



Lingo Reference Guide

Version 1.0

Copyright © 2000 / 2001 Havok.com Inc.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means whatsoever, including photocopying or recording without the prior written permission of Havok.com.

Published in Ireland.

The author makes no representation, express or implied, with regard to the accuracy of the information contained in this guide and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

The information contained in this manual is subject to change without notice.

Havok.com Inc.

www.havok.com

Havok.com and the Havok buzzsaw logo are registered trademarks of Havok.com.

Shockwave and Lingo are registered trademarks of Macromedia Inc.

All other brand names, product names, or trademarks belong to their respective holders.

Table of Contents

Havok Cast Member Lingo Property Reference	1
<i>havok.initialized</i>	1
<i>havok.tolerance</i>	1
<i>havok.scale</i>	1
<i>havok.timeStep</i>	2
<i>havok.subSteps</i>	2
<i>havok.simTime</i>	2
<i>havok.gravity</i>	2
<i>havok.rigidBody</i>	3
<i>havok.spring</i>	3
<i>havok.linearDashpot</i>	3
<i>havok.angularDashpot</i>	4
<i>havok.collisionList</i>	4
<i>havok.deactivationParameters</i>	4
<i>havok.dragParameters</i>	5
Havok Cast Member Lingo Function Reference	6
<i>havok.initialize()</i>	6
<i>havok.reset()</i>	7
<i>havok.step()</i>	7
<i>havok.shutdown()</i>	8
<i>havok.rigidBody()</i>	8
<i>havok.deleteRigidBody()</i>	9
<i>havok.makeMovableRigidBody()</i>	9
<i>havok.makeFixedRigidBody()</i>	10
<i>havok.registerInterest()</i>	10
<i>havok.removeInterest()</i>	12
<i>havok.disableCollision()</i>	12
<i>havok.enableCollision()</i>	12
<i>havok.disableAllCollisions()</i>	12
<i>havok.enableAllCollisions()</i>	12
<i>havok.registerStepCallback()</i>	13
<i>havok.removeStepCallback()</i>	13
<i>havok.spring()</i>	13
<i>havok.makeSpring()</i>	13
<i>havok.deleteSpring()</i>	14
<i>havok.linearDashpot()</i>	14
<i>havok.makeLinearDashpot()</i>	14
<i>havok.deleteLinearDashpot()</i>	15
<i>havok.angularDashpot()</i>	15
<i>havok.makeAngularDashpot()</i>	15
<i>havok.deleteAngularDashpot()</i>	16
Rigid Body Lingo Property Reference	17
<i>hkRigidBody.name</i>	17
<i>hkRigidBody.position</i>	17
<i>hkRigidBody.rotation</i>	18
<i>hkRigidBody.mass</i>	18
<i>hkRigidBody.restitution</i>	19

<i>hkRigidBody.friction</i>	19
<i>hkRigidBody.active</i>	20
<i>hkRigidBody.pinned</i>	20
<i>hkRigidBody.linearVelocity</i>	20
<i>hkRigidBody.angularVelocity</i>	21
<i>hkRigidBody.linearMomentum</i>	21
<i>hkRigidBody.angularMomentum</i>	22
<i>hkRigidBody.force</i>	22
<i>hkRigidBody.torque</i>	22
<i>hkRigidBody.centerOfMass</i>	23
<i>hkRigidBody.corrector.enabled</i>	23
<i>hkRigidBody.corrector.threshold</i>	23
<i>hkRigidBody.corrector.multiplier</i>	23
<i>hkRigidBody.corrector.level</i>	23
<i>hkRigidBody.corrector.maxTries</i>	24
<i>hkRigidBody.corrector.maxDistance</i>	24
Rigid Body Lingo Function Reference	25
<i>hkRigidBody.applyForce()</i>	25
<i>hkRigidBody.applyForceAtPoint()</i>	25
<i>hkRigidBody.applyImpulse()</i>	25
<i>hkRigidBody.applyImpulseAtPoint()</i>	26
<i>hkRigidBody.applyTorque()</i>	26
<i>hkRigidBody.applyAngularImpulse()</i>	26
<i>hkRigidBody.attemptMoveTo()</i>	27
<i>hkRigidBody.interpolatingMoveTo()</i>	27
<i>hkRigidBody.correctorMoveTo</i>	28
Spring Lingo Property Reference	29
<i>hkSpring.name</i>	29
<i>hkSpring.pointA</i>	29
<i>hkSpring.pointB</i>	29
<i>hkSpring.restLength</i>	30
<i>hkSpring.elasticity</i>	30
<i>hkSpring.damping</i>	30
<i>hkSpring.onCompression</i>	30
<i>hkSpring.onExtension</i>	31
Spring Lingo Function Reference	32
<i>hkSpring.setRigidBodyA</i>	32
<i>hkSpring.setRigidBodyB</i>	32
<i>hkSpring.getRigidBodyA</i>	32
<i>hkSpring.getRigidBodyB</i>	32
Linear Dashpot Lingo Property Reference	33
<i>hkLinearDashpot.name</i>	33
<i>hkLinearDashpot.pointA</i>	33
<i>hkLinearDashpot.pointB</i>	33
<i>hkLinearDashpot.strength</i>	34
<i>hkLinearDashpot.damping</i>	34
Linear Dashpot Lingo Function Reference	35

<i>hkLinearDashpot.setRigidBodyA</i>	35
<i>hkLinearDashpot.setRigidBodyB</i>	35
<i>hkLinearDashpot.getRigidBodyA</i>	35
<i>hkLinearDashpot.getRigidBodyB</i>	35
Angular Dashpot Lingo Property Reference	36
<i>hkAngularDashpot.name</i>	36
<i>hkAngularDashpot.rotation</i>	36
<i>hkAngularDashpot.strength</i>	36
<i>hkAngularDashpot.damping</i>	37
Linear Dashpot Lingo Function Reference	38
<i>hkAngularDashpot.setRigidBodyA</i>	38
<i>hkAngularDashpot.setRigidBodyB</i>	38
<i>hkAngularDashpot.getRigidBodyA</i>	38
<i>hkAngularDashpot.getRigidBodyB</i>	38

1 Havok Cast Member Lingo Property Reference

You can access the following properties through the Havok cast member. The **havok** term of each property description below indicates you must access them through a Havok Xtra cast member. It does not mean that the actual word **havok** is part of the syntax. In the example code the variable **havok** is an instance of the Havok cast member:

```
havok = member(havokCastMemberNumber) .
```

havok.initialized

Syntax havok.initialized

Access Get

Description

This property returns the current state of the physical simulation.

Example

The following fragment of Lingo checks the simulation state before either initializing or stepping it.

```
if not havok.initialized then
    havok.initialize( member("scene") )
else
    havok.step( 0.025, 5 )
end if
```

havok.tolerance

Syntax havok.tolerance

Access Get

Description

This property holds the simulation's initial collision tolerance. See **havok.initialize()** for more information about collision tolerances.

Example

The example Lingo below displays the current tolerance value.

```
put havok tolerance
-- 0.1
```

havok.scale

Syntax havok.scale

Access Get

Description

This property holds the current scaling factor for the simulation. See **havok.initialize()** for more information about simulation scale.

Example

The piece of Lingo below displays the current scaling factor.

```
put havok.scale
-- 0.0254
```

havok.timeStep

Syntax havok.timeStep

Access Get/Set

Description

This property holds the current time-step factor for the simulation. Time-step factor represents the amount of time that the physics simulation advances with each call to **havok.step()**. See **havok.step()** for more details.

havok.subSteps

Syntax havok.subSteps

Access Get/Set

Description

This property holds the current number of sub-steps used by Havok during each call to **havok.step()**. See **havok.step()** for more details.

havok.simTime

Syntax havok.simTime

Access Get

Description

This property holds the total physics time that has elapsed since the beginning of the Havok i.e. the total number of time steps * time step.

havok.gravity

Syntax havok.gravity

Access Get/Set

Description

This property holds the current force of gravity for the simulation. You specify gravity display units, so you need to be careful when setting it up. If using a scale factor of 1.0 (i.e. meters) then gravity should be (0, -9.81, 0) to act appropriately (assuming positive Y is up).

Example

The following Lingo example displays the current gravity before changing it.

```
put havok.gravity
-- vector( 0, 0, -386.22 )
```

```

havok.gravity = vector( 0, 0, -100 )
put havok.gravity
-- vector( 0, 0, -100.0 )

```

havok.rigidBody

Syntax havok.rigidBody

Access Get

Description

This property is a list of all rigid bodies in the simulation.

Example

The piece of Lingo below adds an anti-gravity force to all objects in the system.

```

repeat with i = 1 to havok.rigidbody.count
    havok.rigidBody[i].applyForce( -havok.gravity )
end repeat

```

havok.spring

Syntax havok.spring

Access Get

Description

This property is a list of all the springs in the simulation

Example

The piece of Lingo below sets the rest length of all the springs in the simulation to be 10.

```

repeat with i = 1 to havok.spring.count
    havok.spring[i].restLength = 10
end repeat

```

havok.linearDashpot

Syntax havok.linearDashpot

Access Get

Description

This property shows a list of all the linear dashpots in the simulation.

Example

The piece of Lingo below sets the damping of all the linear dashpots in the simulation to be 0.5.

```

repeat with i = 1 to havok.linearDashpot.count
    havok.linearDashpot[i].damping = 0.5
end repeat

```


havok.angularDashpot**Syntax** havok.angularDashpot**Access** Get**Description**

This property shows a list of all the angular dashpots in the simulation.

Example

The piece of Lingo below sets the damping of all the angular dashpots in the simulation to be 0.5.

```
repeat with i = 1 to havok.linearDashpot.count
    havok.angularDashpot[i].damping = 0.5
end repeat
```

havok.collisionList**Syntax** havok.collisionList**Access** Get**Description**

This property returns the current collision list. This list is made up from zero or more sub-lists containing individual collision information. This information includes the names of the colliding bodies, the world position of the contact point and the contact normal.

Example

This following piece of Lingo displays the current list of collisions within a physical simulation.

```
put havok.collisionList
-- [{"BallWhite", "DisplayFelt", 83.6288, 2.1487, 15.0180, 0.0000,
0.0000, 1.0000}]
```

havok.deactivationParameters**Syntax** havok.deactivationParameters**Access** Get/Set**Description**

This property is a list of two frequencies that simulations use to deactivate low-energy objects. Simulations check objects at regular intervals to decide whether or not they should be deactivated. To make deactivation more aggressive, raise the frequencies. To make it less aggressive, lower them. A deactivated object is removed from the physical simulation and therefore takes no CPU time. It is still involved in collision testing, but purely in case objects hit it and so reactivate it.

The property's two frequencies are short- and long-range deactivation parameters. In both cases, they refer to a time period during which the behaviors of all simulated objects are monitored. The short-range frequency selection specifies a time period

during which Havok attempts to deactivate objects that move by very small amounts or not at all, and is typically 1/20th of a second. Sometimes objects are effectively at rest and should be deactivated, but due to the current time step or numerical error, they jitter. In such cases, the short-range non-aggressive deactivator fails, as the objects are moving too much to be considered inactive. The long-range test is more aggressive but acts over a longer time period (typically 10 seconds).

To turn off deactivation, either long-range or short-range, set the appropriate parameter to 0. To make either the short or long range deactivation more aggressive increase the frequency value (a value of 60Hz, when simulating with a frame rate of 60Hz, is the most aggressive possible).

Example

The piece of Lingo below displays the current frequencies used for deactivations. The default short frequency is 2 Hz (1/2 of a second period). The long frequency default is 0.1 Hz (10 second period).

```
put havok.deactivationParameters
-- [2.0000, 0.1000]
```

havok.dragParameters

Syntax havok.dragParameters

Access Get/Set

Description This property is a list which contains the linear drag coefficient and the angular drag coefficient respectively. The drag force is applied to oppose the motion of a rigid body and is applied equally to all bodies in a havok simulation. At high values the drag can nearly instantaneously oppose all motion. A reasonable value for these is [0.1, 0.1].

2 Havok Cast Member Lingo Function Reference

You can access the following functions through the Havok cast member. The **havok** syntax guidelines from the previous chapter apply here.

havok.initialize()

Syntax havok.initialize(W3DMember)
 havok.initialize(W3DMember, tolerance, worldScale)

Description

You can create physical information for the Havok simulation in two ways:

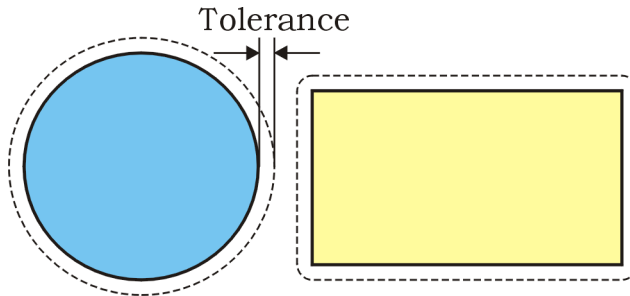
The first method for creating physical simulation information is through a modeling tool. The modeling tool that you use must support exporting **.hke** files. You can import this **.hke** file as a movie cast member using the **File > Import** menu options. **.hke** files already contain world scaling information and tolerance (as specified within the 3D modeler), so you do not have to supply this value when initializing.

The second method for creating physical information is directly from the models from within a 3D scene. In this case, you must create a blank Havok cast member using the **Insert > Media Element > Havok Physics Scene** menu option. It is very important that you establish the scale of the physics scene from the start. Internally, the Havok physics simulation employs the metric system (i.e. default unit is meters). A W3D cast member may have been created in any number of world units (meters, inches, feet, user, generic). The Havok Xtra interface can work with the same units as this W3D cast member. However, in order to perform the proper simulation, Havok Xtra must know the correspondence between the display (3D scene) units and the simulation units.

You must provide a world-scaling factor when initializing the physical simulation. For example, if you designed a scene using inches, then you would supply a scaling value of 0.0254 (1 inch = 0.0254 meter). Be aware that any values in the scene (like gravity, rest length of springs, etc.) are interpreted as scene units rather than internal physics units. That means that a real-world gravity value of 9.81 meters/second² would have to be set as 386.22 inches/sec² if working in inches.

You must also provide a collision tolerance parameter. This tolerance is used to determine when objects are touching (i.e. if they are closer than the tolerance). In general, higher collision tolerance values yield more stable simulations. However, setting too high a value could lead to noticeable gaps between stacked objects. So, it is recommended that you set the collision tolerance to the highest value at which it does not visually affect the scene.

For example, if a scene consists of many objects in a room (crates, tables, chairs, etc.) a tolerance of around 0.1m should be fine. However, if the objects in the scene are dice on a table a smaller tolerance, say 0.01m or less, is preferable. If the objects are cars or buildings, a higher tolerance applies, etc. If no value is supplied then the default tolerance of 0.1 is used. As a general rule of thumb, set the tolerance value close to 10% of the scaling factor used in the simulation.



Example

The world scale in the following example is set to 0.0254 as the scene was constructed in inches i.e. 1 meter * 0.0254 = 1 inch and the collision tolerance is set to 4 inches.

```
havok.initialize(member("MyScene"), 4.0, 0.0254)
```

Note

Collision tolerance is a value measured in scene units. That means that the scaling factor affects its actual value. **havok.initialize()** must be the first Havok function called or other Havok functions will have no effect.

havok.reset()

Syntax havok.reset()

Description

This function resets the current physical simulation to its initial state. This is only really appropriate for physical simulations initialized from a .hke file where a reset() call reverts the entire scene back to the state defined in the .hke file. A physical simulation not using an imported .hke file has an initial state that contains no rigid bodies.

havok.step()

Syntax havok.step()
 havok.step(TimeIncrement)
 havok.step(TimeIncrement, NumOfSubsteps)

Description

This function steps the physical simulation by the time increment and uses the specified number of sub-steps to super sample and split that interval into smaller units. This function is usually called each frame to advance the physics simulation by some small time period. To pause the simulation, simply refrain from calling **step()**. To achieve approximate real-time performance you should step the simulation according to the frame rate of the movie. For example, for a Director tempo of 60 fps you should step the world 0.0167 seconds each frame (= 1.0 / 60).

Example

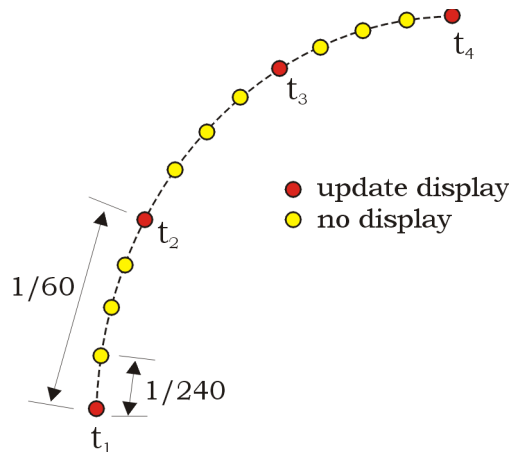
This steps the simulation by 0.0167 seconds (i.e. 60 updates per second) with 4 internal sub-steps.

```
havok.step(0.0167, 4)
```

Note

The frame rate specified in Director is not necessarily the actual frame rate at which the movie plays. It depends on how long it takes Director to render the movie. To achieve true real-time performance, you need to keep track of elapsed absolute time.

The number of sub-steps gives display-independent control over the accuracy of the simulation. You should always try to step the simulation with **NumOfSubSteps = 1** because it is the fastest. Sometimes numerical instability results, for example with large stacks. Increasing the number of sub-steps causes the simulation to make a number of passes over the simulation for each call to **step()**. This gives more accurate results but at the cost of additional CPU overhead.

**havok.shutdown()**

Syntax `havok.shutdown()`

Description

This function stops the current simulation and removes it from memory.

Note

Be careful not to confuse this with Director's **shutdown()** function, which attempts to shut down your computer.

havok.rigidBody()

Syntax `havok.rigidBody(RBName)`

Description

This function queries the physical simulation for a rigid body of a given name. If it finds the rigid body, it returns a reference for it. You can use this to alter properties and call functions on rigid bodies. (See below)

Example

The piece of Lingo below looks for a rigid body and then sets its position to the origin of the world.

```
rb = havok.rigidBody( model.name )
rb.position = Vector( 0.0, 0.0, 0.0 )
```

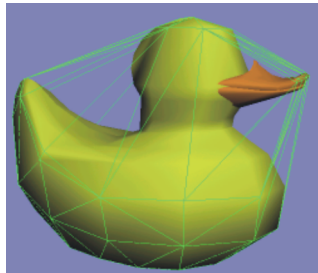
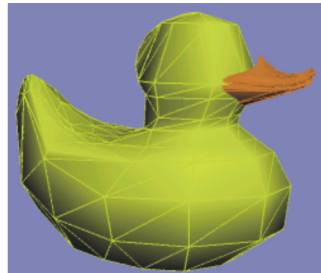
havok.deleteRigidBody()

Syntax havok.deleteRigidBody(RBName or RBIndex)

Description

This function removes a rigid body from the physics simulation given the rigid bodies name or index.

```
havok.deleteRigidBody( "WhiteBall" )
```

**Convex****Concave****havok.makeMovableRigidBody()**

Syntax havok.makeMovableRigidBody(modelName, mass)
 havok.makeMovableRigidBody(modelName, mass, isConvex)
 havok.makeMovableRigidBody(modelName, mass, true, type)

Description

This function creates a movable rigid body, with specified mass greater than zero (specified in kilograms), from a model of name **modelName** and adds it to the simulation. The optional Boolean flag **isConvex** indicates whether the new rigid body is to be convex or concave, where the default is convex. Furthermore if you specify the **type** parameter to be convex (i.e. true), you can then construct a bounding sphere (**#sphere**) or axis-aligned box (**#box**) rather than the default convex hull.

It is easier and faster to use convex geometries to resolve collisions, so you should use them wherever possible. A convex body is one where any line from its inside to the outside world results can only cross the object's boundary once. Convex objects cannot have holes, or hollows or loops like a teapot's handle. Concave geometries have no geometric restrictions, but their collision resolution is much more complex, and slower as a result.

Example

The code below first creates a movable rigid body, 1kg in mass, from the object called "Whiteball" and uses a convex hull representation by default. The second line also

creates a moveable rigid body of 1kg mass but uses a bounding sphere for the physical representation.

```
havok.makeMovableRigidBody( "WhiteBall", 1 )
havok.makeMovableRigidBody( "WhiteBall", 1, true, #sphere )
```

Note

When you are creating a rigid body from a model, you must add the **meshdeform** modifier to the model, i.e. **model.addModifier(#meshDeform)**. Otherwise, Havok Xtra cannot access the geometry of the model.

The convex representation of a rigid body is called a convex hull. When you create a convex hull for a new rigid body, the resulting mesh is heavily dependent on the original mesh of the object. In particular, Havok does handle lots of co-planar polygons easily, and this is common in 3D elements like extruded text.

The resulting convex hull may have many badly formed triangles that seriously degrade performance, and in some rare cases cause failure of the collision detection. In these cases, you are often better off using the actually geometry itself by creating the rigid body as concave. Alternatively, specifically in the case of extruded text, you could use a bounding box.

havok.makeFixedRigidBody()

Syntax

```
havok.makeFixedRigidBody(modelName)
havok.makeFixedRigidBody(modelName, isConvex)
havok.makeFixedRigidBody(modelName, true, type)
```

Description

This function creates a fixed rigid body from a model of name **modelName** and adds it to the simulation. The optional Boolean flag **isConvex** indicates whether the new rigid body is to be convex or concave (see **havok.makeMovableRigidBody**). The default value is convex. Fixed rigid bodies never move, but are still involved in collision detection. These are mostly used for scenery elements like walls.

Note

When you create a rigid body from a model, you must add the **meshdeform** modifier to it, as in **model.addModifier(#meshDeform)**. If you don't add the **meshdeform** modifier, then Havok Xtra cannot access the geometry of the model.

Fixed bodies do not have mass. Mass is a property that only makes sense for objects which are free to move.

```
havok.makeFixedRigidBody( "PoolTable", false )
```

havok.regisiterInterest()

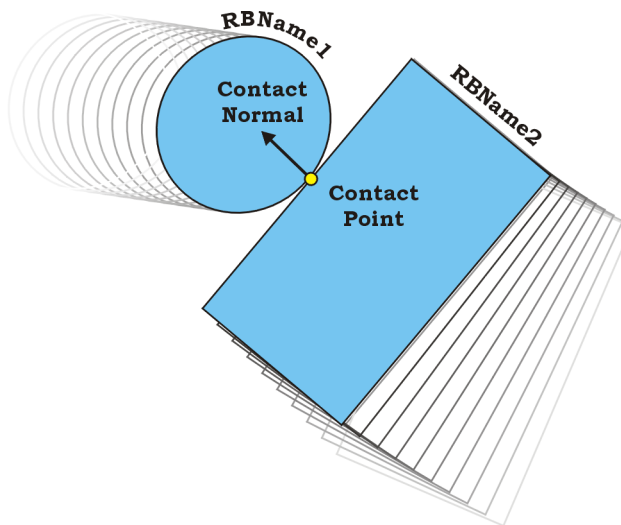
Syntax

```
havok.registerInterest(RBName1, RBName2, Frequency, Threshold)
havok.registerInterest(RBName1, RBName2, Frequency, Threshold, \
#LingoHandler, scriptIntance)
```

Description

This function allows detecting of specific collisions between rigid bodies and passing

details of the collision a specified Lingo callback handler. The following information is passed to a callback handler in the form of a list:



- **RigidBody1**: Name of first object involved in collision
- **RigidBody2**: Name of second object involved in collision
- **Contact Point**: Point of collision between the two objects
- **Contact Normal**: Collision direction equals the normal of the second object at the point of collision
- **Normal Relative Velocity**: The relative velocity of the two objects involved in the collision. This value is the sum of the absolute value of the objects' velocities in the direction of the collision normal. So, for two spheres, each traveling at 10 m/s directly towards the other, the NRV is 20 m/s.

If you do not specify a collision handler, then collision information is simply added to a collision list, which you can access using the **havok.collisionList** function at any time. If the type **#all** is passed in for **RigidBody2** then any collision involving **RigidBody1** initiates a callback. **Frequency** determines how often a collision is recorded. Applications that require callbacks for every collision should set a frequency of zero. The value that you set for frequency determines the number callbacks per second. For example, when you set frequency at ten, you only get ten events raised every second, invoking the callback a maximum of 10 times each second. **Threshold** specifies how strong a collision must be before the Lingo callback is invoked. Threshold is defined in terms of the normal relative velocity (i.e. meters per second), which is the relative speed of the objects in collision. Positive values indicate the objects are heading towards each other.

Example

The following Lingo fragment registers interest in any collisions involving the rigid bodies named **rb1** and **rb2**. When this collision occurs, the Lingo handler **collisionHandler** displays the collision point in the message window.

```
havok.registerInterest( rb1, rb2, 0, 0, #collisionHandler, me )

on collisionHandler(me, collisionDetails)
    put collisionDetails
end
```

To register interest only in collisions involving **rb1** and **rb2** where they are collide at

a relative velocity of greater than 10 m/s, use:

```
havok.registerInterest( rb1, rb2, 0, 10, #collisionHandler, me )
```

Note

You must provide a rigid body name for **RBName1**.

havok.removeInterest()

Syntax havok.removeInterest(RBName)

Description

This function stops collisions involving the specified rigid body being recorded.

```
havok.removeInterest( rb.name )
```

havok.disableCollision()

Syntax havok.disableCollision(RBNameA, RBNameB)

Description

This function disables any collision between two rigid bodies identified by their names.

havok.enableCollision()

Syntax havok.enableCollision(RBNameA, RBNameB)

Description

This function re-enables any collision between two rigid bodies identified by their names.

havok.disableAllCollisions()

Syntax havok.disableAllCollisions(RBNameA)

Description

This function disables any collisions between a rigid body of a given name and the other objects in the physics simulation.

havok.enableAllCollisions()

Syntax havok.enableAllCollisions(RBNameA)

Description This function re-enables all collisions between a rigid body of a given name and the other objects in the simulation.

havok.registerStepCallback()

Syntax `havok.registerStepCallback(#stepHandler, scriptInstance)`

Description

During each physics simulation step the Havok engine may take a number of sub steps (specified by the **havok.subSteps** property). This function allows the users to register a callback to a lingo handler that will get called at each sub step passing the length of time since the last sub step. This allows behaviors to be written that are called after each step of the physics engine (which is important for behaviors involving real-world parameters).

Example

The following Lingo fragment registers the step callback to the lingo handler **stepHandler**.

```
havok.registerStepCallback(#stepHandler, me )

on stepHandler(me, timeStep)
    put "I've been called"
end stepHandler
```

havok.removeStepCallback()

Syntax `havok.removeStepCallback(#stepHandler, scriptInstance)`

Description

This function removes the callback to the given handler and it's script instance.

havok.spring()

Syntax `havok.spring(SpringName)`

Description

This function queries the physical simulation for a spring of a given name. If it finds the spring, it returns a reference for it. You can then use this for altering properties and calling functions on springs (see below).

Example

The piece of Lingo below looks for a spring and then sets its rest length to 5.

```
spring = havok.spring( "TheSpring" )
spring.restLength = 5
```

havok.makeSpring()

Syntax `havok.makeSpring(SpringName, RBNameA, RBNameB)`

`havok.makeSpring(SpringName, RBName, WorldPoint)`

Description

A spring is an object with a preferred rest length which is attached to a pair of objects. When the spring is stretched or squashed it attempts to restore equilibrium by apply-

ing a restoring force to the attached objects.

The first version of this function makes a spring between the centers of mass of two named rigid bodies.

The second version of this function makes a spring between a rigid body given its name (**RBName**), and a world point.

Example

This piece of Lingo creates a spring between the center of mass of two rigid bodies.

```
spring = havok.makeSpring("MySpring", "Box1", "Box2")
```

havok.deleteSpring()

Syntax havok.deleteSpring(SpringName)
 havok.deleteSpring(SpringIndex)

Description

This function removes a spring of a given name or index from the physics simulation.

```
havok.deleteSpring( "TheSpring" )
```

havok.linearDashpot()

Syntax havok.linearDashpot(LinearDashpotName)

Description

This function queries the physical simulation for a linear dashpot of a given name. If it finds the linear dashpot it returns a reference for it. You can then use this to alter properties and call functions on linear dashpots (see below).

Example

The piece of Lingo below looks for an angular dashpot and then sets its damping to 0.5.

```
dashpot = havok.linearDashpot( "TheLinearDashpot" )  
dashpot.damping = 0.5
```

havok.makeLinearDashpot()

Syntax havok.makeLinearDashpot(DashName, RBNameA, RBNameB)
 havok.makeLinearDashpot(DashName, RBName, WorldPoint)

Description

A linear dashpot is a heavily damped zero length spring. It applies forces to objects when the velocities of their attached points begin to differ. Dashpots can be made stiffer than regular springs because velocities are taken into account. In addition you can use a dashpot to attach a point on a body to a fixed point in world space.

The first version of this function makes a linear dashpot between the centers of mass of two named rigid bodies.

The second version of this function makes a linear dashpot between a rigid body given a rigid body's name (**RBName**), and a point in world space where the other end of the dashpot is attached (**WorldPoint**).

Example

This piece of Lingo creates a linear dashpot between the centers of mass of two rigid bodies.

```
dashpot = havok.makeLinearDashpot("MyDash", "Box1", "Box2")
```

havok.deleteLinearDashpot()

Syntax havok.deleteLinearDashpot(DashpotName)
 havok.deleteLinearDashpot(DashpotIndex)

Description

This function removes a linear dashpot of a given name or index from the physics simulation.

```
havok.deleteLinearDashpot( "TheDashpot" )
```

havok.angularDashpot()

Syntax havok.angularDashpot(AngularDashpotName)

Description

This function queries the physical simulation for an angular dashpot of a given name. If it finds the angular dashpot it returns a reference for it. You can use this to alter properties and call functions on angular dashpots (see below).

Example

The piece of Lingo below looks for an angular dashpot and then sets its damping to 0.5.

```
angularDashpot = havok.angularDashpot( "TheAngularDashpot")
angularDashpot.damping = 0.5
```

havok.makeAngularDashpot()

Syntax havok.makeAngularDashpot(DashName, RBNameA, RBNameB)
 havok.makeAngularDashpot(DashName, RBName)

Description

An angular dashpot is the rotation equivalent of a linear dashpot. An angular dashpot tries to align two objects so that they have the same orientation. If the objects' orientations differ, forces are applied to both bodies that push them closer to the same orientation. You can also use an angular dashpot to align a single body to an orientation in world space.

The first version of this function makes an angular dashpot between two rigid bodies given their names (**RBNameA**, **RBNameB**). The initial rotation is the zero angle.

The second version of this function makes an angular dashpot between a rigidbody (**RBName**) and the reference frame. The initial rotation is the zero angle.

Example

The piece of Lingo creates an angular dashpot between the two named rigid bodies.

```
angDashpot = havok.makeAngularDashpot("My Dash", "Box1", "Box2")
```

havok.deleteAngularDashpot()

Syntax havok.deleteAngularDashpot(AngularDashpotName)
 havok.deleteAngularDashpot(AngularDashpotIndex)

Description

This function removes an angular dashpot of a given name or index from the physics simulation.

```
havok.deleteAngularDashpot( "TheAngularDashpot" )
```

3 Rigid Body Lingo Property Reference

The following properties can be accessed through a Havok Rigid Body that can be obtained from a Havok cast member using the function:

```
havok.rigidBody(RBName) or havok.rigidBody[i]
```

The word **hkRigidBody** has been added in front of each property description in order to indicate that access must be made through a **hkRigidBody**. This notation is equivalent to the use of **havok** in previous chapters. In the example code the variable **rb** is an instance of a **hkRigidBody**:

```
rb = member(havokCastMemberNumber).rigidBody("Box01")
```

RigidBody() can take either a string name or an index / number to identify the cast member.

hkRigidBody.name

Syntax `hkRigidBody.name`

Access `Get`

Description

This property returns the name of a rigid body. In general this equates to the rigid body's display equivalent in the 3D scene.

Example

The following fragment of Lingo displays the name of a rigid body in the message window.

```
put rb.name
-- "Box01"
```

hkRigidBody.position

Syntax `hkRigidBody.position`

Access `Get/Set`

Description

This property sets or gets the position of a rigid body. Position is in the form of a Director vector object.

Example

The following fragment of Lingo sets the position of a rigid body to position (2.0, 3.0, 4.0) and then displays the position in the message window.

```
rb.position = vector(2.0, 3.0, 4.0)
put rb.position
-- vector( 2.0000, 3.0000, 4.0000 )
```

Note

If you place a rigid body in a position so that it interpenetrates another rigid body, collisions between these two rigid bodies are not resolved. See **hkRigidBody.attempt-**

MoveTo() for further information.

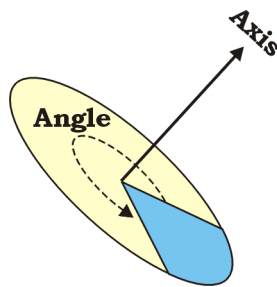
hkRigidBody.rotation

Syntax hkRigidBody.rotation

Access Get/Set

Description

You may use this property to set or get the rotation of a rigid body. Rotation is in the form of a Director list containing a vector object indicating rotation axis and a real number indicating rotation angle. Rotations use the right hand rule (i.e. point the thumb of your right hand in the direction of the axis of rotation and the fingers curl in the direction of the rotation). Angles are specified in degrees.



Example

The following fragment of Lingo script gets the rotation of a rigid body and displays it in the message window.

```
put rb.rotation
-- [vector( 1.0000, 0.0000, 0.0000 ), 90.0000]
```

Note

If you place a rigid body in an orientation such that it interpenetrates another rigid body then collisions between these two rigid bodies are not resolved. See **hkRigidBody.attemptMoveTo()** for further information.

hkRigidBody.mass

Syntax hkRigidBody.mass

Access Get/Set

Description

You can use this property to set or get the mass of a rigid body (specified in kg.).

Example

The following fragment of Lingo script displays the mass of a rigid body in the message window.

```
put rb.mass
-- 1.0000
```

Note

Only movable rigid bodies can have mass. If a fixed rigid body needs to move, then create it as a movable rigid body and lock its position (see **hkRigidBody.pinned** property below)

hkRigidBody.restitution

Syntax `hkRigidBody.restitution`

Access Get/Set

Description

You can use this property to set or get the restitution or bounciness of a rigid body. Restitution relates to an object's energy loss or gain after a collision. If an object has a restitution value of zero, then all energy is lost on collision and it does not bounce. A restitution value of 1 gives a perfect bounce. A restitution value greater than 1 means a bouncing object gains energy after each collision. So, a bouncing ball would reach a higher height after each time it hits the floor.

Example

The following fragment of Lingo script sets the restitution of a rigid body.

```
rb.restitution = 0 -- no bounce
rb.restitution = 1 -- perfect bounce
rb.restitution = 0.5 -- lose 50% of energy after each bounce
```

hkRigidBody.friction

Syntax `hkRigidBody.friction`

Access Get/Set

Description

You can use this to set or get the coefficient of friction or stickiness of a rigid body (0 = no friction, typical values are 0.8...1.0). Friction relates to how much force is required to move or roll one rigid body over another. If a sliding object has a friction value of zero then it never stops sliding. In reality there are two forms of friction: dynamic and static. Objects moving relative to each other (i.e. sliding) use dynamic friction. Objects at rest and stacked are held in place by static friction. The transition from static to dynamic friction is crucial for the realism of a physics simulation and a difficult problem to solve in general.

For example, when you strike a pool ball with a cue, it initially slides over the surface of the pool table and slows due to dynamic friction. Eventually it catches and begins to roll. Static friction takes over and maintains the contact between the ball and table, converting forward momentum into torque. This causes the ball to spin or roll. The ball eventually comes to rest due to energy loss resulting from the static friction. Havok Xtra requires only a single value of friction to be specified, but internally fully simulates both static and dynamic friction behaviors and the transition between them.

Example

The following fragment of Lingo script sets the friction of a rigid body.


```
rb.friction = 0 -- slide forever
```

Note

For objects in contact you need to specify both friction parameters to get the desired result. The friction used is the square root of the sum of the squares of the friction coefficients.

hkRigidBody.active

Syntax hkRigidBody.active

Access Get/Set

Description

You may use this property to set or get whether a movable rigid body is active. A deactivated object never moves until struck by another object or because of a force that acts upon it.

Example

The following piece of Lingo deactivates an object if it falls outside a specified distance from the world origin.

```
if rb.position.length > 10000 then
    rb.active = false
end if
```

hkRigidBody.pinned

Syntax hkRigidBody.pinned

Access Get/Set

Description

You can use this property to set or get whether a movable rigid body is pinned in place. A pinned object never moves when struck by another object, but may be released under Lingo control at a later point in the simulation. This is unlike an object initially created as fixed.

Example

The following piece of Lingo fixes a movable rigid body if currently movable.

```
if not rb.pinned then
    rb.pinned = true -- fix body
end if
```

hkRigidBody.linearVelocity

Syntax hkRigidBody.linearVelocity

Access Get/Set

Description

You may use this property to set or get the linear velocity of a rigid body. A Lingo vector specifies the value. Linear velocity is simply the speed of the object. The magni-

tude of the velocity vector is the actual speed of the body and the vector specifies the direction in which the object is moving.

Example

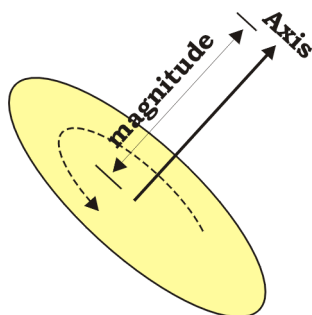
The following fragment of Lingo script prints the linear velocity of a rigid body, which in this case is moving at 10 units per second in the positive Z direction.

```
put rb.linearVelocity
-- vector( 0.0000, 0.0000, 10.0000 )
```

hkRigidBody.angularVelocity

Syntax `hkRigidBody.angularVelocity`

Access Get/Set



Description

You may use this property to set or get the angular velocity of a rigid body. A Lingo vector specifies the value of the velocity. The magnitude of the vector determines the speed in degrees per second CCW about the specified axis. The normalized vector, or the unit length version of the vector, gives its axis. So an angular velocity of (2.0, 0.0, 0.0) corresponds to a rotation speed of 2.0 degrees per second counter-clockwise around the positive X axis.

Example

The following fragment of Lingo script prints the angular velocity of a rigid body.

```
put rb.angularVelocity
-- vector( 10.0000, 0.0000, 0.0000 )
```

hkRigidBody.linearMomentum

Syntax `hkRigidBody.linearMomentum`

Access Get/Set

Description

You may use this property to set or get the linear momentum of a rigid body. A Lingo vector specifies the value. Momentum is mass multiplied by velocity.

Example

The following piece of script sets prints the linear momentum of a rigid body.

```
rb.linearMomentum = vector( 1, 0, 0 ) * rb.mass
```

hkRigidBody.angularMomentum**Syntax** `hkRigidBody.angularMomentum`**Access** `Get/Set`**Description**

You may use this property to set or get the angular momentum of a rigid body. A Lingo vector specifies the value.

Example

The following fragment of Lingo script sets the angular momentum of a rigid body.

```
rb.angularMomentum = vector( 1, 0, 0 ) * rb.mass
```

hkRigidBody.force**Syntax** `hkRigidBody.force`**Access** `Get`**Description**

You may use this property to get the current total force acting on a rigid body. The total force acting on a body depends on the forces applied through Lingo script and also the forces applied by the Havok system during simulation.

Example

The following fragment of Lingo script puts the current force on a rigid body to the message window.

```
put rb.force
-- vector( 15.0000, 0.0000, -9.8100 )
```

hkRigidBody.torque**Syntax** `hkRigidBody.torque`**Access** `Get`**Description**

You may use this property to get the current torque on a rigid body. Torque is the angular analog of force. You should apply a torque to induce a spin in an object. Torque, like angular velocity, is a single vector. The vector's magnitude is that of the exerted torque. The normalized vector specifies the axis about which the torque exerts.

Example

The following fragment of Lingo script displays the current torque exerted on a rigid body in the message window.

```
put rb.torque
-- vector( 0.0000, 10.0000, 0.0000 )
```

hkRigidBody.centerOfMass**Syntax** `hkRigidBody.centerOfMass`**Access** `Get`**Description**

This property gives the user the offset from the models origin to the rigid body's center of mass.

hkRigidBody.corrector.enabled**Syntax** `hkRigidBody.corrector.enabled`**Access** `Get/Set`**Description**

Enables or disables the corrector for a particular rigid body. It can be set to true or false. For more details see the function **hkRigidBody.correctorMoveTo**.

hkRigidBody.corrector.threshold**Syntax** `hkRigidBody.corrector.threshold`**Access** `Get/Set`**Description**

This property is used to determine how close the body is gotten to the final destination before any reckoning occurs (see **hkRigidBody.corrector.level** for the different reckoning levels) . This is measured in design units. The Havok simulation will set this to a default parameter depending on the scene. The user may wish to change it. A reasonable threshold is anywhere from 0.01 to 0.7 depending on the design of the scene. If the threshold is too small the corrector may go into an infinite loop i.e. cannot get to the exact position. If the threshold is too large the corrector may have little or no effect on the body.

hkRigidBody.corrector.multiplier**Syntax** `hkRigidBody.corrector.multiplier`**Access** `Get/Set`**Description**

This property is used to determine how fast a rigid body will move to its destination using **hkRigidBody.correctorMoveTo**. A multiplier of value 1.0 gives 1 unit of time to get there. With a very high multiplier, the objects may hit each other with too much force. The default for this parameter is 5.

hkRigidBody.corrector.level**Syntax** `hkRigidBody.corrector.level`

Access Get/Set

Description

This property sets the reckoning level of the corrector. The levels are:

- 0 - Hold - (never stop correcting) (HOLD)
- 1 - No dead reckoning (leave go with vel=0)
- 2 - Dead Reckoning (set velocities)
- 3 - 2nd order reckoning (set accelerations)

hkRigidBody.corrector.maxTries

Syntax hkRigidBody.corrector.maxTries

Access Get/Set

Description

The property is the number of attempts made to move the rigid body to its final destination during a **hkRigidBody.correctorMoveTo** (if blocked).

hkRigidBody.corrector.maxDistance

Syntax hkRigidBody.corrector.maxDistance

Access Get/Set

Description

If the difference between the current position of the body and the desired position of the body is greater than the maxDistance property then the corrector will simply move the rigid body to it's desired position.

4 Rigid Body Lingo Function Reference

You can access the following properties through a Havok rigid body, which you can obtain from a Havok cast member, using the function:

```
havok.rigidBody(RBName)
or
havok.rigidBody[RNum]
```

As with the entries in chapter 3, each entry in this chapter is preceded by an **hkRigidBody** tag to indicate method of access.

```
rb = member(havokCastMemberNumber).rigidBody("Box01")
```

hkRigidBody.applyForce()

Syntax `hkRigidBody.applyForce(force)`

Description

This function applies a force to a rigid body at its center of mass. A Lingo vector specifies the value of this force. An example application of applying a force would be applying the brakes in a car, which takes time to have an effect. Applying a force at the center of mass does not affect the spin of the object.

Example

The follow fragment of Lingo script applies an anti-gravity force to a rigid body.

```
grav = havok.gravity()
rb.applyForce( -grav )
```

hkRigidBody.applyForceAtPoint()

Syntax `hkRigidBody.applyForceAtPoint(force, point)`

Description

This function applies a force to a rigid body at a specified point in model space. Lingo vectors specify the value of the force and the position. Forces applied at points other than the center of mass of an object induce a torque effect, causing the object to spin.

Note

The model space point does not have to be on or contained within the object. It works as though a lever connects the specified point to the object's center of mass and the force applies to the end of the lever.

hkRigidBody.applyImpulse()

Syntax `hkRigidBody.applyImpulse(impulse)`

Description

This function applies an impulse to a rigid body at its center of mass. A Lingo vector specifies the value of the impulse. An example stopping impulse would be like a car hitting a wall, whereby the car stops immediately. An impulse, unlike a force, has an immediate effect on the velocity of the rigid body so gives a greater degree of control

over the object.

Example

The follow fragment of Lingo script applies an impulse to a rigid body straight up in the air.

```
rb.applyImpulse(vector(0.0, 0.0, 100.0))
```

hkRigidBody.applyImpulseAtPoint()

Syntax `hkRigidBody.applyImpulseAtPoint(impulse, point)`

Description

This function applies an impulse to a rigid body at a specified point relative to the position of the model. Lingo vectors specify the value of the impulse and the position. Similar to applying forces at a point, applying an impulse at a point other than the center of mass of an object induces a torque effect, causing that object to spin.

Example

The follow fragment of Lingo script applies an impulse to a rigid body in it's up direction. As the impulse is offset from the object's center of mass, the body acquires an angular velocity.

```
rb.applyImpulseAtPoint(Vector(0.0, 0.0, 100.0), Vector(0, 0, 5))
```

Note

The follow fragment of Lingo script applies an impulse to a rigid body in it's up direction. As the impulse is offset from the object's center of mass, the body acquires an angular velocity.

hkRigidBody.applyTorque()

Syntax `hkRigidBody.applyTorque(torque)`

Description

This function applies torque to a rigid body at its center of mass. A Lingo vector specifies the value of the torque. The magnitude of the vector determines the size of the torque. The normalized vector gives the axis about which the torque is applied.

Example

The follow fragment of Lingo script applies a torque of magnitude ten times the mass of the rigid body inducing a CCW rotation about the positive X axis. This effectively works like a motor action to the object along the its local X-axis.

```
rb = havok.rigidBody("FrontLeftWheel")
rb.applyImpulse(Vector(10.0, 0.0, 0.0) * rb.mass)
```

hkRigidBody.applyAngularImpulse()

Syntax `hkRigidBody.applyAngularImpulse(impulse)`

Description

This function applies an angular impulse to a rigid body at its center of mass. A Lingo vector specifies the value of the impulse. The magnitude of the vector determines the size of the impulse. The normalized vector gives the axis about which the impulse is

applied. An angular impulse has an immediate effect on the angular velocity of the object in a similar way that an impulse has an immediate effect on the linear velocity of an object.

Example

The follow fragment of Lingo script applies an angular impulse of magnitude 100 to a rigid body. This applies a twist about the object's local Z-axis.

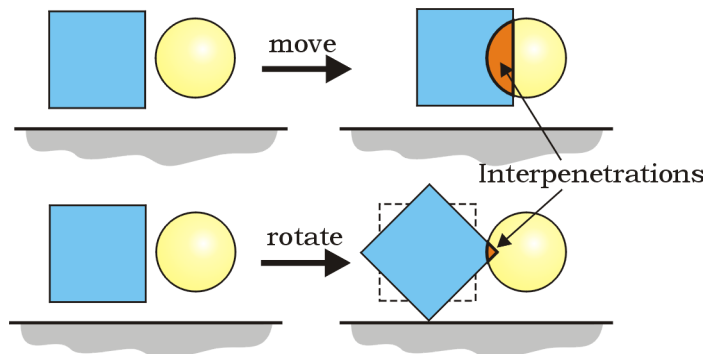
```
rb.applyImpulse(vector(0.0, 0.0, 100.0))
```

hkRigidBody.attemptMoveTo()

Syntax `hkRigidBody.attemptMoveTo(position, rotation)`

Description

This function takes a position (vector) and a rotation (in the form of a list containing an axis and an angle, expressed as a vector and a floating-point value respectively). The function attempts to move the rigid body to a position and orientation specified by the three parameters. If the rigid body is in an acceptable physical state when repositioned, such as not interpenetrating, it is left there and the function returns true. Otherwise the function returns false. In both cases below, the function returns false:



Example

The follow fragment of Lingo script attempts to position a rigid body at world coordinates (0, 0, 100) with its initial orientation.

```
m = rb.attemptMoveTo(vector(0,0,100), [vector(0,1,0), 0])
if not m then
    put "Move Failed"
end if
```

hkRigidBody.interpolatingMoveTo()

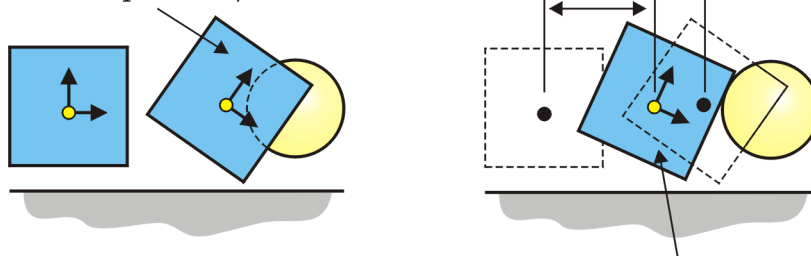
Syntax `hkRigidBody.interpolatingMoveTo(position, rotation)`

Description

This function takes a position (vector) and a rotation (passed in the form of a list containing an axis and an angle, expressed as a vector and a floating-point value respectively). The function attempts to move the rigid body to a position and orientation

specified by the three parameters. If the rigid body is not in an acceptable physical state when repositioned, such as interpenetrating another object, it is moved back from the specified position to the first valid point along a direct path from its initial position to the specified position.

desired new position / orientation



interpolatingMoveTo gets 70% there

This returns a floating-point value between 0 and 1. 0 indicates that the rigid body cannot move from its initial spot. A value of 1 indicates that the object has been successfully repositioned where specified. A value of 0.5 means the object has been repositioned to exactly halfway point between its initial and desired location. This value also applies to the orientation, both axis and angle.

Example

The follow fragment of Lingo script attempts to position a rigid body at world coordinates (0, 0, 100) with its initial orientation. It then prints the result.

```
newPos = vector(0, 0, 100)
newRot = [vector(0, 1, 0), 0]
d = rb.interpolatingMoveTo(newPos, newRot)
put "Got " & (d * 100) & "% of the way there"
```

hkRigidBody.correctorMoveTo

Syntax

hkRigidBody.correctorMoveTo(position, rotation)

hkRigidBody.correctorMoveTo(position, rotation, linearVelocity, \ angularVelocity)

hkRigidBody.correctorMoveTo(position, rotation, linearVelocity, \ angularVelocity, linearAcceleration, angularAcceleration)

Description

A corrector receives a desired state for a body, via the this function. The corrector will then attempt to bring the body to that desired state. This corrector uses impulses to get a body to a desired position. If the body's current position is greater than the max distance property of the corrector away from the desired position the body will be then moved to the desired position. The first function sets the position and rotation of the body. The second version of this function sets position, rotation, linear velocity and angular velocity. The third version sets position, rotation, linear velocity, angular velocity, linear acceleration and angular acceleration.

5 Spring Lingo Property Reference

You can access the following properties through a Havok spring, which you can obtain from a Havok cast member using the function:

```
havok.spring(SpringName) or havok.spring[i]
```

The **hkSpring** of each property description below indicates your means of access to them. It does not mean that the actual word **hkSpring** is part of the syntax. In the example code the variable **spring** is an instance of an **hkSpring**:

```
spring = member(havokCastMemberNumber).spring("Spring01")
```

hkSpring.name

Syntax `hkSpring.name`

Access `Get`

Description

This property gets the name of the spring.

hkSpring.pointA

Syntax `hkSpring.pointA`

Access `Get/Set`

Description

This property contains the position on rigid body A to which the spring is attached. The position is relative to rigid body A.

Example

This piece of Lingo attaches the spring to the origin of rigid body A.

```
newPos = vector(0, 0, 0)
spring.pointA = newPos
```

hkSpring.pointB

Syntax `hkSpring.pointB`

Access `Get/Set`

Description

This property contains the position on rigid body A to which the spring is attached. The position is relative to rigid body B. If there is no rigid body B, then this point is a point in world space to which the spring is attached.

Example

This piece of Lingo attaches the spring to the origin of rigid body B.

```
newPos = vector(0, 0, 0)
spring.pointB = newPos
```

hkSpring.restLength**Syntax** `hkSpring.restLength`**Access** Get/Set**Description**

This property contains the rest length of the spring.

Example

This piece of Lingo sets the rest length of the spring to be 10.

```
spring.restLength = 10
```

hkSpring.elasticity**Syntax** `hkSpring.elasticity`**Access** Get/Set**Description**

This property contains the elasticity of the spring. The higher the elasticity the stronger the spring.

Example

This piece of Lingo sets the elasticity of the spring to be 0.5.

```
spring.elasticity = 0.5
```

hkSpring.damping**Syntax** `hkSpring.damping`**Access** Get/Set**Description**

This property contains the damping value for the spring. Higher damping values cause the spring to come to rest fast. If specified too high however, this can lead to numerically unstable solutions.

Example

This piece of Lingo sets the damping of the spring to be 0.5.

```
spring.damping = 0.5
```

hkSpring.onCompression**Syntax** `hkSpring.onCompression`**Access** Get/Set**Description**

If this property is set to **true**, the spring applies a restoring force when compressed (i.e. distance between pointA and pointB is less than the restLength). If the property is **false**, it does not.

hkSpring.onExtension

Syntax hkSpring.onExtension

Access Get/set

Description

If this property is set to **true**, the spring applies a restoring force when extended (i.e. distance between pointA and pointB is greater than the restLength). If the property is **false**, it does not.

6 Spring Lingo Function Reference

You can access the following functions through a Havok spring, which you can obtain from a Havok cast member using the function:

```
havok.spring(SpringName) or havok.spring[i]
```

The **hkSpring** of each function description below indicates your means of access to them. It does not mean that the actual word **hkSpring** is part of the syntax. In the example code the variable **spring** is an instance of an **hkSpring**:

```
spring = member(havokCastMemberNumber).spring("Spring01")
```

hkSpring.setRigidBodyA

Syntax `hkSpring.setRigidBodyA(RBName)`

Description

This function sets the rigid body connected to the first end of the spring (pointA by convention).

hkSpring.setRigidBodyB

Syntax `hkSpring.setRigidBodyB(RBName)`

`hkSpring.setRigidBodyB("none")`

Description

This function sets the rigid body connected to the second end of the spring (**pointB** by convention). If you pass **#none** as a parameter here the spring will be attached to a world point, specified by the spring property **pointB** (see above).

hkSpring.getRigidBodyA

Syntax `hkSpring.getRigidBodyA(RBName)`

Description

This function returns the name of the rigid body connected to the first end of the spring. If it is not connected to any rigid body it will return **#none**.

hkSpring.getRigidBodyB

Syntax `hkSpring.getRigidBodyB(RBName)`

Description

This function returns the name of the rigid body connected to the second end of the spring. If it is not connected to any rigid body it will return **#none**.

7 Linear Dashpot Lingo Property Reference

You can access the following properties through a Havok linear dashpot, which you can obtain from a Havok cast member using the function:

```
havok.linearDashpot(LinearDashpotName) or havok.linearDashpot[i]
```

The **hkLinearDashpot** of each property description below indicates your means of access to them. It does not mean that the actual word **hkLinearDashpot** is part of the syntax. In the example code the variable **linearDashpot** is an instance of an **hkLinearDashpot**:

```
linearDashpot =  
member(havokCastMemberNumber).linearDashpot("LinearDashpot")
```

hkLinearDashpot.name

Syntax hkLinearDashpot.name

Access Get

Description

This property contains the name of the linear dashpot.

hkLinearDashpot.pointA

Syntax hkLinearDashpot.pointA

Access Get/Set

Description

This property contains the position on rigid body A to which the linear dashpot is attached. The position is relative to rigid body A

Example

This piece of Lingo attaches the linear dashpot to the center of mass of rigid body A.

```
newPos = vector(0, 0, 0)  
linearDashpot.pointA = newPos
```

hkLinearDashpot.pointB

Syntax hkLinearDashpot.pointB

Access Get/Set

Description

If the linear dashpot is attached to another rigid body, this property contains the position on rigid body B to which the linear dashpot is attached. The position is relative to rigid body B. If the linear dashpot is attached to a point in world space, this property contains a position relative to the origin of the scene to which the dashpot is attached.

Example

This piece of Lingo attaches the linear dashpot to the center of mass of rigid body B.

```
newPos = vector(0, 0, 0)
linearDashpot.pointB = newPos
```

hkLinearDashpot.strength

Syntax hkLinearDashpot.strength

Access Get/Set

Description

This property contains the strength of the linear dashpot and controls how quickly the stable state for the dashpot is achieved. High strength values yield very stiff dashpots, which can lead to unstable results. A good initial range is 0.5 - 1.

Example

This piece of Lingo sets the strength of the linear dashpot to be 10.

```
linearDashpot.strength = 10
```

hkLinearDashpot.damping

Syntax hkLinearDashpot.damping

Access Get/Set

Description

This property specifies the damping factor for the linear dashpot. The damping factor controls how quickly the dashpot comes to rest. Very high damping factors can yield unstable results. A good initial value is 0.1.

Example

This piece of Lingo sets the damping of the linear dashpot to be 0.5.

```
linearDashpot.damping = 0.5
```

8 Linear Dashpot Lingo Function Reference

You can access the following functions through a Havok linear dashpot, which you can obtain from a Havok cast member using the function:

```
havok.linearDashpot(LinearDashpotName) or havok.linearDashpot[i]
```

The **hkLinearDashpot** of each function description below indicates your means of access to them. It does not mean that the actual word **hkLinearDashpot** is part of the syntax. In the example code the variable **linearDashpot** is an instance of an **hkLinearDashpot**:

```
linearDashpot =  
member(havokCastMemberNumber).linearDashpot("LinearDashpot")
```

hkLinearDashpot.setRigidBodyA

Syntax `hkLinearDashpot.setRigidBodyA(RBName)`

Description

This function sets the rigid body connected to the first end of the linear dashpot (pointA by convention).

hkLinearDashpot.setRigidBodyB

Syntax `hkLinearDashpot.setRigidBodyB(RBName)`
 `hkLinearDashpot.setRigidBodyB("none")`

Description

This function sets the rigid body connected to one end of the linear dashpot. If you pass **#none** as a parameter here the linear dashpot will be attached to a world point, which is specified in the linearDashpot property **pointB** (see above).

hkLinearDashpot.getRigidBodyA

Syntax `hkLinearDashpot.getRigidBodyB(RBName)`

Description

This function returns the name of the rigid body connected to the first end of the linear dashpot. If it is not connected to any rigid body it will return **#none**.

hkLinearDashpot.getRigidBodyB

Syntax `hkLinearDashpot.getRigidBodyB(RBName)`

Description

This function returns the name of the rigid body connected to the second end of the linear dashpot. If it is not connected to any rigid body it will return **#none**.

9 Angular Dashpot Lingo Property Reference

You can access the following properties through a Havok angular dashpot, which you can obtain from a Havok cast member using the function:

```
havok.angularDashpot (AngDashpotName)
or
havok.angularDashpot [i]
```

The **hkAngularDashpot** of each property description below indicates your means of access to them. It does not mean that the actual word **hkAngularDashpot** is part of the syntax. In the example code the variable **angDashpot** is an instance of an **hkAngularDashpot**:

```
angDashpot =
member (havokCastMemberNumber) .AngularDashpot ("AngDashpot")
```

hkAngularDashpot.name

Syntax hkAngularDashpot.name

Access Get

Description

This property contains the name of the linear dashpot.

hkAngularDashpot.rotation

Syntax hkAngularDashpot.rotation

Access Get/Set

Description

This property contains the angle of rotation that the angular dashpot attempts to maintain between another rigid body or world space.

Example

This piece of Lingo sets the rotation to be zero.

```
newAngle = [vector(1, 1, 1), 0]
angDashpot.rotation = newAngle
```

hkAngularDashpot.strength

Syntax hkAngularDashpot.strength

Access Get/Set

Description

This property contains the strength of the angular dashpot and controls how quickly the stable state for the dashpot is achieved. High strength values yield very stiff dashpots, which can lead to unstable results. A good initial range is 0.5 - 1.

Example

This piece of Lingo sets the strength of the angular dashpot to be 10.

```
angDashpot.strength = 10
```

hkAngularDashpot.damping

Syntax hkAngularDashpot.damping

Access Get/Set

Description

This property specifies the damping factor for the angular dashpot. The damping factor controls how quickly the dashpot comes to rest. Very high damping factors can yield unstable results. A good initial value is 0.1.

Example

This piece of Lingo sets the damping of the angular dashpot to be 0.5.

```
angDashpot.damping = 0.5
```

10 Linear Dashpot Lingo Function Reference

You can access the following properties through a Havok angular dashpot, which you can obtain from a Havok cast member using the function:

```
havok.angularDashpot (AngDashpotName)
or
havok.angularDashpot [i]
```

The **hkAngularDashpot** of each property description below indicates your means of access to them. It does not mean that the actual word **hkAngularDashpot** is part of the syntax. In the example code the variable **angDashpot** is an instance of an **hkAngularDashpot**:

```
angularDashpot =
member (havokCastMemberNumber) .AngularDashpot ("AngularDashpot")
```

hkAngularDashpot.setRigidBodyA

Syntax `hkAngularDashpot.setRigidBodyA(RBName)`

Description

This function sets the rigid body connected to one end of the angular dashpot.

hkAngularDashpot.setRigidBodyB

Syntax `hkAngularDashpot.setRigidBodyB(RBName or "none")`

Description

This function sets the rigid body connected to one end of the angular dashpot. If you pass **#none** as a parameter here the angular dashpot will be attached to a world point, which is specified in the angularDashpot property **rotation** (see above).

hkAngularDashpot.getRigidBodyA

Syntax `hkAngularDashpot.getRigidBodyA(RBName)`

Description

This function returns the name of the rigid body connected to the first end of the angular dashpot. If it is not connected to any rigid body it will return **#none**.

hkAngularDashpot.getRigidBodyB

Syntax `hkAngularDashpot.getRigidBodyB(RBName)`

Description

This function returns the name of the rigid body connected to the second end of the angular dashpot. If it is not connected to any rigid body it will return **#none**.