How To Move Entities

BigWorld Technology 2.1. Released 2012.

Software designed and built in Australia by BigWorld.

Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Introduction

Moving entities is one of the most fundamental processes in implementing a game. BigWorld provides two main mechanisms to accomplish this task: navigation and seeking.

- **Navigation**

  This is the primary way to move NPCs (non-player characters) around the world. It uses a map of navigation information generated in an offline process, and can take into account the size of the entity.

- **Seek**

  This is a client-side function used to line up the player for interactions with other entities.

Navigation can take into account the size of the navigating object, and thus follow paths that do not pass too close to obstacles. The pre-generated navigation information is known as the navigation mesh (navmesh), as it is a collection of polygons generated in an offline tool called NavGen (for details, see the Content Tools Reference Guide's chapter *NavGen*).

Note

The NavGen's executable `navgen.exe` can be found under the `bigworld/tools/misc` folder.

This document describes the steps necessary to make an entity navigate from one point to another in a sensible manner, taking into account their environment.

Chapter 2. Navigation Mesh

Use of the server-side navigation functions require the navigation mesh (or navmesh) be generated for the space. The navmesh is a collection of interconnected convex polygons parallel to the XZ plane. These *navpolys* are generated using the tool NavGen, and are stored in the `.cdata` chunk files of the space.

Note

The chunk files are stored under the `<res>/spaces/<space>`, and contain binary terrain and lighting data.

For details on the information held by this and other chunk files, see Client Programming Guide's section *Chunks* → “Implementation files”.

For details on this and other binary files' grammar, see File Grammar Guide's section `BinSection` *files*.

For details on the generation of navmesh, see the document Content Tools Reference Guide's section *NavGen* → “Generating the navmesh”.

Chapter 3. How to Move Server-Controlled Entities

The server can move entities via either navigation functions, or a simple `moveToPoint`.

Navigation provides full path finding using special mesh data generated by NavGen, while `moveToPoint` simply moves the entity in a straight line, without taking obstacles or terrain into account.

3.1. `moveToPoint`

This is the simplest movement system available — taking a destination, it moves the entity in a straight line until that point is reached.

An example of an entity that uses this mechanism is the `MovingPlatform`, which follows a series of patrol nodes, using `moveToPoint` at each one to move to the next.

```
self.moveToPoint( self.patrolNode[1], self.travelSpeed, 0, self.faceDirection,
 True )
```

`cell/MovingPlatform.py`

Note

For details on the `MovingPlatform` entity, see the document How To Build a Server-Controlled Moving Platform.

3.2. Navigation

Navigation is a path-finding service available to entities running on the CellApps. Navigation uses a heuristically guided breadth first search (A*), initially across the chunks, and then in the navigation mesh within the chunks.

For detailed information on all functions below, see the CellApp Python API.

3.2.1. `canNavigateTo()`

Before using any `navigate` function, you should check that the destination can be reached. The function `canNavigateTo` finds the nearest point to the destination that can be reached by traversing the navigation mesh or `None` if no path can be found. If navigation is attempted to a point that cannot be reached, then an exception will be raised by the `navigateStep` function.

3.2.2. `navigateStep()`

This function creates a movement controller that moves the entity toward the destination. Each time the entity enters a new navpoly, or travels the specified maximum distance, the controller stops and calls the `onMove` callback.

Note

The paths generated by calling `navigateStep` are cached, making subsequent calls to the same destination inexpensive.

To reach the destination, you will have to re-call `navigateStep` each time the entity stops.

```
self.controllerId = self.navigateStep( destination, velocity,
 maximumMovement )
```

cell/Guard.py

3.2.3. navigateFollow()

NavigateFollow is an older function for navigating relative to another entity's current position. New code should use `navigateStep`.

3.2.4. navigate()

Navigate is an older function for navigating entities across spaces. New code should use `navigateStep`.

3.3. Example navigating cell entity

To demonstrate the navigation mechanism, we have constructed a simple example. The example entity randomly picks a location around its current position, then navigates to it. Upon arrival, it chooses a new destination and continues.

3.3.1. Loading

The entity could be created before the rest of the chunk data is loaded. If you use navigation immediately in the `__init__` method, then the start location might be unresolved, causing an exception. Instead, we wait for the navigation mesh to load, using a timer and testing with `canNavigateTo`.

```
def __init__( self ):
 BigWorld.Entity .__init__( self )
 self.destination = self.position
 self.addTimer( 5.0, 0, RandomNavigator.TIMER_WAITING_FOR_NAVMESH )

def onTimer(self, timerId, userId):
 if self.canNavigateTo( self.position ) == None:
  self.addTimer( 5.0, 0, RandomNavigator.TIMER_WAITING_FOR_NAVMESH )
 else:
  self.navigateStep( self.destination, 5.0, 10.0 )
```

cell/RandomNavigator.py

Example navigation during chunk data load

3.3.2. Moving

Calling the first `navigateStep` will result in the `onMove` callback being triggered. At this time, the entity may or may not have reached its destination, so we check how close the entity is. In this example, we require it to be within 0.1 metre of the target before picking a new destination.

Note the use of `canNavigateTo` — this function clamps the destination to the point closest to the destination, and that is accessible via the navigation mesh. The entity then perpetually follows this cycle of picking a destination, running to it and then picking another.

```
def onMove(self, controllerId, userId):
 if ( self.position - self.destination ).length > 0.1:
  self.navigateStep( self.destination, 5.0, 10.0 )
 else:
  self.destination = None
  while self.destination == None:
   randomDestination = (
    self.position.x + random.randrange(-400, 400, 1.0),
    self.position.y,
    self.position.z + random.randrange(-400, 400, 1.0) )
   self.destination = self.canNavigateTo( randomDestination )

  self.navigateStep( self.destination, 5.0, 10.0 )
```

cell/RandomNavigator.py

3.3.3. Client-side

To be able to correctly display the entity on the client machine, we require two things:

- A model.
- The correct filter.

The default filter is `DumbFilter`, which simply places the entity at the location most recently received from the server, thus producing a stuttering motion as it moves about the world. You might also notice that its height above the ground appears to go up in steps — this is the movement of the entity on the server as it traverses the navigation mesh covering slopes.

Instead, we will use `AvatarDropFilter`, which produces fluid movement for the Action Matcher, with the addition that it locks the entity to the ground. For details on `AvatarDropFilter`, see “`AvatarDropFilter`” on page 15 .

```
 def onEnterWorld( self, prereqs ):
  self.model = BigWorld.Model( RandomNavigator.stdModel )
  BigWorld.addShadowEntity( self )
  self.filter = BigWorld.AvatarDropFilter()

 def onLeaveWorld( self ):
  BigWorld.delShadowEntity( self )
  self.model = None
```

client/RandomNavigator.py

3.4. Replacing deprecated navigation methods

To show the method of converting an entity using either `navigate` or `navigateFollow` to use navigateStep, we have provided a simple example. It has two behaviours, gated by a `think` method: A patrol behaviour, which moves between two `UserDataObjects` using `navigate` and a follow behaviour, which follows a supplied entity ID, but stops and waits if it gets too close, using `navigateFollow`.

3.4.1. `navigateFollow()`

`navigateFollow` was used to path to a point relative to an entity. Here is a simple example of using `navigateFollow` to follow an identified entity around.

```
 # Follow brain
  def follow( self ):
   # If self.targetID doesn't exist, switch to patrol mode
   if not BigWorld.entities.has_key( self.targetID ):
    self.stopFollow()
    return

   # If target isn't in this space, switch to patrol mode
   target = BigWorld.entities[ self.targetID ]
   if target.spaceID != self.spaceID:
    self.stopFollow()
    return

   # If we've arrived, wait here for target to move away
   if self.closeEnoughToTarget():
    self.cancel( "Movement" )
    self.addTimer( 5 )
    return

   # Follow our target
   target = BigWorld.entities[ self.targetID ]
   try:
    self.navigateFollow( target, FOLLOW_ANGLE, FOLLOW_DISTANCE, VELOCITY, 500,
  500, True, 0.5 )
   except ValueError, e:
    # No path found
    self.cancel( "Movement" )
    self.addTimer( 5 )

  def closeEnoughToTarget( self ):
   target = BigWorld.entities[ self.targetID ]
   return distance( self.position, target.position ) <= FOLLOW_DISTANCE

  def onMove( self, controllerID, userData ):
   self.think()
```

`cell/ElPolloDiablo.py` - Before

Since `navigateFollow` and `navigateStep` share the same controller internally, the changes are quite simple. We replace the block of code marked `# Follow our target` with the following block of code:

```
  # Follow our target
  yaw = target.yaw + FOLLOW_ANGLE
  offset = ( FOLLOW_DISTANCE * math.sin( yaw ), 0, FOLLOW_DISTANCE *
math.cos( yaw ) )
  dest = self.canNavigateTo( target.position + offset, 500, 0.5 )
  if dest is None:
   # No path found
   self.cancel( "Movement" )
   self.addTimer( 5 )
   return
  self.navigateStep( dest, VELOCITY, 500, 500, True, 0.5 )
```

`cell/ElPolloDiablo.py` - After

The new calculation of `dest` is the same as that performed by `navigateFollow` on the supplied entity. We've also made use of the `canNavigateTo` method to ensure that we attempt to navigate to a reachable spot, now that we have access to the destination point.

As should be clear here, `navigateStep` is a more flexible interface to `navigateFollow`'s existing behaviour, giving more control over movement behaviour and exposing the intended destination to Python scripting code.

3.4.2. `navigate()`

`navigate` was used to start an entity moving to a target point, pathing along the navigation mesh, and would callback to the entity when either the target point was reached, or if the navigation mesh was unable to find a suitable path. Here is a simple example of using `navigate` to path between two points.

```
# Patrol brain
def patrol( self ):
 # If we haven't got any nodes, find a pair
 if self.nextNode is None:
  self.setupNodes()
 if self.nextNode is None:
  # If we can't find a pair of nodes, wait 5 seconds and try again
  self.cancel( "Movement" )
  self.addTimer( 5 )
  return

 # If we've arrived, turn around
 if self.closeEnoughToNode():
  self.swapNodes()

 # Navigate to slightly closer than self.closeEnoughToNode()
 self.navigate( self.nextNode.position, VELOCITY, True, 500, 0.5,
PATROL_DISTANCE * 0.8 )

def closeEnoughToNode( self ):
 target = self.nextNode
 return distance( self.position, target.position ) <= PATROL_DISTANCE

def setupNodes( self ):
 self.prevNode = None
 self.nextNode = None
 closest = None
 dist = 500
 for i in BigWorld.userDataObjects.values():
  if i.__class__.__name__ != "PatrolNode" or len(i.patrolLinks) == 0:
   continue
  if distance( self.position, i.position ) < dist:
   closest = i
   dist = distance( self.position, i.position )
 if closest is not None:
  after = closest.patrolLinks[ 0 ]
  while distance( closest.position, after.position ) < PATROL_DISTANCE * 3:
   after = after.patrolLinks[ 0 ]
   if after is None or after.uuid == closest.uuid:
    after = None
    break
  if after is not None:
   self.prevNode = closest
   self.nextNode = after

def swapNodes( self ):
 temp = self.nextNode
 self.nextNode = self.prevNode
 self.prevNode = temp

def onNavigate( self, controllerID, userData ):
 # Arrived. Turn around.
 self.swapNodes()
 self.think()

def onNavigateFailed( self, controllerID, userData ):
 # Can't get there. Turn around
 self.swapNodes()
 self.think()
```

`cell/ElPolloDiablo.py` - Before

Similarly to replacing `navigateFollow`, we can replace the use of `navigate` very simply. We replace the block of code marked `# Navigate to slightly closer than self.closeEnoughToNode()` with the following block of code:

```
# Navigate towards self.nextNode.position
dest = self.canNavigateTo( self.nextNode.position, 500, 0.5 )
if dest is None:
 # No path found
 self.cancel( "Movement" )
 self.addTimer( 5 )
 return
self.navigateStep( dest, VELOCITY, 500, 500, True, 0.5 )
```

`cell/ElPolloDiablo.py` - After

We use the `canNavigateTo` method to identify a waypoint position on the navigation mesh that matches our desired destination, and then attempt to navigate towards it. This is similar to what `navigate` does internally, except that we receive an `onMove` callback every time we enter a new navpoly, or have travelled the specified maximum distance.

These more frequent callbacks allow the script-level code to control navigation at a finer level, dealing with changed priorities or moving targets without needing to either be triggered by an external event or wait until the previously-selected destination is reached.

At this point, the `onNavigate` callback is unused, and can be removed. `onNavigateFailed` can be renamed to `onMoveFailure`. `onMove` remains unchanged, and simply calls `self.think()` as seen in the above `navigateFollow` example.

If `navigateStep` is called again during the `onMove` callback, the existing `Controller` will be reused if possible. This helps to ensure that calling `navigateStep` more frequently than `navigate` would be for the same navigation activity does not cause extra load on the CellApp through repeated `Controller` object creation.

Chapter 4. Filters

Although an in-depth look at filters is beyond the scope of this document, it is important to mention their existence at this point.

Filters process position and rotation updates from the server into a smooth movement on the client machine. They can also be used to make assumptions about the movement of entities, as is the case with `AvatarDropFilter`[1].

4.1. `AvatarFilter`

This filter produces movement on the client that corresponds to the one on the server.

Use this filter for entities that do not remain stuck to the ground, such as other players and flying vehicles.

4.2. `AvatarDropFilter`

This filter places the client-side entity on the ground, even if the server places it in the air.

It is suitable for entities using navigation, as the navigation mesh is always slightly raised above the terrain.

[1]For details, see the Client Python API's entries Class List → `AvatarDropFilter` and Class List → `AvatarFilter`.

Chapter 5. How to Move Client-Controlled Entities

All client-side movement is done using the physics object that acts like a controller, sending position updates to the server.

The navigation mesh is not present on the client, so the functions `seek()` and `chase()` must be used. They provide simple direct movement, following the terrain and colliding with obstacles.

5.1. seek()

5.1.1. Mouse click movement

In this example we will use `seek` to implement a simple mouse click-based movement.

To access this functionality in the FantasyDemo, press `Z` to bring up the cursor, and right-click on the terrain to move.

```
def moveKey( self, isDown ):
 if isDown:
  mp = GUI.mcursor().position
  type, target = collide.collide( mp.x, mp.y )
  if type == collide.COLLIDE_TERRAIN:
   self._movePlayer( target )
  elif type == collide.COLLIDE_ENTITY:
   self._movePlayer( target.position )

def _movePlayer( self, position ):
 player   = BigWorld.player()
 velocity = player.runFwdSpeed
 timeout  = 1.5 * (position - player.position).length / velocity
 curr_yaw = (position - player.position).yaw
 destination = (position[0], position[1], position[2], curr_yaw)
 player.physics.velocity = (0, 0, velocity)
 player.physics.seek( destination, timeout, 10, self._seekCallback )
 self.isMoving = True
```

`client/MouseControl.py`

Note

The destination needed by `seek()` is a four-member tuple containing the position and yaw.

5.1.2. Coordinated actions

The `seek` method is often used in conjunction with coordinated actions. These are actions involving two models, such as a handshake.

The position and yaw needed for the actions to line up can be extracted from the action, as in the following excerpt:

```
self.physics.seek( partner.model.Shake_B_Accept.seekInv, 5.0, 0.10, onSeek )
self.physics.velocity = ( 0, 0, self.walkFwdSpeed )
```

`client/Avatar.py`

5.2. chase()

To demonstrate the `chase` function we will implement a `/follow` chat console command in FantasyDemo. The command will cause the player to follow the targeted entity, until they press a movement key breaking the pursuit.

5.2.1. Handling the command

The Fantasy Demo chat console will automatically resolve the typed '/follow' command to a function call. All we need to do is add the following function to the `ConsoleCommands.py` module.

```
def follow( player, string ):
    # Follow the current target
    if BigWorld.target() != None:
        player.physics.chase( BigWorld.target(), 2.0, 0.5 )
        player.physics.velocity = ( 0, 0, 6.0 )
```

`client/Helpers/ConsoleCommands.py`

5.2.2. Stopping the pursuit

To cancel the chase action, we need to add the code below to `PlayerAvatar`'s `moveForward`, `moveBackward`, `moveLeft` and `moveRight` functions as in the excerpt below.

```
def moveForward(self, isDown):
    if isDown:
        if self.mouseControl.isMoving:
            self.mouseControl.cancel()

        if self.physics.chasing:
            self.physics.stop()

        self.forwardMagnitude = min(self.forwardMagnitude+1.0,1.0)
        if self.mode == Mode.COMBAT_CLOSE:
            if self.stance == Avatar.STANCE_BACKWARD:
                nst = Avatar.STANCE_NEUTRAL
            else:
                nst = Avatar.STANCE_FORWARD
            self.takeStance( nst )
    else:
        self.forwardMagnitude = max(self.forwardMagnitude-1.0,-1.0)
```

`client/Avatar.py`

Appendix A. Source files

Table of Contents

A.1. <*res*>/scripts/cell/RandomNavigator.py

```
import BigWorld
import math
import random
import Math


class RandomNavigator( BigWorld.Entity ):

    TIMER_WAITING_FOR_NAVMESH = 1

    #-----------------------------------------------------------------------
    # Constructor.
    #-----------------------------------------------------------------------
    def __init__( self ):
        BigWorld.Entity .__init__( self )
        self.destination = self.position
        self.addTimer( 5.0, 0, RandomNavigator.TIMER_WAITING_FOR_NAVMESH )

    #-----------------------------------------------------------------------
    # This method is called when a timer expires.
    #-----------------------------------------------------------------------
    def onTimer(self, timerId, userId):
        if userId == RandomNavigator.TIMER_WAITING_FOR_NAVMESH:
            if self.canNavigateTo( self.position ) == None:
                self.addTimer( 5.0, 0,
 RandomNavigator.TIMER_WAITING_FOR_NAVMESH )
            else:
                self.navigateStep( self.destination, 5.0, 10.0 )


    #-----------------------------------------------------------------------
    # This method is called when we've finished moving to a point.
    #-----------------------------------------------------------------------
    def onMove(self, controllerId, userId):
        if ( self.position - self.destination ).length > 0.1:
            self.navigateStep( self.destination, 5.0, 10.0 )
        else:
            self.destination = None
            while self.destination == None:
                randomDestination = (
                    self.position.x + random.randrange(-400, 400, 1.0),
                    self.position.y,
                    self.position.z + random.randrange(-400, 400, 1.0) )
                self.destination = self.canNavigateTo( randomDestination )

            self.navigateStep( self.destination, 5.0, 10.0 )

# RandomNavigator.py
```

A.2. <*res*>/scripts/client/RandomNavigator.py

```
import math
import BigWorld
from keys import *


# ----------------------------------------------------------------------------
# Section: class RandomNavigator
# ----------------------------------------------------------------------------


class RandomNavigator( BigWorld.Entity ):
    stdModel = 'characters/avatars/base/base.model'

    def __init__( self ):
        BigWorld.Entity.__init__( self )


    def prerequisites( self ):
        return [ RandomNavigator.stdModel ]


    def enterWorld( self ):
        self.model = BigWorld.Model( RandomNavigator.stdModel )
        BigWorld.addShadowEntity( self )
        self.targetCaps = [ CAP_CAN_HIT , CAP_CAN_USE ]
        self.filter = BigWorld.AvatarDropFilter()


    def leaveWorld( self ):
        BigWorld.delShadowEntity( self )
        self.model = None


    def use( self ):
        pass


#RandomNavigator.py
```

A.3. <*res*>/scripts/base/RandomNavigator.py

```
import FantasyDemo

# ----------------------------------------------------------------------------
# Section: class RandomNavigator
# ----------------------------------------------------------------------------

class RandomNavigator( FantasyDemo.Base ):
    def __init__( self ):
        FantasyDemo.Base.__init__( self )


# RandomNavigator.py
```

A.4. <*res*>/scripts/editor/RandomNavigator.py

```
class RandomNavigator:
    def modelName( self, props ):
        return 'characters/avatars/base/base.model'

# RandomNavigator.py
```

A.5. <*res*>/scripts/entity_defs/RandomNavigator.def

```
<root>
    <Volatile>
        <position/>
        <yaw/>
    </Volatile>

    <Properties>

        <destination>
            <Type>            PYTHON                <Type>
            <Flags>            CELL_PRIVATE        </Flags>
        </destination>

    </Properties>

    <ClientMethods>
    </ClientMethods>

    <CellMethods>
    </CellMethods>
</root>
```

A.6. <*res*>/scripts/base/ElPolloDiablo.py

```
 import FantasyDemo

# -------------------------------------------------------------------
# Section: class ElPolloDiablo
# -------------------------------------------------------------------

class ElPolloDiablo( FantasyDemo.Base ):
 pass

# ElPolloDiablo.py
```

A.7. <*res*>/scripts/client/ElPolloDiablo.py

```
import BigWorld
```

```
MODEL = "characters/npc/chicken/chicken.model"

# ------------------------------------------------------------------
# Class ElPolloDiablo:
#
# ElPolloDiablo follows an entity, or wanders between two patrol nodes
# ------------------------------------------------------------------
class ElPolloDiablo(BigWorld.Entity):

 # -----------------------------------------------------------------
 # Method: __init__
 # Description:
 # - Defines all variables used by the entity. This includes
 #   setting variables to None.
 # - Does not call any of the accessor methods. Any variables set are
 #   for the purposes of stability.
 # ------------------------------------------------------------------
 def __init__( self ):
  BigWorld.Entity.__init__( self )
  self.filter = BigWorld.AvatarDropFilter()
  self.model = None


 # ------------------------------------------------------------------
 # Method: prerequisites
 # Description:
 # - Return a list of the resources that we want loaded in the background
 # for us before onEnterWorld() is called.
 # ------------------------------------------------------------------
 def prerequisites( self ):
  return [ MODEL ]


 # ------------------------------------------------------------------
 # Method: onEnterWorld
 # Description:
 # - Creates a model for the ElPolloDiablo.
 # ------------------------------------------------------------------
 def onEnterWorld( self, prereqs ):
  self.model = prereqs[ MODEL ]
  self.model.scale = (4.0, 4.0, 4.0)


 # ------------------------------------------------------------------
 # This method is called when the entity leaves the world
 # ------------------------------------------------------------------
 def onLeaveWorld( self ):
  self.model = None


 # ------------------------------------------------------------------
 # Method: name
 # Description:
 # - Part of the entity interface: This allows the client to get a string
 #   name for the entity.
 # ------------------------------------------------------------------
 def name( self ):
  return "El Pollo Diablo"

#ElPolloDiablo.py
```

A.8. *<res>*`/scripts/entity_defs/ElPolloDiablo.def`

```
<root>
 <Volatile>
  <position/>
  <yaw/>
 </Volatile>

 <Implements>
  <Interface> BaseAndCell </Interface>
 </Implements>

 <Properties>
  <!-- 0 is wander, 1 is follow targetID -->
  <mode>
   <Type>  INT32  </Type>
   <Flags>  CELL_PRIVATE  </Flags>
   <Persistent> false  </Persistent>
   <Default> 0  </Default>
  </mode>

  <targetID>
   <Type>  OBJECT_ID  </Type>
   <Flags>  CELL_PRIVATE  </Flags>
   <Persistent> false  </Persistent>
  </targetID>

  <nextNode>
   <Type>  PATROL_NODE  </Type>
   <Flags>  CELL_PRIVATE  </Flags>
   <Persistent> false  </Persistent>
  </nextNode>

  <prevNode>
   <Type>  PATROL_NODE  </Type>
   <Flags>  CELL_PRIVATE  </Flags>
   <Persistent> false  </Persistent>
  </prevNode>
 </Properties>

 <ClientMethods>
 </ClientMethods>

 <CellMethods>
  <startFollow>
   <Args>
    <id>  OBJECT_ID  </id> <!-- EntityID -->
   </Args>
  </startFollow>

  <stopFollow>
  </stopFollow>
 </CellMethods>

 <BaseMethods>
 </BaseMethods>

</root>
```

A.9. `<res>/scripts/cell/ElPolloDiablo.py` - Before

```
"This module implements the ElPolloDiablo entity."

# BigWorld Modules
import BigWorld

# Python modules
import random
import math

#todo: replace this with math module
def distance(v1, v2):
 "Returns the distance between two 3d vectors"
 x = v2[0] - v1[0]
 z = v2[2] - v1[2]
 #ignore y value due to the current 13k hack
 return math.sqrt(x * x + z * z)


# ----------------------------------------------------------------------------
# Section: class ElPolloDiablo
# ----------------------------------------------------------------------------
class ElPolloDiablo( BigWorld.Entity ):
 "An ElPolloDiablo entity."

 PATROL_MODE  = 0
 FOLLOW_MODE  = 1

 VELOCITY  = 20

 PATROL_DISTANCE  = 2

 FOLLOW_DISTANCE  = 10
 FOLLOW_ANGLE  = math.pi

 #-------------------------------------------------------------------------
 # Constructor
 #-------------------------------------------------------------------------

 def __init__( self ):
  BigWorld.Entity.__init__( self )

  # random yaw
  yaw = random.uniform(-math.pi, math.pi)

  self.direction = (0.0, 0.0, yaw)

  if self.mode == ElPolloDiablo.PATROL_MODE:
   self.stopFollow()
  else:
   self.startFollow( self.targetID )

 def onTimer( self, controllerID, userData ):
  self.think()

 def startFollow( self, targetID ):
  self.mode = ElPolloDiablo.FOLLOW_MODE
  self.targetID = targetID
  self.cancel( "Movement" )
  self.think()
```

```
def stopFollow( self ):
 self.mode = ElPolloDiablo.PATROL_MODE
 self.nextNode = None
 self.cancel( "Movement" )
 self.think()

def think( self ):
 if self.mode == ElPolloDiablo.PATROL_MODE:
  self.patrol()
 else:
  self.follow()


# Patrol brain
def patrol( self ):
 # If we haven't got any nodes, find a pair
 if self.nextNode is None:
  self.setupNodes()
 if self.nextNode is None:
  # If we can't find a pair of nodes, wait 5 seconds and try again
  self.cancel( "Movement" )
  self.addTimer( 5 )
  return

 # If we've arrived, turn around
 if self.closeEnoughToNode():
  self.swapNodes()

 # Navigate to slightly closer than self.closeEnoughToNode()
 self.navigate( self.nextNode.position, ElPolloDiablo.VELOCITY, True, 500,
0.5, ElPolloDiablo.PATROL_DISTANCE * 0.8 )

def closeEnoughToNode( self ):
 target = self.nextNode
 return distance( self.position, target.position ) <=
ElPolloDiablo.PATROL_DISTANCE

def setupNodes( self ):
 self.prevNode = None
 self.nextNode = None
 closest = None
 dist = 500
 for i in BigWorld.userDataObjects.values():
  if i.__class__.__name__ != "PatrolNode" or len(i.patrolLinks) == 0:
   continue
  if distance( self.position, i.position ) < dist:
   closest = i
   dist = distance( self.position, i.position )
 if closest is not None:
  after = closest.patrolLinks[ 0 ]
  while distance( closest.position, after.position ) <
ElPolloDiablo.PATROL_DISTANCE * 3:
   after = after.patrolLinks[ 0 ]
   if after is None or after.uuid == closest.uuid:
    after = None
    break
  if after is not None:
   self.prevNode = closest
   self.nextNode = after

def swapNodes( self ):
   temp = self.nextNode
   self.nextNode = self.prevNode
   self.prevNode = temp

  def onNavigate( self, controllerID, userData ):
   # Arrived. Turn around.
   self.swapNodes()
   self.think()

  def onNavigateFailed( self, controllerID, userData ):
   # Can't get there. Turn around
   self.swapNodes()
   self.think()


  # Follow brain
  def follow( self ):
   # If self.targetID doesn't exist, switch to patrol mode
   if not BigWorld.entities.has_key( self.targetID ):
    self.stopFollow()
    return

   # If target isn't in this space, switch to patrol mode
   target = BigWorld.entities[ self.targetID ]
   if target.spaceID != self.spaceID:
    self.stopFollow()
    return

   # If we've arrived, wait here for target to move away
   if self.closeEnoughToTarget():
    self.cancel( "Movement" )
    self.addTimer( 5 )
    return

   # Follow our target
   target = BigWorld.entities[ self.targetID ]
   try:
    self.navigateFollow( target, ElPolloDiablo.FOLLOW_ANGLE,
  ElPolloDiablo.FOLLOW_DISTANCE, ElPolloDiablo.VELOCITY, 500, 500, True, 0.5 )
   except ValueError, e:
    # No path found
    self.cancel( "Movement" )
    self.addTimer( 5 )

  def closeEnoughToTarget( self ):
   target = BigWorld.entities[ self.targetID ]
   return distance( self.position, target.position ) <=
  ElPolloDiablo.FOLLOW_DISTANCE

  def onMove( self, controllerID, userData ):
   self.think()

# ElPolloDiablo.py
```

A.10. <*res*>/scripts/cell/ElPolloDiablo.py - After

```
"This module implements the ElPolloDiablo entity."

# BigWorld Modules
import BigWorld
```

```
# Python modules
import random
import math

#todo: replace this with math module
def distance(v1, v2):
 "Returns the distance between two 3d vectors"
 x = v2[0] - v1[0]
 z = v2[2] - v1[2]
 #ignore y value due to the current 13k hack
 return math.sqrt(x * x + z * z)


# ----------------------------------------------------------------------------
# Section: class ElPolloDiablo
# ----------------------------------------------------------------------------
class ElPolloDiablo( BigWorld.Entity ):
 "An ElPolloDiablo entity."

 PATROL_MODE  = 0
 FOLLOW_MODE  = 1

 VELOCITY  = 20

 PATROL_DISTANCE  = 2

 FOLLOW_DISTANCE  = 10
 FOLLOW_ANGLE  = math.pi

 #-----------------------------------------------------------------------
 # Constructor
 #-----------------------------------------------------------------------

 def __init__( self ):
  BigWorld.Entity.__init__( self )

  # random yaw
  yaw = random.uniform(-math.pi, math.pi)

  self.direction = (0.0, 0.0, yaw)

  if self.mode == ElPolloDiablo.PATROL_MODE:
   self.stopFollow()
  else:
   self.startFollow( self.targetID )

 def onTimer( self, controllerID, userData ):
  self.think()

 def startFollow( self, targetID ):
  self.mode = ElPolloDiablo.FOLLOW_MODE
  self.targetID = targetID
  self.cancel( "Movement" )
  self.think()

 def stopFollow( self ):
  self.mode = ElPolloDiablo.PATROL_MODE
  self.nextNode = None
  self.cancel( "Movement" )
  self.think()

 def think( self ):
```

```
 if self.mode == ElPolloDiablo.PATROL_MODE:
  self.patrol()
 else:
  self.follow()


# Patrol brain
def patrol( self ):
 # If we haven't got any nodes, find a pair
 if self.nextNode is None:
  self.setupNodes()
 if self.nextNode is None:
  # If we can't find a pair of nodes, wait 5 seconds and try again
  self.cancel( "Movement" )
  self.addTimer( 5 )
  return

 # If we've arrived, turn around
 if self.closeEnoughToNode():
  self.swapNodes()

 # Navigate towards self.nextNode.position
 dest = self.canNavigateTo( self.nextNode.position, 500, 0.5 )
 if dest is None:
  # No path found
  self.cancel( "Movement" )
  self.addTimer( 5 )
  return
 self.navigateStep( dest, ElPolloDiablo.VELOCITY, 500, 500, True, 0.5 )

def closeEnoughToNode( self ):
 target = self.nextNode
 return distance( self.position, target.position ) <=
ElPolloDiablo.PATROL_DISTANCE

def setupNodes( self ):
 self.prevNode = None
 self.nextNode = None
 closest = None
 dist = 500
 for i in BigWorld.userDataObjects.values():
  if i.__class__.__name__ != "PatrolNode" or len(i.patrolLinks) == 0:
   continue
  if distance( self.position, i.position ) < dist:
   closest = i
   dist = distance( self.position, i.position )
 if closest is not None:
  after = closest.patrolLinks[ 0 ]
  while distance( closest.position, after.position ) <
ElPolloDiablo.PATROL_DISTANCE * 3:
   after = after.patrolLinks[ 0 ]
   if after is None or after.uuid == closest.uuid:
    after = None
    break
  if after is not None:
   self.prevNode = closest
   self.nextNode = after

def swapNodes( self ):
 temp = self.nextNode
 self.nextNode = self.prevNode
 self.prevNode = temp
```

```
  def onMoveFailure( self, controllerID, userData ):
   # Can't get there. Turn around
   self.swapNodes()
   self.think()

  # Follow brain
  def follow( self ):
   # If self.targetID doesn't exist, switch to patrol mode
   if not BigWorld.entities.has_key( self.targetID ):
    self.stopFollow()
    return

   # If target isn't in this space, switch to patrol mode
   target = BigWorld.entities[ self.targetID ]
   if target.spaceID != self.spaceID:
    self.stopFollow()
    return

   # If we've arrived, wait here for target to move away
   if self.closeEnoughToTarget():
    self.cancel( "Movement" )
    self.addTimer( 5 )
    return

   # Follow our target
   yaw = target.yaw + ElPolloDiablo.FOLLOW_ANGLE
   offset = ( ElPolloDiablo.FOLLOW_DISTANCE * math.sin( yaw ), 0,
  ElPolloDiablo.FOLLOW_DISTANCE * math.cos( yaw ) )
   dest = self.canNavigateTo( target.position + offset, 500, 0.5 )
   if dest is None:
    # No path found
    self.cancel( "Movement" )
    self.addTimer( 5 )
    return
   self.navigateStep( dest, ElPolloDiablo.VELOCITY, 500, 500, True, 0.5 )

  def closeEnoughToTarget( self ):
   target = BigWorld.entities[ self.targetID ]
   return distance( self.position, target.position ) <=
  ElPolloDiablo.FOLLOW_DISTANCE

  def onMove( self, controllerID, userData ):
   self.think()

 # ElPolloDiablo.py
```