Chapter

17

The Game Scripts

Well, we had a bit of fun in the last Chapter, creating our Terrain. Unfortunately, now is the time to put your programming skills to the test, as we will be creating and amending our Scripts. Throughout this Chapter, I'll explain the changes I've made to the HLDL Framework Scripts, and also any new Scripts that we will need for our Game.

ADDING THE HLD LIBRARY MODELS

The first HLD Library AIModel that you'll need to add is the "HLDMain" AIModel. To do this, drag and drop "HLDMain" from the AIModels folder of the Data Explorer to the "User Main AIModels:" box in the Game Editor. You'll also need to drag and drop the "HLDArial" Font resource from the "Fonts" folder of the Data Explorer to the "Resources" tab in the Game Editor.

To finish setting up the "HLDMain" AIModel, you have to add some Environment Variables to the Game. Environment Variables are Variables that are available to the entire application. In the Game Editor, select the "Environment" tab, and click on the "Add" button. In the dialog box, create your Variables as follows:

Name: HLDStartupScene
Type: String
Value: THIS IS THE NAME OF YOUR SCENE (in my case "dhScene1")

Name: gameOn
Type: Boolean
Value: false

Name: nDinosaurs
Type: number
Value 10.000

Name: HLDStartupText0
Type: String
Value: Dino Hunter

Name: HLDStartupText1
Type: String
Value: a ShiVa Demo Game

Name: HLDStartupText2
Type: String
Value: by Shando69

Obviously, you can change the values of the "nDinosaurs" Variable (which determines the number of dinos in the Game – Be Careful! "Winning" the Game is determined by reaching a set number of points - 1000 – and Ankylosaurs are only worth 50!, so if you get 10 Ankylosaurs, you can never win the Game!), and the last three Strings if you want!

Before we go any further, I'd just like to mention that you can customize the "HLDMain" AIModel quite easily (as you saw above) by adding start up Variables. In the Game Editor, you can add new Variables named "HLDStartupTextX" (or "HLDStartupLogoX"), where X represents the index of the required start-up option, and these will be displayed BEFORE the start-up Scene. So, you could have a logo, followed by some text, and then another logo, if you wanted. You can have up to a maximum of 10 start-up options before the first Scene starts, though this number can be changed in the code if required.

For example, you could have "HLDStartupLogo0", "HLDStartupText1", "HLDStartupLogo2" as Variables, and this will display as Logo -> Text -> Logo before the Scene starts.

The next Model to be added is the Camera, "HLDVisitCamera", which you can drag and drop straight into your Scene from the "Models" folder of the Data Explorer. You'll also need to add a "Tag" to the Camera. So, right click on the Camera and click on "Edit Selection Tag". Name the Camera "HLDStartupCamera", and it will be used, by default, as the main Camera when the Scene is entered.

Lastly, we need to drag and drop the "HLDVisitCamera" from the Data Explorer to the "User Main Camera:" field of the Game Editor, and the "HLDDummy" Model from the Data Explorer to the "Models" tab of the Game Editor.

Oops, nearly forgot, we also need to set the Terrain as a Collider, or we'll just fall through it! Open the Terrain Editor, open the "Geometry" roll-up, and select the check box next to "Is a collider".

Now, we should be ready to walk around our Terrain, so let's try it!

PS: the keys you need to move around the scene are:

"Up" arrow	-	This moves the Camera forwards.
"Down" arrow	-	This moves the Camera backwards.
"Left" arrow	-	This turns the Camera to the left.
"Right" arrow	-	This turns the Camera to the right.

OK, so now you can walk around your Terrain, hide in the grass, and walk around your trees! Note that you are also able to walk straight through your trees! This is because we have not set the trees as "Colliders". To be able to do this, you'll have to add the tree Model as a Model in the Scene, and set it as a Collider.

I've tried to delay it as long as possible, but now we need to get stuck into our Scripts. So, put your programming brain into gear and we'll get started…….

HLDMain

Basically "HLDMain" can be seen as the overall controller of the Game, and "HLDVisitCamera" as the moving part that can be controlled by the user.

"HLDMain" creates the initial fade screen, loads the start-up Scene and, at the beginning of each frame, sends the relevant Events to the other parts of the HLD library that need to be called (depending on what's happening, of course!).

HLDMain consists of the following Scripts and Variables (with brief descriptions – the full Scripts will be described later, and I will highlight any changes I have made to the base Scripts in red):

VARIABLES:

bSwitchingScene	Type – Boolean Initial Value – false This is used for switching Scenes.
hLastUnderCursor	Type – Object Initial Value – 'nil' This is used for remembering the last Object found under the Mouse Cursor.
hUnderCursorObject	Type – Object Initial Value – 'nil' This is used for detecting the Object currently under the Mouse Cursor.
sNextScene	Type – String Initial Value – "" This is used to denote the next Scene.
sStartupScene	Type – String Initial Value – "" This is used to denote the start-up Scene.
tHUDInstances	Type – Table Initial Value – 'empty' This is used to store the HUD instances.
tKeyDownEventTargetObjects	Type – Table (hObject, sAIModel) Initial Value – 'empty' This is used to store the Key down Event Objects.
tKeyUpEventTargetObjects	Type – Table (hObject, sAIModel) Initial Value – 'empty' This is used to store the Key up Event Objects.

tMouseClickEventTargetObjects	Type – Table (hObject, sAIModel) Initial Value – 'empty' This is used to store the Mouse click Event Objects.
tMouseEnterEventTargetObjects	Type – Table (hObject, sAIModel) Initial Value – 'empty' This is used to store the Mouse enter Event Objects.
tMouseLeaveEventTargetObjects	Type – Table (hObject, sAIModel) Initial Value – 'empty' This is used to store the Mouse leave Event Objects.

FUNCTIONS:

buildStartupSequence	This Function builds the start up sequence based on the Environment Variables that you previously set up (such as "HLDStartupScene" above).
destroyRegisteredHUDInstances	This Function destroys any HUD instances that have been created.
findAndUseStartupCamera	This Function determines which Camera will be used as the initial Camera when the first Scene is loaded.
findUnderCursorObject	This Function returns any Object found under the Mouse cursor.
sendMouseClickEventToObject	This Function sends an Event to any Object that has indicated that it needs to be advised that the Mouse button has been clicked.
sendMouseEnterEventToObject	This Function sends an Event to any Object that has indicated that it needs to be advised that the Mouse has fired the "MouseEnter" Event.
sendMouseLeaveEventToObject	This Function sends an Event to any Object that has indicated that it needs to be advised that the Mouse has fired the "MouseLeave" Event.
switchScene	This Function handles the switching between Scenes, and any associated fading in or out.

HANDLERS:

onEnterFrame	This Handler is called once every Frame.
onForwardEvent	This Handler sends an Event to an Object, allowing the passing of up to 4 Parameters.
onInit	This Handler is called on initialization of "HLDMain".
onKeyboardKeyDown	This Handler sends an Event to any Object that has indicated that it needs to be advised that a Keyboard key has been pressed.
onKeyboardKeyUp	This Handler sends an Event to any Object that has indicated that it needs to be advised that a Keyboard key has been released.
onMouseButtonDown	This Handler calls the "findUnderCursorObject" Function, and then if anything is found, calls the "sendMouseClickToObject" Function when the Mouse Button is pressed.
onMouseButtonUp	This Handler does nothing in the base library. It has been added in case you need to send an Event when the Mouse Button has been released (for example: dragging & dropping chess pieces etc.).
onMouseMove	This Handler does nothing in the base library. It has been added in case you need to send an Event when the Mouse is moved (for example: for dragging & dropping chess pieces etc.).
onRegisterHUD	This Handler simply saves an instance of the HUD, passed as a Parameter, into the Table of HUD instances.
onRegisterObjectEvent	This Handler saves an Object and an AIModel (both passed as Parameters) into the Table that corresponds with the final Event type Parameter.
onSetNextScene	This Handler simply sets the Variable "sNextScene" with the Scene that has been passed in as a Parameter.

onSwitchScene	This Handler calls the Function "destroyRegisteredHUDInstances", sets the Scene to the value stored in "sNextScene", calls the Function "findAndUseStartupCamera", and then, finally, tells the main User AI to execute "HLDFadeScreenIn".
onUnregisterHUD	This Handler removes the HUD instance, passed as a Parameter, from the Table of HUD instances.
onUnregisterObject	This Handler does nothing in the base library. It has been added in case you need to unregister an Object that has had Events registered using "onRegisterObjectEvent".

Now that I've explained what is available, it's time to start changing and creating our Scripts.

onInit ( )

This Handler is called once, when "HLDMain" is initialized. It is used to set up any defaults that may be required by "HLDMain", and also to do any pre-processing that is required prior to the start.

The first few lines are fairly self-explanatory, in that they just create a LOCAL Variable for the current User, check it's been created, and if it has, set the default Text shadow colour (to white), and the default Font of the HUD to "HLDArial":

```
local hUser = this.getUser ( )

if ( hUser ~= nil )
then
   hud.setDefaultTextShadowColour ( hUser, 0, 0, 0, 192 )
   hud.setDefaultFont ( hUser, "HLDArial" )
```

This first bit of code introduces a few ShiVa, and general programming, standards:

General - the lining up, and indenting, of code is standard practice and makes the code more readable. In this case it is between the "if" .. "then" .. "end" statements ("if".. "then" is shown above). Not only does it make it more legible, but also makes it easier to spot, at a glance, a "block" of code. i.e: anything indented between the "if".. "then" and the "end" will be executed if the "if" statement returns TRUE or FALSE (depending on what is required).

ShiVa - the use of "local" denotes a LOCAL Variable
the use of "this"returns a Handle to the currently active Object.
the use of "XXX.YYYY" is a way of calling a particular ShiVa API (API = Application Programming Interface – what makes ShiVa tick), in this case the HUD API. Note that the values in the brackets following the call to the API are called Parameters, and denote values that will be passed into the Function that is being called. In the StoneScript API docs you'll see that the majority of Functions in the API require a Parameter, or Parameters, of some sort.

Also, I'd like to make a quick comment regarding the "if" .. "then" statement. The use of this is the main way of asking questions in your scripts (in this case is the value of the Variable "hUser" not equal to "nil"?). Note that the "~=" characters denote the "not equal to" part of the question (if you wanted to check for something equal to something else you would use "=="). The use of "nil" in ShiVa denotes nothing at all (by that I mean empty, nothing, zilch etc.) and is used frequently to check that something has been created (or setup) properly (in this case "this.getUser()" has returned a valid User).

The next section of the code will require a little more explanation, but all it is really doing is fading in, or out, the HUD that has been set up as the fade screen:

```
local fadeScreen = hud.newComponent ( hUser, hud.kComponentTypeContainer,
"HLDFadeScreen" )
```

This line creates a new LOCAL Variable ("fadeScreen"), which is a HUD component, specifically a Container ("kComponentTypeContainer" – a Constant – Note that ALL Constants in ShiVa have "k" as the first letter of their name), called "HLDFadeScreen".

```
if ( fadeScreen )
then
```

These two lines are a test to ensure that "fadeScreen" has been created (always important whenever you create a Variable!)

```
    hud.setComponentPosition ( fadeScreen, 50, 50 )
    hud.setComponentSize ( fadeScreen, 100, 100 )
    hud.setComponentBackgroundColor ( fadeScreen, 0, 0, 0, 255 )
    hud.setComponentBorderColor ( fadeScreen, 0, 0, 0, 0 )
    hud.setComponentVisible ( fadeScreen, true )
    hud.setComponentZOrder ( fadeScreen, 254 )
```

The above lines set the various Parameters of the "fadeScreen" HUD, such as position, size, visibility etc. It is preferable when building a HUD to use percentages (as is the case here) to denote the position of the Component on the screen, as not all of your Users will use the same screen resolution. Some may use 800x600, others 1280x1024 etc. (unless you restrict the screen resolution in your program of course!).

```
    local fadeInAction = hud.newAction ( hUser, "HLDFadeScreenIn" )
    local fadeOutAction = hud.newAction ( hUser, "HLDFadeScreenOut" )
```

These two lines create two new LOCAL Variables, one for fading in, and one for fading out. Both of these Variables are HUD Action Variables (i.e.: they are Actions performed by the HUD).

```
    if ( fadeInAction )
    then
```

Again, a simple test that the "fadeInAction" Variable has been successfully created.

```
        hud.beginActionCommand ( fadeInAction, hud.kCommandTypeInterpolateOpacity )
        hud.pushActionCommandArgument ( fadeInAction, fadeScreen )

        hud.pushActionCommandArgument ( fadeInAction, 0 )
        hud.pushActionCommandArgument ( fadeInAction, hud.kInterpolatorTypeLinear )
        hud.pushActionCommandArgument ( fadeInAction, 1000 )
        hud.endActionCommand ( fadeInAction )
        hud.beginActionCommand ( fadeInAction, hud.kCommandTypeSetVisible )
        hud.pushActionCommandArgument ( fadeInAction, fadeScreen )
        hud.pushActionCommandArgument ( fadeInAction, false )
        hud.endActionCommand ( fadeInAction )
    else
        log.message ( "fadeInAction not created in HLDMain.onInit"
    end
```

You may have noticed that I have added a couple of lines that produce a Log Message if the create Action fails. This is extremely useful for debugging, and if you look at the code in the ".ste", you'll see just how many I have included – I won't be including them all here, as you should easily be able to work this out for yourself.

These lines set up the "fadeInAction" of the HUD.

```
    if ( fadeOutAction )
    then
        hud.beginActionCommand ( fadeOutAction, hud.kCommandTypeSetVisible )
        hud.pushActionCommandArgument ( fadeOutAction, fadeScreen )
        hud.pushActionCommandArgument ( fadeOutAction, true )
        hud.endActionCommand ( fadeOutAction )
        hud.beginActionCommand ( fadeOutAction, hud.kCommandTypeInterpolateOpacity )
        hud.pushActionCommandArgument ( fadeOutAction, fadeScreen )
        hud.pushActionCommandArgument ( fadeOutAction, 255 )
        hud.pushActionCommandArgument ( fadeOutAction, hud.kInterpolatorTypeLinear )
        hud.pushActionCommandArgument ( fadeOutAction, 1000 )
        hud.endActionCommand ( fadeOutAction )
    end
end
```

This whole block of code is almost the same as the one above, but sets the HUD "fadeOutAction".

```
local startupScene = application.getCurrentUserEnvironmentVariable ( "HLDStartupScene" )
```

This line creates a new LOCAL Variable that is initialised with the value that has been stored in the "HLDStartupScene" Environment Variable.

```
if ( startupScene )
then
```

That simple test again!

```
    this.sStartupScene ( startupScene )
```

If "startupScene" has been successfully created, this line sets its value in the current User.

```
else
    log.error ( "HLDMain : Please create an environment variable named
    \"HLDStartupScene\" containing the name of the startup scene." )
end
```

Otherwise, we have a problem, and a Log Error Message is generated accordingly.

```
this.buildStartupSequence ( )
```

Finally, a call is made to build the actual start up sequence.

```
else
    log.message ( "User not found in HLDMain.onInit" )
end
```

Finally I've included the Log Message for when a User is not found.

onEnterFrame ( )

This Handler is called at the start of each Frame, and contains any processing that needs to be carried out on a Frame by Frame basis.

```
if ( not this.bSwitchingScene ( ) )
then
```

Firstly, a simple check on the value of the User Variable "bSwitchingScene". If this is "false" then we continue, otherwise nothing happens.

The next section of code will NOT be used in our Game, so you can either delete it, or comment it out using "--" or "—[[ ]]", so you can always reintroduce it later:

```
    local underCursorObject = this.hUnderCursorObject ( )
    local lastUnderCursorObject = this.hLastUnderCursorObject ( )
    if ( underCursorObject )
    then
        if ( not lastUnderCursorObject )
        then
            this.sendMouseEnterEventToObject ( underCursorObject )
        elseif ( not object.isEqualTo ( underCursorObject, lastUnderCursorObject ) )
        then
            this.sendMouseLeaveEventToObject ( lastUnderCursorObject )
            this.sendMouseEnterEventToObject ( underCursorObject )
        end
    elseif ( lastUnderCursorObject )
    then
        this.sendMouseLeaveEventToObject ( lastUnderCursorObject )
    end
    this.hLastUnderCursorObject ( underCursorObject )
```

We need to keep the next few lines:

```
    if ( this.sNextScene ( ) ~= "" )
    then
        this.switchScene ( )
    end
end
```

These lines check to see if the "sNextScene" Variable contains a value (i.e.: it's not equal to ""). If it does, then "switchScene" is called, otherwise we can just continue without doing anything else.

onForwardEvent (sObjectTag, sAIModel, sHandler, vParam0, vParam1, vParam2, vParam3)

This Script is used to send Events to Objects using the various Parameters:

sObjectTag	The Tag of the Object to receive the Event
sAIModel	The AIModel of the Object
sHandler	The Handler of the AIModel (NB: Only Handlers can receive Events)
vParam0..3	The Parameters to be sent to the Handler (max. 4)

```
local s = application.getCurrentUserScene ( )
if ( s )
then
    local o = scene.getTaggedObject ( s, sObjectTag )
    if ( o )
    then
        object.sendEvent ( o, sAIModel, sHandler, sHandler, vParam0, vParam1, vParam2,
vParam3 )
    end
end
```

This Script gets a Handle on the current Scene, then gets a Handle to the Object with the given Tag and finally sends the Event to the Object.

onKeyboardKeyDown (kKeyCode)

This Script checks the Keyboard Table for any Objects that have registered to receive Events when a key is pressed, and sends the "onEventKeyDown" Event to the relevant Objects.

```
local t = this.tKeyDownEventTargetObjects ( )
```

This line gets a Handle to the LOCAL Table "tKeyDownEventTargetObjects".

```
if ( not table.isEmpty ( t ) )
then
```

These lines check if the Table is empty (or not), and if the Table contains entries then the following lines are executed:

```
    local iObjMax = table.getSize ( t )
```

This line gets the number of entries in the Table

```
    for iObj = 1, iObjMax, 2
    do
```

These lines set up a loop through the Table

```
        local hObj = table.getAt ( t, iObj - 1 )
```

This line retrieves the Object stored at the location currently specified by the above loop.

```
        if ( hObj and object.isActive ( hObj ) )
        then
            object.sendEvent ( hObj, table.getAt ( t, iObj ), "onEventKeyDown", kKeyCode )
        end
    end
end
```

These last few lines check that the Object retrieved from the Table is valid, and "Active", and sends it the "onEventKeyDown" Event with the value of the key that has been pressed.

onKeyboardKeyUp (kKeyCode)

This Script checks the Keyboard Table for any Objects that have registered to receive Events when a key is released, and sends the "onEventKeyUp" Event to the relevant Objects.

```
local t = this.tKeyUpEventTargetObjects ( )
if ( not table.isEmpty ( t ) )
then
    local iObjMax = table.getSize ( t )
    for iObj = 1, iObjMax, 2
    do
        local hObj = table.getAt ( t, iObj - 1 )
        if ( hObj and object.isActive ( hObj ) )
        then
            object.sendEvent ( hObj, table.getAt ( t, iObj ), "onEventKeyUp", kKeyCode )
        end
    end
end
```

This Script is identical to the "onKeyboardKeyDown" Handler, except that it sends the "onEventKeyUp" Event.

The following Handlers are all to do with the Mouse and, for the purposes of our Game, all code in them can be commented out:

onMouseButtonDown (nButton, nPointX, nPointY, nRayPntX, nRayPntY, nRayPntZ, nRayDirX, nRayDirY, nRayDirZ)

onMouseButtonUp (nButton, nPointX, nPointY, nRayPntX, nRayPntY, nRayPntZ, nRayDirX, nRayDirY, nRayDirZ)

onMouseMove (nButton, nPointX, nPointY, nRayPntX, nRayPntY, nRayPntZ, nRayDirX, nRayDirY, nRayDirZ)

onRegisterHUD (sInstanceID)

This is a simple one-liner:

```
table.add ( this.tHUDInstances ( ), sInstanceID )
```

This just adds the HUD instance, specified as the Parameter, to the LOCAL Table "tHUDInstances".

onRegisterObjectEvent (hObject, sAIModel, sEvent)

```
local bOK = false

if ( sEvent == "KeyDown" )
then
    table.add ( this.tKeyDownEventTargetObjects ( ), hObject  )
    table.add ( this.tKeyDownEventTargetObjects ( ), sAIModel )
    bOK = true
elseif ( sEvent == "KeyUp" )
then
    table.add ( this.tKeyUpEventTargetObjects ( ), hObject  )
    table.add ( this.tKeyUpEventTargetObjects ( ), sAIModel )
    bOK = true
```

The above lines add the Object and AIModel specified by the relevant Parameters to the "tXXXEventTargetObjects" (where XXX equates to the Event specified by the sEventParameter) Table.

```
--elseif ( sEvent == "MouseEnter" )
--then
    --table.add   ( this.tMouseEnterEventTargetObjects ( ), hObject  )
    --table.add   ( this.tMouseEnterEventTargetObjects ( ), sAIModel )
    --bOK = true
--elseif ( sEvent == "MouseLeave" )
--then
    --table.add   ( this.tMouseLeaveEventTargetObjects ( ), hObject  )
    --table.add   ( this.tMouseLeaveEventTargetObjects ( ), sAIModel )
    --bOK = true
--elseif ( sEvent == "MouseClick" )
--then
    --table.add   ( this.tMouseClickEventTargetObjects ( ), hObject  )
    --table.add   ( this.tMouseClickEventTargetObjects ( ), sAIModel )
    --bOK = true
```

The above lines are identical to those above, but, since they relate to Mouse Events, they can be commented out.

```
end

if ( bOK )
then
    log.message ( "HLD : Registered event ", sEvent, " for object ", hObject, " with AIModel
", sAIModel )
else
    log.warning ( "HLD : Failed registering event ", sEvent, " for object ", hObject, " with
AIModel ", sAIModel )
end
```

The last few lines check "bOK" for success and write a Log Message accordingly.

onSetNextScene (sScene)

Another simple Script that checks if the current value of the LOCAL Variable "sNextScene" is "", and if so, then it is set to the value of the Parameter "sScene".

```
if ( this.sNextScene ( ) == "" )
then
    this.sNextScene ( sScene )
end
```

onSwitchScene ( )

This Script is called whenever the Scene is changed.

```
local s = this.sNextScene ( )
this.sNextScene ( "" )
```

These couple of lines set the LOCAL Variable "sNextScene" to equal "".

this.destroyRegisteredHUDInstances ( )

This line calls the "destroyRegisteredHUDInstances" Function.

application.setCurrentUserScene (s)

This line calls the Application Function "setCurrentUserScene", passing the Parameter "s", to change the current Scene.

this.findAndUseStartupCamera ( )

This line calls the "findAndUseStartupCamera" Function.

```
local hUser = this.getUser ( )

hud.callAction ( hUser, "HLDFadeScreenIn" )
```

And, this line calls the "HLDFadeScreenIn" Action that is associated with the current User HUD, via a Handle to the current User.

```
local hScene = application.getCurrentUserScene ( )

music.play ( hScene, 5, 1 )
music.setVolume ( hScene, 0.25, 0 )
```

Ok, now we have a few lines of mine. All that these lines do is get a Handle to the current Scene, and use this Handle to set some Music to play at a specified volume.

this.bSwitchingScene (false)

Finally, we set the LOCAL Variable "bSwitchingScene" to "false", to indicate that we are not currently changing Scenes.

onUnregisterHUD (sInstanceID)

Another simple Script that removes a registered HUD from the "tHUDInstances" Table.

```
local t= this.tHUDInstances ( )
local iMax = table.getSize ( t )

for i = 0, iMax - 1
do
    if ( sInstanceID == table.getAt ( t, i ) )
    then
        table.removeAt ( t, i )
        break
    end
end
```

The only thing I will mention here, is the lines for the final "if" statement. These lines check that the "sInstanceID" Parameter exists in the Table, and if so, removes it from the Table. The "break" command stops the "do" loop from executing to save time (since every millisecond counts in a 3D Game!).

onUnregisterObject (hObject)

This is a blank Script that is provided "just in case" anyone wants to unregister an Object. You'll sometimes find that blank Scripts are included in libraries, mainly as a suggestion that this Function/Handler may be useful at some point.

Next, we'll move on to the "HLDMain" Functions.

buildStartupSequence ( )

This Script creates the startup sequence based on the Environment Variables that we entered earlier.

```
local hUser = this.getUser ( )
local startupSequence = hud.newAction ( hUser, "HLDMainStartupSequence" )
```

The first lines create the new HUD Action "HLDMainStartupSequence".

```
for i = 0, 9
do
    local text = application.getCurrentUserEnvironmentVariable ( "HLDStartupText"..i )
    local logo = application.getCurrentUserEnvironmentVariable ( "HLDStartupLogo"..i )
```

These next few lines start the loop (this is where you can change the maximum number of Screens to be displayed before the Game starts, by changing the "9" in the "for" statement), and assign the values of the Environment Variables to the LOCAL Variables "text" and "logo". Notice that the Environment Variables MUST be in the format specified earlier, or both Variables will be set to a "nil" value.

```
    if ( text )
    then
        local h = hud.newComponent ( this.getUser ( ), hud.kComponentTypeLabel )
```

OK, if "text" is anything other than "nil", then we create a new HUD Label Component.

```
        if ( h )
        then
            hud.setComponentPosition ( h, 50, 50 )
            hud.setComponentSize ( h, 90, 50 )
            hud.setComponentBackgroundColor ( h, 0, 0, 0, 255 )
            hud.setComponentBorderColor ( h, 0, 0, 0, 0 )
            hud.setComponentForegroundColor ( h, 127, 127, 127, 255 )
            hud.setComponentVisible ( h, true )
            hud.setComponentOpacity ( h, 0 )
            hud.setComponentZOrder ( h, 255 )
            hud.setLabelText ( h, text )
            hud.setLabelTextHeight ( h, 25 )
            hud.setLabelTextAlignment ( h, hud.kAlignCenter, hud.kAlignCenter )
```

Assuming that our HUD Component is created, we then set the relevant Attributes of the Component. For full information on these Attributes, see the API documents on SDN.

```
            hud.beginActionCommand ( startupSequence,
hud.kCommandTypeInterpolateOpacity )
            hud.pushActionCommandArgument ( startupSequence, h )
            hud.pushActionCommandArgument ( startupSequence, 255 )
            hud.pushActionCommandArgument ( startupSequence,
hud.kInterpolatorTypeLinear )
            hud.pushActionCommandArgument ( startupSequence, 1000 )
            hud.endActionCommand ( startupSequence )
            hud.beginActionCommand ( startupSequence, hud.kCommandTypeSleep )
            hud.pushActionCommandArgument ( start-upsequence, 2000 )
            hud.endActionCommand ( startupSequence )
            hud.beginActionCommand ( startupSequence,
hud.kCommandTypeInterpolateOpacity )
            hud.pushActionCommandArgument ( startupSequence, h )
            hud.pushActionCommandArgument ( startupSequence, 0 )
            hud.pushActionCommandArgument ( startupSequence,
hud.kInterpolatorTypeLinear )
            hud.pushActionCommandArgument ( startupSequence, 1500 )
            hud.endActionCommand ( startupSequence )
            hud.beginActionCommand ( startupSequence, hud.kCommandTypeSleep )
            hud.pushActionCommandArgument ( startupSequence, 1000 )
            hud.endActionCommand ( startupSequence )
            hud.beginActionCommand ( startupSequence, hud.kCommandTypeSetVisible )
            hud.pushActionCommandArgument ( startupSequence, h )
            hud.pushActionCommandArgument ( startupSequence, false )
            hud.endActionCommand ( startupSequence )
        end
```

The above lines may look a bit nightmarish, but in reality they all follow a certain pattern. Each group starts with a "beginActionCommand" line which specifies the Action Command to be executed (see the API for full details on each one), the first of which specifies the following Constant:

"kCommandTypeInterpolateOpacity"

This Constant sets the Action to be one of Interpolating (i.e.: changing) the Opacity of the Component. The next few lines specify the Arguments for the selected Constant (in this case values of "255", "kInterpolator TypeLinear", and "1000"). All these Arguments taken together mean that the first Action is to change the Opacity of the Component from 0 to 255, in a linear way, over a period of 1000 milliseconds.

The remaining Actions are:

1) Sleep for 2000 milliseconds
2) Change the Opacity of the Component back to 0, in a linear way, over a period of 1500 milliseconds
3) Sleep for 1000 milliseconds
4) Make the Component invisible

Taken together, the Component will gradually fade in, stay on the Screen for a while, and then gradually fade out.

```
    elseif ( logo )
    then
        local h = hud.newComponent ( hUser, hud.kComponentTypeContainer )
        if ( h )
        then
            hud.setComponentPosition ( h, 50, 50 )
            hud.setComponentSize ( h, 50, 50 )
            hud.setComponentAspectInvariant ( h, true )
            hud.setComponentBackgroundImage ( h, logo )
            hud.setComponentBackgroundColor ( h, 127, 127, 127, 255 )
            hud.setComponentBorderColor ( h, 0, 0, 0, 0 )
            hud.setComponentVisible ( h, true )
            hud.setComponentOpacity ( h, 0 )
            hud.setComponentZOrder ( h, 255 )
            hud.beginActionCommand ( startupSequence,
hud.kCommandTypeInterpolateOpacity )
            hud.pushActionCommandArgument ( startupSequence, h )
            hud.pushActionCommandArgument ( startupSequence, 255 )
            hud.pushActionCommandArgument ( startupSequence,
hud.kInterpolatorTypeLinear )
            hud.pushActionCommandArgument ( startupSequence, 1000 )
            hud.endActionCommand ( startupSequence )
            hud.beginActionCommand ( startupSequence, hud.kCommandTypeSleep )
            hud.pushActionCommandArgument ( startupSequence, 2000 )
            hud.endActionCommand ( startupSequence )
            hud.beginActionCommand ( startupSequence,
hud.kCommandTypeInterpolateOpacity )
```

```
            hud.pushActionCommandArgument ( startupSequence, h )
            hud.pushActionCommandArgument ( startupSequence, 0 )
            hud.pushActionCommandArgument ( startupSequence,
hud.kInterpolatorTypeLinear )
            hud.pushActionCommandArgument ( startupSequence, 1500 )
            hud.endActionCommand ( startupSequence )
            hud.beginActionCommand ( startupSequence, hud.kCommandTypeSleep )
            hud.pushActionCommandArgument ( startupSequence, 1000 )
            hud.endActionCommand ( startupSequence )
            hud.beginActionCommand ( startupSequence,
hud.kCommandTypeSetBackgroundImage )
            hud.pushActionCommandArgument ( startupSequence, h )
            hud.pushActionCommandArgument ( startupSequence, "" )
            hud.endActionCommand ( startupSequence )
            hud.beginActionCommand ( startupSequence, hud.kCommandTypeSetVisible )
            hud.pushActionCommandArgument ( startupSequence, h )
            hud.pushActionCommandArgument ( startupSequence, false )
            hud.endActionCommand ( startupSequence )
        end
```

All of the above lines of this Script basically do the same as explained above, but for a Logo.

```
    else
        break
    end
end
```

These lines end the Loop, specifically when there is a "nil" value for both "text" and "logo" (the "break" statement).

```
hud.beginActionCommand ( startupSequence, hud.kCommandTypeSendEventToUser )
hud.pushActionCommandRuntimeArgument ( startupSequence,
hud.kRuntimeValueCurrentUser )
hud.pushActionCommandArgument ( startupSequence, "HLDMain" )
hud.pushActionCommandArgument ( startupSequence, "onSetNextScene" )
hud.pushActionCommandArgument ( startupSequence, this.sStartupScene ( ) )
hud.endActionCommand ( startupSequence )
```

This next section sets the final Action Command to be a call to the "onSetNextScene" Handler of "HLDMain", with a Parameter of the Scene that is stored in the LOCAL Variable "sStartupScene". Notice that this Command will be called irrespective of whether the Variables "text" or "logo" contain a value.

```
hud.callAction ( hUser, "HLDMainStartupSequence" )
```

Finally, a call is made to start the HLDMainStartupSequence" Action that we have just created.

destroyRegisteredHUDInstances ( )

```
local t= this.tHUDInstances ( )
local iMax = table.getSize ( t )

for i = 0, iMax - 1
do
    hud.destroyTemplateInstance ( this.getUser ( ), table.getAt ( t, i ) )
end

table.empty ( t )
```

You've seen most of the above before, so I'll only point out the line in the middle of the loop. All this line does is get rid of the Instance of the HUD Template whose Tag (name) has been stored in the "tHUDInstances" Table.

findAndUseStartupCamera ( )

```
    local s = application.getCurrentUserScene ( )
    if ( s )
    then
        local startup = scene.getTaggedObject ( s, "HLDStartupCamera" )
        if ( startup )
        then
            application.setCurrentUserActiveCamera ( startup )
        else
            --
        end
    end
```

Another fairly simple Script that gets the current Scene, checks if there is a Tagged Object called "HLDStartupCamera", and if there is then sets it as the currently active User Camera.

The code in the following Scripts can either be deleted, or commented out as we will not be using the Mouse:

findUnderCursorObject (nRayPntX, nRayPntY, nRayPntZ, nRayDirX, nRayDirY, nRayDirZ)

sendMouseClickEventToObject (hObject, nButton)

sendMouseEnterEventToObject (hObject)

sendMouseLeaveEventToObject (hObject)

switchScene ( )

this.bSwitchingScene (true)

Nice easy first line! All we are doing here is setting the LOCAL Variable "bSwitchingScene" to be "true" as we are in the process of switching Scenes.

```
table.empty ( this.tKeyDownEventTargetObjects( ) )
table.empty ( this.tKeyUpEventTargetObjects ( ) )
--table.empty ( this.tMouseClickEventTargetObjects ( ) )
--table.empty ( this.tMouseEnterEventTargetObjects ( ) )
--table.empty ( this.tMouseLeaveEventTargetObjects ( ) )
--this.hLastUnderCursorObject ( nil )
--this.hUnderCursorObject ( nil )
```

The above lines empty the contents of the various LOCAL Tables and set two Variables to equal "nil". This is done to ensure that no stray key presses or Mouse Events will be sent to the new Scene. As we are not using the Mouse, you can either delete, or comment out (as I have done) the last 5 lines.

```
local hUser = this.getUser ( )

hud.stopAction ( hUser, "HLDFadeScreenIn"  )
hud.stopAction ( hUser, "HLDFadeScreenOut" )
hud.callAction ( hUser, "HLDFadeScreenOut" )
this.postEvent ( 1.1, "onSwitchScene" )
```

These last few lines send "stopAction" Events to stop any currently active Screen Fading. We then proceed to send an Event to call the "HLDFadeScreenOut" Function (to fade out the Screen), and finally send an Event to call the "onSwitchScene" Handler. Notice that this last line has a value of "1.1" BEFORE the Handler name. This value is used to delay the sending of the Event, in this case allowing the Screen to fade out before calling the "onSwitchScene" Function. This ability to delay the sending of Events is very useful if you need to allow something to stop running before something else starts.

Well, that's it for the "HLDMain" AIModel. We'll now move on to the "HLDVisitCamera" AIModel.

HLDVisitCamera

"HLDVisitCamera", as previously stated is the moving part of the Game that can be controlled by the User. "HLDVisitCamera" consists of the following Scripts (as you can see I have created several new ones for various aspects of our Game – again these are marked in red):

VARIABLES:

bGoBackward	Type – Boolean Initial Value – false This is used for movement.
bGoForward	Type – Boolean Initial Value – false This is used for movement.
bLookDown	Type – Boolean Initial Value – false This is used for looking down.
bLookUp	Type – Boolean Initial Value – false This is used for looking up.
bMusic	Type – Boolean Initial Value – true This is used for Music.
bRotateLeft	Type – Boolean Initial Value – false This is used for movement.
bRotateRight	Type – Boolean Initial Value – false This is used for movement.
bStart	Type – Boolean Initial Value – false This is used to determine if the Game has started.
bStrafeLeft	Type – Boolean Initial Value – false This is used for movement.
bStrafeRight	Type – Boolean Initial Value – false This is used for movement.

hDino	Type – Object Initial Value – 'nil' This is used for the current dino.
hDynObject	Type – Object Initial Value – 'nil' This is used for the current Dynamic Object.
hWeapon	Type – Object Initial Value – 'nil' This is used for the current weapon.
nAmmo	Type – Number Initial Value – 100 This is used for the amount of ammo left.
nCurAngleH	Type – Number Initial Value – 0 This is used for the current horizontal angle of the Object.
nCurAngleV	Type – Number Initial Value – 0 This is used for the current vertical angle of the Object.
nCurHeight	Type – Number Initial Value – 1.5 This is used for the current height of the Object.
nCurWeaponBackFactor	Type – Number Initial Value – 0 This is used for the current "Back Factor" of the weapon.
nDstAngleH	Type – Number Initial Value – 0 This is used for the destination horizontal angle of the Object.
nDstAngleV	Type – Number Initial Value – 0 This is used for the destination vertical angle of the Object.
nDstHeight	Type – Number Initial Value – 1.5 This is used for the destination height of the Object.

nDstWeaponBackFactor	Type – Number Initial Value – 0 This is used for the destination "Back Factor" of the weapon.
nRadarTime	Type – Number Initial Value – 0 This is used for updating the radar.
nScore	Type – Number Initial Value – 0 This is used for the current score counter.
nTime	Type – Number Initial Value – 0 This is used for the current time counter.

FUNCTIONS:

addDino	This Function adds the dinosaurs to the Scene.
atan2	This Function is used to replicate the mathematical Function atan2.
centerMouse	This Function centres the Mouse Cursor on the Screen.
createDynObject	This Function creates a Dynamic (i.e.: Physics enabled) Object.
createHUD	This Function creates the main HUD for the Game.
createIntroHUD	This Function creates the HUD that will be displayed to enable the Player to start, or quit, the Game.
gameOver	This Function is called to end the Game.
registerTriggers	This Function registers the Events, with "HLDMain", that can be used by this AIModel.
updateRadar	This Function updates the on-screen radar.
updateWeapon	This Function updates the positioning of the weapon.

vector2Rotation	This Function calculates a Rotation based on two Vectors – Thanks to FoolishFrost for this one.

HANDLERS:

onDinoCollision	This Handler is called when the Player collides with a dinosaur.
onEnterFrame	This Handler is called once every Frame.
onEventKeyDown	This Handler is called by an Event being sent to "HLDVisitCamera" that indicates that a Keyboard key has been pressed.
onEventKeyUp	This Handler is called by an Event being sent to "HLDVisitCamera" that indicates that a Keyboard key has been released.
onForwardEvent	This Handler sends an Event to an Object, allowing the passing of up to 4 Parameters.
onInit	This Handler is called on initialization of "HLDVisitCamera".
onMouseButtonDown	This Handler is called when the Mouse Button is pressed.
onMouseButtonUp	This Handler is called when the Mouse Button has been released.
onUpdateScore	This Handler is called whenever the score needs to be updated.
onWeaponChange	This Handler is called whenever the weapon needs to be changed.
onWeaponShoot	This Handler is called whenever the Player fires the weapon.

onInit ( )

Once again, we'll start with the "onInit" Handler.

```
local hScene = application.getCurrentUserScene ( )

this.bMusic ( true )
music.play ( hScene, 2, 5 )
music.setVolume ( hScene, 0.5, 0 )
```

The first three lines are pretty straightforward. We get a Handle to the current Scene, start playing some Music (number 2 in the Ambience Editor list), and set the required Volume of the Music (not too loud I hope!). Note that the Music will be slowly faded in according to the value of the third Parameter of "music.play" ("3" in this case meaning 3 seconds).

```
this.registerTriggers ( )
```

This line simply calls the "registerTriggers" Function.

```
this.createIntroHUD ( )
```

The last line calls the "createIntroHUD" Function.

Note that we have deleted the call to "createDynObject" as this will be dealt with in another Function.

onDinoCollision (sTag)

```
this.gameOver ( "collision" )
```

This Handler simply calls the "gameOver" Function, passing a Parameter indicating a "collision". I have included the Parameter "sTag" in the call to this Handler so it can be used in the future for differentiating between collisions with different types of dinosaur.

onEnterFrame ( )

This Function is called each Frame, and is where the movement of the Camera, if any, takes place. It is a fairly long Script, but it does contain some very important concepts (not to mention 3D maths!).

```
if ( application.getCurrentUserEnvironmentVariable ( "gameOn" ) )
then
```

Firstly we check to see if the Environment Variable "gameOn" is "true". If it isn't then we exit the Handler without doing anything, as the Game hasn't yet started.

```
    if ( not this.bStart ( ) )
    then
        this.createHUD ( )
        this.bStart ( true )
    else
```

Next, we check to see if the LOCAL Variable "bStart" is "false" (actually we are checking that it is NOT "true"!). This check is only useful the first time that we enter the Handler (when "gameOn" is "true" of course!), since we set "bStart" to "true" in the body of the "if" statement. We also call the Function "createHUD" to generate the HUD we require for the Game.

```
        local o = this.getObject ( )
        local dynObject = this.hDynObject ( )
        local dt = application.getLastFrameTime ( )
        local s = application.getCurrentUserScene ( )
```

These lines create a few LOCAL Variables that we will use later on in this Handler. The most important one to understand is "dt", since this will be used to make our movement frame rate independent (i.e.: our movement will be the same at 100 fps as it will at 30 fps!)

```
        if ( this.bRotateRight ( ) )
        then
            this.nDstAngleH ( this.nDstAngleH ( ) - 30 * dt )
        elseif ( this.bRotateLeft ( ) )
        then
            this.nDstAngleH ( this.nDstAngleH ( ) + 30 * dt )
        end
```

Ok, here's some maths for you, and the first introduction to "dt". All that these lines are doing, are checking if there is any Rotation, either right ("bRotateRight") or left ("bRotateLeft"), and if there is then the horizontal destination angle ("nDstAngleH") is moved by 30 degrees multiplied by the value of "dt" in the corresponding direction (NB: right is negative).

```
        if ( this.bLookUp ( ) )
        then
            this.nDstAngleV ( this.nDstAngleV ( ) + 30 * dt )
        elseif ( this.bLookDown ( ) )
        then
            this.nDstAngleV ( this.nDstAngleV ( ) - 30 * dt )
        end
```

```
this.nDstAngleV ( math.clamp ( this.nDstAngleV ( ), -75, 85 )
```

These lines are very similar to those above, but deal with Rotation in the vertical direction (i.e.: up and down). The main difference here is the last line which stops the Rotation from going beyond "-75" degrees and "85" degrees. This is important since we don't want to be able to rotate our view outside the boundaries of normal Human movement, do we?

```
local rh = math.interpolate ( this.nCurAngleH ( ), this.nDstAngleH ( ), 10 * dt )
local rv = math.interpolate ( this.nCurAngleV ( ), this.nDstAngleV ( ), 10 * dt )
local h  = math.interpolate ( this.nCurHeight ( ), this.nDstHeight ( ), 2 * dt )
```

Now all the Script is doing is interpolating (calculating a value between two known values) the horizontal destination, the vertical destination, and the height destination, again using a multiplier and "dt".

```
this.nCurAngleH ( rh )
this.nCurAngleV ( rv )
this.nCurHeight ( h )
```

These lines just save the values calculated above as the current LOCAL values.

```
object.setRotation ( o, 0, rh, 0, object.kGlobalSpace )
object.rotate ( o, rv, 0, 0, object.kLocalSpace )
```

Now the Object is rotated according to the horizontal and vertical values. Note that the horizontal Rotation occurs in GLOBAL space, and that the vertical rotation is only in LOCAL space. This is so that the Object doesn't move upwards, after all it's not meant to be flying!

```
object.matchTranslation ( o, dynObject, object.kGlobalSpace )
object.translate ( o, 0, h, 0, object.kGlobalSpace )
```

Next the Camera is placed near to the Dynamic Object, with just a small offset, so that it can mimic the Dynamic properties of the Dynamic Object.

```
local oXx, oXy, oXz = object.getXAxis ( o, object.kGlobalSpace )
local oYx, oYy, oYz = object.getYAxis ( o, object.kGlobalSpace )
local oZx, oZy, oZz = object.getZAxis ( o, object.kGlobalSpace )
local fx, fy, fz = 0, 0, 0
```

We are coming up to some more maths, but first we need to create some LOCAL Variables to use in the Vector calculations. These Variables are for the current values of the X, Y and Z axes, and x, y and z values that will be used to calculate the Forces required to move the Dynamic Object.

```
if ( this.bGoForward ( ) )
then
   fx, fy, fz = math.vectorSubtract ( fx, fy, fz, oZx, oZy, oZz )
elseif ( this.bGoBackward ( ) )
then
   fx, fy, fz = math.vectorAdd ( fx, fy, fz, oZx, oZy, oZz )
end
```

Ok, now for some forwards, or backwards movement. This is achieved by using Vector maths, where forward movement is carried out by subtracting the Z axis values from the dummy values, and backward movement is carried out by adding the values.

```
if ( this.bStrafeRight ( ) )
then
    fx, fy, fz = math.vectorAdd ( fx, fy, fz, oXx, oXy, oXz )
elseif ( this.bStrafeLeft ( ) )
then
    fx, fy, fz = math.vectorSubtract ( fx, fy, fz, oXx, oXy, oXz )
end
```

Strafing (i.e.: moving left or right, WITHOUT turning) is carried out in a similar way, with adding being used to go to the right, and subtraction to the left, using the X axis values. Note that the effects of the above movements are cumulative, with forwards/backwards movement being calculated first, and then left/right movement being added, or subtracted, from the forwards/backwards values. Note that this is NOT used in our Game, and so the above lines can be deleted, or commented out.

```
fx, fy, fz = math.vectorNormalize ( fx, 0, fz )
fx, fy, fz = math.vectorScale ( fx, 0, fz, 5000 )
```

To be able to use the calculated values, they have to be normalised (i.e.: change them to values that lie between 0 and 1), and scaled to ensure that they fit the Object. Note that the Y value is ignored, as our Object is not moving vertically.

```
dynamics.addForce ( dynObject, fx, fy, fz, object.kGlobalSpace )
```

Almost there! The Forces calculated above are now added to the Dynamic Object, in GLOBAL space.

```
dynamics.addForce ( dynObject, 0, -2000, 0, object.kGlobalSpace )
```

And finally, Gravity is added to the Dynamic Object, to ensure that it stays on the ground, or at least comes back down to the ground if it's in the air!

```
    this.updateWeapon ( )
end
```

Now, we make a call to the "updateWeapon" Function.

```
this.nTime ( this.nTime ( ) + dt )
```

This line simply increases the LOCAL Variable "nTime" by the value of "dt".

```
if ( this.nTime ( ) >= 900 )
then
    this.gameOver ( "time" )
```

The code above checks the value of "nTime", and if it is greater than, or equal to, "900" seconds (i.e.: 15 minutes – 900/60 = 15) then we make a call to the "gameOver" Function, showing that the Player has run out of time. Obviously this value can be changed if you find that the Game is too hard (or easy!) to complete in that time frame!

```
else
    if ( this.nTime ( ) > 300 and this.nTime ( ) < 301 and this.bMusic ( ) )
    then
        music.stop ( s, 0.75 )
        music.play ( s, 1, 0.75 )
        music.setVolume ( s, 0.5, 0.75 )
        this.bMusic ( false )
    elseif ( this.nTime ( ) > 301 and this.nTime ( ) < 302 and not this.bMusic ( ) )
    then
        this.bMusic ( true )
    elseif ( this.nTime ( ) > 600 and this.nTime ( ) < 601 and this.bMusic ( ) )
    then
        music.stop ( s, 0.75 )
        music.play ( s, 0, 0.75 )
        music.setVolume ( s, 0.5, 0.75 )
        this.bMusic ( false )
    elseif ( this.nTime ( ) > 601 and this.nTime ( ) < 602 and not this.bMusic ( ) )
    then
        this.bMusic ( true )
    end
```

OK, I'll quickly explain these lines, since all they are doing is checking if "nTime" is over, or equal to, a certain value, and if it is then we stop whatever Music is currently playing, and start a new piece. I've included this to show how easy it is to change the currently playing Music during the Game, and also because I think it improves the playing experience if you have different background Music at different times! Note that I only stop the Music if the value of "this.bMusic" is "true".

```
local hUser = application.getCurrentUser ( )
```

Here we get a handle to the current User.

```
if ( this.nTime ( ) > this.nRadarTime ( ) + 2 )
then
    local nCount = sceneGetTaggedObjectCount ( s )
    local sTag = ""
    local hTemp
    local nDist = 0
    local nDiff = 0
```

Next we check if we need to update the radar (every 2 seconds here), and set up our LOCAL Variables.

These LOCAL Variables include one that gets the count of the number of "Tagged" Objects in the Scene. This can be very handy, as you will soon see.

```
    user.sendEvent ( hUser, "HLDMainHUD", "onClearRadar" )
```

Before doing anything else, we send an Event to "HLDMainHUD" to clear the radar. This will make the radar appear to "blink".

Now, I know that this may look a bit daunting, but it's not too bad really! All that's happening is that we use the count of "Tagged" Objects ("nCount") to loop through all the "Tagged" Objects. Note that the Index used for "Tagged" Objects is zero-based, hence the "- 1" in the "for" statement.

```
for nIndex = 0, nCount – 1
do
```

We then get a Handle to the "Tagged" Object, and the "Tag" of the Object.

```
hTemp = scene.getTaggedObjectAt ( s, nIndex )
sTag = scene.getTaggedObjectTag ( s, nIndex )
```

Next, we check that the Handle was created successfully, and that the Object's "Tag" was retrieved successfully.

```
if ( hTemp ~= nil )
then
   if ( sTag ~= nil or sTag ~= "" )
   then
```

OK, time for the fun bit! Since the "Tag" is a String we can use ShiVa's String handling Functions to determine if the "Tag" begins with "Dino", and then to see if the last letter of the String is a ":D", which denotes one of our Dynamic Objects (not one of our dinos!). In the "addDino" Function you'll see that our dinos are "Tagged" as "DinoXN", and our Dynamic Objects are "Tagged" as "DinoXND".

```
if ( string.getSubString ( sTag, 0, 4 ) == "Dino" )
then
   if ( string.getSubString ( sTag, string.getLength ( sTag ) – 1, 1 ) ~= "D" )
   then
```

At this point, we have a Handle to our dino, and it's "Tag", and we need to find the distance between the Player and the dino, and also the angle between them. This is what these next lines do.

Here we use a handy ShiVa Function to find the distance between two Objects:

```
nDist = object.getDistanceToObject ( o, hTemp )
```

We then check if the distance is less than 200 (this is the distance I have picked for this demo Game) and, if so, we call the "vector2Rotation" Function, which is used to convert the Vectors of two Objects to a Rotation (Many thanks to FoolishFrost for posting this code on the WIKI).

```
if ( nDist < 200 )
then
   fx, fy, fz, nDiff = this.vector2Rotation ( o, hTemp )
```

Next, we need to call our "atan2" Function to calculate the angle (in degrees) of the Vector.

```
nDiff = this.atan2 ( fz, fx )
```

Now we use another ShiVa Function to get the Direction Vector of the Player and, again, make a call to our "atan2" Function to return the relevant angle, and subtract the returned value from the previously calculated angle. This is done to ensure that we obtain the correct angle between the two Objects.

```
fx, fy, fz = object.getDirection ( o, object.kGlobalSpace )
nDiff = nDiff – this.atan2 ( fz, fx )
```

Sometimes, the result calculated above will be negative. As we need to only use positive values (0 to 360), we need to add 360 to any negative value.

```
if ( nDiff < 0 )
then
    nDiff = 360 + nDiff
end
```

Finally, for this bit, we call the "updateRadar" Function passing the distance and angle created above.

```
                    this.updateRadar ( nDist, nDiff )
                end
            end
        end
    end
else
    log.message ( "Tagged Object not found in HLDVisitCamera.onEnterFrame" )
end
end
end

user.sendEvent ( hUser, "HLDMainHUD", "onTimeSetValue", this.nTime ( ) )
user.sendEvent ( hUser, "HLDMainHUD", "onScoreSetValue", this.nScore ( ) )
end
end
```

Since the last couple of lines (excluding the "end" statements) are so similar, all I'll say on them is that we now send two Events to "HLDMainHUD", one to update the time value in the HUD, and the other to update the score value.

onEventKeyDown (kKey)

This is a fairly straightforward Handler, as it checks which key has been pressed via the "kKey" Parameter, and then sets the relevant LOCAL Variable to indicate which direction to move. The only exception to this is when the key that has been pressed is the "kKeySpace" (the spacebar), when an Event is sent to the "onWeaponShoot" Handler.

```
if ( nKey == input.kKeyUp )
then
    this.bGoForward ( true )
elseif ( nKey == input.kKeyDown )
then
    this.bGoBackward ( true )
elseif ( nKey == input.kKeyRight )
then
    this.bRotateRight ( true )
elseif ( nKey == input.kKeyLeft )
then
    this.bRotateLeft ( true )
elseif ( nKey == input.kKeySpace )
then
    this.sendEvent ( "onWeaponShoot" )
elseif ( nKey == input.kKeyQ )
then
    this.bLookUp ( true )
elseif ( nKey == input.kKeyA )
then
    this.bLookDown ( true )
end
```

The keys that I have chosen are:

Up Arrow	-	Move Forward
Down Arrow	-	Move Back
Left Arrow	-	Turn Left
Right Arrow	-	Turn Right
Spacebar	-	Shoot
Q	-	Look Up
A	-	Look Down

onEventKeyUp (kKey)

This Handler is identical to the "onEventKeyDown" Handler, except that it sets the relevant direction to "false" (i.e.: don't keep moving in that direction), and also doesn't have a line for the "shoot" key (as it's not needed!).

```
if ( nKey == input.kKeyUp )
then
    this.bGoForward ( false )
elseif ( nKey == input.kKeyDown )
then
    this.bGoBackward ( false )
elseif ( nKey == input.kKeyRight )
then
    this.bRotateRight ( false )
elseif ( nKey == input.kKeyLeft )
then
    this.bRotateLeft ( false )
elseif ( nKey == input.kKeyQ )
then
    this.bLookUp ( false )
elseif ( nKey == input.kKeyA )
then
    this.bLookDown ( false )
end
```

onMouseButtonDown (nButton, nPointX, nPointY, nRayPntX, nRayPntY, nRayPntZ, nRayDirX, nRayDirY, nRayDirZ)

onMouseButtonUp (nButton, nPointX, nPointY, nRayPntX, nRayPntY, nRayPntZ, nRayDirX, nRayDirY, nRayDirZ)

The two Mouse Handlers can have their code either deleted, or commented out, as we will not be using the Mouse.

onUpdateScore (n)

This Handler updates the LOCAL Variable "nScore" by adding the Parameter "n", and then checks to see if the Player has won the Game (i.e.: with a score greater than 1000 points) and, if they have, calls the "gameOver" Function passing the String "score" as a Parameter.

```
this.nScore ( this.nScore ( ) + n )

if ( this.nScore ( ) >= 1000 )
then
    this.gameOver ( "score" )
end
```

onWeaponChange (sWeapon)

This Handler is used to change the current weapon that the Player is using. In our Game, there is only one weapon, so this is only called once. However, it is here to enable the changing of weapons at any time during the Game.

```
local w = this.hWeapon ( )

if ( w == nil )
then
```

The first few lines get the value of the LOCAL Variable "hWeapon", and check to see if it is equal to "nil". If it is then the rest of the code is executed, otherwise nothing happens, so if you were to implement a weapon changing system then you would need to delete the above "if" statement, and allow all calls to this Handler to change the weapon according to the "sWeapon" Parameter.

```
    local o = this.getObject ( )
    local s = object.getScene ( o )

    if ( s ~= nil )
    then
        w = scene.createRuntimeObject ( s, sWeapon )
```

OK, these few lines should be fairly straightforward, as all we are doing is getting a Handle to the current Object, getting a Handle to the current Scene, checking that the Scene exists, and finally creating a new Runtime Object (you would have to add some code to delete the previous Object if you are implementing a weapon changing system).

```
        if ( w ~= nil )
        then
            this.hWeapon ( w )
        end
    end
end
```

All that this last bit does is check that the new weapon has been created and, if so, set the LOCAL Variable "hWeapon" to the value of the new weapon.

onWeaponShoot ( )

This Handler fires the weapon!

```
local ammo = this.nAmmo ( )
```

Firstly, we get the value of the LOCAL Variable "nAmmo".

```
if ( ammo > 0 )
then
```

Then we check to see if it's greater than "0". If it is, we continue:

```
    this.nDstWeaponBackFactor ( 1 )
    this.nAmmo ( ammo - 1 )

    local hUser = application.getCurrentUser ( )

    user.sendEvent ( hUser, "HLDMainHUD", "onAmmoSetValue", ammo - 1 )
```

Firstly, we set the LOCAL Variables "nDstWeaponBackFactor" (to "1" – this is to make the gun appear to move backwards when we shoot, similar to a normal weapon's recoil) & "nAmmo" (to "nAmmo" – 1 – well we have fired a bullet!), and finally we send an Event to "HLDMainHUD" to update the ammo value in the HUD.

```
    local o = this.getObject ( )
    local s = object.getScene ( o )
    local w = this.hWeapon ( )
```

Now we create some more LOCAL Variables that are filled with Handles to the current Object, Scene and weapon.

```
    if ( s ~= nil and w ~= nil )
    then

        local bullet = scene.createRuntimeObject ( s, "Weapon0Bullet" )
```

Next we make a simple check to ensure that the Scene and weapon exist and, if all is well, we create a new Runtime Object "Weapon0Bullet".

```
        if ( bullet ~= nil )
        then
            local srcX, srcY, srcZ = object.getTranslation ( w, object.kGlobalSpace )
            local dirX, dirY, dirZ = object.getDirection ( w, object.kGlobalSpace )
            local dstX, dstY, dstZ = object.getDirection ( o, object.kGlobalSpace )
            local camX, camY, camZ = object.getTranslation ( o, object.kGlobalSpace )
```

Now we check that the bullet was created ok and, if so, we get the values of the Translation of the weapon (srcX etc.), the direction the weapon is pointing (dirX etc.), the direction of the Object (dstX etc. - Destination), and the Translation of the Object (camX etc.).

Then we move on to some more maths! (This maths will be explained in more detail in the "updateWeapon" Function).

```
dstX, dstY, dstZ = math.vectorScale ( dstX, dstY, dstZ, 100 )
dstX, dstY, dstZ = math.vectorAdd ( camX, camY, camZ, dstX, dstY, dstZ )
dirX, dirY, dirZ = math.vectorScale ( dirX, dirY, dirZ, 2 )
srcX, srcY, srcZ = math.vectorAdd ( srcX, srcY, srcZ, dirX, dirY, dirZ )
```

Firstly, we scale the destination values by "100" (so they are in the distance). We then add the Translation of the Object to these values (so the bullet will go the right way). Next we scale the direction values by "2" (the end of the barrel of the gun), and add these values to the current Translation of the weapon (to ensure we are at the end of the barrel!).

```
local oYx, oYy, oYz = object.getYAxis ( o, object.kGlobalSpace )

oYx, oYy, oYz = math.vectorScale ( oYx, oYy, oYz, 0.1 )
srcX, srcY, srcZ = math.vectorAdd ( srcX, srcY, srcZ, oYx, oYy, oYz )
```

Now we get the current Y Axis of the Object, scale it by "0.1" and add it to the current Translation of the Object (again this is done to make the bullet appear roughly at the end of the gun). All of this maths is necessary to ensure that the bullet starts in the right place (at the end of the barrel of the gun), and heads towards the right place (where the crosshairs are currently pointing).

```
object.sendEvent ( bullet, "Weapon0Bullet", "onSetup", srcX, srcY, srcZ, dstX, dstY,
dstZ )
object.sendEvent ( w, "Weapon0", "onRotateHelix" )
```

These two lines fire the bullet (by sending the "onSetup" Event to the bullet), and make the gun move in a realistic way (by sending the "onRotateHelix" Event to the weapon).

```
        sound.stop ( o, 0 )
        sound.play ( o, 0, 1, false, 1 )
    end
end
```

Finally, we stop any Sound that is currently playing, and play the "gunshot" Sound attached to the Object, with a volume of "1" and no repeat.

```
else
    this.gameOver ( "ammo" )
end
```

The last few lines are called if the value of "ammo" is "0". They just make a call to the "gameOver" Function, with a Parameter of "ammo", to say that the Game is over because the Player has run out of ammo.

addDino (n)

This Function is called each time we wish to add a new dino to the Scene.

```
local s = application.getCurrentUserScene ( )

if ( s ~= nil )
then
   local dinoType = math.roundToNearestInteger ( math.random ( 1, 3 ) )

   local dinoPosX, dinoPosY = 0, 0
   local dinoPosY = 0
```

The only thing I'll comment on for the above lines is that we create a random dino type (between 1 and 3) which equates to the three types of dino that I have proposed (Tyrannosaurus Rex, Ankylosaurus and Velociraptor).

```
   repeat
      dinoPosX = math.random ( -450, 450 )
      dinoPosZ = math.random ( -450, 450 )
      dinoPosY = scene.getTerrainHeight ( dinoPosX, dinoPosZ )
   until ( dinoPosY > 0 and dinoPosY <= 25 )
```

This loop creates a random location for the dino, and repeats until the Terrain height is greater than "0", and less than, or equal to "25". We do this as we don't want our dinos to be stuck halfway up a mountain! By the way, the values of "-450" and "450" have been chosen based on the size of our Terrain.

```
   local tagName = "Dino"
   if ( dinoType == 1 )
   then
      this.hDino ( scene.createRuntimeObject ( s, "egg_beak" ) )
      tagName = tagName .. "T" .. n
   elseif ( dinoType == 2 )
   then
      this.hDino ( scene.createRuntimeObject ( s, "egg_beak" ) )
      tagName = tagName .. "A" .. n
   elseif ( dinoType == 3 )
   then
      this.hDino ( scene.createRuntimeObject ( s, "egg_beak" ) )
      tagName = tagName .. "V" .. n
   end
```

Now we create our dino according to the random value in "dinoType". We also Tag our dino using a Tag format of "dinoXN" (note the use of ".." to concatenate a String), where:

X = Type of Dino (T = T Rex, A = Ankylosaurus, V = Velociraptor)
N = Number of Dino (from the input Parameter)

```
    if ( this.hDino ( ) ~= nil )
    then
        object.translateTo ( this.hDino ( ), dinoPosX, dinoPosY, dinoPosZ, object.kGlobalSpace,
100 )
```

This little bit of code moves the Model to the calculated location.

```
        if ( scene.setRuntimeObjectTag ( s, this.hDino ( ), tagName ) )
        then
            object.addAIModel ( this.hDino ( ), "DinoAI" )
            object.postEvent ( this.hDino ( ), 0.1, "DinoAI", "onSetup", dinoType )
        else
            log.message ( "Runtime Object not created in HLDVisitCamera.addDino" )
    end
end
```

This last bit sets the Tag on the Model, adds the AIModel "DinoAI" to the Model and, finally, sends an Event to the "onSetup" Handler of the AIModel, giving a Parameter of "dinoType". Note that this event is "Posted", rather than "Sent". The difference between the "postEvent" and "sendEvent" Functions is that the "Post" Event can be delayed by specifying a value as the second Parameter ("0.1" above). This is useful to allow the Object receiving the Event time to process any other Events that it might have been sent.

atan2 (inZ, inX)

This Function is used to calculate the value of the Mathematical Function arctan2. Basically, this Function takes two input values ("Z" and "X"), and calculates the angle between them, putting the result into the correct quadrant. For more information on the atan2 function, I would suggest looking it up on Wikipedia.

```
local outVal = 0
```

First we create our LOCAL Variable that will be used to store the result of the Function.

```
if ( math.abs ( inX ) > math.abs ( inZ ) )
then
    outVal = math.atan ( inZ / inX )
else
```

Next, we check to see if the Absolute (i.e.: if the value is negative, multiply it by -1 to make it positive) value of the "inX" Parameter is greater than the Absolute value of the "inY" Parameter. If it is then we can simply set "outVal" to the arctangent of "inZ" divided by "inX".

```
    outVal = math.atan ( inX / inZ )

    if ( outVal < 0 )
    then
        outVal = -90 – outVal
    else
        outVal = 90 – outVal
    end
end
```

Otherwise, we set "outVal" to the arctangent of "inX" divided by "inZ". We then check if "outVal" is negative and, if it is, we subtract its value from "-90", else we subtract it from "90". This is done to place the result in the correct quadrant.

```
if ( inX < 0 )
then
    if ( inZ < 0 )
    then
        outVal = outVal – 180
    else
        outVal = outVal + 180
    end
end
```

Next we check to see if "inX" is negative and, if so, we check to see if "inZ" is also negative. If both are negative then we subtract "180" from "outVal". However, if only "inX" is negative then we add "180" to "outVal". Again, this is done to ensure the result is in the correct quadrant.

```
if ( inX == 0 and inZ == -1 )
then
    outVal = -90
end
```

Unfortunately, there is one set of input values that we need to catch ("inX" = 0 and "inZ" = -1) that produce an value of "-90".

```
return outVal
```

Finally, we return the value of "outVal".

centerMouse ( )

This is a very simple Function that sets the position of the Cursor to a point 50% across and 50% down the Screen, and is used to ensure that the crosshairs are in the centre of the Screen at all times.

```
hud.setCursorPosition ( application.getCurrentUser ( ), 50, 50 )
```

createDynObect ( )

This Function is called once, when the "onInit" Script is run. It sets up the Camera to use the inbuilt Physics system, as well as setting the default values as required.

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

if ( s ~= nil )
then
```

The first few lines are pretty easy, in that they create a LOCAL Variable that contains the current Object of the current Scene, and then check that it has been created successfully.

```
    local dynObject = scene.createRuntimeObject ( s, "HLDDummy" )

    if ( dynObject ~= nil )
    then
```

These lines basically do the same thing as above, but they create a new Object called "HLDDummy" which will be used as the Dynamics Object.

```
        this.hDynObject ( dynObject )

        object.matchTranslation ( dynObject, this.getObject ( ), object.kGlobalSpace )
        object.translate ( dynObject, 0, this.nDstHeight ( ), 0, object.kGlobalSpace )
```

This next bit sets the variable "hDynObject" that is part of "HLDVisitCamera" to be equal to the Dynamic Object created above, and then matches its Translation to the Camera.

Now we move into setting up the actual Physics properties of the Dynamic Object:

```
        if ( dynamics.createSphereBody ( dynObject, 0.5 ) )
        then
```

Firstly, we check that we can create a Sphere around our Dynamic Object with a radius of "0.5". If we can, then we do the following:

```
            dynamics.enableDynamics ( dynObject, true )
```

This enables the Dynamic Object to be accessible to the Physics system.

dynamics.enableCollisions (dynObject, true)

This enables the Dynamic Object to be able to participate in collisions with other Objects.

dynamics.enableGravity (dynObject, true)

This enables the Dynamic Object to be affected by Gravity.

dynamics.setLinearDampingEx (dynObject, 5.00, 0, 5.00)
dynamics.setAngularDamping (dynObject, 5.00)

This sets the amount of damping for both linear and angular movements.

dynamics.setLinearSpeedLimit (dynObject, 5)

This sets the maximum linear speed allowable.

dynamics.setMass (dynObject, 80)

This sets the mass of the Dynamic Object.

We now need to add a couple of lines:

dynamics.setGuardBox (dynObject, -500, -5, -500, 500, 100, 500)
dynamics.enableGuardBox (dynObject, true)

These lines set up a box, out of which the Dynamic Object cannot pass. The numbers in "setGuardBox" are the minimum and maximum X, Y & Z values for the GuardBox, and the second line then "enables" this GuardBox. This is used to prevent the Dynamic Object from going beyond the boundaries of our Terrain.

```
        end
    end
end
```

The last few lines just close the relevant "if" statements.

createHUD ( )

This Function creates the crosshairs HUD of the Game

local hCenterMouseAction = hud.newAction (application.getCurrentUser ( ), "HUDCenterMouse")

This first line creates a LOCAL Variable which is linked to a new HUD Action called "HUDCenterMouse". You can see that this Action is also linked only to the current User via "application.getCurrentUser ( )", which will be used frequently throughout our Scripts.

```
if ( hCenterMouseAction ~= nil )
then
```

These two you should be able to work out by now!

```
    hud.beginActionCommand ( hCenterMouseAction, hud.kCommandTypeSetCursorPosition
)
    hud.pushActionCommandArgument ( hCenterMouseAction, 0 )
    hud.pushActionCommandArgument ( hCenterMouseAction, 0 )
    hud.endActionCommand ( hCenterMouseAction )

end
```

This section is pretty similar to what we've seen before, and all it does is set the cursor position to the centre of the HUD.

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

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

local hHideMouseAction = hud.newAction ( application.getCurrentUser ( ),
"HUDHideMouse" )

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

This section is almost identical to the first section, except that it is used for showing and hiding the Mouse Cursor (the crosshairs).

```
local hTarget = hud.newComponent ( application.getCurrentUser ( ),
hud.kComponentTypeLabel )


if ( hTarget ~= nil )
then
    hud.setComponentPosition ( hTarget, 50, 50 )
    hud.setComponentSize ( hTarget, 6, 8 )
    hud.setComponentBackgroundColor ( hTarget, 127, 127, 127, 96 )
    hud.setComponentBorderColor ( hTarget, 127, 127, 127, 0 )
    hud.setComponentBackgroundImage ( hTarget, "HUDTarget" )
```

OK, this is a little bit different, in that it creates a new Label on the HUD. This Label is positioned, sized and coloured in the first few lines, and then has an Image ("HUDTarget") added to it. This basically generates the crosshairs for your weapon.

```
   local hShowTargetAction = hud.newAction ( application.getCurrentUser ( ),
"HUDShowTarget )

   if ( hShowTargetAction ~= nil )
   then
      hud.beginActionCommand ( hShowTargetAction, hud.kCommandTypeSetVisible )
      hud.pushActionCommandArgument ( hShowTargetAction, hTarget )
      hud.pushActionCommandArgument ( hShowTargetAction, true )
      hud.endActionCommand ( hShowTargetAction )
   end

   local hHideTargetAction = hud.newAction ( application.getCurrentUser ( ),
“HUDHideTarget )

   if ( hHideTargetAction ~= nil )
   then

      hud.beginActionCommand ( hHideTargetAction, hud.kCommandTypeSetVisible )
      hud.pushActionCommandArgument ( hHideTargetAction, hTarget )
      hud.pushActionCommandArgument ( hHideTargetAction, false )
      hud.endActionCommand ( hHideTargetAction )
   end
end
```

The above code is identical to the cursor code above, but shows, or hides, the crosshairs instead.

```
local o = this.getObject ( )
object.translateTo ( o, 0, 12, 0, object.kGlobalSpace, 100 )
```

Next, we move the Object up a little.

```
this.createDynObject ( )
```

And then we make it a Dynamic Object.

```
local dinos = application.getCurrentUserEnvironmentVariable ( nDinosaurs )

for myLoop = 1, dinos
do
   this.addDino ( myLoop )
end
```

The above lines add our dinos to the Scene. The number of dinos is determined by the value of the ENVIRONMENT Variable "nDinosaurs".

```
camera.setFieldOfView ( o, 30 )
```

We now set the "Field of View" of the Camera to be "30" degrees.

```
this.sendEvent ( "onWeaponChange", "Weapon0B" )

this.nTime ( 0 )
this.nScore ( 0 )
```

Finally, we send the "onWeaponChange" Event with a Parameter of "Weapon0B" (this is to initialise our weapon before the Game starts), and set the "time" and "score" counters to "0".

createIntroHUD ( )

This Function creates the HUD that will display the "Start" and "Quit" Buttons on the Screen at the beginning, or end, of a Game.

```
local hUser = application.getCurrentUser ( )

local hStartGame = hud.newComponent ( hUser, hud.kComponentTypeButton, "sStart" )

if ( hStartGame ~= nil )
then
   hud.setComponentPosition ( hStartGame, 50, 60 )
   hud.setComponentSize ( hStartGame, 20, 10 )
   hud.setComponentBackgroundColor ( hStartGame, 127, 127, 127, 96 )
   hud.setComponentBorderColor ( hStartGame, 127, 127, 127, 0 )
   hud.setButtonText ( hStartGame, "START GAME" )
end
```

Firstly, we get a Handle to the current User. Then, we create a new HUD Button Component, labelled "sStart". Finally, we check that the Component was created, and then set the relevant values (Screen position, size, background colour, border colour and the Text that will display on the Button).

```
local hQuitGame = hud.newComponent ( hUser, hud.kComponentTypeButton, "sQuit" )

if ( hQuitGame ~= nil )
then
   hud.setComponentPosition ( hQuitGame, 50, 40 )
   hud.setComponentSize ( hQuitGame, 20, 10 )
   hud.setComponentBackgroundColor ( hQuitGame, 127, 127, 127, 96 )
   hud.setComponentBorderColor ( hQuitGame, 127, 127, 127, 0 )
   hud.setButtonText ( hQuitGame, "QUIT GAME" )
end
```

Now, we just do it all over again, to create the "Quit" Button.

```
local hStartButtonAction = hud.newAction ( hUser, "HUDStartGame" )

if ( hStartButtonAction ~= nil )
then
   hud.beginActionCommand ( hStartButtonAction, hud.kCommandTypeSendEventToUser )
   hud.pushActionCommandArgument ( hStartButtonAction, application.getCurrentUser ( ) )
   hud.pushActionCommandArgument ( hStartButtonAction, "HLDMainHUD" )
   hud.pushActionCommandArgument ( hStartButtonAction, "onRun" )
   hud.endActionCommand ( hStartButtonAction )
   hud.setButtonOnClickedAction ( hStartGame, hStartButtonAction )
end
```

OK, all we are doing here is building a line of code that will be executed when the "Start" Button is clicked. This line of code is:

sendEventToUser (application.getCurrentUser ( ), "HLDMainHUD", "onRun")

As you can see, all you need to do to create HUD Actions is to create the line of code one step at a time. Also notice how you set the code to be executed by using the "setButtonOnClickedAction" command.

```
local hQuitButtonAction = hud.newAction ( hUser, "HUDQuitGame" )

if ( hQuitButtonAction ~= nil )
then
   hud.beginActionCommand ( hQuitButtonAction, hud.kCommandTypeSendEventToUser )
   hud.pushActionCommandArgument ( hQuitButtonAction, application.getCurrentUser ( ) )
   hud.pushActionCommandArgument ( hQuitButtonAction, "HLDMainHUD" )
   hud.pushActionCommandArgument ( hQuitButtonAction, "onQuit" )
   hud.endActionCommand ( hQuitButtonAction )
   hud.setButtonOnClickedAction ( hQuitGame, hQuitButtonAction )
end
```

Now we repeat the above code for the "Quit" Button.

Finally, we create a messaging area in the HUD, that will display any Messages that we may need (such as "Well Done – You Won!", or "You Ran Out of Time – Bad Luck!")

```
local hMessage = hud.newComponent ( hUser, hud.kComponentTypeLabel, "sMess" )

if ( hMessage ~= nil )
then
   hud.setComponentPosition ( hMessage, 50, 80 )
   hud.setComponentSize ( hMessage, 100, 20 )
   hud.setComponentBackgroundColor ( hMessage, 127, 127, 127, 96 )
   hud.setComponentBorderColor ( hMessage, 127, 127, 127, 0 )
   hud.setLabelTextHeight ( hMessage, 30 )
   hud.setLabelText ( hMessage, "Welcome to DINO HUNTER!")
end
```

The above code is almost identical to the code used for both of the Buttons, except for the lines that are specifically used to set the Label Text height and value.

gameOver (sType)

```
local hUser = application.getCurrentUser ( )
local nFinalTime = math.roundToNearestInteger ( this.nTime ( ) )
local hMessage = hud.getComponent ( hUser, "sMess" )
local hScene = application.getCurrentUserScene ( )
```

The above lines are fairly simple, but you can see in the third line how we can access our HUD Components that we created previously.

```
music.stop ( hScene, 1 )
music.play ( hScene, 3, 1 )
music.setVolume ( hScene, 0.5, 1 )
```

OK, here we have some more simple code that just stops the current Music from playing, and starts up a different Music loop.

Next we move in to the body of the "gameOver" code:

```
if ( sType == "score" )
then
   hud.setLabelText ( hMessage, "Well done you completed the task in " .. nFinalTime .. "
seconds!" )
else
   if ( sType == "time" )
   then
      hud.setLabelText ( hMessage, "Sorry, you ran out of time! Try again" )
   elseif ( sType == "ammo" )
   then
      hud.setLabelText ( hMessage, "Sorry, you ran out of ammo! Try again" )
   elseif ( sType == "collision" )
   then
      hud.setLabelText ( hMessage, "Sorry, you were attacked by a dino! Try again" )
   end
end
```

All we do in this block of code is to check the value of the "sType" Parameter, and set the Text of the "sMess" HUD Component to the correct value.

```
local hStart = hud.getComponent ( hUser, "sStart" )
local hQuit = hud.getComponent ( hUser, "sQuit" )

hud.setComponentActive ( hStart, true )
hud.setComponentActive ( hQuit, true )
hud.setComponentVisible ( hStart, true )
hud.setComponentVisible ( hQuit, true )
hud.setComponentVisible ( hMessage, true )
```

Right, all we are doing here is activating the two HUD Buttons, and making them visible, along with our HUD's Label.

```
user.sendEvent ( hUser, "HLDMain", "onSetNextScene", "" )
user.sendEvent ( hUser, "HLDMain", "onSwitchScene" )
```

Finally, all we do is send a couple of Events to "HLDMain". The first Event sets the value for the next Scene to be "" (i.e.: Nothing), and the second Event calls the "onSwitchScene" Handler.

registerTriggers ( )

```
local hUser = application.getCurrentUser ( )
local hObj = this.getObject ( )

user.sendEvent ( hUser, "HLDMain", "onRegisterObjectEvent", hObj, "HLDVisitCamera",
"KeyDown" )
user.sendEvent ( hUser, "HLDMain", "onRegisterObjectEvent", hObj, "HLDVisitCamera",
"KeyUp" )
```

This is a very simple Script in that it just sends two Events (KeyDown and KeyUp) to the "onRegisterObjectEvent" Handler in "HLDMain"

updateRadar (nDist, nDiff)

```
local hUser = application.getCurrentUser ( )

if ( nDist < 100 )
then
	if ( nDiff >= 0 and nDiff < 90 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar22" )
	elseif ( nDiff >= 90 and nDiff < 180 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar21" )
	elseif ( nDiff >= 180 and nDiff < 270 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar11" )
	else
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar12" )
	end
```

```
elseif ( nDist < 200 )
then
	if ( nDiff >= 0 and nDiff < 30 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar23" )
	elseif ( nDiff >= 30 and nDiff < 60 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar33" )
	elseif ( nDiff >= 60 and nDiff < 90 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar32" )
	elseif ( nDiff >= 90 and nDiff < 120 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar31" )
	elseif ( nDiff >= 120 and nDiff < 150 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar30" )
	elseif ( nDiff >= 150 and nDiff < 180 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar20" )
	elseif ( nDiff >= 180 and nDiff < 210 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar10" )
	elseif ( nDiff >= 210 and nDiff < 240 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar00" )
	elseif ( nDiff >= 240 and nDiff < 270 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar01" )
	elseif ( nDiff >= 270 and nDiff < 300 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar02" )
	elseif ( nDiff >= 300 and nDiff < 330 )
	then
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar03" )
	else
		user.postEvent ( hUser, 0.5, "HLDMainHUD", "onRadarUpdate", "Radar13" )
	end
end
```

Since this script repeats the same lines multiple times, I'll just explain the first few. All that this does is check the value of "nDist" (the distance between the Player and the dino), and if it is within range of the radar (200 is the outer squares, and 100 is the inner squares) then we check the value of "nDiff" to see what the angle between the two is. Depending on the angle we post an Event to the "onUpdateRadar" Handler of "HLDMainHUD" so that the radar gets updated correctly.

updateWeapon ( )

```
local w = this.hWeapon ( )

if ( w ~= nil )
then
```

These lines create a LOCAL Variable from the current "hWeapon" Variable, and check that it has been created correctly.

```
    local t = application.getTotalFrameTime ( )
    local dt = application.getLastFrameTime ( )
    local o = this.getObject ( )
    local curWeaponBackFactor = this.nCurWeaponBackFactor ( )
    local dstWeaponBackFactor = this.nDstWeaponBackFactor ( )
```

Again, we are creating some LOCAL Variables, this time for:

- t — the TOTAL frame time
- dt — the LAST frame time
- o — the CURRENT Object
- curWeaponBackFactor — the "Back Factor" of the current position of the weapon
- dstWeaponBackFactor — the "Back Factor" of the destination of the weapon

```
    if ( dstWeaponBackFactor > 0 )
    then
        this.nDstWeaponBackFactor ( dstWeaponBackFactor - dt )
    end

    if ( curWeaponBackFactor ~= dstWeaponBackFactor )
        then
        this.nCurWeaponBackFactor ( math.interpolate ( curWeaponBackFactor,
dstWeaponBackFactor, 10 * dt ) )
    end
```

OK, as promised, here is more detail on the "Back Factor". These lines of the Script firstly check to see if the "Back Factor" of the destination of the weapon is greater than "0". If it is then the value of the "nDstWeaponBackFactor" Variable is reduced by the LAST frame time (this is done to ensure that the movement of the weapon is proportional to the frame rate.). Next, a check is made that the position of the weapon is not the same as the destination of the weapon (i.e.: it has NOT finished its movement). If the result is that they are still different, then the value of the "nCurWeaponBackFactor" Variable is changed using an "Interpolate" maths function using the position and destination values, along with a value of "10 * dt" to ensure smooth movement.

```
    local oTx, oTy, oTz = object.getTranslation ( o, object.kGlobalSpace )
    local oRx, oRy, oRz = object.getRotation ( o, object.kGlobalSpace )
    local oXx, oXy, oXz = object.getXAxis ( o, object.kGlobalSpace )
    local oYx, oYy, oYz = object.getYAxis ( o, object.kGlobalSpace )
    local oZx, oZy, oZz = object.getZAxis ( o, object.kGlobalSpace )
```

Now we come to some more Maths....

The above lines are pretty straightforward, in that they create LOCAL Variables based on the Translation, Rotation, and Axes of the current Object.

```
    oYx, oYy, oYz = math.vectorScale ( oYx, oYy, oYz, 0.30 + 0.02 * math.sin ( t * 45 ) )
    oTx, oTy, oTz = math.vectorSubtract ( oTx, oTy, oTz, oYx, oYy, oYz )
    oXx, oXy, oXz = math.vectorScale ( oXx, oXy, oXz, 0.25 + 0.02 * math.sin ( t * 30 ) )
    oTx, oTy, oTz = math.vectorAdd ( oTx, oTy, oTz, oXx, oXy, oXz )
    oZx, oZy, oZz = math.vectorScale ( oZx, oZy, oZz, 0.40 + 0.02 * math.sin ( t * 55 ) - 0.15 *
curWeaponbackFactor )
    oTx, oTy, oTz = math.vectorSubtract ( oTx, oTy, oTz, oZx, oZy, oZz )
```

Now, we actually move the weapon itself. I'll just explain what the "math" functions do, and you should be able to work out the rest:

math.vectorScale	This Function "scales" the Vector (i.e.: it increases or decreases its size based on the final input value).
math.vectorSubtract	As you may have guessed, this Function "subtracts" the second set of input Vector values from the first.
math.vectorAdd	This Function simply "adds" two Vectors together.

Note that we are using the "math.sin" Function. This is used to create the actual circular movement of the weapon.

```
    object.setTranslation ( w, oTx, oTy, oTz, object.kGlobalSpace )
    object.setRotation ( w, oRx, oRy, oRz, object.kGlobalSpace )
    object.rotate ( w, 3 * curWeaponBackFactor, 0, 0, object.kLocalSpace )
end
```

The last few lines carry out the actual movement by setting the Translation of the weapon, to the calculated Translation values, in GLOBAL space, setting the Rotation of the weapon, to the calculated Rotation values, in GLOBAL space, and finally rotating the weapon in LOCAL space.

Why are the first two in GLOBAL space, and the last one in LOCAL space? That's relatively easy to answer. The setting of the Translation and Rotation need to be applied in GLOBAL space to ensure that the weapon stays in the right place relative to the Player. The "rotate" Function is carried out in LOCAL space, so that the weapon can be moved about a bit without moving away from where it is supposed to be (giving it a sort of "swinging" motion)!

vector2Rotation (V2R_OriginObject, V2R_TargetObject)

As I have previously said, this code was posted to the WIKI by FoolishFrost, to whom I am eternally grateful! I've also left in his comments, so that you can see what's going on.

```
-- Get Scene
local hScene = application.getCurrentUserScene ( )

-- Get V2R_OriginObject position and rotation.
local x, y, z = object.getTranslation ( V2R_OriginObject, object.kGlobalSpace )
local vx, vy, vz = object.getDirection ( V2R_OriginObject, object.kGlobalSpace )

-- Get V2R_TargetObject position.
local seex, seey, seez = object.getTranslation ( V2R_TargetObject, object.kGlobalSpace )

-- Calculate normalized (reduced to '1') relative vector
local nx, ny, nz = math.vectorNormalize ( seex - x, seey - y, seez - z )

-- Calculate nDot
local nDot = math.vectorDotProduct ( vx, vy, vz, nx, ny, nz )

-- All this returns the global rotation to target from V2R_OriginObject postion, and the nDot
for Frustum...
return nx, ny, nz, nDot
```

That's it for "HLDVisitCamera", next we move on to "Weapon0".

Weapon0

Next, we're going to create another AIModel for the weapon itself. This AIModel has one Variable, and two Handlers:

VARIABLES:

nHelixRotation	Type – Number Initial Value – 0 This is used for rotating the weapon.

HANDLERS:

onEnterFrame	This Handler is called once every Frame.
onRotateHelix	This Handler rotates the weapon using a "helical"-type rotation.

onEnterFrame ( )

```
local r = this.nHelixRotation ( )
local o = this.getObject ( )

if ( r > 0 )
then
```

Firstly, we create our LOCAL Variable with the value of "nHelixRotation", and then check if it is greater than 0.

```
    object.setRotation ( group.getSubObjectAt ( o, 0 ), 0, -90, 0, object.kParentSpace )
    object.rotate ( group.getSubObjectAt ( o, 0 ), 0, 0, r, object.kParentSpace )
```

If it is, then we set the Rotation of the Object, and "rotate" it accordingly. The major difference here is that we are talking about Groups and SubObjects! Basically, the weapon Object is composed of several SubObjects that have been Grouped together. Being Grouped means that the SubObjects can be changed/moved together, as if they were one Object, or each constituent part can be changed/moved individually if necessary.

The first line above is getting the SubObject at index position "0", and setting its Rotation to "-90" degrees in the Y Axis. This happens in the PARENT space (i.e.: the space of the Group).

The second line "rotates" the same SubObject by "r" in the Z Axis, again in the PARENT space.

```
    this.nHelixRotation ( math.max ( 0, r - 360 * application.getLastFrameTime ( ) ) )
end
```

Finally, "nHelixRotation" is set to a value based on the maximum of "0" and the calculation:

r – 360 * application.getLastFrameTime ( )

with "360" referring to a full circle of course!

onRotateHelix ( )

```
local r = this.nHelixRotation ( ) + 120

if ( r < 240 )
then
    this.nHelixRotation ( r )
end
```

This is a simple Handler that creates a LOCAL Variable by adding “120” to “nHelixRotation”, then checks to see if this value is less than “240”, and if it is, it sets “nHelixRotation” to the new value.

All that this means is that the weapon rotates correctly once fired.

Weapon0Bullet

For the bullet, we need a new AIModel called "Weapon0Bullet". This consists of 1 Function, 2 States and 2 Handlers:

FUNCTIONS:

explosionTest	This Function checks to see if the bullet has collided with anything, and if it has, it creates the "explosion".

STATES:

Explode	This State is used to handle the "explosion".
Fly (DEFAULT STATE)	This State is used to handle the flight of the bullet.

HANDLERS:

onDestroy	This Handler is used to destroy the bullet.
onSetup	This Handler is used to set up the bullet.

explosionTest (o)

This is a Function that takes an input Parameter ("o"), which refers to an Object.

```
local s = object.getScene ( o )
local tx, ty, tz = object.getTranslation ( o, object.kGlobalSpace )
local vx, vy, vz = object.getDirection ( o, object.kGlobalSpace )
```

Firstly, create a few LOCAL Variables for the Scene, and the Translation and Direction of the Object that has been passed as the input Parameter.

```
local hSens, sensDist, sensSID = scene.getFirstHitSensor ( s, tx, ty, tz, vx, vy, vz, 3 )
local terrChunk, terrDist, terrSID = scene.getFirstHitTerrainChunk ( s, tx, ty, tz, vx, vy, vz, 2 )
```

Next, we create some LOCAL Variables that refer to a HitSensor, and a HitTerrainChunk.

The first of these launches a ray into the Scene, and returns the Handle, Distance, and ID of the first "Sensor" Object that it finds. This ray is calculated using:

s	the Scene
tx, ty, tz	the start co-ordinates of the Ray
vx, vy, vz	the direction Vector of the Ray
3	the length of the Ray

The second line is almost identical, except that it checks for the first "Terrain Chunk" that it finds, returning the Handle, the Distance, and the ID of the Object, and the third line returns the same for any Terrain Chunk that it finds. You'll probably have noticed that my ray for the Terrain is less than the Sensor. This is just in case the Terrain is less than 1m thick and there is a Sensor Object just the other side.

```
if ( terrChunk ~= nil )
then
    this.Explode ( )
```

Firstly, we check to see if the bullet has hit the Terrain, and if it has we call the "Explode" State.

```
elseif ( hSens ~= nil )
then
```

Otherwise we check to see if there has been a "collision" with a "Sensor" Object.

```
    local sensX, sensY, sensZ = object.getTranslation ( hSens, object.kGlobalSpace )
    local nCount = scene.getTaggedObjectCount ( s )
    local sTag = ""
    local hTemp
    local hDino
```

Now we create some more LOCAL Variables getting the Translation of the hit "Sensor" and the number of "Tagged" Objects in the Scene.

```
    for nIndex = 0, nCount – 1
    do
```

After that we start our loop through all "Tagged" Objects in the Scene.

```
    hTemp = scene.getTaggedObject ( s, nIndex )

    if ( hTemp ~= nil )
    then
        local dinoX, dinoY, dinoZ = object.getTranslation ( hTemp, object.kGlobalSpace )
```

Then we get the "Tagged" Object according to the current value of nIndex, check it exists and, if so, obtain its current Rotation.

```
        if ( dinoX == sensX and dinoY == sensY and dinoZ == sensZ )
        then
```

Now we check to see if the Translation of the "Tagged" Object matches the Translation of the "Sensor".

```
            sTag = scene.getTaggedObjectAt ( s, nIndex )
```

If it does, then we get its "Tag".

```
            if ( string.getSubString ( sTag, string.getLength ( sTag ) – 1, 1 ) ~= "D" )
            then
                hDino = hTemp
            end
        end
    end
```

Next we check that we have got the dino, not the Dynamic Object.

```
    object.postEvent ( hDino, 1, "DinoAI", "onSleep" )
    this.Explode ( )
```

Then we post an Event to the dino calling its "onSleep" Handler, and call the "Explode" State.

```
    if ( object.hasController ( hSens, object.kControllerTypeDynamics ) )
    then
        vx, vy, vz = math.vectorScale ( vx, vy, vz, 1000 )
        dynamics.addForce ( hSens, vx, vy, vz, object.kGlobalSpace )
    end
end
```

If the bullet has collided with a "Sensor" Object, and the Object has Dynamics abilities, then we add the relevant force to the Object (i.e.: it will move when hit).

onDestroy ( )

```
local o = this.getObject ( )
local s = object.getScene ( o )

if (s ~= nil )
then
    scene.destroyRuntimeObject ( s, 0 )
end
```

As this is only really a one-liner (apart from the required LOCAL Variables, and the usual check), I'll do it in one hit. Basically, all this does is obtain a Handle to the current Scene for the current Object, and then destroys the Object.

onSetup (srcX, srcY, srcZ, dstX, dstY, dstZ)

This Handler takes multiple input parameters:

srcX, srcY, srcZ	these are the X, Y & Z of the source Vector of the bullet
dstX, dstY, dstZ	these are the X, Y & Z of the destination Vector of the bullet

```
local o = this.getObject ( )

object.setTranslation ( o, srcX, srcY, srcZ, object.kGlobalSpace )
object.lookAt ( o, dstX, dstY, dstZ, object.kGlobalSpace, 1 )
```

All we are doing here is creating the LOCAL Object, setting its Translation, in GLOBAL space, based on the source Vector, and then pointing it in the direction of the destination Vector, again in GLOBAL space.

this.postEvent (10, "onDestroy")

This line sends the "onDestroy" Event with a 10 second delay, as we don't want the bullet to travel forever if it doesn't hit anything!

this.explosionTest (o)

Finally, we run the "explosionTest" Function, to see if there needs to be an explosion or not (i.e.: we've hit something!).

Explode

Each State has three possible Actions:

- onEnter
- onLoop
- onLeave

For the "Explode" State, all we need is the "onEnter" Action:

Explode_onEnter ( )

local o = this.getObject ( )

shape.setMeshOpacity (o, 0)
sfx.stopParticleEmitterAt (o, 0)
sfx.startParticleEmitterAt (o, 0)

this.postEvent (5, "onDestroy")

All fairly simple, create a LOCAL Variable for the current Object, set the Opacity of its Mesh to "0", stop any "Particle Emitter" that is currently active on Index "0" of the Object, and then restart the "Particle Emitter". Finally, call the "onDestroy" Handler of the Object with a "5" second delay.

Fly

For the "Fly" State, all we need is the "onLoop" Action:

Fly_onLoop ( )

```
local o = this.getObject ( )
local dt = application.getLastFrameTime ( )

object.translate ( o, 0, 0, -75 * dt, object.kLocalSpace )

this.explosionTest ( o )
```

Again, all fairly simple. We create a LOCAL Variable for the current Object, create another LOCAL Variable for the "Last Frame Time", "Translate" the Object in LOCAL space (which makes the bullet move), and finally, run the "explosionTest" Function.

HLDMainHUD

The "HLDMainHUD" AIModel consists of 6 Functions, and 8 Handlers, and is used to set up the main HUD of the Game:

FUNCTIONS:

createAmmo	This Function creates the ammo part of the HUD.
createOptionScreen	This Function creates the options screen part of the HUD (This is not used in our Game).
createRadar	This Function creates the Radar part of the HUD.
createScore	This Function creates the score part of the HUD.
createTime	This Function creates the time part of the HUD.
createTitle	This Function creates the title part of the HUD.

HANDLERS:

onAmmoSetValue	This Handler is called to set the ammo value.
onClearRadar	This Handler is called to clear the Radar.
onInit	This Handler is called once to initialize the HUD.
onQuit	This Handler is called when the HUD "Quit" button is clicked.
onRadarUpdate	This Handler is called to update the Radar.
onRun	This Handler is called when the HUD "Start" button is clicked.
onScoreSetValue	This Handler is called to set the score value.
onTimeSetValue	This Handler is called to set the time value.

createAmmo (hUser)

This Function takes one input Parameter "hUser", which is a Handle to the required User, and creates, and sets the value of, the "ammo" HUD Label.

```
local c = hud.newComponent ( hUser, hud.kComponentTypeLabel, "MainHUD.Ammo" )

if ( c ~= nil )
then
   hud.setComponentPosition ( c, 10, 10 )
   hud.setComponentSize (c, 40, 10 )
   hud.setComponentBackgroundColor ( c, 0, 0, 0, 0 )
   hud.setComponentBorderColor ( c, 0, 0, 0, 0 )
   hud.setLabelText ( c, "Ammo: 100" )
   hud.setLabelTextHeight ( c, 40 )
end
```

Since we've seen the majority of this before in the "onInit" Handler of "HLDMain", I won't explain this anymore than to say that all we are doing is creating a Label to show the "ammo" value in the User's HUD.

createRadar (hUser)

This Function takes one input Parameter "hUser", which is a Handle to the required User, and creates, and sets the value of, the "Radar" HUD Component.

```
local c = hud.newComponent ( hUser, hud.kComponentTypeContainer, "MainHUD.Radar )

if ( c ~= nil )
then
   hud.setComponentPosition ( c, 90, 80 )
   hud.setComponentSize (c, 15, 15 )
   hud.setComponentBackgroundColor ( c, 0, 0, 0, 0 )
   hud.setComponentBorderColor ( c, 0, 0, 0, 0 )

   local hHUD = c
```

Up to this point all pretty much as we've seen before.

```
   for counterA = 0, 3
   do
      for counterB = 0, 3
      do
```

Here we just set up a couple of loops. Notice that the second loop is inside the first, so we'll go through this next bit 16 times.

```
         c = hud.newComponent ( hUser, hud.kComponentTypeLabel, "Radar" .. counterA ..
counterB )
```

For each pass through the loops we create a Label Component with a "Tag" of "RadarAB", where the "A" and "B" represent the two counters. This will give us 16 Labels "Tagged" from "Radar00" to "Radar33"

```
            if ( c~= nil )
            then
                hud.setComponentContainer ( c, hHUD )
                hud.setComponentPosition ( c, counterA * 25 + 12.5, counterB * 25 + 12.5 )
```

This is a little bit different in that we are setting each Label inside a Container "hHUD", and setting their respective positions at 12.5 plus 25 times the counter value. This is done so that they form squares, as the Component is placed using its mid-point (hence the use of 12.5!).

```
                hud.setComponentSize ( c, 25, 25 )
                hud.setComponentBackgroundColor ( c, 0, 0, 0, 0 )
                hud.setComponentBorderColor ( c, 255, 255, 255, 255 )
            end
        end
    end
end
```

Since we've seen the majority of this last bit before, I won't make any further comment.

createOptionScreen ( )

This is an empty Function, which can be used in the future for creating an options screen for your HUD.

createScore (hUser)

This Function takes one input parameter "hUser", which is a Handle to the required User, and creates, and sets the value of, the "score" HUD Label.

```
local c = hud.newComponent ( hUser, hud.kComponentTypeLabel, "MainHUD.Score" )

if ( c ~= nil )
then
    hud.setComponentPosition ( c, 90, 95 )
    hud.setComponentSize (c, 40, 10 )
    hud.setComponentBackgroundColor ( c, 0, 0, 0, 0 )
    hud.setComponentBorderColor ( c, 0, 0, 0, 0 )
    hud.setLabelText ( c, "Score: 0" )
    hud.setLabelTextHeight ( c, 40 )
end
```

Since this is almost identical to "createAmmo", I'll say no more!

createTime (hUser)

This Function takes one input parameter "hUser", which is a Handle to the required User, and creates, and sets the value of, the "time" HUD Label.

```
local c = hud.newComponent ( hUser, hud.kComponentTypeLabel, "MainHUD.Time" )

if ( c ~= nil )
then
    hud.setComponentPosition ( c, 10, 95 )
    hud.setComponentSize (c, 40, 10 )
    hud.setComponentBackgroundColor ( c, 0, 0, 0, 0 )
    hud.setComponentBorderColor ( c, 0, 0, 0, 0 )
    hud.setLabelText ( c, "Time: 0" )
    hud.setLabelTextHeight ( c, 40 )
end
```

Again I won't comment!

createTitle (hUser)

This Function takes one input parameter "hUser", which is a Handle to the required User, and creates, and sets the value of, the "title" HUD Label.

```
local c = hud.newComponent ( hUser, hud.kComponentTypeLabel, "MainHUD.Title" )

if ( c ~= nil )
then
    hud.setComponentPosition ( c, 50, 95 )
    hud.setComponentSize (c, 40, 10 )
    hud.setComponentBackgroundColor ( c, 0, 0, 0, 0 )
    hud.setComponentBorderColor ( c, 0, 0, 0, 0 )
    hud.setLabelText ( c, "DINO HUNTER" )
    hud.setLabelTextHeight ( c, 40 )
end
```

That's the last of these almost identical Functions.

onAmmoSetValue (n)

This Handler has one input parameter, "n", which is the value of the ammo.

```
local hUser = this.getUser ( )
local hComp = hud.getComponent ( hUser, "MainHUD.Ammo" )

hud.setLabelText ( hComp, string.format ( "%#.3d", n ) )
```

This Handler is a straightforward one-liner, which sets the value of the Text in the Label to be the value of the passed in parameter, "n".

One thing that I will comment on is the "string.format" Function. This Function is obviously used to format the way a String is displayed in the HUD, but how does it do this?

Well, the ShiVa "string.format" Function, is pretty much the same as the standard C "printf" Function, which means there are lots of different options that can be used when formatting a String, for example:

Numerical:

"%d"	Signed Integer
"%u"	Unsigned Integer
"%f"	Floating Point
"%g"	Floating Point, but in either normal or exponential whichever fits better
"%e"	Exponent
"%o"	Octal
"%x"	Hex

Character:

"%s"	String
"%q"	Quoted String
"%c"	Character

Common Flags:

"#"	Specifies that the value is to be converted to an alternate form
"0"	Specifies that the value is padded with "0"s
Number	The required precision

onClearRadar ( )

```
local hUser = this.getUser ( )
local sTag = ""
local hComp

for countA = 0, 3
do
   for countA = 0, 3
   do
      sTag = "Radar" .. countA .. countB
      hComp = hud.getComponent ( hUser, sTag )
      hud.setComponentBackgroundColor ( hComp, 0, 0, 0, 0 )
   end
end
```

Fairly straightforward, I think. Firstly we set out LOCAL Variables then we run our two loops, getting the HUD Component with the relevant "Tag". Finally we set the background colour of the Component.

onInit ( )

```
local hUser = this.getUser ( )
hud.setDefaultFont ( hUser, "HLDArial" )

this.createAmmo ( hUser )
this.createTime ( hUser )
this.createScore ( hUser )
this.createRadar ( hUser )
this.createOptionScreen ( hUser )
this.createTitle ( hUser )
```

Fairly straightforward, I think. Firstly we set the default Font for the HUD "HLDArial", then we call the various "create" Functions to set up the HUD.

onQuit ( )

```
application.quit ( )
```

All that this Handler does is close down the Game.

onRadarUpdate (sTag)

```
local hUser = this.getUser ( )
local hComp = hud.getComponent ( hUser, sTag )

hud.setComponentBackgroundColor ( hComp, 255, 0, 0, 255 )
```

Pretty easy I think. All we've done is created some LOCAL Variables, got the relevant HUD Component based on the input Parameter ('sTag"), and set the background colour of the Component to red.

onRun ( )

```
if ( not application.getCurrentUserEnvironmentVariable ( "gameOn" ) )
then
    local hUser = application.getCurrentUser ( )
    local hStart = hud.getComponent ( hUser, "sStart" )
    local hQuit = hud.getComponent ( hUser, "sQuit )
    local hScene = application.getCurrentUserScene ( )

    application.setCurrentUserEnvironmentVariable ( "gameOn", true )
```

Pretty easy so far. All we've done is check the value of the Environment Variable "gameOn" and, if it equals "false", then we set up some LOCAL Variables.

```
    hud.setComponentActive ( hStart, false )
    hud.setComponentActive ( hQuit, false )
    hud.setComponentVisible ( hStart, false )
    hud.setComponentVisible ( hQuit, false )
    hud.setComponentVisible ( hud.getComponent ( hUser, "sMess" ), false )
```

Now all we do is to set the "start" and "quit" Buttons to be inactive and invisible. The last line sets the message Label of the HUD to be invisible.

```
    music.stop ( hScene, 5 )
    music.play ( hScene, 4, 5 )
    music.setVolume ( hScene, 0.33, 5 )
```

Finally, for this bit, we stop any Music that is currently playing, and start up a different tune.

```
else
    application.restart ( )
end
```

If "gameOn" is equal to "true", then all we do is restart the Game. You'll probably notice that I've taken the easy way out, and just restarted the whole Application!

onScoreSetValue (n)

This Handler has one input Parameter, "n", which is the value of the score.

```
local hUser = this.getUser ( )
local hComp = hud.getComponent ( hUser, "MainHUD.Score" )

hud.setLabelText ( hComp, string.format ( "%#.3d", n ) )
```

Since this Handler is almost identical to "onAmmoSetValue", I won't comment.

onTimeSetValue (n)

This Handler has one input Parameter, "n", which is the value of the time.

```
local hUser = this.getUser ( )
local hComp = hud.getComponent ( hUser, "MainHUD.Time" )

hud.setLabelText ( hComp, string.format ( "%#.4d", n ) )
```

Again, no comment!

Now it's time to introduce some Artificial Intelligence (AI), by building an AIModel specifically for our dinos!

DinoAI

The "DinoAI" AIModel consists of 11 Variables, 3 Handlers and 3 States, and is used to set up the "intelligence" of our dinos:

VARIABLES:

bHerbivore	Type – Boolean Initial Value – false This is used to denote either a "Herbivore", or a "Carnivore".
bReact	Type – Boolean Initial Value – false This is used to control the reaction of the dino.
bRotateRight	Type – Boolean Initial Value – false This is used to control the Rotation of the dino.
hDynObject	Type – Object Initial Value – 'nil' This is used for the current Dynamic Object.
nCurAngleH	Type – Number Initial Value – 0 This is used for the current horizontal angle of the dino.
nDist	Type – Number Initial Value – 0 This is used to denote the collision detection distance of the dino.
nDstAngleH	Type – Number Initial Value – 0 This is used for the destination horizontal angle of the dino.
nDstFactor	Type – Number Initial Value – 0 This is used for the destination animation factor of the dino.

nHunger

Type – Number
Initial Value – 0
This is used to denote how hungry the dino is.

nRunSpeed

Type – Number
Initial Value – 0
This is used to denote the running speed of the dino.

nTimer

Type – Number
Initial Value – 0
This is used to control the timing of the dino's actions.

nValue

Type – Number
Initial Value – 0
This is used to denote the value of the dino.

nWalkSpeed

Type – Number
Initial Value – 0
This is used to denote the standard walking speed of the dino.

sDynTag

Type – String
Initial Value – "Nothing"
This is used to denote the Tag of the Dynamic Object.

sTag

Type – String
Initial Value – ""
This is used to denote the Tag of the dino.

sType

Type – String
Initial Value – ""
This is used to denote the type of dino (i.e.: Ankylosaurus, T Rex, Velociraptor etc.)

FUNCTIONS:

atan2

This Function is used to replicate the mathematical Function atan2.

vector2Rotation

This Function calculates a Rotation based on two Vectors – Thanks to FoolishFrost for this one.

HANDLERS:

onInit	This Handler is the standard initialization Handler for the AIModel.
onSetup	This Handler sets up our dino.
onSleep	This Handler is called when a dino is "put to sleep".

STATES:

Eating	This State controls our dino's eating habits.
Walking (DEFAULT STATE)	This State controls our dino's movement.
Running	This State controls our dino's movement when close to the Player (Carnivores will "attack", and Herbivores will "run away").

atan2 (inZ, inX)

This Function is identical to the "atan2" Function in "HLDVisitCamera".

vector2Rotation (V2R_OriginObject, V2R_TargetObject)

This Function is identical to the "vector2Rotation" Function in "HLDVisitCamera". Thanks again to FoolishFrost!

onInit ( )

```
local myObj = this.getObject ( )
local s = object.getScene ( myObj )

this.sTag ( scene.getObjectTag ( s, myObj ) )
```

A fairly straightforward Script that just sets the value of the LOCAL Variable "sTag" to the value of the Tag of the current Object.

onSetup (dinoType)

This Handler is where the basics of our dino are setup.

```
local o = this.getObject ( )
local s = object.getScene ( o )
local dinoMass = 0  -- Weight in kg
local dinoBody = 0  -- Body Size for Bounding Sphere

if ( s ~= nil )
then
   local dynObject = scene.createRuntimeObject ( s, "HLDDummy" )
```

OK, this is a pretty easy start. All we are doing is creating some LOCAL Variables, checking the Scene exists, and then creating a Runtime Object based on "HLDDummy", which will be used as the Dynamics Object for our dino.

```
   if ( dynObject ~= nil )
   then
      local dTag = this.sTag ( ) .. "D"
      scene.setRuntimeObjectTag ( s, dynObject, dTag )
      this.sDynTag ( dTag )
      this.hDynObject ( dynObject )

      object.matchTranslation ( dynObject, o, object.kGlobalSpace )
      this.nHunger ( math.roundToNearestInteger ( math.random ( 0, 70 ) ) )
```

Next, we check that the Dynamic Object has been created, set the LOCAL Variable "hDynObject", set the "Tag" to the created Object, save the "Tag" in "sDynTag", match the Translation of the current Object to the Dynamic Object, and finally set "nHunger" to a random Integer between "0" and "70".

```
if ( dinoType == 1 )
then
    this.sType ( "T Rex" )
    dinoMass = 5000
    dinoBody = 1
    this.nWalkSpeed ( 1.5 ) -- Walking Speed
    this.bHerbivore ( false )   -- Herbivore?
    this.nValue ( 100 ) -- Points Value
    this.nDist ( 5 )-- Collision Detection Distance
    this.nRunSpeed ( 25 )   -- Running Speed Multiplier
```

Now we set up the relevant values for the dino, based on the "dinoType" Parameter.

```
elseif ( dinoType == 2 )
then
    this.sType ( "Ankylosaurus" )
    dinoMass = 10000
    dinoBody = 2
    this.nWalkSpeed ( 0.75 )
    this.bHerbivore ( true )
    this.nValue ( 50 )
    this.nDist ( 5 )
    this.nRunSpeed ( 40 )
elseif ( dinoType == 3 )
then
    this.sType ( "Velociraptor" )
    dinoMass = 80
    dinoBody = 0.25
    this.nWalkSpeed ( 2 )
    this.bHerbivore ( false )
    this.nValue ( 200 )
    this.nDist ( 5 )
    this.nRunSpeed ( 25 )
end
```

The above lines are used to set up the different types of dinos.

```
this.bReact ( true )

if ( dynamics.createSphereBody ( dynObject, dinoBody ) )
then
    dynamics.enableDynamics ( dynObject, true )
    dynamics.enableCollisions   ( dynObject, true )
    dynamics.enableGravity  ( dynObject, true )
    dynamics.setLinearDampingEx ( dynObject, 5.00, 0, 5.00 )
    dynamics.setAngularDamping  ( dynObject, 5.00 )
    dynamics.setLinearSpeedLimit( dynObject, this.nWalkSpeed ( ) )
```

```
            dynamics.setMass( dynObject, dinoMass )
            dynamics.setGuardBox( dynObject, -500, -10, -500, 500, 20, 500 )
            dynamics.enableGuardBox ( dynObject, true )
        end
    end
end
```

Finally, it's a simple matter of setting the "bReact" Variable to "true", and setting up the relevant Dynamics.

onSleep ( )

This Handler is called when a dino is "put to sleep" (i.e.: hit by a dart!).

```
local o = this.getObject ( )
local s = object.getScene ( o )
local hCam = application.getCurrentUserActiveCamera ( )
local d = this.hDynObject ( )
```

Firstly, we create our LOCAL Variables.

```
object.sendEvent ( hCam, "HLDVisitCamera", "onUpdateScore", this.nValue ( ) )
local dObj = scene.getTaggedObject ( s, this.dTag ( ) )
scene.destroyRuntimeObject ( s, dObj )
scene.destroyRuntimeObject ( s, o )
```

Next, we send an Event to "HLDVisitCamera" to tell it to update the score, based on the value of the dino. After that, we get a Handle to the Object with the "Tag" stored in "dTag", and finally remove both the "Tagged" Object, and the main Object from the Scene.

Eating

Eating_onEnter ( )

```
this.nTimer ( 0 )
dynamics.setIdle ( this.hDynObject ( ), true )
```

All we are doing here is resetting the value of the LOCAL timer Variable to "0", and disabling any Dynamics on the Dynamic Object saved in "hDynObject".

Eating_onLeave ( )

```
dynamics.setIdle ( this.hDynObject ( ), false )
```

All we are doing here is re-enabling the Dynamics on the Dynamic Object saved in "hDynObject".

Eating_onLoop ( )

This State is called once per Frame whilst the dino is eating.

```
local dt = application.getLastFrameTime ( )
local rand = 0
local o = this.getObject ( )
local hCam = application.getCurrentUserActiveCamera ( )
local d = this.hDynObject ( )
```

The first few lines should be what you've come to expect by now!

```
object.matchTranslation ( o, d, object.kGlobalSpace )

local dist = object.getDistanceToObject ( o, hCam )

if ( dist < this.nDist ( ) )
then
   object.postEvent ( hCam, 1, "HLDVisitCamera", "onDinoCollision", this.sTag ( ) )
```

These lines match the Translation of the Object to the Dynamic Object, and then check to see if we have a collision between the dino and the Camera and, if we do, then we post the "onDinoCollision" Event to "HLDVisitCamera" with the Tag of the dino.

```
else
   if ( this.nTimer ( ) > 15 )
   then
      rand = math.random ( 0, 1 )
      if ( rand > 0.333 and rand < 0.667 )
      then
         this.nHunger ( this.nHunger ( ) - math.roundToNearestInteger ( math.random ( 5, 10
) ) )
         if ( this.nHunger ( ) < 0 )
         then
            this.nHunger ( 0 )
         end
      end
      this.nTimer ( 0 )
   else
      this.nTimer ( this.nTimer ( ) + dt )
   end
```

If there is no collision, then we check that our Object exists. Next we check to see if it has been more than "15" seconds since we last reset the timer, and if "true" then we generate a random number between "0" and "1". We then check the value of the generated number and, if it's between "0.333" and "0.667", we reduce the value of the "nHunger" Variable by a random amount between "5" and "10". Notice that we have rounded this amount to the nearest Integer, so that "nHunger" will always contain an Integer value. After this, we check to see if the new value of "nHunger" is less than "0" and, if it is, we set it to "0" (i.e.: "nHunger" cannot be less than "0").

Finally, we set the value of "nTimer" to "0", unless our timer has not yet reached the "15" seconds threshold, in which case we increment our timer by "dt".

```
    if ( ( this.bHerbivore ( ) and dist < 75 ) or ( not this.bHerbivore ( ) and dist < 50 ) )(
    then
        if ( this.nHunger ( ) < 90 )
        then
            this.postStateChange ( 0.1, "Running" )
        end
    else
        rand = math.random ( 25, 50 )

        if ( this.nHunger ( ) < rand )
        then
            this.postStateChange ( "Walking" )
        end
    end
end
```

This last bit checks to see if we have a Herbivore and "dist" is less than "75", or we have a Carnivore (i.e.: NOT a Herbivore) and "dist" < "50". If either of these are "true" then we check the value of "nHunger". If this is less than "90" then we post a "State Change" request to set the dino to the "Running" State. However, if the distance check is "false" then we generate a random value between "25" and "50". We then check the value of "nHunger", and if this is less than the generated random value, we post a "StateChange" request to set the dino to the "Walking" State.

Running

All we need for the "Running" State is the "onEnter" and "onLoop" Actions.

Running_onEnter ( )

```
this.bReact ( true )
this.nTimer ( 0 )
```

All we do here is set the LOCAL Variable "bReact" to "true", and set the value of the LOCAL timer Variable to "0".

Running_onLoop ( )

```
local dt = application.getLastFrameTime ( )
local rand = 0
local o = this.getObject ( )
local d = this.hDynObject ( )
local hCam = this. getCurrentUserActiveCamera ( )
local dist = object.getDistanceToObject ( o, hCam )
```

Again, as always, we set up our LOCAL Variables.

```
if ( dist < this.nDist ( ) )
then
   object.postEvent ( hCam, 1, "HLDVisitCamera", "onDinoCollision", this.sTag ( ) )
```

Again, as in the "Eating" State, we check whether a collision has occurred.

```
else
   if ( ( this.bHerbivore ( ) and dist > 75 ) or ( not this.bHerbivore ( ) and dist > 50 ) )
   then
      this.postStateChange ( 0.1, "Walking" )
   else
```

Now, assuming our Dynamics Object exists, we check to see if the current dino is a Herbivore, or not. This check is made, so that we can have different distances for the "StateChange" request ("75" for Herbivores, and "50" for Carnivores).

```
      this.nHunger ( this.nHunger ( ) + dt / 2 )

      if ( this.nHunger ( ) >= 90 )
      then
         this.postStateChange ( 0.1, "Eating" )
      end
```

In these lines, we increase the value of "nHunger" by a small amount (dt / 2), then check to see if we need to change the State to "Eating". You should be able to work out how this is done.

```
      rand = math.random ( 0, 1 )

      if ( this.nHunger ( ) > 75 and rand > 0.333 )
      then
         this.postStateChange ( 0.1, "Eating" )
      else
```

If "nHunger" is less than "90", we generate a random value. We then check whether "nHunger" is greater than "75" and the random value is greater than "0.333". If both of these are "true", then we post a "StateChange" Event accordingly to make the dino begin "Eating".

```
         local fx, fy, fz, nDiff = 0, 0, 0, 0
         local camX, camY, camZ = object.getBoundingSphereCenter ( hCam )

         fx, fy, fz, nDiff = this.vector2Rotation ( o, hCam )
         nDiff = this.atan2 ( fz, fx )

         if ( nDiff < 0 )
         then
            fy = nDiff + 360
         end

         local objX, objY, objZ = object.getRotation ( o, object.kGlobalSpace )
```

Next, we set up some more LOCAL Variables that we will use for rotating the dino either towards, or away from, the Camera. This is done by obtaining the rotation towards the Camera (using "vector2Rotation", and "atan2"), and also the current Rotation of the Object.

```
if ( this.bHerbivore ( ) )
then
   local decisionMax = 0

   if ( this.bReact ( ) )
   then
      decisionMax = 2
   else
      decisionMax = 5
   end
```

Here, we are just setting up a reaction time for our dino (NOTE: this is for Herbivores ONLY) based on whether "bReact" is "true" or "false".

```
   if ( this.nTimer ( ) > decisionMax )
   then
      this.bReact ( false )
      rand = math.roundToNearestInteger ( math.random ( 130, 230 ) )

      fy = fy + rand

      if ( fy > 360 )
      then
         fy = fy – 360
      end

      this.nTimer ( 0 )
   end
end
```

If our timer is greater than the reaction time calculated above, then we set "bReact" to "false". We then generate a random number between "130" and "230" which will be used as the direction that our dino will head (being the angle away from the Camera). We then add this random number to "fy", and if the result is greater than "360" reduce it by "360", to ensure that we only rotate between "0" and "360" degrees.

```
this.nDstAngleH ( fy )

if ( fy > objY and fy < objY + 180 )
then
   this.bRotateRight ( false )
else
   this.bRotateRight ( true )
end
```

Next, we put the value of "fy" into the horizontal destination angle ("nDstAngleH"), and set "bRotateRight" depending on the difference between "fy" and "objY".

```
        local rh = math.interpolate ( this.nCurAngleH ( ), this.nDstAngleH ( ), 10 * dt )

        this.nCurAngleH ( rh )

        object.setRotation ( o, 0, rh, 0, object.kGlobalSpace )
        object.matchTranslation ( o, d, object.kGlobalSpace )

        fx, fy, fz = 0, 0, 0
        local oXx, oXy, oXz = object.getXAxis ( o, object.kGlobalSpace )
        local oYx, oYy, oYz = object.getYAxis ( o, object.kGlobalSpace )
        local oZx, oZy, oZz = object.getZAxis ( o, object.kGlobalSpace )

        if ( this.bRotateRight ( ) )
        then
            fx, fy, fz = math.vectorAdd ( fx, fy, fz, oXx, oXy, oXz )
        else
            fx, fy, fz = math.vectorSubtract ( fx, fy, fz, oXx, oXy, oXz )
        end

        fx, fy, fz = math.vectorNormalize ( fx, 0, fz )
        fx, fy, fz = math.vectorScale ( fx, 0, fz, this.nRunSpeed ( ) * dynamics.getMass ( d ) *
this.nWalkSpeed ( ) )

        dynamics.addForce ( d, fx, fy, fz, object.kGlobalSpace )
        dynamics.addForce ( d, 0, -5000, 0, object.kGlobalSpace )
    end

    this.nTimer ( this.nTimer ( ) + dt )
end
```

Finally, we move our dino using pretty much the same code that we use to move our Camera.

Walking

All we need for the "Walking" State is the "OnEnter" and "onLoop" Actions.

Walking_onEnter ( )

```
this.nTimer ( 0 )

local rand = math.random ( 0, 360 )

this.nCurAngleH ( rand )
this.nDstAngleH ( rand )

object.setRotation ( this.getObject ( ), 0, rand, 0, object.kGlobalSpace )
```

Another fairly simple Script. All we are doing is resetting the value of the LOCAL timer Variable to "0", selecting a random number between "0" and "360", and using this value as the direction in which our dino will start walking.

Walking_onLoop ( )

```
if ( this.sDynTag ( ) ~= “Nothing” )
then
```

Firstly, we check to see if our “Tags” have been set up yet. If they haven’t then this Action will simply return without doing anything. This has only been done in the “Walking” State, as this is the default State and will begin as soon as we set up our AIModel.

```
    local dt = application.getLastFrameTime ( )
    local rand = 0
    local o = this.getObject ( )
    local d = this.hDynObject ( )
    local hCam = application.getCurrentUserActiveCamera ( )
    local dist = object.getDistanceToObject ( o, hCam )
```

As usual, we set up our LOCAL Variables.

```
    if ( dist < this.nDist ( ) )
    then
        object.postEvent ( hCam, 1, "HLDVisitCamera", "onDinoCollision", this.sTag ( ) )
    else
```

Now we check for collisions.

```
            this.nHunger ( this.nHunger ( ) + dt / 4 )

            if ( this.nHunger ( ) >= 90 )
            then
                this.postStateChange ( 0.1, "Eating" )
            else
                if ( ( dist < 75 and this.bHerbivore ( ) ) or ( dist < 50 and not this.bHerbivore ( ) )
                then
                    this.postStateChange ( 0.1, “Running” )
                else
                    rand = math.random ( 0, 1 )

                    if ( rand > 0.333 or this.nHunger ( ) > 50 )
                    then
                        this.postStateChange ( 0.1, "Eating" )
                    else
```

We then check that our Dynamics Object exists, increase “nHunger” (by a smaller amount than in “Running_onLoop”), and do our calculations to see if the dino needs to stop moving and start eating, or start “Running”.

```
if ( this.nTimer ( ) > 20 )
then
      rand = math.random ( 0, 1 )

   if ( rand > 0.667 )
   then
      this.nDstAngleH ( this.nDstAngleH ( ) - 10 * dt )
      this.bRotateRight ( true )
   elseif ( rand < 0.333 )
   then
      this.nDstAngleH ( this.nDstAngleH ( ) + 10 * dt )
      this.bRotateRight ( false )
   end
   this.nTimer ( 0 )
end
```

The above code checks the timer and, if it is over "20" then we generate a random number between "0" and "1". If this value is over "0.667" then we set our dino to rotate right by a small amount, otherwise if it is less than "0.333" we set our dino to rotate left by a small amount. This is done to give a sort of wandering behaviour where the dino changes direction every so often. Finally, we reset the timer back to "0".

```
local x, y, z = object.getTranslation ( d, object.kGlobalSpace )

local s = application.getCurrentUserScene ( )
local nHeight = scene.getTerrainHeight ( s, x, z )

if ( nHeight > 25 )
then
   if ( this.bRotateRight ( ) )
   then
      this.nDstAngleH ( this.nDstAngleH ( ) + 45 * dt )
      this.bRotateRight ( false )
   else
      this.nDstAngleH ( this.nDstAngleH ( ) - 45 * dt )
      this.bRotateRight ( true )
   end
end
```

Next, we check to see if the current Terrain height is over "25" and, if so, we rotate our dino to try and move it to a lower Terrain height value. This is done to hopefully ensure that the dino doesn't go too high up the Terrain!

```
object.translateTo ( o, x, y, z, object.kGlobalSpace, 10 * dt )
local rh = math.interpolate ( this.nCurAngleH ( ), this.nDstAngleH ( ), 10 * dt
)

this.nCurAngleH ( rh )
object.setRotation ( o, 0, rh, 0, object.kGlobalSpace )
object.matchTranslation ( o, d, object.kGlobalSpace )
```

```
                local oXx, oXy, oXz = object.getXAxis ( o, object.kGlobalSpace )
                local oYx, oYy, oYz = object.getYAxis ( o, object.kGlobalSpace )
                local oZx, oZy, oZz = object.getZAxis ( o, object.kGlobalSpace )
                local fx, fy, fz = 0, 0, 0
                fx, fy, fz = math.vectorSubtract ( fx, fy, fz, oZx, oZy, oZz )

                if ( this.bRotateRight ( ) )
                then
                    fx, fy, fz = math.vectorAdd ( fx, fy, fz, oXx, oXy, oXz )
                else
                    fx, fy, fz = math.vectorSubtract ( fx, fy, fz, oXx, oXy, oXz )
                end

                fx, fy, fz = math.vectorNormalize ( fx, 0, fz )
                fx, fy, fz = math.vectorScale ( fx, 0, fz, 15 * dynamics.getMass ( d ) *
this.nWalkSpeed ( ) )

                dynamics.addForce ( d, fx, fy, fz, object.kGlobalSpace )
                dynamics.addForce ( d, 0, -5000, 0, object.kGlobalSpace )
            end
        end

    end
  end
 end
end
```

Finally, we calculate our movement Forces and apply them to our dino.

Well, we've finally come to the end of the Scripts! And, since our Game is now finished, you should be able to play it (in the Scene Viewer if you have the PLE version of ShiVa). Our counters and Radar should work properly, our dinos should appear to have some semblance of intelligence, our gun should fire, and our dinos should disappear from the Screen when hit.

As you can probably tell, I haven't used the full power of the HLD Library, and my code leaves a bit to be desired in the optimisation stakes! Hopefully by reading this book you have started to get the hang of being a Game Designer / Programmer.

Now, all that's left for me to do is to take a nice long break from my keyboard, and to wish you all the best with your future ShiVa apps.

Oh, and one last thing.....

All Scripts etc. used in this book were tested in v1.7 of ShiVa, so keep an eye on the StoneTrip website, and the Forums for details of new releases as they happen (which is pretty frequently!).

Finally, I would like to express my gratitude to all the guys (and girls?) at StoneTrip for the help they have given me in putting this book together. THANK YOU!