Chapter 5

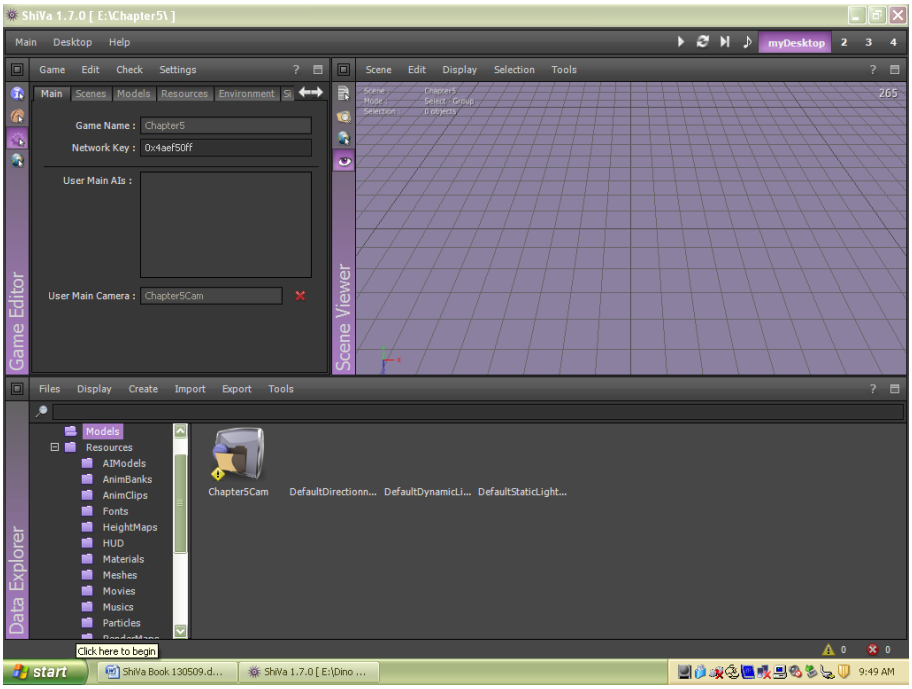
Chasing the Egg (The Revised Version)

In this Chapter I'm going to be showing you how to create a Camera that follows a Model. The Model that you will be using is the animated "egg_beak" Model that has so kindly been provided by Jerome at StoneTrip (for which I am extremely grateful, as I am a pretty atrocious modeller!).

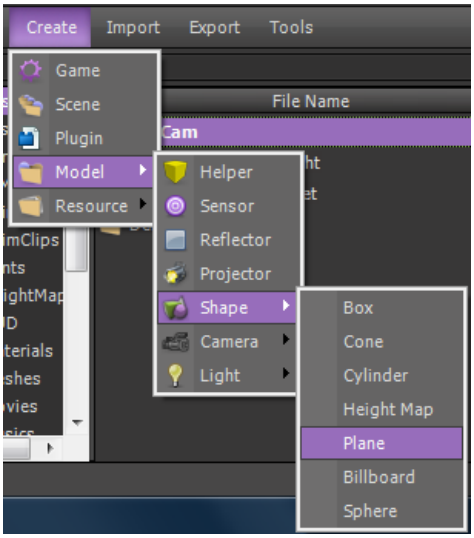
Since this is a revised version of this Chapter, I'd just like to say a HUGE "Thank You" to GeShr on the forum who managed to do what I haven't had time to do recently .... work out where things went wrong in the first version! Thanks once again GeShr!

So, let's get started. First you need to create a new Project, then a new Game, then a new Scene and finally a new Camera, just as you did in the previous Chapter.

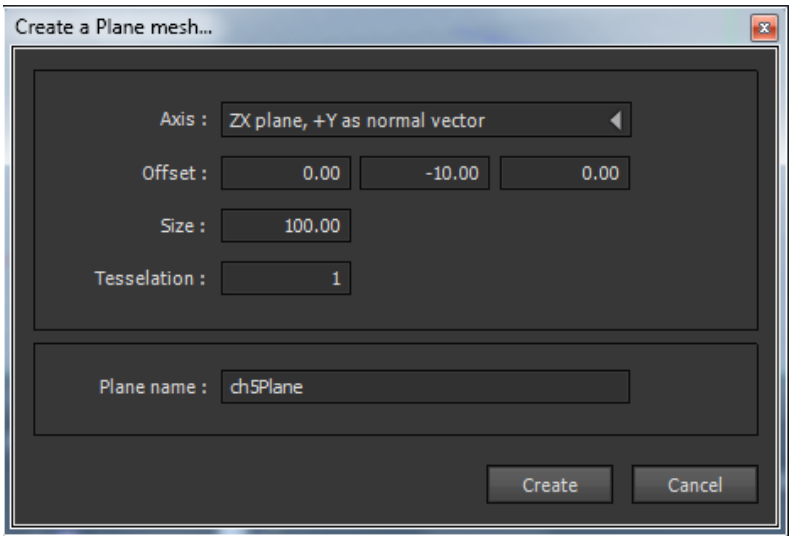
At this point you should have a Desktop looking something like this:



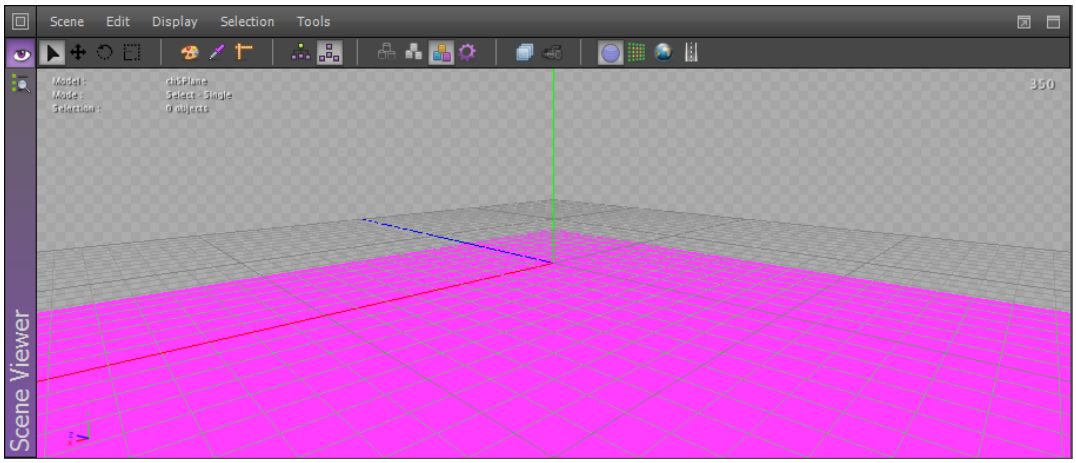
Next you need to create a ground plane on which the Model will move around. To do this, in the "Data Explorer" panel, click on "Create" and then select "Model", then "Shape" and finally "Plane" as shown below:



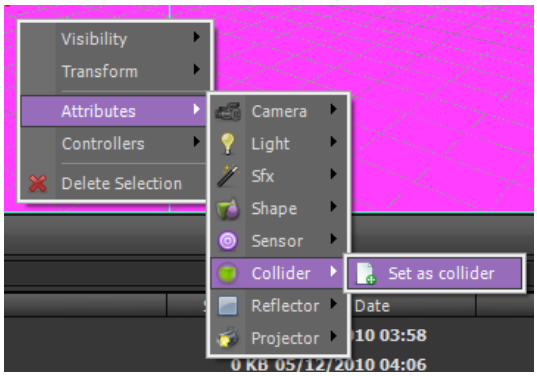
Now give your ground plane a name (like "myGroundPlane"). Next, in the box that appears (as shown below), make sure the "Axis:" field contains "ZX plane, +Y as normal vector", and the "Size:" field contains something like "100". Also, you MUST ensure that the Plane is a little below the "egg_beak" Model (to ensure that it doesn't fall through! – if, when you run the Scene, it looks blank, it is probably because the Model has fallen through, so you may need to lower the Plane a little bit more). To do this, simply enter a negative value in the middle "Offset:" field (this is the Y Axis). Finally, click on the "Create" button.



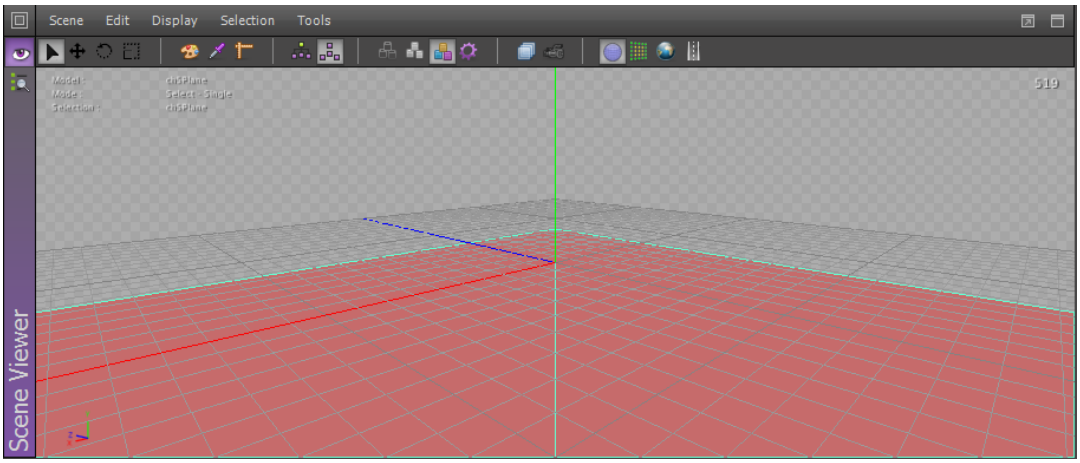
Next you just double-click on the new Plane in the "Data Explorer" and you should see something like this in the "Scene Viewer" (the pink bit is the new Plane ☺):



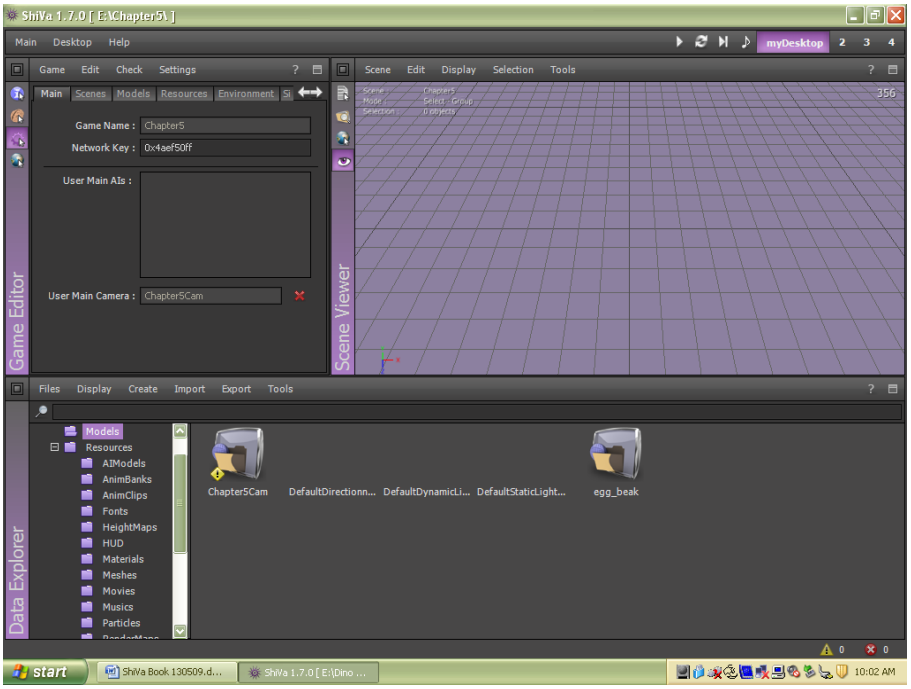
Finally for the Plane, you will need to set it as a "Collider". To do this, right-click on the Plane in the "Scene Viewer", and select "Attributes", "Collider" and lastly "Set as Collider" as shown below:



Your Plane will now look something like this (note the change in colour):



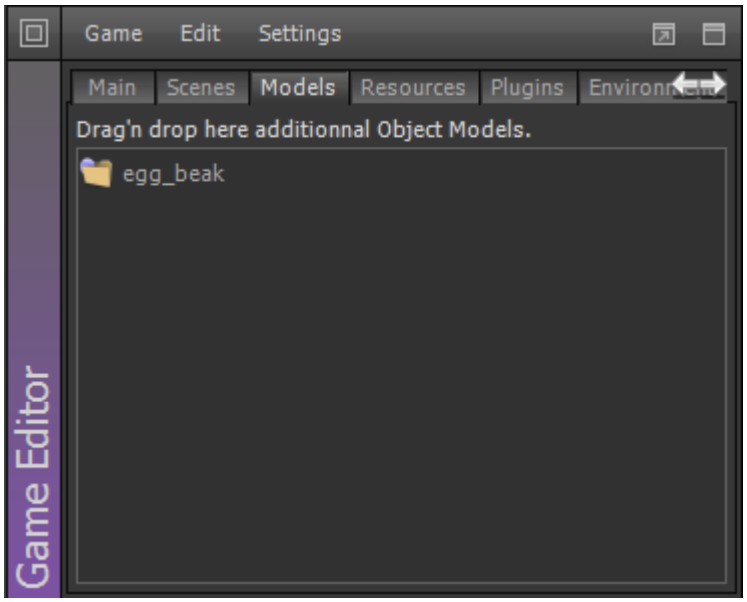
Now you need to import the Model. So, for this step we need to import a ShiVa archive (.ste file). Click on "Import", and then on "Archive". Click on the ellipsis (...) and navigate to where you saved the archive ("eggDyno-20090415.ste" from the Download pack). Finally, click on "Open" and then "Import". ShiVa will display its "Processing" splash screen, and then "egg_beak" should appear in the Models directory of the Data Explorer, as shown below:



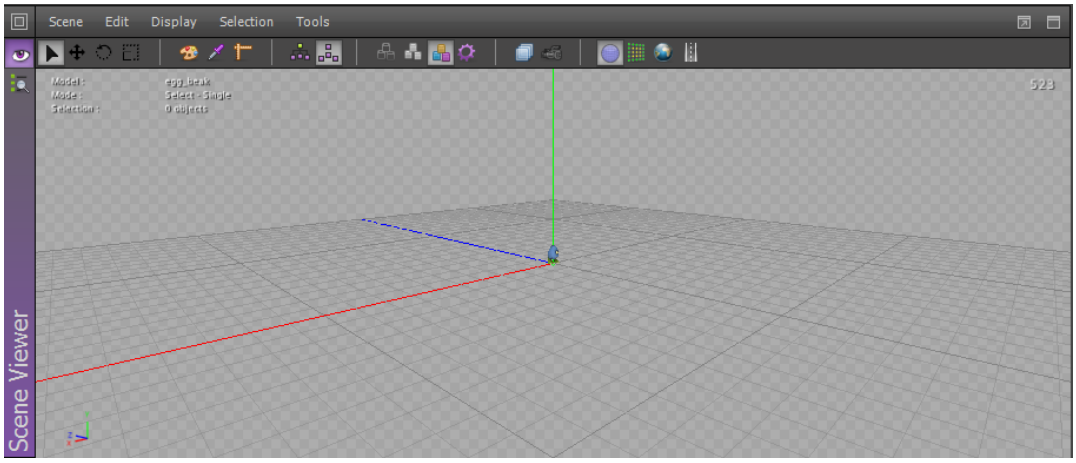
Also, you will notice that the following files have also been added in their respective folders:

AnimBanks	-	Egg_beak
AnimClips	-	egg_beak_walk
Materials	-	egg_beak
Meshes	-	egg_beak_geometries_0
Skeletons	-	egg_beak_Skeleton000
Textures	-	Egg_Beck_AO

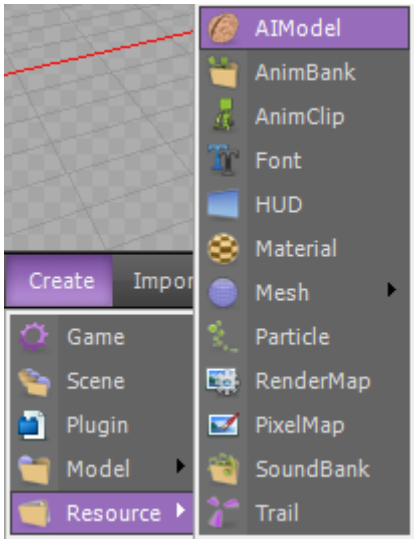
So, now you can add the Model to the Game by navigating to the "Models" folder and dragging and dropping the "egg_beak" Model into the "Models" tab of the "Game Editor":



Now you need to double-click on the "egg_beak" Model in either the "Game Editor", or the "Data Explorer". If prompted, ensure that you save your Scene. The "egg_beak" Model will now appear in the "Scene Viewer", as shown below:

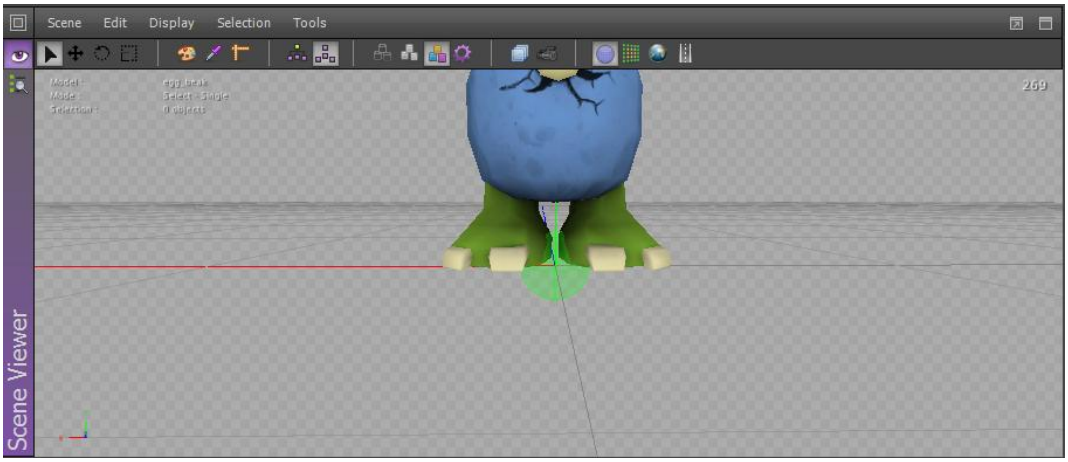


The next step is to create an AI Model to attach to it. To do this, click on "Create" (in the "Data Explorer" Menu bar), and then on "Resource", and finally "AIModel":



Now name your new AIModel (charAI?).

Returning to the "Scene Viewer", you should now rotate and zoom in on the "egg_beak" Model (using the "alt" key and the Left Mouse button to Rotate, and the "alt" key and Right Mouse button to Zoom), until you can see the little green Sphere between the Model's feet (as shown below):



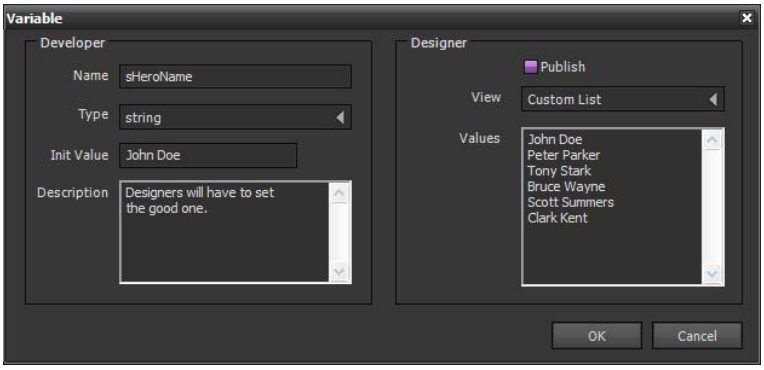
Now, right-click on the Sphere and choose "Controllers", "AI" and "Add AI", and select your charAI "AIModel".

NOTE: this Model has been made such that it sits perfectly on the ground when it has a Y-Axis value of "0.000". This is NOT going to be true for all Models, and so some trial and error may be required to get your Models to sit correctly.

Next you need to add your basic Scripts. So, as previously, go to the Game Editor, click on "Edit", then "User Main AI", and finally on "Create". Give your MainAI a name, and then click on "OK". Before you add any Scripts, you will need to create a Variable in your "MainAI".

So, now would be a good time to explain how you can create AIModel Variables.

Firstly, you'll need to click on "+ Add Variable" directly beneath the "[Variables]" header in the AIModel Editor. This will open a Dialog similar to the following:



To create your new Variable, enter the "Name" ("hCharObj"), and select the "Type" (Object) from the drop-down list, and finally click on "OK". Your new Variable will now appear in the tree view of the AIModel Editor. (Note that you can also add a "Description" if you want.)

Next, you need to create a Boolean Variable called "bGameMode", which is initially set to "false". This is used to determine if your Game is going to respond to Mouse actions, or not.

Throughout this Chapter I'll be using pretty standard naming conventions for Variables:

bXxxxYyyy	-	Boolean
nXxxxYyyy	-	Number
hXxxxYyyy	-	Object (the "h" denotes that it is a Handle to the Object)
sXxxxYyyy	-	String
tXxxxYyyy	-	Table
htXxxxYyyy	-	HashTable – This is not really a standard convention but we've already used "h"!
xXxxxYyyy	-	XML

Also note that the naming of the Variable follows a pattern with the initial letter being in lowercase, and any subsequent words start with an uppercase letter as shown in "hCharObj". Also, it's a very good idea to name your Variables with meaningful names (so that you, and others, can understand what's going on!).

This Variable will be used to store a Handle to the "egg_beak" Model.

Next, you need to add the "onInit" Handler to "MainAI", open the Script Editor, and add the following line to the "onInit" Handler:

```
local bOK = application.setCurrentUserScene ( "NAMEOFSCENE" )
local tempScene = application.getCurrentUserScene ( )

scene.createRuntimeObect ( tempScene, "myGroundPlane" )

local tempChar = scene.createRuntimeObject ( tempScene, "egg_beak" )
this.hCharObj ( tempChar )

object.sendEvent ( application.getCurrentUserMainCamera ( ), "CamAI", "onGetTarget",
this.hCharObj ( ) )
```

Obviously you should change the value of "NAMEOFSCENE" to whatever name you gave to your Scene!

All that you are doing here is creating the Scene, creating a Variable to hold the current Scene (tempScene), creating your Plane (this is created at runtime using the "createRuntimeObject" command), creating another Variable to hold a Handle to the created "egg_beak" Model (again created at runtime using the "createRuntimeObject" command), storing the Handle to the Model in your newly created AIModel Variable (hCharObj), and finally sending a Message (using the "sendEvent" command) to the Camera AI (camAI – see below) telling it to call its "onGetTarget" Function with the Parameter stored in the "hCharObj" Variable (i.e. the Handle to the "egg_beak" Model!).

Finally, save and compile your Script, and you are ready to run your Game.

However, before you can do that, you are going to have to move the Camera, as it is currently sitting under the Model, and you won't be able to see much!

So, as in the previous Chapter, you need to create a new Camera AIModel, and attach it to the Camera.

Once you have attached the AIModel to the Camera, create a new "onInit" Handler, and enter the following lines in the Script:

local hCam = application.getCurrentUserMainCamera ( )

object.translateTo (hCam, 0, 1.5, 5, object.kGlobalSpace, 1)
object.lookAt (hCam, 0, 1.5, 0, object.kGlobalSpace, 1)

The first line gets a Handle to the currently active Camera for the current User. This is very handy if you have multiple Cameras. A Handle is basically a copy of the Object.

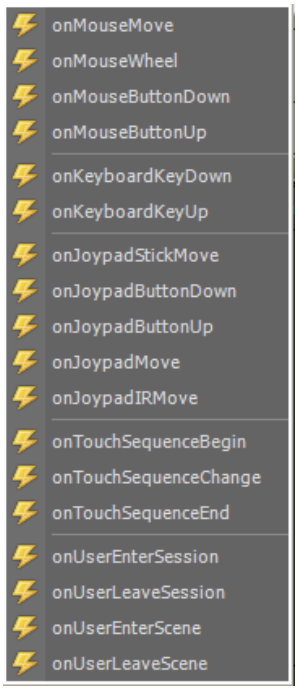
The next lines are almost identical to the "onEnter" Script from the previous Chapter, but here the Camera is placed a little above and behind the Model. With the "lookAt" the Camera is set to a reasonable direction so that it appears that the User is looking just in front of the Model. Obviously these figures can be changed if you do not like the view!

The next step is to move the Model. This is fairly simple, but does require quite a bit of coding.

For this part, I'm going to be using some of the code from the "3rd_person_cam_fw.ste" archive. This can be found in the download section of the ShiVa Developer Network. Thanks go to Clockmaster for his hard work on this (I know it was over 8 months ago, but hey! credit where credit's due).

You now need to create a couple of new Handlers for the MainAI - "onKeyboardKeyDown (kKeyCode)" and "onKeyboardKeyUp (kKeyCode)". Both of these are Handlers that you will need for pretty much any Game that you create using the keyboard as an input device.

To add the Handlers, click on "+ Add Handler", then on "User Handler" and select the Handler from the menu that appears.



Once you have added the two Handlers, open "onKeyboardKeyDown" (by double-clicking on it in the tree), and enter the following:

```
if ( kKeyCode == input.kKeyEscape )
then
    local b = not this.bGameMode ( )
    object.sendEvent ( this.hCharObj ( ), "CharAI", "onCaptureInput", b )
    this.bGameMode ( b )
else
    object.sendEvent ( this.hCharObj ( ), "CharAI", "onKeyboardKeyDown", kKeyCode )
end
```

OK, so once again you create a LOCAL Variable to hold a Handle to the current Object.

Next you create a check to see if the Input Parameter to the Handler "kKeyCode" is equal to the "Esc" key on the keyboard. ShiVa has built-in Constants for the majority of keys on the keyboard – see the documentation for a full list.

If the key that has been pressed is equal to the "Esc" key, then you set another LOCAL Variable ("b") to be equal to the OPPOSITE value of the AIModel Boolean Variable "bGameMode". So, if "bGameMode" is "true" then "b" becomes "false" and vice versa. Next, you send an Event to the "CharAI" of "hObj" to run its "onCaptureInput" Handler with a Parameter of "b". Finally, you set the value of "bGameMode" to equal "b".

OK, the "else" part of the above code runs if ANY key other than the "Esc" key has been pressed, and sends an Event to the "onKeyboardKeyDown" Handler of the "CharAI" with an Parameter of "kKeyCode".

So, in a nutshell, all that this code does is check to see if the "Esc" key has been pressed and, if so, sets "bGameMode" to "true" (if "false") and sends an Event to "CharAI" to advise it that the "Esc" key has been pressed. Otherwise it sends an Event to "CharAI" telling it which key has been pressed.

The next bit of code is for the "onKeyboardKeyUp" Handler:

This is really just a one-liner to send an Event to "CharAI" to tell it that the key has been released.

```
object.sendEvent ( this.hCharObj ( ), "CharAI", "onKeyboardKeyUp", kKeyCode )
```

Finally, for "MainAI", you need to create another "User Handler" from the "onMouseMove" option.

For this, you need to enter the following code:

```
object.sendEvent ( this.hCharObj ( ), "CharAI", "onMouseMove", nPointX, nPointY, nDeltaX,
nDeltaY, nRayPntX, nRayPntY, nRayPntZ, nRayDirX, nRayDirY, nRayDirZ )
```

This may look a bit confusing, but all its doing is sending an Event to "CharAI" telling it to run its "onMouseMove" Handler with the relevant Parameters.

The Parameters for the "onMouseMove" Handler are:

nPointX	-	The current Screen position of the Mouse in the X axis.
nPointY	-	The current Screen position of the Mouse in the Y axis.
nDeltaX	-	The distance the Mouse has moved, in the last Frame, in the X axis.
nDeltaY	-	The distance the Mouse has moved, in the last Frame, in the Y axis.
nRayPntX	-	The starting X axis position of the Ray into the Scene.
nRayPntY	-	The starting Y axis position of the Ray into the Scene.
nRayPntZ	-	The starting Z axis position of the Ray into the Scene.
nRayDirX	-	The direction of the Ray in the X axis.
nRayDirY	-	The direction of the Ray in the Y axis.
nRayDirZ	-	The direction of the Ray in the Z axis.

I'll quickly explain what's going on here, as it can be somewhat confusing. Basically, the Mouse pointer operates in 2 dimensions (X and Y axes), but the Game will be operating in 3 dimensions (X, Y and Z axes). Because of this, you need to be able to convert the 2D Mouse into a 3D position in the Game. This is done by projecting a Ray (similar to an invisible laser beam) into the Scene, hence the need for a 3D starting point for the Ray, and a 3D direction that it needs to be "fired". Invisible Rays are used a lot in 3D programming as they are extremely useful for detecting things such as collisions etc.

Well, that's it for "MainAI", so before you get stuck in to "CharAI", I'll cover off the AI for the Camera ("CamAI").

For this AI, you need to create the following:

Variables

Name		Type		Init Value
hTarget	-	Object		
hDynObject	-	Object		
bMoveBack	-	Boolean	-	False
bMouseMoved	-	Boolean	-	False
nDstAngleH	-	Number	-	0
nDstAngleV	-	Number	-	0

Functions

followTarget ( )

Handlers

onEnterFrame ( )	-	Note that this is a Standard ShiVa Handler
onGetDyn (target)	}	
onGetTarget (target)	}	These are all Custom Handlers
onGoBack (b)	}	
onMouseMover (angleH, angleV)	}	

I'll start with "followTarget":

local hCam = application.getCurrentUserMainCamera ( )
local dt = application.getAverageFrameTime ( )
local smoothFactor = 3
local hObj = this.hTarget ( )

All you are doing here is setting up some LOCAL Variables:

hCam	-	a Handle to the currently active Camera.
dt	-	the average Frame time of the application.
smoothFactor	-	a factor that will be used later on for smoothing movement.
hObj	-	a Handle to the AI Variable "hTarget".

local nObjDirX, nObjDirY, nObjDirZ = object.getDirection (hObj, object.kGlobalSpace)
local nObjX, nObjY, nObjZ = object.getBoundingSphereCenter (hObj)

These two lines will need a bit more explanation....

The first line creates three LOCAL Variables that are populated directly from the ShiVa in-built Function "getDirection". This Function returns the current direction of the Object ("o") in the X, Y and Z axes, in the specified space ("kGlobalSpace").

The second gets the location of the centre of the Sphere that encloses the Object.

I'll quickly explain the three possible values for the specified space:

kLocalSpace	-	The Object's own personal space.
kGlobalSpace	-	The space denoted by the entire Scene.
kParentSpace	-	The space denoted by the "Parent" of an Object (if there is one).

NOTE: for more information on "Parent" space, see ShiVa's documentation.

Now you need to code the actual movement of the Camera:

object.translateTo (hCam, nObjX – nObjDirX * 25, nObjDirY + 15, nObjZ – nObjDirZ * 25, object.kGlobalSpace, smoothFactor * dt)
object.lookAt (hCam, nObjX, nObjY + 6, nObjZ, object.kGlobalSpace, 6)

These two lines, whilst looking complicated, are in fact pretty easy to understand. All you are doing is moving the Camera ("hCam") to the location specified by the 3 axes ("nObjX – nObjDirX * 25", "nObjDirY + 15" and "nObjZ – nObjDirZ * 25"), in GLOBAL space, with a smoothing factor of "smoothFactor * dt".

The reason that the X and Z components use a subtraction is so that the Camera is placed behind the Model, and the reason that the Y component uses an addition is to place the Camera a little above the Model. The smoothing factor ensures that the Camera moves in accordance with the framerate of the application ("dt"), and hence makes the movement appear to be the same no matter what hardware the application runs on!

The second line makes the Camera look at the location specified by the X, Y and Z coordinates, in GLOBAL space, with a reasonably high factor ("6"). The factor determines how quickly the Camera looks at the required point, which is the centre of the Bounding Sphere, with a small increase in the Y axis.

Next, we will move on to the "onEnterFrame" Handler.

This is a simple one-liner that calls the "followTarget" Function that you have just created.

this.followTarget ( )

Note that to call Functions, from within the same AIModel, all you need to do is use "this.".

The next Handler you'll code is the "onGetDyn (target)" Handler. Again, this is a simple one-liner:

this.hDynObject (target)

All that this line does is set the value of the AI Variable ("hDynObject") to be equal to the value of the Parameter ("target").

Moving quickly on, you now need to code the "onGetTarget (target)" Handler. Once again, this is only one line of code:

this.hTarget (target)

And, once again, all you are doing is setting the value of an AI Variable ("hTarget") to equal the value of the Parameter ("target").

At this point, you only have two more Handlers to code to finish "CamAI":

The first is "onGoBack (b)":

this.bMoveBack (b)

Another one-liner that sets the value of the AI Variable ("bMoveBack") to the value of the Parameter ("b").

The second is "onMouseMover (angleH, angleV)"

this.nDstAngleH (angleH)
this.nDstAngleV (angleV)
this.bMouseMoved (true)

These three lines are pretty much identical to those you've coded before, and I shouldn't have to comment, I hope!

Well, that's it for "CamAI", so we'll now move on to the biggie "CharAI".....

"CharAI" consists of the following:

Variables

Name	Type	Init Value
hDynObj	Object	
bMoveForward	Boolean	False
bMoveBack	Boolean	False
bCaptureInput	Boolean	True
nCurAngleH	Number	0
nCurAngleV	Number	0
nDstAngleH	Number	0
nDstAngleV	Number	0
nSpeed	Number	90000
nGravity	Number	-390000

Functions

centerMouse ( )
createDynamic ( )
createHUD ( )
showMouse (b)
updateCharPos ( )
updateDynamics ( )

Handlers

onCaptureInput (b)
onEnterFrame ( )
onInit ( )
onKeyboardKeyDown (kKeyCode)
onKeyboardKeyUp (kKeyCode)
onMouseMove (nPointX, nPointY, nDeltaX, nDeltaY, nRayPntX, nRayPntY, nRayPntZ, nRayDirX, nRayDirY, nRayDirZ)

To begin, open the "centerMouse" Function, and type the following:

hud.callAction (application.getCurrentUser ( ), "HUDCenterMouse")

Another one-liner that makes a call to a HUD action ("HUDCenterMouse") of the HUD attached to the current User ("getCurrentUser ( )"). HUD calls are made in a very similar way to the sending of Events (as you can see!).

Now, you'll move on to a slightly more complex Function – "createDynamic":

local hObj = this.getObject ( )
local s = object.getScene (hObj)

These first two lines are pretty straightforward, in that they populate a LOCAL Variable ("hObj") with a Handle to the current Object, and another LOCAL Variable ("s") with a Handle to the current Scene that the Object is part of using ShiVa's "getScene" Function.

I'm going to include some error checking in this next part. Error checking should be done whenever a Handle is obtained, or an Object created, just to make sure that it has been successful.

```
if ( s ~= nil )
then
   local hDyn = scene.createRuntimeObject ( s, "dummy" )

   if ( hDyn ~= nil )
   then
      this.hDynObj ( hDyn )

      if ( dynamics.createSphereBody ( hDyn, 2 ) )
      then
         dynamics.enableDynamics ( hDyn, true )
         dynamics.enableCollisions ( hDyn, true )
         dynamics.enableRotations ( hDyn, false )
         dynamics.setLinearDamping ( hDyn, 1.00 )
         dynamics.setAngularDamping ( hDyn, 5.00 )
         dynamics.setLinearSpeedLimit ( hDyn, 2.00 )
         dynamics.setFriction ( hDyn, 60 )
         dynamics.setMass ( hDyn, 200 )
         object.sendEvent ( application.getCurrentUserMainCamera ( ), "camAI", "onGetDyn",
this.hDynObj ( ) )
      else
         log.message ( "Sphere Body Not Created in CharAI.createDynamic" )
      end
   else
      log.message ( "Dynamic Object Not Created in CharAI.createDynamic" )
   end
else
   log.message ( "Scene Not Found in CharAI.createDynamic" )
end
```

Well, that looks like a lot of code but, in reality, it's not that much. If you were to remove the Error checking, then you would lose approximately half of the code! Unfortunately, Error checking is extremely important whilst developing a Game, since it is pretty much the only way that you can be fairly sure where something is going wrong. The ShiVa Log (which can be found in the "Logs" Folder of your ShiVa installation) is your friend when it comes to tracing your mistakes as you can see by my use of "log.message", which writes a message to the Log. Since you can pause Games when running them in the Scene Editor, you can also keep checking the Log for any messages that may have been created up to that point. Also, you can keep track of Variables using the Log BUT, be warned, if you're Game is running at 60 Frames per second then you can easily end up with 60 Log messages per second!

An example of part of a Log file is shown below:

```
[04/18 13:00:46] ... Registered AIModel 'HLDMain'
[04/18 13:00:46] ... Registered AIModel 'HLDMainHUD'
[04/18 13:00:53] ... Changing current user scene to : 'dhScene1'
[04/18 13:00:53] ... Decompressing OGG Vorbis file 'W0' (channels:1, frequency:22050Hz, duration:0.86s, size:37760)
[04/18 13:00:53] ... Terrain loaded (64 chunks, 85 nodes, 4 levels)
[04/18 13:00:53] ... Reinitializing scene 'dhScene1'
[04/18 13:00:55] ... Scene 'dhScene1' reinitialized
[04/18 13:00:55] ... Scene 'dhScene1' loaded ( 4 objects )
[04/18 13:00:55] ... Registered AIModel 'HLDVisitCamera'
[04/18 13:00:55] ... HLD : Registered event KeyDown for object HLDVisitCamera (1) with AIModel HLDVisitCamera
[04/18 13:00:55] ... HLD : Registered event KeyUp for object HLDVisitCamera (1) with AIModel HLDVisitCamera
[04/18 13:01:24] ... Created fragment program : 0x00000301 (ALUIns:6, TEXIns:0, TEXInd:2, Par:3, Att:2, Tmp:3)
[04/18 13:01:24] !!! Vertex Program Parameters : 216
[04/18 13:01:24] ... Created vertex program 11 : 0x01000013 (Ins:67, Par:216, Tmp:5, Mat:64)
[04/18 13:01:24] !!! Vertex Program Parameters : 211
[04/18 13:01:24] ... Created vertex program 12 : 0x0000000b (Ins:57, Par:211, Tmp:5, Mat:64)
[04/18 13:01:24] ... Registered AIModel 'DinoAI'
[04/18 13:01:24] ... Created T Rex
[04/18 13:01:24] ... Created T Rex
[04/18 13:01:24] ... Created T Rex
```

As you can see, the last three entries are identical, and were all generated by a "log.message" command. Also, you may have noticed that the Log registers certain ShiVa internal actions, such as the "Decompressing OGG Vorbis file....." line.

Now, back to the code....

Once you have understood the Error checking, (basically checking that something exists i.e. is NOT "nil") the rest of the code narrows down to the following:

1) Creating a runtime Object
2) Setting an AI Variable with a Handle to the Object created in (1)
3) Creating a Dynamics Sphere Body around the newly created Object
4) Setting up the Dynamics parameters for the newly created Object
5) Sending a Message to the Camera telling it to call its "onGetDyn" Function

So, taking them one at a time:

1) The creation of runtime Objects is very useful when you need to add Objects to a Scene during the running of a Game, such as enemies, Helper Objects etc. In this case, you are creating a Helper Object in the Scene using the "dummy" Object. Before this Script can work though, this Object needs to be created as follows:

   Open the Data Explorer Module, and click on "Create", select "Model" and click on "Helper". Enter the Name of the Object ("dummy"), and that's it, your Helper Object is created. After you have created this Model, drag'n'drop it into the "Models" tab of the Game Editor (just like you did with "egg_beak").

2) You shouldn't have any problems with this by now.

3) The Dynamics Body that is created around an Object can be either a Sphere or a Cube. For most simple Games, you can use either, but for characters a Sphere is generally used. The Parameter "2" at the end of the "createSphereBody" Function determines the radius of the Sphere in ShiVa units.

4) Finally, the Dynamics have to be set up. The first three lines are pretty straightforward, as they simply enable (or disable) the specified Action. The remainder of the lines set up the actual Parameters that will be used by the Physics Engine. It is way beyond the scope of this book to explain all of the settings for the Physics Parameters, and a lot of the setting up of Physics systems is down to trial and error. If you did Physics at school, you'll remember that there were an awful lot of factors that affect how something moves. So, if in doubt, post to the Forum and, if someone has already done what you're trying to do, hopefully you'll get plenty of assistance.

Well, that's it for the "createDynamic" Function, so we'll move on to the "createHUD" Function. At this point of the book I'm going to be using Script to build a HUD but, in a later Chapter, we'll be using the new WYSIWYG HUD Editor.

```
local hUser = application.getCurrentUser ( )

local hCenterMouseAction = hud.newAction ( hUser, "HUDCenterMouse" )

if ( hCenterMouseAction ~= nil )
then
   hud.beginActionCommand ( hCenterMouseAction, hud.kCommandTypeSetCursorPosition )
   hud.pushActionCommandArgument ( hCenterMouseAction, 0 )
   hud.pushActionCommandArgument ( hCenterMouseAction, 0 )
   hud.endActionCommand ( hCenterMouseAction )
end
```

OK, looks a bit complicated but, once again, it's not really. The first line creates a Handle to the current User. The second creates a new HUD Action called "HUDCenterMouse".

The "if" .. "then" statements check that the Command has been created and, if so, basically sets the cursor position to (0, 0). This is accomplished as follows:

beginActionCommand	-	Sets the type of the Command – "kCommandTypeSetCursorPosition"
push..Argument	-	Adds the first argument of the above Command
push..Argument	-	Adds the second argument of the above Command
endActionCommand	-	Signifies the end of the Command.

So, as you can see, it's pretty straightforward once you've got the hang of having to list out every part of the overall Command on a separate line. It's a bit like programming in a low level language!

The next few lines are pretty much the same, so I won't comment. As I said before, we'll be using the new WYSIWYG HUD Editor later, so you will be able to compare the two ways of creating HUDs for yourselves.

```
local hShowMouseAction = hud.newAction ( hUser, "HUDShowMouse" )

if ( hShowMouseAction ~= nil )
then
   hud.beginActionCommand ( hShowMouseAction, hud.kCommandTypeSetCursorVisible )
   hud.pushActionCommandArgument ( hShowMouseAction, true )
   hud.endActionCommand ( hShowMouseAction )

end

local hHideMouseAction = hud.newAction ( hUser, "HUDHideMouse" )
```

```
if ( hHideMouseAction ~= nil )
then
   hud.beginActionCommand ( hHideMouseAction, hud.kCommandTypeSetCursorVisible )
   hud.pushActionCommandArgument ( hHideMouseAction, false )
   hud.endActionCommand ( hHideMouseAction )
end
```

As you can probably tell, the first block sets the Cursor to be visible, and the second block sets the Cursor to be invisible.

That's all for the HUD. Next stop is the "showMouse (b)" Function:

```
local hUser = application.getCurrentUser ( )

if ( b )
then
   hud.callAction ( hUser, "HUDShowMouse" )
else
   hud.callAction ( hUser, "HUDHideMouse" )
end
```

All that this Function does is call one of the two HUD Actions created above (to show, or hide, the Mouse Cursor) depending on the value of the input Parameter ("b").

Moving on, we come to something a bit more interesting, moving our Model. To do this, open the "updateCharPos" Function and type the following:

```
local hDyn = this.hDynObj ( )
local hObj = this.getObject ( )
local dt = application.getLastFrameTime ( )
```

The first lines are pretty straightforward, and you should have no problems with them. The only difference here is that "dt" is set to the time taken since the last Frame was displayed on the Screen. This is another extremely common Command in 3D programming, as pretty much all movements are made relative to the Frame time.

```
if ( hDyn ~= nil )
then
   local x, y, z = object.getTranslation ( hDyn, object.kGlobalSpace )
```

OK, here you are checking to make sure that your Dynamic Object is valid and, if so, getting its Position in Global space.

```
   object.translateTo ( hObj, x, y, z, object.kGlobalSpace, 10 * dt )
```

This line moves the Object ("hObj") to the current position of the Dynamic Object ("hDyn") worked out from the previous line. The final Parameter "10 * dt" is very important. This Parameter basically tells the movement to occur over a period of time, and not instantaneously, which is vital for making the movement appear to be realistic. Otherwise, the Model would seem to jump from location to location, which doesn't look too good!

The next few lines are used to set the orientation of the Model:

```
    local rh = math.interpolate ( this.nCurAngleH ( ), this.nDstAngleH ( ), 20 * dt )
    this.nCurAngleH ( rh )

    object.setRotation ( hObj, 0, rh, 0 , object.kGlobalSpace )
end
```

The first line creates a LOCAL Variable ("rh") and populates it with an interpolated value between "nCurAngleH" (the current Horizontal angle), and "nDstAngleH" (the desired Horizontal Angle). The Interpolation is linear, and controlled by the final Parameter "20 * dt". Again you are using a multiple of the Frame time to control the movement, only this time the movement should be quicker, hence the larger multiplier.

The next line saves the new value of "nCurAngleH" for the next Frame, and the last line (excluding the "end") actually causes the Model to move by setting its Rotation (again in Global Space). The reason that the Rotation is carried out about the Y axis is simple really. Imagine how you turn around, you rotate around a line from the bottom of your feet to the top of your head (your own personal Y axis!).

OK, so your Model turns around, but it can't move from its current location. The next Function ("updateDynamics") is where all of the Model's movement is carried out:

```
local hObj = this.getObject ( )
local hDyn = this.hDynObj ( )

if ( hDyn ~= nil )
then
```

Up to here is pretty much the same as you'll see in any Function that relates to movement, so I won't comment as you've seen it all before.

```
    local oXx, oXy, oXz = object.getXAxis ( hObj, object.kGlobalSpace )
    local oYx, oYy, oYz = object.getYAxis ( hObj, object.kGlobalSpace )
    local oZx, oZy, oZz = object.getZAxis ( hObj, object.kGlobalSpace )

    local fx, fy, fz = 0, 0, 0
```

These lines create some LOCAL Variables, with the first three lines getting the current X, Y and Z values for each of the three axes of the Object.

```
    if ( this.bMoveForward ( ) )
    then
        fx, fy, fz = math.VectorSubtract ( fx, fy, fz, oZx, oZy, oZz )
    elseif ( this.bMoveBack ( ) )
    then
        fx, fy, fz = math.VectorAdd ( fx, fy, fz, oZx, oZy, oZz )
    end
```

This is where it starts getting trickier as you have introduced some Vector maths into the calculations. All that's happening here is a check to see if "bMoveForward" is "true". If it is, then the value of the Z axis is subtracted from the LOCAL Variables "fx", "fy", and "fz". Why Subtract? Well, that's easy, this is because the GLOBAL Z axis is pointing directly OUT of the Screen, and because of this, if you need to move forward, you need to reduce the value of "Z". However, if you need to move backwards, then you need to increase the value of "Z", hence the addition.

```
fx, fy, fz = math.vectorNormalize ( fx, 0, fz )

if ( this.bMoveForward ( ) == false )
then
   this.nSpeed ( 100000 )
else
   this.nSpeed ( 200000 )
end

fx, fy, fz = math.vectorScale ( fx, 0, fz, this.nSpeed ( ) )
```

Firstly, as with the majority of Vector operations, you need to normalise the Vector before using it. Normalising a Vector means that the Vector is shortened (or lengthened) to have a length (magnitude) of 1. This is done by the following calculation:

$$\frac{v}{|v|}$$

where v = the Vector and |v| = the length of the Vector.

So, since the length of a Vector (L) is calculated using:

$$L = \sqrt{x^2 + y^2 + z^2}$$

Then the above equation becomes:

$$x = \frac{x}{L}$$

$$y = \frac{y}{L}$$

$$z = \frac{z}{L}$$

for each coordinate of the Vector.

The next part of the above code sets the movement speed ("nSpeed") depending on whether the Object is going forwards or backwards. To make it more realistic, the backward speed is lower than the forward speed.

Finally, the vector is scaled according to the value of "nSpeed". Since the Object is not leaving the ground, the value of the Y component is set to "0".

The final bit of this Script moves the Dynamic Object by applying the Forces that you have just calculated, in Global space:

```
    dynamics.addForce ( hDyn, fx, fy, fz, object.kGlobalSpace )
    dynamics.addForce ( hDyn, 0, this.nGravity ( ), 0, object.kGlobalSpace )
end
```

Note that Gravity is always added separately. I don't personally know why, but I'm sure if you asked in the Forum someone will know!

The next Script to be entered is "onCaptureInput (b)":

```
this.bCaptureInput ( b )
this.showMouse ( not b )
this.bMoveForward ( false )
this.bMoveBack ( false )
```

A pretty straightforward Script that sets the value of "bCaptureInput" according to the Parameter ("b"), calls the Function "showMouse" with a Parameter of "not b" (i.e. the opposite of "b"), and finally sets the values of "bMoveForward" and "bMoveBack" to "false", to stop any movement.

Next, we'll move on to the "onEnterFrame" Script, which is run once every Frame:

```
this.updateDynamics ( )
this.updateCharPos ( )

object.sendEvent ( application.getCurrentUserMainCamera ( ), "CamAI", "onGoBack", this.bMoveBack ( ) )
```

The first two lines call the relevant Functions in "CharAI". Note how "updateCharPos" is called AFTER "updateDynamics". This is so that the Dynamic Object can be moved and then the Model can be moved to the new position of the Dynamic Object, as described earlier.

The last line shows how easy it is to send commands to other AIs. This line sends an Event to "CamAI", telling it to run its "onGoBack" Handler with a Parameter of "bMoveBack". This is done so that the Camera responds the same as the Model (i.e. goes backwards (or forwards) when the Model is moving backwards (or forwards)).

Now you need to make some changes to the "onInit" Handler that you created oh so long ago. So, delete everything that you did before, and type the following:

```
this.createDynamic ( )
this.createHUD ( )
this.showMouse ( false )
```

OK, all standard stuff here, and hopefully you'll be able to work out what's going on!

Just a few more Scripts and you should be able to run the Game and "Chase the Egg"!

The next couple of Scripts are pretty straightforward as all they are doing is checking which key has been pressed (or released) and then setting some AI Variables depending on the value.

Firstly, create the "onKeyboardKeyDown (kKeyCode)" Handler:

```
if ( kKeyCode == input.kKeyW )
then
    this.bMoveForward ( true )
    this.bMoveBack ( false )
elseif ( kKeyCode == input.kKeyS )
then
    this.bMoveForward ( false )
    this.bMoveBack ( true )
end
```

All that this means is that if the Player presses the "W" key then the Model should move forward, and if the Player presses the "S" key, then the Model should move backwards.

Next, create the "onKeyboardKeyUp (kKeyCode)" Handler:

```
if ( kKeyCode == input.kKeyW )
then
    this.bMoveForward ( false )
elseif ( kKeyCode == input.kKeyS )
then
    this.bMoveBack ( false )
end
```

This is even easier than the last Script, since all it does is set the relevant movement AI Variable to "false" when the key is released.

The final bit of movement code is how to rotate the Model. This is all done in the "onMouseMove (nPointX, nPointY, nDeltaX, nDeltaY, nRayPntX, nRayPntY, nRayPntZ, nRayDirX, nRayDirY, nRayDirZ)" Handler:

```
if ( this.bCaptureInput ( ) )
then
```

Firstly, a check is made to ensure that movement is currently possible (i.e the Game isn't paused).

```
    this.nDstAngleH ( this.nDstAngleH ( ) – 90 * nDeltaX )
    this.nDstAngleV ( math.clamp ( this.nDstAngleV ( ) + 90 * nDeltaY, -85, 85 ) )
```

The first of these two lines changes the current value for the Horizontal turn by an amount equivalent to "90 * nDeltaX". Remember that nDeltaX is the amount that the Mouse has moved in the last Frame. Also, if the Mouse moves to the left, then "nDeltaX" will be negative, hence the need to subtract the turn amount so that ShiVa responds correctly (a positive turn in ShiVa is to the left). This also applies to a right turn where ShiVa requires a negative value, and mouse movement to the right would be positive, hence the need to make the turn negative.

The second line is pretty much the same, except for two things:

1) An upwards turn is Positive for both the Mouse and ShiVa.
2) The "math.clamp" in-built Function restricts the up/down movement to between "-85" and "+85" degrees (ever tried moving your own head up or down by more than these values? Painful isn't it?

```
    if ( nPointX < -0.1 or nPointX > 0.1 or nPointY < - 0.1 or nPointY > 0.1 )
    then
        this.centerMouse ( )
    end
end
```

This next bit checks to see if the Mouse movement is less than 0.1 in all directions and, if it is, the "centerMouse" Function is called to keep the Mouse in the centre. This stops movements occurring when the Mouse has hardly moved and, as such, stops some of the jerking that may occur with such small movements.

```
object.sendEvent ( application.getCurrentUserMainCamera ( ), "CamAI", "onMouseMover", this.nDstAngleH ( ), this.nDstAngleV ( ) )
```

This final line sends an Event to "CamAI" telling it to run its "onMouseMover" Function with the relevant Parameters.

Well, that's it for the Scripts!

Now it's time to compile all the Scripts and try running your Game. Just make sure that you remember the following controls:

1) The "W" key moves the Egg forward
2) The "S" key moves the Egg backward
3) The Mouse is used to look around and change direction.
4) The "Esc" key switches control of the Mouse between the Game and your computer.

Well, That's All - Have Fun!