Control Freaks

Mixing game play, physics and 3D graphics in Lingo

This tutorial outlines how to combine some simple game play, the Havok Xtra plugin and the Shockwave 3D plugin, to produce a car-driving demo. Each step from scene construction onward is covered in detail. Once it's all over you should have a running version of a simple car demo that you can use as a base to build a more interesting car game.

Contents

© 2000 Havok.com Inc.

1 Overview:

This tutorial produces a simple car driving behavior. It is reasonably realistic, but most of all it is fun to drive. The emphasis is on making the demo fun rather than making it physically accurate.

Here's a checklist of the tools you'll need:

1. 3ds max: For 3D scene construction
2. Shockwave 3D Exporter: To export the 3D scene
3. MAX Havok plugin: To export the physical scene
4. Director: To put it all together
5. Havok Xtra: To control the dynamics in Director
6. Havok Behaviors: To link the physics and the visuals
7. Tutorial Files: These include the driving behavior and the models

Make sure you have each of these tools installed. The MAX Havok plugin and the Havok Xtra are available from the Havok Website http://www.havok.com.

The most important (and difficult) task when working with physics is to mix the game play and realism in just the right amounts. It's often the case that physically realistic behavior is not exactly what you want. Instead it's often hyper real behavior, like power-slides and cornering at 150mph that adds all the fun. The bottom line is that the more control both the game designer and the users have the better.

A Lingo behavior controls the car driving in this example. It's designed specifically with game play in mind. It's relatively straightforward and simple to tweak. It comes with a range of parameters that let you adjust anything from top speed to type grip. The tutorial covers it in detail later.

© 2000 Havok.com Inc.

2 Scene Construction

Begin by creating a simple scene in 3ds max. This scene can be found in simplecar.max.

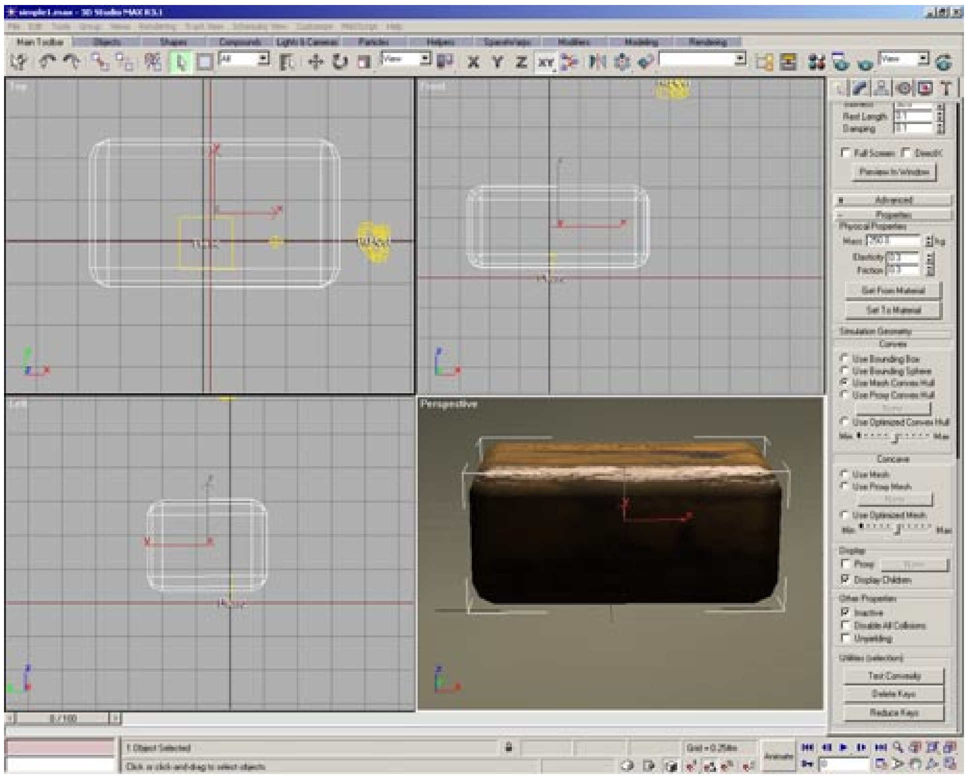


Figure 1 A simple physical scene

- Create a chamfered box.
- Rename the box to something that we can easily identify later – in this example it's called `chassis00`
- Add a plane to the scene to act as a floor.
- Add a Rigid Body Collection (RBCollection) to the scene. This can be found in the helpers rollout under Havok Dynamics.
- Add the chamfered box and the plane to the Rigid Collection
- Select the chamfered box and open the Havok Dynamics Utility Rollout.
- Change the box properties and give it some mass – in this case we use 250kgs
- Select the plane and assign the 'Concave – Use Mesh' property.

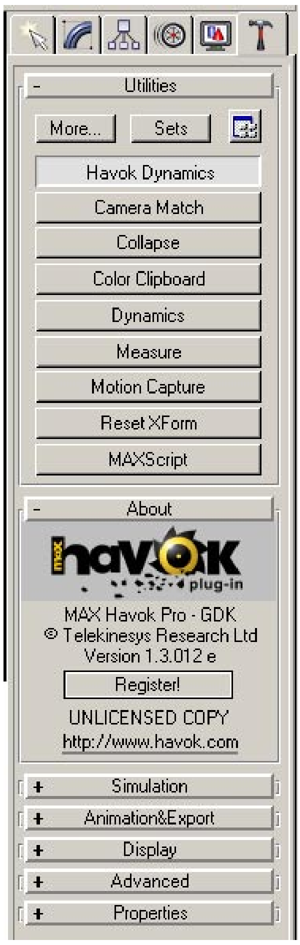
© 2000 Havok.com Inc.

- Preview the physical scene to ensure it works. Click on Preview in window in the Havok Dynamics Utility Rollout. Use the right mouse button to pick up the box and throw it about.

- Before constructing the scene in Director, export both the physical and graphical representations of the scene.

- To export the physical representation, go to the Animation and Export rollout in the Havok Dynamics Utility. Click on export to file. You don't need to export display information (the checkbox should be unchecked) and it is more efficient to export the file in binary format.

- To export the 3D graphical representation, choose Export from the file menu and save the Shockwave 3D file.



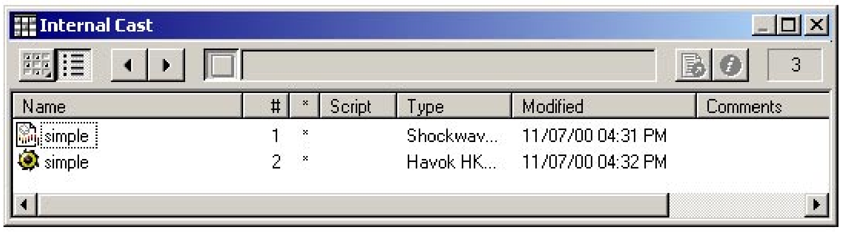
The Havok Dynamics Utility Rollout

© 2000 Havok.com Inc.

3 Adding Havok Dynamics in Director

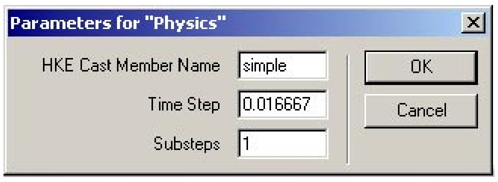
The next step is to put all the pieces together in Director

- Launch Director.
- Import the Shockwave 3D file as a cast member and drag it onto the stage.
- Import the Havok HKE file as a cast member.



Name	#	*	Script	Type	Modified	Comments
simple	1	*		Shockwav...	11/07/00 04:31 PM	
simple	2	*		Havok HK...	11/07/00 04:32 PM	

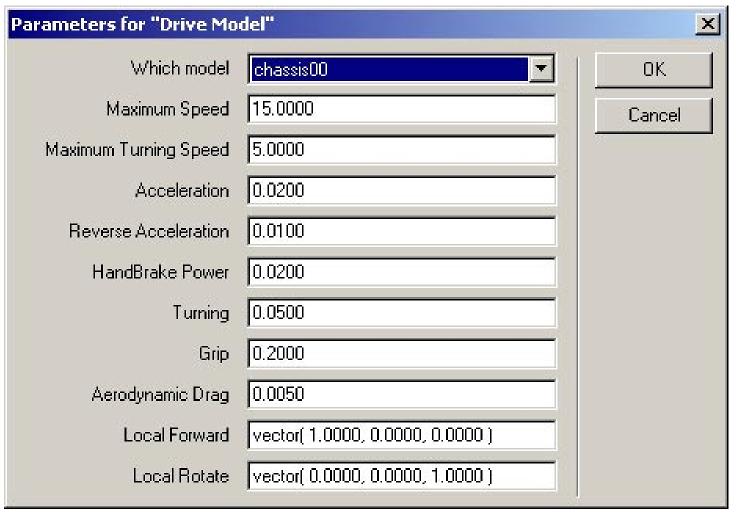
- Next we need to link the physical and graphical representations of the scene. To do this we add The Havok Physics behavior to the 3D sprite
- Open the library palette and choose Havok > Setup from the list. Drag and Drop the Physics behavior onto the 3D sprite on stage. The following dialog will appear:



HKE Cast Member Name	simple
Time Step	0.016667
Substeps	1

- The "Physics" behavior links a simulation to the 3D sprite. The Havok member to be used can be selected from the dropdown list. The time step specifies how far forward the physical world is advanced on each animation frame. This demo should run at 60fps so a time step of 1/60 (0.016667) is used. The substeps parameter indicates how many internal steps Havok will split the time step into. This can be used to balance the CPU load between physics and display.
- Set the tempo of the animation to 60fps
- Now the physical world is linked to display. When you play, the animation physical simulation is performed. You won't see much at this stage because the box is resting on the plane.
- As we will be driving our box around we will want to have the camera follow it. Open the External cast that comes with this tutorial (demo.cst) and drag and drop the Track Model behavior onto the 3D sprite. To keep the camera stationary set the "Num Steps to Source Position" to zero.

- Finally, drag and drop the Drive Model behavior onto the 3D Sprite and the following dialog will appear.



Which model	chassis00
Maximum Speed	15.0000
Maximum Turning Speed	5.0000
Acceleration	0.0200
Reverse Acceleration	0.0100
HandBrake Power	0.0200
Turning	0.0500
Grip	0.2000
Aerodynamic Drag	0.0050
Local Forward	vector(1.0000, 0.0000, 0.0000)
Local Rotate	vector(0.0000, 0.0000, 1.0000)

- All the default parameters should be correct for the current scene. A full description of each parameter is included in the Lingo behavior.

- Click OK and play the animation.

- Use the arrow keys to drive forward, reverse and turn. Press space to use the handbrake!

- Now is a good time to tweak the values for the behavior and get a feel for the kind of effects the model can produce. You can find a pre-built version of this demo in simplecar.dir.

© 2000 Havok.com Inc.

4 Smoke and Mirrors

Now that you have a simple block driving around a plane it's time to add a few more effects to the demo. Create a new max scene and add a few more objects to it. Don't forget to include a representation for the chassis. There's one pre-built in `car.max`.

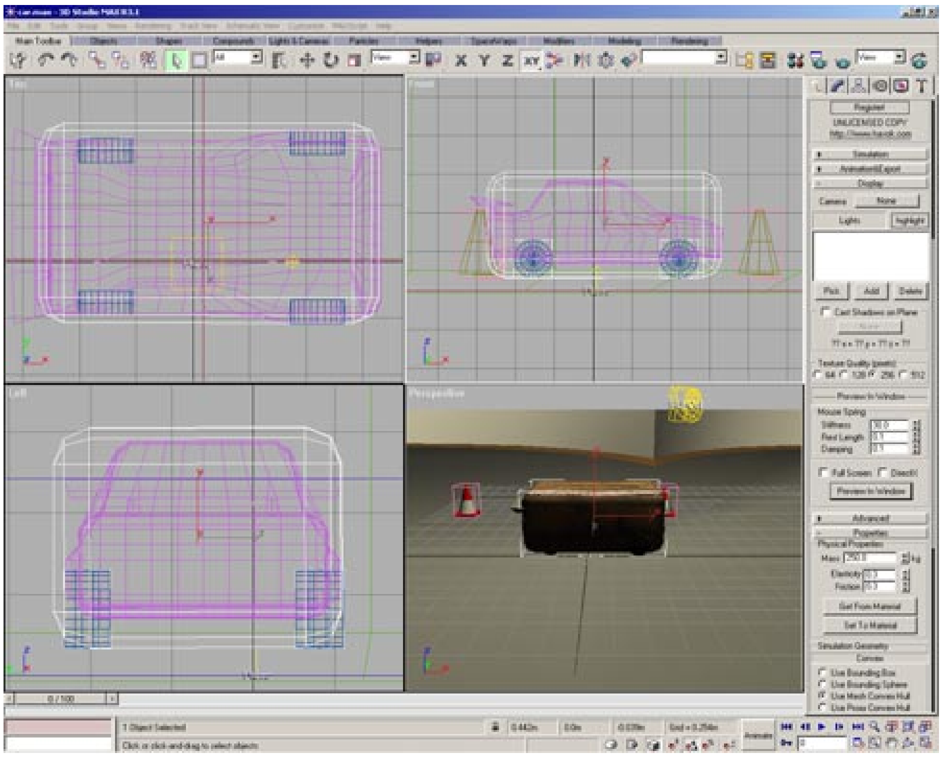


Figure 2 A more complex physical scene

- The scene contains our original chamfered box, a few traffic cones, a ramp and some curved walls.
- You can also see a car embedded in the middle of the chamfered box! This car is displayed instead of the box, giving the illusion that a car is being driven, when in fact only a box is being simulated. The actual car model could have been used from the start, but often you will want to use a simpler or more controllable physical representation for a graphical object.
- If you click on the chamfered box and go to the Properties rollout in the Havok Dynamics Utility you will see that the box uses a display proxy.
- When you click on preview in window it appears as if the actual car is being simulated. In fact the original box is simulated and you can

© 2000 Havok.com Inc.

choose to display “Sim Edges” (the edges of the simulated objects) from the geometry window to see what’s going on.

- As before you need to save two versions of the scene. One physical and one graphical.

- Export the Havok HKE file as before using the Animation and Export rollout. Again there is no need to export display information and choose binary format.

- At present the Havok Xtra does not use any HKE display information. The simple Physics behavior, that links a physical and graphical scene, can’t be told that a more complex car model should be displayed instead of the box. There is a simple way around this.

- Just before you export the Shockwave 3D file, delete the chamfered box and rename the body of the car to `chassis00`. The Havok Physics Behavior current binds physical information to display information according to the names associated with each. By renaming the car body it will now be associated with the chamfered box. See the proxy.dir demo for more a example of this process.

- Once exported launch Director and add in the dynamics as before.

- You can attach the Track Model behavior (included in demo.cst) to the chassis and the camera will automatically follow the car.

© 2000 Havok.com Inc.

5 The Drive Model Behavior

All the real work for this demo is done inside the Drive Model behavior. This section strips it down and shows exactly how it was constructed. The script attempts to reproduce a few of the most commonly experienced vehicle dynamics effects.

- A car should be able to accelerate / decelerate and turn.
- A car has a top speed and a maximum turning circle.
- Cars should be able to brake and when they brake they skid, at least in most games!
- A car can move more easily backward and forward then left and right i.e. it should only skid or slide sideways in extreme circumstances.
- All cars experience drag.

We treat each of these effects individually and let the Havok physics engine take care of combining them all into a realistic driving behavior.

The `on enterFrame` handler controls the behavior at a high level.

```
on enterFrame(me)

  -- Check the keys and set the state variables
  me.getKeys()

  -- Apply the controller to drive the vehicle
  me.driveController()

end enterFrame
```

It calls two simple tasks.

1. It polls the keyboard to see which keys are held down. This sets some script properties which are used later to determine if the player is accelerating / braking etc.
2. It applies the heuristics above to the physical car model to make it drive – in fact this is where all the work is. The rest of the script is just the standard Lingo.

© 2000 Havok.com Inc.

The `driveController` handler begins by working out some simple geometric information. It needs to work out the direction the car is currently heading and the axis it rotates about when it turns. This information is originally stored when the car is created (it's in local space) and must be transformed to world space.

```
  -- Transform forward and axis vectors into world space
  trans = pMember.model(pModel).transform.duplicate()
  trans.position = Vector(0,0,0)
  trans.scale = Vector(1,1,1)

  currentFwd  = trans * pForward
  currentAxis = trans * pTurnAxis
  currentRight = currentFwd.cross(currentAxis)
```

The current transform of the model being driven is duplicated and the position and scaling information reset. Now the transform only represents a rotation from local space to world space and is used to work out the direction the car is currently heading, the axis it should rotate about and, for convenience, the left/right orientation of the car.

The Havok Xtra is queried to return some physical information about the vehicle.

```
  rb = pHavok.rigidBody( pModel )

  -- Current Velocity
  currentVel   = rb.linearVelocity

  -- Magnitude of velocity in the forward direction
  currentSpeed = currentVel.dot(currentFwd)

  if pMaxSpeed > 0 then
    -- Forward velocity as a proportion of max speed
    propSpeed = min(abs(currentSpeed) / pMaxSpeed , 1)
  else
    propSpeed = 1
  end if

  -- Ask Havok for the angular velocity
  angularVel   = rb.angularVelocity
  -- Ask Havok for the mass
  mass         = rb.mass
```

The vehicle's current velocity, its speed in the forward direction, its angular velocity and its mass are all stored. In addition the speed in the forward direction as a proportion of the maximum speed is calculated here and used later.

Now that all the initial configuration information has been obtained the effects can be added

```
  -- If going forward we want to reach our Max Speed
  if (pGoingForward) then
    diff = pMaxSpeed - currentSpeed
    -- Apply an impulse proportional to the difference
    -- between our desired speed and our actual speed
    imp = (diff * pAccGain * mass) * currentFwd
    rb.ApplyImpulse(imp)
  end if
```

© 2000 Havok.com Inc.

The player wants to accelerate. The pGoingForward property was set if the up arrow key is held down. To make the car accelerate you can apply a force or an impulse in the current forward direction. Impulses are used instead of forces here because they instantaneously change the velocity of the car and give you more direct control.

When accelerating, the player wants to reach maximum speed. The difference between the current speed and the maximum speed is evaluated and an impulse in the forward direction is applied which is proportional to this difference. The constant pAccGain determines how quickly the user can reach top speed. The impulse is multiplied by the mass so the behavior is independent of the mass of the vehicle. This makes it easier to choose sensible values for pAccGain: 0 means no acceleration and 1 means reach maximum speed instantaneously. Any value in between gives a gradual acceleration.

Exactly the same calculation is done for reversing, except the direction of the impulse is reversed.

```
-- If braking we reach 0
if (pBraking) then
  imp = (-currentSpeed * pBrakeGain * mass) * currentFwd
  imp.z = 0
  rb.ApplyImpulse(imp)
end if
```

When braking, a similar calculation causes the vehicle to come to a complete stop.

When turning, the turning circle for the vehicle is automatically adjusted. The car can turn in a tighter circle at lower speed.

```
-- Check if we can turn.
-- We check if the magnitude of our angular velocity is less than
-- our max turn speed.
-- Max Turning speed depends on velocity
mTS = min(abs(currentSpeed), (1.0 - propSpeed ) * pMaxTurnSpeed)
canTurn = (abs(angularVel.dot(currentAxis)) <  mTS)
```

The maximum turning circle (or turning speed) is inversely proportional to our current speed. The vehicle will turn sharper at lower speeds. In addition the calculation ensures the vehicle can never turn faster than it is traveling forward i.e. there’s no over-steer. Next, our current angular velocity, i.e. how fast the car is moving, is compared against this newly computed turning speed and it sets a variable which indicates if the vehicle is allowed to turn.

```
-- If we can turn and we’re turning left
if (canTurn and pGoingLeft) then
  -- Apply an angular impulse to turn us left
  imp = currentAxis * (pTurnGain * mass)
  rb.ApplyAngularImpulse(imp)
end if
```

If the vehicle can turn, an angular impulse is applied to cause the vehicle to spin. Angular impulses affect angular velocity, normal impulses change linear velocity.

© 2000 Havok.com Inc.

The vehicle should not move or slide laterally as easily as it moves forward or backward. Impulses are applied to compensate for this sliding – this kind of 'fake friction' emerges automatically from the physics engine if a fully physical car model with real wheels and proper suspension had been used instead.

```
-- Work out how fast we're sliding left/right
-- An compensate according to the grip property
-- We just allow the car to slide if its braking
slideSpeed = currentRight.dot(currentVel)
if not (pBraking) then
  imp = currentRight * (-slideSpeed * mass * pGrip)
  -- Apply an impulse to compensate for sliding
  rb.ApplyImpulse(imp)
end if
```

The lateral speed of the vehicle is evaluated and an impulse proportional to this is applied. The impulse is applied in a direction opposite to the direction the vehicle is sliding, to compensate for the loss of traction. This compensation is deliberately not performed when braking to allow the car to skid and slide when the handbrake is on.

The same kind of compensation is performed to stop the car spinning if the user stops turning – this mimics auto centering the wheels when the steering is released.

```
-- Apply an angular impulse to compensate for spinning
if (canTurn=false) or ((pGoingLeft=false) and (pGoingRight=false)) then
  imp = currentAxis * angularVel.dot(currentAxis) * (-pTurnGain * mass)
  rb.ApplyAngularImpulse(imp)
end if
```

Finally the last effect, aerodynamic drag, is added.

```
-- Apply Drag proportional to speed
imp = -currentVel * (pDrag * mass)
rb.ApplyImpulse(imp)
```

Drag is implemented just like a braking force except it works in all directions, not just forward / back. It is proportional to the current speed and is always applied to the vehicle.

This behavior is run on every frame of the animation and creates a tightly coupled feedback loop that works in harmony with the physics of the scene. The gain parameters let you decide exactly how much control you want over the vehicle's behavior.

© 2000 Havok.com Inc.

6 Things to Try

At the moment the driving behavior is applied regardless of the current state of the vehicle – it may be upside down, on its roof, or in mid air after jumping off the ramp. A quick check just before any of the real work is done helps.

```
-- Transform forward and axis vectors into world space
trans = pMember.model(pModel).transform.duplicate()
if (trans.position.z > 20) then
 return
end if
trans.position = Vector(0,0,0)
:
:
```

The chassis in the example usually rides along a flat plane at a height of 16.5 units. We extend this to give the driving some tolerance but this is still quite naïve: the car can drive upside down! See if you can come up with a better heuristic. How would the car be handled if it is driving over a varied terrain?

© 2000 Havok.com Inc.