Chapter 13

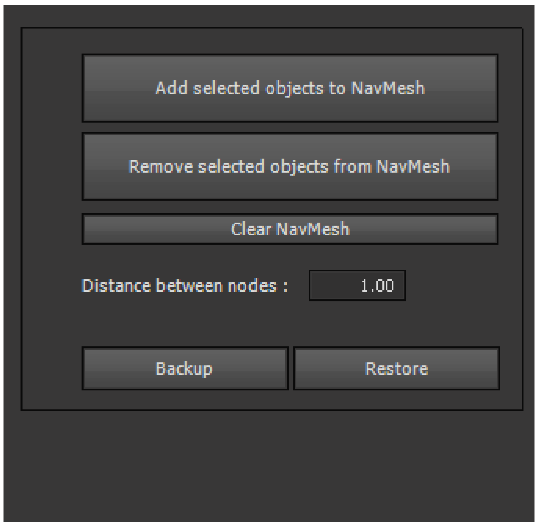
Getting from A to B

In this Chapter I'm going to be showing you how to use the NavMesh Editor to set up an Object that will move between pre-defined points in a grid, which is commonly known as "path-finding", and is used in many Games to control the actions on NPCs (Non-Player Characters). It is useful when there is a need to have NPCs patrolling a certain area, as the NavMesh can be set up to avoid obstacles, such as rocks, trees etc. so that the NPC doesn't bump into them, or go through them!

The NavMesh Editor makes it possible to create grids of navigation points (NavMeshes). Its role is to indicate which parts of a 3D scene are "navigable". The grid of navigation points is a regular grid in which each point can have up to 8 neighbours. This module allows the creation of a sequence of navigation points by adding points to, or removing points from, the initial empty grid.

Basically, the NavMesh Editor allows you to create & edit Mesh navigation points for automatic pathfinding.

NOTE: This Editor is **only** available when a Scene has been loaded into the Scene Viewer, and an Object within the Scene has been clicked on, since the NavMesh can only be added to Meshes.


Add selected objects to NavMesh
Remove selected objects from NavMesh
Clear NavMesh
Backup
Restore

Distance between nodes : 1.00

Add selected objects to NavMesh: This option allows the addition of NavMesh points onto the surface of the Object(s) that have been selected in the Scene Viewer.

Remove selected objects from NavMesh: This option allows the removal of NavMesh points from the surface of the Object(s) that have been selected in the Scene Viewer.
NOTE: the NavMesh points will be removed instantly, and no warning message will be displayed.

Clear NavMesh: This option removes all points from the NavMesh.
NOTE: the NavMesh points will be removed instantly, and no warning message will be displayed.

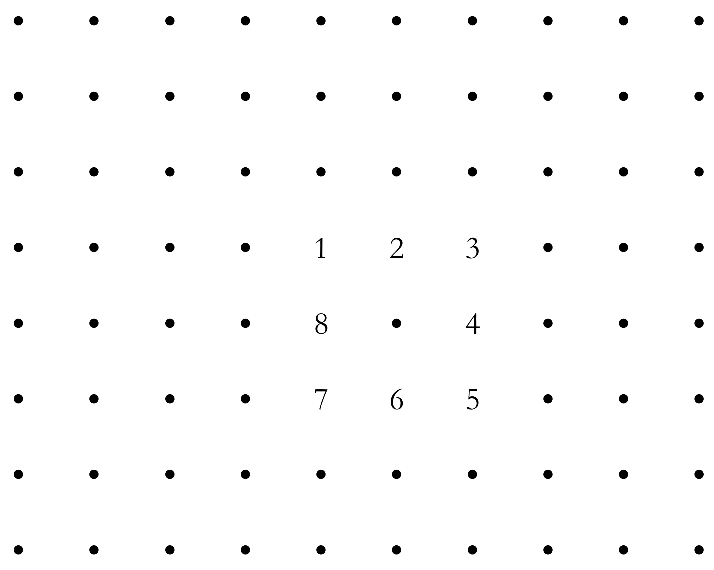
Distance between nodes: This option allows the setting of the distance between the points of the NavMesh grid.

TIP: Changing the "Distance between nodes" will reinitialise the NavMesh. So this value must be selected carefully before starting to work with it and placing points. If 1 ShiVa unit is the equivalent to 1 meter, which is a strongly recommended assumption, then a value between 0.5 and 1.5 should be a good value.

Backup: This option copies the current NavMesh to the Clipboard.

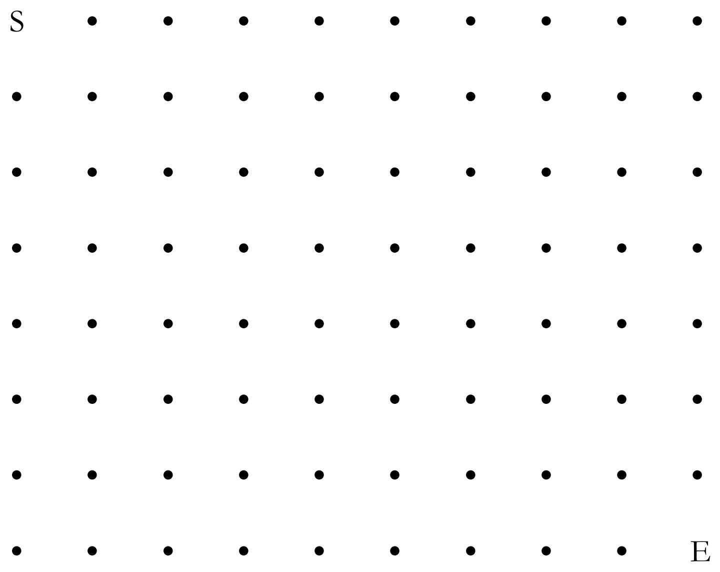
Restore: This option restores the "backed up" NavMesh from the Clipboard.

An empty NavMesh would look something like this:



Note in the above how all points (except those at the edges) have 8 neighbours. I have numbered the neighbours of one of the central points (coloured in Red) to illustrate this.

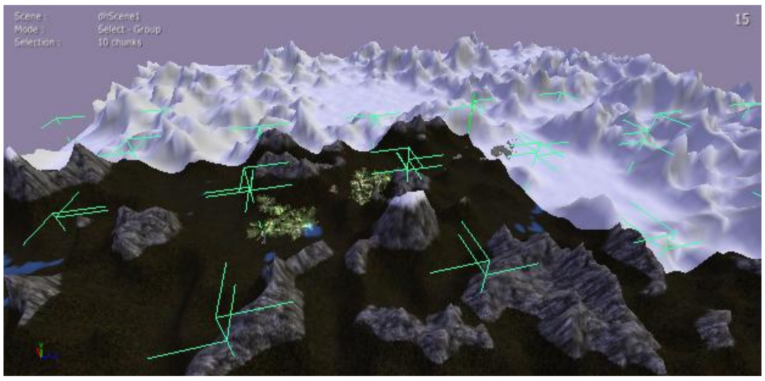
To set up a path, certain nodes on the NavMesh would have to be "de-activated" by using the relevant Function (navigation.enableNode), and also, a starting point would have to be specified (using navigation.setNearestNode). Once the various Nodes have been disabled, the NavMesh may look something like this (where "S" is the starting point, "E" is the ending point, and the red dots are the "enabled" Nodes:



The big question, of course, is how to get from "S" to "E"? I'll be showing you how to do this once I've introduced how to set NavMeshes to Terrain, and gone through the relevant NavMesh Functions.

Adding NavMesh to Terrain

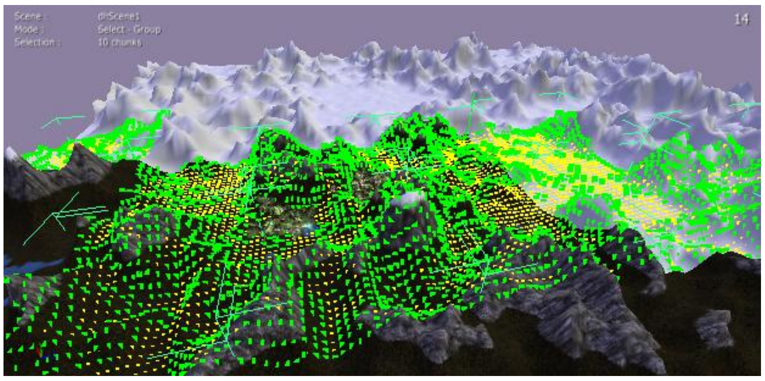
To add a NavMesh to a ShiVa generated Terrain, you need to select the Terrain "Chunks" in the Scene Viewer by clicking on the "Selection" menu, then clicking on the "Terrain chunks" option that appears in the sub-menu. This allows the selection of Terrain "Chunks", as shown by the green lines below:



Scene : dirScene1
Mode : Select - Group
Selection : 10 chunks

15

Once the relevant "Chunks" have been selected, switch to the NavMesh Editor, and select the required distance between Nodes (I've used 5 in the screenshot below), and finally, click on "Add selected objects to NavMesh". Your Terrain will now look something like this (the green and yellow dots are the NavMesh points):



NOTE: processing will take quite some time depending on the size of the Terrain, but once it has been done, the actual usage of the NavMesh is extremely lightweight at runtime!

Accessing the NavMesh from Script

Once you have your NavMesh created, the points can then be accessed via Script using the following set of Functions:

navigation.setTargetNode

Prototype

bOK = navigation.setTargetNode (hObject, hNode)

Parameters

hObject	The Object Handle
hNode	The Node Handle

Return

Boolean	Indicating if the operation is successful

Description

This Function sets the Target Node of the Navigation Controller.

navigation.setAcceleration

Prototype

bOK = navigation.setAcceleration (hObject, nAccel)

Parameters

hObject	The Object Handle
nAccel	The acceleration value

Return

Boolean	Indicating if the operation is successful

Description

This Function sets the acceleration of the Navigation Controller.

navigation.setSpeedLimit

Prototype

bOK = navigation.setSpeedLimit (hObject, nLimit)

Parameters

hObject	The Object Handle
nLimit	The speed limit

Return

Boolean	Indicating if the operation is successful

Description

This Function sets the speed limit of the Navigation Controller.

navigation.setHeightOffset

Prototype

bOK = navigation.setHeightOffset (hObject, nHeight)

Parameters

hObject	The Object Handle
nHeight	The height value

Return

Boolean	Indicating if the operation is successful

Description

This Function sets the height offset of the Navigation Controller.

navigation.getNode

Prototype

hNode = navigation.getNode (hObject)

Parameters

hObject	The Object Handle

Return

Object	The current Node Handle

Description

This Function returns the current Node of the Navigation Controller.

navigation.getTargetNode

Prototype

    hNode = navigation.getTargetNode (hObject)

Parameters

hObject	The Object Handle

Return

Object	The Target Node Handle

Description

This function returns the next Target Node of the Navigation Controller.

navigation.getTargetNodeDistance

Prototype

    nDistance = navigation.getTargetNodeDistance (hObject)

Parameters

hObject	The Object Handle

Return

Number	The distance to the Target Node from the current Node

Description

This Function returns the distance between the Target Node and the current Node of the Navigation Controller.

navigation.getSpeed

Prototype

    nSpeed = navigation.getSpeed (hObject)

Parameters

hObject	The Object Handle

Return

Number	The speed of the Navigation Controller.

Description

This Function returns the speed of the Navigation Controller.

navigation.getVelocity

Prototype

    vx, vy, vz = navigation.getVelocity (hObject)

Parameters

hObject	The Object Handle

Return

Numbers	Each of the velocity components

Description

This Function returns the velocity (in the x, y and z directions) of the Navigation Controller.

navigation.setRandomTargetNode

Prototype

    bOK = navigation.setRandomTargetNode (hObject)

Parameters

hObject	The Object Handle

Return

Boolean	Indicating if the operation is successful

Description

This Function sets a random Target Node for the Navigation Controller.

navigation.setNearestTargetNode

Prototype

    bOK = navigation.setNearestTargetNode (hObject, hOtherObject)

Parameters

hObject	The Object Handle
hOtherObject	The other Object Handle

Return

Boolean	Indicating if the operation is successful

Description

This Function sets the Target Node of the Navigation Controller to be the Node that is the nearest to "hOtherObject".

navigation.setNearestNode

Prototype

bOK = navigation.setNearestNode (hObject, hOtherObject)

Parameters

hObject	The Object Handle
hOtherObject	The other Object Handle

Return

Boolean	Indicating if the operation is successful

Description

This Function sets the current Node of the Navigation Controller to the Node that is the nearest to "hOtherObject".

navigation.setPathMaxLength

Prototype

bOK = navigation.setPathMaxLength (hObject, nMaxLength)

Parameters

hObject	The Object Handle
nMaxLength	The maximum number of Nodes

Return

Boolean	Indicating if the operation is successful

Description

This Function sets the maximum number of Nodes (i.e. length of the Path) for the Navigation Controller.

navigation.enableNodesInBox

Prototype

bOK = navigation.enableNodesInBox (hScene, nBoxMinX, nBoxMinY, nBoxMinZ, nBoxMaxX, nBoxMaxY, nBoxMaxZ, bEnable)

Parameters

hScene	The Scene Handle
nBoxMinX	The first point box X coordinate
nBoxMinY	The first point box Y coordinate
nBoxMinZ	The first point box Z coordinate
nBoxMaxX	The second point box X coordinate
nBoxMaxY	The second point box Y coordinate
nBoxMaxZ	The second point box Z coordinate
bEnable	true = enable ; false = disable

Return

Boolean	Indicating if the operation is successful

Description

This Function enables, or disables, the Navigation Nodes contained in the box described by the Parameters.

navigation.enableNode

Prototype

bOK = navigation.enableNode (hScene, hNode, bEnable)

Parameters

hScene	The Scene Handle
hNode	The Node Handle
bEnable	true = enable ; false = disable

Return

Boolean	Indicating if the operation is successful

Description

This Function enables, or disables, the Navigation Node specified by “hNode”.

OK, so you now know how to set up a NavMesh, and the Functions that are available to access the various parts of the NavMesh. The next step is to put all of this together, and create a simple "Path Finding" system to get from "S" to "E".

This is surprisingly easy to implement. Firstly, place the Object that has the Navigation Controller at NavMesh point "S", and then use the following code:

```
local o = this.getObject ( )   -- this is the Object to be moved (at "S")
local t = this.target ( )      -- this is the location of a "dummy" Object that has been placed at
"E"

-- check to see if "o" has a Navigation Controller
-- if not, set the Node nearest to it (should be "S")
if ( not object.hasController ( o, object.kControllerTypeNavigation ) )
then
    navigation.setNearestNode ( o, o )
end

-- check to see if "o" has a Target Node set
-- if not, set the Target Node to that nearest to "t"
if ( navigation.getTargetNode ( o ) == nil )
then
    navigation.setNearestTargetNode ( o, t )
end
```

That's it! You've created a path from "S" to "E" and set the Object ("o") to follow it. Now all that needs doing is to move the Object towards the Target.

The demo I'll be using for this Chapter is called "TEST_ClickAndGo-20081120", which can be downloaded from the SDN.

So, once again, create a new project in ShiVa, and import the archive.

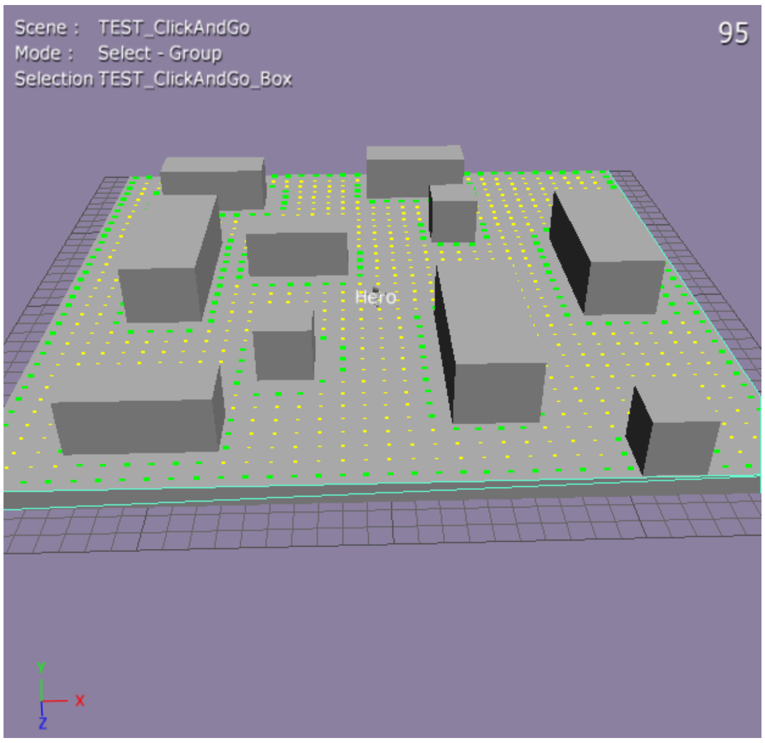
This demo consists of the following files:

Games	-	TEST_ClickAndGo
Models	-	BoxMan TEST_ClickAndGo_Box TEST_ClickAndGo_Dummy
AIModels	-	TEST_ClickAndGo TEST_ClickAndGo_Character
AnimBanks	-	perso
AnimClips	-	run wait walk
Materials	-	BoxMan TEST_ClickAndGo_Box

Meshes	-	perso_Mesh000 TEST_ClickAndGo_Box
Scripts	-	TEST_ClickAndGo_Character_Function_setupAnimations TEST_ClickAndGo_Character_Function_setupDynamics TEST_ClickAndGo_Character_Function_setupNavigation TEST_ClickAndGo_Character_Function_updateAnimations TEST_ClickAndGo_Character_Function_updateDynamics TEST_ClickAndGo_Character_Handler_onEnterFrame TEST_ClickAndGo_Character_Handler_onGoto TEST_ClickAndGo_Character_Handler_onInit TEST_ClickAndGo_Handler_onInit TEST_ClickAndGo_Handler_onMouseButtonDown
Skeletons	-	perso_Skeleton000
Scenes	-	TEST_ClickAndGo

Again, the ones highlighted in Red will be the files that I'll be dissecting here. Since this Chapter is about Navigation, we will start with the Scene:

So, open up the Scene in the Scene Editor, and you should see the following:



The above Scene consists of 10 different sized boxes (all created using the TEST_ClickAndGo_Box Model), our "Hero" (good old "BoxMan"), a large "base" (for everything to sit on), and a grid of yellow and green dots (the NavMesh!). Notice also that the dots are placed in such a way that they are not under any of the boxes. Oh, and you may have noticed that BoxMan is Tagged ("Hero"), which we will be using in the Scripts later. Finally, the large "base" has also been set as a Collider, which is vital if we want to set up the NavMesh!

If you run the demo, all you have to do is click on a "dot", a line will appear from our "Hero" to the "dot", and BoxMan will run to the selected "dot" following the line (hopefully taking the shortest possible route!)

So, how is this wonderful example of "Pathfinding" done? Well, to explain that, I'll have to delve into the Scripts that come with the demo:

<u>TEST_ClickAndGo</u>

This AIModel consists of the following:

Handlers

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

<u>onInit ( )</u>

A nice easy one-liner, which you should be able to work out by now:

application.setCurrentUserScene ("TEST_ClickAndGo")

<u>onMouseButtonDown (nButton, nPointX, nPointY, nRayPntX, nRayPntY, nRayPntZ, nRayDirX, nRayDirY, nRayDirZ)</u>

A little more complex, but all fairly routine stuff, that gets executed when the Mouse button is pressed.

local s = application.getCurrentUserScene ( )

if (s)
then

The first few lines get a Handle to the current Scene, and check that it exists. If it does, then we continue:

    local hero = scene.getTaggedObject (s, "Hero")

    if (hero)
    then

Again, pretty standard stuff. We get a Handle to the Tagged Object (BoxMan), and check if it exists.

local hHitObject, nHitDist, nHitSID = scene.getFirstHitCollider (s, nRayPntX, nRayPntY, nRayPntZ, nRayDirX, nRayDirY, nRayDirZ, 500)

Assuming BoxMan exits, we fire a ray into the Scene at the location of the Mouse pointer, and get the following information:

hHitObject	-	a Handle to any "Collider" Object that the Ray hits
nHitDist	-	the distance the Ray has travelled when it hits the Object
nHitSID	-	the ID of the hit Object

```
        if ( hHitObject )
        then
            local x, y, z = math.vectorScale ( nRayDirX, nRayDirY, nRayDirZ, nHitDist )
            x, y, z = math.vectorAdd ( nRayPntX, nRayPntY, nRayPntZ, x, y, z )

            object.sendEvent ( hero, "TEST_ClickAndGo_Character", "onGoto", x, y, z )
        end
    end
end
```

Finally, we check that an Object has been "hit" and, if so, we do a little Vector math to create a Vector using the Ray direction, the "Hit Distance" (using "vectorScale" to set the size of the Vector), and the Ray location (using the "Scaled" Vector and the location of the Ray). This Vector is then passed to the "*TEST_ClickAndGo_Character.onGoto*" Handler.

That's it for this AIModel. Now we'll move on to the one that actually moves our "Hero":

<u>**TEST_ClickAndGo_Character**</u>

This AIModel consists of the following:

Variables

hDynObject	-	Object	-	Initial Value – 'nil'
hNavObject	-	Object	-	Initial Value – 'nil'
hNavTarget	-	Object	-	Initial Value – 'nil'
nAnimSpeedSmoother	-	Number	-	Initial Value – 0.000

Functions

setupAnimations ( )
setupDynamics ( )
setupNavigation ( )
updateAnimations ( )
updateDynamics ( )

Handlers

onEnterFrame ( )
onGoto (x, y, z)
onInit ( )

setupAnimations ( )

This Function is used to set up the various Animations, which must be arranged in the AnimBank as follows:

0	-	Idle
1	-	Walk
2	-	Run

```
local hObject = this.getObject ( )
```

First things first, we get a Handle to the current Object.

```
if ( object.hasController ( hObject, object.kControllerTypeAnimation ) )
then
    animation.changeClip ( hObject, 0, 0 )
    animation.changeClip ( hObject, 1, 1 )
    animation.changeClip ( hObject, 2, 2 )
```

Next, we check to see if the current Object has an Animation Controller and, if so, we set up the Animation blending starting with the "Idle" Animation (ClipIndex = 0 – remember that the AnimClip Index is zero-based), then the "Walk" Animation (ClipIndex = 1), and, finally, the "Run" Animation (ClipIndex = 2).

```
    animation.setPlaybackKeyFrameBegin ( hObject, 0, animation.getClipKeyFrameRangeMin (
hObject, 0 ) )
    animation.setPlaybackKeyFrameBegin ( hObject, 1, animation.getClipKeyFrameRangeMin (
hObject, 1 ) )
    animation.setPlaybackKeyFrameBegin ( hObject, 2, animation.getClipKeyFrameRangeMin (
hObject, 2 ) )

    animation.setPlaybackKeyFrameEnd ( hObject, 0, animation.getClipKeyFrameRangeMax (
hObject, 0 ) )
    animation.setPlaybackKeyFrameEnd ( hObject, 1, animation.getClipKeyFrameRangeMax (
hObject, 1 ) )
    animation.setPlaybackKeyFrameEnd ( hObject, 2, animation.getClipKeyFrameRangeMax (
hObject, 2 ) )
```

The next step is to set up the beginning and ending KeyFrames for the three Animations. Note the use of "*getClipKeyFrameRangeMin*" and "*getClipKeyFrameRangeMax*" to obtain the start and end respectively of the various AnimClips.

```
    animation.setPlaybackMode ( hObject, 0, animation.kPlaybackModeLoopMirrored )
else
    log.warning ( "TEST_ClickAndGo_Character : Missing animation bank for object ",
this.getObject ( ) )
end
```

Finally, we set the Playback Mode of the Animation, starting with Blend Layer "0" ("Idle"), and using a mirrored loop to ensure that the Animation continues playing (albeit in mirror when it gets to the end!).

setupDynamics ( )

This Function is used to set up the Dynamics of the Character.

```
local hObject = this.getObject ( )
local hScene = object.getScene ( hObject )

if ( hScene ~= nil )
then
    local o = scene.createRuntimeObject ( hScene, "TEST_ClickAndGo_Dummy" )
```

OK, fairly standard stuff to begin. We get handles to the current Object, and the Scene that contains it. We then check to ensure the Scene exists and, if it does, then we create a "Dummy" Object using "*TEST_ClickAndGo_Dummy*".

```
    if ( o ~= nil )
    then
        local nRadius = 0.25

        object.matchTranslation ( o, hObject, object.kGlobalSpace )
        object.translate ( o, 0, nRadius, 0, object.kGlobalSpace )
```

Next, assuming the "Dummy" Object is created successfully, we match its Translation to the current Object, allowing a small shift in height **upwards** of "0.25" units.

```
        if ( dynamics.createSphereBody ( o, nRadius ) )
        then
            dynamics.enableDynamics ( o, true )
            dynamics.enableCollisions ( o, true )
            dynamics.enableGravity ( o, true )
            dynamics.setMass ( o, 80 )

            this.hDynObject ( o )
        end
    end
end
```

Finally, we create a Sphere shaped Dynamics Body using a radius of "0.25", and enable Dynamics, Collisions and Gravity. We also set its Mass to "80"kg, and store it in "*hDynObject*" for future use.

The final "setup" Script deals with the actual Navigation:

setupNavigation ( )

This Function is used to set up the Navigation behaviour of the Character.

```
local hObject = this.getObject( )
local hScene= object.getScene ( hObject )

if (hScene ~= nil )
then
   local o = scene.createRuntimeObject ( hScene, "TEST_ClickAndGo_Dummy" )
```

The first few lines are identical to those in the previous Script, so I won't say anymore.

```
   if (o~= nil )
   then
      object.matchTranslation ( o, hObject, object.kGlobalSpace )
```

Moving on, we check to see that our "Dummy" Object has been created successfully and set its Translation to match the current Object.

```
      if ( navigation.setNearestNode ( o, hObject ) )
      then
         navigation.setAcceleration( o, 1000 )
         navigation.setSpeedLimit( o, 5 )

         this.hNavObject ( o )
      end
   end
```

The next step checks if a Nearest Node exists for the Object and, if so, we set the Acceleration and Speed Limit of the Navigation Object. Finally, we save it to "*hNavObject*" for future reference.

```
   local o2= scene.createRuntimeObject ( hScene, "TEST_ClickAndGo_Dummy" )

   if( o2 ~= nil )
   then
      this.hNavTarget ( o2 )
   end
end
```

The final lines of this Script create another "Dummy" Object, check that it has been created, and save it in the LOCAL Variable "*hNavTarget*".

Now, we'll move on to the "update" Scripts, where everything happens:

updateAnimations ( )

This Function is used to update the Animation of the Character.

```
local hObject = this.getObject( )
local hDynObj = this.hDynObject ( )

if ( hDynObj and object.hasController ( hObject, object.kControllerTypeAnimation ) )
then
```

Here we go again! All we are doing is setting up our Handles to the current Object and the Dynamic Object, and then checking that the Dynamic Object exists, and that the current Object has an Animation Controller attached.

```
    local dt = application.getLastFrameTime ( )
    local vx, vy, vz = dynamics.getLinearVelocity ( hDynObj, object.kGlobalSpace )
    local vLength= math.vectorLength ( vx, 0, vz )
```

Now we start getting into the fun bits. Firstly, we get the last Frame time, next, we obtain a Vector containing the Linear Velocity of the Dynamic Object in GLOBAL space, and finally, we get the length of the Linear Velocity Vector in the x and z axes only (since we are not moving up or down, we don't need the y axis).

```
    local nMaxSpeed= 3
    local nSSmoother = this.nAnimSpeedSmoother ( )
    local nSSFactor= 10

    if ( nSSmoother > vLength )
    then
        nSSFactor = 3
    end

    local nSpeed = math.interpolate ( nSSmoother, vLength, nSSFactor * dt )
    local f= math.clamp ( nSpeed / nMaxSpeed, 0, 1 )
```

Now for some math! All that the above lines are doing is setting the speed for the movement of the Character. Firstly, we are setting the maximum speed to be "3". Next, we get the value of the AIModel Variable "*nAnimSpeedSmoother*", which will be used to make the speed more even. The third line sets another LOCAL Variable "*nSSFactor*" to "10" (this will be used as the factor in the interpolation of the speed by Frame time).

The next section basically checks if the value of "*nSSmoother*" is greater than the length of the Vector we obtained above and, if so, we restrict "*nSSFactor*" to "3" (to limit the interpolation).

Finally, the last two lines interpolate the speed between "*nSSmoother*" and "*vLength*" using the factor calculated above ("*nSSFactor*") as the multiplier for the Frame time, and then set another LOCAL Variable ("*f*") to a value that is between "0" and "1", based on the value of "*nSpeed*" divided by "*nMaxSpeed*" (ie: the ratio that the actual speed has to the maximum speed).

```
if ( nSpeed < 0.001 )
then
    nSpeed = 0
end

this.nAnimSpeedSmoother ( nSpeed )
```

The next step is to see if the value of "*nSpeed*" is less than "0.001" and, if so, set it to "0". This is done to cancel minor movements, and eradicate any small "jerky" movements. The last line above simply sets the value of the AIModel Variable "*nAnimSpeedSmoother*" to the calculated value of "*nSpeed*", for later use.

```
animation.setPlaybackLevel ( hObject, 0, math.max ( 0, 1 - 2 * f ) )
```

All this line does is set the Playback Level of the Animation of our Object to the higher value of "0", and a calculated value using the LOCAL Variable "*f*" calculated above (1 – 2 * f). Remember that "*f*" is clamped between "0" and "1", and as such this calculation will return a value between "-1" (1 – 2 * 1), and "1" (1 – 2 * 0), hence the value returned by the "*math.max*" Function will be between "0" and "1".

```
local walkAndRunLoopSpeed = nSpeed * animation.getClipKeyFrameRangeMax ( hObject,
1 )
local walkLevel = 0
local runLevel= 0
```

OK, here we are setting the speed of the loop that will be used for the "Walk" and "Run" Animations, based on the value of "*nSpeed*" and the maximum of the Key Frame range for our Object's AnimClip "1". We have also created two new LOCAL Variables and set their values to "0".

```
if ( f > 0.3 )
then
    runLevel= ( f - 0.3 ) / 0.7
    walkLevel = 1 - runLevel
    walkAndRunLoopSpeed = walkAndRunLoopSpeed * ( 1 - 0.5 * runLevel )
```

Next, we check if the value of "*f*" is greater than "0.3" and, if so, we firstly set the value of "*runLevel*" to be equal to (*f* – 0.3) divided by 0.7. This will give us a range of approximately 0.1/0.7 (0.14 approx.) to 0.7/0.7 (ie: 1). Secondly, we set the value of "*walkLevel*" to be equal to 1 minus the value of "*runLevel*" (0.86 approx. to 0). Finally, we set the value of the "*walkAndRunLoopSpeed*" Variable to equal the current value of this Variable multiplied by 1 – 0.5 * *runLevel* (0.57 approx. to 0.5).

```
else
    walkLevel = f / 0.3
end
```

However, if "*f*" is less than or equal to "0.3", then we just divide "*f*" by "0.3", which gives us a range of 0 to 1 for the "*walkLevel*".

All that the above code is doing is setting boundaries for when the Object should play its "Run" Animation, and when it should play its "Walk" Animation. Basically, this is so that the Object will "Walk" up to a certain point, then "Run" if necessary, and then slow down to a "Walk" when it closes in on its destination. So, if the destination is close, it will not "Run" at all, but if the destination is far away, then it will start with a "Walk", break into a "Run", and then slow down to a "Walk" again.

```
    animation.setPlaybackLevel( hObject, 1, walkLevel )
    animation.setPlaybackSpeed( hObject, 1, math.max ( 20, walkAndRunLoopSpeed ) )
    animation.setPlaybackLevel( hObject, 2, runLevel )
    animation.setPlaybackSpeed( hObject, 2, math.max ( 20, walkAndRunLoopSpeed ) )
    animation.matchPlaybackCursor ( hObject, 2, 1 )
end
```

Finally, we just set the relevant Playback Levels and Playback Speeds for the "Walk" and "Run" Animations, based on the "Levels" calculated above, and the value of "*walkAndRunLoopSpeed*" (with a minimum value of "20").

That's it for updating the Animations, now we need to update the Dynamics of the Character, to actually get it to move!

updateDynamics ( )

This Function is used to update the Dynamics of the Character.

```
local hObject = this.getObject( )
local hDynObj = this.hDynObject ( )
local hNavObj = this.hNavObject ( )
```

To begin, we create the LOCAL Variables to store the Handle to the following:

hObject	-	the current Object
hDynObj	-	the Dynamic Object
hNavObj	-	the navigation Object

```
if ( hNavObj and hDynObj )
then
    local dt = application.getLastFrameTime ( )
```

Now, we check that the navigation and Dynamics Objects exist, and set a LOCAL Variable "*dt*" with the last Frame time.

```
    local fx, fy, fz = 0, 0, 0
    local dist = object.getDistanceToObject ( hObject, hNavObj )
```

The next step is to create some more LOCAL Variables which will be used to compute the Force required to move the Dynamics Object, including getting the distance between the current Object and the navigation Object.

```
    if ( dist > 0.25 )
    then
        local x1, y1, z1 = object.getTranslation ( hDynObj, object.kGlobalSpace )
        local x2, y2, z2 = object.getTranslation ( hNavObj, object.kGlobalSpace )

        fx, fy, fz = math.vectorSubtract( x2, 0, z2, x1, 0, z1 )
        fx, fy, fz = math.vectorSetLength ( fx, fy, fz, dist * 3 )
    end
```

The next phase of our calculation checks to see if the distance between the two Objects calculated above is less than "0.25" (ie: the radius of the Dynamics Object) and, if so, we get the relative Translations of the two Objects (in GLOBAL space), and then create a Vector, again using just the x and z components, with a length of 3 times the distance.

```
    if ( math.vectorLength ( fx, fy, fz ) > 0 )
    then
        fx, fy, fz = math.vectorScale ( fx, fy, fz, 1000 )
```

Now, we check that the length of this Vector is greater than "0" and, assuming it is, we scale it by a factors of "1000".

```
        dynamics.addForce ( hDynObj, fx, fy, fz, object.kGlobalSpace )
        dynamics.setLinearDampingEx ( hDynObj, 10,0,10 )
```

OK, we now have a Vector (fx, fy, fz), which we can use as a Force to move the Dynamics Object, and we also add some Linear Damping, just to ensure smooth movement.

```
        local tx, ty, tz = object.getTranslation ( hObject, object.kGlobalSpace )

        object.lookAt ( hObject, tx + fx, ty, tz + fz, object.kGlobalSpace, 5 * dt )
```

Finally, we modify the orientation of the current Object by getting its current Translation, and then we make it rotate to face the direction calculated by the following:

tx + fx	-	current Translation in the x axis plus the x Force calculated above
ty	-	current Translation in the y axis – Note that there is no Force component
tz + fz	-	current Translation in the z axis plus the z Force calculated above

```
    else
        -- Force the object to stop using damping
        --
        dynamics.setLinearDampingEx ( hDynObj, 1000, 0, 1000 )
    end
end
```

If the Vector length calculated above is not greater than zero, then we just need to stop the Dynamics Object from using Damping, by setting the Linear Damping to high values.

onEnterFrame ( )

This Handler is called at the beginning of each Frame.

```
this.updateDynamics ( )
this.updateAnimations ( )
```

The first thing we do here is to make calls to the Functions that update the Dynamics and the Animations.

```
local hObject = this.getObject( )
local hDynObj = this.hDynObject ( )
```

Next, we get Handles to the current Object ("hObject"), and the Dynamic Object ("hDynObj").

```
if ( hDynObj ~= nil )
then
    local nRadius = 0.25

    object.matchTranslation ( hObject, hDynObj, object.kGlobalSpace )
    object.translate( hObject, 0, -nRadius, 0,object.kGlobalSpace )
end
```

Finally, we check to ensure that the Dynamic Object exists and, if so, we "move" the current Object to the position of the Dynamic Object. Note that the Translation is carried out to a position that is "0.25" **below** the current height of the Object. This is done to ensure that the current Object has its feet on the floor (literally!).

onGoto (x, y, z)

This Handler is called when the Mouse button has been clicked, and takes three Parameters (x, y and z) which represent a Vector.

```
local hNavObj = this.hNavObject ( )
local hNavTgt = this.hNavTarget ( )
```

Firstly, we obtain Handles to the current Object ("hNavObject"), and the Target ("hNavTarget").

```
if ( hNavObj and hNavTgt )
then
    object.setTranslation ( hNavTgt, x, y, z, object.kGlobalSpace )
    navigation.setNearestTargetNode ( hNavObj, hNavTgt )
end
```

Next, we check that they both exist and, if so, we set the Translation of the Target Object to be equal to the input Parameter Vector (represented by x, y and z), and we set the nearest Target Node in the NavMesh between the current Object and the Target Object.

onInit ( )

This Handler is called once when the AIModel is initialised.

```
this.setupDynamics ( )
this.setupAnimations ( )
this.setupNavigation ( )
```

An extremely simple Script that calls the three "setup" Scripts described above.

And that, Ladies and Gentlemen, is that (well, as far as this Chapter goes!). Before we get stuck into our Game, I'd like to take a little side trip into XML and ShiVa. We won't be using ShiVa's XML capabilities in the Game, but it does have lots of other uses such as "High Score Tables" etc.!