How To Build a Server-Controlled Moving Platform

BigWorld Technology 1.9.1. Released 2008.

Software designed and built in Australia by BigWorld.

Level 3, 431 Glebe Point Road
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Introduction

Most MMOGs have some form of moving vehicle/platform that transports player avatars and/or NPCs from one location to another. In this document, we will present a simple sample of a server-controlled moving platform using BigWorld Python script.

Chapter 2. Features

In this example, we will implement the following features:

- A server-controlled `MovingPlatform` moving along a path built in WorldEditor.
- A PlatformNode User Data Object to intuitively define the path of the platform in WorldEditor.
- Support for an arbitrary number of players to ride the platform.

Chapter 3. Example Code

The example code is part of the FantasyDemo example game.

- The example platform vehicle in FantasyDemo is the `MovingPlatform`. Its behaviour is to follow a path of User Data Objects of the type `PlatformNode`. The platform will wait at specific nodes and can randomly choose a direction when the path forks.
- The PlatformNode editor script demonstrates changing in editor appearance based on property data as well as making path building easy and intuitive.

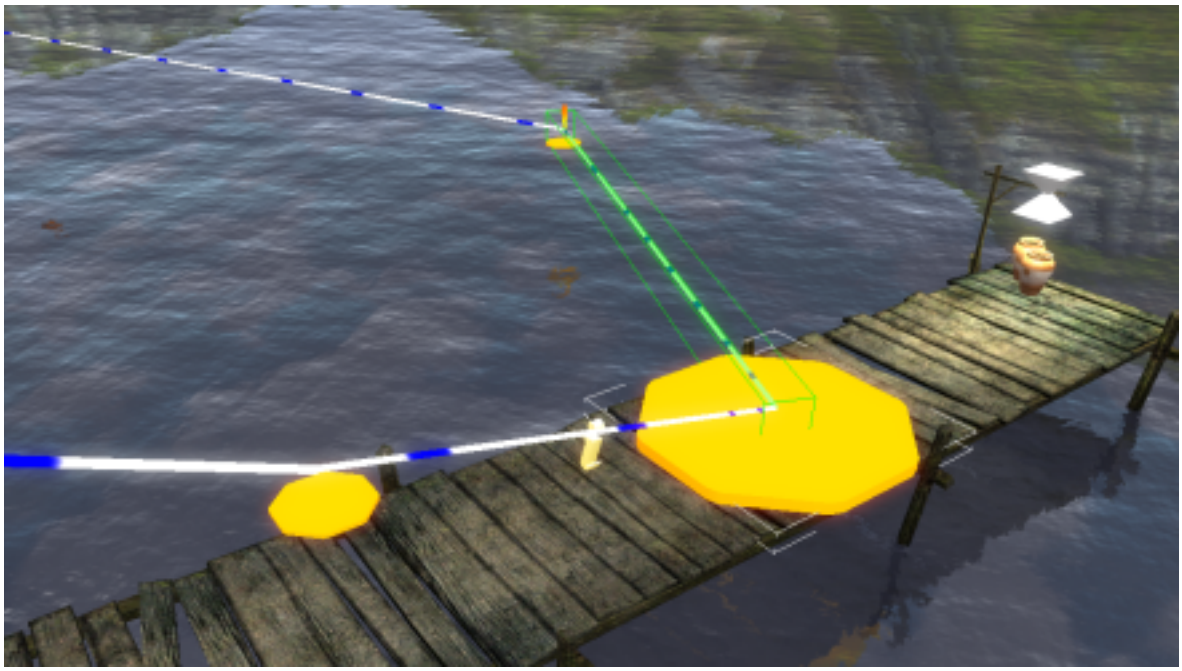
3.1. `MovingPlatform` entity

`MovingPlatform` is a Non-Player Character (NPC) entity that moves around the game scene along a specified path. The path is a connected graph of user data objects of type `PlatformNode` built in WorldEditor. `MovingPlatform` moves from one node to the next by applying and acceleration — this is achieved using the `Entity.acclerateToPoint` movement controller call on `MovingPlatform`'s cell entity. The next node to which the `MovingPlatform` will traverse is randomly selected from those connected in the direction the platform it travelling. The editor script for the Patrol nodes ensures that no backwards or dual direction links exist so the platform will never double back on its self.

3.1.1. The path

The `MovingPlatform` takes its path from a set of connected `PlatformNode` user data objects, which were implemented for building paths — for details see the document Content Tools Reference Guide's section WorldEditor → Useful notes → Patrol path editing with User Data Objects.

The path appears as a connected graph of nodes, with paths linking them together in a definite direction. From the path, the `MovingPlatform` can find traversable nodes to continue its movement. Each node also contains information such as how long the platform should wait at the node and how fast it should travel as it approaches the node. This node's data is also used to modify its appearance in `WorldEditor`. Below you will observe that one node is lager than the others. This is used to highlight nodes at which the `MovingPlatform` will stop.



Path of user data objects displayed in WorldEditor

3.1.2. Movement control

The example `MovingPlatform` that follows a world builder defined path though the air. To move the platform the function `Entity.accelerateToPoint()` is used. This movement controller provides simple movement with smooth acceleration but makes no effort to avoid obstacles. It is entirely the world builder's responsibility to ensure the platform does not fly through any walls.

3.1.3. Put `MovingPlatform` in client collision scene

Normally, we load models for various entities on the client using `BigWorld.Model` method. These loaded models, however are not added to the collision scene, *i.e.*, instead of colliding with the entity/model, your player avatar (or NPC) will pass through the colliding entity.

In the case of our `MovingPlatform`, the player avatar will fall through the vehicle, instead of standing on top of it. We need to use `BigWorld.PyModelObstacle` to load the moving platform's model, in order to put the model in the collision scene on the client.

For example, in `client/MovingPlatform.py` file's `enterWorld` method:

```
self.model = BigWorld.PyModelObstacle( MovingPlatform.PLATFORM_MODEL,
self.matrix, True )
```

3.1.4. Board and Alight from vehicles

Note

Any entity will become a vehicle when another entity boards it — there are no restrictions on which entity can become a vehicle.

Player-controlled avatar entity and server-controlled NPC entities use different mechanisms to board and alight from vehicles. A server-controlled NPC would need to explicitly use `Entity.boardVehicle` and `Entity.alightVehicle`, to board or alight from a vehicle.

Player-controlled avatar entities do not require explicit API calls to board or alight from a vehicle because the gravity calculations in the physics controller automatically detect if the player is standing on a vehicle. When player avatar moves away from the platform such that it is no longer underneath our player's feet, the entity physics controller will automatically alight them from the vehicle.

There are two requirements for this automatic boarding and alight mechanism to operate and they are as follows:

- The player avatar must have its gravity attribute set in its physics object, as illustrated below:

  ```
  BigWorld.player().physics.fall=1
  ```

- The would-be vehicle entity must have its model `vehicleID` attribute set to itself — in our `client/MovingPlatform.py`'s file `enterWorld` method:

  ```
  self.model.vehicleID = self.id
  ```

3.1.5. Server physics validation

Note

Server physics validation only works on client-controlled entities with `topSpeed` attribute greater than zero.

The server can perform limited physics validations on player-controlled avatar entities — apart from validate speed of player avatar, the server can also permit or disallow avatar getting on/off a vehicle. For details, see the document Server Programming Guide's section Proxies and Players → Physics correction.

The `onPassengerBoardAttempt` and `onPassengerAlightAttempt` optional callback on vehicle entity are invoked when a player avatar boards or alights the vehicle. Returning *True* permits avatar's vehicle transition movement — otherwise, the movement is disallowed, and avatar will be forced back to its earlier position in relation to the vehicle.

The current `MovingPlatform` implementation does not have any additional logic in the two callbacks, apart from a simple print statement.

Appendix A. Example

Table of Contents

A.1. `MovingPlatform.py on client`

```
import BigWorld

# This implements the MovingPlatform on the Client.
class MovingPlatform( BigWorld.Entity ):

    PLATFORM_MODEL = 'sets/items/platform.model'

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


    def prerequisites( self ):
        return [ MovingPlatform.PLATFORM_MODEL ]


    # This is called by BigWorld when the Entity enters AoI, through creation
 or movement.
    def onEnterWorld( self, prereqs ):
        self.model = BigWorld.PyModelObstacle( MovingPlatform.PLATFORM_MODEL,
 self.matrix, True )

        # Set appropriate filter for server controlled Entity
        self.filter = BigWorld.AvatarFilter()
        self.model.vehicleID = self.id

        BigWorld.addShadowEntity( self )


    # This is called by BigWorld when the Entity leaves AoI, through creation
 or movement.
    def onLeaveWorld( self ):
        BigWorld.delShadowEntity( self )
        self.model = None


# MovingPlatform.py
```

`MovingPlatform.py` on client

A.2. `MovingPlatform.py` on cell

```
import BigWorld
import Math
import random

# This implements the moving platform on the cell.
```

```
# The platform follows a path of PlatformNodes that is set in
# the editor, starting at the node named in startPatrolNode
# property.
class MovingPlatform( BigWorld.Entity ):

     WAIT_AT_NODE_TIMER = 1

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


     def onTimer( self, ctrlID, timerID ):
          if timerID == MovingPlatform.WAIT_AT_NODE_TIMER:
               try:
                    self.moveNext()
          except BigWorld.UnresolvedUDORefException:
                    self.addTimer( random.uniform( 5, 6 ), 0,
                                   MovingPlatform.WAIT_AT_NODE_TIMER )


     def moveNext( self ):
          if len( self.startNode.links ) == 0:
               return

          self.startNode = random.choice( self.startNode.links )

          self.accelerateToPoint( self.startNode.position,
                                  self.startNode.approachSpeed,
                                  self.startNode.approachAcceleration,
                                  0, # FACING_NONE
                                  self.startNode.waitTime > 0 )


     def onMove( self, ctrlID, waitTime ):
          try:
               if self.startNode.waitTime > 0:
                    self.addTimer( self.startNode.waitTime, 0,
                                   MovingPlatform.WAIT_AT_NODE_TIMER )
               else:
                    self.moveNext()

          except BigWorld.UnresolvedUDORefException:
               self.addTimer( random.uniform( 5, 6 ), 0,
                              MovingPlatform.WAIT_AT_NODE_TIMER )


     def onPassengerAlightAttempt( self, alightEntity ):
          print "entity", alightEntity.id, "departing"
          return True


     def onPassengerBoardAttempt( self, boardEntity ):
          print "entity", boardEntity.id, "boarding"
          return True


     # Called from cell/FantasyDemo.py when onAllSpaceGeometryLoaded() event
 is
     # received by the cellapp personality script.
     def onAllSpaceGeometryLoaded( self, spaceID, isBootstrap, mapping ):
          self.addTimer( 0, 0, MovingPlatform.WAIT_AT_NODE_TIMER )
```

```
# MovingPlatform.py
```

MovingPlatform.py on cell

A.3. MovingPlatform.py editor script

```
class MovingPlatform:
    def modelName( self, props ):
        return 'sets/items/platform.model'

    # platform nodes linking conditions
    def canLink( self, propName, thisInfo, otherInfo ):
        if otherInfo['type'] == 'PlatformNode':
            return True
        else:
            return False

# MovingPlatform.py
```

MovingPlatform.py editor script

A.4. PlatformNode.py editor script

```
# Importing WorldEditor to use linking functions
import WorldEditor
import re


# name of the patrol path link property
LINK_PROP_NAME = "links"


class PlatformNode:

    linkPropertyRE = re.compile( "^links$|^links\[\d+\]$" )

    # return the patrol node model
    def modelName( self, props ):
        try:
            if props['waitTime'] > 0:
                return "resources/models/floating_platform_stop.model"
            else:
                return "resources/models/floating_platform_node.model"
        except:
            return "helpers/props/standin.model"


    # patrol nodes always show the "+" gizmo
    def showAddGizmo( self, propName, thisInfo ):
        return True


    # patrol nodes linking conditions
    def canLink( self, propName, thisInfo, otherInfo ):
        if PlatformNode.linkPropertyRE.match( propName ):
            if otherInfo["type"] != thisInfo["type"]:
                return False    # PlatformNode to PlatformNode only
```

```
            elif ( thisInfo["guid"], thisInfo["chunk"] ) in
otherInfo["properties"]["links"]:
                return False    # no backlinks
            elif ( otherInfo["guid"], otherInfo["chunk"] ) in
thisInfo["properties"]["links"]:
                return False    # single links only
        else:
            return False


    # helper method to remove empty links
    def cleanupEmptyLinks( self, links ):
        emptyLink = ("","")
        while links.__contains__( emptyLink ):
            links.remove( emptyLink )


    # when a node is deleted, other nodes must be relinked
    def onDeleteObject( self, nodeInfo ):
        links = nodeInfo["properties"][ LINK_PROP_NAME ]
        backLinks = nodeInfo["backLinks"]
        self.cleanupEmptyLinks( links )
        self.cleanupEmptyLinks( backLinks )

        relinkNode = None
        linksBegin = 0
        backLinksBegin = 0
        if len( links ) > 0:
            # relink everything to the first outgoing node
            relinkNode = links[0]
            linksBegin = 1
        elif len( backLinks ) > 0:
            # relink everything to the first incoming node
            relinkNode = backLinks[0]
            backLinksBegin = 1
        else:
            return    # nothing to do!

        # Relinking outgoing links
        for link in links[linksBegin:]:
            WorldEditor.udoCreateLink( relinkNode[0], relinkNode[1], link[0],
link[1], LINK_PROP_NAME )

        # Relinking incoming links (BackLinks)
        for backlink in backLinks[backLinksBegin:]:
            WorldEditor.udoCreateLink( backlink[0], backlink[1],
relinkNode[0], relinkNode[1], LINK_PROP_NAME )

        WorldEditor.addUndoBarrier( "Relink after deleting PatrolNode" )


    # called to get extra context menu commands from the script
    def onStartLinkMenu( self, startInfo, endInfo ):
        # we should localise these strings
        return ("Split link",
                "",
                "Swap Link Direction",
                "Swap Link Directions (Run Of Links)" )


    # called when one of the extra context menu commands is clicked
    def onEndLinkMenu( self, command, startInfo, endInfo ):
```

```
        if command == 0:
            self.splitLink( startInfo, endInfo )
        elif command == 1:
            self.swapLink( startInfo, endInfo )
        elif command == 2:
            self.swapLinksRun( startInfo, endInfo )
        else:
            WorldEditor.addCommentaryMsg( "PatrolNode: received unknown
command " + str(command) + ".", 0 )


    # split a link, creating a new node in the middle
    def splitLink( self, startInfo, endInfo ):
        startProps = startInfo["properties"]
        endProps = endInfo["properties"]

        # find out whether the nodes are linked in one or both directions
        startToEndLink = False
        endToStartLink = False
        if ( endInfo["guid"], endInfo["chunk"] ) in
startProps[LINK_PROP_NAME]:
            startToEndLink = True
        if ( startInfo["guid"], startInfo["chunk"] ) in
endProps[LINK_PROP_NAME]:
            endToStartLink = True

        # create new node
        midPoint = ( (startInfo["position"][0] + endInfo["position"][0])/2.0,
                     (startInfo["position"][1] + endInfo["position"][1])/2.0,
                     (startInfo["position"][2] + endInfo["position"][2])/2.0 )
        newInfo = WorldEditor.udoCreateAtPosition( startInfo["guid"],
startInfo["chunk"], midPoint, False )
        newProps = newInfo["properties"]

        # delete old links
        WorldEditor.udoDeleteLinks( startInfo["guid"], startInfo["chunk"],
endInfo["guid"], endInfo["chunk"] )

        # relink
        if startToEndLink:
            WorldEditor.udoCreateLink( startInfo["guid"], startInfo["chunk"],
newInfo["guid"], newInfo["chunk"], LINK_PROP_NAME )
            WorldEditor.udoCreateLink( newInfo["guid"], newInfo["chunk"],
endInfo["guid"], endInfo["chunk"], LINK_PROP_NAME )

        if endToStartLink:
            WorldEditor.udoCreateLink( endInfo["guid"], endInfo["chunk"],
newInfo["guid"], newInfo["chunk"], LINK_PROP_NAME )
            WorldEditor.udoCreateLink( newInfo["guid"], newInfo["chunk"],
startInfo["guid"], startInfo["chunk"], LINK_PROP_NAME )

        WorldEditor.addUndoBarrier( "Split link" )


    # swap the direction of the link
    def swapLink( self, startInfo, endInfo ):
        startProps = startInfo["properties"]

        # find out whether the nodes are linked in one or both directions
        startToEndLink = False
        if startProps[LINK_PROP_NAME].__contains__( ( endInfo["guid"],
endInfo["chunk"] ) ):
```

```
            startToEndLink = True

        WorldEditor.udoDeleteLinks( startInfo["guid"], startInfo["chunk"],
endInfo["guid"], endInfo["chunk"] )

        if startToEndLink:
            WorldEditor.udoCreateLink( endInfo["guid"], endInfo["chunk"],
startInfo["guid"], startInfo["chunk"], LINK_PROP_NAME )
        else:
            WorldEditor.udoCreateLink( startInfo["guid"], startInfo["chunk"],
endInfo["guid"], endInfo["chunk"], LINK_PROP_NAME )

        WorldEditor.addUndoBarrier( "Swap link direction" )


    # swap the direction of the link
    def swapLinksRun( self, startInfo, endInfo ):
        newDir = "START_END"
        if ( endInfo["guid"], endInfo["chunk"] ) in
startInfo["properties"][LINK_PROP_NAME]:
            newDir = "END_START"

        self.changeDir( startInfo, endInfo, newDir )

        WorldEditor.addUndoBarrier( "Swap direction of links (run of links)" )


    # internal method to change the direction of all traversable links
    def changeDir( self, startInfo, endInfo, newDir ):
        # follow links and store them in resultLinks
        resultLinks = []
        for dir in [ "forward", "backward" ]:
            # this loop has two iterations: one for following links from start
            # to end, and one for following links from end to start.
            lastInfo = startInfo
            curInfo = endInfo
            skipFirstOne = False
            if dir == "backward":
                lastInfo = endInfo
                curInfo = startInfo
                skipFirstOne = True # skips the first link, processed in
"forward"

            while curInfo != None:
                # Loop for following links.
                if skipFirstOne:
                    skipFirstOne = False # skip this one
                else:
                    # add the link to the result, or break if we hit a link
that
                    # is in the results already to avoid infinite loops.
                    startEnd = ( ( lastInfo["guid"], lastInfo["chunk"] ), (
curInfo["guid"], curInfo["chunk"] ) )
                    endStart = ( ( curInfo["guid"], curInfo["chunk"] ), (
lastInfo["guid"], lastInfo["chunk"] ) )
                    if resultLinks.__contains__( startEnd ) or
resultLinks.__contains__( endStart ):
                        break

                    if dir == "forward":
                        resultLinks.append( startEnd );
                    else:
```

```
                        resultLinks.append( endStart );

                # find the next node
                links = curInfo["properties"][LINK_PROP_NAME]
                self.cleanupEmptyLinks( links )
                numLinks = len(links)
                if links.__contains__( ( lastInfo["guid"], lastInfo["chunk"] )
):
                    numLinks -= 1

                backLinks = curInfo["backLinks"]
                self.cleanupEmptyLinks( backLinks )
                numBackLinks = len(backLinks)
                if backLinks.__contains__( ( lastInfo["guid"],
lastInfo["chunk"] ) ):
                    numBackLinks -= 1

                if numLinks == 1:
                    # there's a outgoing link, so use it.
                    nextNode = links[0];
                    if nextNode == ( lastInfo["guid"], lastInfo["chunk"] ):
                        # The first link is to the lastInfo, so there must be
                        # two links, and our next node must be in the second.
                        nextNode = links[1];

                    lastInfo = curInfo
                    curInfo = WorldEditor.udoGet( nextNode[0], nextNode[1] );

                elif numBackLinks == 1:
                    # no outgoing links, but there's a backlink, so use it.
                    nextNode = backLinks[0];
                    if nextNode == ( lastInfo["guid"], lastInfo["chunk"] ):
                        # The first link is to the lastInfo, so there must be
                        # two links, and our next node must be in the second.
                        nextNode = backLinks[1];

                    lastInfo = curInfo
                    curInfo = WorldEditor.udoGet( nextNode[0], nextNode[1] );

                else:
                    # no suitable links so break.
                    break

        # do the actual operation on the found links
        for link in resultLinks:
            id1 = link[0][0]
            chunk1 = link[0][1]
            id2 = link[1][0]
            chunk2 = link[1][1]

            WorldEditor.udoDeleteLinks( id1, chunk1, id2, chunk2 )

            if newDir == "START_END" or newDir == "BOTH":
                WorldEditor.udoCreateLink( id1, chunk1, id2, chunk2,
LINK_PROP_NAME )

            if newDir == "END_START" or newDir == "BOTH":
                WorldEditor.udoCreateLink( id2, chunk2, id1, chunk1,
LINK_PROP_NAME )
```

PlatformNode.py editor script