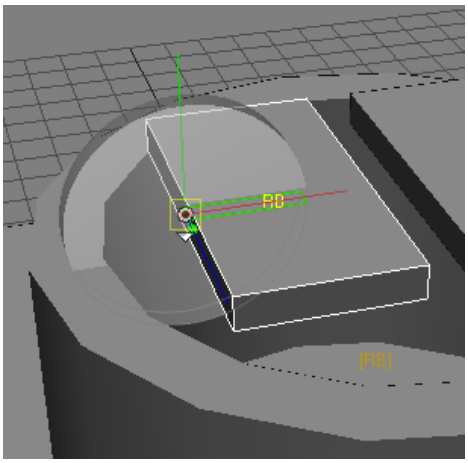
First of all, notice how there is no parent object specified: this means that this constraint will only affect this rigid body: it's position, orientation and limits will be absolute, rather than relative to another (parent) rigid body. This is what we want.

The constraint is represented in the viewport as an axis and a volume of allowed rotation (with another axis specifying where zero-rotation is). By default these axes are aligned to the object's Z and X axis, and the constraint is placed at the object's pivot point. In our case, however, we'd like to place the constraint on the side of the trapdoor and orientate it so that the trap door will rotate up and down rather than sideways. We can change the position of the constraint in two ways: by modifying the object pivot or by manipulating the constraint spaces.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Positioning the constraint by modifying the object pivot

With the trap door selected, select XSI's translate tool and hold the " **Alt** " key - this will put the tool into pivot mode. Then, drag the pivot of the trap door to the side which we want to act as the fixed point of the the door. Observe how the constraint representation updates to follow the pivot point:



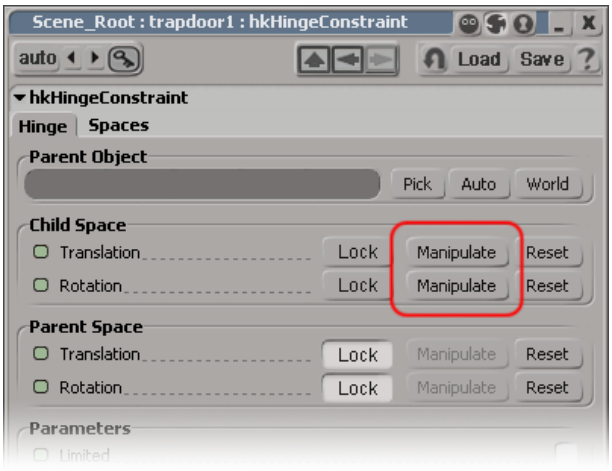
Note:
An option is provided by the Havok Content Tools to 'include pivot in transforms'. If this option is disabled then moving the rotate pivot of an object will have no effect whatsoever on the constraint. See the XSI Export Options for more.

Positioning/orientating the constraint by manipulating it's constraint spaces

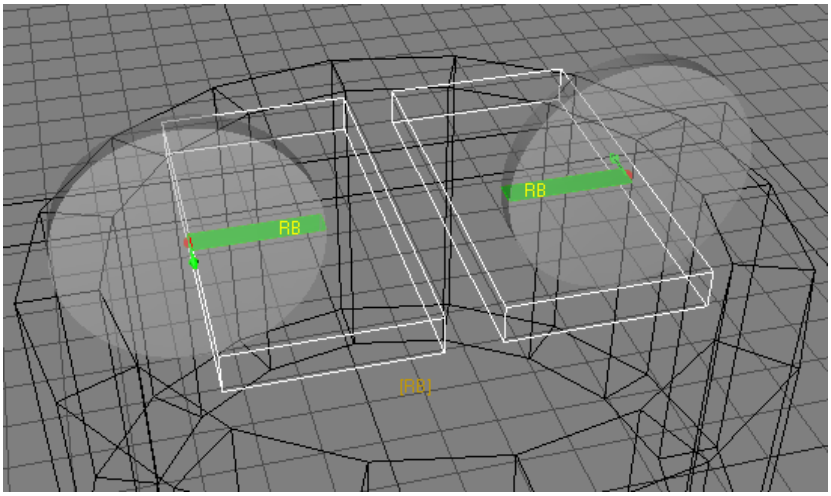
As observed, the constraint position normally follows the pivot point of the object. However, we cannot change the orientation of the hinge using the pivot point. We can only do this by manipulating the constraint space(s).

Let's now experiment with the orientation of the hinge constraint for the trap door. Open the property panel for the hinge constraint parameter set. In the 'child space' section under the 'hinge' tab, click one of the 'manipulate' buttons. Now drag the manipulator in the viewport to position/orient the hinge as desired. We want the main axis to point along the length of the trap door, and the zero-rotation axis (the one within the volume) to point towards the other trap door (as in the screenshot above).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Repeat the above process with the other trapdoor to create a hinge constraint with the appropriate position and rotation. In this case try positioning the constraint by manipulating the constraint space only, instead of moving the pivot point. Our setup should now look like this:



Note:

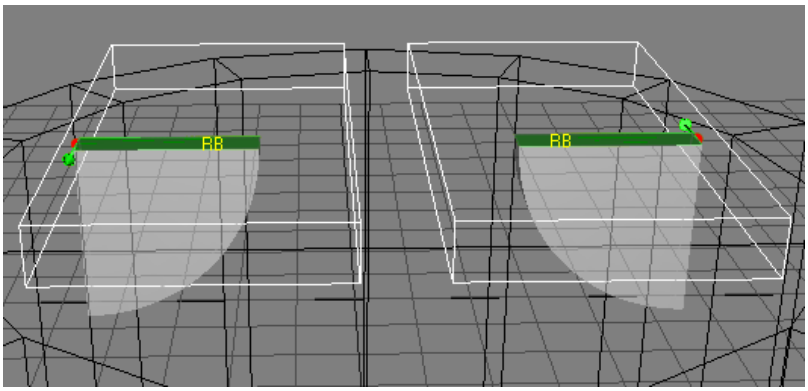
If you prefer, you can edit/tweak the spaces numerically, by switching to the 'spaces' tab in any constraint's property panel. This is useful if you want to specify an exact rotation, for example 90 or 180 degrees.

Limiting the hinges

By default, hinge constraints allow full freedom of rotation around the main axis. It is often desirable to limit that amount of rotation to a specific range.

Select one of the trap doors, and examine it's hinge constraint parameters in a property editor. Enable the *Limited* checkbox. Notice how the **Min** and **Max** parameters are now enabled, and how the limits are reflected in the hinge volume in the viewport (the limits are specified counterclockwise from the zero-rotation axis). Tweak the limits of each hinge so that they appear as follows:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

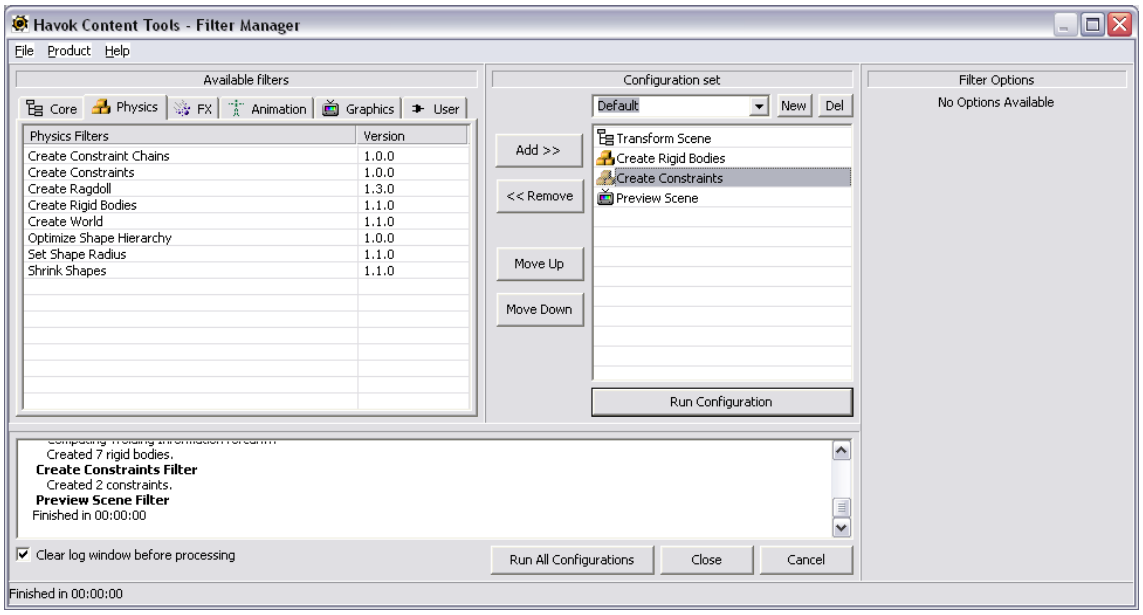


6.5.6.6 Previewing the Scene

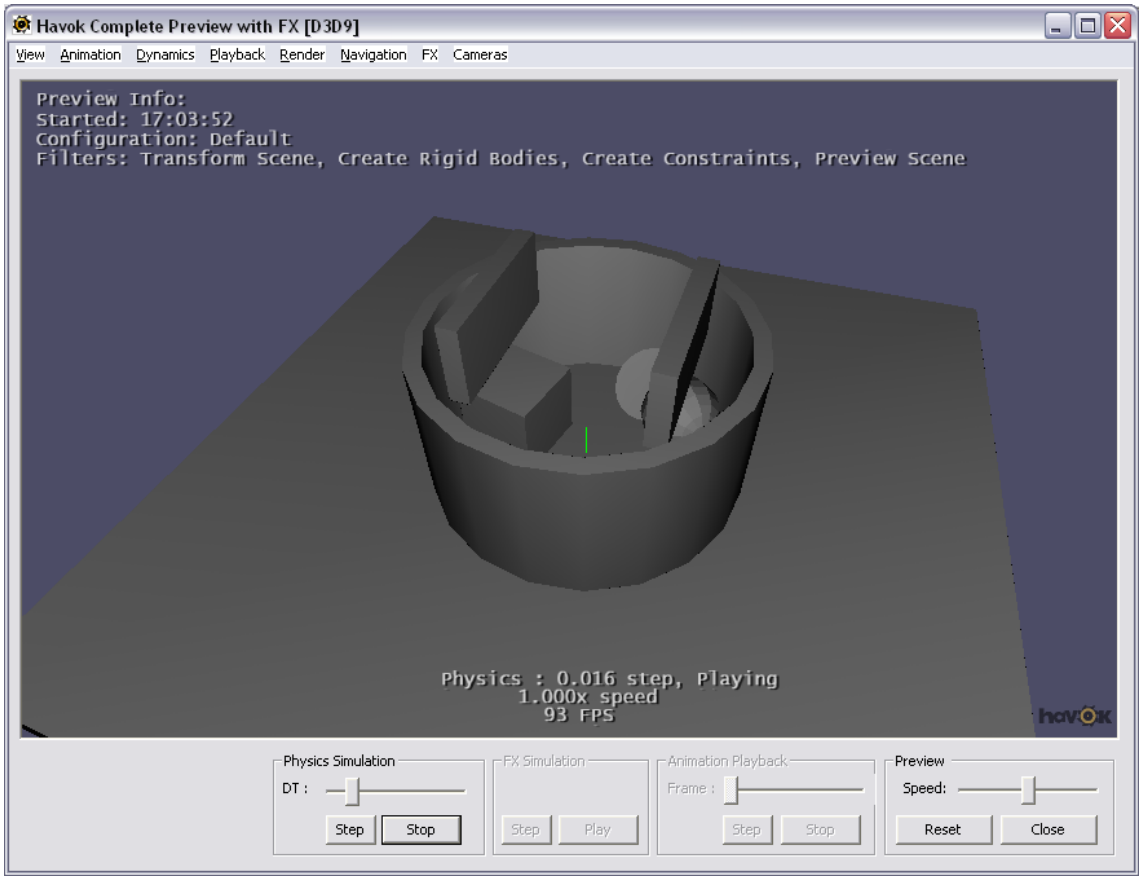
Now that we have set up our constraints, let's see what effect they have on the simulation.

Open the filter manager again by exporting the scene .

We now need to add a new filter, the Create Constraints filter (in the **Physics** category), in order to create constraint objects from the data we set up in XSI. This must be placed after the Create Rigid Bodies filter, and before the Preview Scene filter.



If you now execute the processing by pressing **Run Configuration** , you will see the constraints in action:



You can check that the limits we applied to the hinges are working by using mouse picking - you will notice that the doors won't move beyond the limits we specified.

Close the preview window and the filter manager, and return to XSI.

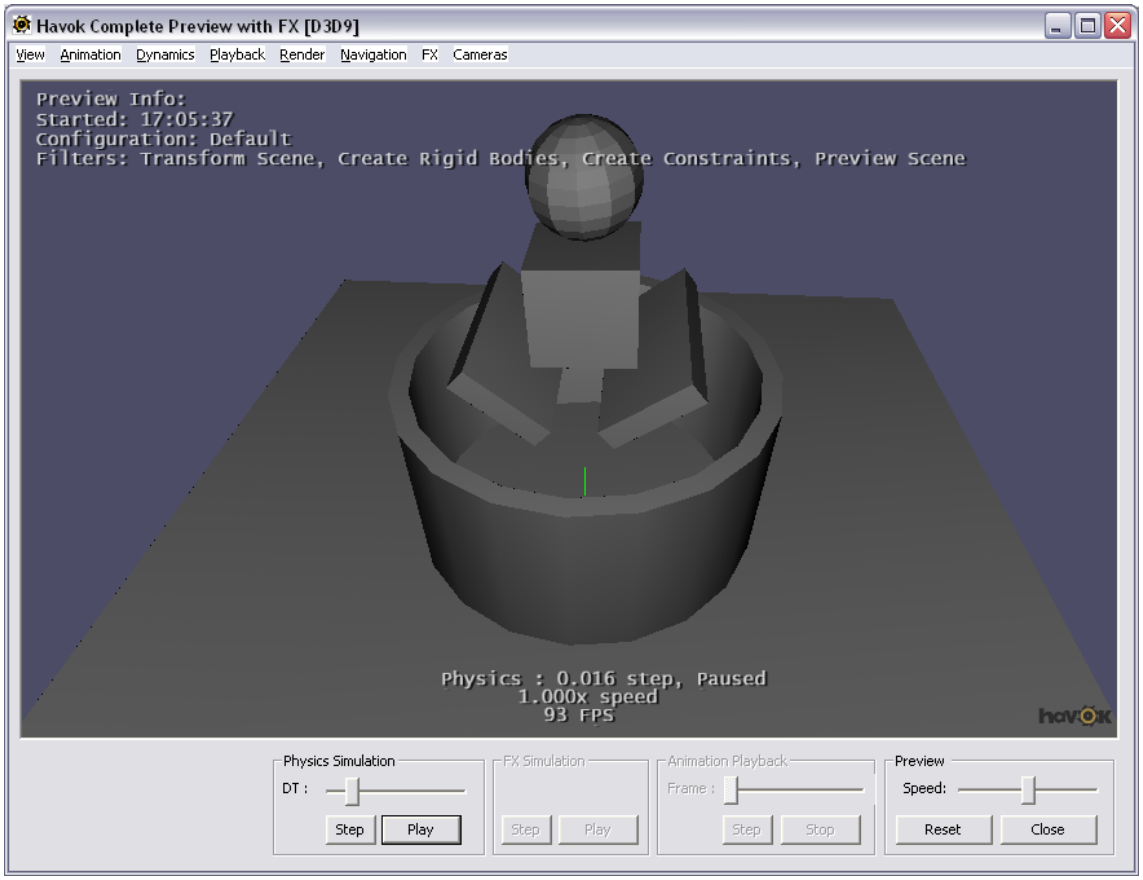
6.5.6.7 Adding friction to constraints

Our setup is almost finished. The last thing we'd like to fix is the fact that the trap doors open even before the objects hit them - this is due to the gravity in the previewer. We'd like to add some friction to the hinges so the doors only move when pushed by the objects falling on them.

Select each of the trap doors and open the hinge constraint parameter set in a property editor. Change the **Max Friction Torque** parameter to 40. This parameter specifies how much torque (angular force) is absorbed by the hinge's friction - only torques above this value will cause the hinge to move.

If you now export and preview the scene again you will notice how the trap doors remain closed until the objects fall on top of them.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



And this finishes our tutorial. You can find the fully completed tutorial in the scene file "`physicsBasics/tutorialEnd.scn`".

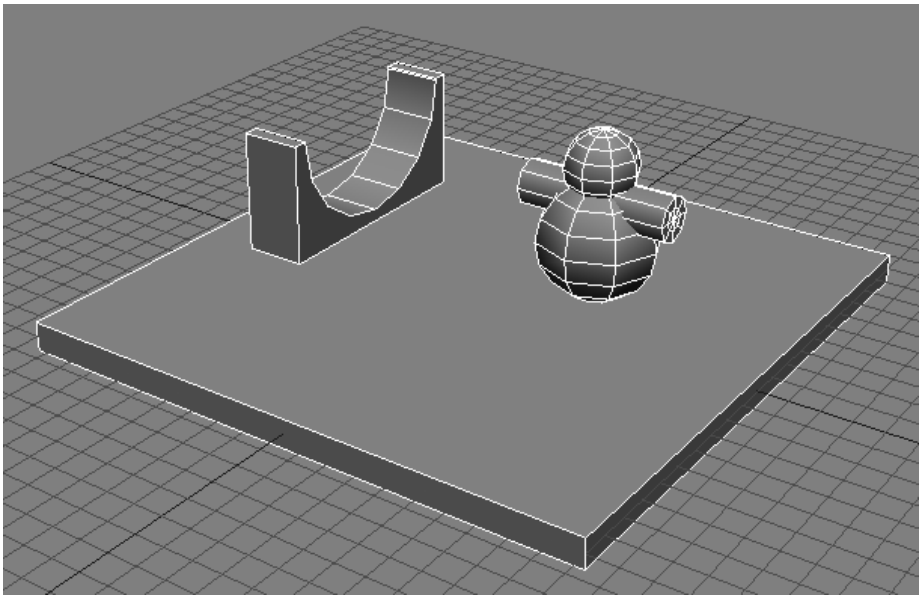
Now would be a good time to try tweaking the parameter values of each of the Rigid Body and Shape parameter sets in the scene, to familiarize yourself with the effects of each. Run the export and preview process any time you want to see the results.

6.5.7 Tutorial: More on Rigid Bodies

In this tutorial we will explore some more concepts regarding rigid body setup, such as creating compound rigid bodies and changing the center of mass. The tutorial assumes that you are acquainted with the basics of exporting and processing assets and working with rigid bodies. We recommend you follow the first two tutorials (Export and Animation Basics and Physics Basics) before proceeding with this one.

6.5.7.1 Getting Started

Start by loading the scene "`moreOnRigidBodies/TutorialStart.scn`", in the *tutorials* project (which should be installed in `[INSTALLDIR]/Addons/HavokContentTools/tutorials`):



In the scene you should see the following:

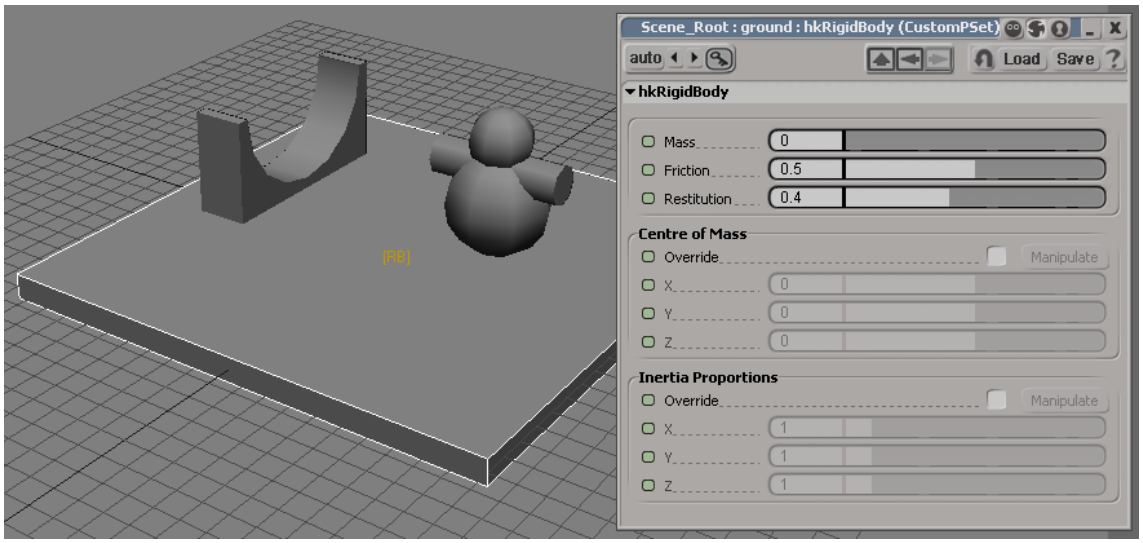
- Two spheres and a cylinder, making up the shape of a little toy man
- A concave object (a little ramp)
- A ground box

We are going to create a rigid body object for each of the toy man and the ramp, and we will use the ground box as a fixed body so that we can preview the behaviour.

6.5.7.2 Creating a Fixed Rigid Body

Let's start by making a rigid body out of the ground box. Select it and press the **Create Rigid Body** button or menu option.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



The default mass is zero, so the ground box will be fixed in space (notice the square brackets around the label, indicating a fixed body). That's exactly what we want.

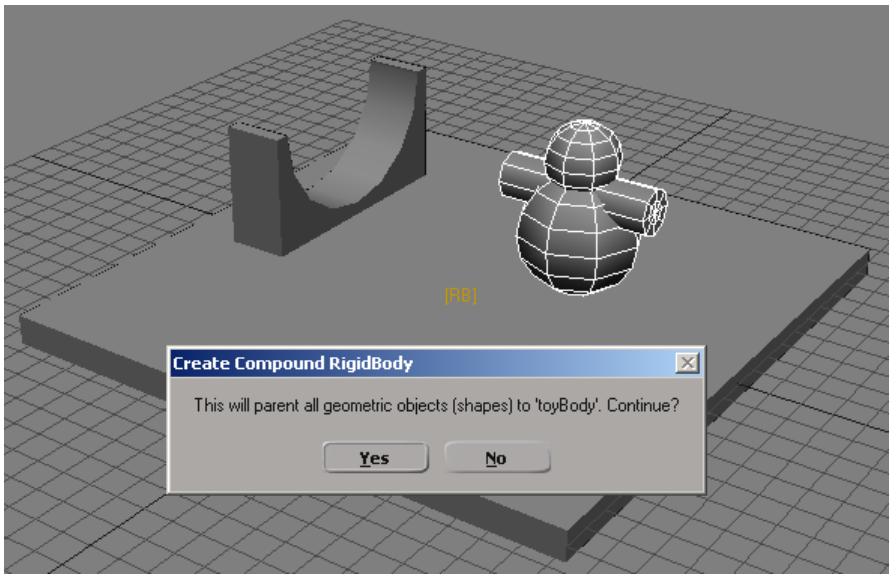
6.5.7.3 Creating a Compound Rigid Body

For the toy man we want to create a single rigid body from the two spheres and the cylinder. When we create a rigid body like this out of multiple objects, we are creating a compound rigid body.

When creating a compound rigid body, one of the objects needs to take the role of rigid body - this object will hold the rigid body information, and the rigid body will assume this object's name. The other objects need to be children of this rigid body object. If the objects are not in the correct hierarchy already, the last object selected will be considered to be the rigid body object and the other objects will be reparented accordingly.

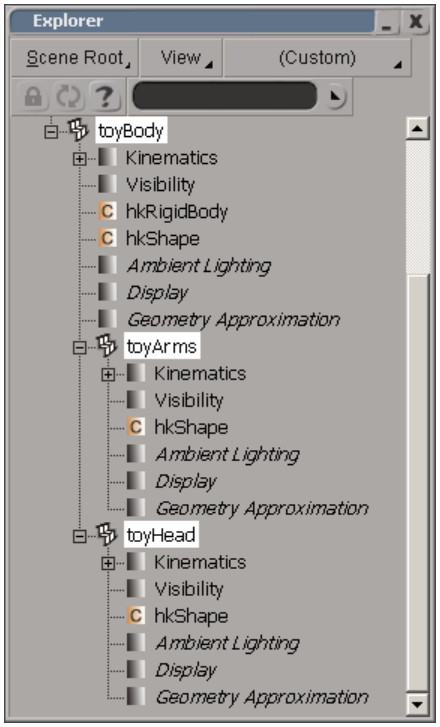
Select, in turn, the head, the arms and finally the body of the toy (holding SHIFT for multiple selection). Then click on the **Create Compound Rigid Body** button or menu option:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



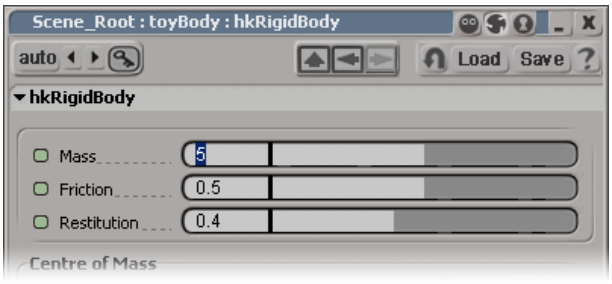
A confirmation dialog appears. This is because, as we mentioned, compound rigid bodies need to form a hiearchy. The Havok Content Tools have assumed that the last object selected (*toyBody*) should become the parent (the one that will hold the rigid body information) - but is asking us for confirmation before proceeding to reparent the arms and the head. Click **Yes** to continue.

Now, *toyArms* and *toyHead* become reparented to *toyBody*. Rigid Body information (a Havok Rigid Body Parameter Set) is be added to *toyBody*. Shape information (a Havok Shape Parameter Set) is added to all three objects (as the three of them are used for collision detection). Verify this setup using the scene explorer:



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

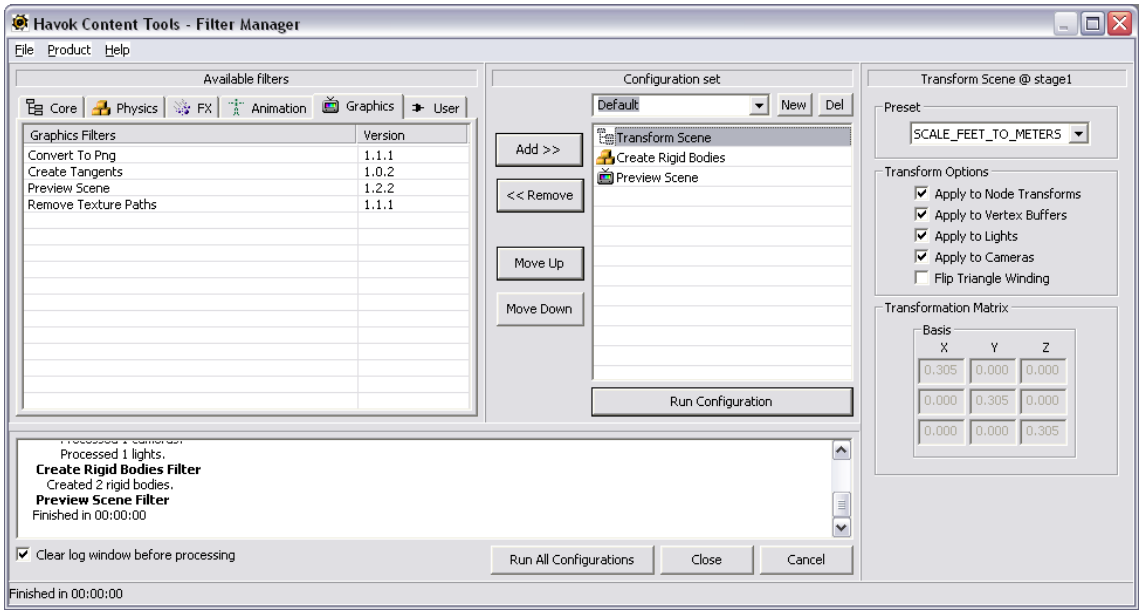
All rigid bodies have a default mass of zero. We want this rigid body to be dynamic, so open the property editor for the the rigid body parameter set and set the mass to a different value, let's say 5 Kg:



It is also useful to open the property editor for each Havok shape parameter set, and verify that the 'shape type' parameter is correct. The arms of the toy should be set to " **Cylinder** " and the head and body to " **Sphere** ". The Havok Content Tools will automatically detect some XSI shapes and set the appropriate shape type for them when creating rigid bodies, but one should always verify that the parameter has the desired setting.

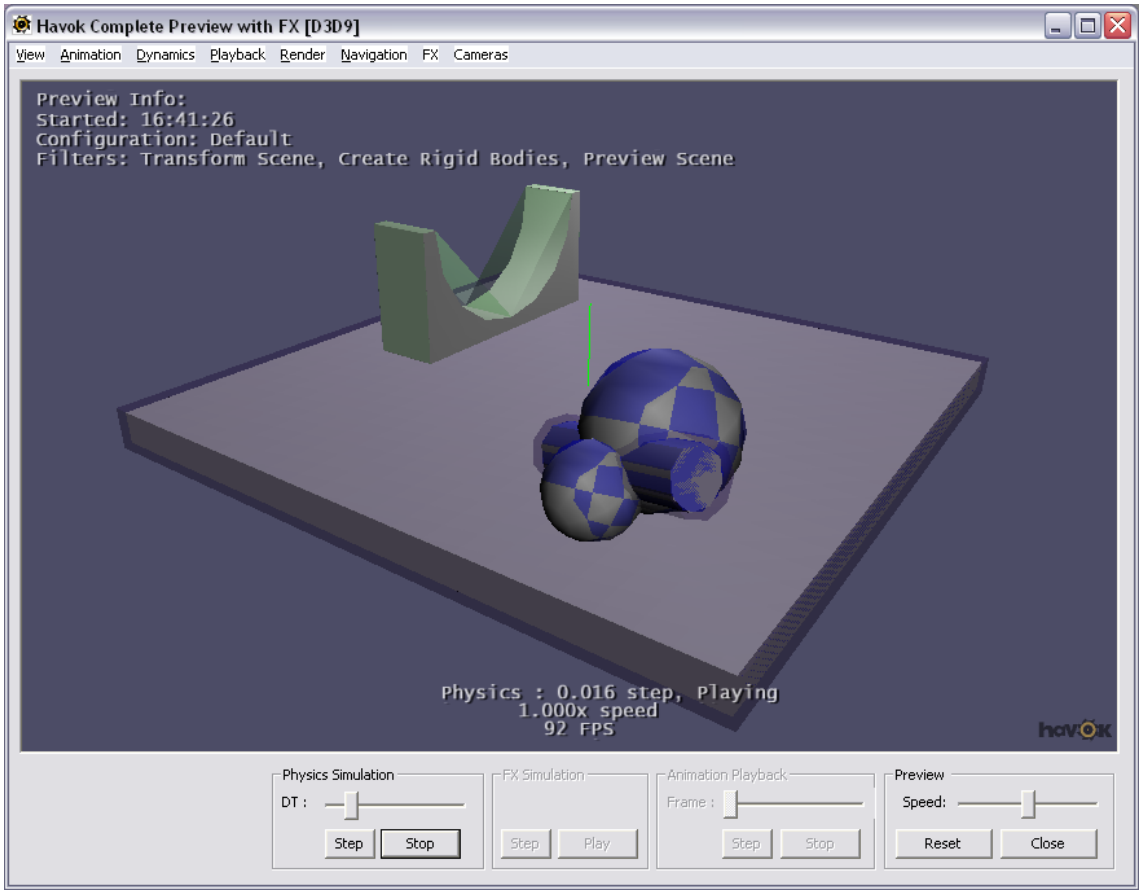
Previewing the Scene

Let's now preview the behaviour of our compound rigid body. Click on **Export** and, in the filter manager, add a Scene Transform filter (using the **SCALE_FEET_TO_METERS** preset), a Create Rigid Bodies filter, and a Preview Scene filter (as we did for the previous tutorial):



Click on 'Run Configuration' - this will process the asset and open the preview window. Press the 'Play/Stop' button to start the physics:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



As you can see (use mouse picking to interact with the toy man) the three objects behave as a single rigid body.

To toggle whether the meshes (in gray) and/or the physics shapes (in blue) are displayed, use the **Display Meshes** and **Physics Ghosts** items in the previewer's **View** menu.

Close the preview and the filter manager and return to XSI.

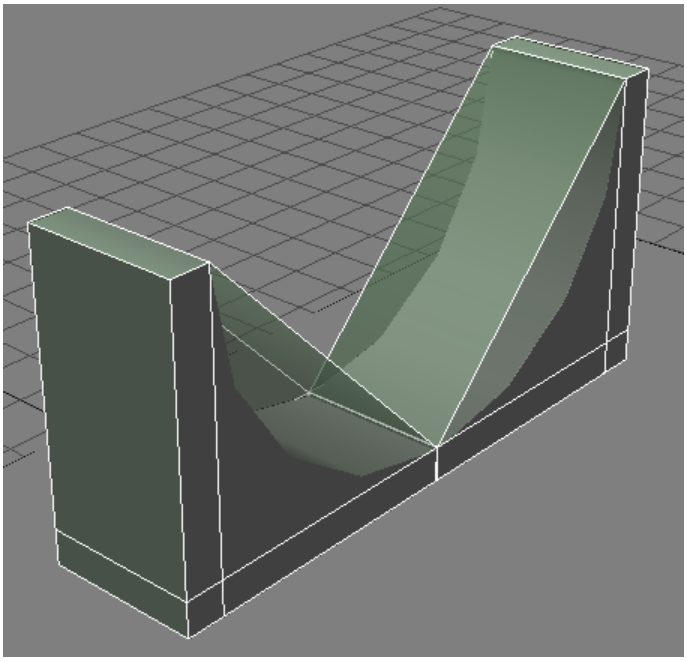
6.5.7.4 Creating a Rigid Body with Proxy Shapes

Sometimes we want to simulate an object as a rigid body, but use another object or set of objects for the collision detection rather than the mesh of the object itself. The usual example of this case is when we may want to simulate a concave object as the combination of multiple convex pieces.

Creating a rigid body with proxy shapes is very similar to creating a compound rigid body. The only difference being that in this case, the object containing the rigid body information (a Rigid Body Parameter Set) does not contain any shape information (a Shape Parameter Set) since it's mesh is not used for collision detection. Shape information is instead taken only from its descendants.

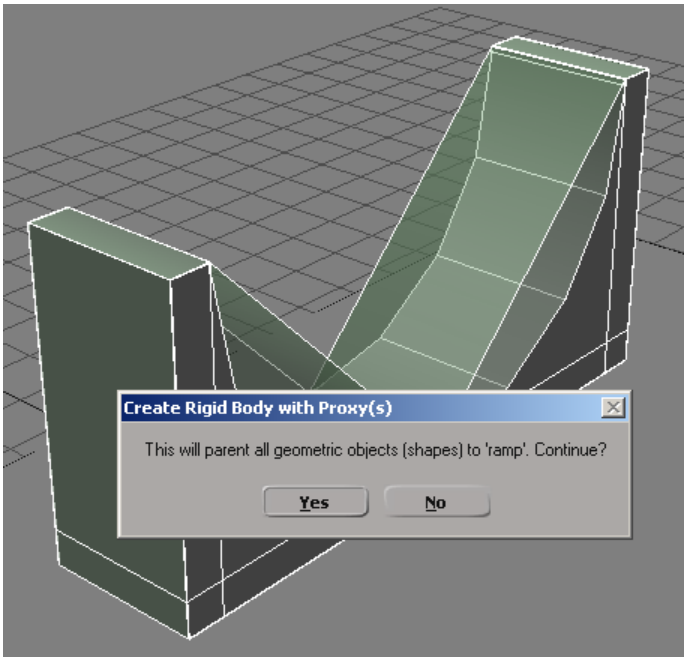
Let's see this in action in this scene. Notice how the ramp is a concave object. We'd like to approximate it's rigid body shape using two separate convex objects, since simulating convex objects is far more efficient than simulating concave objects.

Locate the two hidden objects in the scene (*rampProxy1*, *rampProxy2*) and unhide them. We will use these two convex pieces to create a proxy shape representation of the ramp. Let's start by making sure that the two proxy objects are positioned so that they match the location of the ramp in the scene (we have assigned a transparent green material to the proxy objects to be able to distinguish them easier):



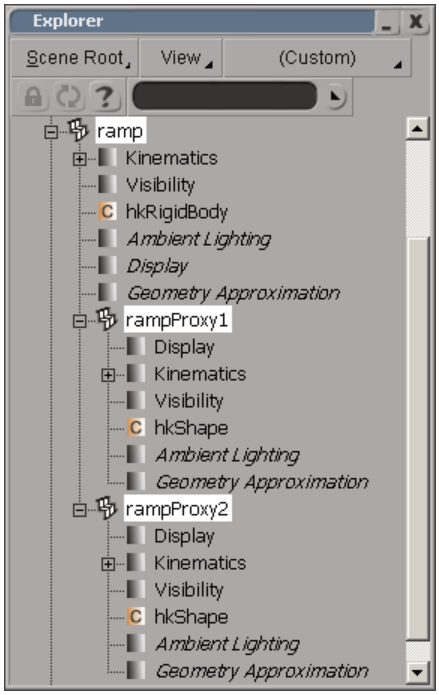
We now want to create a rigid body from the ramp while using the two convex pieces as the collision detection shapes. In the same way as compound objects are created, proxied objects require one object to be the parent of all the others. So, select the two proxy objects first and then the ramp (holding SHIFT for multiple selection, and using the scene explorer if necessary), then click on the **Havok** > **Physics** > **Create Rigid Body with Proxy(s)** menu option:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Click on **Yes** to confirm. Alternatively, you could start by first parenting the two convex pieces to the ramp manually, and then with the three objects selected click the **Havok > Physics > Create Rigid Body with Proxy(s)** menu option.

Notice how the *ramp* object now has a Havok Rigid Body Parameter Set only, and how the two convex pieces (which are now children of it) each have a Havok Shape Parameter Set only:

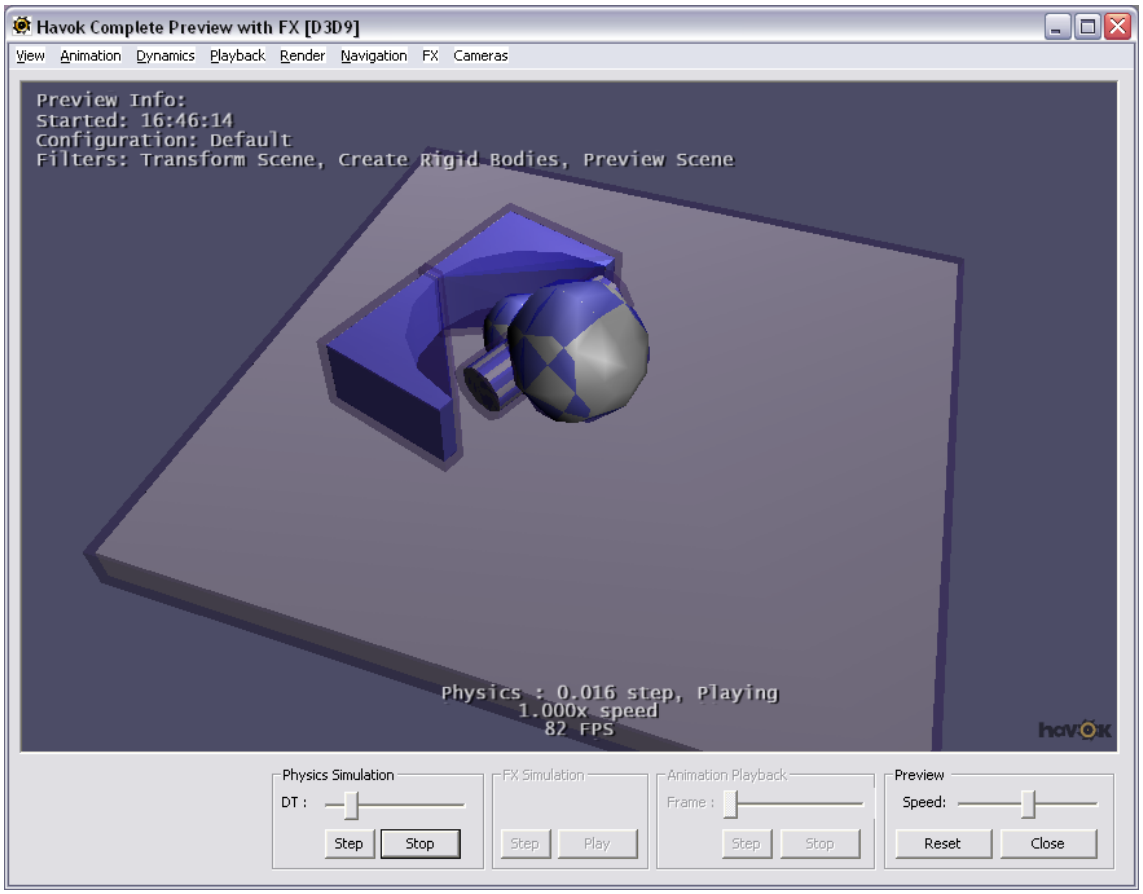


Since the two convex pieces are neither boxes, spheres, capsules or cylinders, their 'Shape Type' parameter

should be set to " **Convex Hull** ". Open each of the Havok Shape Parameter Sets and ensure that the shape type is set accordingly. Also notice how, again, the default mass of our rigid body is zero. Since we want it to be dynamic (movable), it should be set to something different (5 Kg for example).

Previewing the Scene

Let's see how the ramp behaves in our preview. Export the scene and press on **Run Configuration** to execute the preview. Using mouse picking, you can observe how the ramp is simulated using the two convex pieces:



Again, to toggle whether the meshes (in gray) and/or the physics shapes (in blue) are displayed, use the **Display Meshes** and **Physics Ghosts** items in the previewer's **View** menu.

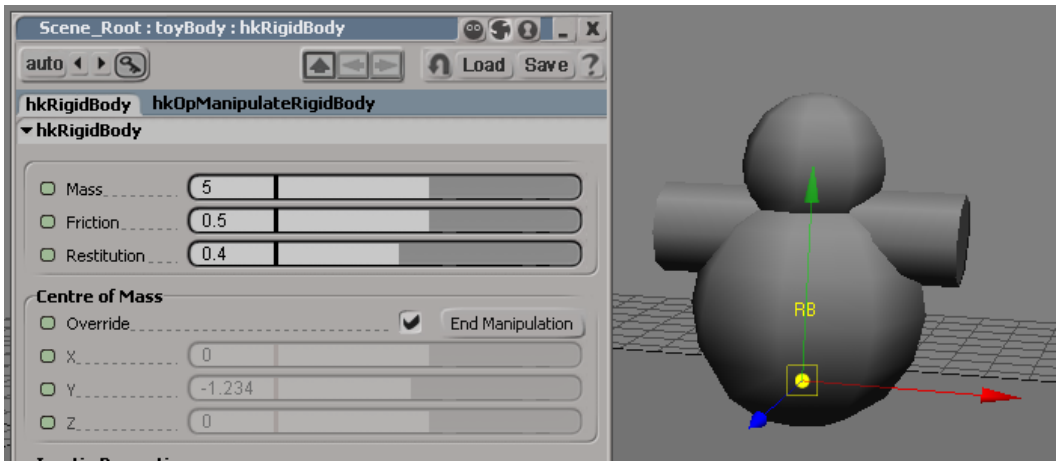
Close the preview and the filter manager and return to XSI.

6.5.7.5 Changing the Center of Mass

We'd like our toy to behave like a tumbler - one of those toys that keep themselves straight. Those toys work by having a very low center of mass (usually by having a lead weight inside them). The Havok Content Tools allow you to easily change the center of mass of any type of rigid body.

In XSI, open the *toyBody*'s rigid body parameter set in a property editor, and check the **Override Center Of Mass** check box. This enables editing of the local center of mass coordinates. The center of mass is now represented as a yellow point in the viewport. Click the 'manipulate' button in the property editor, then drag the gizmo in the viewport to reposition the center of mass. You may enter the center of mass values manully instead if you prefer.

Postion the center of mass towards the base of the toy:



Note:
While manipulating the center of mass, the 'manipulate' buttons change to 'end manipulation'. If you wish to tweak the center of mass values manually after a manipulation, you may need to press the 'end manipulate' button first.

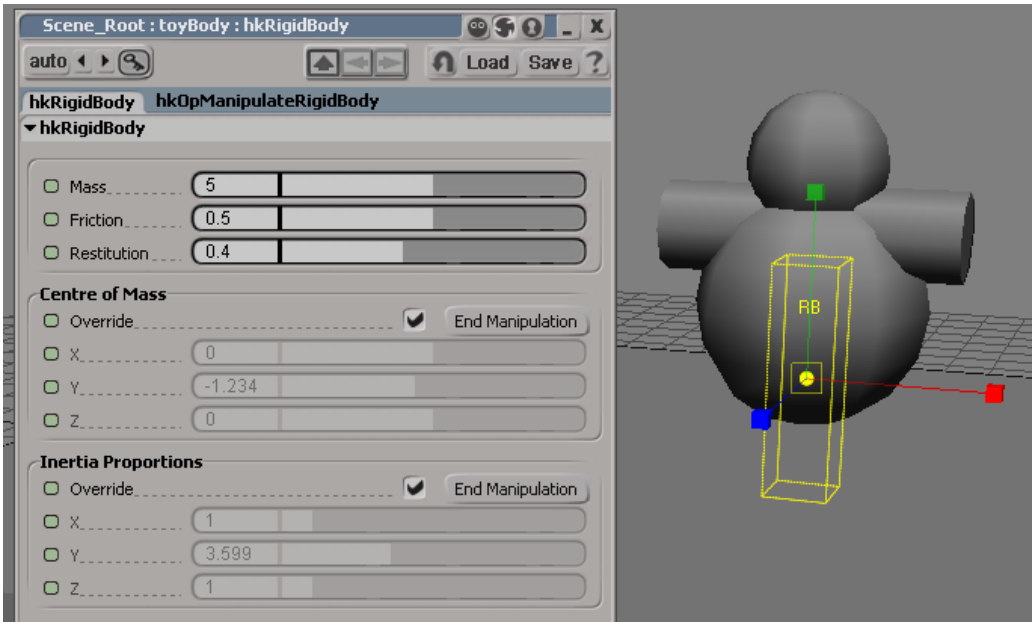
If you export and execute the preview again, you will see how the toy now tends to keep itself upright as it is dragged around the scene.

6.5.7.6 Changing the Inertia Tensor

We'd now like to customize the motion of our toy even more by changing the proportions of its inertia tensor.

Select its rigid body parameter set again, and check the box **Override Inertia Tensor** . This enables editing of the inertia tensor proportions of the rigid body. The center of mass is now represented as a yellow box in the viewport. Click the 'manipulate' button in the property editor, then drag the gizmo in the viewport to scale the inertia tensor (or tweak the values manually):

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Note:
While manipulating the inertia tensor, the 'manipulate' buttons change to 'end manipulation'. If you wish to tweak the inertia tensor values manually after a manipulation, you may need to press the 'end manipulate' button first.

If you now preview again, you will see that our little man will tend to wobble more around this vertical direction, while it won't lean as much as before. Experiment further with the center of mass and inertia tensor values, observing the results using the previewer.

And this finishes our tutorial. You can find the fully completed tutorial in the "`moreOnRigidBodies/tutorialEnd.scn`" file.

Try experimenting with the parameters further, or creating your own compound/proxied rigid bodies.

6.5.8 Tutorial: Rag Doll Toolbox

The "Rag Doll Toolbox" is a set of tools which facilitates the creation of proxies and the reuse of constraint setups. In this tutorial we will will use the Rag Doll Toolbox to set up a (low res) hierarchy of constrained proxy rigid bodies to represent a (high res) hierarchy of bones.

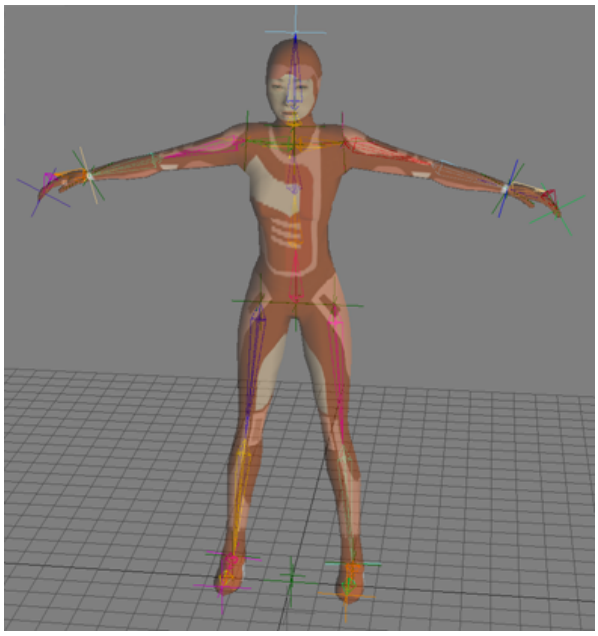
This tutorial consists of three distinct parts, in which we will:

1. Create and associate rigid bodies with an existing skeleton using the Rag Doll Toolbox.
2. Setup the hierarchy and create appropriate constraints between the rigid bodies using the Rag Doll Toolbox.
3. Setup and preview the mapping between the rigid body skeleton (ragdoll) and the original high-res skeleton.

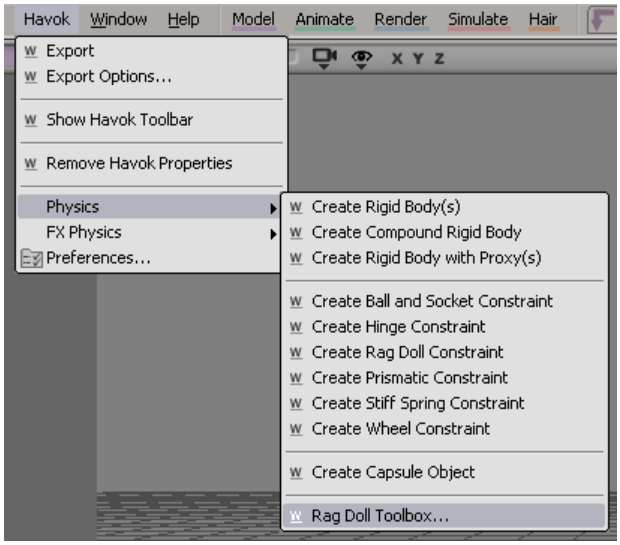
6.5.8.1 Getting Started

For this turorial, you can use any (skinned) skeleton. We will use the "`red_jaiqua.scn`" sample file which ships with XSI.

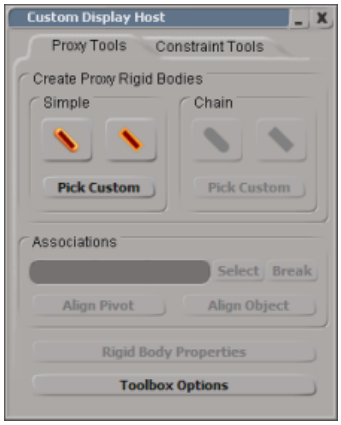
Start by loading the scene "`ragDollToolbox/TutorialStart.scn`", in the *tutorials* project (which should be installed in `[INSTALLDIR]/Addons/HavokContentTools/tutorials`):



Observe that this scene contains a skinned skeleton. Next, open and examine the Rag Doll Toolbox, via the main *Havok* menu or the associated button in the Havok toolbar:



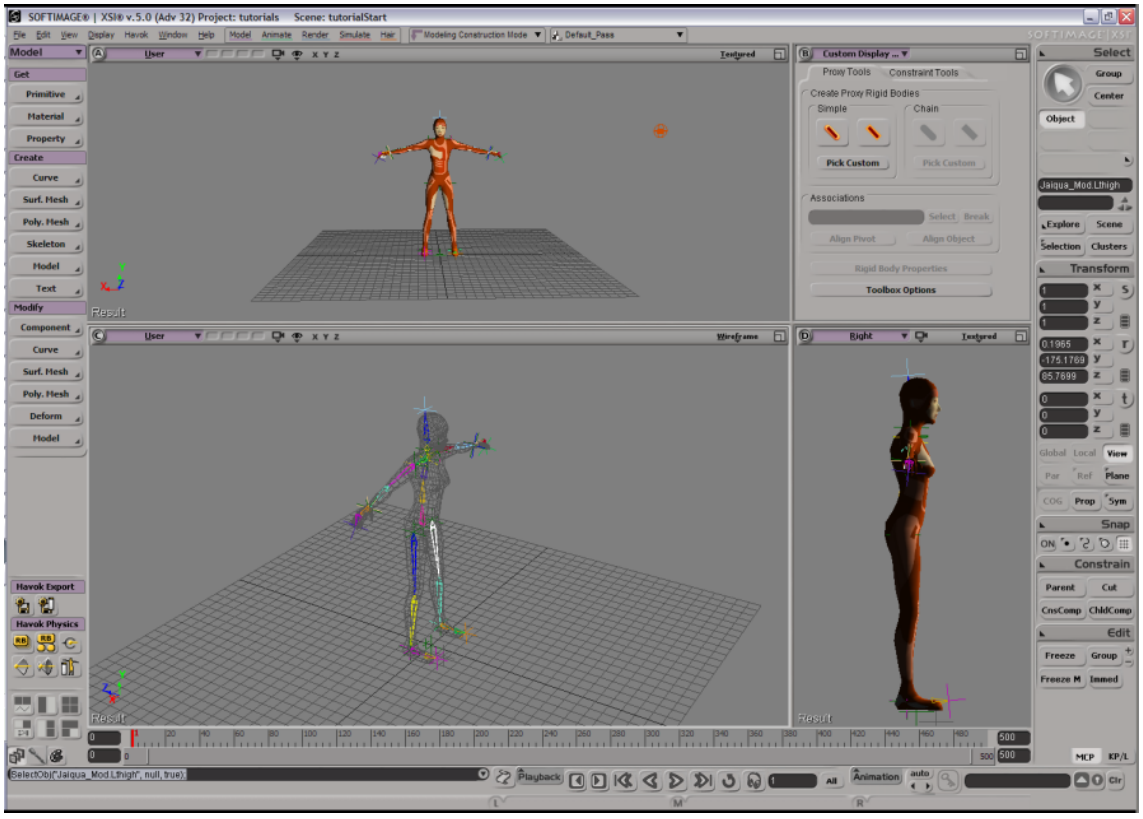
The rag doll toolbox appears as a Custom Display Host consisting of two tabs:



- *Proxy Tools*: Utilities to create proxy (rigid body) representations of animation bones, and manage associations between them.
- *Constraint Tools*: Utilities to setup hierarchies and create constraints between proxies, and reuse constraint data between characters.

XSI can also open Custom Display Hosts in any of the viewport windows by locating them in the viewports menu. This can be very useful if you are extensively using the tool:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



6.5.8.2 Creating Rigid Body Proxies

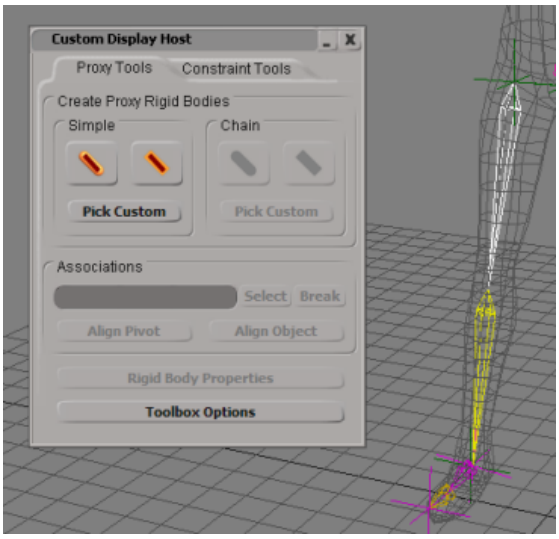
In this section we will focus on creating a rigid body representation of this character (a ragdoll) by creating and associating proxies to the character's bones. The *Proxy Tools* tab of the Rag Doll Toolbox provides the tools to achieve this.

Note:
It is useful to hide/lock the skin at this point (if not already), since we will be working only with the bones.

Simple Proxies

Simple proxies are those which have a one-to-one association with a bone.

Start, for example, by selecting one of the thigh bones of the character. Notice how the set of buttons in the *Simple* section of the *Create Proxy Rigid Bodies* group now become enabled:



Click one of the buttons to create a new capsule/box proxy for the thigh bone. A new capsule/box shape will be created to fit around the bone, with Havok rigid body and shape parameter sets attached and initialized to a set of default values:

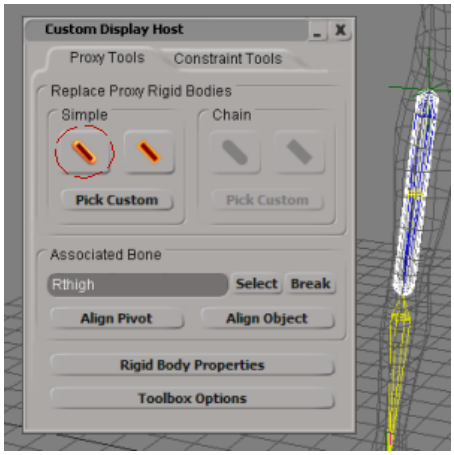
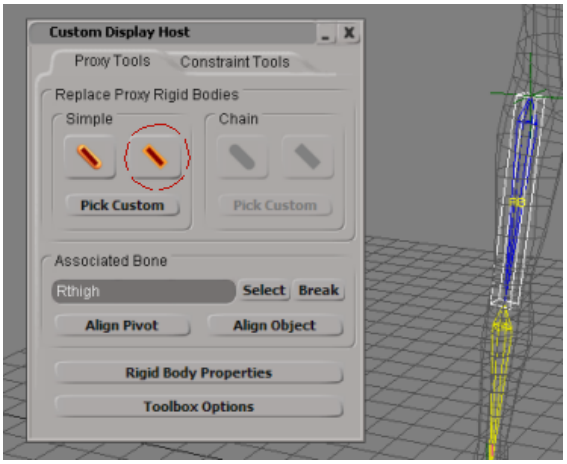
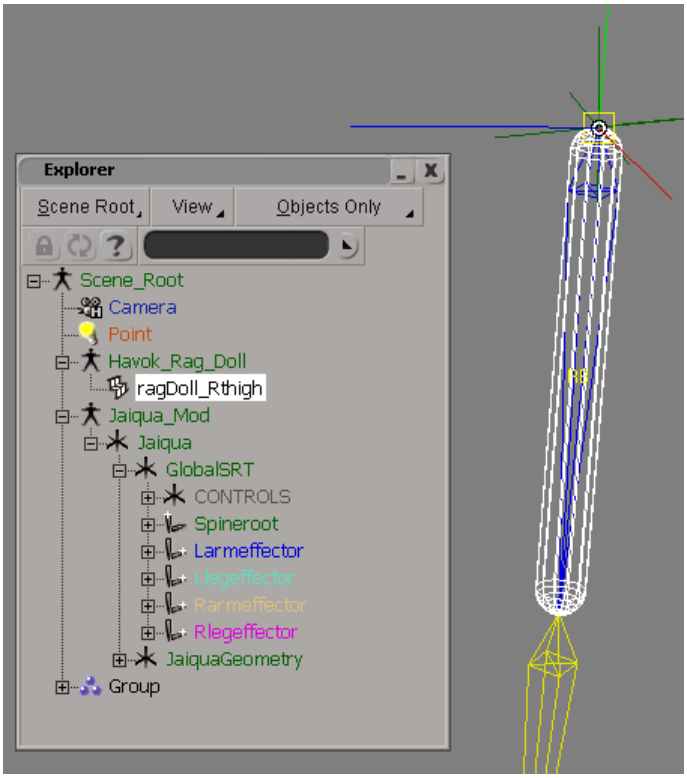
	

Table 6.7:

Once a proxy has been created, it is simple to replace it with a different type of proxy. Simply select the newly created proxy (if not already selected) and click of the same two buttons. This will replace the selected proxy, while retaining any rigid body parameters which may have been changed (or any constraints which may have been added). You can replace any type of proxy at any time.

Observe that whenever you create/replace a proxy, the new proxy is placed in both a new model and a new layer (both called `"RagDollProxys"` by default), so as to keep it separated from the original skeleton hierarchy (this behaviour can be modified in the Toolbox Options). Also observe also that the pivot point of any new proxy automatically becomes aligned to that of the associated bone. This is an important step, as it ensures that the mapping we will create during export processing is accurate:

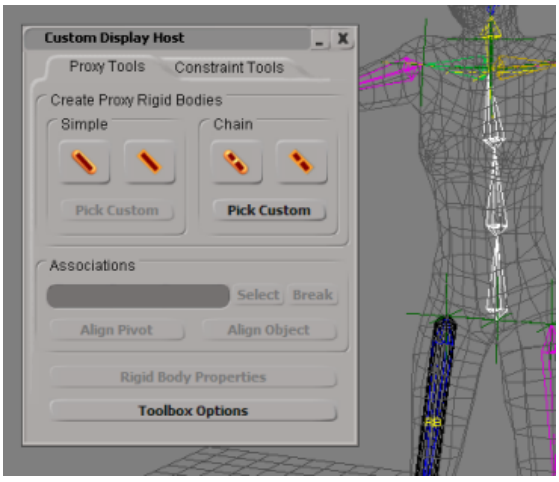
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Chain Proxies

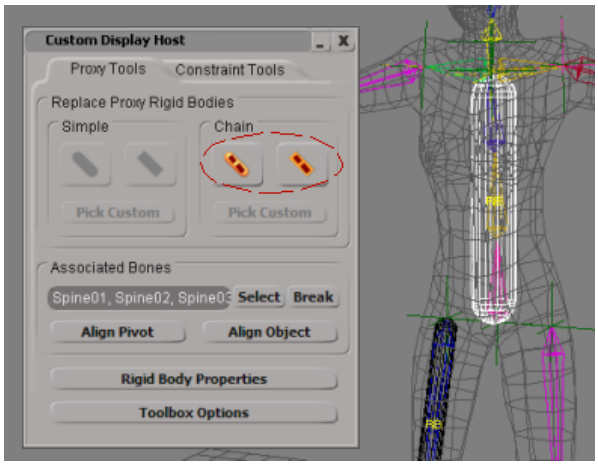
Chain proxies are those which represent a chain of two or more connected bones.

Take, for example, the spine in this example. Select the three bones representing the spine. When a valid chain of bones is selected, the buttons in the **Chain** section of the *Create Proxy Rigid Bodies* group becomes enabled, allowing chain proxies to be created:



Click one of the buttons to create a capsule/box proxy for the selected bones. Once a chain proxy has been created it can be replaced in the same way as simple proxies - by selecting the proxy and clicking the capsule/box button again:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



The only different between proxies created from a chain and proxies created from a single bone is that chain proxies keep a reference to both the start and end of the chain (while single proxies keep a reference to a single bone). Pivots of chain proxies are aligned to that of the first bone in the chain. For any other purposes, chain proxies operate the same way as simple proxies.

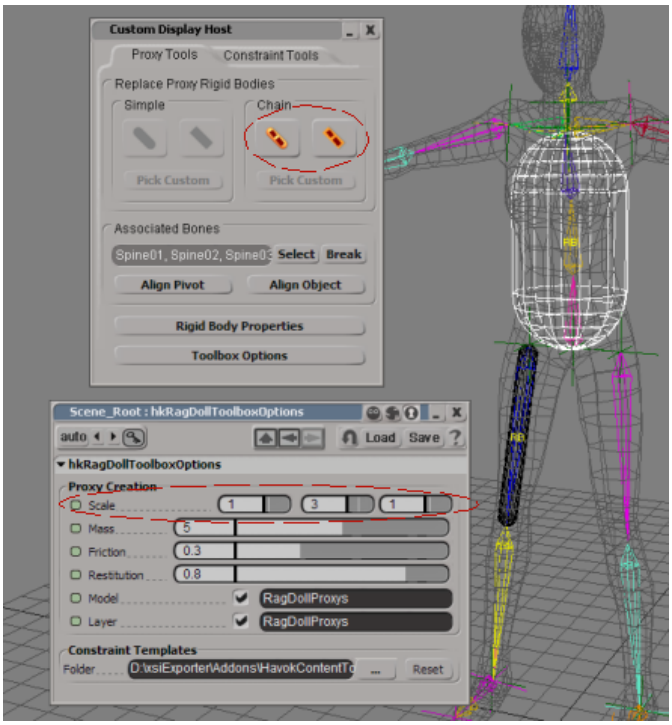
Reshaping/resizing Proxies

After a proxy has been created, it may not accurately represent the skin volume. In this case the proxies we have created so far are too narrow. We have two methods to correct this:

- Changing default proxy creation scale

  This is usually the easiest and safest way to resize the proxies. When creating a proxy, the Rag Doll Toolbox will use a size that bounds the selected bone(s), scaled by a user-defined scale value. You can change this scaling vale by opening the *Toolbox Options*:

  The Proxy Creation Options group contains a set of X,Y,Z scale parameters. This represents the scaling factor applied to the bone's bounding box before a proxy is created for it. For example, to re-create the spine proxy from a larger bounding box: select the proxy, set the scale values to 1, 3, 3 , and press the capsule/box button again:



Note:

The primary axis of XSI bones is X by default. For example, by leaving the X value at 1 above, the proxy length stays the same as that of the bone.

Tip:

It is usually desirable to have some overlap between proxy representations of adjacent bones. This avoids objects going through the gaps at joints during run-time.

- Using the standard SRT tools

  Any proxies can be transformed as per usual using XSI's Scale, Rotate and Translate tools. However it is important that non-uniform scale should be avoided so that the rigid bodies are created properly during export. It is also important to press the **Align Pivots** button (in the 'Association' section of the toolbox) after performing any transformation to ensure that the proxy pivot matches that of the bone. This helps the rag doll mapping filter to create accurate mappings.

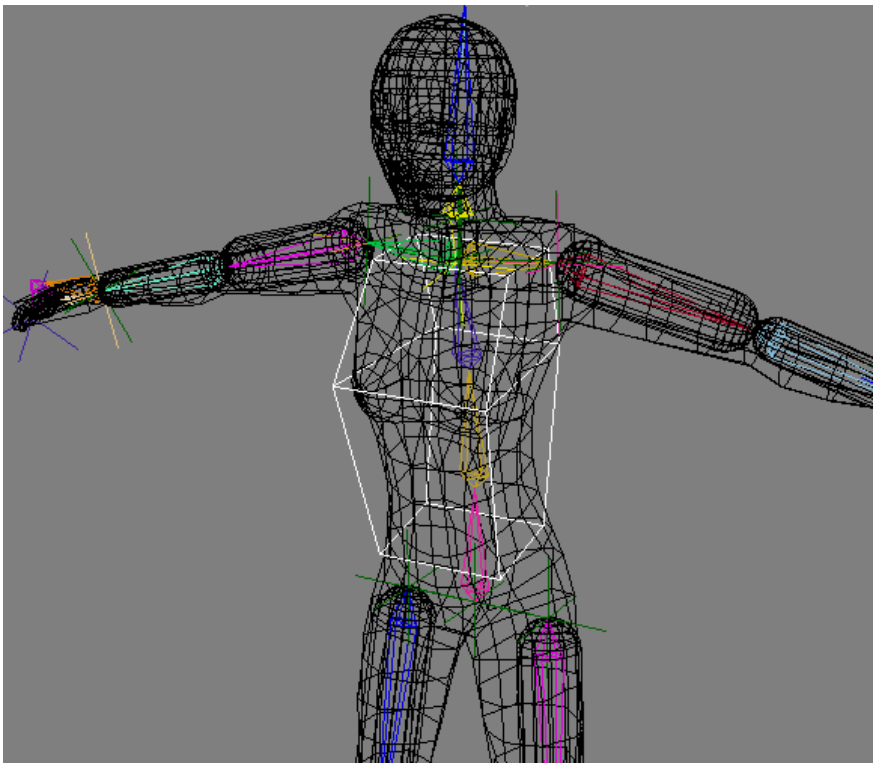
Experiment further with creating single and chain proxys for different parts of the character. Make sure that you are familiar with the general process before continuing.

Custom Proxies

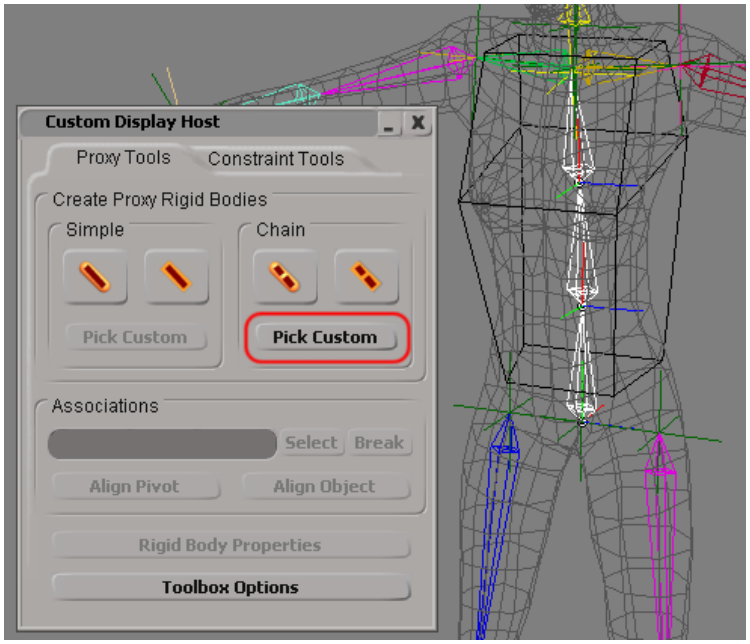
Sometimes a box or a capsule is not enough to accurately represent the volume around a bone or a chain of bones. For example, take the torso region of this model. We have already created a box/capsule proxy associated to the spine bones. Lets replace this with a custom proxy. Start by deleting the existing spine proxy.

Create a new convex polygon mesh in world space to represent the characters torso, as shown, using any of the standard XSI tools:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



To associate this mesh as a chain proxy for the spine bones, select the three spine bones, then click the **Pick Custom** button in the toolbox:



This will start a picking session - now pick your custom mesh.

The custom mesh now becomes associated as a proxy for the set of spine bones. From this point it

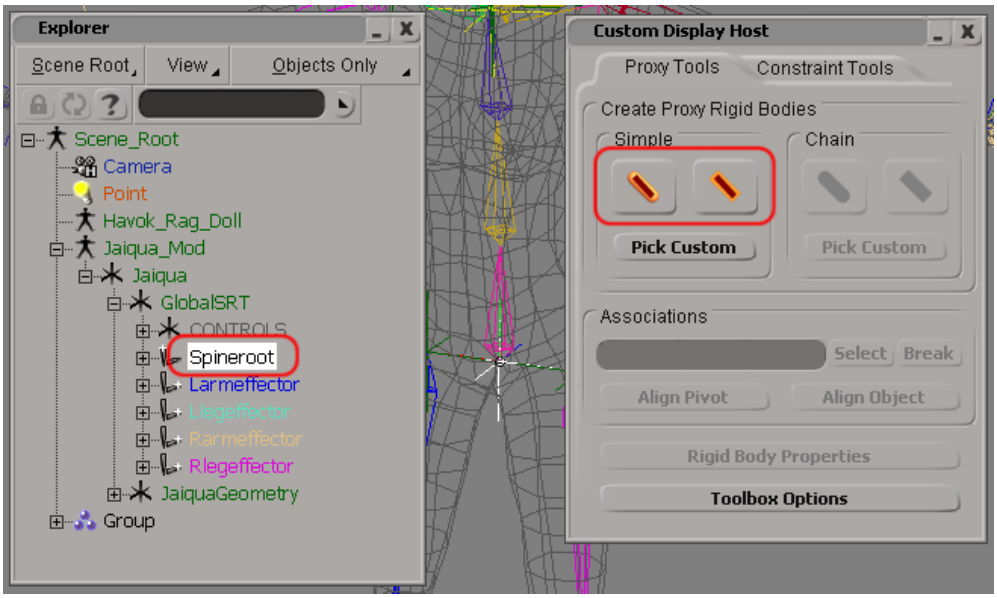
behaves like any other chain proxy - i.e: it inherits default Havok parameter sets; it can be replaced by clicking the other chain proxy buttons; it's pivot is aligned to the first of the associated bones, etc.

One final step is to ensure that the "shape type" parameter of the new proxy is set to "*Convex Hull*". This is required to accurately represent a mesh such as this. Open the *hkpShape* parameter set of the proxy and set the parameter accordingly.

Null-based Proxies

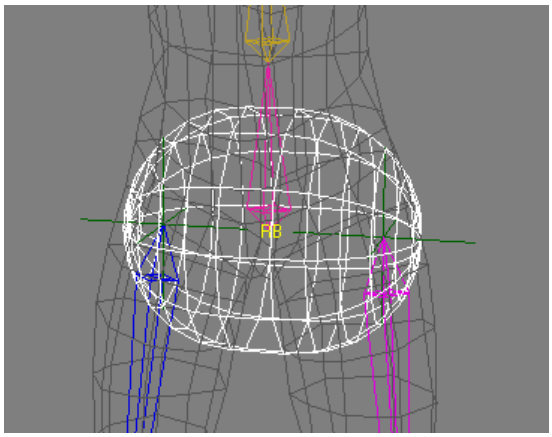
The scene we are working on does not have any specific 'pelvis' bone. However it is desirable for the rag doll representation to have a pelvis, both for realism and for mapping purposes. Using the following method, we can create a suitable proxy even though there is no suitable bone.

First, select the the root of the spine chain, called *Spineroot* - use the explorer if necessary. Then click the create capsule/box button:



The proxy is created by taking the bounding box of the selected null object combined with any null children that it may have - in this case it considers the spine root plus the two leg roots. Tweak the scale values and recreate the proxy until it has the required dimensions:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

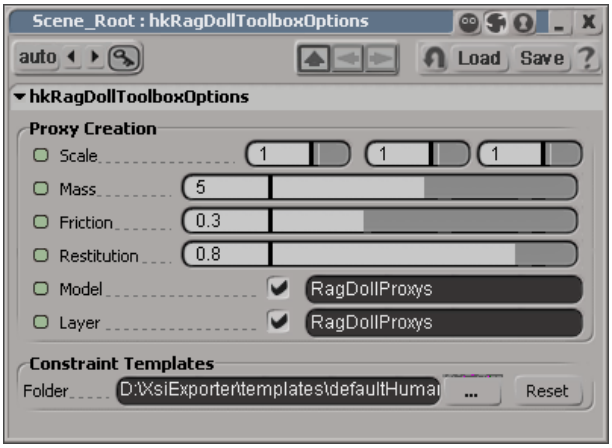


Note:

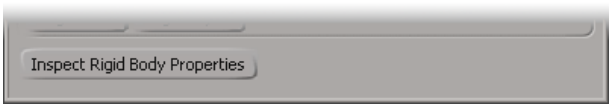
If the toolbox cannot determine a suitable bounding box for an object (eg. a single null object with no children), it will always assume a bounding box of unit dimensions.

Setting Rigid Body Properties

When a new proxy is created, it assumes a set of default parameter values. These defaults can be configured from the Rag Doll Toolbox Options, and are saved as part of the scene:



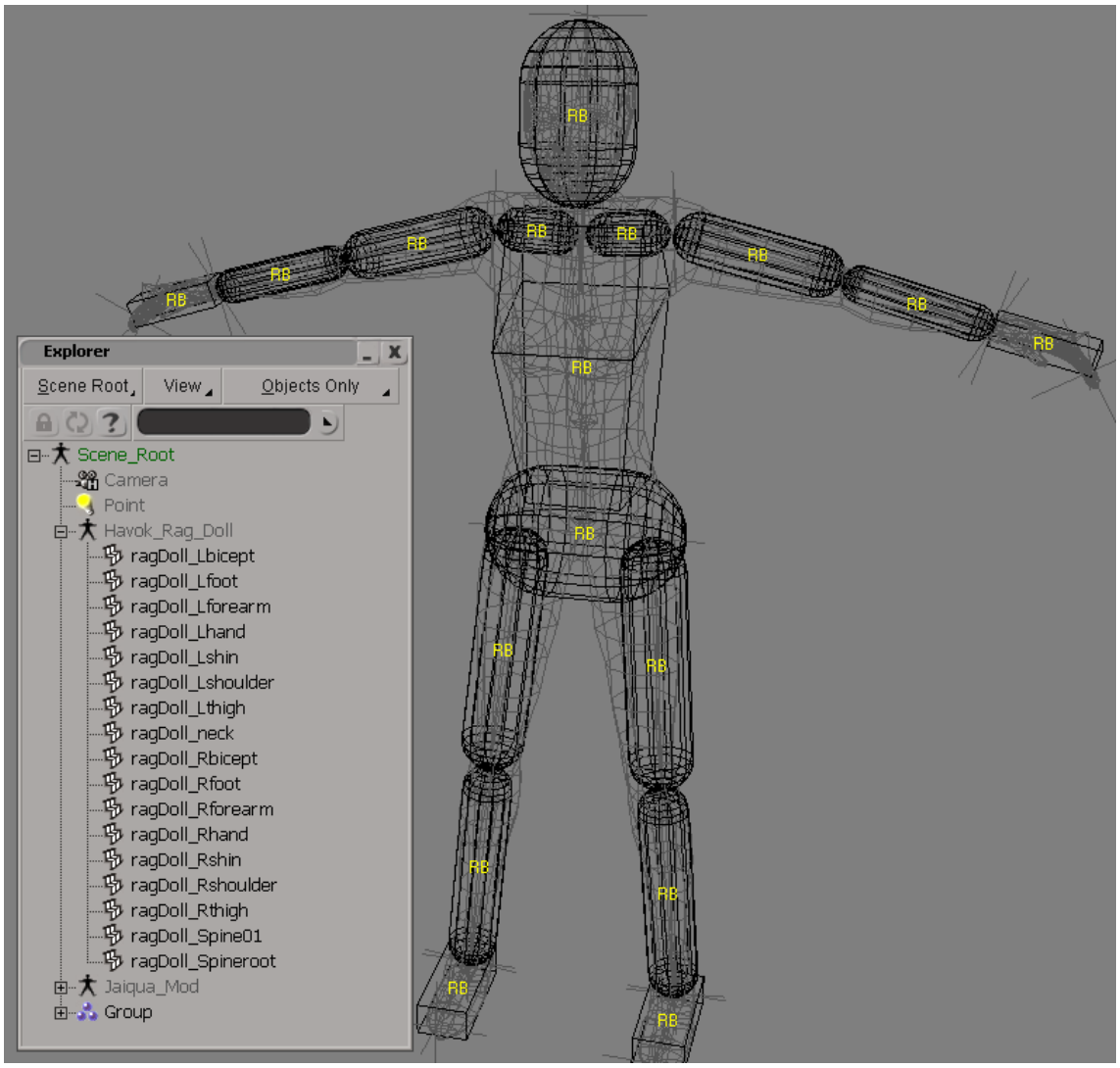
After a proxy has been created, it is simple to change the properties on an individual or group basis. Select one or more proxies, and click the *Inspect Rigid Body Properties* button in the Proxy Tools section. This opens a PPG from which the propertes can be edited as per usual.



Now set appropriate rigid body properties for each proxy that you have created.

Finishing with rigid bodies

Using the techniques above, you should now be able to create enough proxies to represent the character. The pivots of all your proxies should be aligned to the associated bone(s) - use the *Align Pivots* button on each proxy to be certain, and appropriate rigid body properties should be set for each proxy. The `"ragDollToolbox/tutorialProgress1.scn"` file contains a fully proxied character if you wish to use it:



The proxys in this scene were created as follows:

Simple proxies for:

- Thighs
- Calves
- Clavicles
- Upper arms
- Forearms

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- Pelvis (aka spine root)

Chain proxies (with additional transformation and re-alignment) for:

- Hands
- Feet
- Head/Neck

Custom proxy for:

- Spine

As you can see from the screenshot above, all of the proxies are grouped together in a model, but the do not (yet) have any parenting.

6.5.8.3 Creating Constraints

After creating the proxies, we are left with a flat collection of rigid bodies - they are not parented to each other, or connected in any way. To finish setting up the rag doll, we need to organize the rigid bodies into a constrained hierarchy. The *Constraint Tools* tab in the Rag Doll Toolbox gives us the tools to achieve this.

Note:
It is useful to hide/lock the bones at this point (if not already), since we will be working only with the proxies.

Templates

Templates are representations of Havok constraints, saved as text files. Each represents a particular type of constraint, plus information about its spaces, limits, etc.. so that it can be reused between different characters. For example, an "elbow" template might describe a hinge constraint with certain limits and spatial orientation.

Templates are stored together in an specific directory. Although a default set of templates (`defaultHuman`) is provided, in most cases you will be creating your own templates to match the structure of your characters.

You can change the current location of the templates folder at any time from within the Rag Doll Toolbox Options.

Note:
When working with templates, some assumptions need to be made regarding the local axis of the bones. In particular, the main axis (an axis pointing towards the length of the bone, in the "outside" direction) and the bend axis (an axis around which a positive rotation (using the right hand rule) causes a bending of the joint) need to be defined. These axes are currently assumed to be +X and Z, since this is the convention used by XSI's default skeletons.
If a proxy's transform does not follow this convention, then any applied templates will need to be rotated into the correct orientation.

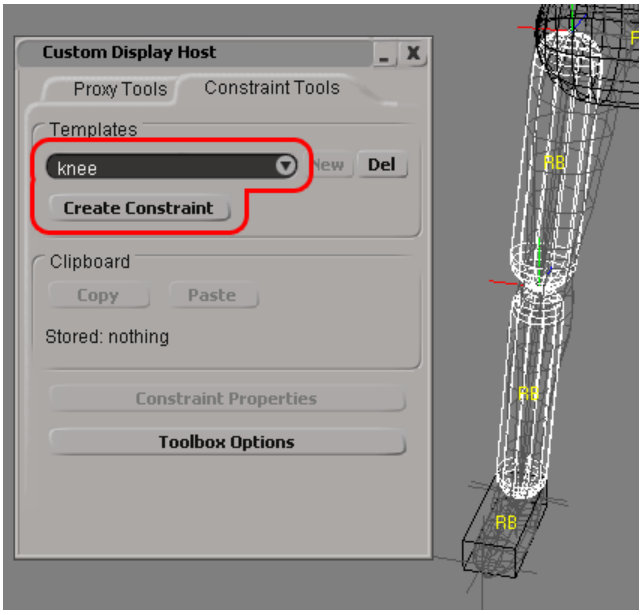
Applying Templates

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

When we apply a template to a pair of proxies, several things happen in order:

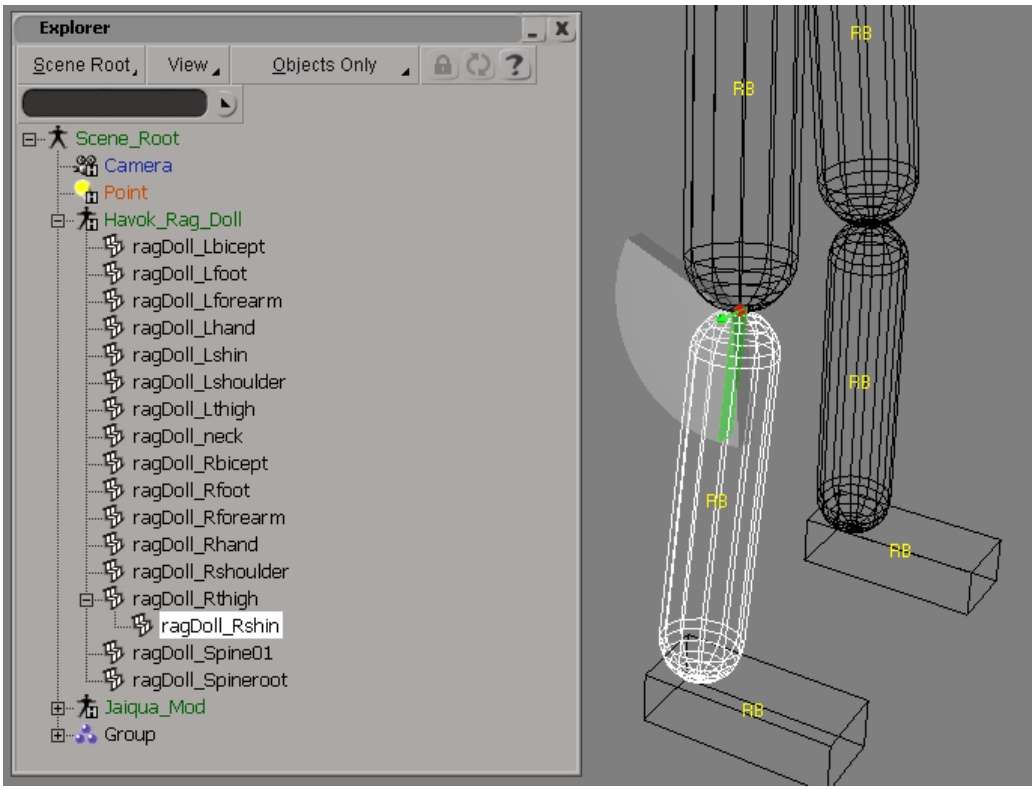
1. The two proxies become parented to each other, matching the parenting of their associated bones.
2. A constraint is added to the child proxy, based on type stored in the template.
3. The parameters of the constraint are set from those stored in the template.

For example, select a pair of thigh and calf proxies. In the templates combo box, select the "`knee`" template and then click on *Create Constraint*:



Observe the result. The calf proxy becomes a child of the thigh proxy, and a hinge constraint is created on the calf:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Replacing Constraints

After a template has been applied, it is simple to replace the existing constraint with a new template. Select either the pair of proxies or just the child proxy, then press the " **Replace Constraint** " button. Any existing constraint is deleted, and the same procedure occurs as when a new template is applied.

Modifing Constraints

After a template is applied, you can always modify the parameters of the constraint as per usual. Use the *Inspect Constraint Properties* button for quick access to the constraint parameters. You can save a constraint as a new template (or replace an existing one) at any time by clicking on the " **New** " button beside the template combo box.

Copy and Paste

Symmetric characters usually have similar constraint properties on their left and right side. The copy and paste mechanism of the toolbox is useful in this case.

With a constrained proxy selected, press the "copy" button to store its constraint as a template in memory. The constraint can then be applied to:

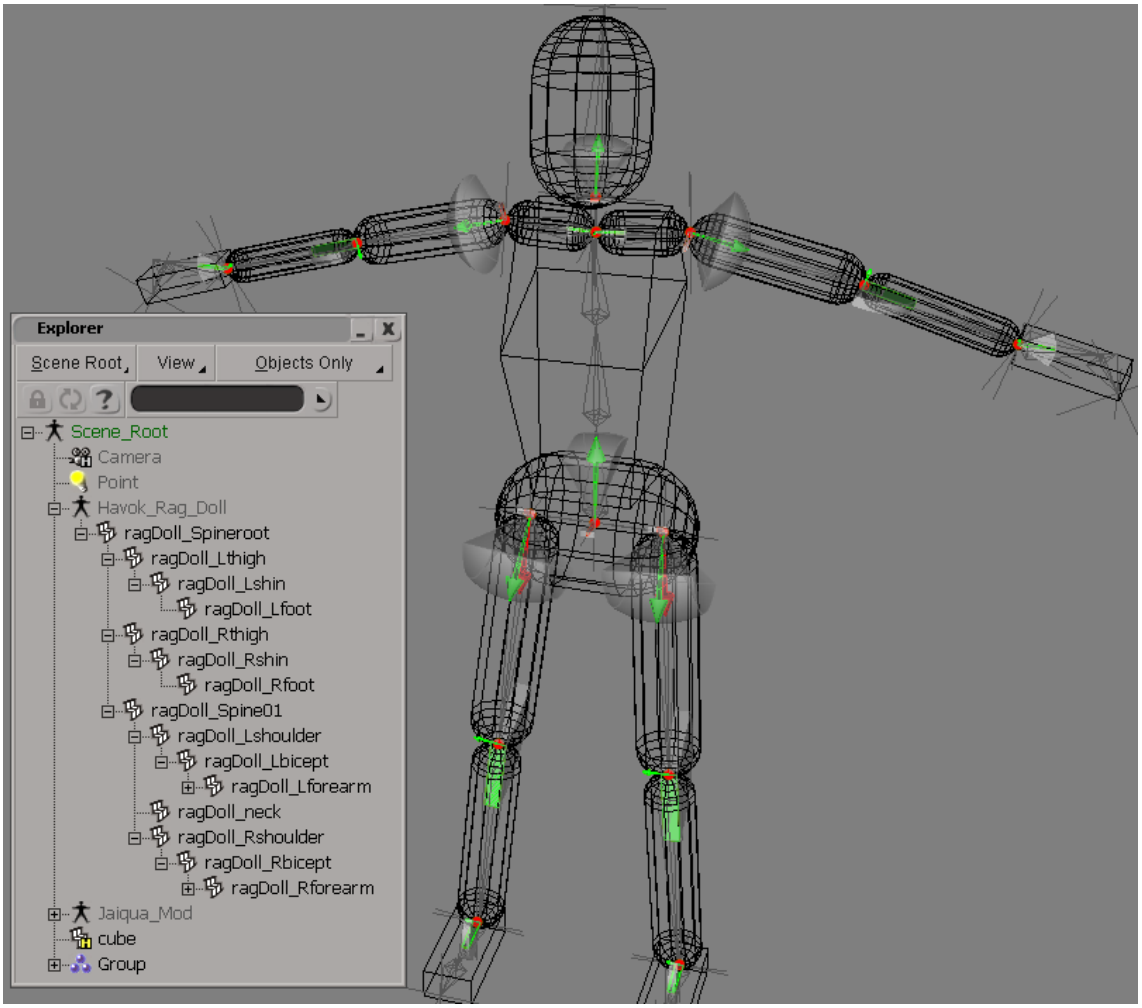
- A pair of unconstrained proxies, by selecting the pair and pasting. This behaves in the same way as creating a new joint.
- An already constrained proxy, by selecting the proxy and pasting. This behaves in the same way as replacing an existing joint.

For example, let's say we have modified the left knee friction torque, and we now want to apply the same constraint with the same values to the other knee:

- Select the calf, and change the **Max Friction Torque** parameter value of the constraint
- Click on **Copy** in the Constraint Tools Clipboard
- Select the opposite thigh and calf together in the viewport
- Click on **Paste** in the Constraint Tools Clipboard
- Observe that a new constraint has been created/replaced, with the same parameter values as the copied constraint

Finishing with Constraints

Continue applying templates to each pair of proxies, using the default templates provided (feel free to modify any parameters). You may wish to open the sample `"ragDollToolbox/tutorialProgress2.scn"` file - this contains the skeleton, the skin, and the constrained proxy hierarchy. It also contains a simple floor so that the rag doll has something to land on when we preview the scene:



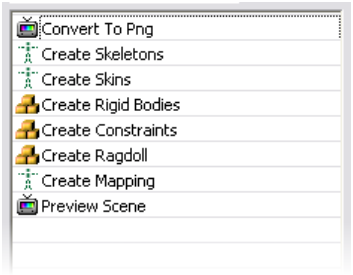
Observe that the proxies now form a hierarchy, matching that of the bones with which they are associated.

6.5.8.4 Processing the Scene

Now that we have the rag doll set up to match the skeleton in the scene, we want to process so that:

- Runtime versions of the original skeleton and skin are generated
- The rigid bodies and constraints for the ragdoll are generated
- A "ragdoll instance" and skeleton for the ragdoll are generated
- A pose mapper between ragdoll skeleton and original skeleton is generated (so the ragdoll can eventually drive the skin)
- We can preview it

Export the scene to the filter manager using the Havok exporter. The following filter setup is required to process the ragdoll:



- **Convert To PNG**

  This embeds the textures into the scene data, so that we can view them in the preview.

- *Create Skeleton*

  This will create a skeleton instance from objects in the the scene data. Choose the '*Automatic*' setting - this will automatically create a skeleton from the entire 'Jaiqua_Mod' model.

- **Create Skin**

  This will create a skin object and associate it with the skeleton we just created.

- **Create Rigid Bodies**

  This will detect the rigid body and shape properties in the scene and create the actual rigid body objects for the simulation.

- **Create Constraints**

  This will create actual physical constraints based on the setup we defined in the scene.

- **Create Rag Doll**

  This will create a Ragdoll instance based on the rigid bodies and constraint we just created. Choose the ' **Automatically** ' setting, which works fine in this case as there is only one rag doll in the scene.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

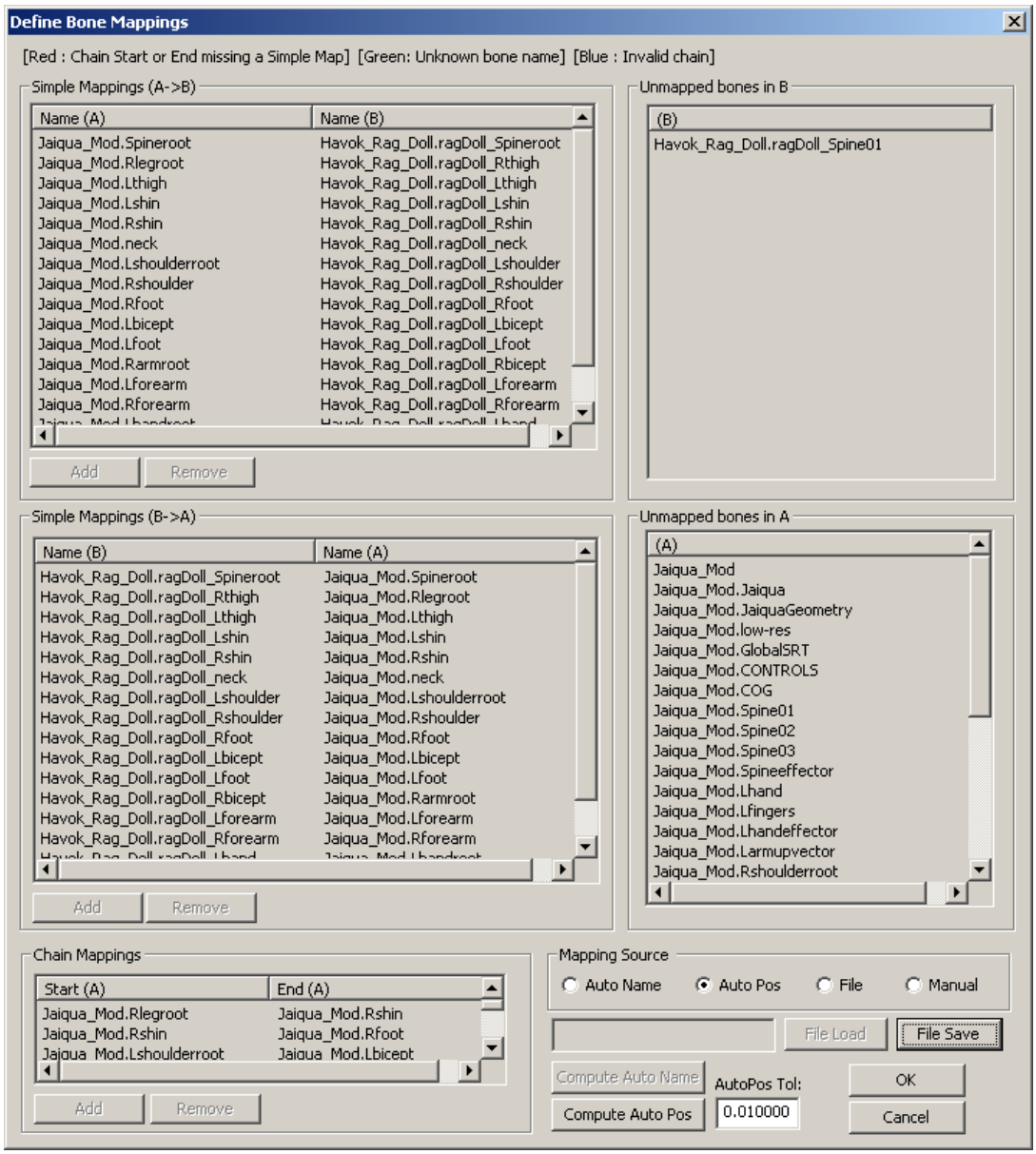
- **Create Mapping**

  This filters sets up skeleton mapper information that allows the transformation of poses between two different skeletons (in our case, the original boned skeleton and the Rag Doll skeleton).

  For ' **Skeleton A** ', pick 'Jaiqua_Mod' (the skeleton root).

  For ' **Skeleton B** ', pick 'Havok_Rag_Doll.ragDoll_Spineroot' (the rag doll root).

  Click on ' **Define Mappings** ' to examine the mapping:



  The default behaviour of the mapping filter is to match the skeletons by comparing the positions of the bones in each. This is why it was important to ensure that the pivot points of the rigid bodies were aligned to the pivot points of the joints.

  For a successful mapping, all of the bones in one skeleton should be mapped to (some of) the bones in the the other skeleton. We can see that this is not currently the case. We will fix this later.

  Click ' **OK** '.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

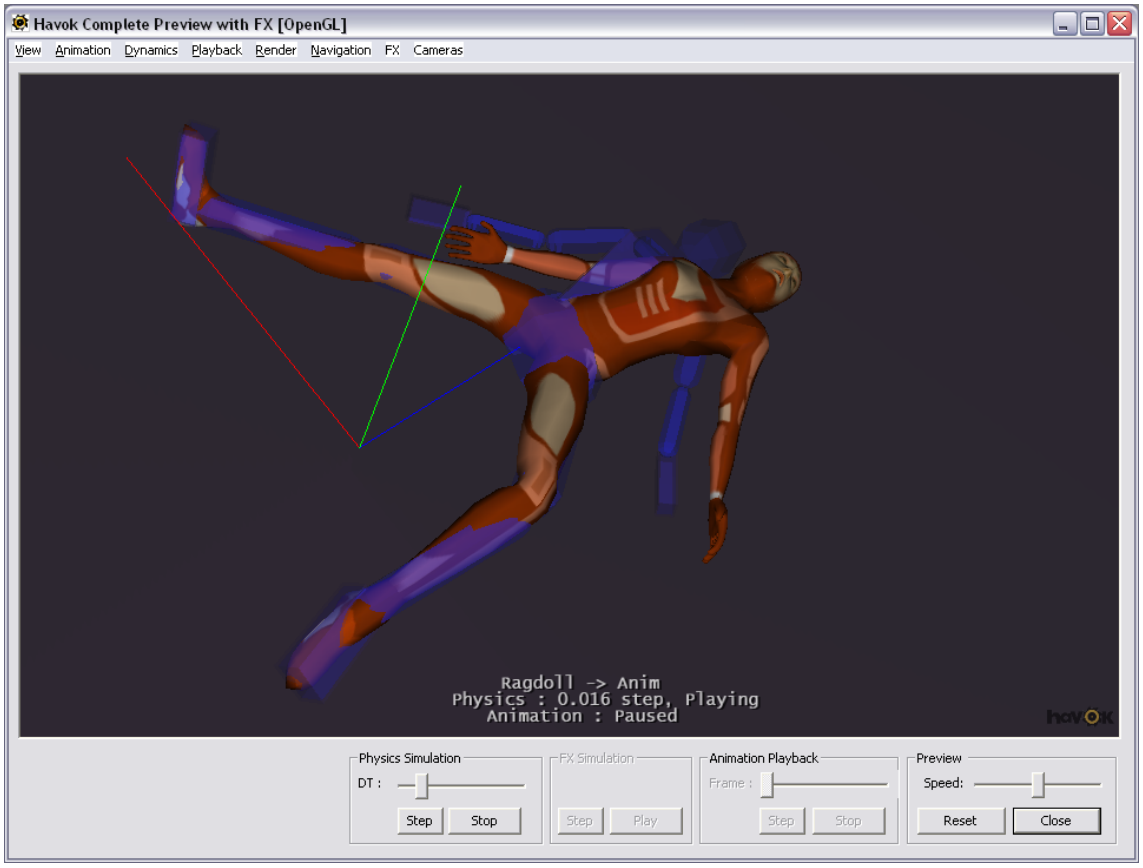
- **Preview Scene**

  So that we can interactively test the rag doll behaviour.

Fixing the Mapping

Run the filter setup above. When the preview window appears, ensure that *Display Meshes* and *Physics Ghosts* are enabled in the 'View' menu.

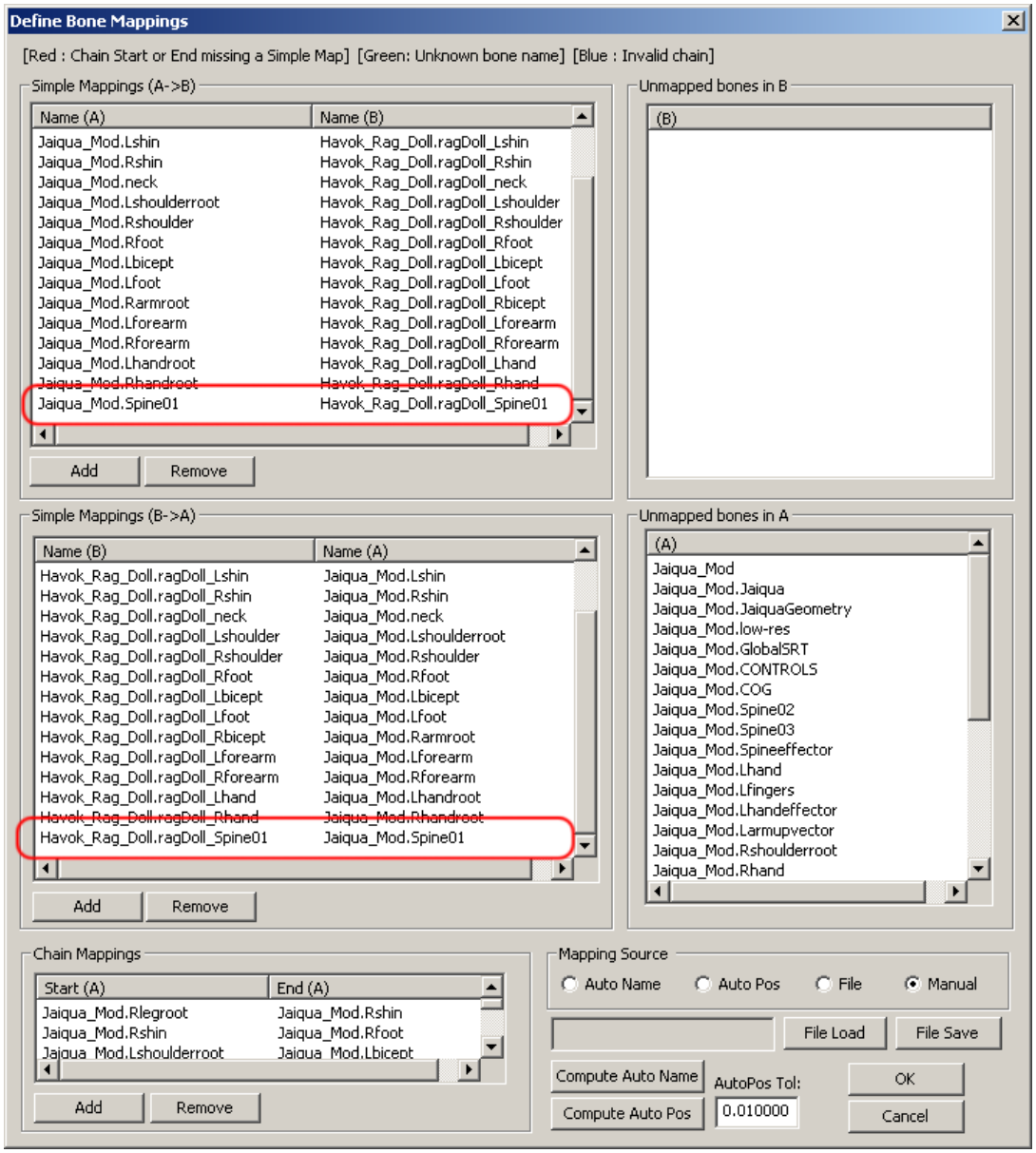
Play the physics and interact with the rag doll using the space bar to pick and drag the rigid body shapes. Observe that the character's upper body does not follow the rag doll correctly:



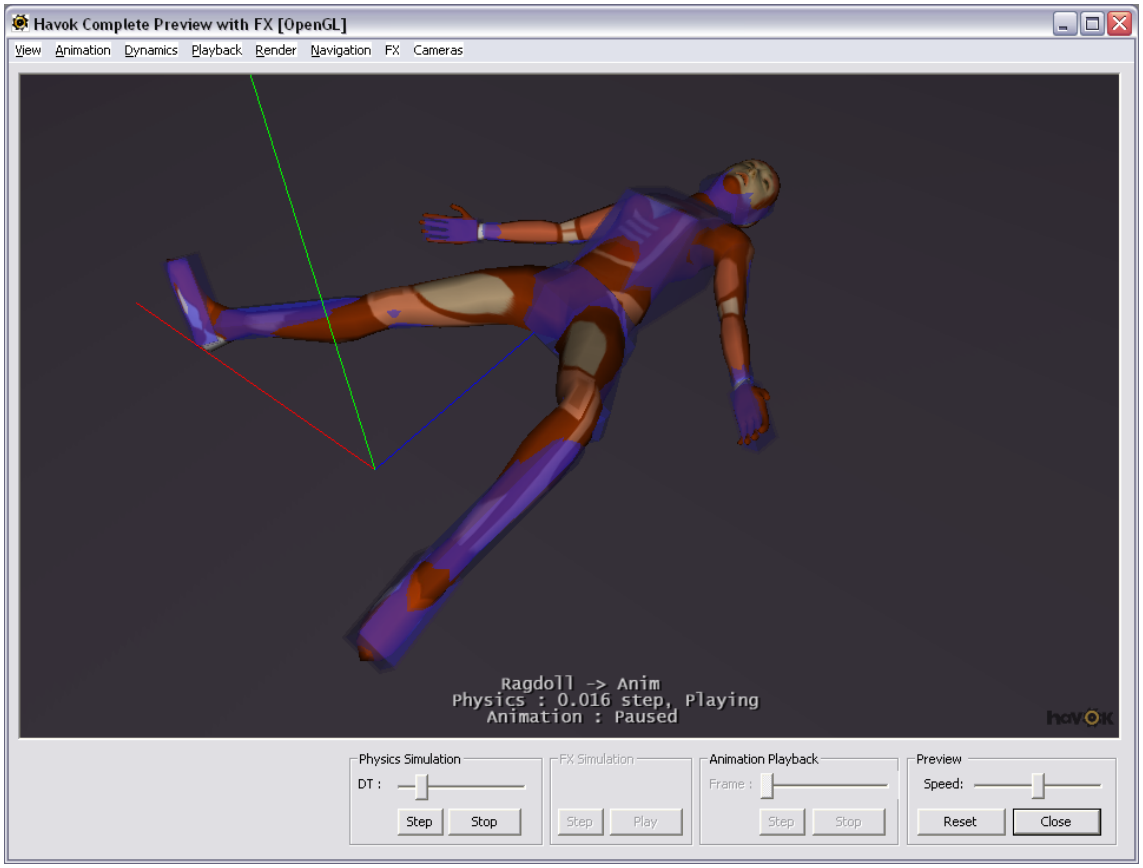
To fix this, close the preview and return to the 'Define Mappings' dialog in the 'Create Mapping' filter.

Observe that *Spine01* is unmapped in both directions. This happened because *Spine01* (the first bone in the spine) and *Spineroot* (the root of the spine) both have the same position - this confuses the mapper when set to 'Auto Pos', and leaves *Spine01* unmapped. Therefore we need to manually add this mapping:

- Switch the 'Mapping Source' to '*Manual*' - this allows us to edit the mapping.
- In the upper mapping section (A->B) press the '*Add*' button, and add a mapping from *Jaiqua_Mod.Spine01* to *Havok_Rag_Doll.ragDoll_Spine01*.
- In the lower mapping section (B->A) press the '*Add*' button, and add a mapping from *Havok_Rag_Doll.ragDoll_Spine01* to *Jaiqua_Mod.Spine01*.



Click '*OK*', and run the filter setup again. Now observe that the rag doll maps correctly to the character:



Note:

Another solution in this case is to use the *Auto Name* mapping option instead of Auto Position. This compares the last element of the bone names in each skeleton (delimited by underscores/dots/space) and builds the mapping based on any matches, which works well in this scene. Try it to see for yourself.

Tweaking the Scene

The rag doll setup is now complete. At this stage it is useful to return to XSI and begin to tweak parameter values. Try altering the Rigid Body masses as well as the Constraint limits and frictional values. You can run the filter setup at any time to test the result.

The `"ragDollToolbox/tutorialEnd.scn"` file contains the final tweaked rag doll. Note that a scene transform filter has also been added to the export pipeline to scale the character to a realistic size of ~1.5m.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.6 Common Concepts

6.6.1 Introduction

This chapter contains information regarding basic concepts which are used by the Havok Content Tools, independant of the actual modeler being used to create the content.

6.6.2 Physics: Rigid Body Concepts

Rigid bodies are the basic building blocks of Havok simulations for an object to be simulated, it must be a rigid body.

A simulated rigid body contains both dynamics (rigid body) and collision (shape) information. The Havok Content Tools separate these two elements into distinct objects for use in each supported modeler. If a Create Rigid Bodies filter is applied during the export process, any rigid body information found in the scene is used to define the dynamics of a new runtime rigid body object, and any related shape information is used to define the collision properties of that rigid body.

Each modeler supported by the Havok Content Tools allows rigid body and/or shape information to be attached to any object in the scene. The means in which this information is attached depends upon the modeler: 3ds max attaches the information through the use of modifiers, Maya attaches it through the use of nodes, while XSI attaches it though the use of custom parameter sets.

It is important to note that both rigid body and shape information is required for an object in the modeler for the Create Rigid Bodies filter to create a runtime rigid body object on export. If rigid body information exists without related shape information then no rigid body can be created. Similarily, if shape information exists but is unrelated to any rigid body information, then it is ignored. The shape information may be placed in the rigid body object's children as well or instead of in the rigid body object itself, which allows for the creation of compound/proxied rigid bodies.

This section describes each of the rigid body and the shape properties as used in any supported modeler, and how they can be combined to create various types of runtime rigid body objects.

6.6.2.1 Rigid Body Properties

A Havok rigid body created in a modeler contains the following basic set of properties which govern the dynamic behaviour of the exported body at runtime:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Mass	The mass of a rigid body governs how the object will interact with other objects. If the mass is set to zero, the resulting rigid body will be fixed in space.
Friction	The coefficient of friction for the rigid body's surface. This affects its motion relative to surfaces it comes into contact with. Although this is actually a pair-wise coefficient, friction is specified on a per rigid body basis. When affecting the relative motion of two surfaces, the surfaces' friction coefficients are combined to produce a coefficient for the interaction. Usual values for Friction are between 0 and 1. Notice that values greater than 1, although physically invalid, are allowed.
Restitution	The coefficient of elasticity for the rigid body. This governs the effect that collisions have on the velocities of colliding rigid bodies. Like Friction, this is a pair-wise coefficient - when two bodies collide, their elasticity values are combined to produce a coefficient for the interaction. Usual values for Elasticity are between 0 and 1. Notice that values greater than 1, although physically invalid, are allowed.
Center of Mass	The center of mass of a rigid body defines it's local balance point. This is normally calculated automatically.
Inertia Tensor Proportions	The inertia tensor is normally a representation of the mass of a rigid body and how that mass is distributed throughout the body. Since mass is provided as a distinct property of these rigid bodies, the inertia tensor in this case is used only to specify the proportional distribution of that mass. This is normally calculated automatically.

Table 6.8: Basic Rigid Body Properties

In addition to these basic properties, there are advanced properties which most users will not want to alter from the default, but which are available for finer control if needed (see chapters Havok Dynamics and Collision Detection for more details on their effects):

Max Linear Velocity	The maximum allowable linear velocity of the rigid body in m/s.
Max Angular Velocity	The maximum allowable angular velocity of the rigid body in degrees/simulation frame.
Linear Damping	The linear damping to be applied to the rigid body, in units of inverse seconds. The linear velocity of the body then decays exponentially in time with rate constant given by the damping value.
Angular Damping	The angular damping to be applied to the rigid body, in units of inverse seconds. The angular velocity of the body then decays exponentially in time with rate constant given by the damping value.
Allowed Penetration	The maximum allowed penetration depth for this object.
Quality Type	The quality type governs general features of the behaviour of the rigid body on interaction with other bodies. See the Continuous Physics chapter for more information. Options are "Fixed", "Keyframed", "Keyframed Reporting", "Debris", "Moving", "Critical", "Bullet". Note that when "Fixed", "Keyframed" or "Keyframed Reporting" are seleted, the "Mass" control is disabled since the body is effectively fixed.
Solver Deactivation	Allows you to enable an extra single object deactivation schema. That means the engine will try to deactivate single objects if those objects get very slow. This does not save CPU resources, though it can dramatically reduce slow movements in big stacks of objects. Options are "Off", "Low, "Medium", "High".
Deactivator Type	Determines which mechanism Havok will use to classify the rigid body as deactivated. Options are "Spatial", "Never".
Collision Filter Info	This (integer) value can be used by collision filters to identify the entity - for example, if a group collision filter is used, this value would specify the entity's collision group.

Table 6.9: Advanced Rigid Body Properties

Overriding the Center of Mass / Inertia Tensor

The values of the "Center of Mass" and "Inertia Tensor" properties of a rigid body are normally determined automatically by the Create Rigid Bodies filter. This is done by gathering the shape information related to a rigid body and integrating over its volume, which results in accurate and realistic values for objects with uniform density.

However, it may be desirable to specify a specific value for either of these properties from within the modeler. In each supported modeler, options are provided to override each of the center of mass and the inertia tensor. If 'override' is enabled, then the filter will use the value specified in the modeler instead of automatically calculating a value.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.6.2.2 Shape Properties

Shapes work together with a rigid body to define the collision behaviour of (a part of) the body. A rigid body can use a single shape or it can use several shapes grouped together (these are known as *compound* rigid bodies).

Shape Type	This specifies the type of shape which should be used for runtime collision detection. An appropriate runtime shape of this type is created by the Create Rigid Bodies filter, by determining the closest possible match to the object's geometry.
Extra Radius	Specifies an additional collision radius to be used by the shape - this can be useful in optimizing the simulation speed at runtime. The interpretation of this radius depends upon the Shape Type: For convex bodies this specifies the convex radius, for meshes this specifies the per-triangle radius.

Table 6.10: Shape Properties

Modeler vs. Simulation Shapes

The Havok Content Tools allow you to specify a shape type to be created from a given a mesh in a modeler. That means that, for the physical simulation, you generally use a different representation for the object to the actual representation in the modeler. For example, you can simulate a car chassis as a box, a head as sphere, or a forearm as a capsule. The advantage of this is that a simpler representation (box, sphere, cylinder) will be simulated faster than the original mesh models could be, with no noticeable changes on behaviour. Another advantage is that it is more memory-efficient to represent these simple objects (eg. a sphere is just a center and a radius) than it is to represent a whole mesh. Therefore, it is important to carefully select a shape type for each shape in your scene.

The following list presents the different kinds of Havok shapes, ordered by their complexity (simpler shapes will be simulated faster):

1. Sphere (simplest)
2. Capsule
3. Box
4. Convex Hull / Cylinder [6]
5. Mesh (most complex)

Notice that this is only an indication. Complexity of the simulation also depends on many other variables - whether the object is fixed or not, whether objects are in contact or proximity, relative size, etc..

Shapes used by run-time rigid bodies can not change throughout a simulation. If an object whose geometry changes over time is used to define a shape in the modeler, then only its geometry at the initial export frame is used to define the exported shape.

Convex vs. Concave Objects

A shape is defined as *convex* if, given any two points inside the object, you can always go in a straight line from one to the other without leaving the object. Convex objects include spheres, cylinders, and boxes. On the other hand, a donut or a cereal bowl are *concave*.

[6] Cylinders may have similar overheads to their low-tesselation hull equivalent, but their representation is more accurate (they roll) and, being implicit shapes, require less memory.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

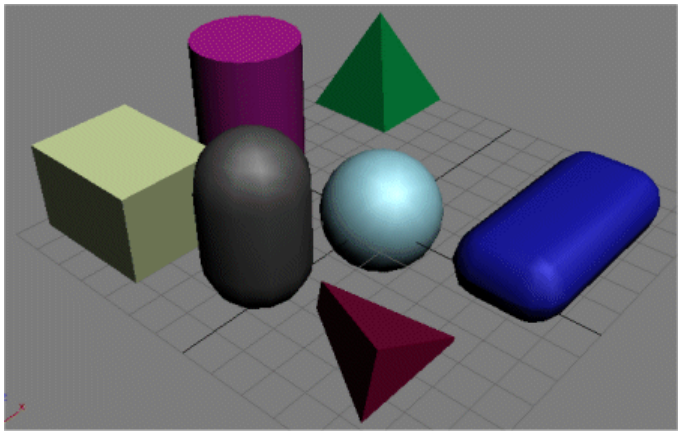


Figure 6.8: Examples of Convex Objects

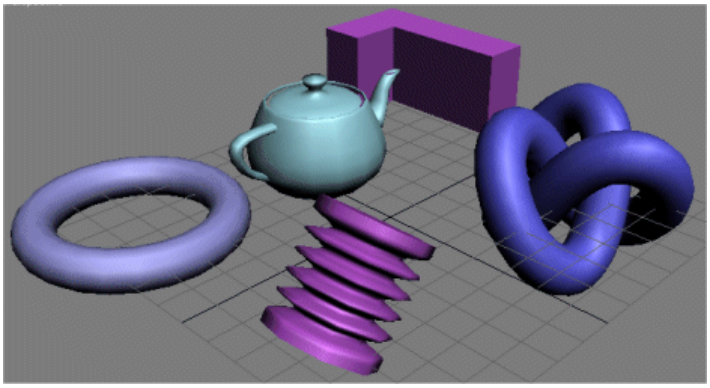


Figure 6.9: Examples of Concave Objects

Concave movable objects are inefficient for runtime simulation. The Create Rigid Bodies filter can (and by default, will) construct convex shapes from any concave meshes. The only shape type which will replicate the polygonal surface of concave objects is the **Mesh** type.

In many cases, you can approximate a concave object by combining multiple convex objects. For example, the L-shaped object in the picture above could easily be simulated as a compound rigid body with two box shapes. Check the compound rigid bodies section for more details.

6.6.2.3 Simple Rigid Bodies

Simple rigid bodies are those in which both rigid body and shape information are attached to the same object:

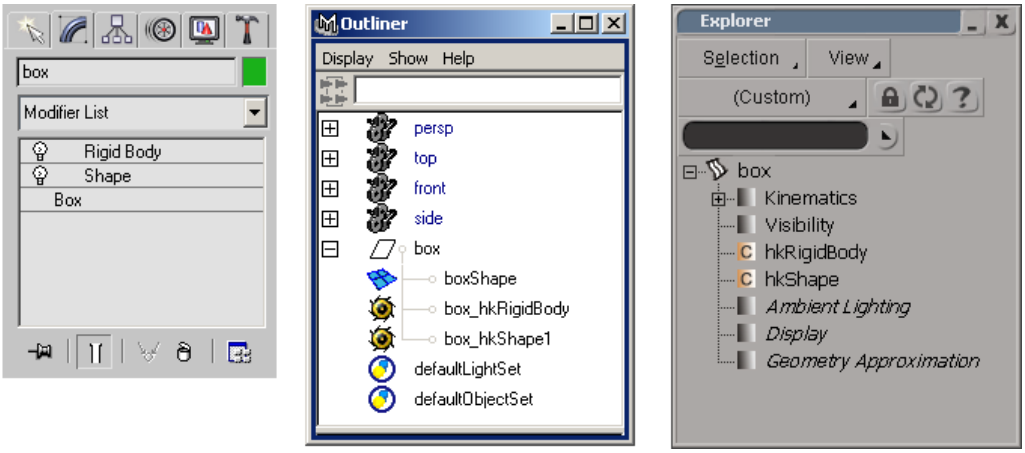
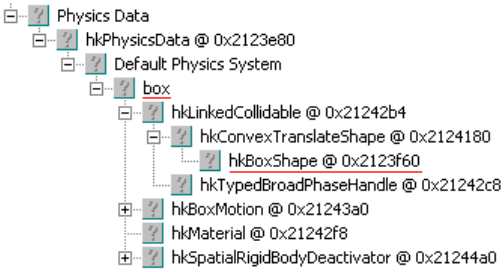


Figure 6.10: Simple Rigid Bodies

In this case the modelers have attached rigid body and shape information to the same object, named 'Box' - in 3ds max they are attached as modifiers, in Maya, they are attached as nodes, and in XSI they are attached as parameter sets.

On export, a Create Rigid Bodies filter will create a runtime rigid body object named 'Box', with a single box shape based on that of the box's mesh. This can be seen from the resulting XML (using a View XML filter):



Note:
The shape type of new shape modifiers/nodes/parameter sets is set automatically by the Havok Content Tools. In this case it has automatically been set to 'Box', since the shape information is attached to a box mesh in each modeler. Also, a non-zero mass has been specified in the rigid body information (or the exported body will always be fixed).

6.6.2.4 Compound Rigid Bodies

Compound rigid bodies are those in which rigid body and shape information are attached to some object, and additonal shape information is attached to some children of that object:

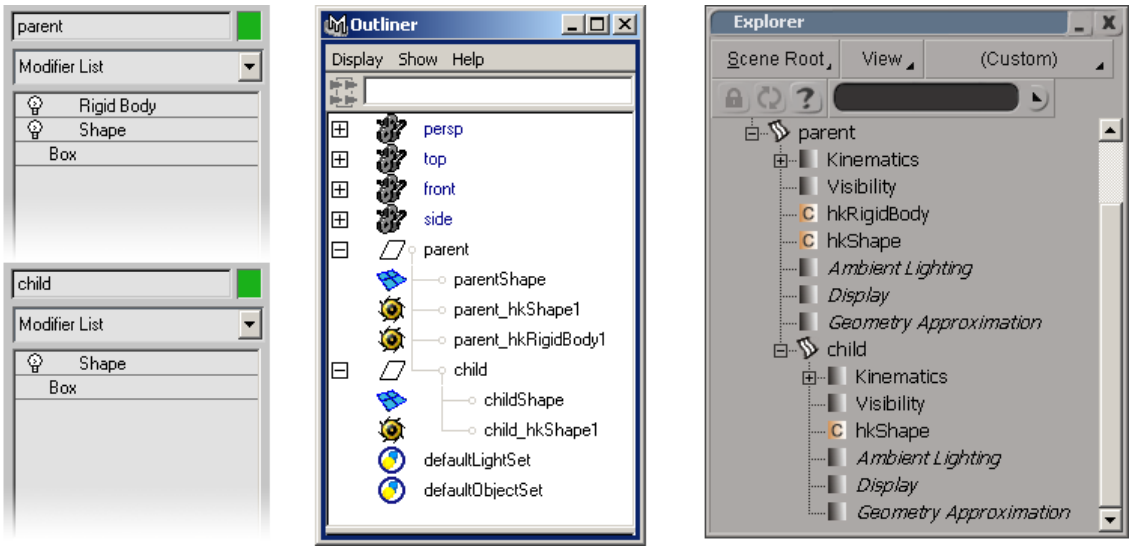
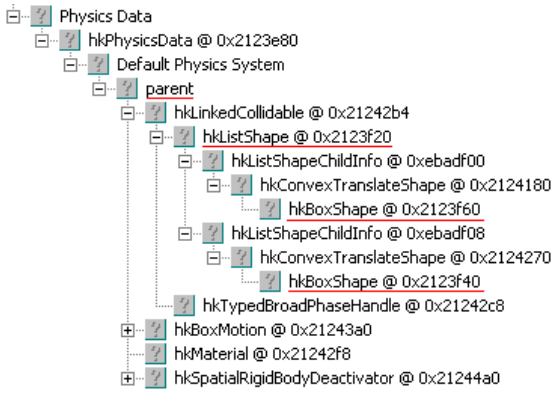


Figure 6.11: Compound Rigid Bodies

On export, a Create Rigid Bodies filter will in this case create a single runtime rigid body object (called 'parent') which contains a list of (2) box shapes:



One of the `hkpBoxShape`'s above is generated from the parent mesh, while the other is generated from the child mesh.

6.6.2.5 Rigid Bodies with Proxies

The final, and most unusual, type of rigid body is one in which only rigid body information is attached to some object, and only shape information is attached to some of its children:

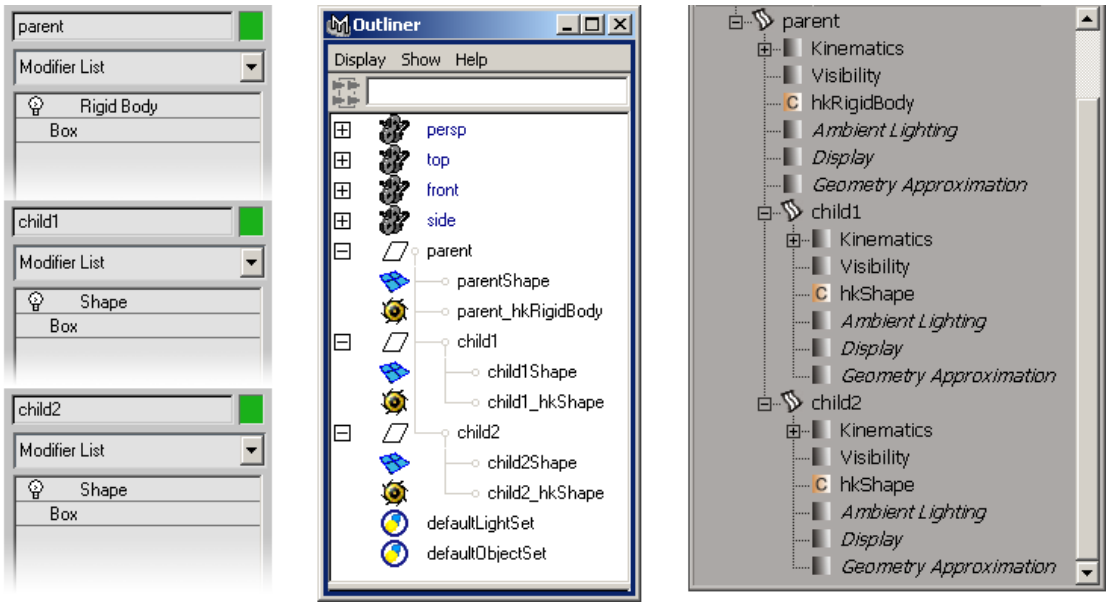
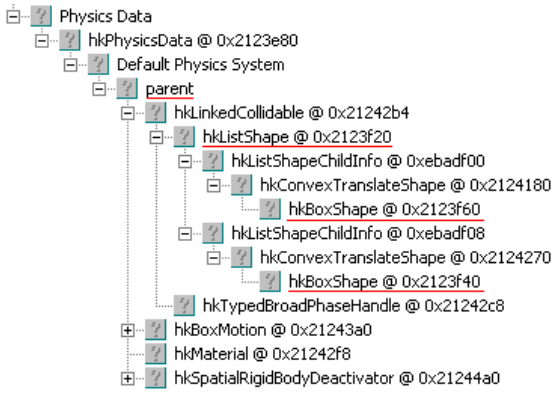


Figure 6.12: Rigid Bodies with Proxies

On export, a Create Rigid Bodies filter will again create a single runtime rigid body object (called 'parent') which contains a list of (2) box shapes:



In fact, the resulting Havok rigid body structure is the similar to that of the compound rigid body case. However, in this case both `hkpBoxShape`'s are generated from the child meshes - no shape is generated from the parent mesh.

6.6.3 Physics: Constraint Concepts

6.6.3.1 Constraint Spaces

Setting up physical constraints between rigid bodies is the next logical step once rigid bodies have been created.

The most important concept associated with constraint setup in the Havok Content Tools is that of

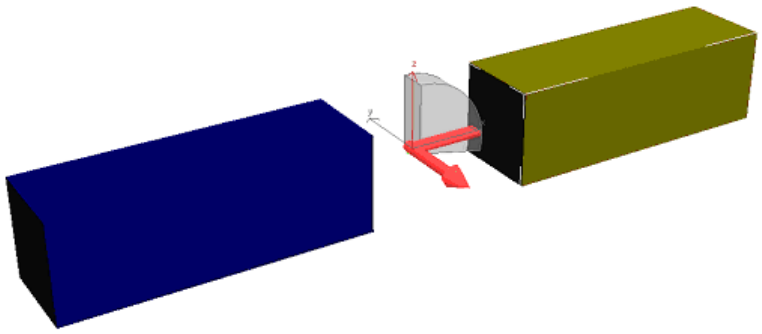
constraint spaces. In this section we will explain this concept and how is it used during constraint setup.

The Create Constraints filter is based on these concepts, and uses them to convert constraint definitions in a modeler into runtime constraint objects during the export process.

Parent and Child Rigid Bodies

Constraints operate symmetrically: they affect two rigid bodies, and since movement is relative, it can be argued that there is no distinction of the roles of each rigid body. This is very clear, for example, in a spring connecting two rigid bodies.

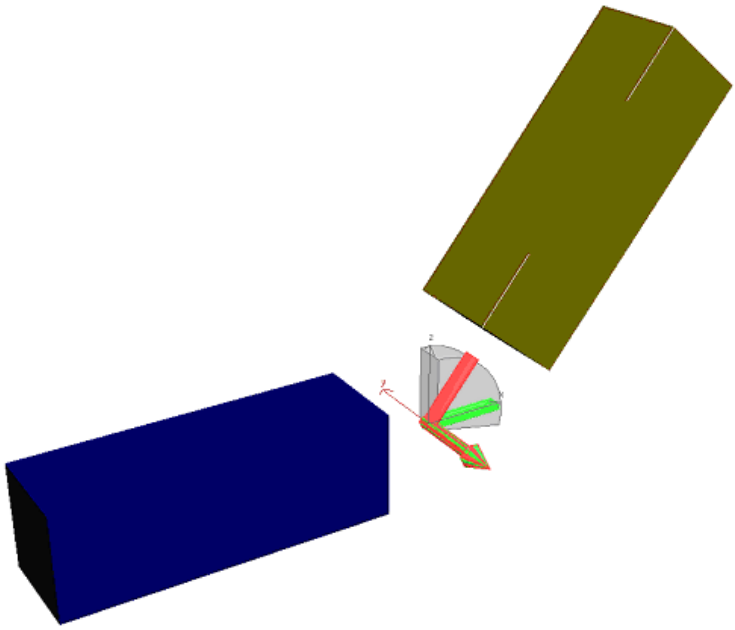
However if we consider a hinge, we can begin to make a distinction:



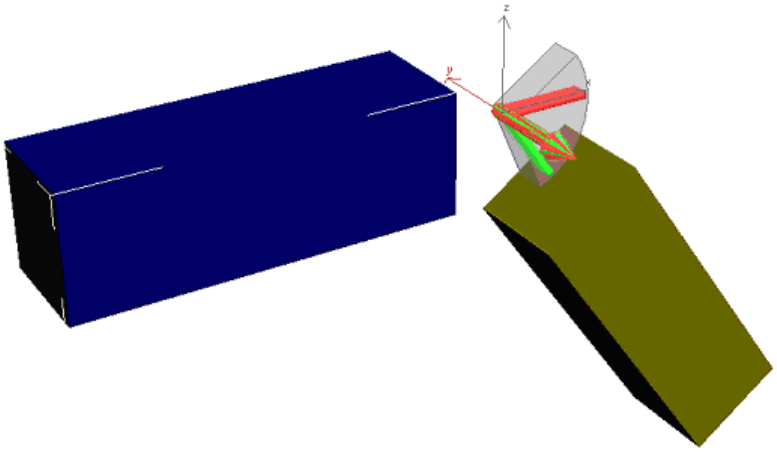
Given the configuration above where the shaded volume defines hinge limits, can the brown box move 90 degrees up, or 90 degrees down? The answer is: it depends on the frame of reference of those limits.

If the limits are displayed based on the blue box and the red mark follows the brown box, then the brown box can move upwards:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



However, if only the red mark follows the blue box and the limits follow the brown, then the brown box can move downwards:



This example illustrates the following idea:

When displaying information to represent constraint limits, the rigid bodies acquire a role in the constraint. One of them (which we denote as the parent) is associated with the spatial representation of the limits, and the other one (which we denote as the child) is associated with the free element that moves inside those limits.

The reasoning behind naming the roles of the rigid bodies as *parent* and *child* is the analogy with bone hierarchies and forward kinematics. Movement of every joint in a hierarchy is usually defined relative to

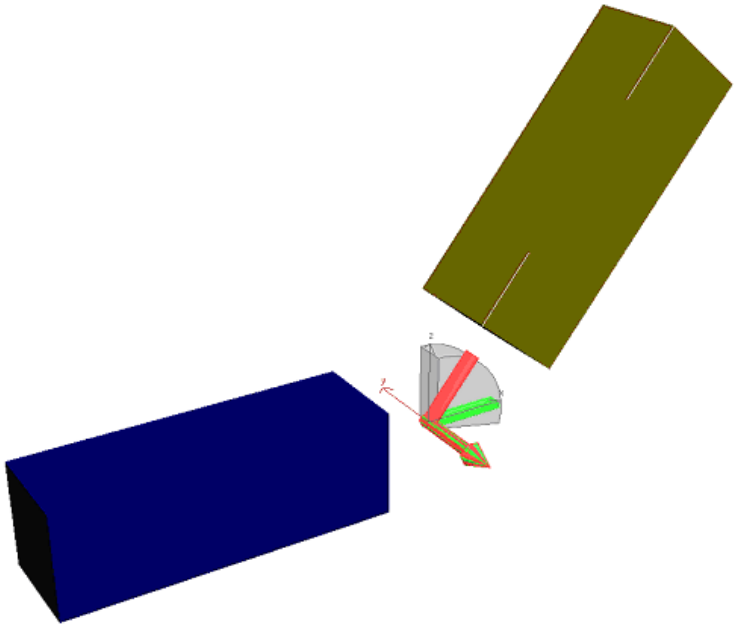
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

the parent bone of the joint: it is much more intuitive to define the limits based on the parent bone than on the child bone.

Consider, for example, a shoulder: you would usually define the joint as a rotation of the arm inside a cone that follows the torso, rather than the rotation of the torso inside a cone that follows the arm. They are both equivalent, but the first is much more intuitive.

Parent and Child Constraint Spaces

Closely related to the concept of parent and child roles for rigid bodies, there is also the concept of parent and child *spaces* associated with a constraint.



The space represented in green, which includes also the limits displayed in gray, is the *Parent Space* - this follows the parent rigid body at runtime. The space represented in red above is what we call the *Child Space* - this follows the child rigid body at runtime.

The important thing to remember is that a constraint operates by considering two spaces: Parent Space (which follows the Parent rigid body) and Child Space (which follows the Child rigid body). At run-time, the constraint will try to ensure that both spaces match (within the limits of each constraint) in world space. For example, a ball-and-socket constraint will ensure that the position of both spaces matches, but will ignore orientations. A hinge constraint will ensure the position and the direction of the main axis match, and will only allow some (possibly limited) rotation around that main axis.

Note:

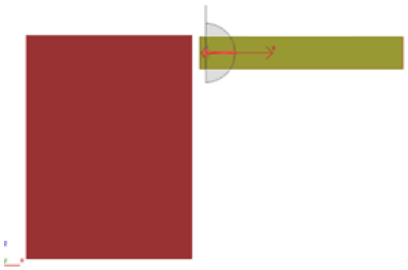
When a constraint constrains a rigid body to the world, the rigid body in question is always the child rigid body. The parent rigid body is considered to be a fixed rigid body (i.e., the parent space is fixed in world).

Manipulating Constraint Spaces

When working with constraints, therefore, we have four spaces to consider:

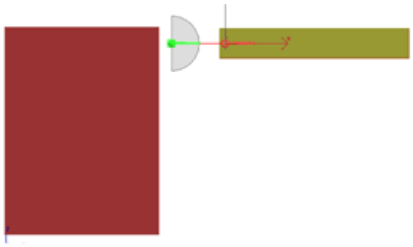
- **Parent Rigid Body Space** : The position and orientation of the Parent rigid body
- **Parent (Constraint) Space** : The position and orientation of the Parent Space of the constraint
- **Child Rigid Body Space** : The position and orientation of the Child rigid body
- **Child (Constraint) Space** : The position and orientation of the Child Space of the constraint

As we have seen, at runtime the parent and child constraint spaces always follow the relevant rigid body spaces. When setting up constraints in a modeler, however, we may want to limit how the spaces move relative to each other during manipulation to make setting up the constraints easier and more intuitive. Consider the following example:

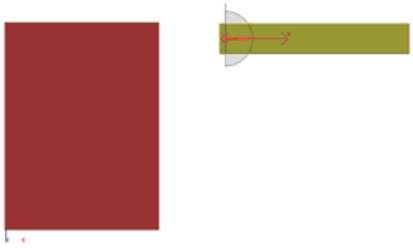


We have setup a hinge to connect the brown box (arm) to the red box (torso). Notice that the two spaces of the constraints are aligned.

Imagine we now decide that we want the brown box to be a little further away, so we move the brown box to the right. If we keep each space relative to it's associated rigid body, we'll get something like this:

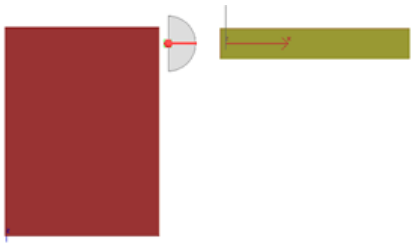


Which basically means that now the two constraint spaces are not aligned, since they have different world positions. Unless we fix this, at runtime the constraint will snap in order to align the translation components of both spaces. In most cases, what we would have liked would have been for the whole constraint to move with the child rigid body:



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

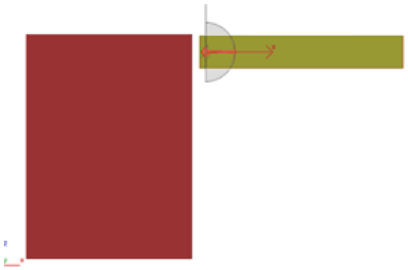
or instead for the whole constraint to remain where it was:



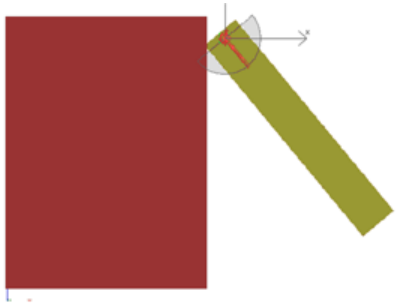
So, one general rule is that:

Very often during constraint setup, we want both spaces of the constraint to move as a unit.

But notice that this not may always be the case. Consider the similar example where we start with the same setup as before:

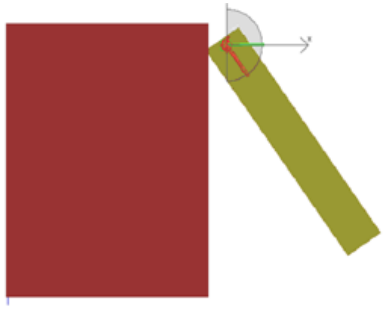


Imagine now that we are happy with the constraint setup, but want our 'skeleton' to be in a different pose, maybe in order to match another skeleton or skin. In particular, we want the arms to be lowered down. If the two spaces of the constraints move as one, then what we would end up with is:



This would change the behaviour of the constraint - the arms won't be able to go as high as before. So in this case, we would prefer both spaces to remain independent, so only the child space (in red) moves.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



So, refining the rule above:

Very often during constraint setup, we want both spaces of the constraint to move as a unit. But not always.

In particular, for constraints like hinges and ball-and-sockets, we usually don't want the translation of the spaces to differ (because it will cause snapping) but for rotations we may. And for springs or prismatic constraints having differing positions for the two spaces is common.

There is another general rule regarding constraint spaces and the pivot of rigid bodies:

Very often the child constraint space, in particular its position, is the same space as the child rigid body space.

The reason for that is that when we create hierarchical systems of objects (like bones) in the modeler, we usually place the pivot points at the locations where the bones rotate relative to each other. When replicating the structure with rigid bodies, we usually place the constraints at those locations.

For example, in the case of the arm above, the most common location for the pivot of the brown box in the modeler would be the location where we have placed the constraint.

Constraint Manipulation in the Modelers

The Havok Content Tools package provides tools for setting constraints and manipulating constraint spaces in each supported modeler. These tools include options to customize the behaviour during constraint manipulation (as described above).

In each modeler, constraints are always associated with the *child* rigid body. There are many reasons for this:

- When representing a hierarchy with rigid bodies and constraints, there are N rigid bodies and N-1 constraints. This means there is a 1-to-1 association between rigid bodies and constraints, except for the root rigid body, where there are no constraints.
- This also reflects the close relationship between bones and joints in skeleton setups: wrist/hand, knee/calf, ankle/foot.
- As we have seen, the constraint spaces are usually located at the pivot location of the child rigid body.
- A constraint always has a child rigid body, while the parent rigid body is optional - it may be constrained to the world.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

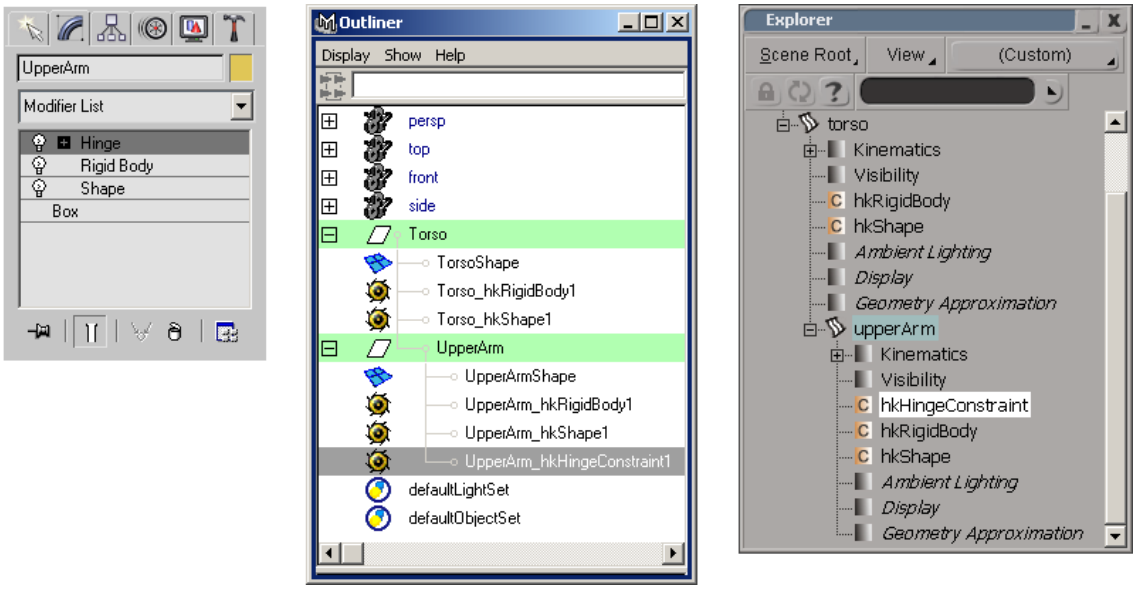


Figure 6.13: Constraints are always associated with the Child Rigid Body

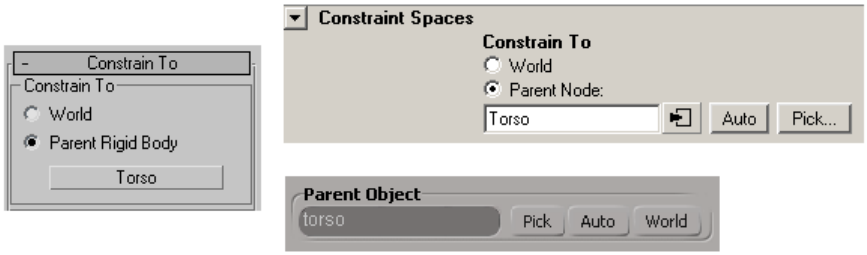


Figure 6.14: The (optional) Parent Rigid Body is a parameter of the constraint

Manipulation Tools

Constraint spaces can always be specificed numerically within each modeler (through a constraint's modifer/attribute editor/property panel). However it is much more intuitive to be able to interact with a constraint dynamically in the viewports. This behaviour is provided by the manipulation tools. Each modeler has a specific implementation of the manipulation tool, since the capabilities and internal behaviour of the modelers differ greatly:

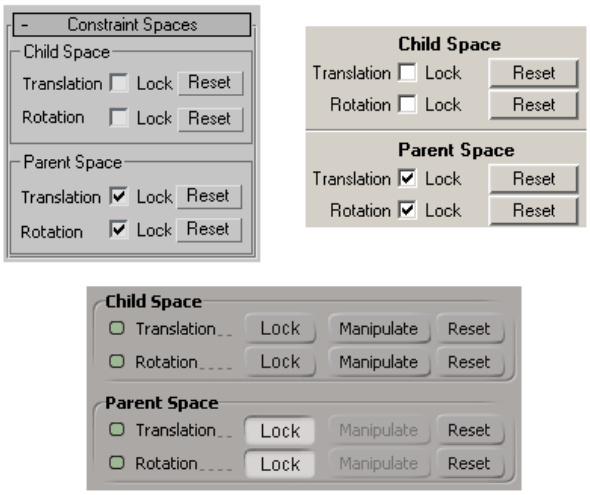
- In 3ds max, Constraint Spaces are represented as subobjects of each constraint modifier, and can be manipulated using standard move and rotate tools.
- In Maya, Constraint Spaces are modified through manipulators associated with each constraint node. These manipulators can be accessed by directly selecting a constraint node and activating the 'manipulate' tool, or by indirectly selecting a constraint node (eg. by selecting its parent transform) while the 'Havok Physics' tool is active - this tool provides and quicker means to access and manipulate Havok nodes in general.
- In XSI, Constraint Spaces are modified through a custom setup, which is activated by the 'manipulate' buttons in a constraint parameter set's property panel. A temporary hidden null object is created in the scene, and is wired to the constraint space parameters. By translating/rotating this null object, an effective manipulation scheme is achieved. Once manipulation is complete, we

need to remove the temporary nodes - this occurs if the selection changes or if the 'end manpiulate' button is pressed.

See the individual modeler documention for more on each manipulation tool.

Locking/Unlocking of Spaces

In order to allow different manipulation modes and specific behaviours of constraints how are modified within a scene (as described earlier), Havok constraints allow the user to lock and unlock the constraints spaces, both in terms of their translation and rotation:



Child Space

- **Lock Translation**

  If ON, the translation of the child space is fixed and cannot be manipulated (by default, the child space is located at the pivot location of the child rigid body). If OFF, the translation component can be manipulated.

  The Child Space is always kept relative to (always follows) the Child Rigid Body.

- **Reset Translation**

  Pressing this button aligns the translation of the Child Space to that of the pivot of the child rigid body.

- **Lock Rotation**

  If ON, the orientation of the child space is fixed and cannot be manipulated (by default, this orientation matches the orientation of the child rigid body pivot). When OFF, the orientation of the child space can be modifier / manipulated.

  The Child Space is always kept relative to (always follows) the Child Rigid Body.

- **Reset Rotation**

  Pressing this button will align the orientation of the Child Space to match the orientation of the child rigid body.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Parent Space

- **Lock Translation**

  If ON:

  - The translation of the Parent Space cannot be manipulated (by default is aligned with the Child Space).
  - The translation of the Parent Space is kept relative to the Child Space (i.e., both spaces will move together).

  If OFF:

  - The translation of the Parent Space can be manipulated.
  - The translation of the Parent Space is kept relative to the Parent Rigid Body (if there is any) or fixed in world (if there is no parent).

- **Reset Translation**

  Pressing this button will align the translation of the Parent Space to that of the Child Space

  Note:
  The exception being the Stiff Spring constraint, where the Parent Space will be aligned with the parent rigid body.

- **Lock Rotation**

  If ON:

  - The orientation of the Parent Space cannot be manipulated (by default is aligned with the Child Space).
  - The orientation of the Parent Space is kept relative to the Child Space (i.e., both spaces rotate together).

  If OFF:

  - The orientation of the Parent Space can be manipulated.
  - The orientation of the Parent Space is kept relative to the Parent Rigid Body (if there is any) or fixed in world (if there is no parent).

- **Reset Rotation**

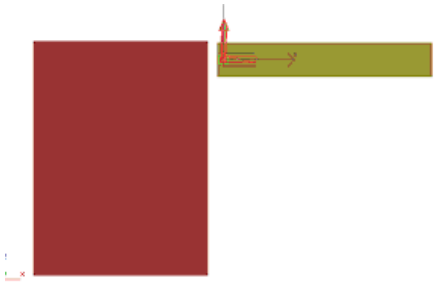
  Pressing this button will align the orientation of the Parent Space to that of the Child Space.

Notice how locking/unlocking the child space only affects whether manipulation is available or not; while in the case of the parent space it also affects the behaviour of that space whenever the rigid bodies in the constraint move.

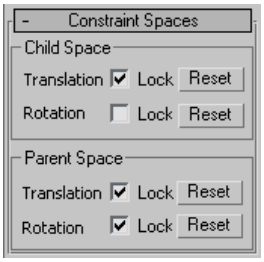
Example

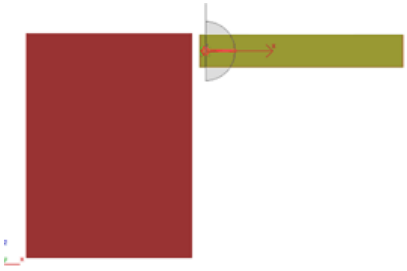
In this case a constraint is created with the 'arm' as the child rigid body and the 'torso' as the parent rigid body:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

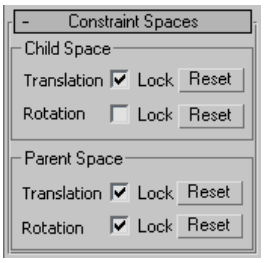


Creating the constraint by default aligns both parent and child spaces with the pivot of the child rigid body. We may want to orientate the whole constraint, for example to match the desired rotation axis for a hinge. We unlock rotation of the child space (so we can change it), and keep the parent space fully locked, so it follows the child space (and the constraint moves as one). Now when we rotate the child space, the parent space follows:

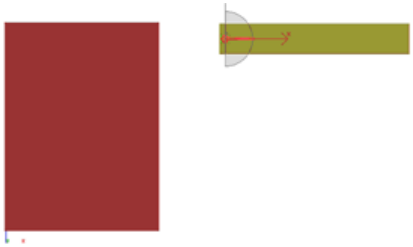




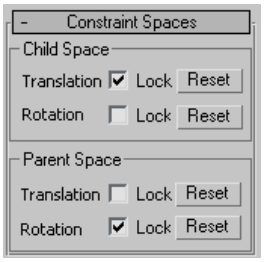
Imagine that we now want to move the arm to the right (or the torso to the left). By keeping the parent space translation locked, both spaces move/remain together with the child rigid body:

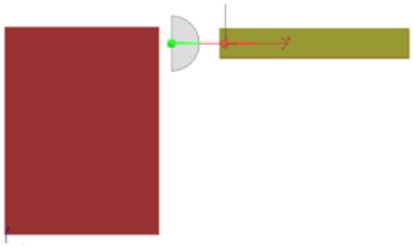


Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

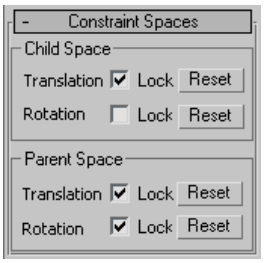


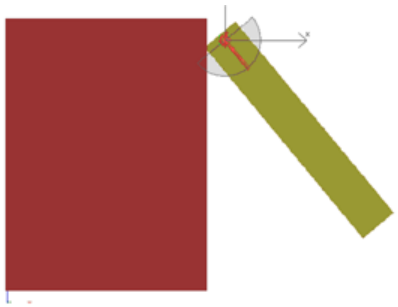
However, if the translation of the parent space is unlocked before the translation, the parent space moves/remains with the parent rigid body:





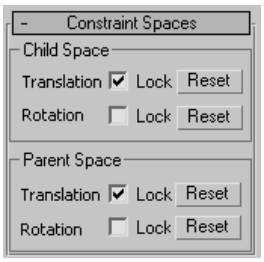
Consider now the case where we want to rotate the arm clockwise (or the torso counter-clockwise). In the first case the parent space rotation is locked, so both spaces rotate together:

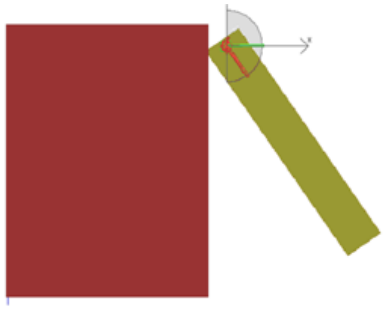




Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

However, if we unlock the parent space rotation, it remains relative to the parent rigid body and does not follow the child space:





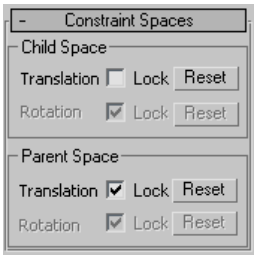
6.6.3.2 Constraint Types

The Havok Content Tools provide several types of constraint for each supported modeler. The Create Constraints filter converts any of these constraints found in a scene into their runtime equivilants during the export process.

The Ball and Socket Constraint

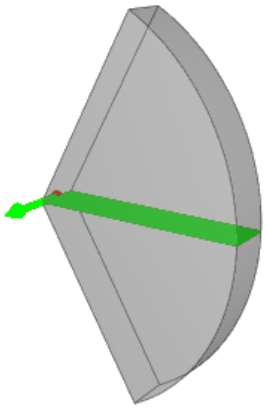
The ball and socket constraint is the simplest type of constraint provided by the Havok Content Tools. At runtime, this constrains the child space position in the child rigid body to the parent space position in the parent rigid body / world.

This constraint does not provide any additional properties beyond those provided by the common constraint space properties. Also, this constraint has no concept of rotation so the rotation elements of the constraint space properties are disabled/hidden:



The Hinge Constraint

The hinge constraint allows you to constrain two rigid bodies together relative to a position and axis in each body's space, or to constrain an axis in one body's space to an axis in world space. The constrained objects are then free to rotate about this axis. The allowed rotation between the attached objects may also be limited. This limit is defined with respect to an axis perpendicular to the hinge axis (the *zero-axis*) for each body. These sets of axes are defined by the child and parent constraint spaces.



The properties provided by the hinge constraint include the standard constraint space properties plus: the option to limit the angle of rotation; a frictional torque value for the rotation; a motor type to drive the rotation:

Some modeler's also provide a 'display size' property, which affects the display in the viewports only.

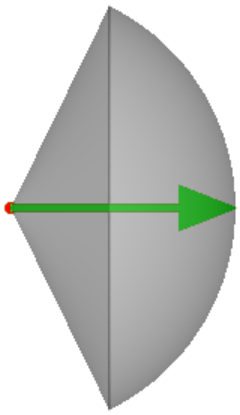
The Rag Doll Constraint

Rag doll constraints are useful for simulate complex joints, such as those found in some human and animal joints (shoulders, hips). These joints may require some freedom of rotation in more than one axis, as well as some ability to twist.

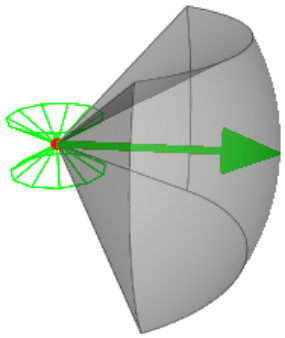
Like the other constraints we have seen so far, this one constrains the child space position in the child rigid body to the parent space position in the parent rigid body / world. However the rotational constraints have a somewhat complex parameterization which requires some additional explanation.

The first parameter is a 'cone angle', which is measured from the main axis of the parent space. This ranges from 0 to 180 degrees (no freedom to full freedom):

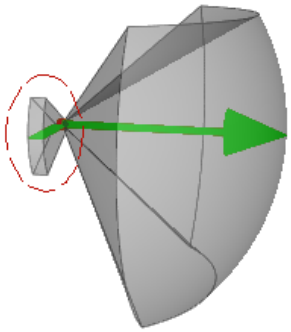
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Two other angular parameters (which we call 'plane max' and 'plane min') are used to specifiy cones which subtract from the volume above. These cones are specified from axes which are perpendicular to the main axis, one from the positive axis and the other from the negative axis. Their effect is clear when a three dimensional view is taken:



The main axis of the child space is now constrained to be located within the shaded volume in the diagram above. We can also constrain the rotation of the child space around it's main axis. This behaves the same way as the hinge constriaint, and can be represented in a similar way - as an additional volume at the base of the rag doll constraint:



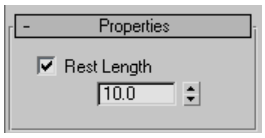
The properties provided by the rag doll constraint allow these various angular parameters to be edited, as well as a frictional torque value and a motor type. Options are also provided which give some control over the visual representation of the constraint in the modeler:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

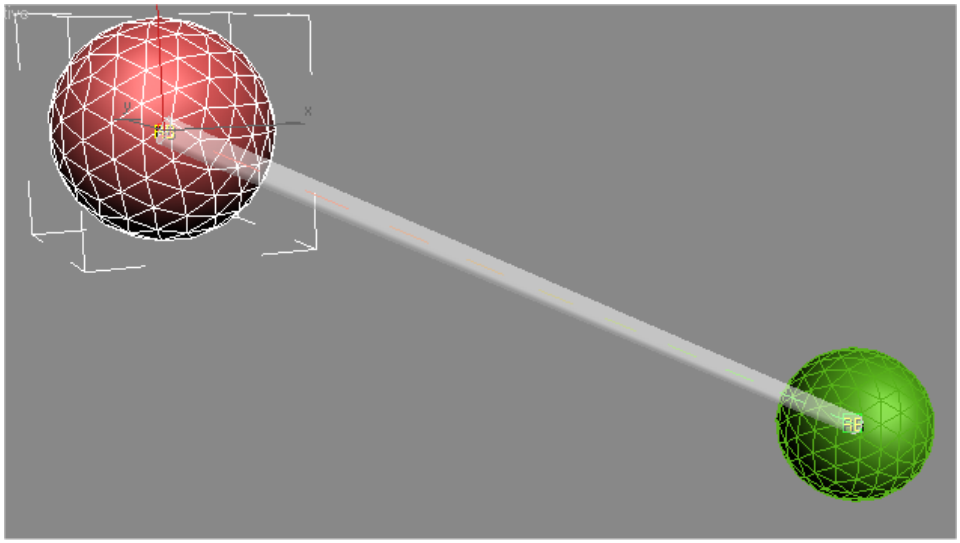
Again, some modeler's also provide a 'display size' property, which affects the display in the viewports only.

The Stiff Spring Constraint

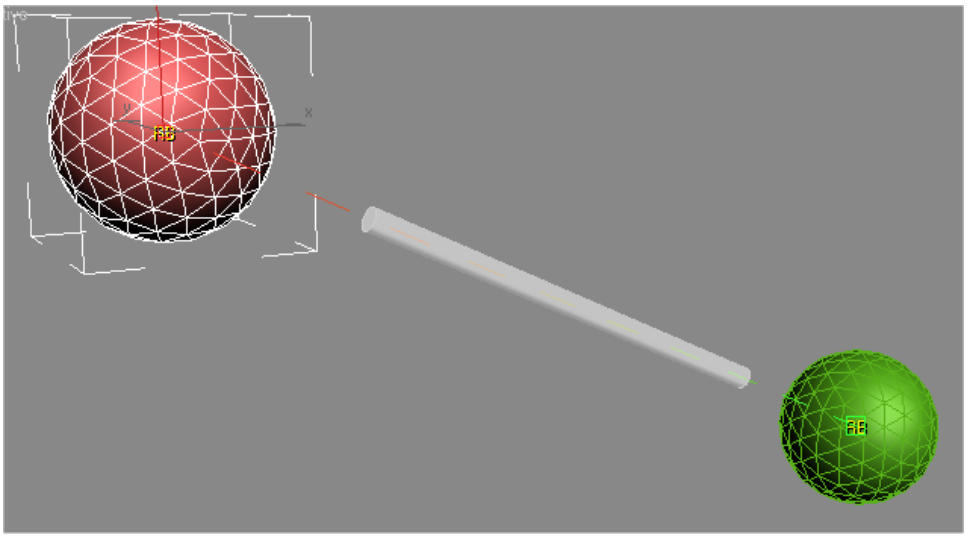
The stiff spring constraint constrains the child space position in the child rigid body to lie a set distance from the parent space position in the parent rigid body (as if they were connected by an inflexible rod). As in the ball and socket constraint, there is no constraint on the child or parent space rotation.



The properties provided by the stiff spring constraint include the standard constraint space properties plus an optional rest length parameter. If this parameter is disabled, the spaces are constrained by default to be separated by their current distance.



Specifying a rest length parameter overrides this default.



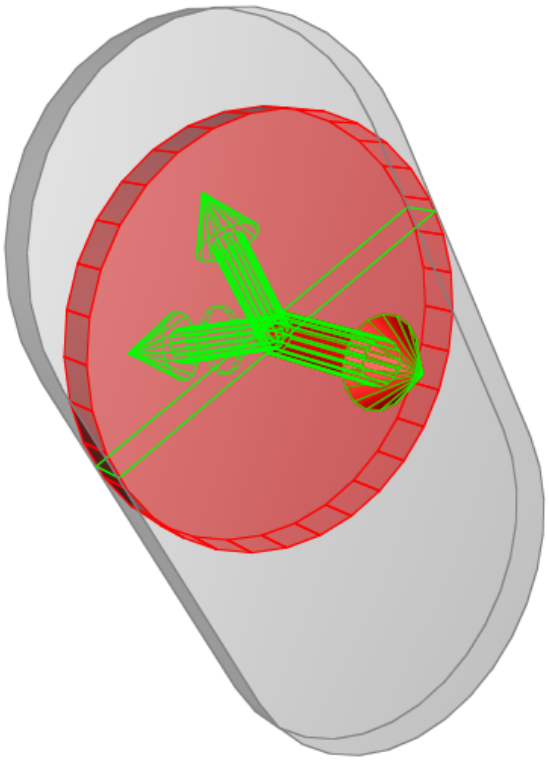
The rest length is displayed as a solid gray bar (of that length) in between the two spaces. A dashed line is also shown in between both spaces.

The Wheel Constraint

The wheel constraint constrains the child rigid body to rotate about a specified "spinning" axis relative to the parent body, and optionally to translate along a specified "suspension" axis for specified distances in each direction. The "steering" axis is by default aligned with the suspension axis, but may be offset relative to the suspension axis by a specified rotation angle about the spinning axis.

In the constraint display, the child space is represented as a wheel in the plane perpendicular to the spinning axis, indicated by an arrow. The suspension axis is indicated by an arrow with a perpendicular plane marking the zero point of the suspension limits. The limits are represented by a surface showing the volume swept out by the wheel as it moves along the suspension axis between the limits. The steering axis, which differs from the suspension axis if the offset angle is set to be non-zero, is denoted by an arrow with an annular ring around it.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



The properties provided by the wheel constraint include the offset angle between the steering and suspension axis, the suspension limit minimum and maximum (negative and positive values respectively), and frictional torque and damping parameters.

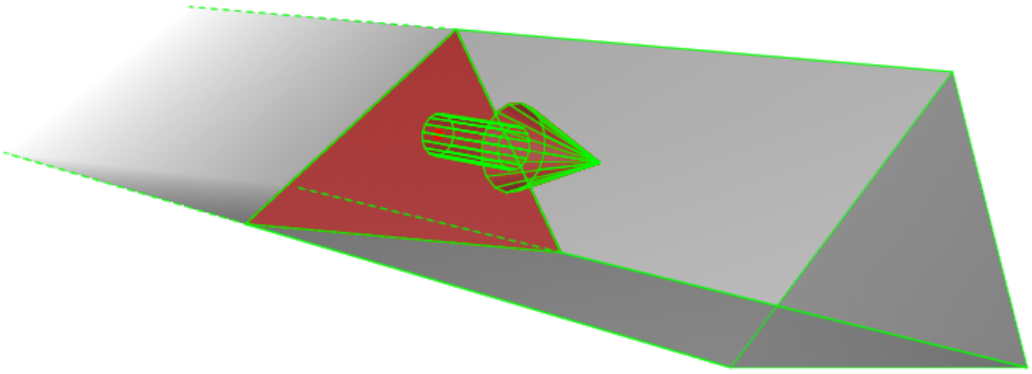
The 'display size' property in this case scales the size of the displayed wheel, plane and arrows while keeping the distance between the suspension limits constant.

The Prismatic Constraint

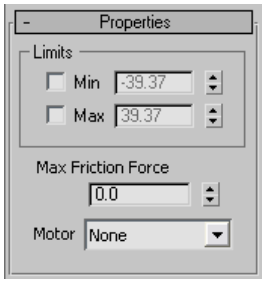
The prismatic constraint constrains the child rigid body to move along a specified axis relative to the parent body by specified distances in each direction (or if not specified, by an unlimited distance).

In the constraint display, the child space is represented as a plane perpendicular to the main axis, indicated by an arrow. The limits are represented by a plane perpendicular to the main axis indicating the zero position, and a surface showing the boundary of the volume swept out by the plane as it moves along the prism axis between its defined limits. If the minimum or maximum limit is not enabled, the prism volume in that direction fades out.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



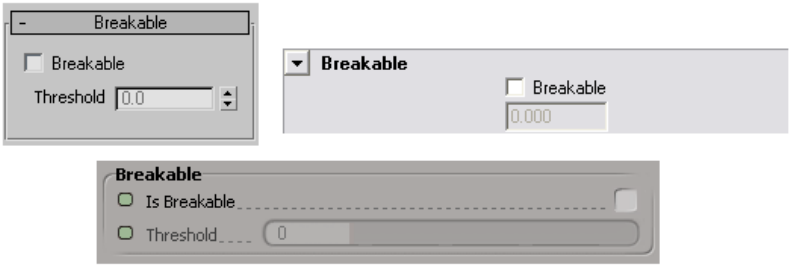
The properties provided by the prismatic constraint include optional minimum and maximum limits (negative and positive values respectively), a friction value, and motor type.



The ' **Display Size** ' property in this case scales the size of the displayed prism and arrows while keeping the distance between the limits constant.

6.6.3.3 Breakable Constraints

Any constraint can be made breakable if desired, by selecting the " **Breakable** " checkbox. The " **Threshold** " parameter specifies the force beyond which the constraint breaks.



Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.6.4 Animation: Annotations

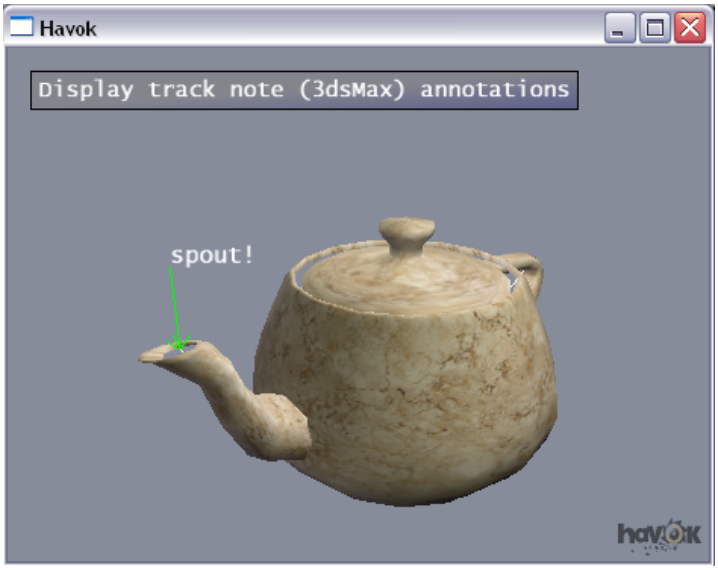
Havok Animation supports the concept of annotations. Annotations are animated string variables associated to bones. They are normally used to mark (annotate) particular frames of the animation that are interesting from the point of view of the application running the animation. Examples of usage would be:

- In a walk/run animation, annotate the frames where the foot is planted/lifted (to trigger sounds, or to do IK)
- Mark transition points between different animation clips
- Mark interesting poses in an animation (for example, Havok's GDC 2005 demo uses multiple annotations in a single "get up" animation in order to identify good poses from which to drive a rag doll before getting up - check the *Havok Animation Use Cases* section of the *Havok Animation* manual for details).

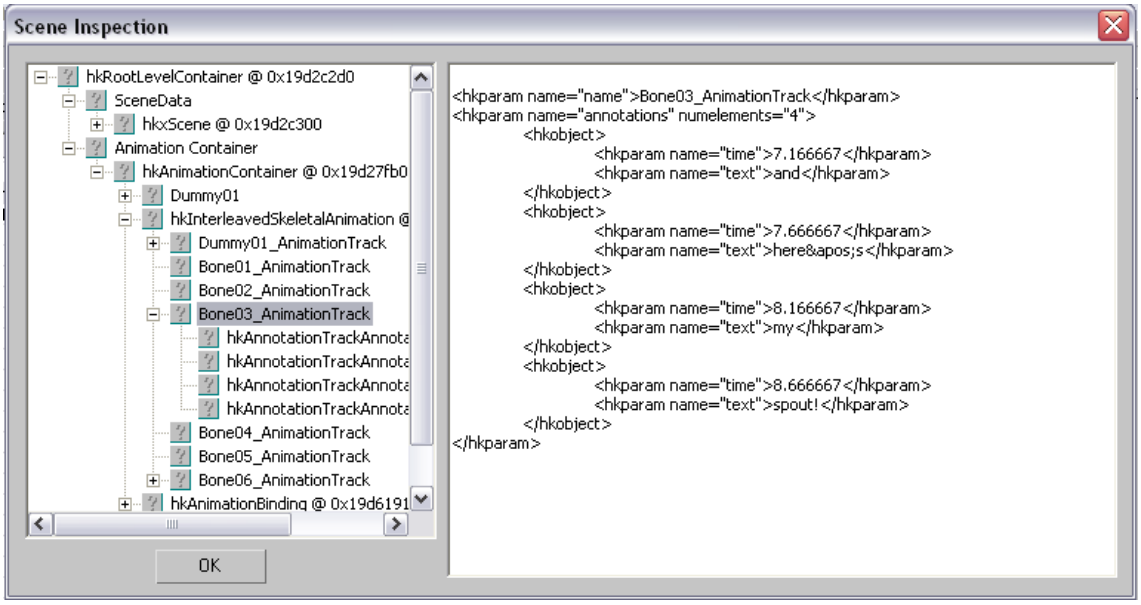
The Havok Content Tools provide support for exporting annotation data from each supported modeler. See the Havok Content Tools documention for your relevant modeler for more on the specific implementations.

6.6.4.1 Runtime Access to Annotations

The Annotation demo (Animation Api > Playback > Annotation), featuring an animated teapot which displays annotated strings, is a good starting point for learning about using annotations at run-time:



The `hkaAnnotationTrack` class stores annotation data. An `hkaAnnotationTrack` simply consists of a `time` and a `text` value, so multiple `hkaAnnotationTrack`s are used to represent an annotation as it changes value, with each track containing the new value of the annotation and the time at which it changes. This can be seen in the following screenshot of the View XML filter:



A skeleton's annotated tracks can be accessed via the `hkaAnimatedSkeleton::BoneAnnotation` struct, which also stores the ID of the bone to which the annotation track belongs. The `getNumAnnotations()` and `getAnnotations()` methods in `hkaAnimatedSkeleton` analyse the specified timestep and return any relevant annotation data. The following code snippet from the Annotation demo shows how this is done:

```
// Grab any track note annotations that are present and valid for the time step
// Usually we would just query for this delta, but for demo purpose we extend this.
// This means the annotations are displayed on screen for longer
hkReal delay = 0.25f;

// Query to see how many annotation tracks we have.
const hkUint32 numAnnotations = m_skeletonInstance->getNumAnnotations( delta + delay );

// Create a temporary array and query for the annotations.
hkLocalArray<hkaAnimatedSkeleton::BoneAnnotation> annotations( numAnnotations );
annotations.setSize( numAnnotations );
m_skeletonInstance->getAnnotations( delta + delay, annotations.begin() );

// Advance the active animations
m_skeletonInstance->stepDeltaTime( delta );

const int boneCount = m_skeleton->m_numParentIndices;
```

6.6.5 Animation: Controlling Compression

There are several tweakable parameters available when using Havok's Animation compression pipeline (using the Delta Compression and Wavelet Compression filters). The following guide provides a short explaination on how to choose these parameter values to balance size, quality and decompression speed.

6.6.5.1 Controlling Quality

The sections of the compression pipeline which may introduce error into the animation signal are:

- Truncation during track analysis
- Precision loss during quantization
- Wavelet coefficient truncation/thresholding
- 3-Component Quaternion Compression

Track Analysis

Track analysis attempts to remove static and clear degrees of freedom from your rig. You independently specify tolerances for position, rotation and scale. This analysis does not degrade gracefully; if you pick over-aggressive tolerances then the error is generally very obvious.

Symptoms of over-aggression are generally *low-frequency* error and include:

- Loss of movement in a limb
- A complete subskeleton appears rotated
- Collapsed bones (usually small bones collapse first e.g. fingers, toes and clavicle bones)
- Loss of root motion

To compensate, simply reduce the tolerances presented. Usually you can be more aggressive with positional tolerances than with rotational tolerances (the values compared for rotation are actually quaternion components; the tolerance does not represent an angular difference). It is a good idea to double-check the values displayed during track analysis to ensure that the automatic identification of degrees of freedom correspond to your expected rig setup. For example if none of the bones in your rig change scale you should expect to only see three dynamic position tracks for the root movement. All other dynamic tracks should correspond to rotation, for example:

Bad: (Using the default Relative Position Tolerance = 0.01)

```
[Track Analysis]
        Total DOFs 760
        Static position DOFs 201
        Static rotation DOFs 77
        Static scale DOFs 0
        Clear position DOFs 6
        Clear rotation DOFs 36
        Clear scale DOFs 228
        Dynamic position DOFs 21
        Dynamic rotation DOFs 191
        Dynamic scale DOFs 0
        Redundancy 3.584906
```

Good: (In this case Relative Position Tolerance = 0.1)

```
[Track Analysis]
        Total DOFs 760
        Static position DOFs 219
        Static rotation DOFs 99
        Static scale DOFs 0
        Clear position DOFs 6
        Clear rotation DOFs 36
        Clear scale DOFs 228
        Dynamic position DOFs 3
        Dynamic rotation DOFs 169
        Dynamic scale DOFs 0
        Redundancy 4.418605
```

As a rule of thumb, be as aggressive here as you can (larger tolerances), but do not stray far from the default values. In most situations the values here depend on the rig set up, so you can generally choose a good set of values for a given character and reuse them.

Note:
Rotation tolerance is usually far smaller than position tolerance and also far more critical as it is likely that most of your dynamic data will be rotational.

For more specific details on how the tolerances are used to analyze data see the section 'Track Analysis' in the Havok Animation manual.

Quantization Bit Width

After track analysis all remaining degrees of freedom are transformed (using a delta or wavelet transformation) and then quantized. Quantization typically introduces more error than other stages.

Quantization error degrades more gracefully than track analysis, but is generally *high-frequency* (and thus may be more noticable). Symptoms include:

- Jittering motion
- Shaking end effectors
- Popping motion

When choosing an appropriate quantization value pay close attention to the end effectors (hands and feet). Quantization error usually appears as shaking here and the obvious solution is to increase the number of bits used.

To combat quantization error it may be worth considering shortening chains in your skeleton. Currently error accumulates down the chains in hierarchies as all animation data is sotred (and compressed) in local/bone space.

Also consider the range of values that needs to be quantized. Obviously the larger this range is the larger the error each bit represents. For example, if your root motion moves for `x=0` to `x=10000` and you are using 8 bit quantization you may see your character snap to regular positions. If the error is contained in the root motion, it is worth considering adding motion extraction to the pipeline.

Before quantization is performed, each degree of freedom is scanned to find the signal range for that track (min, max, offset and scale). These values are used for this track for the whole animation. For long

animations it is often worth dissecting them into two or more shorter animations to improve both quality and compression. For example, if hands only move in the second half of the animation then splitting it in 2 will both improve the individual ranges for quantization and potentially find more redundancy during track analysis.

Coefficient Truncation/Thresholding

Wavelet compression adds an extra truncation/thresholding and encoding stage to the pipeline. Once wavelet transformation has taken place many of the values are zero or close to zero. The truncation/thresholding phase clamps a proportion of the total values to zero. The wavelet transformation itself is lossless. It is simply this truncation phase that introduces loss. This loss degrades the quality somewhat gracefully.

For wavelet compression to be effective, a heuristic for dropping/discarding coefficients of the wavelet-transformed data must be picked. For versions of Havok pre Havok 5.0.0 this heuristic is to discard a fixed percentage (truncation fraction) of all (sorted) wavelet coefficients, which gives a roughly linear change in filesize (a truncation ratio of 50% or 0.5 would half the size of a file with truncation ration of 0% or 0). This linearity is not exact because of the overhead of static data storage and the RLE used on the final data (hence it depends on bitwidth choice also). A better heuristic is to threshold the coefficients by an absolute value, and discard coefficients less than this value, or whose contribution to the final sample is less than a value. In this case the track analysis parameters can be used directly as the thresholding values, and the Compression slider is no longer used. For the same quality of animation, this heuristic gives about 10-20% improvement in compression. We anticipate that this improvement should always be used in the future (hence the truncation algorithm is marked as 'Deprecated'). However we offer backward compatibility in both the Content Tools and the runtime. In the Content Tools, a checkbox for Use Truncation Ratio (Deprecated) can be used.

Note:

If you are using 'truncation' the only other consideration is block size. The longer the input signal is, the better the wavelet transform will perform i.e. more values will be zero or close to zero. With our block encoding scheme the signal is presented to the wavelet transformer in blocks, each of which has 'block size' elements. As a rule-of-thumb the shorter the block size is the lower the truncation proportion should be. e.g. with a block size of 8 usually a truncation of 50 to 60 percent may be acceptable (this will set 4 or 5 of the 8 transformed values to zero!). With a larger block size, much higher proportions are possible with little visual loss.
A good example of this is cut-scene data. If there are no playback restrictions (see below) then the entire length of the animation can be wavelet transformed (block size = -1 or full) and a very high threshold used for the truncation proportion.

3-Component Quaternion Compression

In addition to the tolerances above above, track analaysis may optionally compress the rotation component of all tracks by using '3-component quaternion compression', reconstructing the 4th components from the other 3, and from Havok 5.0.0 onward does this by default. Since the remaining coefficients will be quantized (and in the case of a wavelet-compressed animation, wavelet-transformed before quantization), reconstruction *may* introduce some noticable error if very aggressive compression settings are being used in conjunction with this algorithm. However, we do not expect that this option need be disabled for most use-cases. If you find problematic animations which exhibit noticable error unless this heuristic is disabled, please contact Havok support.

6.6.5.2 Controlling Decompression Speed and Memory

When setting block size it is important to consider the runtime implications. An entire block must be decompressed together. A block size of 8 implies the decompression system must fully decompress 8

frames for each dynamic degree of freedom in your animation. Either you take the CPU hit for doing this or you pass a cache during sampling and cache the data.

See the section on caching in the Animation Runtime for more detail on how to choose an appropriate cache size.

Note:

If you are decompressing cut-scene data (compressed with block size=full) it is entirely possible to create a temporary cache for use while the cut-scene is playing back. This cache can be discarded when the cut-scene has finished playing.

6.6.5.3 Computing and Controlling Size

The following shows the statistics displayed at the end of compression:

```
[Compression]
      Original Size 125952 Compressed Size 5924 [1886 static/incompressible]
      Compression Ratio: 21.261311:1
```

This original size is computed from the storage requirement for a full 4x4 matrix (32 bit floats) per frame for each bone in the animation. (The example used here has 48 bones and 41 frames). The compressed size is the total in-memory requirement for the compressed animation. This data is comprised of a compressible part (the final output from the pipeline) and an incompressible part (data used for pipeline set up. e.g. the quantization ranges for each DOF, the track analysis mask and values etc.). The value here is deliberately broken out as it will not change when you tweak either quantization bit size or truncation. This allows you to see how much further you can continue to compress (in this animation 4038 bytes are being used to represent the signal).

Also note that if you are using delta compression, the final size of the compressed animation doesn't reduce linearly with decreases in bit width. For example 16 bit delta compression is not twice the size of 8 bit. The reason here is that the block encoding is storing a full float at the start of each block. With a block size of 8 A full float is stored followed by 7 quantised values. Only the quantised values change as the bit width changes. The float storage remains constant. Similarly there is not a linear relationship between the wavelet truncation coefficient and the final size of the compressed data. When data is transformed into the wavelet basis many values are already very small and after quantisation go to zero. The truncation coefficient simply guarantees a proportion of the data is in fact zero - this allows the encoder to work well.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.7 Integrating with the Havok Content Tools

6.7.1 Introduction

The Havok filter pipeline provides a framework for generic asset processing. This framework consists of several components which are customizable and extensible to suit most user's requirements. This chapter provides technical details on how to extend and integrate with the various components of the pipeline.

Tip:
This documentation references a few classes documented in the Reference Manual. If you are reading this document as part of the overall *Havok* manual (which includes that Reference Manual), you will see links between some C++ class names and the reference for those classes. If you are just reading the Havok Content Tools Manual standalone CHM those links will not be available.

6.7.2 Offline Processing

6.7.2.1 .HKO Files

The filter configurations used to process an asset are always saved with their respective assets, but they can also be saved as separate .HKO files - check the Configurations section to see how to save and load these files. HKO files are saved in XML form using Havok's serialization SDK, which means that they can be edited by hand.[7] An options file consists of a series of serialized objects (in `<hkobject>` tags) of different types. The order of these objects is important, as it controls the order of appearance of the configurations and their filters when they are displayed in the filter manager.

[7] Although HKO files can easily be edited by hand (and autogenerated), be careful when doing so as there are limits to what the serialization framework and the filter manager can handle in terms of incorrect input.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
<?xml version="1.0" encoding="utf-8"?>
<hkoptions>
    <hkobject class="hctConfigurationSetData">
        <hkparam name="filterManagerVersion">65537</hkparam>
        <hkparam name="activeConfiguration">0</hkparam>
    </hkobject>
    <hkobject class="hctConfigurationData">
        <hkparam name="configurationName">HKX Preview</hkparam>
        <hkparam name="numFilters">1</hkparam>
    </hkobject>
    <hkobject class="hctFilterData" name="PreviewScene">
        <hkparam name="id">2744629365</hkparam>
        <hkparam name="ver">66049</hkparam>
        <hkparam name="hasOptions">true</hkparam>
    </hkobject>
    <hkobject class="hkFilterPreviewOptions" name="PreviewScene">
        <hkparam name="renderer">RENDERER_DIRECTX9</hkparam>
        <hkparam name="hardwareSkinning">false</hkparam>
        <hkparam name="stepSimulation">true</hkparam>
        <hkparam name="passOnSimulationResult">true</hkparam>
    </hkobject>
</hkoptions>
```

The first object in the example is of type `hctConfigurationSetData`. This is a header object found once at the start of every options file. It contains the version of the Havok Filter Manager the options file was created with, and the index of the active configuration, which will be displayed in the Filter Manager when the options are loaded.

The next object, of type `hctConfigurationData`, contains the information for a specific configuration. It lists the configuration's name, and the number of filters it contains. When adding or removing filters by hand be sure to modify the `numFilters` value appropriately. `hctConfigurationData` objects are followed directly by XML describing the filters they contain, which is in turn followed by a `hctConfigurationData` object for the next configuration in the set.

Every filter is described by either one or two XML objects. The first, `hctFilterData`, is always present. It contains the filter's ID and version number, and a `hasOptions` element. If `hasOptions` is true, the `hctFilterData` is followed by a second XML object detailing the options saved with the filter. If it's false then the filter has no options, and the single `hctFilterData` object is an adequate description.

The filter options object in the example is of type `hkFilterPreviewOptions`. Each type of filter will have different options data saved against it. When modifying this information refer to the explanations of the individual filters in the Filter Pipeline section to see what the various values represent.

6.7.2.2 Overriding and Upgrading Filter Settings

It is possible to automatically specify which filters to use when exporting an asset through the use of an HKO file. This can be useful in a number of situations. For example, you may want to:

- Ensure that exactly the same filter options are used whenever you export an asset.
- Change (upgrade) the filter settings of multiple assets.
- Export data using a specific filter set, but without modifying the original filter settings saved with the assets.

This is done by copying a HKO file containing the desired filter set to your Havok Content Tools installation directory ("`%ProgramFiles%\Havok\HavokContentTools`" by default) and renaming it appropriately. To create such a HKO file simply launch the Filter Manager, build up a configuration set, and select **File->Save Configuration** from the main menu.

Overriding Filter Settings

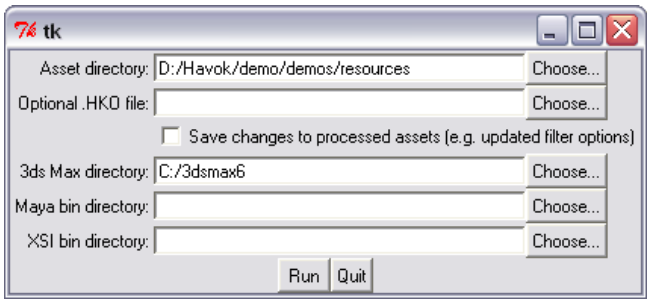
To temporarily override the filter settings put an options file called "`hkOverride.hko`" in your Havok Content Tools installation directory (same folder as "`hkfiltermanager.dll`"). If the filter manager finds such an override file when an asset is processed, it will use its filter set to process the data instead of the usual filter set. Override options can never be saved with their asset.

Upgrading Filter Settings

If you want to upgrade the filter settings of an asset, place an options file called "`hkUpgrade.hko`" in your Havok Content Tools installation directory (same folder as "`hkfiltermanager.dll`"). If the filter manager finds such an upgrade file when an asset is processed, it will be used to process it and will also replace whatever filter set the asset originally had (you will need to resave the scene to save those changes with the asset).

6.7.2.3 Batch Processing Scripts

A Python script for batch processing assets is provided as part of the Havok SDK: this script can be found in "`tools/bin/batchprocess/processAssets.py`".[8]



This script will attempt to invoke 3ds Max, Maya and XSI in silent mode and will execute scripts on each to recursively examine all the asset files and folders inside the specified *asset directory*. Any compatible scenes will be loaded and processed in batch (silent mode) in turn.

It is also possible to specify a single .hko file containing specific filter setup options whch should be used to process each asset. If none is provided, then each asset will use whatever filter setup is present in its scene (provided there is no upgrade or override HKO set up).

If the " **Save Changes to Asset Files** " check box is selected, the asset files will be resaved after processing. This is useful you want to upgrade the filter settings of the asset.

The **installation directories** for each of 3ds Max, Maya and XSI are required so the script can find the suitable executables. If a modeller's application folder is included in the `PATH` environment variable then

[8] You will need Python 2.3 or later to execute this script. You can download Python free of charge from www.python.org

you may leave the installation directory blank (Maya usually adds itself to this environment variable on installation).

It is also possible to run this script silently by passing the above arguments via the command line. See the script source for argument usage.

6.7.3 The Havok Scene Data Format

In this section we will describe the format and structure of the information that is processed in the filter pipeline. Please note that some of the information in this section should be complemented by the *Serialization* chapters in this manual.

6.7.3.1 Variants and Named Variants

Sometimes, when describing data, we need to have the ability of storing information the type of which is unknown at compile time. For example, a node in the scene may contain a mesh, a camera, a light.. While inheritance can help in some cases, in many cases we want to leave the door open for arbitrary types.

A Variant (`hkVariant`) object is defined as

```
struct hkVariant
{
    void* m_object;
    const hkClass* m_class;
};
```

That is, an arbitrary object described through an `hkClass` (check the *Serialization* documentation for details on `hkClass`). The `hkClass` object is required because otherwise there is no information available about the type or structure of the object pointed by `m_object`.

A Named Variant is nothing more than combination of a Variant and a name:

```
struct NamedVariant
{
    const char* m_name;
    const char* m_className;
    hkVariant m_variant;
};
```

Note:
The `NamedVariant` class contains cached information (the `className`) and thus access to its members is done through `getName()`, `getObject()` and `getClass()` methods as well as a `set()` method.

6.7.3.2 Root Level Container

The information passed around through the filter pipeline is structured through the use of a Root Level Container object. An `hkRootLevelContainer` object is just a collection of Named Variants:

```
class hkRootLevelContainer
{
    [...]
        class NamedVariant* m_namedVariants;
        int m_numNamedVariants;
};
```

It also defined some methods to easily find object by name and type. Structuring the processed data in this way allows for different sets of data (scene, animation, physics) to be added by different filters to be grouped and found easily.

6.7.3.3 Environment Data (hkxEnvironment)

An `hkxEnvironment` object is normally used during asset processing in order to store information which is relevant to the processing but may not be required to persist.

An `hkxEnvironment`, in a very similar fashion to an OS environment, stores a dictionary of string pairs (variable, value):

```
class hkxEnvironment
{
    public:
[...]
        hkResult setVariable (const char* name, const char* value);
        const char* getVariableValue (const char* name) const;
[...]
        hkString convertToString () const;
        hkResult interpretString(const hkString& str);
[...]

    private:
        struct Variable
        {
            const char* m_name;
            const char* m_value;
        };
        hkArray<struct Variable> m_variables;
};
```

The `hkxEnvironment` dictionary is case-insensitive when searching for a variable name - that is, a variable stored as "Output" will also be retrieved when looking for "output". Values are returned with the case they were last stored.

The class also provides methods to convert to and from a string representation. This is used by the 3ds Max, Maya and XSI export options to allow user to specify their own environment variables.

These scene exporter automatically add an `hkxEnvironment` object which contains some variables. In

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

particular, these variables are added:

- *modeller* : The name and version of the modeller exporting the scene. For example "*3ds Max 8.0*"
- *assetPath* : The full path of the asset being exported. For example "*c:\scenes\myScene.max*"
- *assetFolder* : The folder where the asset being exported is stored. For example "*c:\scenes*"
- *asset* : The name, without folder or extension, of the asset. For example "*myScene*"
- *selected* : The name of the first selected object, if any. For example "*sphere01*"

The exporters may also add other environment variables specified by the user. When processing an asset, the filter manager will also add another environment variable:

- *configuration* : The name of the configuration being executed. For example "*Default*"
- *infoString*: A string describing the filters executed before the current filter - used, for example, by the Preview Scene filter (which displays that string in the viewport so the user can distinguish multiple preview windows).
- *slot* : A string with a number representing the position of the current filter in the processing.

Environment variables are useful when trying to communicate processing information between user, exporters and filters without modifying the actual content being exported. For example, the Platform Writer filter will pickup and substitute any variables used in its *filename* parameter.

6.7.3.4 Scene Data (hkxScene)

When a scene is exported from a modeler like 3ds Max, Maya and XSI, there is usually one Named Variant, apart from the hkxEnvironment, in the Root Level Container: an `hkxScene` object.

As its name suggests, an `hkxScene` object provides a modeler-agnostic description of the scene (nodes, attributes, meshes, lights, etc..). The role of each of the 3ds Max, Maya and XSI Scene Exporters is to extract information from the scene in the modeler and convert it into the data stored by this object.

The `hkxScene` class and other classes used with it are defined in the `hkscenedata` project. An `hkxScene` object contains:

- General information about the asset : name of the asset and of the modeler, length (in seconds) of the animated data
- A scene graph of nodes (`hkxNode`) : Nodes are described below.
- Arrays of lights, meshes, materials, textures and skin bindings : These objects are also referenced from the actual associated nodes in the scene.
- An *applied transform* matrix : This matrix is modified by the Scene Transform filter, and can be used for processing that depends on spatial data that is not explicit in the scene (and therefore is cannot transformed by the filter).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Scene Nodes (hkxNode)

Nodes (`hkxNode` objects) are stored in a tree structure, starting by a single root node in the `hkxScene` object. Thus, every `hkxNode` has an array (possibly empty) of child `hkxNodes`. Each node also contains:

- A name
- A pointer to the object (mesh, light, camera..) associated with the node. A Variant is used for storage since there are multiple types possible.
- An array of keyframes (4x4 column major matrices) representing the (animated) local transform of the node.
- An array of annotations associated with the node.
- A user properties buffer.
- A selection flag (set to "true" when the node was selected before export)
- An array of attribute groups - described below.

Attribute Groups (hkxAttributeGroup)

A Havok node (`hkxNode`) or material (`hkxMaterial`) can contain any number of attribute groups. Attribute groups are named groups of attributes, and as such they contain:

- A name : usually describes what the object/property its attributes describe. For example : "hkpRigidBody"
- An array of Havok attributes (hkxAttribute) : described below

```
struct hkxAttributeGroup
{
    char* m_name;
    struct hkxAttribute* m_attributes;
    int m_numAttributes;
[....]
};
```

The class also defines multiple methods to easily find attributes (and their values) in the attribute group.

Attributes (hkxAttribute)

Since attributes can be of different types (string, ints, etc), the `hkxAttribute` class is very similar to a Named Variant :

```
struct hkxAttribute
{
    char* m_name;
    hkVariant m_value;
};
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Although `m_value` is an `hkVariant`, and therefore it can reference any object, only certain types are expected. These are:

- `hkxAnimatedFloat`
- `hkxAnimatedMatrix`
- `hkxAnimatedQuaternion`
- `hkxAnimatedVector`
- `hkxSparselyAnimatedBool`
- `hkxSparselyAnimatedEnum`
- `hkxSparselyAnimatedInt`
- `hkxSparselyAnimatedString`

These classes store the animated values for the attribute (if the attribute is not animated, a single value is stored). The animation for floats, matrices, quaternions and vector values is stored as a sequence of regularly sampled values. For booleans, enums, integers and strings, the animation is stored sparsely as pairs `(time, value)`.

Attribute Processing : Attribute Repository

The way in which attributes are associated to nodes and materials during export depends on the choice of modeler (check the *Exporting Custom Data from 3ds Max, from Maya* and from XSI sections for details). Since different modelers support different sets of attribute types (for example, Enums are supported by Maya but not 3ds Max), and ideally we want to have the same abilities in all modelers, it is useful to give the Scene Exporters some hints so that exported attributes can be transformed/complemented with additional information.

For example, since 3ds Max doesn't support enum (it implements them as simple integers), we can provide extra information for specific attributes, where we associate integer values to string representations. Then, during export, this information can be used to export the attribute as an enum.

Sometimes we also want to add extra semantics to attributes so they can be manipulated generically. For example, a float value can represent very different things : a percentage, a mass, a distance.. and depending on what it represents, it should be processed in different ways. For example, if a float attribute represents a distance value, then it should be affected by the Scene Transform filter. But if it represents an absolute value (mass, for example), then it shouldn't.

All this extra information is stored in what is called an *attribute repository*: a set of XML files which contains sets of `hctAttributeDescription` objects, grouped in `hctAttributeGroupDescription` objects. This file is read by the scene exporters during the export process and is used to transform/complement the attributes extracted from the scene. For each attribute that may need to be transformed, it specifies:

- Whether the attribute depends on another (boolean) attribute. If the value of the other attribute is "false", then this attribute will be ignored (won't be exported)

  Note:

  This is still supported, but it's deprecated. From version 4.0, attributes (for example "*centerOfMass*") are automatically ignored if an integer/boolean attribute of the same name, preceded with with the word "*change*" (for example "*changeCenterOfMass*") is found and its value its set to 0/false.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- Whether the attribute should be converted (if necessary) to an specific type. If that type is an enum, then an enum definition (for the case where the attribute should be converted to enum) is also specified in order to assign string values to integer values.
- A scale value for float attributes (defaults to 1.0) - currently used to convert XSI angle attributes (stored in degrees) into radians.
- Flags specifying whether the attribute should be affected by scene transformations, and how.

The relevant files installed by the Havok Content Tools are inside the `AttributeProcessing` folder in the main installation folder. This folder contains subfolders ("max", "maya" and "xsi") with attribute processing information that is specific to each one modellers. Additional XML files files (with their name starting with "hct") can also be added to any of these folder - the exporters will parse them and process attributes accordingly.[9]

Attribute Selection: Exporting built-in properties

To export some of the (huge variety of) built-in properties available in each modeler (for example the transparency of a particular material, the cone angle of a spotlight, and so on) one approach is to create custom attributes in the modeler with names beginning with "hct", and wire them to the desired property. This can be fairly cumbersome. A much more flexible alternative is available – a user-edited XML file which describes which built-in attributes should be exported. There is support for wildcarding, so that for example the entire set of properties associated with a material, light or camera say can be exported easily. In addition, it is possible to specify properties that should NOT be exported, to prevent bloating the exported scene.

During export the exporter searches for relevant XML files, inside the `attributeSelection` folder (in the main installation folder). It will also search for files inside the subfolder "`max`", "`maya`" and "`xsi`", depending on the modeller being used..

An example such file is as follows:

[9] It also installs some files specific to the tutorial in this chapter.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
<?xml version="1.0" encoding="utf-8"?>
<hkpackfile classversion="2">

    <hksection name="__data__">

        <hkobject class="hctAttributeSelectionDatabase" name="#0000">
            <hkparam name="attributeAdditions" numelements="0">

                <!-- -->
                <!-- Note : If you add /uncomment any attribute selection, make sure you update the "
                  numelements" value above -->
                <!-- -->

                <!-- This example selects the "wirecolor" property of the "display" property set-->

                <hkobject>
                    <hkparam name="typeName">display</hkparam>
                    <hkparam name="subTypeName">*</hkparam>
                    <hkparam name="attributeNames" numelements="1">
                        <hkcstring>wirecol</hkcstring>
                    </hkparam>
                </hkobject>

                <!-- This would export all attributes (but commented out here) -->
                <!-- Note that file size will be greatly increased. -->

                <!--
                <hkobject>
                    <hkparam name="typeName">*</hkparam>
                    <hkparam name="subTypeName">*</hkparam>
                    <hkparam name="attributeNames" numelements="1">
                        <hkcstring>*</hkcstring>
                    </hkparam>
                </hkobject>
                -->

            </hkparam>

            <hkparam name="attributeRemovals" numelements="0">

                <!-- Attributes that should NOT be exported can be added here as well - the format is the
                  same as for additions -->

            </hkparam>
        </hkobject>
    </hksection>
</hkpackfile>
```

Notice that the XML file is nothing more than an XML-serialized version of an object of type `hctAttributeSelectionDatabase` (defined in `hctSceneExport/AttributeSelection/hctAttributeSelection.h`), containing two arrays (`attributeAdditions` and `attributeRemovals`) of objects of type `hctAttributeSelection`.

The attribute *additions* section specifies which attributes from the scene graph should be added to the exported scene, and the section for attribute *removals* specifies which attributes should be removed (removals are interpreted after additions, so they can be used to complement each other). Note that the `numelements` values in the `<hkparam>` tags must be updated by hand as new entries are added.

Each one of these sections are arrays of entries of type `hctAttributeSelection`, which contain three members. These members are interpreted differently depending on the modeller being used:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Member	3ds Max Interpretation	Maya Interpretation	XSI Interpretation
`typeName` (string)	Type name of the Modifier, Material, Custom Attribute or Base Object. For example, "*GeoSphere*" for geospheres.	Type name of the Maya Node. For example, "*PointLight*" for point lights.	Type name of the XSI Property (parameter set). For example, "Display" for the Display parameter set, "customparamset" for any User-Defined parameter set (like Havok's).
`subTypeName` (string)	Name of the Parameter Block containing the parameter. This name may or not be visible in the UI.	Not used	Name of the XSI Property (parameter set) object. This is useful in order to distinguish between user-defined parameter sets (since they all have the same type name)
`attributeNames` (array of strings)	Internal (MAXScript) names of the parameters.	Names of the Maya Attributes.	Names of the XSI parameters.

Table 6.11: Attribute Selection for each modeller

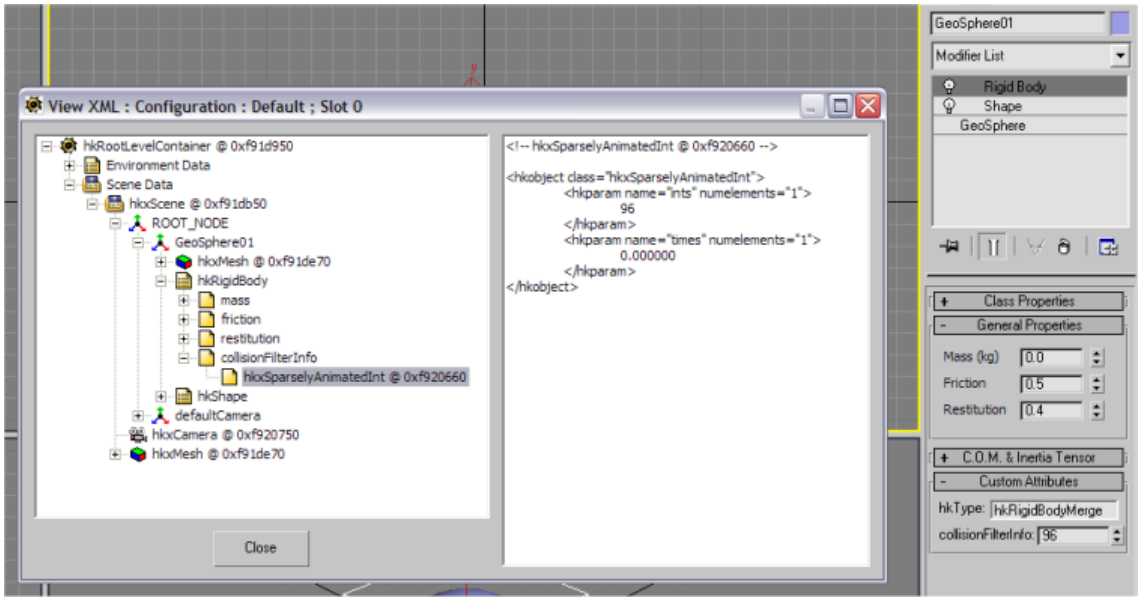
For all the string entries, an asterisk can be used as a wildcard (by itself or at the end of a name).

More Attribute Processing

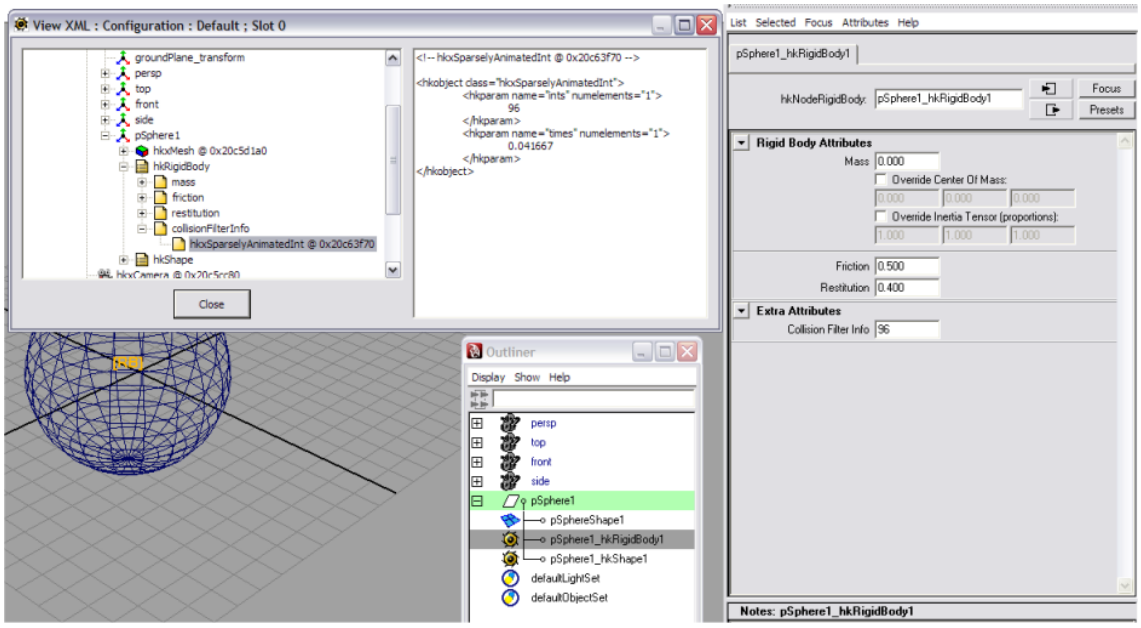
There is also more attribute processing common to all scene exporters which is not driven by XML files / `hkxAttributeDescriptor` objects. In particular:

- Attribute groups named "*hkXXXXXMerge*" will be merged into any previous attribute group of name "*hkXXXXX*" .

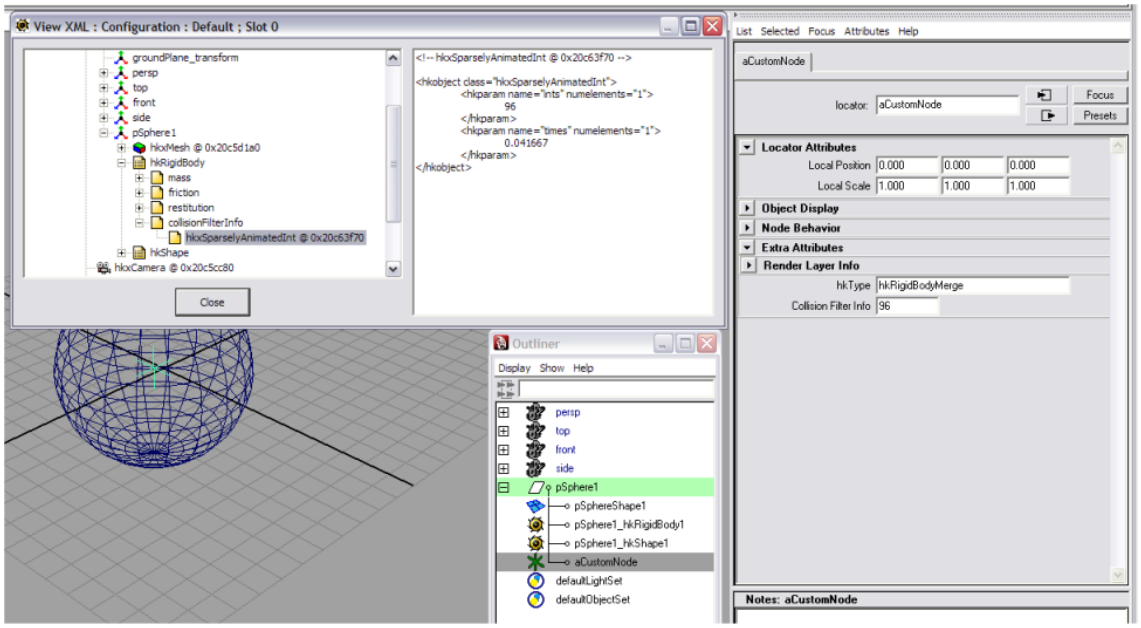
  This allows extending attribute groups already defined with more attributes. For example, in 3ds Max, you can add extra attributes to the *hkpRigidBody* attribute group of an object by adding a Custom Attribute group named *hkRigidBodyMerge* (hkType == "`hkRigidBodyMerge`") to the hkpRigidBody modifier, rather than modifying the source of the hkpRigidBody modifier or creating a completely new modifier / custom attribute:



  In Maya, adding user-defined attributes to a Havok node will automatically add them to the same attribute group (the exporter doesn't distinguish between user-defined and built-in attributes).

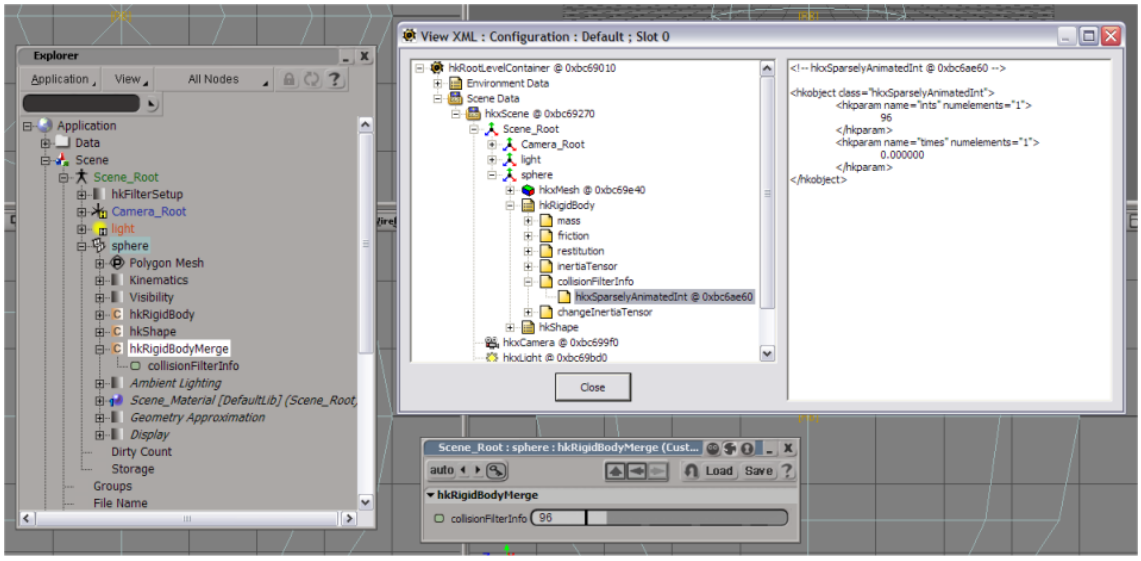


But if you add the extra attributes to a different node (which would usually create two attribute groups) you can ensure the the attribute groups are merged by following the same mechanism.



In XSI you can add custom parameters to an existing parameter set; but they usually won't show in the UI since the original parameter set is not aware of them. Therefore adding a new parameter set is the common option. As with 3ds Max and Maya, you can ensure the attributes are merged into a single group by naming the second group with the "Merge" suffix :

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Of course, the attribute will be ignored by the filters unless it's an attribute they support - you may want to extend or create new filters to interpret any newly exported attributes.

- Attribute pairs with names *xxxx* + *changeXxxx* (where *xxxx* can be any name, and *changeXxxx* is a boolean or integer attribute) are treated as one : if the *changeXxxx* attribute is set to false or 0, the *xxxx* attribute is ignored. The *changeXxxx* attribute is always ignored (not exported).

  This is used extensively for the physics tools in order to provide optional properties that, if not set, it will be calculated during processing (by a filter) and not in the modeller. For example, the center of mass and the inertia tensor of rigid bodies are processed in this way.

- Float attribute triplets named *xxxx_X*, *xxxx_Y* and *xxxx_Z* will be converted to a single vector attribute (named *xxxx*) with those three components. This is used in particular formodelers that don't have built-in support for vector attributes.

- Pairs of attributes with names *xxxx_Rotation* and *xxxx_Translation* (where *xxxx_Rotation* is a quaternion attribute and *xxxx_Translation* is a vector attribute) will be merged into a single attribute (named xxxx) of type matrix, built from those rotation and translation components. Again, this is used for platforms that don't have built-in support for matrix attributes.

Since this functionality is common to all scene exporters, it is implemented in the `hksceneexport` library. In particular, the `hctAttributeProcessingUtil` (in the `hksceneexport` library) utility class is used (check the Reference Manual for details).

6.7.3.5 Physics Data (hkpPhysicsData)

This type is also found in the Root Level Container when processing using Physics Filters. It contains rigid bodies and constraints grouped into Physics Systems (`hkpPhysicsSystem` objects); check the *Havok Physics SDK User Guide* for more information about Physics Systems. It also (optionally) contains information about the physical world (`hkpWorldCinfo`).

```
class hkpPhysicsData
{
    [.....]
        class hkpWorldCinfo* m_worldCinfo;

        hkArray<class hkpPhysicsSystem*> m_systems;
};
```

6.7.3.6 Animation Container (hkaAnimationContainer)

Another commonly used type found in the Root Level Container is an Animation Container (`hkaAnimationContainer`). This container is used by the Animation Filters to access and store objects related to animation. The class itself just contains arrays of basic animation types:

```
class hkaAnimationContainer
{
        class hkaSkeleton** m_skeletons;
        hkInt32 m_numSkeletons;

        class hkaSkeletalAnimation** m_animations;
        hkInt32 m_numAnimations;

        class hkaAnimationBinding** m_bindings;
        hkInt32 m_numBindings;

        class hkaBoneAttachment** m_attachments;
        hkInt32 m_numAttachments;

        class hkaMeshBinding** m_skins;
        hkInt32 m_numSkins;
};
```

6.7.3.7 Other Root-Level Data

Some asset data may be stored outside scene data, animation container or physics data objects. For example, ragdoll instances created by the Create Rag Doll filter, or skeleton mappings created by the Create Mapping filter are stored using their own custom Named Variants in the Root Level Container.

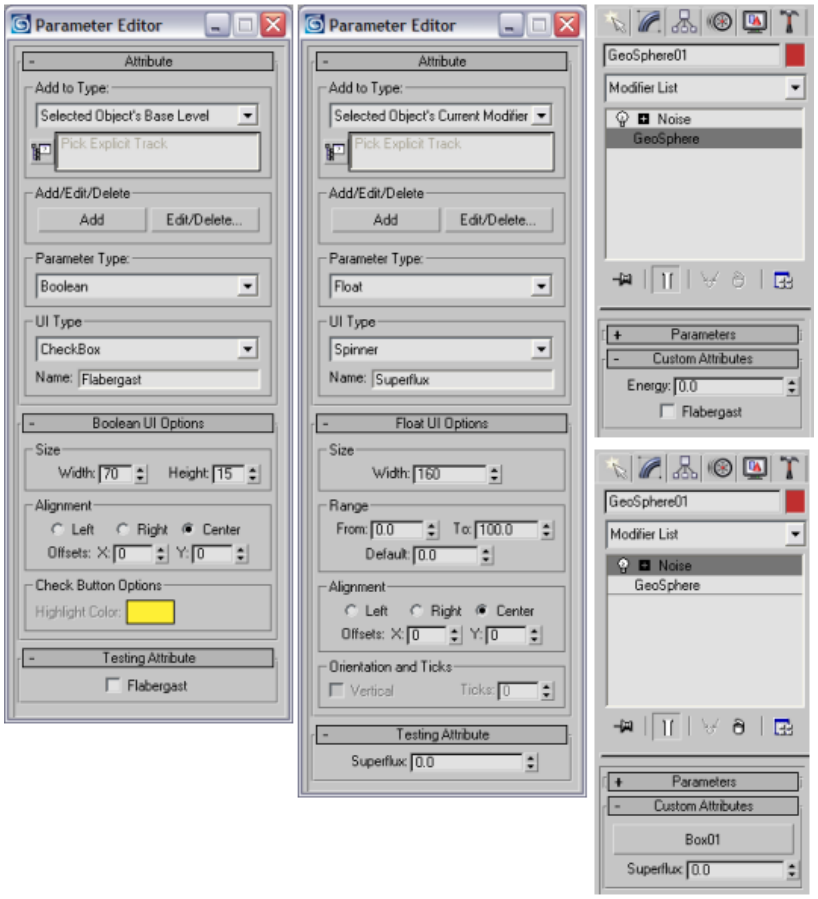
6.7.4 Extending the 3ds Max Tools

6.7.4.1 Exporting Custom Data

The best way to export custom data from 3ds Max is by exporting extra attributes associated with Havok nodes. This is how the Havok Physics Tools add extra rigid-body and constraint information to nodes (which is later interpreted by filters). There are two ways to associate attributes to nodes so they are exported:

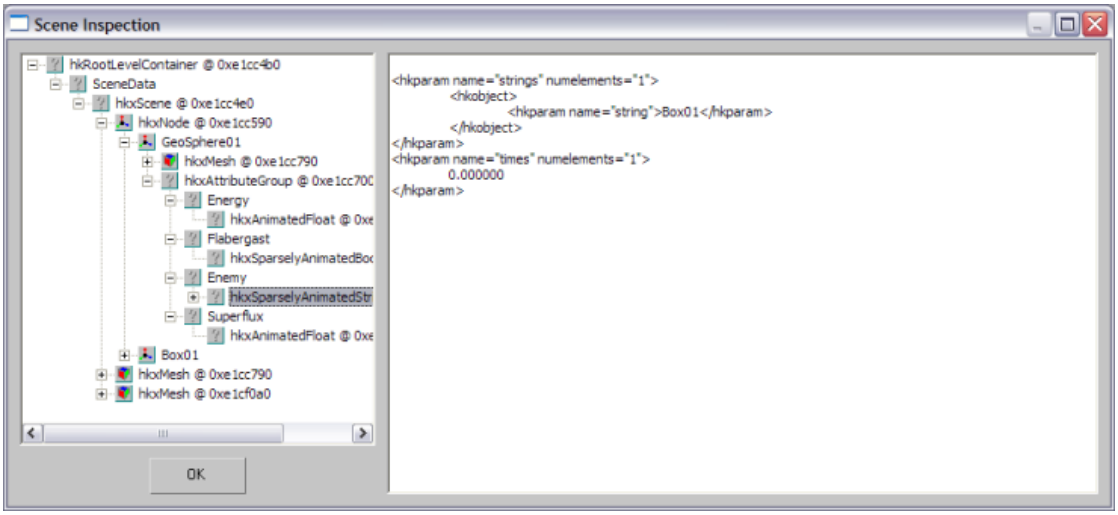
- *3ds Max Custom Attributes* : The easiest way; you can do this through the 3ds Max menus or

MAXScript. Custom Attributes associated to an object or modifier will always be exported as Havok attributes associated with the node linked to the object/modifier. Custom Attributes added to a 3ds Max material will be exported as Havok attributes of the associated hkxMaterial.



- *Custom Modifier / Objects* : If a parameter block's name starts with "`hk`" then any parameters within that block will be exported as attributes associated with the Havok node or material. Therefore, defining and applying a custom object, material or modifier with a parameter block named "`hk...`" is another way to associate extra attributes to a node or material (this is how attributes used by the Physics Tools - within the modifiers - are exported).
- *Attribute Selection* : You can also export built-in or custom attributes associated with nodes or materials (regardless on how they are named) by using attribute selection XML files.

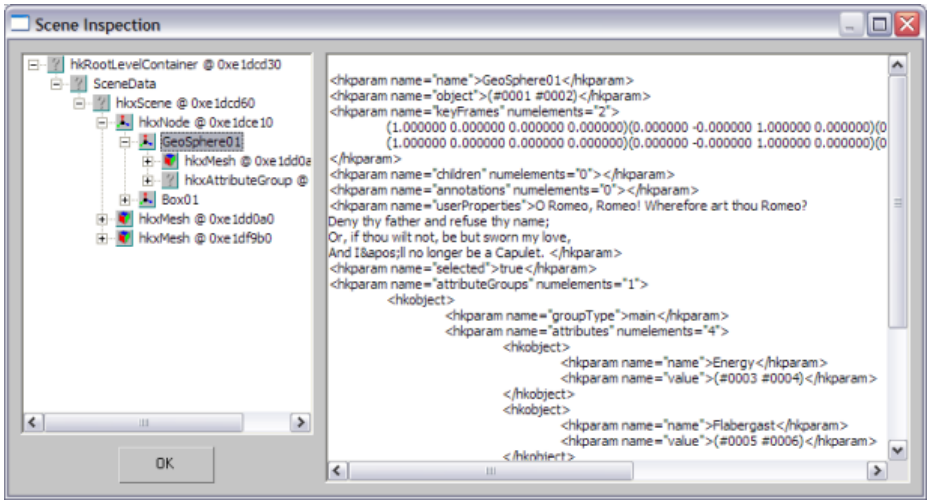
In all cases, the name of the attribute group holding the exported attributes will be taken from the name of the parameter block (usually "main" for Custom Attributes created through 3ds Max UI). However if an "`hkType`" string parameter is found then its value will be used instead to name the attribute group.



User Properties

The 3ds Max Scene Exporter also supports *User-Defined Properties*. They are exported as a single string (`m_userProperties`) associated with the Havok node (`hkxNode`).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



6.7.4.2 MAXScript Access

All properties and methods of the Scene Exporter and the Physics Tools for 3ds Max are exposed to MAXScript.

Discovering Properties and Interfaces

3ds Max uses two different mechanisms to expose objects to MAXScript : properties and interfaces.

Most non-static objects (modifiers, helpers, etc) expose their UI parameters through properties. You can query MAXScript for all the properties of an object by using the `showProperties` method:

```
ragdollModifier = \$.modifiers[#Rag_Doll]
Havok_Rag_Doll_Constraint:Rag Doll

showProperties ragdollModifier
  .childSpaceTranslation (Child_Space_Translation) : point3
  .childSpaceRotation (Child_Space_Rotation) : matrix3
  .parentSpaceTranslation (Parent_Space_Translation) : point3
  .parentSpaceRotation (Parent_Space_Rotation) : matrix3
  .constrainTo (Constrain_To) : integer
  .parent : node
  .childSpaceTranslationLocked (Child_Space_Translation_Locked) : boolean
  .childSpaceRotationLocked (Child_Space_Rotation_Locked) : boolean
  .parentSpaceTranslationLocked (Parent_Space_Translation_Locked) : boolean
  .parentSpaceRotationLocked (Parent_Space_Rotation_Locked) : boolean
  .coneAngle (Cone_Angle) : angle
  .planeAngleMin (Plane_Angle_Min) : angle
  .planeAngleMax (Plane_Angle_Max) : angle
  .twistMin (Twist_Angle_Min) : angle
  .twistMax (Twist_Angle_Max) : angle
  .maxFrictionTorque (Maximum_Friction_Torque) : float
  .motorType (Motor_Type) : integer
  .displayTwistLimits (Display_Twist_Limits) : boolean
  .displayPlaneLimits (Display_Plane_Limits) : boolean
false
```

You can easily query or change a property in an object by using the dot (.) notation:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
ragdollModifier.twistMin
-10.0
ragdollModifier.twistMin = -30.0
-30.0
```

Interfaces are used for exposing methods and non-UI (derived) parameters. They are also used by static objects like utilities, where properties cannot be used.

You can query MAXScript for the interfaces available for an object by using the `showInterfaces` method:

```
ragdollModifier = \$.modifiers[#Rag_Doll]
Havok_Rag_Doll_Constraint:Rag Doll

showInterfaces ragdollModifier
  Interface: hkConstraintModifierInterface
   Properties:
    .childSpaceInWorld : matrix3 : Read|Write
    .parentSpaceInWorld : matrix3 : Read|Write
   Methods:
    <boolean>resetChildSpaceRotation()
    <boolean>resetChildSpaceTranslation()
    <boolean>resetParentSpaceRotation()
    <boolean>resetParentSpaceTranslation()
   Actions:
OK
```

As you can see, properties and interfaces complement each other.

MAXScript Access to the Scene Export Utility

The Scene Export Utility is accesible through a static MAXScript object named `hctSceneExportUtility`. All the properties and methods available are exposed to MAXScript through the use of interfaces:

```
showInterfaces hctSceneExportUtility
  Interface: hkSceneExportUtilityInterface
   Properties:
    .useOptionsFile : boolean : Read|Write
    .optionsFile : string : Read|Write
    .environmentVariables : string : Read|Write
    .exportVisibleOnly : boolean : Read|Write
    .exportSelectedOnly : boolean : Read|Write
    .exportMeshes : boolean : Read|Write
    .exportMaterials : boolean : Read|Write
    .exportAttributes : boolean : Read|Write
    .exportAnnotations : boolean : Read|Write
    .exportLights : boolean : Read|Write
    .exportCameras : boolean : Read|Write
    .addDefaultCamera : boolean : Read|Write
    .animatedDataExport : enum : Read|Write
       animatedDataExport enums: {#current|#currentRange|#specificRange}
    .animationStart : time : Read|Write
    .animationEnd : time : Read|Write
   Methods:
    <boolean>exportScene <boolean>batchMode
    <string>getVersionString()
    <integer>getVersionNumber()
   Actions:
  Interface: hctSceneExportUtilityActions
   Properties:
   Methods:
    <boolean>doExport()  -- Action Interface
   Actions:
    Category: Havok_Scene_Export_Utility; Action: Export_Scene; Shortcut: -- none defined --
OK
```

The properties exposed are those described in the 3ds Max Scene Export Options section and visible in the UI, with the addition of two methods:

- `exportScene batchMode` : Exports and process the scene. If `batchMode` is TRUE, the scene is exported and processed silently (no dialogs are shown).
- `getVersionString()` : Returns a string describing the current version of the Havok Content Tools / 3ds Max Scene Exporter (for example "Build 4.0.0.3 Beta")
- `getVersionNumber()` : Returns an integer value representing the current build of the Havok Content Tools. The number is built in hexadecimal, using two digits for each of the four components of the version number (major, minor, point, build). For example 4.0.0.3 = 0x04000003 = 67108867 in decimal.
- `doExport()`: Equivalent to pressing the **Export** button , or calling "`exportScene false`".

Example

```
-- Change some export options
if (hctSceneExportUtility.animatedDataExport != #specificRange) then
(
    hctSceneExportUtility.animatedDataExport = #specificRange
    hctSceneExportUtility.animationStart = 10f
    hctSceneExportUtility.animationEnd = 70f
)

-- Use filter and options in HKO file
hctSceneExportUtility.useOptionsFile = true
hctSceneExportUtility.optionsFile = "c:\\data\\compressionOptions.hko"

-- Export in batch mode
hctSceneExportUtility.exportScene true
```

MAXScript Access to Other Havok Objects in 3ds Max

The following table gives the names of some of the havok objects exposed through MAXScript. Given an instance of them, you can discover their properties and/or interfaces through the methods described above.

Name	Type	Description
`hctSceneExportUtility`	Static Object (Utility)	Described above. Exposes functionality and options in the 3ds Max Scene Exporter
`hctToolbarGUP`	Static Object (GUP)	Exposes functionality in the Havok Content Tools toolbar.
`hctRigidBodyModifier`	Modifier Type	Rigid Body Modifier
`hctShapeModifier`	Modifier Type	Shape Modifier
`hctBallAndSocketConstraintModifier`	Modifier Type	Ball And Socket Contraint Modifier
`hctHingeConstraintModifier`	Modifier Type	Hinge Constraint Modifier
`hctRagdollConstraintModifier`	Modifier Type	Ragdoll Constraint Modifier
`hctStiffSpringConstraintModifier`	Modifier Type	Stiff Spring Constraint Modifier
`hctPrismaticConstraintModifier`	Modifier Type	Prismatic Constraint Modifier
`hctWheelConstraintModifier`	Modifier Type	Wheel Constraint Modifier
`hkBoneProxyCA`	Custom Attribute Type	Used by the Rag Doll Toolbox to keep bone-proxy associations
`hkChainProxyCA`	Custom Attribute Type	Used by the Rag Doll Toolbox to keep chain-proxy associations
`hctFxRigidBodyModifier`	Modifier Type	FX Rigid Body Modifier
`hctFxInstanceModifier`	Modifier Type	FX Instance Modifier
`hctFxClothModifier`	Modifier Type	FX Cloth Modifier
`hctFxSphereHelper`	Helper Type	FX Sphere Helper - The helpers used to represent FX Spheres in the viewports.
`hctFxSpheresUtility`	Static Object (Utility)	FX Sphere Distribution Utility

Table 6.12: Havok Objects and Types exposed to MAXScript

Examples

```
-- Export
hctSceneExportUtility.doExport()
true

-- Add a button to the toolbar and dock it
hctToolbarGUP.addMacro "Modifiers" "Taper"
OK
hctToolbarGUP.dock()
OK

-- Add a shape modifier to the selection
modPanel.addModToSelection (hctShapeModifier ())
OK
```

Customizing the 3ds Max Rag Doll Toolbox

The Rag Doll Toolbox has been implemented in MAXScript. In order to facilitate customization, some of the behaviours and default options for this tool have been isolated in a single MAXScript file, `hvkTools_Customizable.ms`, which can be found in the `"3dsmax/scripts/havokPhysics/ragdollTools" folder`.

Among the customizable behaviours exposed in that file are:

- Default values for all the options (bone scale, axis, templates folder, etc)
- Default rigid body properties (mass, restitution, friction) for proxy rigid bodies
- Default naming methods for new proxies and joints.

6.7.4.3 C++ Access

It is also possible to interface with the 3ds Max-based Havok Content Tools component through the use of C++. Full source is provided for all the tools (subject to the type of license agreement), but in order to reduce the amount of dependencies and facilitate C++ integration, full abstract interfaces to the Havok objects are exposed in a single, headers-only, project : `hkmaxfpinterfaces` (availability of this project is subject to the type of license agreement).

This project contains:

- C++ Interfaces (abstract classes) for all Havok types in 3ds Max
- Class IDs and Function Publishing Interface IDs for all those types
- Parameter Block IDs and Parameter IDs for IParamBlock2-based parameters
- Other global IDs and Enums.

By using the headers in this project you can interface with all the Havok objects in 3ds Max without statically linking with any libraries. For details on how to use Function Publishing Interfaces, ClassIDs and Parameter Blocks in 3ds Max, please check the 3ds Max SDK documentation.

Example

```
#include <ContentTools/Max/MaxFpInterfaces/Export/hctSceneExportUtilityInterface.h>

void exportCurrentFrameInBatchMode()
{
  hkSceneExportUtilityInterface* theUtility =
    (hkSceneExportUtilityInterface*) GetInterface ( UTILITY_CLASS_ID,
                                                    HK_SCENE_EXPORT_UTILITY_CLASS_ID,
                                                    HK_SCENE_EXPORT_UTILITY_FPINTERFACE_ID);

  theUtility->iSetAnimatedDataExport (ADE_CURRENT_FRAME);
  theUtility->iExportScene (TRUE);
}
```

6.7.5 Extending the Maya Tools

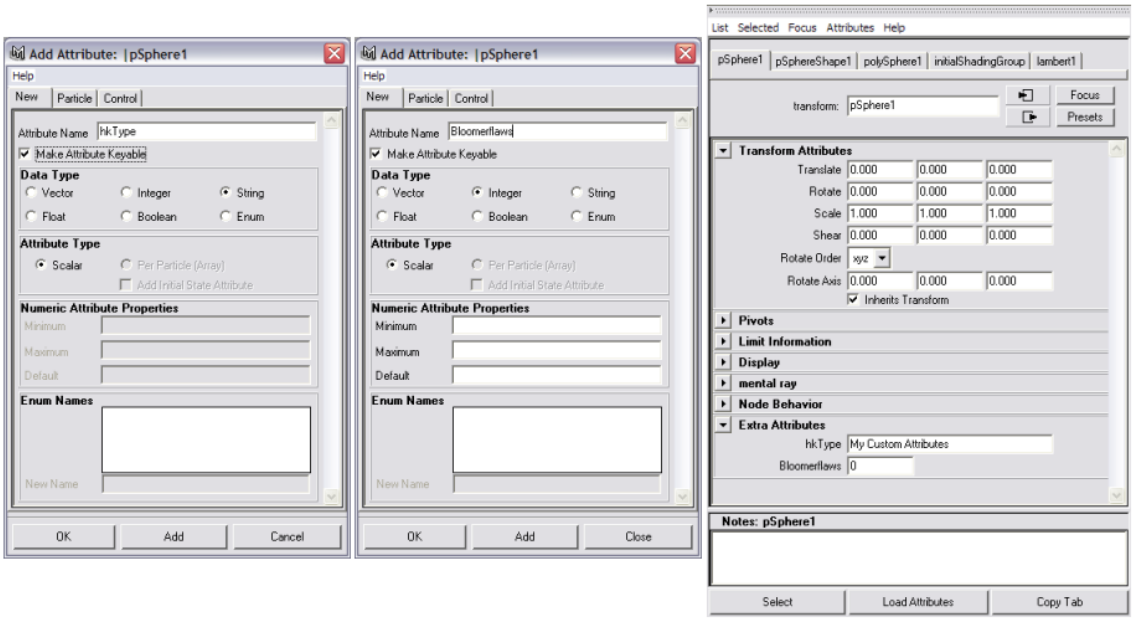
6.7.5.1 Exporting Custom Data

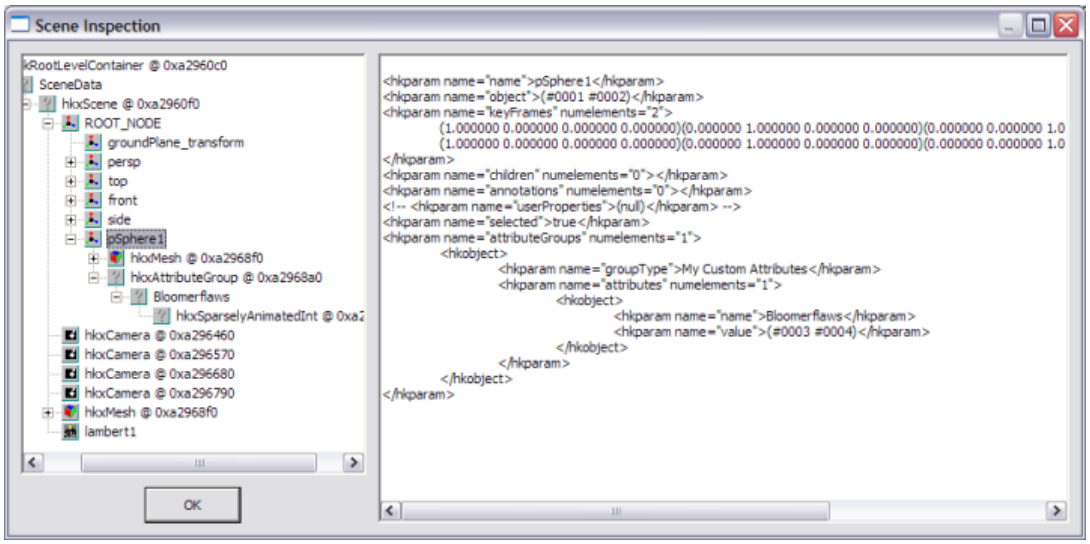
The best way to attach custom data to nodes within Maya is through the use of custom attributes. Since every Maya node has a multitude of built-in attributes as well as any additional custom attributes, exporting all of them would bloat the scene data considerably. Instead, the following method for tagging and exporting groups of attributes is used by the Havok Exporter:

1. Given a transform node which is being exported, gather a set of 'interesting' nodes containing:
   - the transform node itself
   - all of it's non-transform child nodes
   - any other nodes which are connected from the transform node's `message` attribute to an attribute named `hkMessage`
2. For each of these interesting nodes, step though all of the visible attributes looking for string attributes named `hkType*` (e.g. `hkType`, `hkTypeFoo`, `hkTypeBar`). The exporter interprets such an attribute as the beginning of an exportable group, so for every one of these it finds, a new attribute group is created in the exported HKX node. The name of the group is taken from the string value of the attribute, and each subsequent visible attribute in the current Maya node gets exported as a member of the group.

Once custom attributes have been exported as part of a HKX node in the scene data, they are then available to be used in the filter pipeline. Generally some custom filter work will be required to take that exported data and process it further, depending on what that data is intended for.

For example: any custom DAG node which contains at least an `hkType*` attribute, and is a child of a transform node, will have it's subsequent attributes exported as a group within the transform's HKX node. This is exactly how the rigid body and constraint information generated by the Havok Physics Tools for Maya gets exported. The physics filters then take that data and use it to runtime physics objects.





Attribute Selection

In addition to the above, version 4.5 introduces the ability to export explicitely selected attributes by using attribute selection XML files.

6.7.5.2 MEL Access

A large amount of the Maya-based component of the Havok Content Tools have been written using MEL. Those components (commands, nodes and attributes) written in C++ are automatically exposed to MEL by Maya. Therefore, using MEL is the best option for extending and integrating with the Havok Tools in Maya.

MEL Access to the Scene Exporter

The Maya Scene Exporter can be invoked through MEL using the `hkCmdExportScene` command or the `hkProcExporter_export()` procedure:

- The **hkCmdExportScene** MEL command invokes the export process. This command can take multiple flags, which explicitly set the different export options.
- The `hkProcExporter_export()` MEL procedure does not take any parameters. It invokes the `hkCmdExportScene` command with flags reflecting the current export options in the scene. Calling this procedure is equivalent to pressing the **Export** button in Maya.

The hkCmdExportScene command

The following table show the flags available on the **hkCmdExportScene** command. Notice how they reflect the options exposed by the export options dialog.

Flag	Short Flag	Argument Type	Default
`-exportMeshes`	`-ms`	boolean	`true`
`-exportMaterials`	`-mt`	boolean	`true`
`-exportAttributes`	`-att`	boolean	`true`
`-exportAnnotations`	`-ann`	boolean	`true`
`-exportLights`	`-l`	boolean	`true`
`-exportCameras`	`-c`	boolean	`true`
`-visibleOnly`	`-v`	no argument	flag not present
`-selectedOnly`	`-s`	no argument	flag not present
`-startTime`	`-t1`	time (i.e., frame number)	playbackOptions -q -minTime;
`-endTime`	`-t2`	time (i.e., frame number)	playbackOptions -q -maxTime;
`-optionsFile`	`-o`	string	flag not present
`-useRotatePivot`	`-rp`	boolean	true
`-batchMode`	`-b`	no argument	flag not present
`-environmentVariables`	`-ev`	string	empty

The `batchMode` flag is only available to **hkCmdExportScene** - it is not exposed by the export options dialog. If this flag is specified, the command will export and process the scene silently, without opening any dialogs. It is most useful for batch processing assets (in conjuntion with the -o flag, for example).

```
// Export with the current options
hkProcExport_export();

// Export a specific frame range
hkCmdExportScene -t1 20 -t2 35;

// Export the current frame only
hkCmdExportScene -t1 ‘currentTime -q‘ -t2 ‘currentTime -q‘;

// Export silently using an options file
hkCmdExportScene -o "c:/data/compressionOptions.hko" -b;
```

The hctCmdGetVersion command

The Havok Scene Exporter for Maya also defines a command to query the current build and version of the Havok Content Tools: hctCmdGetVersion

Flag	Short Flag	Argument Type	Default
`-string`	`-s`	no argument	`flag not present`

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

If the `-string` (`-s`) flag is specified, the command will return a string representation of the current version and build. If the flag is not present, an integer representation will be returned. The number is constructed in hexadecimal, using two digits for each of the 4 components (major, minor, point, build) of the version number. For example, version 4.0.0.3 would be represented as 0x04000003, which in decimal is 67108867.

```
hctCmdGetVersion;
// Result: 67108867 //

hctCmdGetVersion -string;
// Result: 4.0.0.3 (4.0.0 Beta) //
```

MEL Access to other Havok Objects

Apart from the `hkCmdExportScene` command, the main Havok objects in Maya are nodes and can therefore be fully accessed through their attributes. The following table enumerates the names of the Havok nodes in Maya:

Name	Description
`hkNodeRigidBody`	Holds Rigid Body data
`hkNodeRigidBodyManip`	Manipulator for the Rigid Body Node
`hkNodeShape`	Holds Shape data
`hkNodeShapeManip`	Manipulator for the Shape Node
`hkNodeBallAndSocketConstraint`	Holds Ball-And-Socket Constraint Data
`hkNodeBallAndSocketConstraintManip`	Manipulator for the Ball-And-Socket Constraint Node
`hkNodeHingeConstraint`	Holds Hinge Constraint Data
`hkNodeHingeConstraintManip`	Manipulator for the Hinge Constraint Node
`hkNodeRagDollConstraint`	Holds Ragdoll Constraint Data
`hkNodeRagDollConstraintManip`	Manipulator for the Ragdoll Constraint
`hkNodeStiffSpringConstraint`	Holds Stiff Spring Constraint Data
`hkNodeStiffSpringConstraintManip`	Manipulator for the Stiff Spring Constraint
`hkNodePrismaticConstraint`	Holds Prismatic Constraint Data
`hkNodePrismaticConstraintManip`	Manipulator for the Prismatic Constraint
`hkNodeWheelConstraint`	Holds Wheel Constraint Data
`hkNodeWheelConstraintManip`	Manipulator for the Wheel Constraint
`hkNodeOptions`	Filter options are stored with the scene by using this node.
`hkNodeCapsule`	Capsule Mesh Node
`hkNodeDecomposition`	Used internally to decompose matrices into S,R,T components.
`hkNodeFxRigidBody`	Holds FX Rigid Body data.
`hkNodeFxInstance`	Holds FX Instance data.
`hkNodeFxSphere`	Displays an FX Sphere representation.
`hkNodeFxCloth`	Holds FX Cloth data.

Table 6.13: Havok Nodes in Maya

The hctCmdCreateFxSpheres command

This command is used by the FX Sphere Distribution utility - it attempts to automatically distribute FX Spheres on a given geometry.

Flag	Short Flag	Argument Type	Default
`-distribution`	`-dt`	`string` : choice of `"convex"`, `"box"`, `"sphere"` or `"capsule"`	`"convex"`

The `-distribution` (`-dt`) flag specifies what type of distribution to attempt. Notice that the command will always create new FX Spheres, so you should remove any previous spheres if you are trying to redistribute them/

Customizing the Maya Rag Doll Toolbox

The Maya Rag Doll Toolbox has been implemented in MEL. In order to facilitate customization, some of the behaviours and default options for this tool have been isolated in a single MEL file, `hkRagDollTools_Custom.mel`, which can be found in the `"scripts/ragdollTools"` folder.

Among the customizable behaviours exposed in that file are:

- Default values for each of the rag doll toolbox options (bone scale, templates folder, etc)
- Default rigid body properties (mass, restitution, friction) for proxy rigid bodies
- Default naming methods for new proxies and constraints.

6.7.5.3 C++ Access

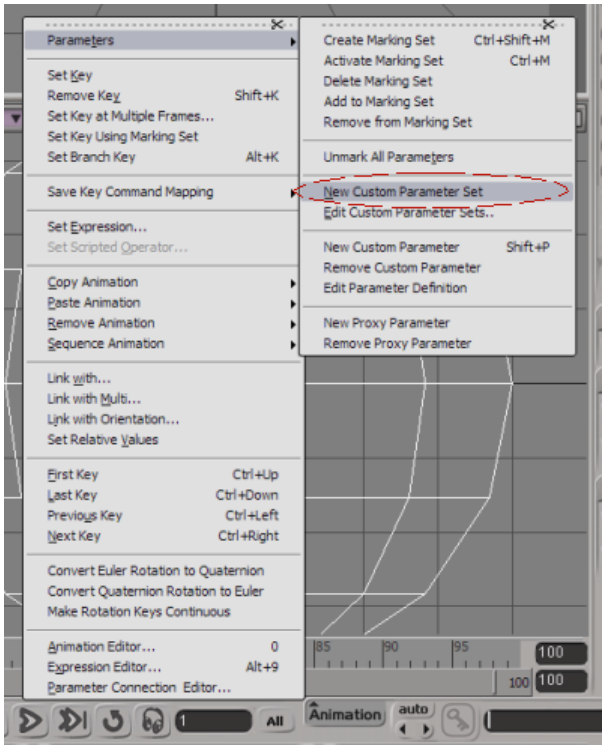
Most communication between Maya plugins is done through connections in Maya's DG, and therefore there is very little need for C++ exposure of plug-ins. For those requiring C++ access, full source is provided for the Havok Content Tools. The `"hctMayaNodeIds.h"` header file, in the `"hkmayasceneexport"` project , contains a full reference of the IDs associated with individual Havok nodes (availability of this file depends on the type of license agreement).

6.7.6 Extending the XSI Tools

6.7.6.1 Exporting Custom Data

The best way to export custom data from XSI is by exporting extra attributes associated with Havok nodes or materials. This is how the XSI Physics Tools add extra rigid body and constraint information to nodes (which is later interpreted by filters).

In XSI, scene nodes and materials store custom information in *Custom Parameter Sets* (also called *Properties*), which contain many individual parameters. Users and third-party tools can define and add their own custom parameters sets to store custom data. Using the XSI UI, you can easily create your own by using the **New Custom Parameter Set** and **New Custom Parameter** commands in the **Animation** menu:

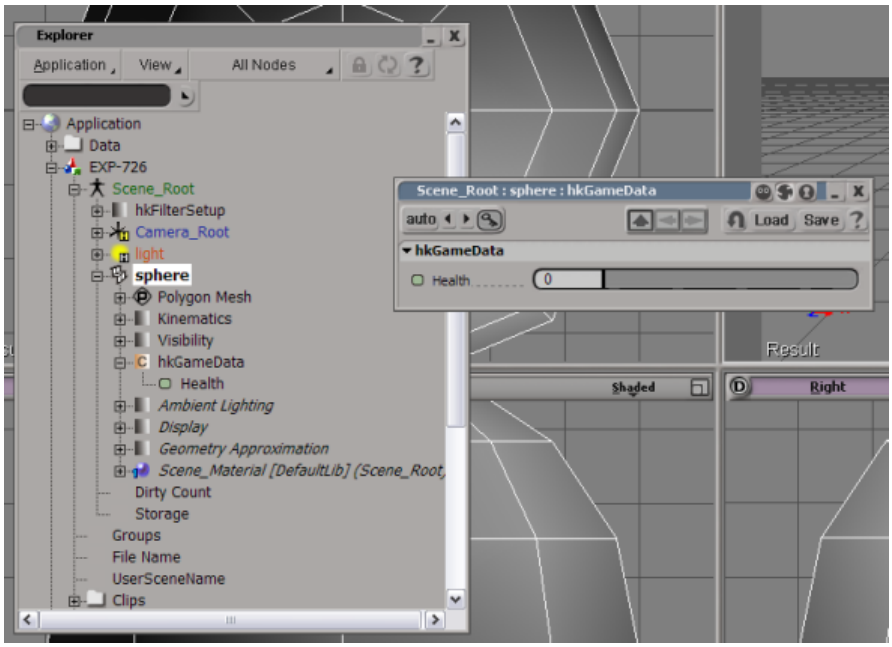


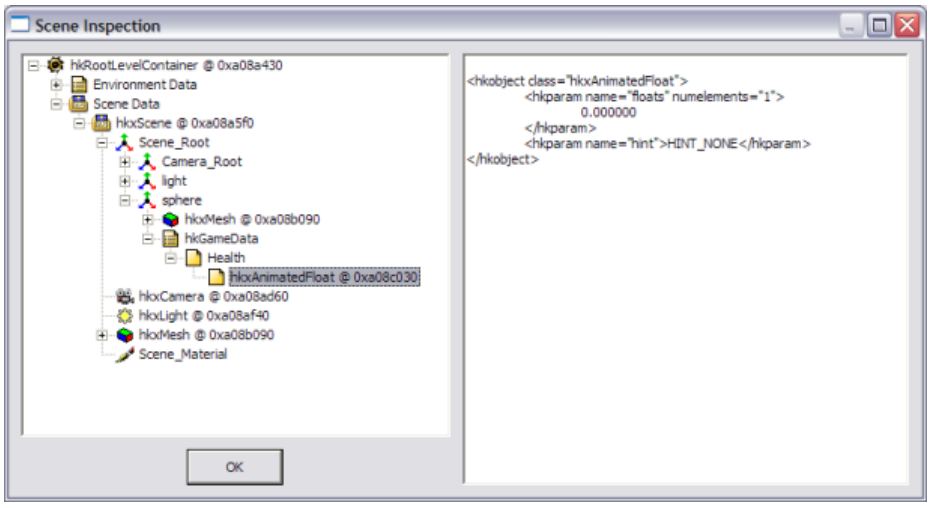
Since custom parameter sets are used extensively for many purposes, the Havok XSI Scene Exporter will export custom parameter sets that meet certain criteria:

- Either the type or the name of the custom parameter set must start with "*hk*"
- The custom parameter is not be an annotation (which are exported separately)

If the above criteria are met, the custom parameter set will be exported as an attribute group. For custom parameter sets which are registered XSI plugins, the name of the exported attribute group will be taken from the type of the parameter set. For custom parameter sets defined using XSI UI, the name is taken from the actual name of the parameter set (since its type is always "*customparamset*").

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.





Attribute Selection

In addition to the above, version 4.5 introduces the ability to export explicitely selected attributes (either in custom or built-in parameter sets) by using attribute selection XML files.

6.7.6.2 Scripting and C++ Access

XSI doesn't have its own scripting language - it's API support multiple languages : C++, VBScript, JScript, Perlscript and Python.

The functionality implemented by the Havok Tools in XSI can be split into two sets

- Data : The majority of our data is stored in XSI using custom-defined parameter sets (also called

properties) and some operators. XSI provides built-in functionality to access these objects to all supported languages.

- Actions : Actions are exposed to XSI through the use of Commands. Once a command is defined, it is immediately available through all supported languages.

Parameter Sets

Havok data is stored in custom parameter sets. It is very easy to access data from custom parameter sets - the easiest way is to use XSI's **GetValue** and **SetValue** commands (available in any language):

```
// JScript example
a = GetValue ("sphere.hkpRigidBody.mass");

' VBScript example
a = GetValue ("sphere.hkpRigidBody.mass");

# Python example
a = Application.GetValue ("sphere.hkpRigidBody.mass")

# PerlScript example
\$a = \$Application->GetValue("sphere.hkpRigidBody.mass");
```

In the case of C++, you can also use those two commands through the **Application::ExecuteCommand()** method, although this is a little convoluted:

```
// C++ example using commands
CValueArray args(1);
args[0] = L"sphere.hkpRigidBody.mass";
CValue retval;
const CStatus status = Application().ExecuteCommand( L"GetValue", args, retval );
const double a = (double) retval;
```

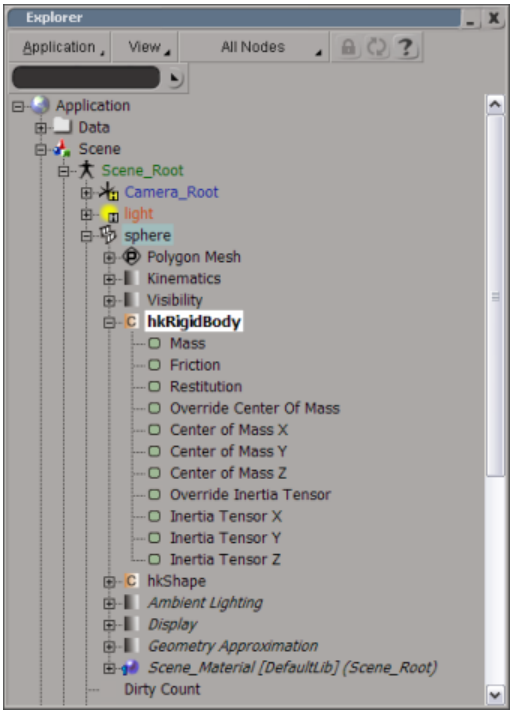
XSI's API can also work with nodes, parameters and parameter sets as actual objects - this makes code, particularly in C++, easier to work with:

```
//
// C++ Example using API objects
//

// Access the parameter directly and query for its value
{
    CRef paramRef; paramRef.Set(L"sphere.hkpRigidBody.mass");
    Parameter massParam(paramRef);
    const double a = (double) massParam.GetValue();
}

// Access the property and query the value one of its parameters
{
    CRef propRef; propRef.Set(L"sphere.hkpRigidBody");
    Property rigidBodyProp(propRef);
    const double a = (double) rigidBodyProp.GetParameterValue(L"mass");
}

// Access the object, find the property, get the parameter, then query its value
{
    CRef objectRef; objectRef.Set(L"Sphere");
    X3DObject sphereObject (objectRef);

    CRef propRef;
    sphereObject.GetProperties().Find(L"hkpRigidBody", propRef);

    Property rigidBodyProp (propRef);
    Parameter massParam = rigidBodyProp.GetParameter(L"mass");

    const double a = (double) massParam.GetValue();
}
```

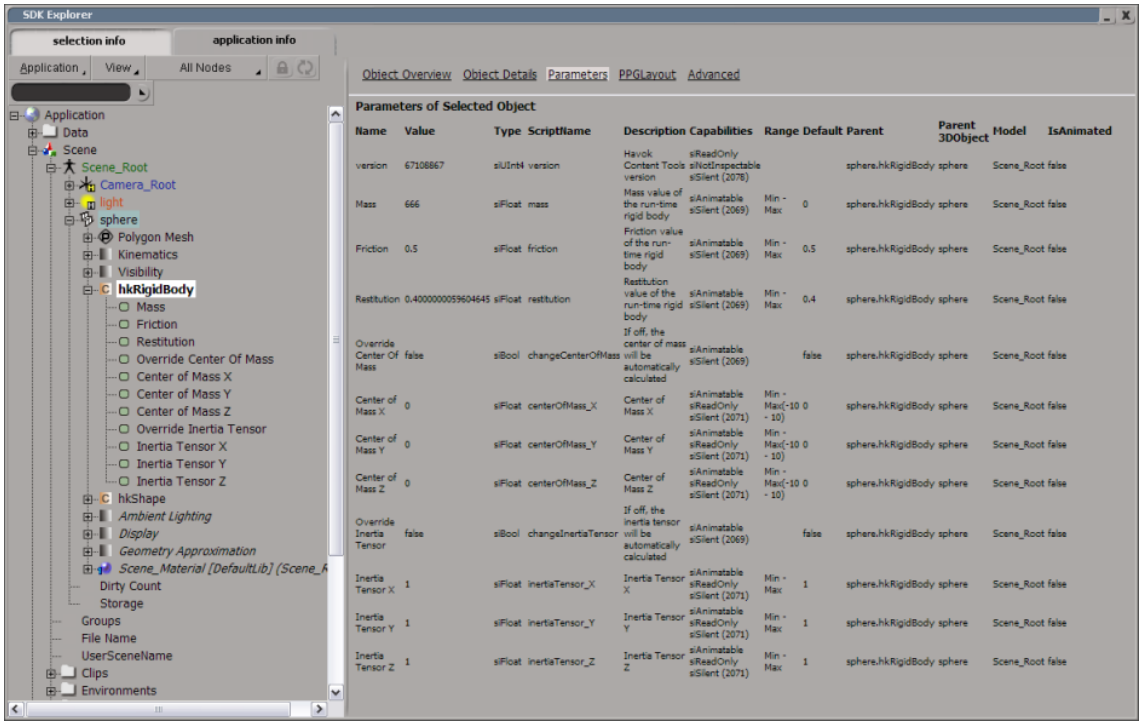
For more information about this **CRef** , **X3DObject** , **Property** , **CustomProperty** and **Parameter** classes please check the XSI SDK Guide.

As you can see the names and type names of properties and parameters are used by XSI to find and identify them in the scene. These names are presented in the XSI UI - for example you can see them in the *XSI Scene Explorer* (if you have the " **All Nodes** " option selected):

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



An detailed description of properties and their parameters can be found using *XSI's SDK Explorer* view:



In the following table gives a short description of the Havok data types (custom properties and operators) used in XSI:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Name	Description
hkExportOptions	Parented to the root of the scene, contains the Scene Exporter Options
hkpRigidBody	Rigid body parameter set
hkpShape	Shape parameter set
hkBallAndSocketConstraint	Ball-And-Socket Constraint parameter set
hkHingeConstraint	Hinge Constraint parameter set
hkRagDollConstraint	Rag Doll Constraint parameter set
hkStiffSpringConstraint	Stiff Spring Constraint parameter set
hkPrismaticConstraint	Prismatic Constraint parameter set
hkWheelConstraint	Wheel Constraint parameter set
hkPhysicsPreferences	Stored with the application, contains the Physics preferences
hkRagDollToolboxOptions	Parented to the root of the scene, contains the Rag Doll Toolbox Options
hkRagDollProxy	[Internal] Stores the association between a proxy created using the Rag Doll Toolbox and the original bone.
hkFxRigidBody	FX Rigid Body parameter set.
hkFxInstance	FX Instance parameter set.
hkFxSphere	FX Sphere parameter set.
hkFxCloth	FX Cloth parameter set.

Table 6.14: Havok Parameter Sets

Commands

Plugin functionality is exposed to XSI using commands. All Havok commands in XSI are named with the prefix "`hkCmd...`" in order to distinguish them from Havok Parameter Sets.

Commands are very easy to execute through any of the languages supported by XSI. For example, the **hkCmdExportScene** command invokes scene export, using batch mode if its "*batchMode* parameter is set to true. So, in order to export the scene in batch mode we would use :

```
// JScript example
hkCmdExportScene (true);

' VBScript example
hkCmdExportScene true

# Python example
Application.hkCmdExportScene(1)

# PerlScript example
\$Application->hkCmdExportScene(1);

// C++ Example
CValueArray args(1);
args[0] = true;
CValue retval;
const CStatus status = Application().ExecuteCommand( L"hkCmdExportScene", args, retval );
```

Export-related Commands

The following table shows the names and usage of commands related to the XSI Scene Exporter:

Command	Description
hkCmdExportScene [batchMode]	Exports the scene. If *batchMode* is set to true, it will export in batch (silent) mode, with no UI shown. By default, *batchMode* is false.
hctCmdShowExportOptions	Opens the Export Options dialog.
hctCmdGetVersionNumber	Returns a number representing the build of the Havok Content Tools. This number is built in hexadecimal, using two digits for each component of the version. For example, build 4.0.0.3 is represented as 0x04000003, which in decimal is 67108867.
hctCmdGetVersionString	Returns a string describing the build of the Havok Content Tools. For example: "*4.0.0.3 (4.0.0 Beta)*"
hctCmdShowHavokToolbar	Ensures the XSI Havok Toolbar is shown in the UI

Table 6.15: Havok XSI Scene Exporter commands

Physics-related Commands

The following table shows the names and usage of commands related to the XSI Physics Tools. Many of them are equivalent to an associated menu or toolbar option. Some of them take an optional *selection* parameter - by default, this parameter is set to the list of object currently selected in XSI.

Command	Description
hctCmdCreateRigidBodies [selection]	Creates rigid bodies from the selected 3d object(s).
hctCmdCreateCompoundRigidBody [selection]	Creates a compound rigid body from the selected 3d objects.
hctCmdCreateRigidBodyWithProxies [selection]	Creates a rigid body with proxy shapes from the selected 3d objects.
hctCmdCreateCapsule [name] [parent]	Creates a capsule object. If *name* is specified, the object is created with that name. If *parent* is specified, the object is created as a child of that parent.
hctCmdCreateBallAndSocketConstraint [selection]	Creates a ball-and-socket constraint on the selected rigid body(s).
hctCmdCreateHingeConstraint [selection]	Creates a hinge constraint on the selected rigid body(s).
hctCmdCreateRagDollConstraint [selection]	Creates a rag doll constraint on the selected rigid body(s).
hctCmdCreateStiffSpringConstraint [selection]	Creates a stiff spring constraint on the selected rigid body(s).
hctCmdCreatePrismaticConstraint [selection]	Creates a prismatic constraint on the selected rigid body(s).
hctCmdCreateWheelConstraint [selection]	Creates a wheel constraint on the selected rigid body(s).
hkCmdShowRagDollToolBox	Opens the Rag Doll Tool Box.
hctCmdCreateFxRigidBodies [selection]	Creates FX Rigid Bodies from the selected 3d objects .
hctCmdCreateFixedFxRigidBodies [selection]	Creates Fixed FX Rigid Bodies from the selected 3d objects .
hctCmdCreateFxInstances [selection]	Creates FX Instances from the selected nodes.
hctCmdShowFxToolbox	Opends the Distribute FX Spheres toolbar.
hctCmdCreateFxCloth [selection]	Creates FX Cloth objects from the selected geometries.
hctCmdShowPhysicsPreferences	Opens a property editor for the Physics Preferences.
hctCmdSelectRigidBodies [selection]	Given the current selection, or a list of objects, it focus the selection on the rigid body parameter sets associated to them.[a]
hctCmdSelectShapes [selection]	Given the current selection, or a list of objects, it focus the selection on the shape parameter sets associated to them
hctCmdSelectConstraints [selection]	Given the current selection, or a list of objects, it focus the selection on the constraint parameter sets associated to them
hctCmdCreateConstraint [selection] [constraintType]	Used internally, it creates a constraint of the given type.
hctCmdCreateEmptyRigidBodies [selection]	[Advanced] Adds a rigid body parameter set to the selected objects, without adding any shape parameter sets.
hctCmdCreateShapes [selection]	[Advanced] Adds a shape parameter set to all the selected objects, without adding any rigid body parameter sets.
hctCmdDistributeFxSpheres distributionType [selection]	Used internally, it distributes FX Spheres on the selected object using the given distributionType algorithm. Valid distribution types are : "`box`", "`sphere`", "`capsule`" and "`convex`".
hctCmdRemoveHavokProperties [selection]	Removes Havok-specific properties on the given objects.
hctCmdManipulateConstraint	For internal use only.
hctCmdManipulateRigidBody	For internal use only.
hkCmdRagDollToolBoxEvent	For internal use only.

Table 6.16: Havok XSI Physics Tools commands

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

[a] These commands can, for example, be associated to a keyboard shortcut. That way, it is possible to quickly access and edit specific properties of a scene object. The Havok Content Tools for XSI install a Keyboard Mapping (named "*Havok XSI Key Map*") which associates **CTRL+ALT + R / C / S** to the Select Rigid Bodies / Constraints / Shapes commands.

Other Havok Objects

Although Custom Parameter Sets and Commands make up most of the Havok functionality in XSI, there are also some other Havok objects registered with XSI. Check the XSI SDK Guide for details on how to use these types of objects.

Name	Type	Description
hctOpCapsuleObject	Operator	The Havok Capsule Operator, used to generate a polygonal capsule mesh.
hctCustomDisplayRagDollToolbox	Custom Display	The Rag Doll Toolbox.
hctDisplayCallbackPhysics	Display Callback	[Internal]. Used to display all Havok information in the viewports.
Havok_RBs	Filter	A selection filter that will select only those objects in the scene with Rigid Body Parameter Sets.
hctMenuExport	Menu	The **Havok** menu.

Table 6.17: Other Havok objects in XSI

6.7.7 Writing your own Filters

6.7.7.1 Overview

In some cases, you may want to extend the Havok Content Tools export pipeline by adding you own filters. Some examples of situations where you may want to do this are:

- You want to transform some mesh data to your own format, different from the formats available in the Havok SDK. You would then create a filter that replaces Havok meshes with your own custom meshes
- You want to manipulate the information on some objects - for example, append a user-specified prefix to all bones. You can then create a new filter takes a prefix from the user, and then looks for bones in the scene, modifying their name accordingly.
- You want to add extra information to the file based on the scene - for example: based on the user properties of some nodes in the scene, create specific game objects that you can load at run-time.
- You have implemented a specific compression algorithm that works particularly well for your game and you want the exported animations to be compressed using this algorithm.

Basically, if you have algorithm that you'd rather apply to the assets before export instead of doing so after import, then writing a filter is a sensible solution. The following sections will guide you on how to do so.

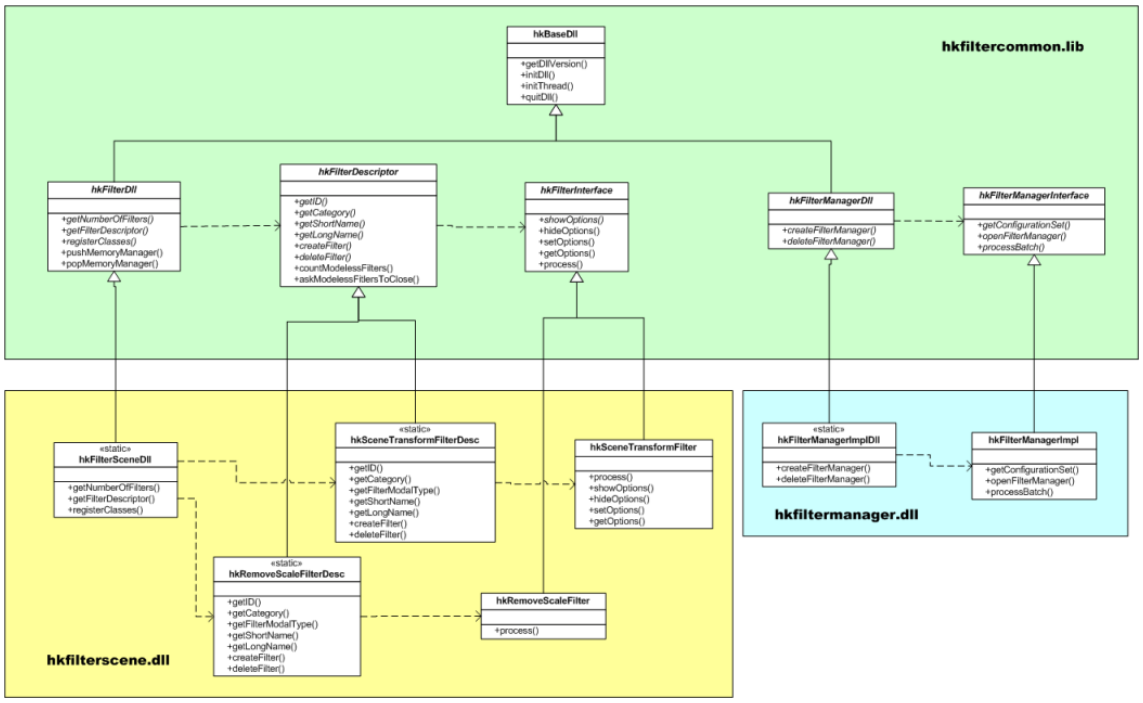


Figure 6.15: Some of the main classes involved in filter processing

The figure above shows an example of some classes involved in filter processing. The base classes are all defined in the library `hkfiltercommon` (shown in green in the picture). This library is fully documented in the Reference Manual. These classes define interfaces and default implementations for DLL interfaces, filter managers and filters. Both filters (in yellow) and the filter manager implementation (in blue) use these classes.

In the following sections we'll be discussing these classes and interfaces.

6.7.7.2 Handling DLLs : hctBaseDll, hctFilterDll and hctFilterManagerDll

The filter toolchain is based on DLLs : the Filter Manager DLL is responsible for loading all of the the other DLLs (the filters) and executing them when the user requests so. The Filter Manager DLL itself gets loaded by the actual application (3ds Max, Maya and XSI scene exporters, or the standalone filter manager).

Having the filter manager and filters as DLLs has several advantages: adding a new filter to the toolchain is as easy as building a new DLL and dropping it into the correct folder (more about this next). Also, the same toolchain is used by all modelers/applications without the need to build specialized tools for each modeler/application.

If you have worked with DLLs before you'll be aware that there are some issues that need to handled carefully:

- Versioning : It is important to ensure that the DLL that is loaded matches the version expected by the loading process - otherwise unexpected errors may happen, as memory corruption and crashes.
- Memory management : As each DLL has its own copy static data, any class or algorithm that relies

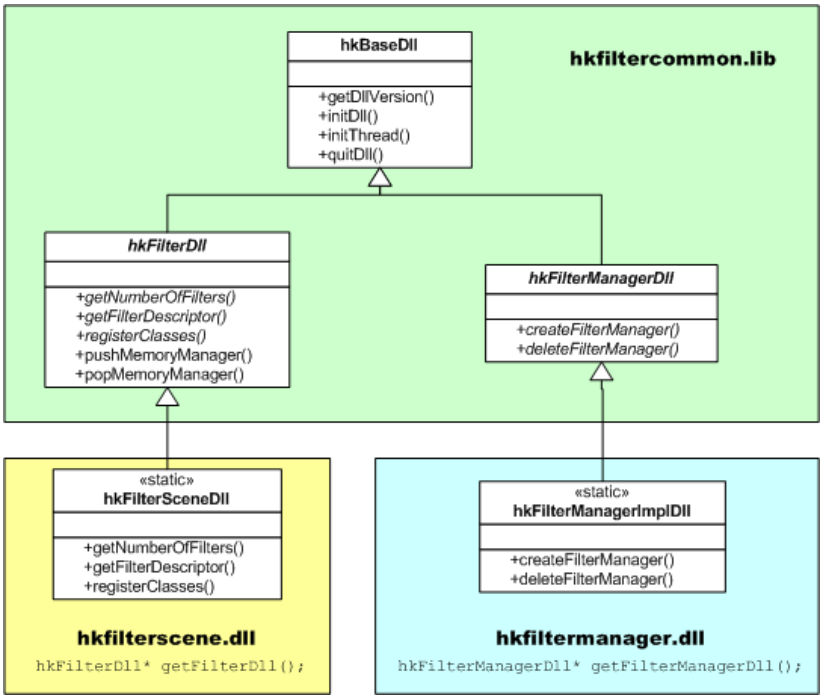
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

on static data or singletons need to ensure all DLLs are synchronized and use the same singletons. Examples of static data are error handlers, memory managers, etc..

- Interface : Extending/modifying functionality in DLLs, if not done carefully, may convert the DLL interface into a bit of a mess, with a mixture of many, old and new, exported functions growing over time.

In order to simplify and reuse the logic to handle the above, the `hctBaseDll` class is used. This class defines functionality required for versioning and memory management, and groups all required methods into a single interface (so rather than exporting many functions, DLLs export a single function returning an instance of this class).

Two types of DLLs are used by the Havok filter processing : the filter manager DLL and multiple filter DLLs. Associated with each, there are the classes `hctFilterDll` and `hctFilterManagerDll`:



Each of these two classes extend the interface in `hctBaseDll` with methods required by DLLs implementing filters and the filter manager:

- `hctFilterDll` : Adds interface methods to access a virtual array of filter descriptors (`hctFilterDescriptor` objects - we'll talk about these later). This class also must register all classes created by any of the filters it implements. Finally, it defines (and provides default implementations) methods to handle temporary replacement of memory manager.
- `hctFilterManagerDll` : Adds interface methods in order to create and destroy a filter manager (`hkFilterManager` object).

Each DLL contains a single, static instance of a subclass of either `hctFilterDll` (for filter DLLs) or `hctFilterManagerDll` (for the filter manager DLL). This instance must be returned by the only exported function in the DLL, **getFilterDll()** or **getFilterManagerDll()** (depending on the type of

DLL). In the example shown, these static objects are instances of the classes `hctFilterSceneDll` (for `hkfilterscene.dll`) and `hctFilterManagerImplDll` (for h`kfiltermanager.dll`).

Placing DLLs

Since the filter manager DLL is usually loaded by the scene exporters (or the standalone filter manager) and the filter DLLs are automatically loaded by the filter manager itself, you don't usually need to worry about loading DLLs - you just need to place your filter DLL in the right place.

The filter manager and the filter DLLs are kept together in a particular file structure so they can be found by the scene exporters. This file structure is automatically created in the specified installation directory by the Havok Content Tools installer. However, if you are developing your own filters, extending the filter manager, or would simply like separate installations of the filter manager for each of your modelers then you'll need to know the following:

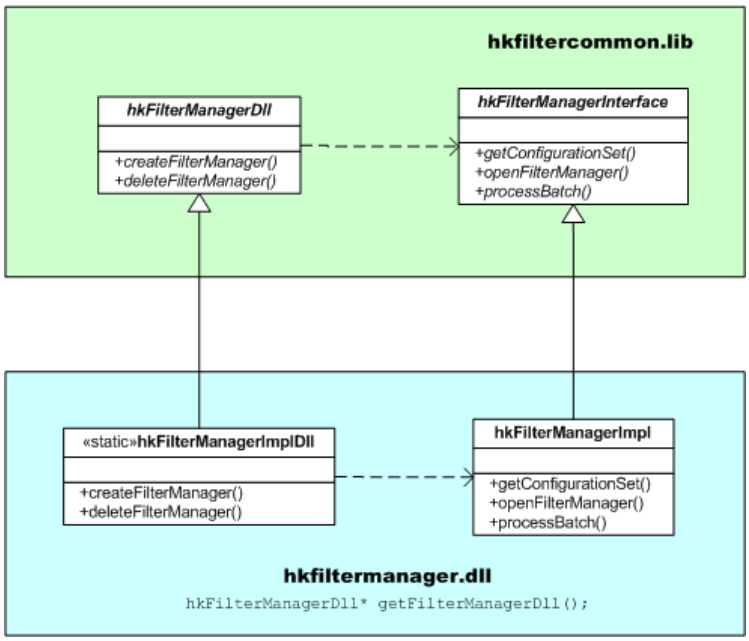
- The filter manager looks for filter DLLs in a "`.\filters`" subdirectory, relative to itself. For example, if the filter manager DLL is at "`c:\havokfilters\hkfiltermanager.dll`", it will look for filter DLLs in "`c:\havokfilters\filters\`". Any new or modified filter DLLs should be kept here.
- The installer also creates the registry entry " **HKEY_CURRENT_USER\Software\Havok\hkFilters** ", which stores (among other things) the location(s) of the filter manager. You'll need to modify the values here if you have multiple instances of the filter manager.

This is how the various applications look for the filter manager DLL:

Standalone Filter Manager	The standalone filter manager expects to find the filter manager DLL in the same directory as the the `hkStandaloneFilterManager.exe` executable. If it does not find the filter manager there, it checks the **FilterPath** registry value, which points to the filter manager's default location.
3ds Max Scene Exporter	The 3ds Max scene exporter first looks for a "`havok`" folder inside the installation directory (for example, "`c:\3dsmax8\havok`"). If the filter manager dll is not found, it then checks for the **MaxPath** registry value containing the filter manager location. If it does not find the filter manager there, **FilterPath** , the default location, is used. If you have a separate installation of the filter manager for 3ds Max simply enter its location in the **MaxPath** registry value and the 3ds Max scene exporter will find it.
XSI Scene Exporter	The XSI scene exporter first looks for a "`havok`" folder inside the installation directory (for example, "`c:\softimage\xsi5.0\havok`"). If the filter manager dll is not found, it then checks for an **XsiPath** registry value containing the filter manager location. If it does not find the filter manager there, **FilterPath** , the default location, is used. If you have a separate installation of the filter manager for XSI simply enter its location in the **MaxPath** registry value and the XSI scene exporter will find it.
Maya Scene Exporter	The Maya scene exporter first looks for a "havok" folder inside the location of the havok module (for example, if the maya exporter is located in "`c:\mayaTools\plugins\`", then it will look for the filter manager dll in a folder canmed "`c:\mayaTools\havok\`"). If not found, the Maya exporter will look in the folder specified by the **MayaPath** registry value. If it does not find the filter manager there, **FilterPath** , the default location, is used. If you have a separate installation of the filter manager for Maya simply enter its location in the **MayaPath** registry value and the Maya exporter will find it.

Finally, if all the above steps fail for any application, the application looks for a `HAVOK_EXPORT_FILTERS_ROOT` environment variable, which you can set up to point to the filter manager. Note that this variable isn't created by the installer, and is only used if the registry values are incorrect or missing.

6.7.7.3 The Filter Manager : hctFilterManagerInterface



Both filters and exporters can access the filter manager through an `hctFilterManagerInterface` object.

There is only one implementation of this class (`hctFilterManagerImpl`), but all external access is done through the interface class. We use an interface class not because we expect multiple implementations of it, but rather because we want to decouple the methods from their implementation, so methods are implemented in the DLL rather than statically linked to the caller (filter or exporter).

You can check the Reference Manual for details on the many methods defined in this interface.

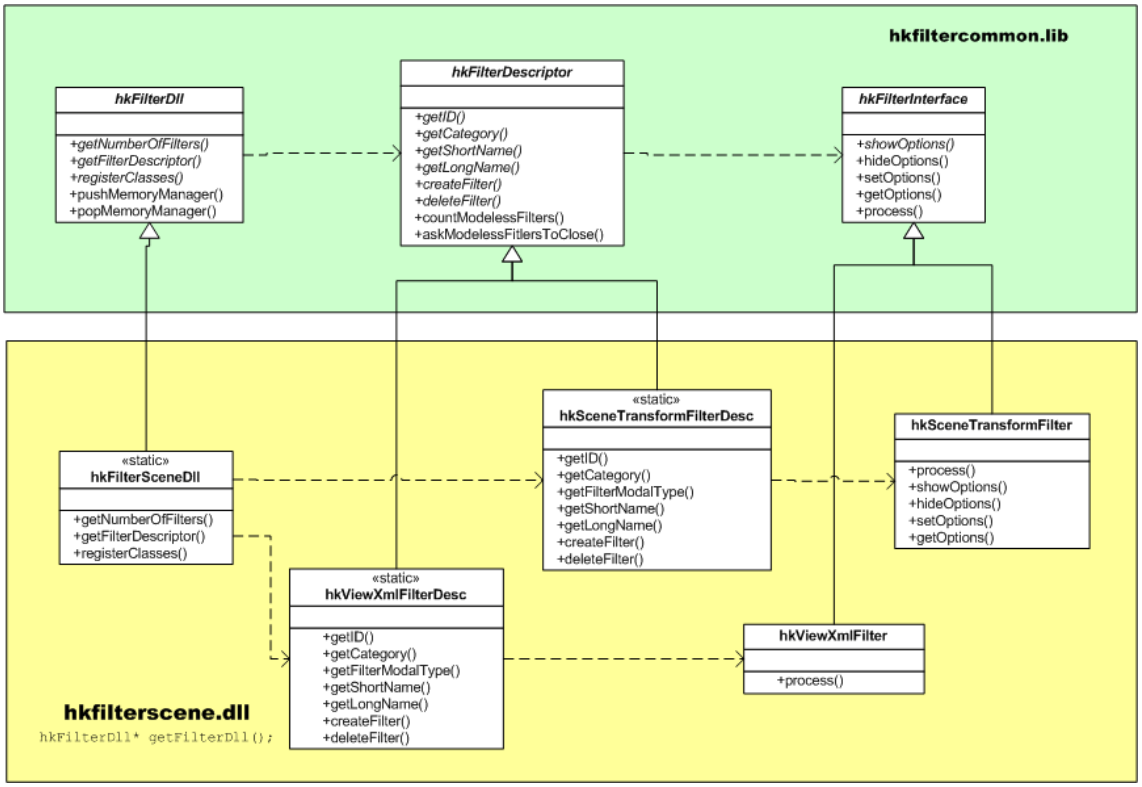
The `hkfiltermanager(.dll)` project provides the implementation class.

The hksceneexport library

In order to isolate some of the common, reusable functionality shared by the scene exporters and the standalone filter manager (loading the filters, triggering the processing, etc..), this functionality has been put in a library, `hksceneexport.lib`. This library is also fully documented in the Reference Manual.

6.7.7.4 Filters : hctFilterDescriptor, hctFilterInterface

Multiple filters are described and implemented in a single DLL. In this example we show two of the many filters implemented in `hkfilterscene`:



hctFilterDll

As we saw earlier, the `hctFilterDll` object exposed by each filter DLL provides access to a virtual array of `hctFilterDescriptor` objects through the **hctFilterDll::getNumberOfFilters()** and **hctFilterDll::getFilterDescriptor()** interfaces. Filters also register all classes they create through the **hctFilterDll::registerClasses()** interface.

hctFilterDescriptor

This interface class defines methods to create, destroy and describe (name, category, id..) filters. These descriptors are used by the filter manager in order to present the list of available filters to the user without the need of creating them. Therefore, a single static instance is used for each filter available, returned through the **hctFilterDll::getFilterDescriptor()** method.

hctFilterInterface

This is the interface class to the actual filters that process the content. Each filter type, therefore, must override this class and implement methods to:

- Retrieve / Set option data (if any)
- Open / Close its options UI (if any)
- Process the contents

The **hctFilterInterface::process()** method is therefore the most important method for each filter.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.7.7.5 Modeless Filters : hctModelessFilterDll, hctModelessFilterDescriptor, hctModelessFilter

While most filters modify the contents they process (and therefore the next filter must wait for the filter to finish processing before continuing), some filters (like the View XML Filter and the Preview Scene Filter) make a copy of it and allow processing to continue while the filter remains active. These are what we call *modeless filters*.

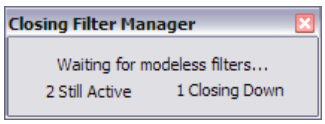
Creating modeless filters has some extra implications:

- In order to keep the filter active while processing continues, a new thread has to be created for the new active filter. This thread (and its associated memory, etc) needs to be properly created, managed and destroyed.
- A private copy (clone) of the contents has to be created for the filter to use (since the "main" contents will be modified by other filters).
- The memory manager of a DLL containing modeless filters cannot be replaced before and after processing since the memory may still be used by an active filter. So memory initialization and cleanup can only be done on DLL load and unload (for modal filters it is done before and after each call to `hctFilterInterface::process()`).

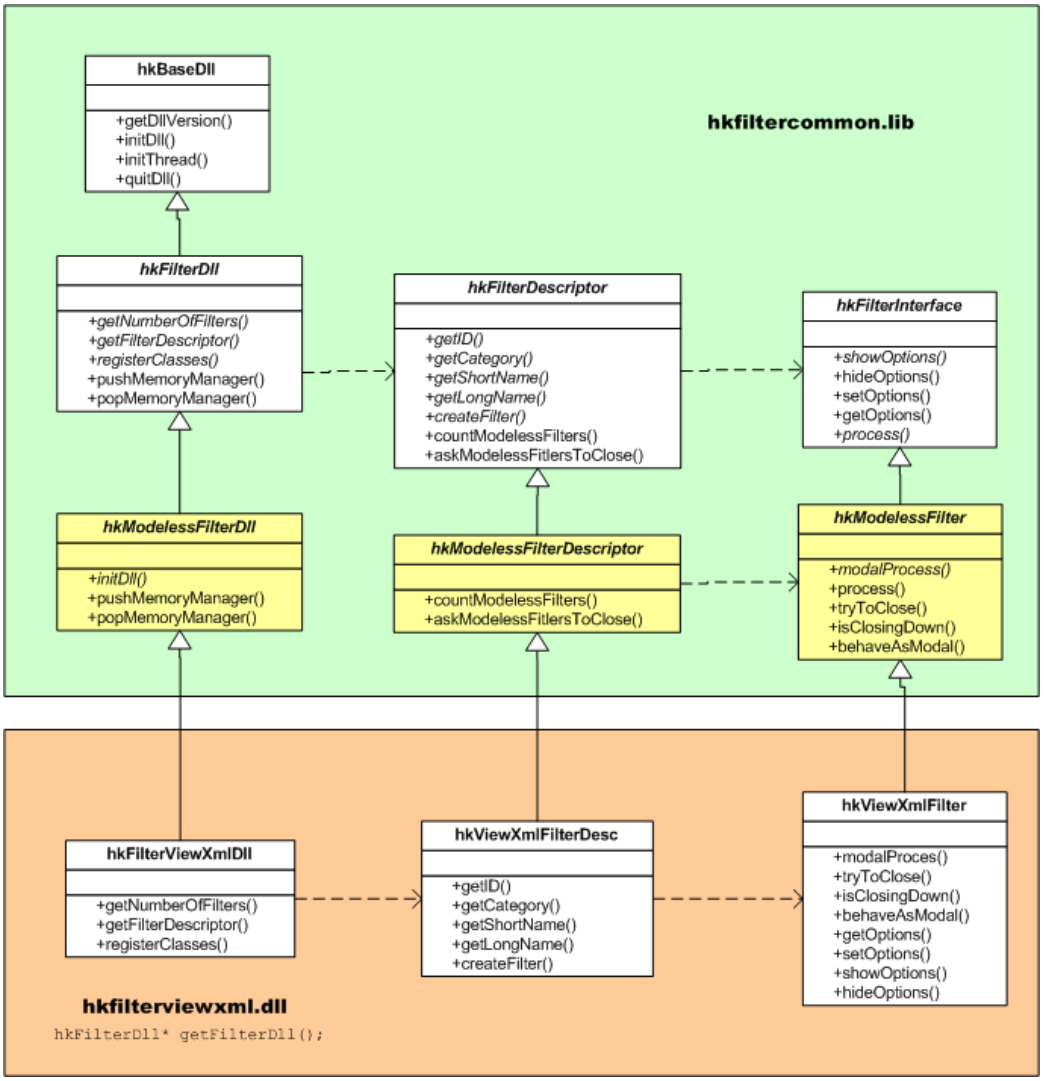
  Important:

  This means that m*odeless and modal filters cannot be mixed in the same DLL* (since they manipulate their memory manager in different ways).

- We need to keep track of active instances of each modeless filter - and since each one runs in a different thread, we need to ensure we avoid race conditions and deadlocks.
- When the user closes the filter manager, active instances of modeless filters should "try to close" and report their progress on closing down (they may not be able to close, or they may require some time to do so).



In order to simplify the development of modeless filters, and since the functionality required to implement the above is independant of the actual processing done by each filter, specialized versions of `hctFilterDll`, `hctFilterDescriptor` and `hctFilterInterface` are provided (highlighted in yellow in the diagram below):



- `hctModelessFilterDll`

  It overrides the methods `pushMemoryManager()` and `popMemoryManager()` from `hctFilterDll` with empty implementation. For modeless filters it is no longer possible to replace the memory manager on the fly (another thread may be using it). It also overrides `initDll()` from `hctBaseDll` since, due again to memory managent, error messages are handled differently (they are not reported to the filter manager window).

  If your DLL implements modeless filters (remember, though, that you cannot mix modeless and modal filters in the same dll), you should inherit from this class rather than `hctFilterDll`.

- `hctModelessFilterDescriptor`

  This class provides implementations for keeping track of active instances of modeless filters, as well as asking them to close and to report their state.

- `hctModelessFilter`

  This class overrides `process()` from `hctFilterInterface` - it implements it by making a copy of the contents to process, creating a thread and then invoking `processModal()`. Filters inheriting from `hctModelessFilter` must therefore implement the `processModal()` method instead of `process()`.

  Code inside the `processModal()` method can act as if the filter was modal (since it operates on a private copy of the data and it runs in a separate thread).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

By using these classes, implementing a modeless filter is a very easy task, almost identical to implementing a modal filter, with the only differences being:

- Modeless filters cannot be mixed with modal filters in the same DLL
- Implementations must inherit from `hctModelessFilterDll`, `hctModelessFilterDescriptor` and `hctModelessFilter`
- The filter must implement `hctModelessFilter::processModal()` instead of `hctFilterInterface::process()`

6.7.7.6 Looking for examples

If you are a Havok Physics or Havok Complete customer (excluding Havok XS), you may find helpful to take a look at the source for some of the filters provided by the Havok SDK. You can use some of the simpler filters, like the **SceneTransform** filter (in the `hkfilterscene` project) or the **PruneTypes** filter (in the `hkfilterasset` project) as a template for your own filter. For modeless filters, check out the **ViewXml** filter (in the `hkfilterviewxml` project). If you are interested in invoking the filter manager DLL from your own application, you can take a look at the standalone filter manager (`tools/standalonefiltermanager` project) as an example.

Check out the "Extending the Toolchain" tutorial for an step-by-step example on how to extend the tools by exporting extra data from 3ds Max, Maya and XSI and creating a custom filter.

6.7.8 Customizing your Installation Method

Running the Havok Content Tools installer on a given PC creates the files required for each of the Filter Pipeline, 3ds Max tools, Maya tools and XSI tools, and also creates several registry settings. These files are set up in such a way as to allow the Havok Content Tools to operate normally on that PC. If you would prefer not to use our installer, for example to install to a network or to provide a zip file instead, it is possible to do so. This section of the documention describes the steps taken by our installer, and describes the basics of how to customize your installation method.

6.7.8.1 Steps taken by the Installer

The Havok Content Tools installer installs a required set of core files, plus optional sets of files for whichever modeler versions are present.

Core Files

The core files are those which provide the Havok Filter Pipeline, specifically the filter manager DLL, the individual filter DLLs, and the attribute description XML files. The standalone filter manager, a set of batch processing scripts and some help files are also installed alongside the core files.

These files are installed into any standalone path ("`%ProgramFiles%\Havok\HavokContentTools`" by default), forming a particular folder structure.

Several registry keys and values are then set up under "`[HKEY_CURRENT_USER\Software\Havok]`". The most important of these is the "`hkFilters\FilterPath`" string value, containing the path to the installed

filter manager DLL. This value is read by the scene exporters whenever they need to locate and load the filter manager.

3ds Max Files

The optional 3ds Max files provide the tools used by 3ds Max to create and export Havok content. These consist of two plugins (one for export, one for physics tools) and a set of associated Maxscripts, icons, and tutorial scenes.

These files are installed directly into a 3ds Max application directory (chosen during installation), adding themselves to various subfolders (`\plugins`, `\UI`, `\scripts`). On startup, 3ds Max will automatically find these files and load the plugins.

Maya Files

The optional Maya files provide the tools used by Maya to create and export Havok content. These consist of two plugins (one for export, one for physics tools) and a set of associated MEL scripts, icons, and tutorial scenes.

These files are installed into any standalone path (chosen during installation), forming a particular folder structure - that of a Maya "*Module*".

The installer then instructs Maya to use this module by creating the following 'pointer' file: "`%CommonFiles%\Alias Shared\modules\maya\<x.x>\HavokContentTools.txt`". This content of this simple text file is of the following form:

```
+ HavokContentTools <x.x> <module path>
```

On startup, Maya should locate this file and add the contents of the corrosponding module to it's list of available plugins. The plugins may then need to be manually loaded from within Maya's plugin manager.

XSI Files

The optional XSI files provide the tools used by XSI to create and export Havok content. These consist of two plugins (one for export, one for physics tools) and a set of associated SPDL files, layouts, toolbars, presets, and tutorial scenes.

These files are installed into any standalone path (chosen during installation), forming a particular folder structure - that of an XSI "*Workgroup*".

On startup, XSI may automatically load the workgroup or it may have to be manually loaded: if it was installed to XSI's *factory* of *user* path then it will automatically loaded; otherwise it should be loaded by connecting to it from within XSI's plugin manager.

6.7.8.2 Creating a Custom Installation

Extracting the Files

The first step is to extract the required set of files. Since we do not provide a zipped or other form of the installation files, the best way to do this is to create a temporary installation using the installer:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

1. Create a new folder to store the extracted files.
2. Run the Havok Content Tools installer.

   Choose only the modeler(s) and version(s) that you require.

   Install each of the components (core files, modeler versions) to separate subfolders of our main folder, instead of to the default locations.

   Close the installer.
3. Duplicate the main folder.
4. Open `regedit`, locate and export the installed "`[HKEY_CURRENT_USER\Software\Havok]`" key to a file (if required).
5. Uninstall the Havok Content Tools.

This will leave you with a folder containing the extracted files for each required component, and a registry file containing the installed registry values (if required). Now is a good time to examine the contents of each folder to familiarize yourself with their contents.

Distributing the Files

The next step is to package the files into whichever form you want and distribute them. There are some important things to consider, whichever way you distribute the files:

- It is not normally necessary to install all of the registry values - they will be created when required.
- The exporters must be able to find the filter manager DLL. The path to the DLL can be stored in either
  - The "`[HKEY_CURRENT_USER\Software\Havok\hkFilters\FilterPath]`" registry value, or
  - An environment variable named "`HAVOK_EXPORT_FILTERS_ROOT`"

  See 'Placing DLLs' earlier in this documentation for more on where and how the DLLs should be placed.
- For Maya, the plugins will only appear in the plugin manager window (and therefore only be usable) if the module 'pointer' file is set up correctly. To view a list of paths in which this module file may be placed, run the following MEL command within Maya:

  `getenv MAYA_MODULE_PATH`
- If you are not using Havok Complete, you should select your Havok product in the Product menu of the Filter Managers main menu once you load it up for the first time (Havok Complete is the default value), so that incompatible filters are displayed as such. The installer usually prompts the user for this information and sets it in the registry, so you could also distribute the saved registry settings to achieve the same result.

6.7.9 Tutorial : Extending the Toolchain

While the most common parameters of rigid bodies, shape types and constraints are already exposed and processed by the Havok Content Tools (and more will be supported in the future), you may find the need to extend this coverage into particular functionality of the Havok SDK or your custom objects. This tutorial provides an step-by-step guide on how to extend both the modelling tools and filter processing in order to handle a new type of object. You may want to become familiar with the concepts and classes discussed in this chapter before working on this tutorial, as most of them will be used during it.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.7.9.1 Introduction

For this tutorial, we will consider the example of setting up and creating a "`phantom`" run-time object. A phantom is a shape in the scene which receives events whenever a rigid bodies enter or exit it's volume. We will also create some configurable actions which will be applied to any rigid bodies which move within the phantom volume.

We would like to be able to use this added functionality with any supported modeler (3ds Max / Maya / XSI). Due to the design of the toolchain, this is a relatively simple process. We can separate the required work into two distinct areas:

1. We need a way to flag objects in a scene as phantoms, and to specify any number of additonal attributes which determine the run-time behaviour of the phantom and/or the actions.

   This will be done within the modeler, through the use of 'custom attributes'. As described earlier in this chapter, each of the supported modelers provides a way in which custom data may be attached to any object in the scene (3ds Max, Maya, XSI). This data will automatically be detected and exported by the Havok Scene Exporters. Very little work is therefore required within the modeler, and this work can easily be automated/scripted.

2. We need a way to interpret the exported phantom information, and convert it to a run-time phantom with associatied actions.

   This will be done within the filter pipeline, by creating a custom filter. This filter will look for any objects which contain our custom phantom data, and create appropriate run-time objects in the processed scene. Most of the work will therefore be performed by the filter, which is modeler-independant.

Before getting started, lets consider the structure of an `hkpPhantom` object - observe that an `hkpShape` is required. Therefore when our filter wants to create a phantom based on a scene object, the first thing it will need to do is create an `hkpShape` representing the object's mesh. One solution would be to have the filter analyze the mesh and determine a good shape based on that. However, the Havok Content Tools already provide the ability to create `hkpShape` objects as part of rigid bodies, and provide a filter to interpret that information (the 'Create Rigid Bodies' filter). These tools already contain all of logic required to create `hkpShape` objects (including compound shapes), so it is much more beneficial to re-use this existing logic. In fact, we can just wrap a rigid body's shape with an `hkpPhantomCallbackShape` object, allowing us to also re-use any rigid body information (eg. transform).

Therefore the solution which we will demonstrate in this tutorial is as follows:

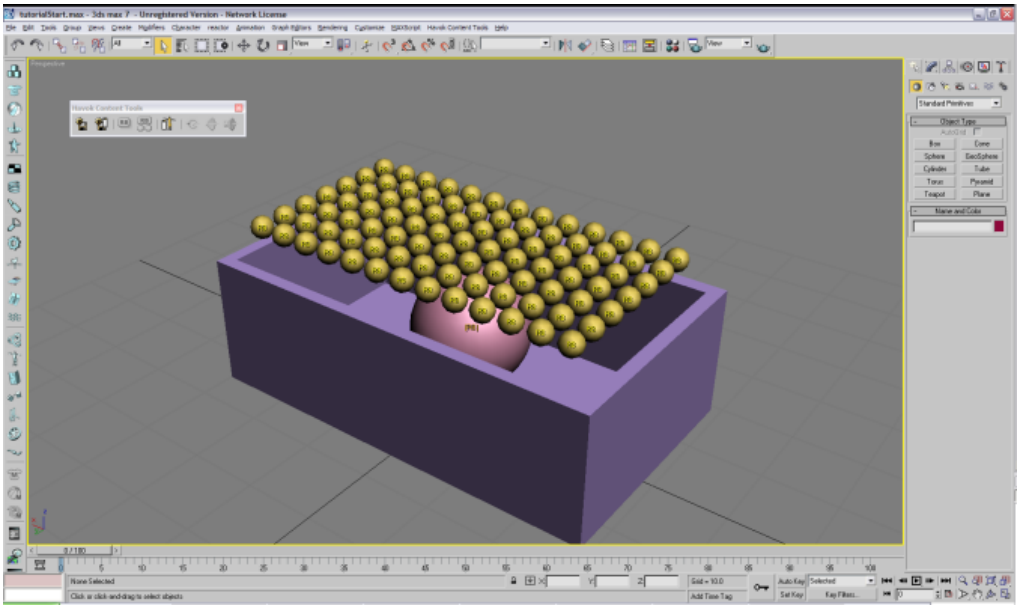
1. When we want to create a phantom from an object in the modeler, we will create a rigid body for it as per usual (which includes shape information). We will then add an *extra* attribute group to the object specifiying phantom information. This attribute group will be called '*hkPhantomAction*'. We will also specify some additional processing rules for these attributes through the use of 'attribute description' XML files.

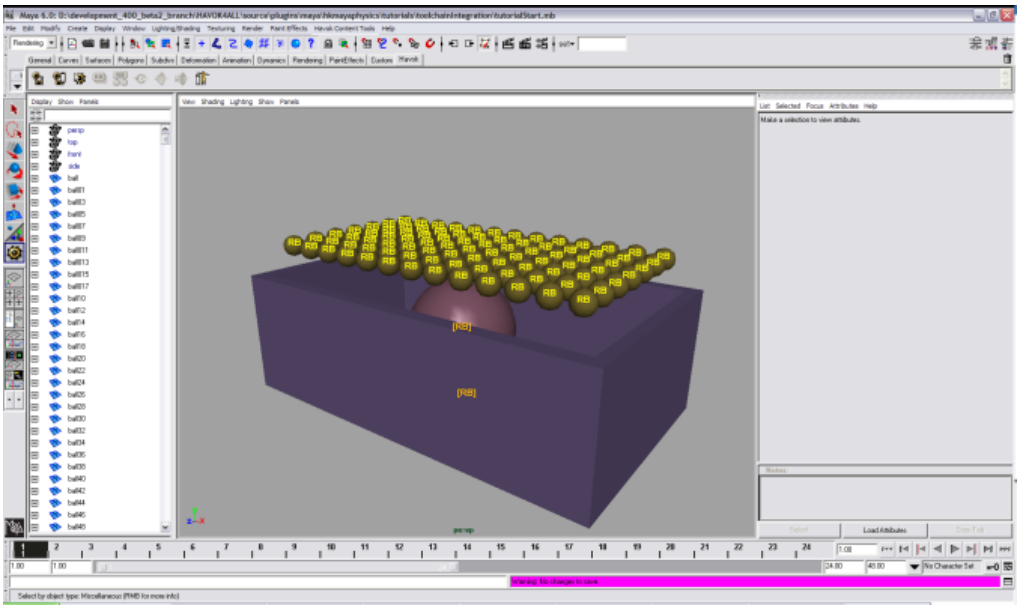
2. After exporting the data, we will add a 'Create Rigid Bodies' filter to the filter configuration - this will create a `hkpRigidBody` object, which includes a `hkpShape` object. Following that, we will add our custom filter - this will search for our specific attribute group in the scene and wrap each corresponding rigid body's `hkpShape` object with an `hkpPhantomCallbackShape` object. The values of the attributes which we specified in the modeler will be used to customise the behaviour of the actions which will act on any rigid bodies entering the phantom volume. We will call this filter "*convertToPhantomAction*".

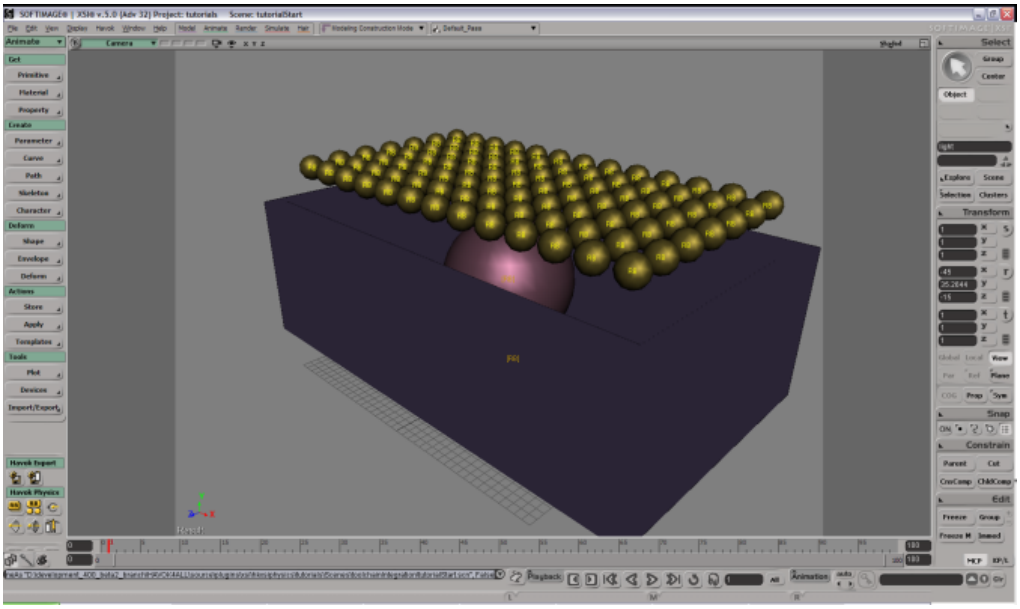
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.7.9.2 Starting up

We will use a very simple scene to test our toolchain extensions. Load the tutorial file `toolchainIntegration/tutorialStart` (`.max` / `.mb` / `.scn`) :







This very simple scene contains an open box and multiple spheres. The large sphere in the middle (*phantom*) is what we want to use as our phantom object - we want any spheres which enter that volume to be affected by our magic action.

6.7.9.3 Creating and Exporting the Attribute Group

The first step in the process is to create a rigid body within a modeler and add our custom attribute group to it, so it can be picked up later on by our custom filter. We would like to add our atributes as a group by the name of "*hkPhantomAction*" - this could be any name, but it should be descriptive of the group contents. We would like to add the following attributes to this group:[10]

action	A STRING value, specifying a type of action which should be applied to the phantom once it is created by the filter (from a selection of "wind", "attract", "deflect").
direction	A VECTOR value, specifiying some direction which may be used by the action.
strength	A FLOAT value, specifiying some power value which may be used by the action.

In the following subsection we will see how to add these attributes to our object using each specific modeller. We will then also see how to solve the problem of handling differences between modellers regarding attribute types and ranges by using an attribute repository.

Adding the Attributes

Our phantom object (named "phantom") already has the rigid body and shape attributes (added by hkpRigidBody and hkpShape modifiers/nodes/custom parameter sets) - we now want to also add our new attributes. There are multiple ways to specify extra attributes to be exported from each modeller. In this tutorial, we will use the simplest one, which uses the modeller's UI to add them.

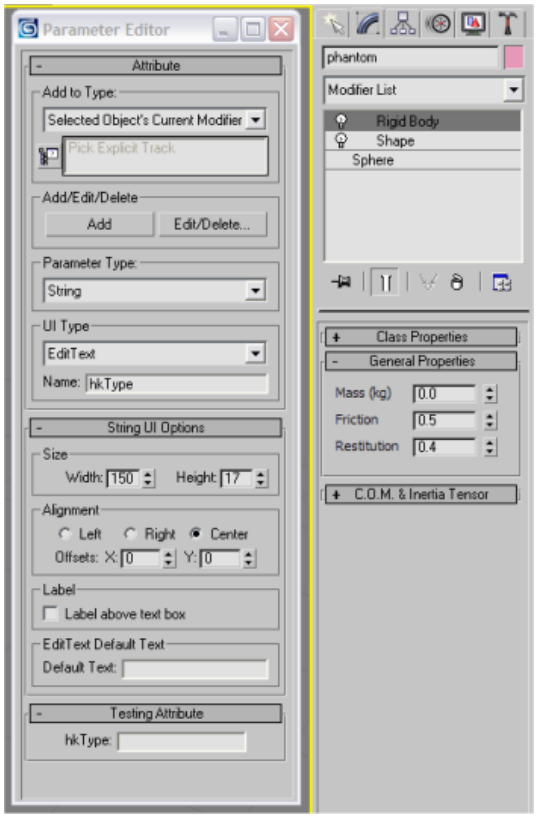
[10] These attributes were chosen for this tutorial so that we can demonstrate how to create and process various attribute types. Feel free to use whatever you come up with.

Note:

This has the advantage of not requiring any knowledge of scripting or plugin development, so it suits this tutorial. It is also possible, however, to add attributes by many other means, including scripting and creating custom nodes/modifiers/property sets using C++ or other supported languages. Attributes created this way have the advantage of being easier to customize, to reuse and to extend - but you'll need some knowledge of the modeler's scripting language and/or sdk in order to do so.

3ds Max

Make sure the rigid body modifier of our *phantom* object (the big sphere) is selected, then click on *Animation > Parameter Editor...* from the main menu. This opens a dialog which allows us to define and add custom attributes:



We want our set of attributes to be exported as a named group. The name of the default custom attribute container in 3ds Max in '*main*' (this can be changed using scripting), and as such any attributes we add using the UI would be exported under a group called 'main' by default. Since we would like to use a specific group name, begin by adding the following 'special' atrribute:

- A *String* named "`hkType`".

  When the exporter finds an attribute with this name (*hkType*), it uses its value to replace the name of the exported attribute group - in this case give it a value of "`hkPhantomAction`".

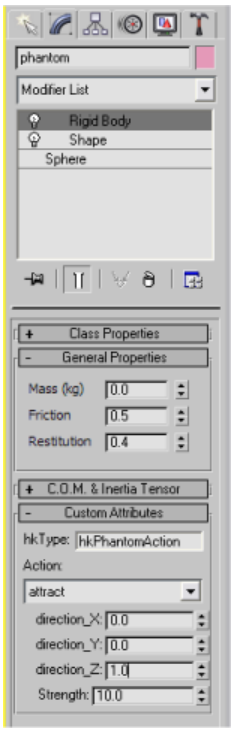
Next add the 'real' attributes in turn:

- An *Array* named "`action`", with the following elements: "`wind`", "`attract`", "`deflect`".

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- Three individual *WorldUnits* named "`direction_X`", "`direction_Y`", and "`direction_Z`".
- A single *Float* named "`strength`".

Which should leave you with the following UI:

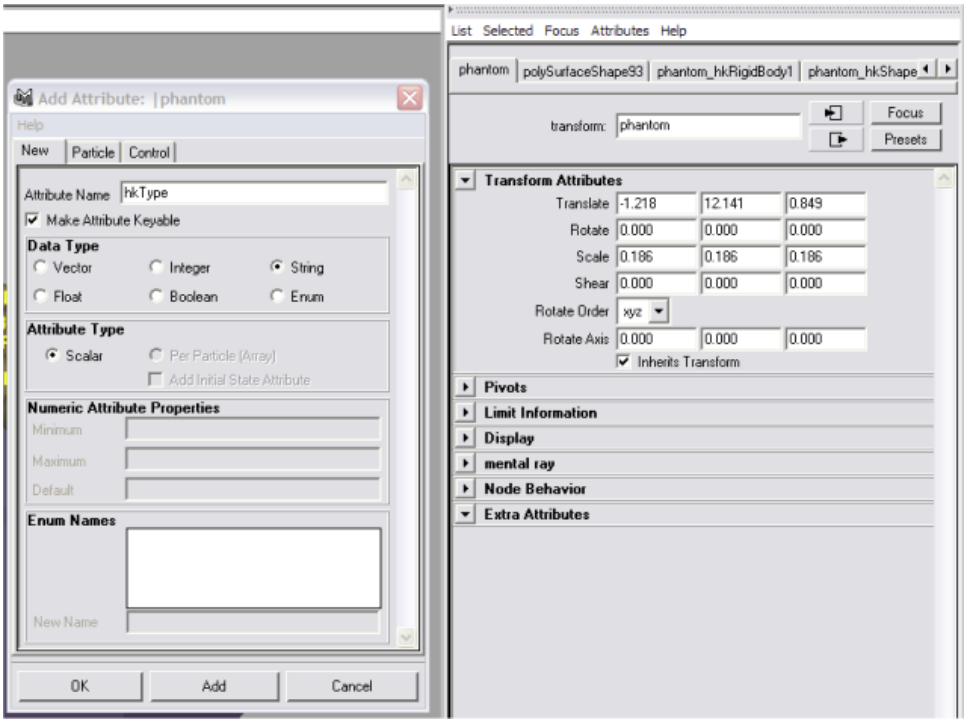


Set the following values:

- **hkType** : *hkPhantomAction*
- **Action** : *attract*
- **Direction** : *0,0,1*
- **Strength** : *10*

Maya

Make sure the *phantom* node (that is, the transform node named "*phantom*") is selected, then click on *Attributes* > *Add Attribute...* from the attribute editor panel. This opens a dialog which allows us to define and add new attributes:



We want our attributes to be exported as a named attribute group. By default in Maya, any attributes we add will be in the same group as all of the existing attributes of a node. Therefore we require some way to specifiy a new group. This is done by first adding the following 'special' atrribute:

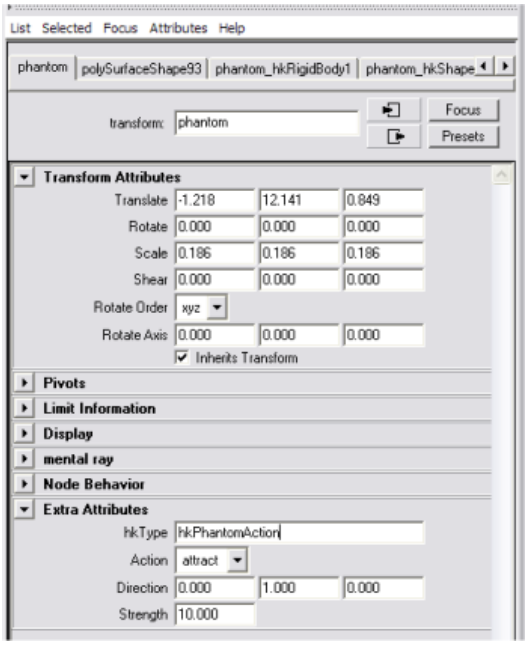
- A *String* attribute named "`hkType`".

  When the Maya scene exporter finds a string attribute with this name, it starts a new attribute group named with the value of this attribute. In this case give it a value of "`hkPhantomAction`".

Next add the 'real' attributes in turn:

- An *Enum* named "`action`", with the following elements: "`wind`", "`attract`", "`deflect`".
- A *Vector* named "`direction`".
- A *Float* named "`strength`".

Which should leave you with the following UI (you may need to expand the ' **Extra Attributes** ' rollout):



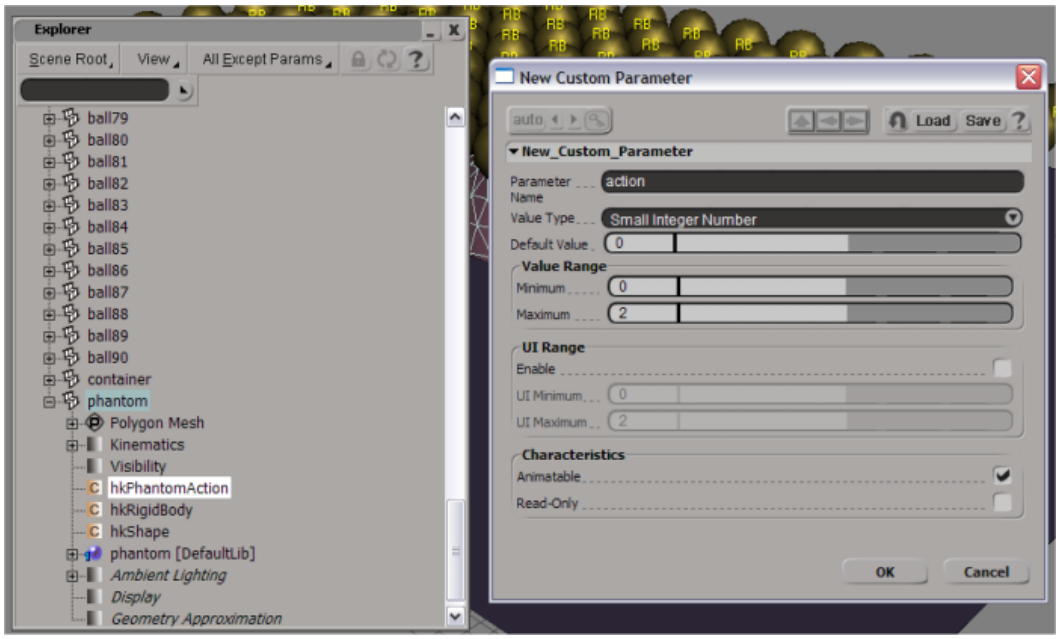
Set the following values for the attributes:

- **hkType** : *hkPhantomAction*
- **Action** : *attract*
- **Direction** : *0,1,0*
- **Strength** : *10*

XSI

Making sure our phantom object is selected, click on *Animate* > *Parameter* > *New Custom Parameter Set* from the main menu. Enter "`hkPhantomAction`" as the name of the new parameter set. Now, with the new parameter set selected, click on *Animate* > *Parameter* > *New Custom Parameter Set* from the main menu (or press SHIFT+P). This opens a dialog which allows us to define and add a new custom parameter/attribute:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Add the following attributes in turn:

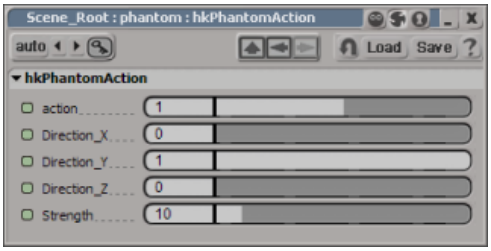
- An *Small Integer Number* named "`action`", with a range of zero to two (representing the three action types: "wind", "attract", "deflect").

  Note:

  There is no way to define enum attributes in XSI - we will use integers instead. Integers can be represented as strings in property editors by writing custom PPG code, however for this tutorial we just 'know' that the integers values 0,1,2 represent "wind", "attract" and "deflect".

- Three individual *Floating Point Numbers* named "`direction_X`", "`direction_Y`", and "`direction_Z`".
- A *Floating Point Number* named "`strength`".

This should leave you with the following PPG for the custom parameter set:



Set the following values for the attributes:

- **Action** : *1* (attract)
- **Direction** : *0,1,0*

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- **Strength** : *10*

Pre-processing Attributes

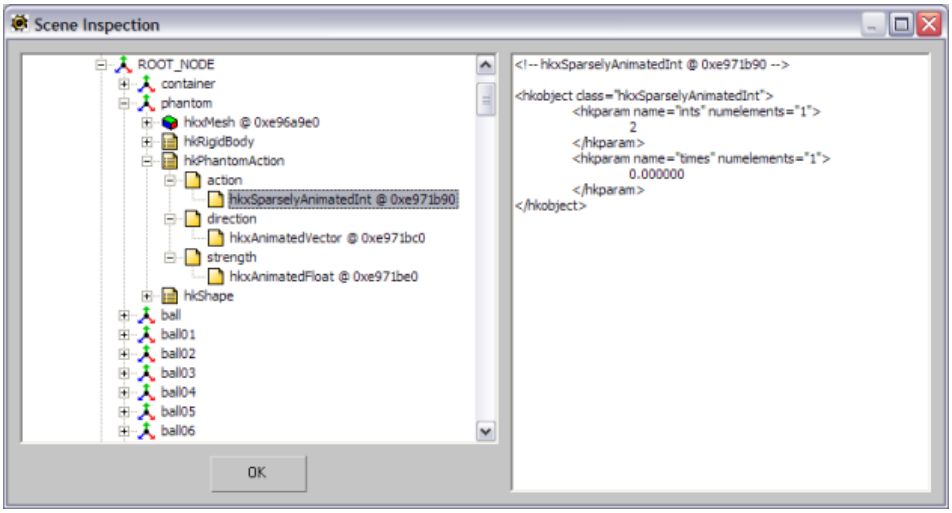
Each modeller has different means and capabilities to store and represent attributes. For example, some modellers (XSI and to some extent 3ds Max) do not have support for enumerated values, some others can't represent vector attributes with a single attribute, etc..

Since we want to build our filters in a modeller-independant way, the scene exporters provide the ability to pre-process the attributes before they are passed to the filters. Some processing is always done (for example merging X,Y,Z float attributes into a single vector attribute) while some other is driven by the user through XML files (an attribute repository): for example, integer -> string conversions or marking a float as "scalable" (world units) values.

We will need to drive some preprocessing of our attributes from each modeller in order to match our original *hkPhantomAction* attribute group definition (a string, a vector and a float). The XML files for each modeller are automatically installed for you - for the purpose of this tutorial, you may want to temporarily remove them to see their effect. The files can be found in the `max` / `maya` / `xsi` folders inside the main installation folder (there the filter manager DLL is placed) : for example, `c:\program files\havok\Havok Content Tools\attributeProcessin\max\hctMaxTutorialAttributeData.xml`.

3ds Max

Without adding anything to the attribute repository (you can test this by temporarily removing the `attributeProcessing/max/hctMaxTutorialAttributeData.xml` file) this would be the output associated to our *phantom* node:



Notice how the X, Y and Z values of *direction* have been merged into a single vector value - this is always done automatically. Also, if you examine the "*direction*" attribute, you see that it has the flag **HINT_TRANSFORM_AND_SCALE** . This is because we created the X, Y and Z attributes in 3ds Max as "world units" and therefore the exporter assumes that the value should be fully transformed by any scene transformations.

Notice however how the *action* parameter is being exported as an integer (with value = 2 for the 2nd value, "*attract*"): 3ds Max doesn't keep track of the value-name association so the exporter cannot see

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

the name, it only sees the integer value.

Here is where the attribute repository comes into play - we can tell the exporter to apply changes to attributes by using a max-specific XML file (the `attributeProcessing/max/hctMaxTutorialAttributeData.xml` we temporarily removed - place it back now). Let's see the contents of this file:

```
<?xml version="1.0" encoding="utf-8"?>
<hkpackfile classversion="2">

    <hksection name="__data__">

        <hkobject class="hctAttributeDescriptionDatabase" name="#0000">
            <hkparam name="groupDescriptions" numelements="1">

                <hkobject>
                    <hkparam name="name">hkPhantomAction</hkparam>
                    <hkparam name="attributeDescriptions" numelements="2">
                        <hkobject>
                            <hkparam name="name">action</hkparam>
                            <hkparam name="forcedType">FORCE_STRING</hkparam>
                            <hkparam name="enum">#0001</hkparam>
                            <hkparam name="hint">HINT_NONE</hkparam>
                        </hkobject>
                        <hkobject>
                            <hkparam name="name">direction</hkparam>
                            <hkparam name="hint">HINT_TRANSFORM</hkparam>
                        </hkobject>
                    </hkparam>
                </hkobject>

            </hkparam>
        </hkobject>

        <hkobject class="hkClassEnum" name="#0001">
            <hkparam name="name">hkShapeTypeEnum</hkparam>
            <hkparam name="items" numelements="3">
                <hkobject>
                    <hkparam name="value">1</hkparam>
                    <hkparam name="name">wind</hkparam>
                </hkobject>
                <hkobject>
                    <hkparam name="value">2</hkparam>
                    <hkparam name="name">attract</hkparam>
                </hkobject>
                <hkobject>
                    <hkparam name="value">3</hkparam>
                    <hkparam name="name">deflect</hkparam>
                </hkobject>
            </hkparam>
        </hkobject>

    </hksection>

</hkpackfile>
```

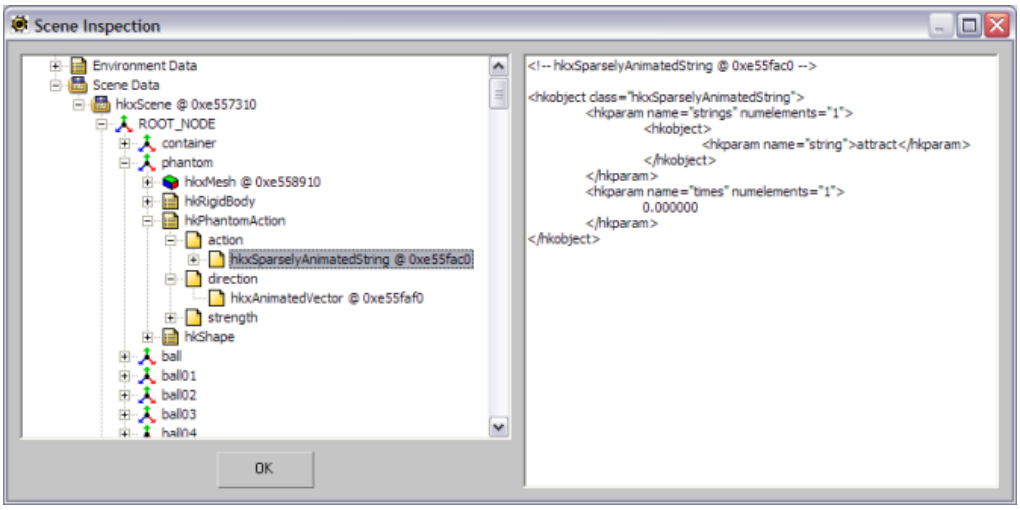
The file contains an attribute description database (`hctAttributeDescriptionDatabase`) object, with a single attribute group description (`hctAttributeGroupDescription`) for *hkPhantomAction*. We only have two attribute descriptions (`hctAttributeDescription`).

- One for our *action* attribute, specifying that we want to force its type to be a string (

FORCE_STRING). We also provide, in order to do the conversion, a hkClassEnum object that maps values with names (1,2,3 for *wind*, *attract* and *deflect*).

- One for the *direction* attribute - 3ds Max assumed it should be transformed and scale (as it could represent a position) - but since we know it represents a direction, we don't want it to scale with the scene. Therefore we set the flag **HINT_TRANSFORM** .

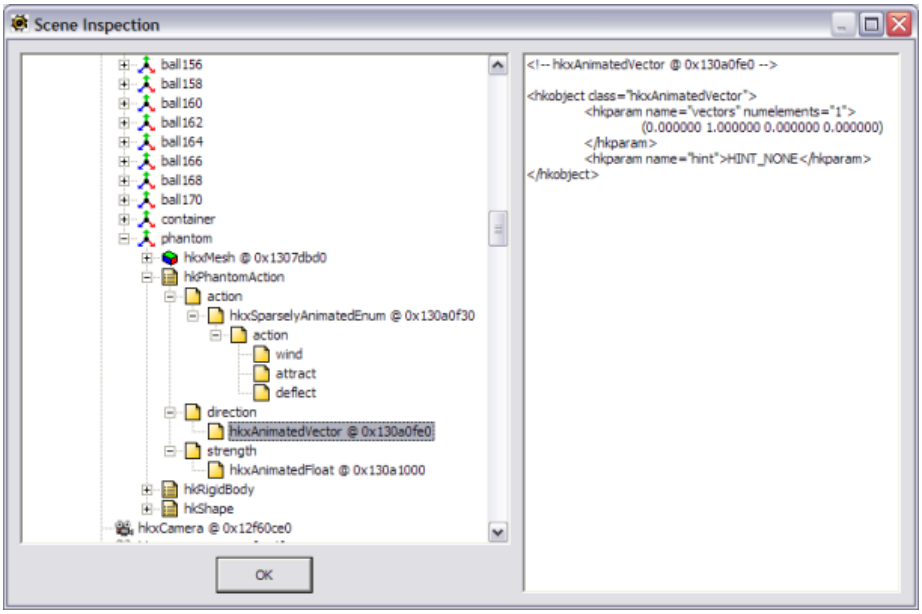
With this file in place, the exported attributes now look like this:



The attributes now match what our filter expects - so we are done!.

Maya

Without adding anything to the attribute repository (you can test this by temporarily removing the `attributeProcessing/maya/hctMayaTutorialAttributeData.xml` file) this would be the output associated to our *phantom* node:



Notice that our action attribute is exported as an enum - Maya has built-in support for enumerated attributes. Our filter expects a string - it is very easy to convert from an enum to an string - you just take convert the value to its associated name (notice how the value<->name association is already exported from Maya).

The direction value is exported as a vector. That's also what we want. However, Maya doesn't know what the vector represents - it could be a color, a position, a direction... We'd like to specify that it represents a direction, that is, that it should be transformed (rotated), but not scaled, by a scene transformation.

Here is where the attribute repository comes into play - we can tell the exporter to apply changes to attributes by using a maya-specific XML file (the `attributeProcessing/maya/hctMayaTutorialAttributeData.xml` we temporarily removed - place it back now). Let's see the contents of this file:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

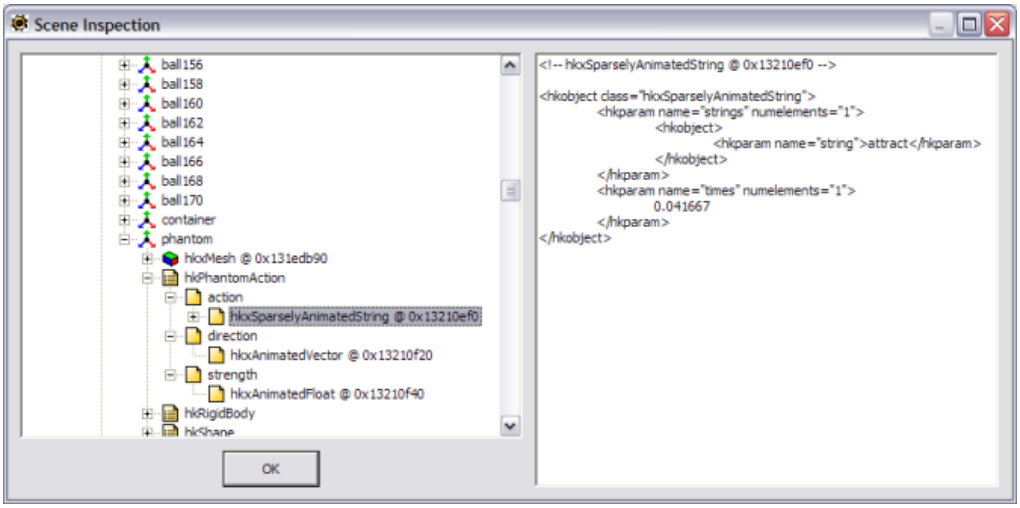
```
<?xml version="1.0" encoding="utf-8"?>
<hkpackfile classversion="2">

    <hksection name="__data__">

        <hkobject class="hctAttributeDescriptionDatabase" name="#0000">
            <hkparam name="groupDescriptions" numelements="1">

                <hkobject>
                    <hkparam name="name">hkPhantomAction</hkparam>
                    <hkparam name="attributeDescriptions" numelements="2">
                        <hkobject>
                            <hkparam name="name">action</hkparam>
                            <hkparam name="forcedType">FORCE_STRING</hkparam>
                        </hkobject>
                        <hkobject>
                            <hkparam name="name">direction</hkparam>
                            <hkparam name="hint">HINT_TRANSFORM</hkparam>
                        </hkobject>
                    </hkparam>
                </hkobject>

            </hkparam>
        </hkobject>

    </hksection>

</hkpackfile>
```

The file contains an attribute description database (`hctAttributeDescriptionDatabase`) object, with a single attribute group description (`hctAttributeGroupDescription`) for *hkPhantomAction*. We only have two attribute descriptions (`hctAttributeDescription`).

- For the action attribute, we specify that we want to convert it to an string attribute using **FORCE_STRING** . Notice that the original enum attribute can easily be converted to an string attribute - we don't need to specify any value-to-name conversion since it was already contained in the enum attribute.
- For the direction attribute, we add the flag **HINT_TRANSFORM** to note that this vector should rotated, but not scaled, by any scene transformations applied.

With this file in place, the exported attributes now look like this:

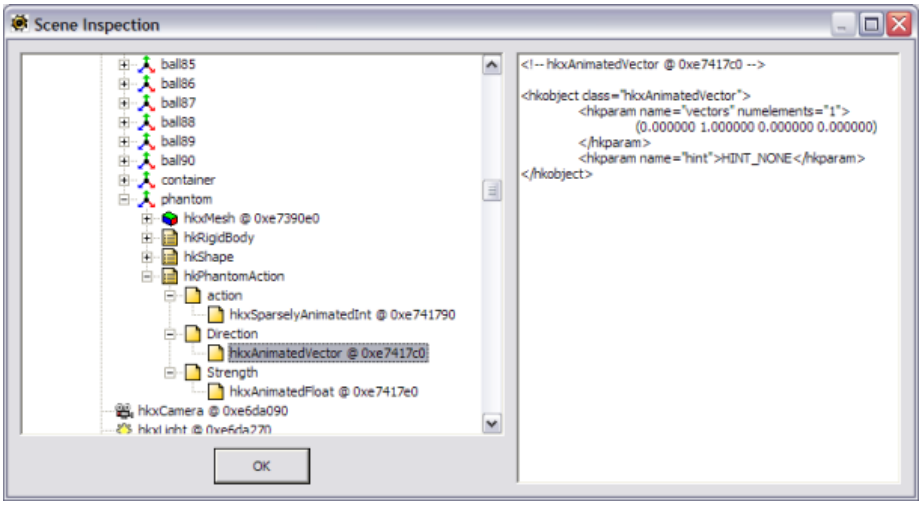
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



The attributes now match what our filter expects.

XSI

Without adding anything to the attribute repository (you can test this by temporarily removing the `xsi/hctXsiTutorialAttributeData.xml` file) this would be the output associated to our *phantom* node:



Notice that our action attribute is exported as an integer (with value = 1 for the 2nd value, "attract"), since XSI doesn't support enumerated attributes

Notice also how our three X, Y and Z attributes for the direction were automatically merged into a single vector attribute. That's good. However, XSI doesn't know what the vector represents - it could be a color, a position, a direction... We'd like to specify that it represents a direction, that is, that it should be transformed (rotated), but not scaled, by a scene transformation.

Here is where the attribute repository comes into play - we can tell the exporter to apply changes to attributes by using an XSI-specific XML file (the `attributeProcessing/xsi/hctXsiTutorialAttributeData.xml` we temporarily removed - place it back now). Let's see the contents of this file:

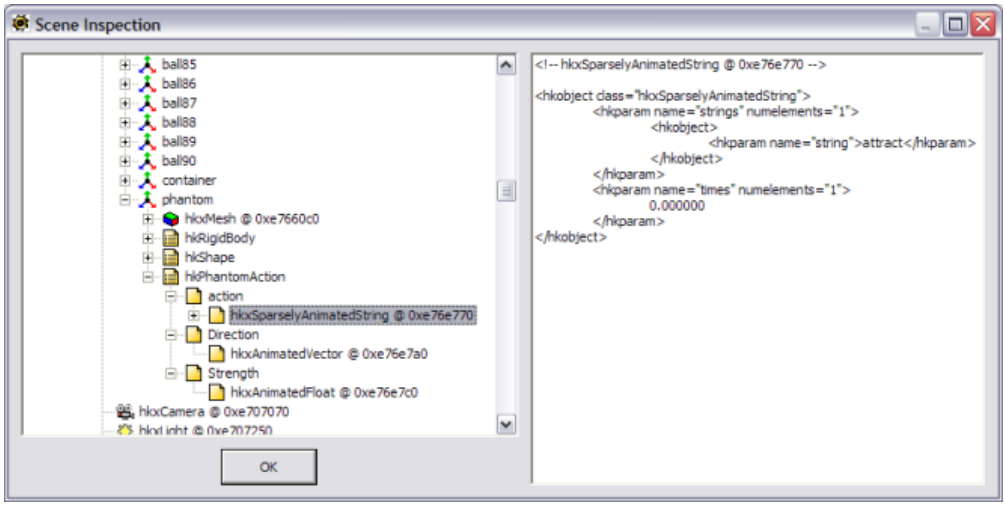
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

```
<?xml version="1.0" encoding="utf-8"?>
<hkpackfile classversion="2">

    <hksection name="__data__">

        <hkobject class="hctAttributeDescriptionDatabase" name="#0000">
            <hkparam name="groupDescriptions" numelements="1">

                <hkobject>
                    <hkparam name="name">hkPhantomAction</hkparam>
                    <hkparam name="attributeDescriptions" numelements="2">
                        <hkobject>
                            <hkparam name="name">action</hkparam>
                            <hkparam name="forcedType">FORCE_STRING</hkparam>
                            <hkparam name="enum">#0001</hkparam>
                            <hkparam name="hint">HINT_NONE</hkparam>
                        </hkobject>
                        <hkobject>
                            <hkparam name="name">direction</hkparam>
                            <hkparam name="hint">HINT_TRANSFORM</hkparam>
                        </hkobject>
                    </hkparam>
                </hkobject>

            </hkparam>
        </hkobject>

        <hkobject class="hkClassEnum" name="#0001">
            <hkparam name="name">hkShapeTypeEnum</hkparam>
            <hkparam name="items" numelements="3">
                <hkobject>
                    <hkparam name="value">0</hkparam>
                    <hkparam name="name">wind</hkparam>
                </hkobject>
                <hkobject>
                    <hkparam name="value">1</hkparam>
                    <hkparam name="name">attract</hkparam>
                </hkobject>
                <hkobject>
                    <hkparam name="value">2</hkparam>
                    <hkparam name="name">deflect</hkparam>
                </hkobject>
            </hkparam>
        </hkobject>

    </hksection>

</hkpackfile>
```

The file contains an attribute description database (`hctAttributeDescriptionDatabase`) object, with a single attribute group description (`hctAttributeGroupDescription`) for *hkPhantomAction*. We only have two attribute descriptions (`hctAttributeDescription`).

- One for our *action* attribute, specifying that we want to force its type to be a string (**FORCE_STRING**). We also provide, in order to do the conversion, a `hkClassEnum` object that maps values with names (0,1,2 for *wind*, *attract* and *deflect*).
- For the direction attribute, we add the flag **HINT_TRANSFORM** to note that this vector should rotated, but not scaled, by any scene transformations applied.

With this file in place, the exported attributes now look like this:



The attributes now match what our filter expects.

6.7.9.4 Writing the Filter

Now that we have the data exported from the modeller, we need to create a filter to interpret that data and create our phantom object. We have called the DLL containing this filter `hkfiltertutorial` and the filter itself "*Convert to Phantom Action*". Depending on your type of product license, you may find the source of this `hkfiltertutorial` project in your `include\plugins\common\hkfiltertutorial` folder.

Setting up the project

A single DLL can contain multiple filters, but in our case we will create a DLL for just a single filter (alternatively, we could add this filter to an existing filter DLL, for example `hkfilterphysics`). We will use the convention of placing the filter-independant files in the root folder of `hkfiltertutorial`, while we'll place the files specific to the "*Convert To Phantom Action*" filter inside the `hkfiltertutorial/converttophantomaction` folder.

If you start the Visual Studio project from scratch, create a DLL project with Multithreaded DLL configurations (that's the run-time library used by the exporters and filters). In particular, you may want to create the following configurations:

- Hybrid Multithreaded DLL : Uses release multithreaded DLL run-time libraries, but it compiles using full debug information and no optimizations. This configuration is useful for debugging.
- Release Multithreaded DLL : Uses release multithreaded DLL run-time libraries, with full optimizations and no debug information. This configuration is useful to build the final DLL to ship to users.

Notice that both configurations use Release Multithreaded DLL run-time libraries - this is because that's the run-time library used by the application (max, maya, XSI) and mismatching them would cause instability.

Add the following two files to the project:

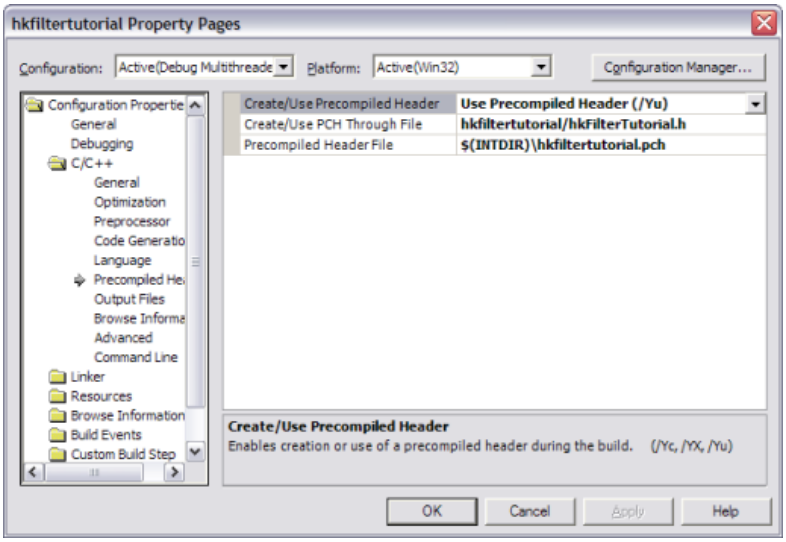
6.7.9.5 Example: hctFilterTutorial.h

```
#ifndef HAVOK_FILTER_TUTORIAL__H
#define HAVOK_FILTER_TUTORIAL__H

#include <ContentTools/Common/Filters/Common/hctFilterCommon.h>

#include <ContentTools/Common/Filters/FilterTutorial/resource.h>

#endif // HAVOK_FILTER_TUTORIAL__H
```

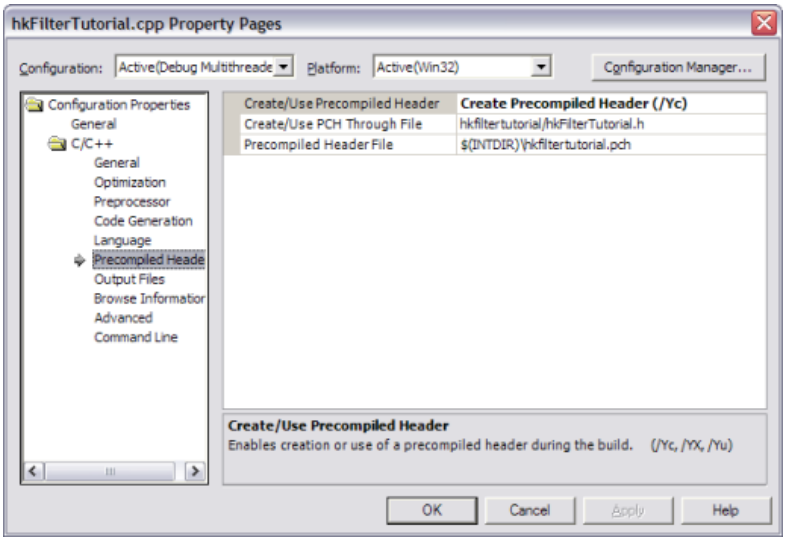
6.7.9.6 Example: hctFilterTutorial.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>

// Keycode
#include <Common/Base/keycode.cxx>

// DLL main
HINSTANCE hInstance;
static BOOL CommonControlsInitialized = FALSE;

BOOL WINAPI DllMain(HINSTANCE hinstDLL, ULONG fdwReason, LPVOID lpvReserved)
{
    switch ( fdwReason )
    {
        case DLL_PROCESS_ATTACH:
        {
            hInstance = hinstDLL;               // Hang on to this DLL's instance handle.

            if (!CommonControlsInitialized)
            {

                CommonControlsInitialized = TRUE;
                InitCommonControls();           // Initialize Win32 controls
            }
            break;
        }

        case DLL_PROCESS_DETACH:
        {
            if (CommonControlsInitialized)
            {
                CommonControlsInitialized = FALSE;
            }
        }

        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
            break;
    }
    return TRUE;
}
```

We will use the `hctFilterTutorial.h` header filer as our precompiled header, since it contains very common includes and will speed up our compile times:



Then set `hkfiltertutorial.cpp` as the one to create that precompiled header:



The file `hkfiltertutorial.cpp` contains the `DllMain()` function, which here is only used to initialize the common controls library and to store a handle to the DLL instance.

Defining our custom phantom shape and actions

Before we continue with more filter and dll specific classes and files, let's first define our custom phantom shape. Since it is specific to our filter, we'll place the files inside the `converttophantomaction` folder:

6.7.9.7 Example: convertophantomaction/MyPhantomShape.h

```
#ifndef INC_MYPHANTOM_SHAPE_H
#define INC_MYPHANTOM_SHAPE_H

#include <Physics/Collide/Shape/Misc/PhantomCallback/hkpPhantomCallbackShape.h>

/// meta information
extern const class hkClass MyPhantomShapeClass;

// Our phantom call back shape will be notified when an object enters or exits the shape
// We use those event to dynamically add / remove an action.
class MyPhantomShape : public hkpPhantomCallbackShape
{
    //+vtable(1)
    public:

        HK_DECLARE_REFLECTION();

        MyPhantomShape();
        MyPhantomShape( hkFinishLoadedObjectFlag flag ) : hkpPhantomCallbackShape(flag) {}

        // hkpPhantom interface implementation
        /*virtual*/ void phantomEnterEvent( const hkpCollidable* phantomColl, const hkpCollidable*
          otherColl, const hkpCollisionInput& env );
        /*virtual*/ void phantomLeaveEvent( const hkpCollidable* phantomColl, const hkpCollidable*
          otherColl );

    public:

        enum ActionType
        {
            ACTION_WIND,
            ACTION_ATTRACT,
            ACTION_DEFLECT
        };

        hkEnum<ActionType, hkInt8>    m_actionType;
        hkVector4                        m_direction;
        hkReal                            m_strength;
};


#endif //INC_MYPHANTOM_SHAPE_H
```

Check the Havok Physics SDK for details on hkpPhantomCallbackShape. We basically want to override the two even handlers (enter and leave) so an action is applied to the object when it enters our shape and removed when it leaves. The implementation can be found in MyPhantomShape.cpp

6.7.9.8 Example: convertophantomaction/MyPhantomShape.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>
#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/MyPhantomShape.h>

#include <Physics/Dynamics/Action/hkpBinaryAction.h>
#include <Physics/Dynamics/Entity/hkpRigidBody.h>


//
// Actions
//

// Apply a constant force to the rigid body
// (Use a binary action so that it maintains a ref to the phantom)
class MyWindAction : public hkpBinaryAction
{
public:

    MyWindAction( hkpRigidBody* phantom, hkpRigidBody* rb, const hkVector4& force ) : hkpBinaryAction(
      phantom,rb), m_force(force) {}
    MyWindAction( hkFinishLoadedObjectFlag flag ) : hkpBinaryAction(flag) {}

    /*virtual*/ void applyAction( const hkStepInfo& stepInfo )
    {
        hkpRigidBody* rb = (hkpRigidBody*)getEntityB();
        rb->applyForce( stepInfo.m_deltaTime, m_force );
    }

    /*virtual*/ MyWindAction* clone( const hkArray<hkpEntity*>& newEntities, const hkArray<hkpPhantom*>&
      newPhantoms ) const
    {
        return HK_NULL;
    }

    hkVector4 m_force;
};

// Apply a 'point' force to the rigid body
class MyAttractAction : public hkpBinaryAction
{
public:

    MyAttractAction( hkpRigidBody* phantom, hkpRigidBody* rb, float strength ) : hkpBinaryAction(phantom,
      rb), m_strength(strength) {}
    MyAttractAction( hkFinishLoadedObjectFlag flag ) : hkpBinaryAction(flag) {}

    /*virtual*/ void applyAction( const hkStepInfo& stepInfo )
    {
        hkVector4 force;
        force.setSub4( getEntityA()->getMotion()->getPosition(), getEntityB()->getMotion()->getPosition()
          );
        float distance = force.length3();
        force.mul4( m_strength / ( ( distance * distance ) + HK_REAL_EPSILON ) );

        hkpRigidBody* rb = (hkpRigidBody*)getEntityB();
        rb->applyForce( stepInfo.m_deltaTime, force );
    }

    /*virtual*/ MyAttractAction* clone( const hkArray<hkpEntity*>& newEntities, const hkArray<hkpPhantom*>
      & newPhantoms ) const
    {
        return HK_NULL;
    }
```

... continued from above:

```
    float m_strength;
};



//
// hkpPhantom interface implementation
//

MyPhantomShape::MyPhantomShape()
{
    m_actionType = ACTION_WIND;
    m_direction.setZero4();
    m_strength = 0.0f;
}

/*virtual*/ void MyPhantomShape::phantomEnterEvent( const hkpCollidable* phantomColl, const hkpCollidable*
  otherColl, const hkpCollisionInput& env )
{
    hkpRigidBody* phantom = hkGetRigidBody( phantomColl );
    hkpRigidBody* rb = hkGetRigidBody( otherColl );
    if( !phantom || !rb )
    {
        return;
    }

    // Create the appropriate action
    hkpAction* action = HK_NULL;
    switch( m_actionType )
    {
        case ACTION_WIND:
            {
                hkVector4 force;
                force.setMul4( m_strength, m_direction );
                action = new MyWindAction( phantom, rb, force );
            }
            break;
        case ACTION_ATTRACT:
            {
                action = new MyAttractAction( phantom, rb, m_strength );
            }
            break;
        case ACTION_DEFLECT:
            {
                // use the attract action with a negative strength
                action = new MyAttractAction( phantom, rb, -m_strength );
            }
            break;
    }

    // Add it to the world
    if( action )
    {
        rb->getWorld()->addAction( action );
        action->removeReference();
    }
}
```

...continued from above:

```
/*virtual*/ void MyPhantomShape::phantomLeaveEvent( const hkpCollidable* phantomColl, const hkpCollidable*
  otherColl )
{
    hkpRigidBody* phantom = hkGetRigidBody( phantomColl );
    hkpRigidBody* rb = hkGetRigidBody( otherColl );
    if( !phantom || !rb )
    {
        return;
    }

    // Find and remove the appropriate action
    int numActions = phantom->getNumActions();
    for( int i=numActions-1; i>=0; --i )
    {
        hkpAction* action = phantom->getAction(i);

        if (action)
        {
            hkArray<hkpEntity*> entities;
            action->getEntities( entities );
            for( int j=0; j<entities.getSize(); ++j )
            {
                if( entities[j] == rb )
                {
                    action->getWorld()->removeAction( action );
                }
            }
        }
    }
}
```

In this CPP we define three custom binary actions (`MyWindAction`, `MyAttractAction` and `MyDeflectAction`) that apply different types of forces to the associated objects.

Note:

For simplicity, we didn't properly implement the `clone()` method of those actions - but you should do so if you wanted your action to be cloneable.

Our phantom shape (`MyPhantomShape`) implements the `phantomEnterEvent()` method by creating one of the three actions (depending on the `m_actionType` member) and adding it to the world. The `phantomLeaveEvent()` method, in turn, removes those actions.

Making the Phantom Shape serializable

You may have noticed the **HK_DECLARE_REFLECTION()** ; macro in `MyPhantomShape.h`. This macro is used by serialization, it tells our scripts that this class should have an associated `hkClass` and `hkTypeInfo`.

Since our filter will create objects of this new type (`MyPhantomShape`) that will be serialized, we need to declare an `hkClass` associated with it (please check the *Serialization* documentation for details). If you run the `regenerateXml.py` script (please check the *Serialization > Using the Infrastructure* documentation for details), the following CPP will be generated - add it to the project:

6.7.9.9 Example: convertophantomaction/MyPhantomShapeClass.cpp (autogenerated)

```
// WARNING: THIS FILE IS GENERATED. EDITS WILL BE LOST.
// Generated from 'ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/MyPhantomShape.h'
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>
#include <Common/Base/Reflection/hkClass.h>
#include <Common/Base/Reflection/hkInternalClassMember.h>
#include <Common/Base/Reflection/hkTypeInfo.h>
#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/MyPhantomShape.h>



//
// Enum MyPhantomShape::ActionType
//
static const hkInternalClassEnumItem MyPhantomShapeActionTypeEnumItems[] =
{
    {0, "ACTION_WIND"},
    {1, "ACTION_ATTRACT"},
    {2, "ACTION_DEFLECT"},
};
static const hkInternalClassEnum MyPhantomShapeEnums[] = {
    {"ActionType", MyPhantomShapeActionTypeEnumItems, 3, HK_NULL, 0 }
};
extern const hkClassEnum* MyPhantomShapeActionTypeEnum = reinterpret_cast<const hkClassEnum*>(&
  MyPhantomShapeEnums[0]);

//
// Class MyPhantomShape
//
HK_REFLECTION_DEFINE_VIRTUAL(MyPhantomShape);
static const hkInternalClassMember MyPhantomShapeClass_Members[] =
{
    { "actionType", HK_NULL, MyPhantomShapeActionTypeEnum, hkClassMember::TYPE_ENUM, hkClassMember::
      TYPE_INT8, 0, 0, HK_OFFSET_OF(MyPhantomShape,m_actionType), HK_NULL },
    { "direction", HK_NULL, HK_NULL, hkClassMember::TYPE_VECTOR4, hkClassMember::TYPE_VOID, 0, 0,
      HK_OFFSET_OF(MyPhantomShape,m_direction), HK_NULL },
    { "strength", HK_NULL, HK_NULL, hkClassMember::TYPE_REAL, hkClassMember::TYPE_VOID, 0, 0, HK_OFFSET_OF
      (MyPhantomShape,m_strength), HK_NULL }
};
extern const hkClass hkpPhantomCallbackShapeClass;

extern const hkClass MyPhantomShapeClass;
const hkClass MyPhantomShapeClass(
    "MyPhantomShape",
    &hkpPhantomCallbackShapeClass, // parent
    sizeof(MyPhantomShape),
    HK_NULL,
    0, // interfaces
    reinterpret_cast<const hkClassEnum*>(MyPhantomShapeEnums),
    1, // enums
    reinterpret_cast<const hkClassMember*>(MyPhantomShapeClass_Members),
    HK_COUNT_OF(MyPhantomShapeClass_Members),
    HK_NULL, // defaults
    HK_NULL, // attributes
    0
    );
```

We will see later on how to register this class. Notice that, since the three actions (`MyWindAction`, `MyReflectAction` and `MyAttractAction`) are only run-time actions (that is, are created on the fly and never serialized) we don't need to create or register any `hkClass` for them.

Creating our Convert To Phantom Action filter

Let's now create our filter. Filter (`hctFilterInterface`) classes are always paired with filter descriptors (`hctFilterDescriptor`). There is only one descriptor object per class, giving information about the type of filter (its name, category, etc) while each filter object represent an individual filter in the filter manager.

6.7.9.10 Example: convertophantomaction/hctConvertToPhantomActionFilter.h

```
#ifndef HAVOK_FILTER_CONVERT_TO_PHANTOM_ACTION_H
#define HAVOK_FILTER_CONVERT_TO_PHANTOM_ACTION_H

#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionOptions.h>

#include <Common/Base/Container/Array/hkObjectArray.h>

/*
** This filter is part of the "Toolchain Integration" tutorial of the Havok Content Tools.
** Check the Havok Content Tools documentation for details
*/
class hctConvertToPhantomActionFilter : public hctFilterInterface
{
    public:

        HK_DECLARE_CLASS_ALLOCATOR(HK_MEMORY_CLASS_EXPORT);

        // Constructor - takes a pointer to the filter manager that created the filter
        hctConvertToPhantomActionFilter(const hctFilterManagerInterface* owner);

        // Destructor
        /*virtual*/ $\sim$hctConvertToPhantomActionFilter();

        // The main method in any filter, it does the job
        /*virtual*/ void process( class hkRootLevelContainer& data, bool batchMode );

        // Asks the filter to show its UI for options
        /*virtual*/ HWND showOptions(HWND owner);

        // Asks the fitler to close the options UI (and store any changes)
        /*virtual*/ void hideOptions();

        // Storage of option data
        /*virtual*/ void setOptions(const void* optionData, int optionDataSize, unsigned int version);
        /*virtual*/ int getOptionsSize() const;
        /*virtual*/ void getOptions(void* optionData) const;

    public:

        // The options dialog window
        HWND m_optionsDialog;

        // Called by other methods, it makes sure the stored options (m_options) match what the values in
          the UI
        void updateOptions();

        // The options associated with this filter
        hctConvertToPhantomActionOptions m_options;

        // A buffer for storing the options in XML form.
        mutable hkArray<char> m_optionsBuf;

};

// The filter descriptor is used by the filter manager to query information about the filter (name,
  category, etc)
// as well as to create individual instances.
class hctConvertToPhantomActionFilterDesc : public hctFilterDescriptor
{
    public:
```

. . . continued from above:

```
        /*virtual*/ unsigned int getID() const { return 0x9376be9a; }
        /*virtual*/ FilterCategory getCategory() const { return HK_CATEGORY_USER; }
        /*virtual*/ FilterBehaviour getFilterBehaviour() const { return HK_DATA_MUTATES_INPLACE; }
        /*virtual*/ const char* getShortName() const { return "Convert To Phantom Action"; }
        /*virtual*/ const char* getLongName() const { return "[TUTORIAL] Converts specific rigid bodies to
          phantom actions. Must be placed after the Create Rigid Bodies filter. "; }
        /*virtual*/ unsigned int getFilterVersion() const { return HCT_FILTER_VERSION(1,0,0); }
        /*virtual*/ hctFilterInterface* createFilter(const class hctFilterManagerInterface* owner) const {
          return new hctConvertToPhantomActionFilter(owner); }

        /*virtual*/ HavokComponentMask getRequiredHavokComponents () const { return HK_COMPONENT_PHYSICS;
          }
};

extern hctConvertToPhantomActionFilterDesc g_convertToPhantomActionDesc;

#endif // HAVOK_FILTER_CONVERT_TO_PHANTOM_ACTION_H
```

We've marked virtual methods defined in parent classes (`hctFilterInterface` and `hctFilterDescriptor`) and overriden here with the **/*virtual*/** comment in order to distinguish them from other methods.

Let's start with the filter descriptor, `hctConvertToPhantomActionFilterDesc`. As you can see (and as its name implies) one its responsibilities is to describe the filter (check the documentation for hctFilterDescriptor for details):

- It gives it a unique ID
- It assigns a category (the tab in the filter manager). For user-defined filter, we recommend you use the **HK_USER** category.
- It specifies how does the filter operate with data - in our case, it changes it.
- It returns a short name and a long name (description) for the filter
- It defines what Havok components are required to support the output of this filter. In this case, the Havok Physics component is required.
- It creates individual instances of the filter.

The only instance of the descriptor (`g_convertToPhantomActionDesc`) is declared here (and defined in `hkConvertToPhantom.cpp`)

Let's look at the filter, `hkConvertToPhantonActionFilter`. The most important method is `process()`. We will look at the other methods (related to the display and storage of filter options) later on when discussing filter options.

Let's see how to implement this `process()` method:

6.7.9.11 Example: convertophantomaction/hctConvertToPhantomActionFilter.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>

#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionFilter.h>

#include <Physics/Utilities/Serialize/hkpPhysicsData.h>

#include <Common/SceneData/Graph/hkxNode.h>
#include <Common/SceneData/Attributes/hkxAttributeGroup.h>

#include <Common/Base/Math/Util/hkMathUtil.h>

#include <Physics/Dynamics/Entity/hkpRigidBody.h>
#include <Physics/Dynamics/World/hkpPhysicsSystem.h>

#include <Common/Serialize/Util/hkBuiltinTypeRegistry.h>
#include <Common/Base/Reflection/hkClass.h>
#include <Common/Base/Reflection/hkTypeInfo.h>

#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/MyPhantomShape.h>
#include <Physics/Collide/Shape/Misc/Bv/hkpBvShape.h>

// The only instance of our filter descriptor
hctConvertToPhantomActionFilterDesc g_convertToPhantomActionDesc;

hctConvertToPhantomActionFilter::hctConvertToPhantomActionFilter(const hctFilterManagerInterface* owner)
:    hctFilterInterface (owner)
{
    m_options.m_removeMeshes = false;
    m_optionsDialog = HK_NULL;
}

hctConvertToPhantomActionFilter::$\sim$hctConvertToPhantomActionFilter()
{

}

void hctConvertToPhantomActionFilter::process( hkRootLevelContainer& data, bool batchMode )
{
    // Find and hkxScene and a hkPhysics Data objects in the root level container
    hkxScene* scenePtr = reinterpret_cast<hkxScene*>( data.findObjectByType( hkxSceneClass.getName() ) );
    if (scenePtr == HK_NULL || (scenePtr->m_rootNode == HK_NULL) )
    {
        HK_WARN_ALWAYS(0xabbaa5f0, "No scene data (or scene data root hkxNode) found. Can't continue.");
        return;
    }
    hkpPhysicsData* physicsPtr = reinterpret_cast<hkpPhysicsData*>( data.findObjectByType(
      hkpPhysicsDataClass.getName() ) );
    if (physicsPtr == HK_NULL)
    {
        HK_WARN_ALWAYS(0xabbaa3d0, "No physics data found, you need to use Create Rigid Bodies before this
          filter or have bodies already in the input data.");
        return;
    }

    // Keep track of how many phantoms we created so we can reported
    int numConverted = 0;

    // Search for rigid bodies in the scene
    for (int psi=0; psi<physicsPtr->getPhysicsSystems().getSize(); psi++)
    {
        const hkpPhysicsSystem* psystem = physicsPtr->getPhysicsSystems()[psi];
```

... continued from above:

```
        for (int rbi=0; rbi<psystem->getRigidBodies().getSize(); rbi++)
        {
            hkpRigidBody* rbody = psystem->getRigidBodies()[rbi];
            const char* rbName = rbody->getName();

            // Require an associated node in the scene
            hkxNode* rbNode = (rbName)? scenePtr->findNodeByName(rbName): HK_NULL;
            if( !rbNode )
            {
                continue;
            }

            // Require an 'hkPhantomAction' attribute group
            const hkxAttributeGroup* attrGroup = rbNode->findAttributeGroupByName("hkPhantomAction");
            if (!attrGroup)
            {
                continue;
            }

            // Create our phantom shape
            MyPhantomShape* myPhantomShape = new MyPhantomShape();
            {
                // Set action type (required)
                const char* actionTypeStr = HK_NULL;
                attrGroup->getStringValue( "action", true, actionTypeStr );
                if( actionTypeStr )
                {
                    if( hkString::strCasecmp( actionTypeStr, "wind" ) == 0 )
                    {
                        myPhantomShape->m_actionType = MyPhantomShape::ACTION_WIND;
                    }
                    else if( hkString::strCasecmp( actionTypeStr, "attract" ) == 0 )
                    {
                        myPhantomShape->m_actionType = MyPhantomShape::ACTION_ATTRACT;
                    }
                    else if( hkString::strCasecmp( actionTypeStr, "deflect" ) == 0 )

                    {
                        myPhantomShape->m_actionType = MyPhantomShape::ACTION_DEFLECT;
                    }
                    else
                    {
                        HK_WARN_ALWAYS(0xabbad3b4, "Unknow action type ("<<actionTypeStr<<").");
                    }
                }
                else
                {
                    HK_WARN_ALWAYS(0xabba9834, "Can't fine \"action\" attribute");
                }

                // Set other attributes (optional)
                attrGroup->getVectorValue( "direction", false, myPhantomShape->m_direction );
                attrGroup->getFloatValue( "strength", false, myPhantomShape->m_strength );

                // Useful warnings
                if ((myPhantomShape->m_actionType == MyPhantomShape::ACTION_WIND) && (myPhantomShape->
                  m_direction.lengthSquared3()==0.0f))
                {
                    HK_WARN_ALWAYS(0xabbadf82, "Wind direction is invalid - action will have no effect");
```

... continued from above:

```
                }
                if (myPhantomShape->m_strength==0.0f)
                {
                    HK_WARN_ALWAYS(0xabbacc83, "Strength is 0 - action will have no effect");
                }

            }

            // Set the phantom as a new bounding shape for the body
            {
                const hkpShape* oldShape = rbody->getCollidable()->getShape();

                hkpBvShape* bvShape = new hkpBvShape( oldShape, myPhantomShape );
                myPhantomShape->removeReference();

                rbody->setShape( bvShape );
                bvShape->removeReference();
            }

            // Remove meshes if the user chose to do so
            if (m_options.m_removeMeshes && (hkString::strCmp(rbNode->m_object.m_class->getName(), "
              hkxMesh")==0) )
            {
                rbNode->m_object.m_object = HK_NULL;
            }

            HK_REPORT("Converted rigid body \""<<rbName<<"\" to phantom action");
            numConverted++;
        }
    }

    // Give a warning if the filter didn't do anything useful
    if (numConverted==0)
    {
        HK_WARN_ALWAYS(0xabba7632, "No rigid bodies converted to phantom action.");
    }
    else
    {
        HK_REPORT("Converted "<<numConverted<<" rigid bodies.");
    }
}
```

The method takes an `hkRootLevelContainer` object (*data*), and a boolean (`batchMode`) flag. The *batchMode* flag tells the filter whether the user is processing in batch mode or not. When running in batch mode, filters should not open any UI or block the processing. Our filter won't do any of both, so we can safely ignore this flag.

The `hkRootLevelContainer` object (data) contains the contents that our filters works with and modifies. We start by looking for scene data (an `hkxScene` object) -so we can look at the attributes- and physics data (an `hkpPhysicsData` object) -so we can look at rigid bodies- inside it.

Once those two are found, we look at all rigid bodies (`hkpRigidBody`) objects in our physic data, and find the matching `hkxNode` in the scene (by comparing names). If that node has an `hkPhantomAction` attribute group, then we have a candidate to be processed. Otherwise we ignore it.

If we are succesful (we have an `hkpRigidBody` with an `hkPhantomShape` attribute group), we create our phantom shape (`MyPhantomShape`) and use methods in `hkxAttributeGroup` to query for attribute values and pass them to the phantom shape. We then wrap the original shape and the phantom with an

`hkpBvShape` (check the Havok Dynamics chapter for details).

The last step optionally removes the association node -> mesh (so the rigid body mesh won't be displayed in the preview scene filter). We will discuss how this option can be added to the filter in the following section (*Adding options to the filter*).

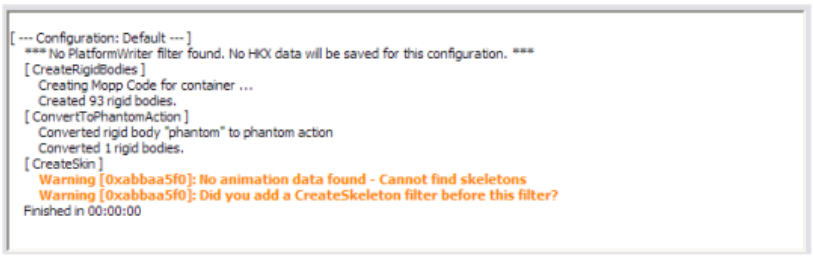
Warnings and Reporting

When processing assets, it is very useful to report the activity of the filter and raise any warnings and errors that may be detected. In our example above, we raise warnings whenever processing fails or has no effect. We also report the name of any rigid body converted, as well as a total at the end of processing.

You can use the following macros to report progress and warnings:

- **HK_REPORT (text)** : Reports some information
- **HK_REPORT_SECTION_BEGIN (id, name)** : Starts a section (following HK_REPORT calls will be nested in this group)
- **HK_REPORT_SECTION_END ()** : Finishes a section
- **HK_WARN_ALWAYS (id, text)** : Reports a warning. The id can be used to identify individual warnings. For toolchain warnings, we use the range `0xabba0000 - 0xabbaffff` as a convention.
- **HK_WARN (id, text)** : Reports a warning *in debug only*.

You can also use other SDK error reporting macros, like **HK_ASSERT** and **HK_ERROR** . These macros, however, will abort execution of the filter. For filter processing, it is usually a better and safer approach to detect, warn, and recover gracefully from unexpected situations rather than using assertions.



Adding options to the filter

Let's now take a look at how to add options to our filter. In our case, we are going to give the option to remove the association of a rigid body node to its original mesh whenever it is converted to a phantom (so the phantom won't be displayed by the preview scene filter).

We will call this option "*removeMeshes*" - we saw in the previous section how to use it during the `process()` method. Let's see now how to store this option with the filter and how to display it in the filter manager.

Storing options with a filter

Each filter gets an opportunity to save and retrieve option data, which is stored with the filter configuration (and usually saved with the asset, or into an .HKO file). Since options need to be serialized, we'll start by putting our option in a struct:

6.7.9.12 Example: convertophantomaction/hctConvertToPhantomActionOptions.h

```
#ifndef HKCONVERTOPHANTOMOPTIONS__H
#define HKCONVERTOPHANTOMOPTIONS__H

/// hkConvertToPhantomOptions meta information
extern const class hkClass hctConvertToPhantomActionOptionsClass;

/// Options associated to our Convert To Phantom Action filter.
struct hctConvertToPhantomActionOptions
{
    HK_DECLARE_NONVIRTUAL_CLASS_ALLOCATOR( HK_MEMORY_CLASS_EXPORT, hctConvertToPhantomActionOptions );
    HK_DECLARE_REFLECTION();

    /// Option to remove associated meshes
    hkBool m_removeMeshes;
};

#endif // HKCONVERTOPHANTOMOPTIONS__H
```

As with our custom action, adding the **HK_DECLARE_REFLECTION()** macro flags this class so the serialization scripts will generate an associated `hkClass` object in a .cpp. If you run the `regenerateXml.py` script (please check the *Serialization > Using the Infrastructure* documentation for details), the following CPP will be generated - add it to the project too:

6.7.9.13 Example: convertophantomaction/hctConvertToPhantomActionOptionsClass.cpp

```
// WARNING: THIS FILE IS GENERATED. EDITS WILL BE LOST.
// Generated from 'ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionOptions.h'
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>
#include <Common/Base/Reflection/hkClass.h>
#include <Common/Base/Reflection/hkInternalClassMember.h>
#include <Common/Base/Reflection/hkTypeInfo.h>
#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionOptions.h>



//
// Class hctConvertToPhantomActionOptions
//
HK_REFLECTION_DEFINE_SIMPLE(hctConvertToPhantomActionOptions);
static const hkInternalClassMember hctConvertToPhantomActionOptionsClass_Members[] =
{
    { "removeMeshes", HK_NULL, HK_NULL, hkClassMember::TYPE_BOOL, hkClassMember::TYPE_VOID, 0, 0,
      HK_OFFSET_OF(hctConvertToPhantomActionOptions,m_removeMeshes), HK_NULL }
};
extern const hkClass hctConvertToPhantomActionOptionsClass;
const hkClass hctConvertToPhantomActionOptionsClass(
    "hctConvertToPhantomActionOptions",
    HK_NULL, // parent
    sizeof(hctConvertToPhantomActionOptions),
    HK_NULL,
    0, // interfaces
    HK_NULL,
    0, // enums
    reinterpret_cast<const hkClassMember*>(hctConvertToPhantomActionOptionsClass_Members),
    HK_COUNT_OF(hctConvertToPhantomActionOptionsClass_Members),
    HK_NULL, // defaults
    HK_NULL, // attributes
    0
    );
```

Notice in `hctConvertToPhantomActionFilter.h` how an instance of this `hctConvertToPhantomActionOptions` class (`m_options`) is stored in the filter, initialized in the constructor and used during processing. In order to store these options with the filter, three methods must be implemented:

- `int getOptionsSize ()` : Must return the size (in bytes) of the serialized XML representing the options
- `void getOptions (buffer)` : Must filled the passed buffer with the XML representing the options
- `void setOptions (buffer, size, version)` : The filter can read the given XML options from the buffer. The version parameter can be used to detect older versions of the options (and upgrade them accordingly).

Let's check how to implement these methods:

6.7.9.14 Example: convertophantomaction/hkConvertToPhantomAction_Options.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>

#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionFilter.h>

void hctConvertToPhantomActionFilter::setOptions(const void* optionData, int optionDataSize, unsigned int
  version )
{
    // Get the options from the XML data.
    if ( hctFilterUtils::readOptionsXml( optionData, optionDataSize, m_optionsBuf,
      hctConvertToPhantomActionOptionsClass ) == HK_SUCCESS )
    {
        hctConvertToPhantomActionOptions* options = reinterpret_cast<hctConvertToPhantomActionOptions*>(
          m_optionsBuf.begin() );

        m_options.m_removeMeshes = options->m_removeMeshes;
    }
    else
    {
        HK_WARN_ALWAYS( 0xabba482b, "The XML for the " << g_convertToPhantomActionDesc.getShortName() << "
          option data could not be loaded." );
        return;
    }
}

int hctConvertToPhantomActionFilter::getOptionsSize() const
{
    // We write the options to a temporary buffer and return the size of it. The buffer itself will be
      used
    // later on by getOptions().
    hctFilterUtils::writeOptionsXml( hctConvertToPhantomActionOptionsClass, &m_options, m_optionsBuf,
      g_convertToPhantomActionDesc.getShortName() );
    return m_optionsBuf.getSize();
}

void hctConvertToPhantomActionFilter::getOptions(void* optionData) const
{
    // Get options is always called after getOptionsSize() - so we reuse the temporary buffer we used in
      getOptionsSize()
    hkString::memCpy( optionData, m_optionsBuf.begin(), m_optionsBuf.getSize() );
```

In order to read the options, we use the utility `hctFilterUtils::readOptionsXML()`, and then copy the temporary data in the buffer to our stored options, `m_options`.

For retrieving the options, we serialize them into a temporary buffer, returning its size in `getOptionsSize()` (which always called first) and its contents in `getOptions()`.

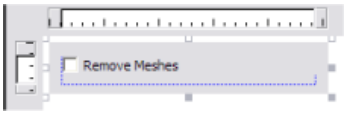
Displaying Options

Now that we know how to use, store and load our options, let's see how to show them in the filter manager UI. Two methods in hctFilterInterface are used for this:

- `HWND showOptions (HWND owner)` : Must open the filter options window inside the given window control, and it must return the dialog handle of the new created window.
- `void hideOptions()` : This is called whenever option editing must be finished (when closing the filter manager or when executing filters). The filter must make sure the stored options match the

displayed options.

We'll start by creating the dialog (**IDD_CONVERT_TO_PHANTOM_ACTION_DIALOG**) to show our options. In order to fit in the right side pane of the filter manager, the width of the dialog must be 153 pixels, and it's has to be of *Child* style, with no border. We add a check box control for the "*Remove Meshes*" option, which we name **IDC_CB_RemoveMeshes**



Let's see know how to show and hide this dialog:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.7.9.15 Example: convertophantomaction/hkConvertToPhantomAction_Dialog.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>

#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionFilter.h>

extern HINSTANCE hInstance;

BOOL CALLBACK _convertToPhantomActionDialogProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    // We store a pointer to the filter associated with this dialog using Get/SetWindowLongPtr()
    hctConvertToPhantomActionFilter* filter = reinterpret_cast<hctConvertToPhantomActionFilter*> ( (
      hkUlong) GetWindowLongPtr(hWnd,GWLP_USERDATA)) ;

    switch(message)
    {
        case WM_INITDIALOG:
            {
                filter = (hctConvertToPhantomActionFilter*)lParam;
                SetWindowLongPtr(hWnd, GWLP_USERDATA, (LONG)lParam); // so that it can be retrieved later

                CheckDlgButton(hWnd, IDC_CB_RemoveMeshes, filter->m_options.m_removeMeshes);

                return TRUE; // did handle it
            }
    }
    return FALSE; //didn't handle it
}

void hctConvertToPhantomActionFilter::updateOptions()
{
    // Ensure the options we store match the options shown in the UI
    if (m_optionsDialog)
    {
        m_options.m_removeMeshes = IsDlgButtonChecked(m_optionsDialog, IDC_CB_RemoveMeshes) == TRUE;
    }
}


HWND hctConvertToPhantomActionFilter::showOptions(HWND owner)
{
    if (m_optionsDialog)
    {
        hideOptions();
    }

    m_optionsDialog = CreateDialogParam(hInstance, MAKEINTRESOURCE(IDD_CONVERT_TO_PHANTOM_ACTION_DIALOG),
        owner, _convertToPhantomActionDialogProc, (LPARAM) this );

    return m_optionsDialog;
}

void hctConvertToPhantomActionFilter::hideOptions()
{
    // Update any changes before we close UI
    updateOptions();

    if (m_optionsDialog)
    {
        DestroyWindow(m_optionsDialog);
    }

    m_optionsDialog = NULL;
```

. . . continued from above:

```
}
```

We use `CreateDialogParam()` and pass a pointer to the filter so the dialog proc (`convertToPhantomActionDialogProc`) can properly access the filter. Notice how, when closing the dialog, we make sure that whatever changes the user did in the UI are reflected in the stored options.

The filter DLL

Now that we have created our filter and its descriptor, we need to expose them to the DLL so the filter manager can detect and load them. We do so by inheriting a custom class from `hctFilterDll` (see handling dlls for details):

6.7.9.16 Example: hctFilterTutorialDll.h

```
#ifndef HAVOK_FILTER_TUTORIAL_DLL_H
#define HAVOK_FILTER_TUTORIAL_DLL_H

#include <ContentTools/Common/Filters/Common/Filter/hctFilterDll.h>

// This class describes our filter DLL : It provides access to the filter descriptor
// for each filter implemented in the DLL, and registers all classes possibly created in those
// filters.
class hctFilterTutorialDll : public hctFilterDll
{
    public:

        hctFilterTutorialDll (HMODULE dllModule);

        // Must return how many types of filters are implemented in this DLL
        /*virtual*/ int getNumberOfFilters() const;

        // For each filter implemented in this DLL, we must return a filter descriptor
        /*virtual*/ hctFilterDescriptor* getFilterDescriptor (int index) const;

        // Since our filter creates objects, we must register our classes
        /*virtual*/ void registerClasses (hctFilterClassRegistry& classReg) const;

        // There are some statics in the Physics SDK that we want to make sure are initialized
        /*virtual*/ void initDll ( const BaseSystemInfo* baseSystemInfo );

};


// Function exported by the DLL
__declspec( dllexport ) hctFilterDll* getFilterDll (HMODULE dllModule);

#endif // HAVOK_FILTER_TUTORIAL_DLL_H
```

We need to override `getNumberOfFilters()` and `getFilterDescriptor()` since they are pure virtual methods in `hctFilterDll` - and we need to return our own filter descriptor. We also override `registerClasses()` since we must register any serializable classes possibly created by our filters, and initDll() in order to initialize some static data used by Havok Physics.

Notice also how we declare `getFilterDll()` as a function exported by this DLL. This is the only function exported by the DLL.

6.7.9.17 Example: hctFilterTutorialDll.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>

#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorialDll.h>

#include <Common/Serialize/Util/hkBuiltinTypeRegistry.h>

#include <Common/Serialize/Version/hkVersionRegistry.h>

#include <Physics/Collide/Util/Welding/hkpWeldingUtility.h>
#include <Physics/Dynamics/World/hkpWorldCinfo.h>


// Filters we want to expose in this dll:
#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionFilter.h>

__declspec( dllexport ) hctFilterDll* getFilterDll (HMODULE dllModule)
{
    static hctFilterTutorialDll gFilterAssetDll (dllModule);

    return &gFilterAssetDll;
}


hctFilterTutorialDll::hctFilterTutorialDll (HMODULE dllModule) : hctFilterDll (dllModule)
{

}

/*virtual*/ int hctFilterTutorialDll::getNumberOfFilters() const
{
    return 1;
}

/*virtual*/ hctFilterDescriptor* hctFilterTutorialDll::getFilterDescriptor (int index) const
{
    static hctFilterDescriptor* m_filterDescs[] =
    {
        &g_convertToPhantomActionDesc,
    };

    return m_filterDescs[index];
}

/*virtual*/ void hctFilterTutorialDll::initDll ( const BaseSystemInfo* baseSystemInfo )
{
    hctFilterDll::initDll(baseSystemInfo);

    // HVK-3632
    hkpWorldCinfo defaults;
    hkpWeldingUtility::initWeldingTable(defaults.m_snapCollisionToConvexEdgeThreshold,
      defaults.m_snapCollisionToConcaveEdgeThreshold);
}

/*virtual*/ void hctFilterTutorialDll::registerClasses (hctFilterClassRegistry& classReg) const
{
    classReg.registerClasses (hkBuiltinTypeRegistry::StaticLinkedTypeInfos, hkBuiltinTypeRegistry::
      StaticLinkedClasses);

    #define HK_CUSTOM_CLASS(C) \
        extern const hkTypeInfo C##TypeInfo;                    \
        extern const hkClass C##Class;                            \
```

...continued from above:

```
        classReg.registerClass( &(C##TypeInfo), &(C##Class) );

    // Our phantom class
    HK_CUSTOM_CLASS(MyPhantomShape)
}


// REGISTER ALL COMPLETE CLASSES
#define HK_CLASSES_FILE <Physics/Collide/Classes/hkpCollideClasses.h>
#include <Common/Serialize/Util/hkBuiltinTypeRegistry.cxx>

// NO COMPATIBILITY NEEDED
#include <Common/Compat/hkCompat_None.cxx>
```

The implementation of the DLL-exported method `getFilterDll()` just returns a pointer to an statically defined instance of `hctFilterTutorialDll`, and `getFilterDescriptor()` just returns a pointer to the static descriptor declared in `convertophantomaction/hctConvertToPhantomActionFilter.h`.

The `registerClasses()` method registers all classes in `hkCollide` (since we create some of them, like `hkpBvShape`[11]). Notice how we also register our custom shape class, `MyPhantomShape`.

Finally, we create an `hkfiltertutorial.def` file with our DLL exports:

6.7.9.18 Example: hctFilterTutorial.def

```
LIBRARY hctFilterTutorial
EXPORTS
    getFilterDll    @1

SECTIONS
    .data READ WRITE
```

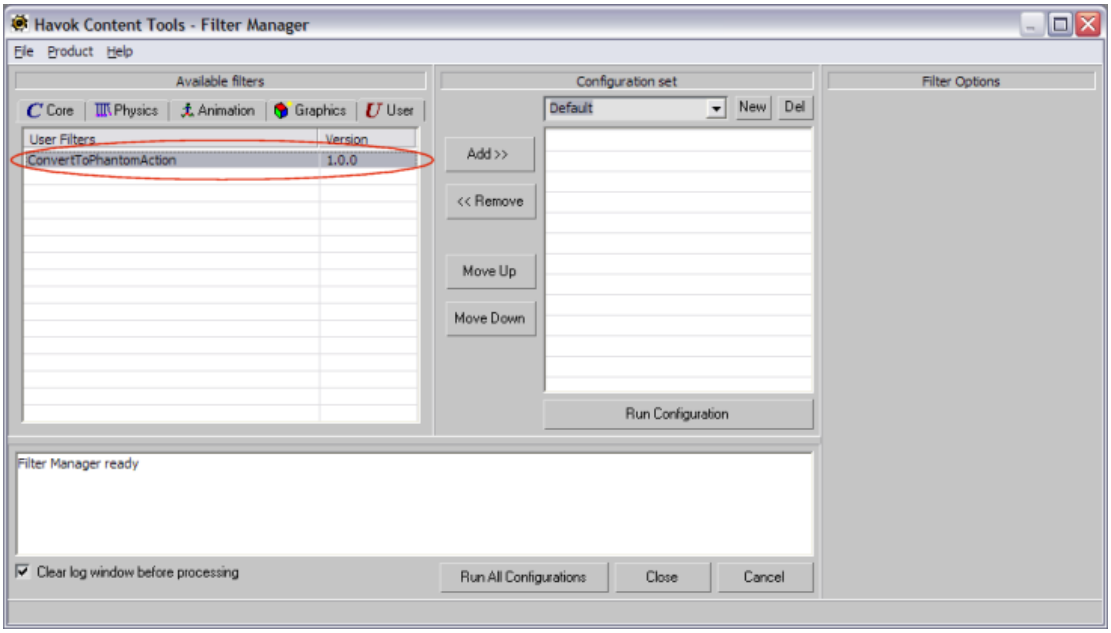
And we are done!

Placing the filter DLL

Make sure you place our new filter DLL in the right folder - look for a "`filters`" folder inside the install folder of the Havok Content Tools (usually "`%ProgramFiles%/havok/HavokContentTools/filters`") and place it there[12]. If you've done that properly, the filter should appear in the right category ("*User*" in our case)

[11] It is very important not to miss any created class - and there is no overhead with registering classes that are actually not created. So in this case we register the whole hkCollide classes even though there will be many classes which we don't create.

[12] The DLL is actually installed by the Havok Content Tools installer so you can check it out before writing it yourself. Feel free to replace it with your custom one.

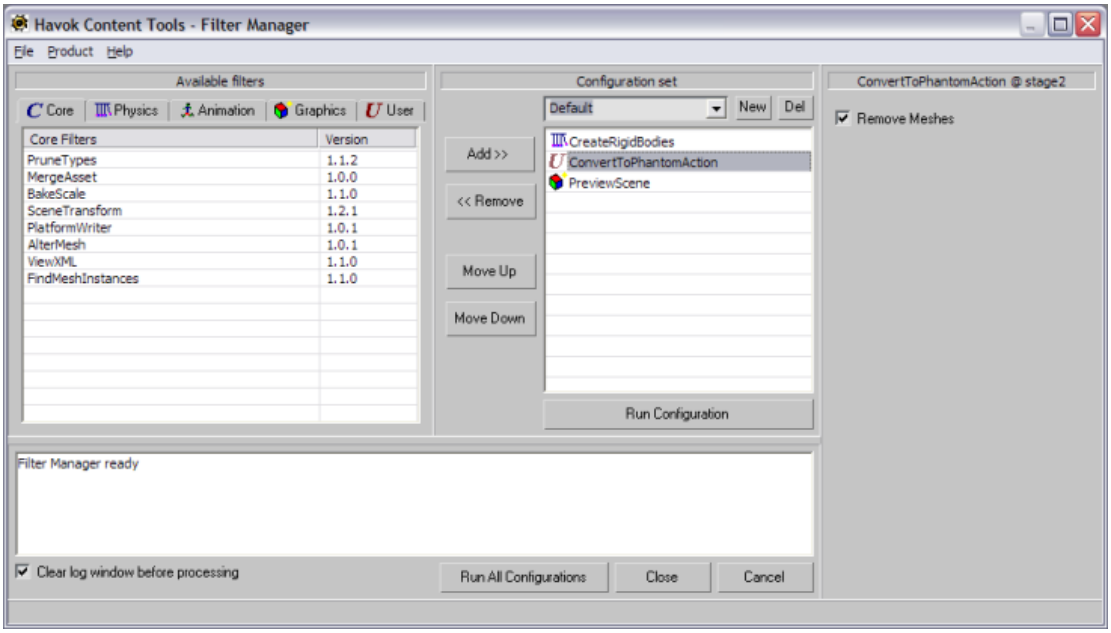


6.7.9.19 Processing

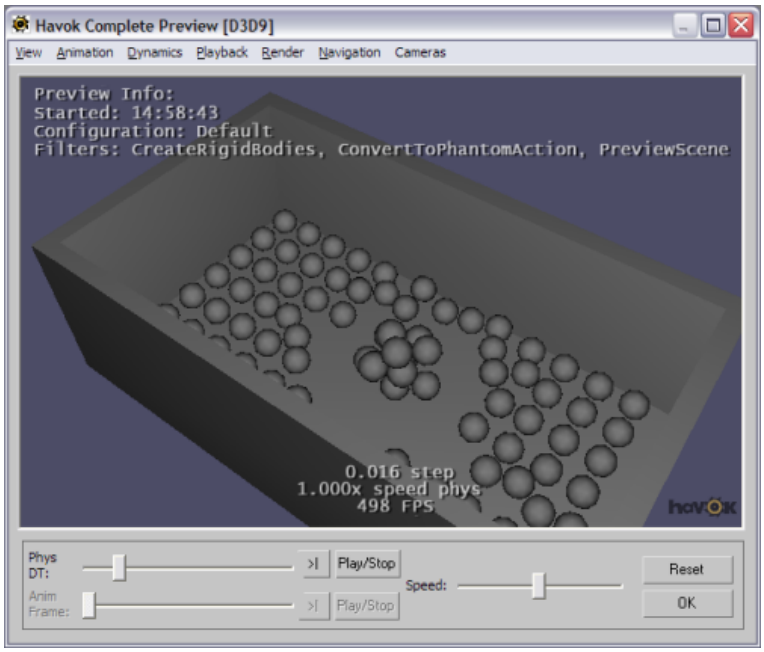
Now that we have the content created and the filter ready, let's set up the processing to create our phantom action in the scene we started with.

Open the filter manager and add the following filters : *Create Rigid Bodies* (*Physics* Category) , *Convert To Phantom Action* (*User*) and *Preview Scene* (*Graphics*):

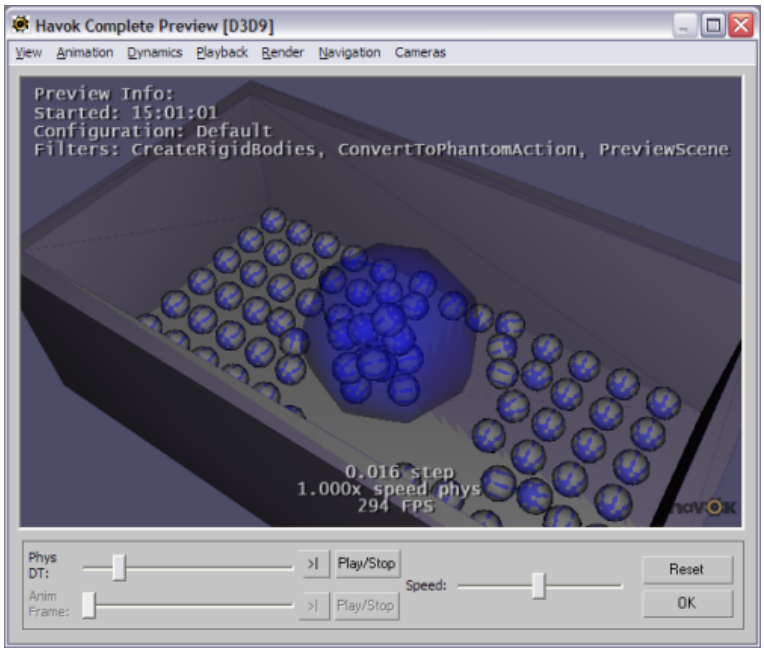
In the *Convert To Phantom Action* filter, check the " **Remove Meshes** " options.



If you press the **Run Configuration** button, the filters will be execute and the Preview Scene filter will open a preview window. Press **Play** to start the simulation:



Notice how the spheres entering the volume of our phantom are now attracted to its center, and how the mesh for the phantom is not displayed. You can use the rigid body ghosts display to view the phantom shape. Click on **View** > **Physics Ghosts** :



And we are done! Feel free to experiment with the different action types and different parameter values.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.7.9.20 Going further

In this tutorial we've focused on the basic ideas behind extending the toolchain both on the modeller side and the filter processing side. Here are some other suggestions and improved ways to extend the toolchain

More powerful attribute definitions

In our tutorial we showed how to attach extra attributes to nodes using each of the modeller's UI. This had the advantage of not requiring any knowledge about scripting or plugin development. However, in order to provide more powerful and reusable attribute definitions and UI, you will need to resort to each of the modeller-specific ways to define types. We will mention a few of them:

3ds Max

- In 3ds Max you can create Custom Attribute Definitions using the MAXScript language (and to some extent, C++). This has the advantatge of being able to customize the UI behaviour, handle versioning, and reuse the attribute definition across multiple objects.
- Modifiers are also inspected during export and exported as attribute groups, so you can use custom modifiers to add attributes to objects. Modifiers are the most powerful option - they have the advantage of providing callbacks for display and functionality for manipulation. The Havok Content Tools for 3ds Max use custom modifiers for their rigid body, shape and constraint attributes.

Please check the exporting custom data from 3ds Max for details on how the scene is explored and attributes exported in 3ds Max. Check the 3ds Max SDK and MAXScript documentation for details on how to create Custom Attributes and modifiers.

Maya

- In Maya you can use MEL to automate the creation of custom attributes - the disadvantage being that you have little control on how they are displayed and versioned.
- The most powerful option in Maya is to create custom nodes (shape or locators) using C++ - the exporter will export them (if named and connected properly) as attribute groups associated with the node. Among the advantages are the fact that you can customize the attribute UI (using templates), versioning, and, in case of locators, you can also have a custom display in the viewports. The Havok Content Tools for Maya use custom nodes for their rigid body, shape and constraint attributes.

Please check the exporting custom data from Maya for details on how the scene is explored and attributes exported in Maya. Check the Maya documentation for details on how to use MEL and the C++ SDK to create attributes, custom nodes and locators.

XSI

- You can automate the attribute creation process by using scripting to add the attributes - this, however, still doesn't provide much flexibility regarding UI behaviour.
- You can create Custom Properties (Parameter Sets) with self-installing plugins, using any of the languages (C++, VBScript, JScript, Python..) supported by XSI. Doing this has the advantage

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

that you can customize the layout and event handling of the property pages (PPG) when working with those properties. The Havok Content Tools for XSI use Custom Properties for their rigid body, shape and constraint attributes.

Please check the exporting custom data from XSI for details on how the scene is explored and attributes exported in XSI. Check the XSI SDK documentation for details on how to create attributes and custom properties.

Extending current types

If you want to extend an already defined attribute group with a custom parameter, you can either:

- Modify the original definition of the attribute, or
- Attach an extra attribute group named as "*xxxxMerge*", where "*xxxx*" is the name of the original one : the exporters will merge the new attributes into the original group.

In order to handle the extra attribute during processing, you can either extend the original filter to handle the new attribute or create a new filter that takes the output from the previous filter and then interprets the new attribute (very similar to what we have done in this tutorial).

Sometimes it may be easier to extend the toolchain with transformations rather than new types. Consider the example of our tutorial : rather than creating a completely new "hkPhantomShapeAction" type and filter process, we reused the data and processing for rigid bodies, and then applied a "convert rigid body to phantom" process. Similarly, if you are considering adding a new type to the toolchain, it may be worth considering the option of treating the new type as a "mutation" of a previous type.

Other examples

This tutorial showed how to extend the toolchain with a custom action but the same mechanism is applicable to other toolchain extensions. Here are some examples:

Using user-defined "materials" to override rigid body properties

If you wanted to override the mass, friction and restitution of rigid bodies by a value based on a choice of predefined "materials", the steps would be very similar to those explained in this tutorial:

- Define a custom attribute group (*hkCustomMaterial* for example) with a Material attribute (an enum) to the modeller, adding it to those rigid bodies you want to apply the material to
- Create a custom filter (*Apply Custom Material* for example) that, whenever this attribute group is found, it overrides the mass, friction and restitution of the rigid body with values stored in a "material table". The object's mass, for example, could be calculated from the material's density and the rigid body's volume.

The "material table" could be statically defined in the filter, exposed to filter options, it could be based on external text files, etc..

Replicating a rigid body at multiple marker points

If you wanted the ability to specify arbitrary points in space to represent locations where a rigid body should be replicated, the steps would again be very similar:

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

- Define a custom attribute group (*hkRigidBodyClone* for example) with a RigidBodyName attribute (a string[13]), and apply it to dummy objects in the scene representing the location where the rigid body should be cloned
- Create a custom filter (*Clone Rigid Bodies* for example) that detects nodes in the scene with that attribute group, searches for a rigid body with the specified name, and clones it at the location of the node.

[13] Or a "node reference" if the modeller supports that type (3ds Max for example). Node references are converted to strings during export.

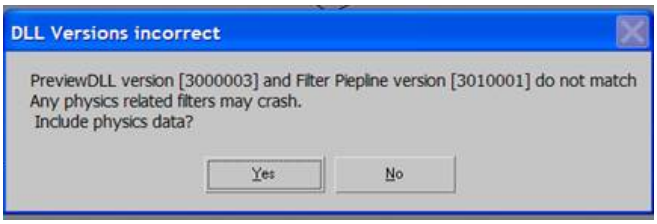
Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.8 Troubleshooting

6.8.1 Troubleshooting: Processing Assets

6.8.1.1 "DLL Versions Incorrect" dialog appears during export

If when exporting the following dialog appears:



This means that you have the Legacy Physics Exporter installed and the current scene has physics content created with it. The Legacy Physics Exporter creates its run-time content by using a DLL (`hkpreview.dll`) which contains the latest Havok runtime. The Scene Exporters for max and maya are able to detect the presence of the Legacy Physics Exporter and will attempt to merge the physics content created with it with the exported scene. However, in order for this to work properly, it is very important that the (`hkpreview`) DLL is built with the same Havok SDK version as the filters and filter manager.

Usually the installer for the Havok Content Tools will replace the hkpreview DLL with the proper version - but if, for example, you installed the Legacy Physics Exporters after the Havok Content Tools, an incompatible version may be installed. You can either:

- Reinstall the Havok Content Tools so they update the hkpreview DLL, or
- Copy the `hkpreview.dll` manually from the Havok SDK installation (in the "`tools`" folder) into the proper location for your modeller. The `ReadMe.txt` file located in the same directory contains directions for 3ds Max and Maya.

You can also ignore any legacy physics content by switching off the " **Invoke Legacy Physics Exporter** " option in the Scene Exporter Options (check the related documentation for the 3ds Max and Maya Scene Exporters)

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.8.1.2 Filter Manager doesn't appear after exporting the scene

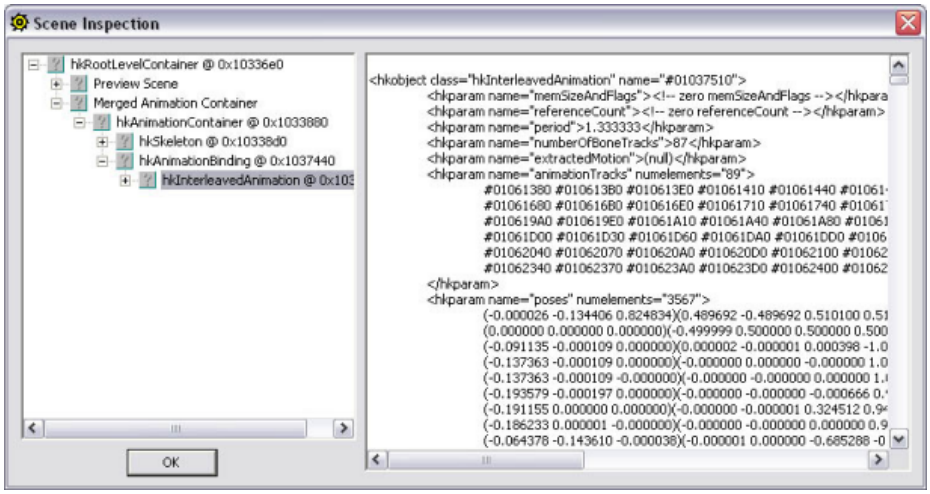
If the Filter Manager dialog does not appear then it's likely that processing has failed during export.

- If you have build a custom version of the export utilities then use normal modeller plugin debugging procedures.
- If you have not modified the export utilities then send the original asset (.max, .mb file) to support for analysis.

6.8.1.3 Asset is processed, but does not load at run-time

If the asset is processed successfully through the filter pipeline, but it does not load at runtime:

- Ensure the correct versions of the filters and filter manager are installed. Version information for filters is listed beside each filter in the available filters window. Version information for the filter manager can be found in the Help->About menu.
- Check the contents of the asset using the standalone filter manager:
  - Launch the standalone filter manage and load the required asset. As you load assets the title bar in the Loading dialog box changes to reflect the number of files successfully loaded. If you are attempting to load files check the output console window. A warning is shown for each specified file which fails to load. Failure usually occurs because with the file is simply incorrect (i,e, does not contain Havok data) or is the incorrect version.
  - Once the application has launched you will be presented with the filter manager. Include a View XML filter in the stack and use this to browse the asset contents. Ensure that the expected data is contained within the asset and has not been pruned. The image below shows and example of a scene being investigated



6.8.1.4 Create Rigid Bodies filter doesn't create any rigid bodies

Make sure you have added rigid body and shape information to the objects (check the Physics Tools for 3ds Max, Maya or XSI for details on how to do so).

Physics information is stored using node attributes, and collision information is created from shape nodes by looking at the meshes associated with nodes. Ensure that the " **Export Attributes** " and " **Export Meshes** " options are selected in the Scene Export Options (for 3ds Max, Maya or XSI).

6.8.1.5 Can't see anything in the Preview Scene filter

The most common cause for the Preview Scene filter not showing anything is problems with the cameras.

Particularly in 3ds Max, clipping planes may be causing the issue. Try changing the clipping planes through the **Cameras** > **Adjust Camera...** option in the preview.

Ensure that the **View** menu in the preview has enough options switched on, such as **Display Meshes** and/or **Rigid Body Ghosts** .

Problems may also be caused by graphics drivers. Most card manufacturers upgrade their drivers regularly - try downloading the latests version, particularly if you are using Havok FX.

6.8.1.6 The Preview Scene filter doesn't use my chosen renderer

Currently it is not possible to have multiple preview windows open at the same time using different renderers. Each preview that is opened will always use the same renderer as that of any previews which are already open. If the chosen renderer is different to that of any which are already open, then the window title will indicate that the renderer in use is different to the desired one, eg: "`Havok Preview [ D3D9 \textbf{- forced} ]`".

6.8.1.7 Crash in the Preview Scene filter

If you experience a crash in the Preview Scene filter, it may be caused by a configuration-specific problem. Try upgrading your graphics card drivers, and switching to a different renderer (OpenGL / Direct X) in order to fix or workaround the issue (but do not hesitate to report it).

6.8.1.8 Filter configuration changes don't seem to be stored

If your changes to the filter configurations in the filter manager don't seem to be stored/saved when you close the filter manager, ensure that you are not overriding those changes by having the " **Use HKO file** " option selected in your scene exporter (3ds Max / Maya / XSI) options or by having an override / upgrade HKO file present.

6.8.1.9 Can't [properly] export NURBS skin

The Havok Content Tools do not support skinned NURBS - the NURBS surface may be exported as (converted to) a static mesh but the bone-skin binding information will not be exported. This is because skinning a NURBS surface modifies control points while skinning a polygonal mesh modifies vertices - therefore it is not possible to accurately convert a NURBS skin-binding to a polygonal skin-binding [in general, `meshSkin(nurbsToMesh(nurbs),t) != nurbsToMesh(nurbsSkin(nurbs.t))`].

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.8.1.10 Skin not updated in the Preview Scene filter

- Ensure that you have correctly used the Create Skeleton, Create Skin and Create Animation filters before the Preview Scene filter.
- Multiple mesh binding mappings are only supported by hardware skinning. If you have are limiting the number of bones in the Create Skin filter, this will likely have caused the mesh binding to be split into multiple mappings. Enable **Hardware Skinning** in the Preview Scene filter options (DirectX only) or disable the limit on the number of bones in the Create Skin filter.
- Skinned NURBS surfaces are not supported - use polygonal meshes instead.

6.8.1.11 Can't get Extract Motion filter to work

The most common user errors when using the Extract Motion filter are:

- *Wrong coordinate system* : Make sure you have selected the proper coordinate system in the filter options (the **Up/Down** and the **Forward/Back** direction)
- *Non-linear motion* : Make sure you are using the right amount of samples : using 2 samples only (constant velocity) will ignore any non-linear component of the motion - switch to **Same as Animation** in order to extract non-linear motion.

Use the **View** > **Extracted Motion** in the Preview Scene filter to visualize the extracted motion and ensure you get the expected results.

6.8.1.12 Textures do not match original texture in the modeller

This may be caused by the original texture being transformed inside the modeller. When exporting bitmap textures, the scene exporters will just export a reference to the file. Therefore, information about extra transformations applied to the texture (color correction, cropping, scaling or UV transformation) is not exported. In order to get a faithful match, you can:

- Bake the texture transformation into the texture bitmap file (so no transformations are done by the modeller)
- Use UV transformations on the mesh rather than on the texture

6.8.1.13 Skin not updated in the Preview Scene filter

- Ensure that you have correctly used the Create Skeleton, Create Skin and Create Animation filters before the Preview Scene filter.
- Multiple mesh binding mappings are only supported by hardware skinning. If you have are limiting the number of bones in the Create Skin filter, this will likely have caused the mesh binding to be split into multiple mappings. Enable **Hardware Skinning** in the Preview Scene filter options (DirectX only) or disable the limit on the number of bones in the Create Skin filter.Can

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.8.1.14 Can't get Extract Motion filter to work

The most common user errors when using the Extract Motion filter are:

- *Wrong coordinate system* : Make sure you have selected the proper coordinate system in the filter options (the **Up/Down** and the **Forward/Back** direction)
- *Non-linear motion* : Make sure you are using the right amount of samples : using 2 samples only (constant velocity) will ignore any non-linear component of the motion - switch to **Same as Animation** in order to extract non-linear motion.

Use the **View** > **Extracted Motion** in the Preview Scene filter to visualize the extracted motion and ensure you get the expected results.

6.8.1.15 Textures do not match original texture in the modeller

This may be caused by the original texture being transformed inside the modeller. When exporting bitmap textures, the scene exporters will just export a reference to the file. Therefore, information about extra transformations applied to the texture (color correction, cropping, scaling or UV transformation) is not exported. In order to get a faithful match, you can:

- Bake the texture transformation into the texture bitmap file (so no transformations are done by the modeller)
- Use UV transformations on the mesh rather than on the texture

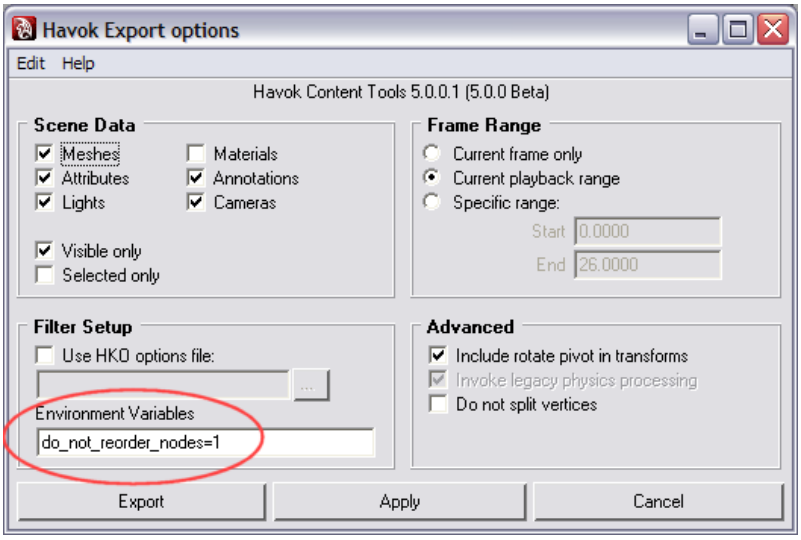
6.8.1.16 Order of nodes/bones doesn't match order in modeller

Previous to Havok 4.5, the order in which nodes were exported was taken from the order given by the modeler's export API/SDK. This was causing problems since, in many cases, this order was arbitrary and uncontrolable by the user.

Since Havok 4.5.0, the XSI, Maya and 3ds Max exporters will export nodes in alphabetical order. This has the advantage of given a predictable, deterministic ordering, as well as being controllable by the user (by changing node names).

The drawback of the change is that old (pre-4.5) assets and new (post 4.5) assets will have different ordering of nodes (and therefore, of bones in skeleton, etc). The recommended approach is to reexport the old assets to benefit from the new alphabetical ordering. However, if this is not possible or desirable, you can force the exporters to use the old ordering -that is, telling them not to reorder the nodes alphabetically - by defining an environment variable, "`do_not_reorder_nodes`" (set it to any value). This can be defined either in the Havok export options, as shown below, or as a Windows environment variable.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



6.8.1.17 Other problems processing assets

The View XML filter can be a powerful debugging tool - it will help you spot any problems with the contents of the scene and the results of each filter. Place it at different stages of your filter processing and examine the data at each stage.

Processing assets will also generate a processing log, where information about asset processing, as well as any warnings or errors found, is reported. Ensure that the contents of the processing log matches your expectations.

```
Setting up Filter Manager...
Setup Complete!

[ --- Configuration: Default --- ]
   *** No PlatformWriter filter found. No HKX data will be saved for this configuration. ***
   [ SceneTransform ]
      Processed all nodes in the scene.
      Processed 7 meshes.
      Processed 4 cameras.
   [ CreateRigidBodies ]
      Creating Mopp Code for pCylinder1 ...
      Created 7 rigid bodies.
   [ CreateRigidBodies ]
      Warning [0xabbaa5f0]: Some physics data found already, have you already created rigid bodies?
      Creating Mopp Code for pCylinder1 ...
      Created 7 rigid bodies.
   [ CreateConstraints ]
      Created 2 constraints.
   [ PreviewScene ]
Finished in 00:00:00
```

If you are still running into problems processing assets and the cause of the problem is unclear, use the Support Data snapshot utility to create a set of data files suitable for later analysis

- Open the problem asset and launch the filter manager in the usual way.
- Once the manager has launched set up the filters so the problem is reproduced.
- From the help menu choose **Support Data** > **Save** .
- You will be prompted for a filename and 2 files will be created with the extension `support.hkx` and `support.hko` respectively.
- Send these file to support for analysis.

- NOTE: Before sending the files you can check their validity using the stand alone previewer.
  - Launch the previewer and load the `support.hkx` file
  - Once the filter manager has loaded use the **File** > **Load Configuration** option to load the options from the `support.hko` file.

6.8.2 Troubleshooting: 3ds Max Tools

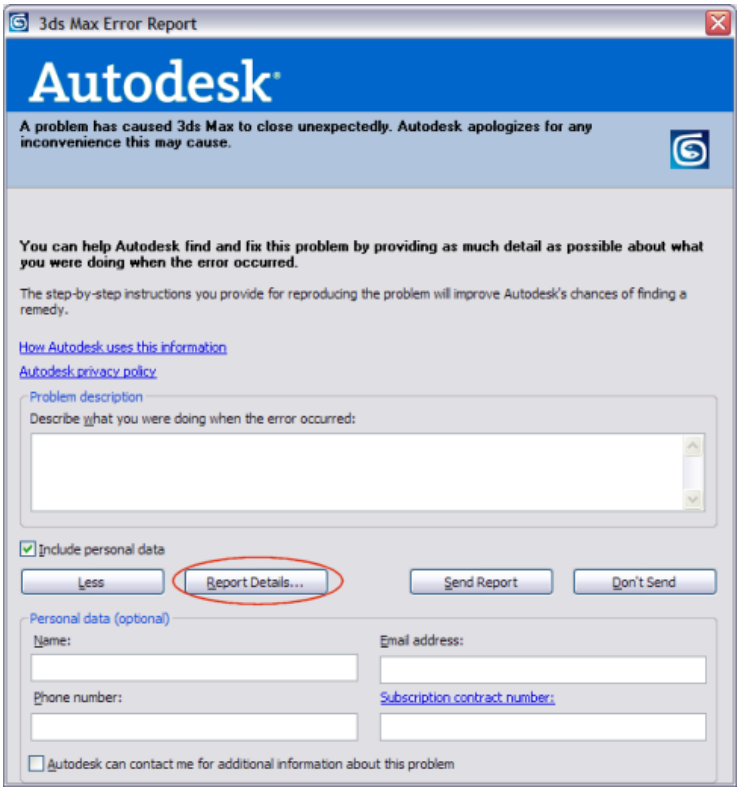
6.8.2.1 MAXScript Exceptions in general

When the Havok Content Tools are invoked through the **Havok Content Tools** menu or toolbar, they are called from MAXScript code. When a crash happens inside the tools, MAXScript reports an exception and tries to recover - most times, unsucessfully. Therefore, anytime you see a MAXScript exception raised when executing an operation with the Havok Content Tools (usually during export) the most likely cause is a crash.

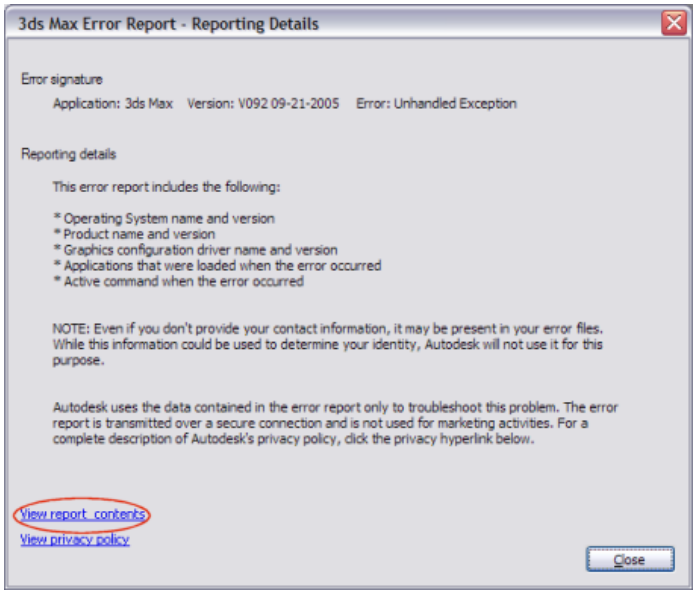
If you export by first opening the Scene Export Utility ( ), and then clicking on the **Export Scene** button, MAXScript is not used and the exception should be replaced by a full crash - which is more useful as it allows you to send us the crash information (see Reporting Crashes below)

6.8.2.2 Reporting Crashes

Should you experience crashes during export which are not solved by any of the items in this troubleshooting section you can send us the crash information reported by 3ds Max (if you are getting a MAXScript exception please check the section above in order to generate a full-blown crash). When the " **3ds Max Error Repor** t" window appears..

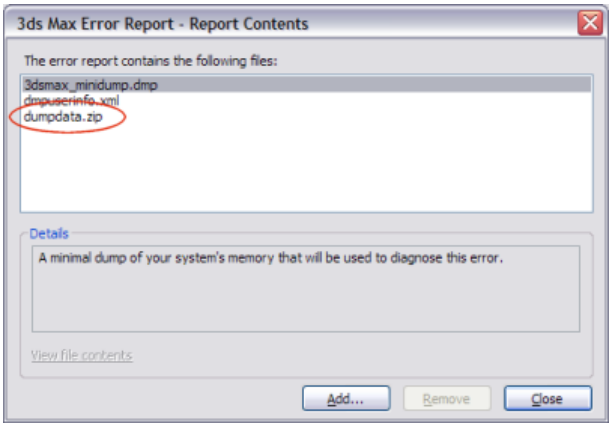


..instead of sending it to Autodesk, please click on the **Report Details..** button and then on the **View Report Contents** link.



This will list the files that would be sent to Autodesk. This includes `dumpdata.bin` (hovering the mouse over the file name will show its location in your hard drive).

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.



Please include this file when reporting the issue to Havok support (no need to send it to Autodesk), as well as the asset you were trying to process. This will help us reproduce your problems.

6.8.2.3 Can't find Havok's HKX format in the File > Export... dialog

The 3ds Max Scene Exporter is implemented as a 3ds Max Utility object and not as a file exporter (since it exports the scene to the filter pipeline rather than directly to a file) - so it no longer appears as a choice of format/extension in 3ds Max **File** > **Export...** dialog. Check the 3ds Max Scene Exporter section for details.

6.8.2.4 I can't make a rigid body from an object

You cannot create a rigid body from an object if:

- The object is already a rigid body
- The object doesn't have a mesh representation (dummies, curves or helpers)
- The object is a Character Studio bone - Use the Rag Doll Toolbox to create proxies instead.

6.8.2.5 Can't see the Havok Content Tools toolbar / menu

If cannot see neither the **Havok Content Tools** toolbar nor the menu, then the Havok Content Tools have not been properly installed for your version of 3ds Max. Try reinstalling them.

If you can't see the **Havok Content Tools** toolbar, but you can see the **Havok Content Tools** menu, click on the menu option **Show Toolbar** . If that still doesn't work, then the problem may be that the toolbar is placed outside the visible desktop area. Try docking it by executing the following MAXScript command:

```
hctToolbarGUP.dock()
```

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.8.2.6 Havok Content Tools menu is empty

3ds Max remembers the configuration of menus between sessions, so uninstalling the Havok Content Tools may leave an empty **Havok Content Tools** menu in 3ds Max. You can remove it through the **Customize** > **Customize User Interface...** menu option in 3ds Max.

6.8.3 Troubleshooting: Maya Tools

6.8.3.1 Errors of the form "Node Xxx has no attribute Xxx" on loading a scene

When moving from one version of the Havok Content Tools to another, some attributes of our custom node types may be added or removed. In the case that they are removed, any scenes which were saved with these attributes present may complain when loaded into a version where they are not present. Maya currently does not provide any way for plugins to deal with this scenario, and instead reports it as an error. By examining the script window, you can see which attributes are missing and determine wheter it is something to worry about.

Normally it is safe to ignore these errors. When the scene is resaved the errors will no longer appear.

6.8.3.2 Can't find Havok's HKX format in the File > Export... dialog

The Maya Scene Exporter is implemented as a Maya command and not as a file exporter (since it exports the scene to the filter pipeline rather than directly to a file) - so it no longer appears as a choice of format/extension Maya's **File** > **Export...** dialog. Check the Maya Scene Exporter section for details.

6.8.3.3 Can't see the Havok Content Tools menu or the Havok shelf

Make sure you have installed the Havok Content Tools for the version of Maya you are using, and the plug-ins have been properly loaded in Maya.

6.8.3.4 Havok Shelf is Empty

Maya remembers the configuration of shelves during sessions, so uninstalling the Havok Content Tools may leave an empty Havok shelf in Maya. You can remove it through the **Window** > **Settings/Preferences** > **Shelves...** menu option in Maya.

6.8.3.5 Can't see Havok rigid bodies and or constraints in the viewports

In order to see Havok content (rigid bodies, constraints, etc.) in the viewports, the Havok Physics Tool must be active. In addition, double-clicking on the tool icon in Maya's toolbox provides customisation options for this tool. It is possible to disable the display of rigid bodies and/or constraints on a group basis, so make sure that the content you require is not hidden in this way.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

Any Havok node which is directly selected (via the outliner for example) will always be displayed in the viewports, regardless of the tool settings.

6.8.3.6 Exported node transforms don't match the original transforms in Maya

By default the Maya scene exporter will override the node transforms in order to match the location of the rotate pivot point. This is required by the Maya Physics tools, so changing this option is not recommended if you are creating physics content. The " **Include Rotate Pivot in Transforms** " option can be found in the Maya Scene Exporter Options.

6.8.3.7 Shape Type defaults to Convex Hull although object is a sphere/box/capsule/cylinder

The Havok Maya tools will try to detect the type of a mesh by checking its construction history - for example, if a *"polySphere"* node is found, the shape type will be flagged as "Sphere" by default.

If the construction history is lost (for example, by using the **Duplicate** tool) then the Havok tools no longer have direct knowledge of the mesh type and will default to "*Hull*". Therefore you will need to manually set the **Shape Type** parameter for those objects.

For duplication, you can use the " **Duplicate input graph** " option with the **Duplicate Special** tool, instead of the **Duplicate** tool - this will keep the construction history of the duplicated node.

6.8.3.8 Can't see Havok rigid bodies and or constraints in the viewports

In order to see Havok content (rigid bodies, constraints, etc.) in the viewports, the Havok Physics Tool must be active. In addition, double-clicking on the tool icon in Maya's toolbox provides customisation options for this tool. It is possible to disable the display of rigid bodies and/or constraints on a group basis, so make sure that the content you require is not hidden in this way.

Any Havok node which is directly selected (via the outliner for example) will always be displayed in the viewports, regardless of the tool settings.

6.8.3.9 Exported node transforms don't match the original transforms in Maya

By default the Maya scene exporter will override the node transforms in order to match the location of the rotate pivot point. This is required by the Maya Physics tools, so changing this option is not recommended if you are creating physics content. The " **Include Rotate Pivot in Transforms** " option can be found in the Maya Scene Exporter Options.

6.8.3.10 Adding Havok nodes breaks UV Texture editor "UV Sets" menu

Maya gets confused if the first child shape of a transform node is not a mesh. A workaround is to select the mesh itself (rather than its parent transform node) before going into the UV editor.

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.

6.8.3.11 Adding Havok nodes breaks "Assign Material" context menu

This is due to a bug in Maya - Maya is checking for the number of renderable objects on the selected node, and is expecting to receive an int, but is instead receiving an int[] array, and thus failing to build the menu. This has been fixed in Maya 8.5.

6.8.4 Troubleshooting: XSI Tools

6.8.4.1 Physics preferences are not persisted across different scenes

This may occur if the XSI tools are installed into the 'factory' addons folder. A bug in XSI means that preferences saved inside this folder are not detected by XSI, and as such they cannot be persisted across different scenes. If this case is detected, any preferences are instead stored as a node in the scene, and a warning is given to the user. Any new scene will use the default preference values.

6.8.4.2 Rag Doll Toolbox disappears / gets replaced by white background

This is due to a bug in XSI. If a custom display host becomes hidden (by minimizing the window or the application, by switching to a diffferent tab in a custom relational view, etc.) then the custom display host may lose the value of the attribute which specifies the particular custom display to use. To retrieve the Rag Doll Toolbox, simply right click on the white background and select 'Havok Rag Doll Toolbox'.

There are also some redraw issues in XSI which may cause parts of the toolbox to be erased. Simply activate/move the toolbox to force a full redraw.

6.8.4.3 Exported node hierarchy doesn't match the original XSI hierarchy

By default the XSI exporter will reparent the exported nodes during export to ensure that chain end effectors are always children of the last link of the chain. This option, " **Reparent Chain Effectors** ", can changed in the XSI Scene Exporter Options.

6.8.4.4 Exported node transforms don't match the original transforms in XSI

By default the XSI exporter will override the translation component of the node transforms in order to match the location of the pivot point. This is required by the XSI Physics tools, so changing this option is not recommended if you are creating physics content. The " **Include Pivot in Transforms** " option can be found in the XSI Scene Exporter Options.

6.8.5 Troubleshooting: Toolchain Integration

6.8.5.1 Can't see my custom filter in the filter manager

- Make sure you've placed it in the right location

- Check the filter log window for any warnings raised by the filter manager
- Ensure your custom filter is built with the same sdk version as the filter manager
- Make sure you defined and exported the necessary DLL functions

Confidential Information of Havok. ©Copyright 1999-2008 Havok. All Rights Reserved.