Tutorial

BigWorld Technology 2.0. Released 2010.

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

1. Overview 5
    1.1. Conventions 5
        1.1.1. Files and directories 5
        1.1.2. Linux development environment 5
    1.2. Provided files 5
    1.3. Debugging 6
2. A Basic Client-Only Game (`CLIENT_ONLY`) 7
    2.1. Creating a new project 7
    2.2. Defining resource paths 7
    2.3. Creating the resources directory 8
    2.4. Creating our first entity 8
        2.4.1. `entities.xml` 8
        2.4.2. Defining the Avatar entity type 8
        2.4.3. Implementing the `Avatar` entity type 9
    2.5. The personality script 11
    2.6. XML configuration files 13
    2.7. A simple space 14
    2.8. Running the client for the first time 16
3. A basic client-server game (`CLIENT_SERVER`) 19
    3.1. Server Installation and Configuration 19
    3.2. A `Space` entity. 19
        3.2.1. entities.xml 19
        3.2.2. Entity definition 19
        3.2.3. Base part 20
        3.2.4. Cell part 20
    3.3. Server-side personality scripts 21
    3.4. The server-side `Avatar` scripts 21
    3.5. Connecting the client to the server 22
    3.6. Going 3rd person 24
    3.7. Server-side XML configuration 24
    3.8. Starting and connecting to the server 25
        3.8.1. Indie Edition 25
        3.8.2. Commercial/Indie Source Edition 26
        3.8.3. Starting a Server 26
4. Implementing a chat system (`CHAT_CONSOLE`) 27
    4.1. GUI text console 27
    4.2. Modifications to the `Avatar` entity 27
5. EntityLoader (`ENTITY_LOADER`) 29
    5.1. Implementation 29
6. A Basic NPC Entity (`BASIC_NPC`) 31
    6.1. Design 31
    6.2. Art 31
        6.2.1. Exporting the model 31
        6.2.2. Configuring the model 32
    6.3. Scripts 32
        6.3.1. entities.xml 33
        6.3.2. Entity definition 33
        6.3.3. Base part 34
        6.3.4. Cell part 35
        6.3.5. Client part 36
        6.3.6. Editor script 39
    6.4. Testing 39
    6.5. Possible improvements 40

Chapter 1. Overview

This tutorial provides a brief overview of the minimum steps needed to get a basic game working from scratch. Game developers and technical artists working with BigWorld for the first time should work through this tutorial to get a feel for the way the various files and directories fit together to produce a working game.

The game demo that ships with the BigWorld package is called FantasyDemo. If you are reading this tutorial, you have probably already spent some time playing through it and seeing some of the things that the BigWorld engine can do. Unfortunately for new developers, FantasyDemo is actually a rather large and involved project, so using it as a reference point for implementing a new game can be quite confusing. In general, it is not obvious what can and cannot be stripped out to create a skeleton game.

Instead, this document will work from an empty directory and build the project file by file, to give you a clear understanding of what each file and directory is for.

Note

For details on BigWorld terminology, see the document Glossary of Terms.

1.1. Conventions

1.1.1. Files and directories

This document uses Unix filesystem conventions for file naming *i.e.*, files will be named `<res>/scripts/db.xml`, and not `<res>\scripts\db.xml`. You should follow this practice when developing your game, whether or not you are dealing with client-side or server-side scripts and/or assets.

This tutorial assumes you are working on a Windows box, with the files mounted on a local filesystem. The early stages of the tutorial are entirely client-side, so any issues regarding the synchronisation of files between the client and server are not addressed here. Cross machine synchronisation is discussed in “Server Installation and Configuration” on page 19 .

Note

This tutorial assumes that the BigWorld package was extracted to the directory `C:\BigWorld`.

1.1.2. Linux development environment

This tutorial assumes that you are using a UNIX user account called `Fred`. The parts of this tutorial that involve resources mounted on a Linux filesystem assume that they are mounted at `$HOME/mf` (*i.e.*, `/home/fred/mf`).

1.2. Provided files

All files used in this tutorial are provided in the `tutorial` directory of your BigWorld package.

As shipped, the files represent the final state of the completed tutorial. If you are new to BigWorld development, then you probably want to see the minimal set of files required at each stage of the tutorial, instead of just diving into the completed tutorial (which while much simpler than FantasyDemo, still consists of a fair number of files). To help you with this, BigWorld provides a utility (`tutorial/generate_res_trees.py`) that strips down the resource tree to the minimal state needed for a particular stage of the tutorial. If you run the utility with the symbolic name of a chapter (*e.g.*, `./tutorial.py`

CLIENT_SERVER), then the stripped resources are extracted to an appropriately named <*res*> directory in the tutorial directory (*e.g.*, `tutorial/res_client_server`). You can then alter the `paths.xml`[1] and `.bwmachined.conf`[2] settings to point to these stripped trees.

Even if you are doing the final stage of the tutorial, it may be helpful to run this stripping utility before looking through the source code, as it removes all inclusion/exclusion steps that we have inserted to facilitate the stripping process and makes the code easier to read.

Note

The symbolic constants for each chapter are given in the chapter heading *e.g.*, `CLIENT_ONLY`.

1.3. Debugging

There may be times while working through the tutorial that the client won't start due to some error in the scripts. In order to discover the cause of the error, use a program such as DebugView (available on the Microsoft website) which captures and displays debug output.

[1]For details on how to configure `paths.xml`, see "Defining resource paths" on page 7 , and "A simple space" on page 14 .

[2]For details on how to configure `.bwmachined.conf`, see "Starting and connecting to the server" on page 25 .

Chapter 2. A Basic Client-Only Game (`CLIENT_ONLY`)

This chapter describes how to get a bare-bones client up and running with its own resources and scripts. This involves:

- Creating a new BigWorld project directory.
- Creating files and directories necessary to define a single client-side player entity.
- Creating a new space.

By the end of this part of the tutorial, it will be possible to walk around a trivial space in the client using a first-person view.

2.1. Creating a new project

The FantasyDemo project is located in the `fantasydemo` directory in `C:\BigWorld`. Following that convention, we will start our new tutorial project in the same directory, by creating a new directory called `tutorial` in `C:\BigWorld`. All resources and scripts specific to this project will be located within this directory.

Please note that the tutorial project is shipped as part of your package. A skeleton project called `my_game` is also shipped as part of the Indie edition, in order to allow you to start a new project easily. Please review the Getting Started document for more details.

2.2. Defining resource paths

The BigWorld client is a generic executable, located at `bigworld\bin\client\bwclient.exe`. Since it is independent of the game resources it loads, it needs to be instructed as to where to find your project's resources.

The easiest way to go about this is to use the `--res` command line switch in conjunction with a batch file to provide a convenient way to start the client for your particular game. A benefit of doing it this way is that it also keeps the resource path configuration self contained within your project folder. Typically, you would create a batch file named `run.bat` and it would be located at the root level of your project folder (i.e. in `my_game`) and would look something like:

```
"..\bigworld\bin\client\bwclient.exe" --res %~dp0res;../../../bigworld/res
```

Keep in mind that paths are relative to the executable location, *not* the current working directory. The above example uses `%~dp0` to grab the batch file's directory as an absolute path in order to keep the batch file generic.

Note

Remember, the %~dp0 trick will only work in a .BAT file. If you want to launch the from the command prompt directly, you will need to specifiy the full path explicitly.

Note

For details on how the client searches for resources, see the “Resource search paths” section in the Client Programming Guide.

2.3. Creating the resources directory

Resource directories for BigWorld games are typically named `res`, therefore you can simply create a directory called `res` in the `tutorial` directory. This top-level resources directory will contain all game-specific scripts, assets, and configuration files.

2.4. Creating our first entity

Entities are game objects that have a position. Not every class that you write in your game must be an entity, but most objects that are part of the game mechanics will be. Examples of entities would be the player, NPCs, chat rooms, dropped items, etc.... Examples of objects that need not be entities might be helper classes that are only attached to/used by a single entity type.

Note

For details on this and other BigWorld server terms, see the document Glossary of Terms.

2.4.1. `entities.xml`

Entity scripts for a BigWorld game must reside in a `res/scripts` directory. One of the files that must exist in this directory is `entities.xml`[1], which lists the game entities that will be used.

Create a basic `tutorial/res/scripts/entities.xml` file that contains a player entity called `Avatar`:

```
<root>
   <Avatar/>
</root>
```

Example `tutorial/res/scripts/entities.xml`

2.4.2. Defining the Avatar entity type

The other directory that must exist is `res/scripts/entity_defs`, which contains the `.def`[2] files, with definitions of the properties and methods for each entity.

It might be helpful to think of these definition files as being similar to C/C++ header files as they specify the types of properties and the method calls attached to the entity.

Create the `tutorial/res/scripts/entity_defs/Avatar.def` file, with the following contents:

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

[1] For details on this file, see the Server Programming Guide's section *Physical Entity Structure for Scripting* in "The entities.xml File".

[2] For details on these files, see the Server Programming Guide's section *Physical Entity Structure for Scripting*, in "The Entity Definition File".

```
        <Type>   UNICODE_STRING       </Type>
        <Flags>  ALL_CLIENTS  </Flags>
      </playerName>
   </Properties>
   <ClientMethods>
   </ClientMethods>
   <CellMethods>
   </CellMethods>
   <BaseMethods>
   </BaseMethods>
</root>
```

Example `tutorial/res/scripts/entity_defs/Avatar.def`

This is a very basic entity definition which defines properties for the entity, but no methods. Notice that the properties are separated into two sections: *volatile* and *non-volatile*.

2.4.2.1. Volatile properties

For a BigWorld entity, volatile properties are positional/directional properties. They are described as *volatile* because they are constantly changing. The volatile properties' current value are only considered to be important thing while the history of changes on the property is less important. In a bandwidth-constrained environment only the current value should be sent.

The supported volatile properties are `position`, `yaw`, `pitch`, and `roll`. For simplicity, the `tutorial/res/scripts/entity_defs/Avatar.def` that we have just defined only sends `position` and `yaw` of the `Avatar` entity.

For details on volatile properties, see the Server Programming Guide's section *Properties*, in *Properties*.

2.4.2.2. Non-volatile properties

In contrast to volatile properties, regular properties tend to change infrequently, and therefore all changes to a particular property should be sent down to the client. Each property can be named as you wish, and can have a number of different settings attached to it.

We have defined a simple property for storing the player's name, and for simplicity, we are only using the most necessary property settings, specifying the type `STRING` and distribution flags `ALL_CLIENTS`. The `ALL_CLIENTS` tags means that this property will be visible to the player controlling the client entity, as well as any other player that can see his entity. For details on this and other distribution flags, see the Server Programming Guide's section *Properties*, in *Properties*.

For details on entity properties, see the Server Programming Guide's section *Properties*.

2.4.3. Implementing the `Avatar` entity type

The scripts that control the client-side entity logic are located in the `res/scripts/client`, and the ones that control the server-side entity logic are located in `res/scripts/cell` and `res/scripts/base` directories.

Create each of these directories within the `tutorial/res/scripts` directory. Your directory structure should now look like this:

```
tutorial
+-res
  +-scripts
```

```
    +-base
    +-cell
    +-client
    +-entity_defs
```

Folder structure at this stage of the tutorial

For details on the exact structure and mechanics of the `scripts` directory, see the Server Programming Guide's section *Physical Entity Structure for Scripting*.

Up to this point, we have declared the `Avatar` entity in `tutorial/res/scripts/entities.xml`[3] and defined it in `tutorial/scripts/entity_defs/Avatar.def`[4]Now we must provide (at least part of) the script implementation of that entity. Since we are working only on the client-side at the moment, just create the `tutorial/res/scripts/client/Avatar.py` script:

```
import BigWorld

# These are constants for identifying keypresses, mouse movement etc
import Keys

class Avatar( BigWorld.Entity ):

   def onEnterWorld( self, prereqs ):
      pass

class PlayerAvatar( Avatar ):

   def onEnterWorld( self, prereqs ):

      Avatar.onEnterWorld( self, prereqs )

      # Set the position/movement filter to correspond to an player avatar
      self.filter = BigWorld.PlayerAvatarFilter()

      # Setup the physics for the Avatar
      self.physics = BigWorld.STANDARD_PHYSICS
      self.physics.velocityMouse = "Direction"
      self.physics.collide = True
      self.physics.fall = True

   def handleKeyEvent( self, event ):

      # Get the current velocity
      v = self.physics.velocity

      # Update the velocity depending on the key input
      if event.key == Keys.KEY_W:
         v.z = event.isKeyDown() * 5.0
      elif event.key == Keys.KEY_S:
         v.z = event.isKeyDown() * -5.0
      elif event.key == Keys.KEY_A:
         v.x = event.isKeyDown() * -5.0
      elif event.key == Keys.KEY_D:
         v.x = event.isKeyDown() * 5.0

      # Save back the new velocity
```

[3] See "`entities.xml`" on page 8 .

[4] See "Defining the Avatar entity type" on page 8 .

```
        self.physics.velocity = v
```

Example tutorial/res/scripts/client/Avatar.py

Notice that the script declares two classes: `Avatar` and `PlayerAvatar`. These two classes are required to satisfy a hard-coded requirement in the BigWorld client that any entity type that can act as a client proxy must have a sub-class called `Player<class>` that is used when attaching to the client.

We are only interested in the player at the moment, so the implementation of the base `Avatar` class is left blank. For the moment, we have just provided implementations of callbacks for initialisation (where we set up the position filter and player physics) and keyboard events (where we provide basic `WASD` controls).

Notice that the `Avatar` script imports a module called `Keys`. This module defines constants for things like keyboard character codes, mouse events, joystick events, and other commonly used constants. It is located in `bigworld/res/scripts/client`, so we do not need to copy it or do anything special to access it from our scripts.

2.5. The personality script

The next required script for our basic client is the *personality* script. The easiest way to think of this script is as the bootstrap script for each component of a BigWorld system.

Note

For details on this and other BigWorld client terms, see the Glossary of Terms.

There should be one personality script in each script directory (*i.e.*, for cell, base, and client) and they are used for defining callbacks to be called on startup and shutdown, as well as other global, non-entity-related functionality. On the client, this might include menu systems, user input management, camera control, etc...

For details on the client personality script, see the Client Programming Guide's section *Scripting*, in "Personality script".

Save the basic personality script below as `tutorial/res/scripts/client/BWPersonality.py`:

```
# This is the client personality script for the BigWorld tutorial.  Think of
# it as the bootstrap script for the client.  It contains functions that
# are called on initialisation, shutdown, and handlers for various input
# events.
import BigWorld

# ----------------------------------------------------------------------------
# Section: Required callbacks
# ----------------------------------------------------------------------------
# The init function is called as part of the BigWorld initialisation process.
# It receives the BigWorld xml config files as arguments.  This is the best
# place to configure all the application-specific BigWorld components, like
# initial camera view, etc...
def init( scriptConfig, engineConfig, prefs ):

   initOffline( scriptConfig )

   # Hide the mouse cursor and restrict it to the client area of the window.
   GUI.mcursor().clipped = True
   GUI.mcursor().visible = False
```

```
# This is called immediately after init() finishes.  We're done with all our
# init code, so this is a no-op.
def start():
   pass

# This method is called just before the game shuts down.
def fini():
   pass

# This is called by BigWorld when player moves from an inside to an outside
# environment, or vice versa.  It should be used to adapt any personality
# related data (eg, camera position/nature, etc).
def onChangeEnvironments( inside ):
   pass

# This is called by the engine when a system generated message occurs.
def addChatMsg( msg ):
   print "addChatMsg:", msg

# Keyboard event handler
def handleKeyEvent( event ):
   return False

# Mouse event handler
def handleMouseEvent( event ):
    return False

# Joystick event handler
def handleAxisEvent( event ):
    return False

# ----------------------------------------------------------------------------
# Section: Helper methods
# ----------------------------------------------------------------------------
def initOffline( scriptConfig ):

   # Create a space for the client to inhabit
   spaceID = BigWorld.createSpace()

   # Load the space that is named in script_config.xml
   BigWorld.addSpaceGeometryMapping(
      spaceID, None, scriptConfig.readString( "space" ) )

   # Create the player entity, using positions from script_config.xml
   playerID = BigWorld.createEntity( scriptConfig.readString(
 "player/entityType" ),
                                     spaceID, 0,
                                     scriptConfig.readVector3(
 "player/startPosition" ),
                                     scriptConfig.readVector3(
 "player/startDirection" ),
                                     {} )

   BigWorld.player( BigWorld.entities[ playerID ] )

   # Use first person mode since we are not using models yet.
   BigWorld.camera().firstPerson = True
```

Example `tutorial/res/scripts/client/BWPersonality.py`

This personality script provides an `initOffline` method that contains enough code to get a basic client going, as well as stub implementations of all other required callbacks. The initialisation code expects various

configuration files to be passed to it, and expects `scriptConfig` to contain particular settings, such as `space`, `player/entityType`, and so on.

The following sections describe how to set up those files, so they will be ready to be passed to the personality script on startup.

2.6. XML configuration files

At a minimum, the BigWorld client expects three XML configuration files to be passed into the personality script at startup:

- `<engine_config>.xml`
- `<scripts_config>.xml`
- `<preferences>.xml`

> **Note**
>
> For details on these files, see the Client Programming Guide's section *Scripting*, in "Personality script", sub-sections "File `<engine_config>.xml`", "File `<scripts_config>.xml`", and "File `<preferences>.xml`" respectively.

The `<engine_config>.xml` file is used for setting various configurable properties on the client engine, including the name of the game's personality. We will re-use the engine settings used for FantasyDemo by copying `fantasydemo/res/engine_config.xml` to `tutorial/res/engine_config.xml`, ensuring that we change the `<personality>` setting to `BWPersonality`. Notice that this corresponds to the file `BWPersonality.py` that we created in "The personality script" on page 11).

The `<scripts_config>.xml` file is used to define the settings that the personality script is expecting — save the following into `tutorial/res/scripts_config.xml`:

```
<scripts_config.xml>
   <!-- The contents of this file are passed to the personality script
        as the first argument in the init function (as a data section). Its
        grammar is solely defined by the personality script. -->
   <space> spaces/main </space>
   <player>
      <entityType> Avatar </entityType>
      <!-- This is the entity type of the player that will be created. You
 must implement
           a Player<class> type (e.g. PlayerAvatar) to use this type as a
 client proxy. The following options -->
      <startPosition>  0.0 1.25 0.0 </startPosition>
      <startDirection> 1.0 0.0  0.0 </startDirection>
      <!-- are used by the personality script to provide a start position and
           facing dir for players if there is no space specific spawn point.
 -->
    </player>
</scripts_config.xml>
```

Example `tutorial/res/scripts_config.xml`

At this stage, the values for the configuration settings expected by the personality script's `init` method have been provided. The only thing still missing for our basic client is the actual space data. The script

configuration passes the string `spaces/main` into the personality script as the space in which the client entity will be created, so next we will create a basic space to walk around in.

2.7. A simple space

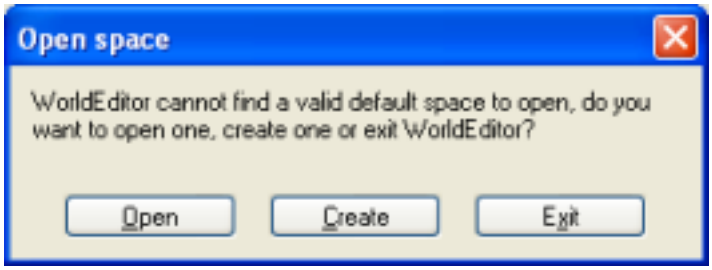
Before starting World Editor, you will need to tell it where to find the resources for your particular project. To do this, open `bigworld/tools/worldeditor/paths.xml` and replace the reference to FantasyDemo to your own project. For example,

```
<root>
    <Paths>
        <Path>../../../tutorial/res</Path>
        <Path>../../../bigworld/res</Path>
    </Paths>
</root>
```

Example `bigworld/worldeditor/paths.xml`

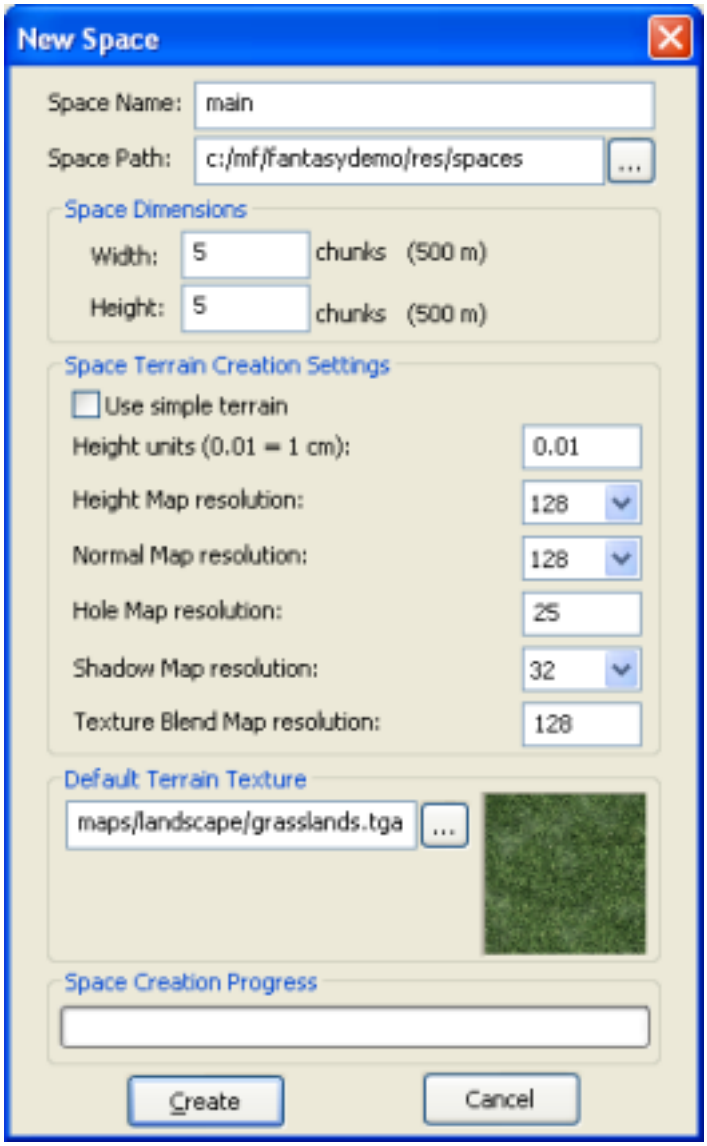
To create a simple space that can be navigated, follow the steps below:

- Start WorldEditor (`bigworld/tools/worldeditor/worldeditor.exe`).
- In the **Open Space** dialog box , click the **Create** button.



Open Space dialog box

- In the **New Space** dialog box:
  - Set the **Space Name** field to **main**.
  - Set the **Space Dimensions** group box's **Width** and **Height** fields to **5**.
  - Set the **Default Terrain Texture** field to a texture of your choosing.
  - Click the **Create** button.

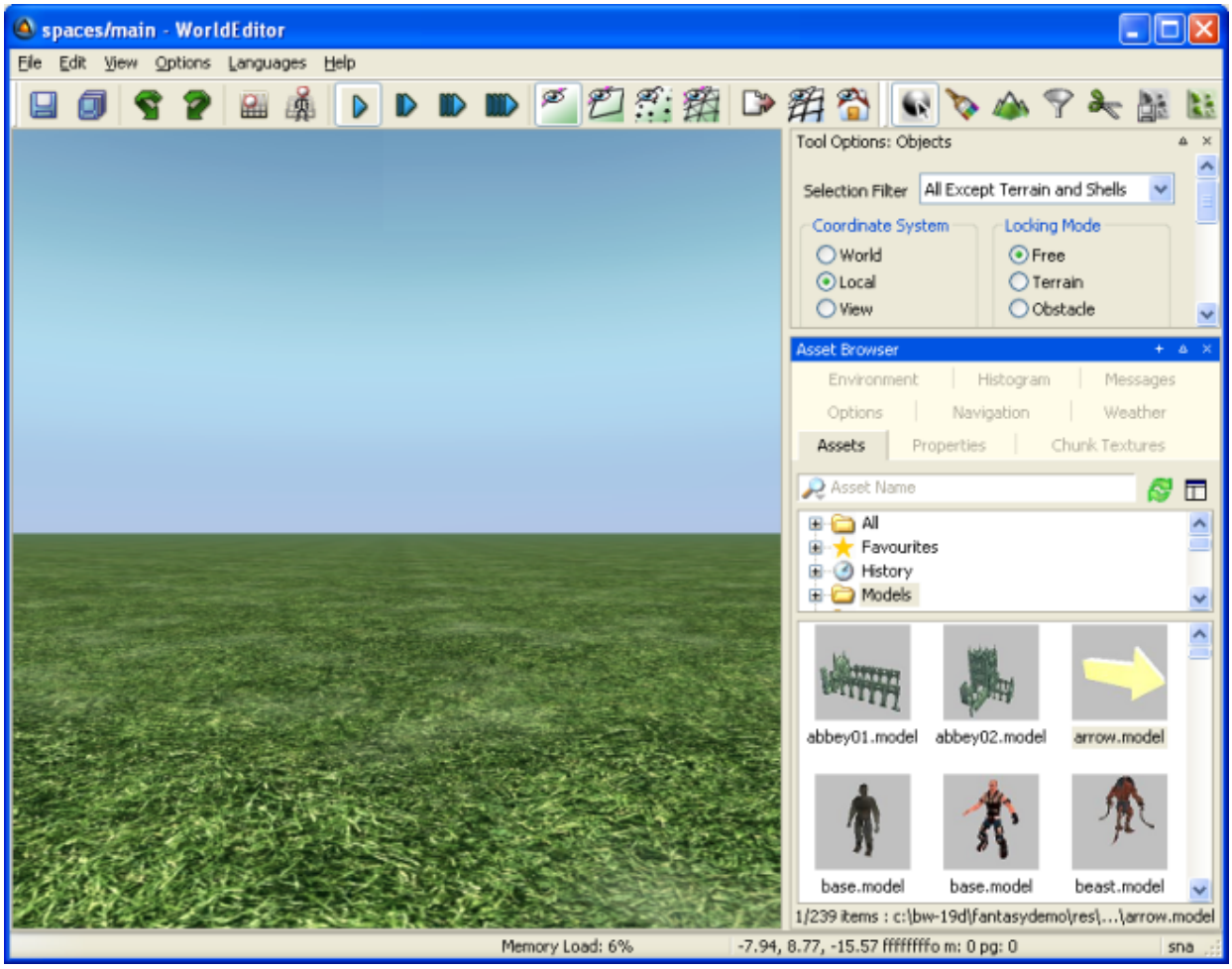


New Space dialog box

Note

For details on this dialog box, see the Content Tools Reference Guide's section "Dialog boxes", in "New Space dialog box".

- The new space `main` will be created and displayed in WorldEditor, as displayed below.

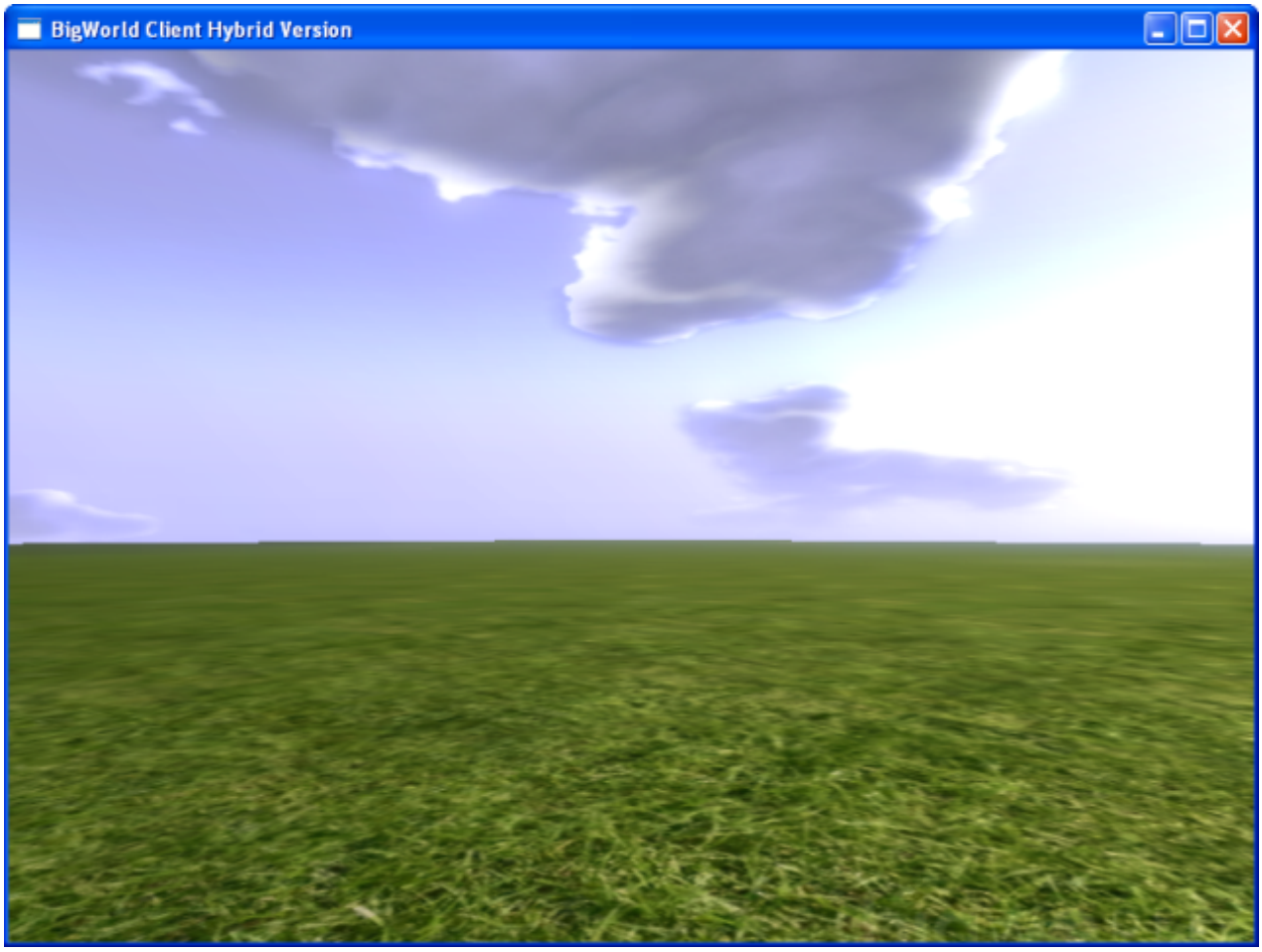


The `main` space

- Select the **File → Save** menu item to save the new space
- Select the **File → Exit** menu item to close WorldEditor.

2.8. Running the client for the first time

Having carried out the steps in the previous sections of this tutorial, you can now run the client . To do that, use the `run.bat` you created earlier. You should have a basic first-person player that can walk around a space using mouse-look and `WASD` controls.



A simple first-person client

Chapter 3. A basic client-server game (CLIENT_SERVER)

In *A Basic Client-Only Game (CLIENT_ONLY)* on page 7 we set up a basic client resources tree that would allow us to walk around a simple space using a first-person view. In this chapter of the tutorial we will extend the game to the server, so that multiple clients can log in and see each other walking around.

3.1. Server Installation and Configuration

Prior to progressing through this part of the tutorial it is necessary to install and configure the BigWorld server. If you haven't already done this please proceed to the Server Installation Guide

At this point it is also relevant to address the issue of sharing files between Linux and Windows machines. Since there are many files that are read by both the client and the server (`tutorial/res/scripts/entity_defs/*`, space data, etc), it is necessary to keep them all on a single file system that is shared between the client and server, rather than having to keep them synchronised manually. Please refer to the Client Programming Guide's section on *Shared Development Environments* for more information on this topic.

For the purposes of this tutorial, we will assume that you have mounted your Windows directory tree at `$HOME/bigworld_windows_share` on your Linux file system.

3.2. A Space entity.

In BigWorld, spaces are separate coordinate systems. Each space can have one or more geometry mappings (as created in the World Editor). Cell entities are associated with a single space at any one time. These may be used to implement things like planets, mission instances, apartments or game sharding.

A new space is created by creating a cell entity in a new space. It is typical to have an entity type that is responsible for space creation.

3.2.1. entities.xml

Every entity must be defined in the `entities.xml` file located at `tutorial/res` path.

```
<root>
    <Space/>
    <Avatar/>
</root>
```

3.2.2. Entity definition

The `Space` entity type has a single string property `spaceDir`. This will be used to indicate which space geometry to load.

```
<root>
   <Properties>
      <spaceDir>
         <Type>          STRING          </Type>
         <Flags>         BASE            </Flags>
      </spaceDir>
   </Properties>

   <ClientMethods>
   </ClientMethods>

   <CellMethods>
      <addGeometryMapping>
```

```
        <Arg>           STRING          </Arg>
      </addGeometryMapping>
   </CellMethods>

   <BaseMethods>
   </BaseMethods>
<root>
```

Example tutorial/res/scripts/entity_defs/Space.def

3.2.3. Base part

The base entity calls `self.createInNewSpace()` to create a new space, put its cell entity in it, and tells the cell to add a space geometry mapping. It registers itself globally as `"DefaultSpace"` so that the base entity can easily be found later.

```
import BigWorld

class Space( BigWorld.Base ):

    def __init__( self ):

        BigWorld.Base.__init__( self )

        # Create this entity in a new space
        self.createInNewSpace()
        self.cell.addGeometryMapping( self.spaceDir )

        self.registerGlobally( "DefaultSpace", self.onRegistered )

    def onRegistered( self, succeeded ):
        if not succeeded:
            print "Failed to register space."
            self.destroyCellEntity()

    def onLoseCell( self ):

        # Once our cell entity is destroyed, it's safe to clean up the Proxy.
        # We can't just call self.destroy() in onClientDeath() above, as
        # destroyCellEntity() is asynchronous and the cell entity would still
        # exist at that point.
        self.destroy()
```

Example tutorial/res/scripts/base/Space.py

3.2.4. Cell part

The cell entity maps the geometry to load after receiving a call from the base to `addGeometryMapping()`, with an appropriate path to a geometry (e.g. `spaces/main`).

```
import BigWorld

class Space( BigWorld.Entity ):

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

        # This is the first entity created for the space
        assert( nearbyEntity is None )
```

```
    def onDestroy( self ):
        # Destroy the space and all entities in it
        self.destroySpace()

    def addGeometryMapping( self, geometryToMap ):
        # The base informs us what geometry to map.
        BigWorld.addSpaceGeometryMapping( self.spaceID, None, geometryToMap )
```

Example tutorial/res/scripts/cell/Space.py

3.3. Server-side personality scripts

Just like the client, the server uses personality scripts to perform bootstrap functionality on each CellApp and BaseApp. For the moment, we are only interested in the `onBaseAppReady` callback in the BaseApp personality script, which we will use to create a space.

Our initial revision of `tutorial/res/scripts/base/BWPersonality.py` is displayed below:

```
# Base bootstrap script
import BigWorld

def onInit( isReload ):
   pass

def onBaseAppReady( isBootstrap, didAutoLoadEntitiesFromDB ):
   # Only on the first baseapp
   if isBootstrap:
     # Create a Space entity that will create a space with our geometry.
     BigWorld.createBaseLocally( "Space", spaceDir = "spaces/main" )
```

Example tutorial/res/scripts/base/BWPersonality.py

A `Space` entity is created with the `spaceDir` property set to `"spaces/main"`.

Our initial revision of `tutorial/res/scripts/cell/BWPersonality.py` is displayed below:

```
# Base bootstrap script
import BigWorld

def onInit( isReload ):
   pass

def onCellAppReady( isFromDB ):
   pass
```

Example tutorial/res/scripts/cell/BWPersonality.py

Our implementation of the scripts is trivial and provides only stub implementations of callbacks that will be explained later.

For a complete list of the available personality script callbacks, see the documentation for the `BWPersonality` module in BaseApp Python API, CellApp Python API, and Client Python API.

3.4. The server-side `Avatar` scripts

The next step is to define the server-side logic that goes with our `Avatar` class. Even if we did not want to define any server-side logic for our `Avatar`, we would still need to provide at least stub implementations

of Avatar.py in the base and cell directories so that the base and cell parts of our Avatar entity can be created.

First we need to define the base part of the Avatar in tutorial/res/scripts/base/Avatar.py:

```
import BigWorld

# Must derive from BigWorld.Proxy instead of BigWorld.Base if this entity type
# is to be controlled by the player.
class Avatar( BigWorld.Proxy ):

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

      # Set our spawn position.
      self.cellData[ "position" ] = (0,0,0)

      # Spawn in the default space.
      self.createCellEntity( BigWorld.globalBases[ "DefaultSpace" ].cell )

   def onClientDeath( self ):
      # We ensure our cell entity is destroyed when the client disconnects.
      self.destroyCellEntity()

   def onLoseCell( self ):
      # Once our cell entity is destroyed, it is safe to clean up the Proxy.
 We cannot
      # just call self.destroy() in onClientDeath() above, as
 destroyCellEntity()
      # is asynchronous and the cell entity would still exist at that point.
      self.destroy()
```

Example tutorial/res/scripts/base/Avatar.py

The constructor for the base entity creates the cell entity in our space created earlier. It was registered in BigWorld.globalBases as "DefaultSpace".

There is a little bit of housekeeping here too — we have provided implementations for the onClientDeath and onLoseCell callbacks, which clean up the cell and base parts of the entity when the client disconnects from the server.

At this stage we do not need to define any interesting logic on the cell entity, so we provide a stub implementation in tutorial/res/scripts/cell/Avatar.py:

```
import BigWorld

class Avatar( BigWorld.Entity ):
   def __init__( self, nearbyEntity ):
      BigWorld.Entity.__init__( self )
```

Example tutorial/res/scripts/cell/Avatar.py

3.5. Connecting the client to the server

We need to add code to our basic client to have it connect to a server. If you have used FantasyDemo, you will have experienced the various GUI-based methods that can be used to connect to a server. Since we are not writing GUI code yet, we will just enter the address of our server into tutorial/res/scripts_config.xml and have the personality script read it from there.

We will also add an entry to control whether the client should attempt to connect to a server, or just explore the space offline as in the previous stage of the tutorial.

The relevant changes to `tutorial/res/scripts_config.xml` are displayed below:

```
...
   <server>
      <online> true </online>
      <!-- Whether the client actually connects to the server. -->
      <host> 10.40.3.23 </host>
      <!-- The server to connect to. Ideally we would allow this to be entered
 via an in-game
           GUI (or leverage the server discovery stuff) but for now we'll just
 hardcode it. -->
   </server>
...
```

Example `tutorial/res/scripts_config.xml`

Note

If you are using multiple users on the same server machine, you will need to specify the port as well as the IP address. The port for the LoginApp can be found by inspecting the `loginapp/nubExternal/address` watcher value on the Web Console. For example, if the IP address is `10.40.3.23` and it is on port `20013`, then put `10.40.3.23:20013` inside the `<host>` tag. You may need to update the port after restarting the server.

The next step is to implement the function call `initOnline` in the client personality script `tutorial/res/scripts/client/BWPersonality.py` and switch between calling it and calling `initOffline` based on the `online` option in `tutorial/res/scripts_config.xml`.

To achieve that, make the changes to `tutorial/res/scripts/client/BWPersonality.py` as illustrated below.

```
...
def init( scriptConfig, engineConfig, prefs ):
   if scriptConfig.readBool( "server/online" ):
      initOnline( scriptConfig )
   else:
      initOffline( scriptConfig )
...

def initOnline( scriptConfig ):
   class LoginParams( object ):
      pass

   def onConnect( stage, step, err = "" ):
      pass

   # Connect to the server with an empty username and password.  This works
   # because the server has been set up to allow logins for any user/pass.
   BigWorld.connect( scriptConfig.readString( "server/host" ),
                     LoginParams(), onConnect )
```

Example `tutorial/res/scripts/client/BWPersonality.py`

Notice that we no longer need to do client-side space creation, geometry mapping, or entity creation; these functions now happen on the server side. The client will automatically perform the necessary client-side actions based on the server-side game state.

3.6. Going 3rd person

The last line of `initOffline` in the personality script sets the camera to use first-person mode. We chose to do this in the first part of the tutorial because we wanted to get a client up and running as quickly and simply as possible, and using first-person mode allowed us to ignore the issue of rendering the player himself.

However, since we are now implementing a client-server game where multiple clients can log in and inhabit the same space, it will be helpful if they have models so that they can see each other!

We have provided a basic biped model in `res/characters/bipedgirl.model`, which we will use for all Avatars. Edit the `enterWorld` callback for the `Avatar` class in `tutorial/res/scripts/client/Avatar.py` as follows:

```
...
class Avatar( BigWorld.Entity ):

   def enterWorld( self ):

      # Set the position/movement filter to correspond to an avatar
      self.filter = BigWorld.AvatarFilter()

      # Load up the bipedgirl model
      self.model = BigWorld.Model( "characters/bipedgirl.model" )
...
```

Example `tutorial/res/scripts/client/Avatar.py`

3.7. Server-side XML configuration

The BigWorld server uses the file *`your_game`*`/res/server/bw.xml` for configuring options on the various server components. For a comprehensive list of configuration options along with a detailed description, see the Server Operations Guide's section *Server Configuration with* `bw.xml`.

Typically, the `bw.xml` file *includes* a BigWorld provided default configuration file which contains recommended default values for all the available configuration options. This is achived by using the `<parentFile>` tag as follows:

```
<root>
   ...
   <parentFile> server/development_defaults.xml </parentFile>
```

You will notice that in the example above, the included file is `development_defaults.xml`. This file provides good working defaults for a game development environment that will generate more warnings and intentionally crash the server in certain circumstances to ensure that critical issues are caught prior to the release of a game. The development defaults file however is only a small file that modifies a subset of values from the file `bigworld/res/server/production_defaults.xml`. The production defaults file aims to provide a comprehensive set of options and default values to be used for a game in a live production environment and can be used as a reference point when searching for a specific option.

While it is anticipated that the majority of configuration options will not need to be modified, if you need to change a value or are simply curious as to the purpose of an option, complete documentation for the BigWorld

server configuration options can be found in the Server Operations Guide's section *Server Configuration with* `bw.xml`.

To get our basic game up and running, we need to set a few options to specify what entity type the player should be connected to once logged in, and to allow players to log in with unknown usernames (just for convenience while developing).

Save the following in `tutorial/res/server/bw.xml`:

```
<root>
   <parentFile> server/development_defaults.xml </parentFile>
   <billingSystem>
      <entityTypeForUnknownUsers>  Avatar </entityTypeForUnknownUsers>
      <shouldAcceptUnknownUsers>   true   </shouldAcceptUnknownUsers>
      <shouldRememberUnknownUsers> false  </shouldRememberUnknownUsers>
   </billingSystem>
</root>
```

Example `tutorial/res/server/bw.xml`

3.8. Starting and connecting to the server

At this point of the tutorial, it is assumed that you have set up your Linux machine as described in the Server Installation Guide. In particular, this assumes you have installed BWMachined on your Linux machine and have installed the Web Console somewhere on the local network. For details on Web Console see the Server Operations Guide's section *Cluster Administration Tools*, in "WebConsole").

Before we can start the server, we need to specify where the server should get its binaries and resources from. This is a concept similar to the `paths.xml` files used by the client and tools.

We firstly need to know the directory the game resources are located on the Linux machine. If you have been developing the game resources on your Windows machine and have shared them using the setup_win_dev script, the resources are most likely located in `$HOME/bigworld_windows_share`. Check the directory where you believe the resources are located actually contain the correct files. For example:

```
$ ls $HOME/bigworld_windows_share
bigworld  fantasydemo  my_game  readme.html  server_installation  template
 tutorial
```

We now run the bw_configure script providing the location of the game resources we wish to use. This will differ slightly depending on the BigWorld Edition you are using.

3.8.1. Indie Edition

```
$ bw_configure
Game resource path [~/my_game/res]: ~/bigworld_windows_share/tutorial/res
Writing to /home/fred/.bwmachined.conf succeeded

Installation root : /opt/bigworld/current/server
BigWorld resources: /opt/bigworld/current/server/res
Game resources    : /home/fred/bigworld_windows_share/tutorial/res
```

The contents of the file `$HOME/.bwmachined.conf`[1] has now become:

[1] Note the leading `.` in the filename.

```
# Generated by ./bw_configure
/opt/bigworld/current/server;/home/fred/bigworld_windows_share/tutorial/res:/
opt/bigworld/current/server/res
```

Example $HOME/.bwmachined.conf

This file can then be edited whenever required to update the resource paths as your game development proceeds.

3.8.2. Commercial/Indie Source Edition

```
$ bw_configure
Installation root [~/mf]: ~/mf
Game resource path [~/my_game/res]: ~/mf/tutorial/res
Writing to /home/fred/.bwmachined.conf succeeded

Installation root : /home/fred/mf
BigWorld resources: /home/fred/mf/bigworld/res
Game resources    : /home/fred/mf/tutorial/res
```

The contents of the file `$HOME/.bwmachined.conf`[2] has now become:

```
# Generated by ./bw_configure
/opt/bigworld/current/server;/home/fred/bigworld_windows_share/tutorial/res:/
opt/bigworld/current/server/res
```

Example $HOME/.bwmachined.conf

This file can then be edited whenever required to update the resource paths as your game development proceeds.

3.8.3. Starting a Server

You can now use the WebConsole's ClusterControl module to start the server. You should see six active processes in the process listing. Once the server is up and running, run the client and you should be able to connect to the server and control a basic biped Avatar from a 3rd person perspective. Connect multiple clients and watch each other moving around.

[2] Note the leading . in the filename.

Chapter 4. Implementing a chat system (CHAT_CONSOLE)

At this stage we have a basic client-server game working, so it is a good time to write our first entity methods and learn how method calls propagate in BigWorld.

As an easy first example, we will write a simple chat system that allows players to talk to the other players around them. The implementation is in two parts:

- Implementing a basic GUI for displaying and entering chat messages on the client.
- Writing the entity methods to propagate the messages between clients and the server.

4.1. GUI text console

The example tutorial scripts currently use the deprecated ConsoleGUIComponent. Please refer to the Fantasy Demo scripts for an example chat console implementation (`fantasydemo/res/scripts/client/FDGUI/ChatConsole.py`) for reference on implementing a fully featured chat window.

4.2. Modifications to the `Avatar` entity

We need to implement methods on both the client and the server to make our chat system work:

- The server-side methods are responsible for receiving messages and forwarding them to other clients whose player entities are close enough to the speaker.
- The client-side methods are responsible for displaying incoming messages on-screen.

Before implementing these methods, they need to be declared in `tutorial/res/scripts/entity_defs/Avatar.def`:

```
...
   <ClientMethods>
      <!-- Chat to people within 50 metres -->
      <say>
         <Arg> UNICODE_STRING </Arg> <!-- message -->
         <DetailDistance> 50     </DetailDistance>
      </say>
   </ClientMethods>
   <CellMethods>
      <!-- Cell part of the chat implementation -->
      <say>
         <Exposed/>
         <Arg>            UNICODE_STRING </Arg>
      </say>
   </CellMethods>
...
```

Example `tutorial/res/scripts/entity_defs/Avatar.def`

The step above adds the method definitions to the previously empty client and cell method sections. The cell method definition includes the `<Exposed/>` tag, which exposes the method to the client. Without this, the method cannot be called from the client. The definition file also uses BigWorld's method LODing feature, by declaring a `<DetailDistance>` of 50m, which means that referring to `self.allClients` or `self.otherClients` from within this method will not refer to all clients in that entity's AoI, just those within 50m.

Having declared these methods, we must now provide their implementations. In `tutorial/res/scripts/cell/Avatar.py`, add the following:

```
...
   def say( self, id, message ):
      if self.id == id:
         self.otherClients.say( message )
```

Example `tutorial/res/scripts/cell/Avatar.py`

Even though we prototyped the cell method to take only the message as an argument in the definition file, our implementation expects another argument (`id`) before the declared arguments. This is because this method was declared as `<Exposed/>`, and the ID passed as an argument is that of the client who called the exposed method. Please note that this may not be the client who is attached to this `Avatar`, so we add a check to make sure the calling client is in fact the owner of this entity.

Note

We only forward the message to `self.otherClients`, not to `self.allClients`. This is because in our earlier implementation of `ChatConsole.editCallback` in `tutorial/res/scripts/client/Helpers/ChatConsole.py` (for details, see "GUI text console" on page 27) when the user enters a line of text it is immediately displayed on his client, so we do not want to send the message back to him. Therefore, we only need to call the `say` method on other clients.

Now we implement the client entity's `say` method in `tutorial/res/scripts/client/Avatar.py`:

```
class Avatar( BigWorld.Entity ):
   ...
   def say( self, msg ):
      chatConsole.write( "%d says: %s" % (self.id, msg) )
```

Example `tutorial/res/scripts/client/Avatar.py`

Now you should have a basic usable chat system. Connect a couple of clients to a running server and test it out!

Chapter 5. EntityLoader (ENTITY_LOADER)

Currently, the server loads the spaces/main space as the default space on startup. However, it is only the CellApp which is loading the space, and it is only loading the space geometry. In order to be able to place entities in World Editor and have them appear on the server, we need to create a more advanced space loading mechanism. To this end, we will make a helper class called EntityLoader which will be used by the Space entity. It will be the responsibility of this class to parse the space `.chunk` files and create entity instances for every entity encountered.

Note

You may be wondering why the engine doesn't just create entities automatically. While it could, this would remove flexibility from the scripts. This way, the game specific scripts are able to tailor how and when entities are created.

While at the end of this chapter it will appear to the end-user that nothing has changed, we will have laid the groundwork for the next chapter which covers creation of a editor placeable entity. Inspecting the BaseApp logs after running this server shows that it was unable to actually load the Greeter entity. This is added in the next chapter.

5.1. Implementation

The EntityLoader class exists only on the base entity, and in this tutorial will be implemented in the same Python module file as the Space entity. The following operations are performed:

- The Space entity creates a new instance of the `EntityLoader` class, and passes that instance into the `BigWorld.fetchEntitiesFromChunks` function. This function instructs the BaseApp to parse all `.chunk` files in the given path, which is done asynchronously in the background loading thread (in order to avoid IO from blocking the main thread).
- Whenever a non client only `<entity>` section is encountered within the `.chunk` files, the engine will call `EntityLoader.onSection` with the relevant `<DataSection>`.
- The script uses the properties passed in to the onSection method in order to create an entity instance using `BigWorld.createBaseAnywhere`. It passes the `Space` entity's cell mailbox so that the new entity knows which space to create itself in. Note that this paradigm assumes that all entity scripts will accept `createOnCell` as a property.
- The engine notifies the `EntityLoader` when chunks have finished being parsed via the `onFinish` callback.

```
# scripts/base/Space.py

class Space( BigWorld.Base ):
    ...

    def onGetCell( self ):
        print "Space.onGetCell loading entities from '%s'" % self.spaceDir
        BigWorld.fetchEntitiesFromChunks( self.spaceDir,
            EntityLoader( self ) )
    ...

class EntityLoader( object ):
    def __init__( self, spaceEntity ):
        self.spaceEntity = spaceEntity
```

```
    def onSection( self, entity, matrix ):
        entityType = entity.readString( "type" )
        properties = entity[ "properties" ]
        pos = matrix.applyToOrigin()

        # Create entity base
        BigWorld.createBaseAnywhere( entityType,
            properties,
            createOnCell = self.spaceEntity.cell,
            position = pos,
            direction = (matrix.roll, matrix.pitch, matrix.yaw) )

    def onFinish( self ):
        print "Finished loading entities for space", self.spaceEntity.spaceDir
```

Adjusted example `tutorial/res/scripts/base/Space.py`

Chapter 6. A Basic NPC Entity (BASIC_NPC)

This chapter will cover the basic steps of creating a non-player entity. While the entity presented is quite simple in terms of functionality, it covers all the common essentials required in order to get a new entity up and running in the engine (including exporting the model from 3D Studio Max and configuring the model to work correctly).

6.1. Design

Before creating an entity, we need to determine what functionality is required. For this tutorial we will to create an NPC which will greet a player when they get within a certain radius (think of a person who stands in a supermarket entrance greeting people).

Entity requirements:

- It should be placeable in the World Editor so that the run-time instance is created by the SpaceLoader entity on the server.
- Model should be loaded asynchronously on the client.
- The entity will not move. It should stand on the spot as placed in the World Editor.
- A server-side trap should be used to trigger a greet action. The server should notify all clients in the area that it has greeted a player (including which player).
- When the entity greets a player, a wave animation should be played on all clients in the area.
- A message, generated on the server, should be displayed above the Greeter's head for a couple of seconds.
- It should be possible to deactivate (and reactivate) the Greeter from the client, but only if the client is within the trigger radius.

We shall give this entity the class name `Greeter`.

6.2. Art

For this tutorial, we have provided the 3D Studio Max source file to the Barbarian model (a fantasy themed human). The model needs to prepared for use by the engine and needs to be configured to satisfy the requirements of the Greeter entity. This section assumes the BigWorld exporters have already been installed (see the Content Tools Reference Guide).

Detailed documentation about the exporters and tools can be found in the Content Tools Reference Guide and the Content Creation Manual.

Note

If you want to skip this section, the barbarian model has been pre-prepared for this tutorial (in `C:/bigworld/tutorial/res/characters`).

6.2.1. Exporting the model

1. Open `tutorial/sourceart/barbarian.max` in 3D Studio Max.
2. Copy the textures to `tutorial/res/characters`. The textures must be in the target directory before exporting (the exporter will display an error message and fail if they are not).
3. Re-apply the textures to the model so that the Max scene points to the textures copied in the step above.
4. Go to **File → Export** and choose the BigWorld visual exporter.

5. Save the model to `tutorial/res/characters/barbarian.model`.

6.2.2. Configuring the model

In order to automatically play an idle animation and to create the action required for the Greeter entity to wave, we need to add animations to the model and configure the appropriate actions. This is done in the Model Editor.

While an animation is a raw sequence of key frames, an action is a higher level concept. Actions are animation wrappers that contain extra information such as animation blending and what game-play situations will trigger the animation (e.g. idling, walking or running based on the velocity of the entity).

The Greeter will have two actions: an Idle action which is automatically selected when the entity is standing still, and a Wave action which will be explicitly invoked from the Greeter's Python scripts.

1. Open up `tutorial/res/characters/barbarian.model` in Model Editor.
2. Before we can setup the actions, we need to add references to the idle and wave animations. In the Animations tab, click the "New animation" button and select the `tutorial/res/characters/idle_a.animation` animation file. A new animation will be added to the list which can be previewed in the 3D view. Repeat this for the `m_waveonehand.animation` file.
3. To setup an Idle action that is automatically invoked when the entity is standing still,
   a. Open the Actions tab in Model Editor.
   b. Click the New Action button and select the m_idle animation in the pop-up dialog. Set the action name to Idle.
   c. Select the new Idle action from the list.
   d. Setup the parameters in the Match section to allow the action matcher to automatically select the action when the entity is not moving. To do this set the following values:
      - Minimum speed=0.0, Maximum speed=0.0
      - Minimum turn=-360.0, Maximum turn=360.0
      - Minimum direction=-360.0, Maximum direction=360.0

      As you can see, the action will be picked whenever the speed of the entity is exactly zero and is facing in any direction.
4. To setup a Wave action that is invoked explicitly by the Python scripts (i.e. not automatically picked by the engine),
   - Click the New Action button and select the m_wave animation in the pop-up dialog. Set the action name to Wave.

   No match settings need to be set for this action since we will manually invoke the action from the Python scripts.

6.3. Scripts

In order to insert the model as an entity into a space, we need to create the entity scripts. These are written in Python and perform game-specific logic and are split up into three parts: base, cell, and client.

Please refer to the Python API reference documents[1] for detailed information on the API's mentioned here.

[1] BaseApp Python API, CellApp Python API, Client Python API.

6.3.1. entities.xml

First off, we need to tell the engine about our new entity. Every entity must be defined in the `entities.xml` file located at `tutorial/res` path.

Simply add a new empty tag between the `<root>`:

```
<root>
    <Space/>
    <Avatar/>
    <Greeter/>
</root>
```

Remember that since the entity name corresponds with a Python class name, the name used here must conform with Python naming rules.

6.3.2. Entity definition

In order to allow the engine to know what methods and properties the entity has, we need to create a special file known as the entity definition file. In some ways this file is the most important part of an entity, as it defines how properties and methods are handled by the engine (e.g. property type, whether or not a property or method is exposed to clients, prioritisation of remote method calls and property updates, and configuring distance based LoD parameters for individual properties).

See the Server Programming Guide chapter "The Entity Definition File" for a detailed description of entity definitions.

For the Greeter entity, create a new file named `Greeter.def` and place it in `tutorial/res/scripts/entity_defs/`. We will define the following information for our entity:

- Three properties:
  - A `radius` property which controls the trigger region for the Greeter. This is exposed to the World Editor so that it can be tweaked by the world builder. Its type is `FLOAT`, it has a default value of 3 metres and is declared as `CELL_PRIVATE` (since this property is only needed on the cell part of the entity and does not need to be publicly accessible by other entities).

    Note

    To provide a more intuitive interface for the World Editor, some extra meta-data has been defined for this property. The `RADIUS` widget allows the property to be manipulated via a visual spherical widget.

  - A property named `activated` which is a boolean property representing whether or not the Greeter is currently active. It's flags is set to `ALL_CLIENTS` so that changes to this property on the cell are automatically propagated to the clients.
  - The `createOnCell` property which indicates which space the entity should be created in (a requirement for entities loaded via the `SpaceLoader` entity).
- Two methods:
  - A client-side method named `greet`. This will be remotely called by the server on all nearby clients whenever the entity greets a player (i.e. whenever the server-side trap is triggered). It takes two parameters, the ID of the entity is greeting, and a personalised greet message.

- A method called `toggleActive` which is exposed to the client which allows the client to toggle the Greeter on and off. By default methods are not callable by the client (for security purposes), so the `<Exposed>` keyword is used to explicitly expose it to clients. It does not take any arguments.

```
<root>
    <Properties>
        <radius>
            <Type>          FLOAT
                <Widget>    RADIUS
                    <colour>    255 0 0 192   </colour>
                    <gizmoRadius>    2        </gizmoRadius>
                </Widget>
            </Type>
            <Flags>          CELL_PRIVATE    </Flags>
            <Default>        3.0             </Default>
            <Editable>       true            </Editable>
        </radius>

        <activated>
            <Type>           INT8            </Type>
            <Flags>          ALL_CLIENTS     </Flags>
            <Default>        1               </Default>
        </activated>

        <createOnCell>
             <Type>    MAILBOX        </Type>
             <Flags>   BASE           </Flags>
        </createOnCell>
    </Properties>

    <ClientMethods>
        <greet>
            <Arg> UINT32 </Arg> <!-- Entity ID of who we are greeting -->
            <Arg> STRING </Arg> <!-- Our greeting message -->
        </greet>
    </ClientMethods>

    <CellMethods>
        <toggleActive>
            <Exposed/>
        </toggleActive>
    </CellMethods>

    <BaseMethods>
    </BaseMethods>
</root>
```

Example `tutorial/res/scripts/entity_defs/Greeter.def`

6.3.3. Base part

The base part of the entity is the first part that gets created by the server. The base is created on one of the BaseApp processes, and is used to define entity logic which does not require spatial information (e.g. character inventory). The base part of an entity does not migrate between BaseApps after it has been created.

The base script for the Greeter entity is very simple and performs two tasks:

- It creates the cell part of the entity within the cell specified by the `createOnCell` property (as setup by the `EntityLoader` class when it loads the entity information from the space's chunk file).

- It destroys itself when the cell part of the entity disappears.

```
import BigWorld

class Greeter( BigWorld.Base ):
    def __init__( self ):
        BigWorld.Base.__init__( self )
        self.createCellEntity( self.createOnCell )

    def onLoseCell( self ):
        self.destroy()
```

Example `tutorial/res/scripts/base/Greeter.py`

6.3.4. Cell part

The cell part of an entity represents the current position, orientation, and movement for an entity within a particular space. Managed by the CellApp processes, the cell part of an entity can be moved between CellApp processes at any time based on CPU load. Generally, all entity logic that requires access to spatial information is implemented in the cell part of an entity (e.g. any code that needs to find out about other nearby entities, such as AI).

The cell part of the Greeter performs the following tasks:

- Creates a trap when the entity is created using the radius specified in the World Editor.
- Greets any Avatars that walk into the trap by calling `greet` on all clients that have the Greeter entity within their AoI.
- Allow clients to toggle `activated` state of the entity, but only if they are within the radius. Note that exposing a method to the client implicitly adds an argument which is the ID of the Avatar entity which invoked the method. This can (and should) be used to validate that the Avatar is actually allowed to perform the desired command (remember, never trust the client).

Cell entities must derive from the `BigWorld.Entity` class.

```
import BigWorld
import Avatar
import random

MESSAGES = [ "Hello BigWorld", "Have a nice day" ]

class Greeter( BigWorld.Entity ):

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

        # Setup the trap
        self.addProximity( self.radius, 0 )

    def onEnterTrap( self, entityEntering, range, controllerID ):
        # If we are not active, do nothing.
        if not self.activated:
            return

        # Filter by entity class type
        if not isinstance( entityEntering, Avatar.Avatar ):
            return
```

```
        # Notify clients.
        self.allClients.greet( entityEntering.id, random.choice(MESSAGES) )

    def toggleActive( self, sourceID ):
        # Get the entity who called us. If the entity can't be found then they
        # obviously not near by so just bail out.
        try:
            sourceEntity = BigWorld.entities[ sourceID ]
        except KeyError:
            return

        # Get the distance between ourself and the Avatar
        dist = sourceEntity.position.distTo( self.position )

        # Do a check to make sure they are close enough.
        if dist > self.radius:
            return

        # All good, toggle our state. The activated property will be
 automatically
        # propagated to all clients once this server tick is complete.
        self.activated = not self.activated
```

Example `tutorial/res/scripts/cell/Greeter.py`

6.3.5. Client part

The client part of the entity is automatically created by the engine whenever an entity appears within your Avatar's area of interest (AoI). It is the job of the client scripts to coordinate all resources and logic required to represent the entity on the client based on the information provided by the server.

6.3.5.1. Entity module

The client-side of an entity must derive from `BigWorld.Entity`. The bare-bones Greeter module script looks like this:

```
# Greeter.py

import BigWorld
import GUI
import Math

class Greeter( BigWorld.Entity ):
    def __init__( self ):
        BigWorld.Entity.__init__( self )
```

Basic structure of `tutorial/res/scripts/cell/Greeter.py`

6.3.5.2. Prerequisites list

To avoid stalling the main thread client when the entity is created, we will use the prerequisites functionality to load the model asynchronously in the background loading thread. This is done by implementing the `prerequisites` method which returns a list of resources to be loaded. This means that whenever the server notifies the client that a Greeter entity has entered the AoI for the client, the client will first schedule the resources to be loaded asynchronously.

```
GREETER_MODEL_NAME = "characters/barbarian.model"
```

```
class Greeter( BigWorld.Entity ):
    ....

    def prerequisites( self ):
        return [ GREETER_MODEL_NAME ]
```

6.3.5.3. Entering and leaving the world

Once the prerequisite resources have been loaded, the `onEnterWorld` method is called. Since the entity class instance can leave the AoI and then re-enter the AoI, the bulk of the initialisation code will be done in here rather than in `__init__` (so it can re-initialised each time).

For the `Greeter` entity, the primary entity model (`Entity.model`) is set, and a network filter is setup. Since the entity will not be moving around, we can use a simple `DumbFilter` which simply snaps the entity to the last network update.

```
class Greeter( BigWorld.Entity ):
    ....

    def onEnterWorld( self, prereqs ):
        # Setup our model.
        self.model = BigWorld.Model( GREETER_MODEL_NAME )

        # Setup an appropriate filter.
        self.filter = BigWorld.DumbFilter()

    def onLeaveWorld( self ):
        # Clean up.
        self.model = None
        self.filter = None
```

6.3.5.4. Implementing greet

The bulk of the client-side logic for the Greeter entity will go in the implementation of the `greet` method. This method is remotely called from the cell part whenever an `Avatar` enters the trap.

```
class Greeter( BigWorld.Entity ):
    ....

    def greet( self, targetID, msg ):
        # Grab the entity instance, if for some reason we don't have it just
 do nothing.
        try:
            targetEntity = BigWorld.entities[targetID]
        except KeyError:
            return

        # Try to play the Wave action. If it doesn't exist, print a warning.
        try:
            self.model.Wave()
        except AttributeError:
            print "WARNING: Greeter model missing Wave action (%s)" %
 self.model.sources

        # Display the greet message above our head.
        addressee = targetEntity.name
        if targetID == BigWorld.player().id:
            addressee += "! Yes you"
```

```
        self._displayMessage( "Hey %s! '%s'!" % (addressee, msg) )
```

6.3.5.5. Displaying the message

The script that displays a text message above the Greeter's head will be implemented in a private helper method called `_displayMessage` (note the usage of an underscore to denote a private member - this is not required but it is a useful convention to follow). The `TextGUIComponent` class from the GUI module will be used and will be inserted into the 3D scene using the `GUI.Attachment` class (as opposed to being rendered in screen space). The text is attached to the root node of the entity model and is positioned above the head of the model by inspecting the model's `height` attribute.

```
class Greeter( BigWorld.Entity ):
    ....

    def _displayMessage( self, msg ):
        # First make sure any previous message is cleared.
        self._clearMessage()

        # Create our text component. Since we want to display it in the world
        # we shall explicitly set our width and height in world units.
        text = GUI.Text( msg )
        text.explicitSize = True
        text.size = ( 0, 0.5 )             # Specifying 0 for x to
 auto-calculate aspect ratio.
        text.colour = (255, 0, 0, 255)     # Change the colour.
        text.filterType = "LINEAR"         # Don't use point filtering.
        text.verticalAnchor = "BOTTOM"     # Position relative to the bottom
 of the text.

        # The origin of our model is at our feet. To place the text above
        # our head, move it up on the Y by our model's height.
        text.position = (0, self.model.height + 0.1, 0)

        # Setup our GUI->World attachment. Tell it that we want the GUI
        # component to always face the camera.
        atch = GUI.Attachment()
        atch.component = text
        atch.faceCamera = True

        # Attach to our model's root node.
        self.model.root.attach( atch )

        # Save a reference to the attachment so we can clean it up later.
        self._messageAttachment = atch

        # Setup the timer.
        self._setMessageHideTimer()
```

To make the message disappear after a certain amount of time, the `BigWorld.callback` function is used. The hide message timer functionality is wrapped up in some additional helper methods.

- `_clearMessage` clears any existing message attachment above the entity's head.
- `_setMessageHideTimer` sets up the timer, while first cancelling any existing timer.
- `_cancelMessageTimer` cancels the timer by passing the previously created timer handle into `BigWorld.cancelCallback`.
- `_handleMessageHideTimer` is the Python callable that is given to `BigWorld.callback`. It is executed after the timer has elapsed, clearing the stored timer handle and removing the current message.

```
class Greeter( BigWorld.Entity ):
    ....

    def _clearMessage( self ):
        self._cancelMessageTimer()
        if self._messageAttachment is not None:
            self.model.root.detach( self._messageAttachment )
            self._messageAttachment = None

    def _setMessageHideTimer( self, timeout=5.0 ):
        self._cancelMessageTimer()
        self._messageTimerHandle = \
            BigWorld.callback( timeout, self._handleMessageHideTimer )

    def _cancelMessageTimer( self ):
        if self._messageTimerHandle is not None:
            BigWorld.cancelCallback( self._messageTimerHandle )
            self._messageTimerHandle = None

    def _handleMessageHideTimer( self ):
        self._messageTimerHandle = None
        self._clearMessage()
```

6.3.5.6. Handling activation change

The engine will automatically notify the entity script whenever a property has been changed by the server. It does this by looking for a method on the entity class named `set_propertyName` which is expected to take a single parameter for the previous value of the property. The Greeter script will take advantage of this notification and display a message whenever the `activated` state has changed.

```
    def set_activated( self, oldValue ):
        if self.activated:
            self._displayMessage( "Alright! I'm now ready to GREET." )
        else:
            self._displayMessage( "Shutting up now." )
```

6.3.6. Editor script

The editor script for an entity allows programmatic control over how the entity behaves in the World Editor. For the Greeter entity, the script will simply override the default model used to represent the entity in the editor (it otherwise defaults to a red box).

Editor scripts are located in `res/scripts/editor`.

```
class Greeter:
    def modelName( self, props ):
        return "characters/barbarian.model"
```

6.4. Testing

To test the entity it will first need to be placed into a space in World Editor. Open the `spaces/main` and place the entity by dragging the Greeter entity into the scene from the Resources tab. Save the space.

If you are not using a Windows mount, update the resources on the server side and then restart the server. If all is well, you should be able to connect as per-normal and see the Greeter in the space.

Greeter entity in action

If you do not see the entity, there are a couple of things to check:

- Check the server startup logs for any Python exceptions.
- Check the cell logs to make sure the entity is actually being created. You should see a message along the lines of:

```
CellApp INFO Cell::createEntity: New Greeter
        (2)
```

- Check the client for any client-side Python errors (e.g. bring up the in-game client console or use Debug View).

Note that at this point the only way to toggle the activation state is to use the in-game Python console. For example, on the client,

```
>>> $B.entities.items() # Find the ID for the Greeter
[(2402, Greeter at 0x088CFFE8), (2405, PlayerAvatar at 0x088CFC10)]
>>> greeter = $B.entities[2402]
>>> greeter.cell.toggleActive()
```

6.5. Possible improvements

While the entity satisfies the basic requirements, there are some improvements that could be made.

- The most obvious improvement would be to allow the user to toggle the active state of the Greeter entity by clicking on the entity. This could be achieved by leveraging the entity targeting system of the client. See the Client Python API documentation for `BigWorld.target`.

- If many player entities enter the trap at the same time, the client will try to greet everyone at once. Instead of simply playing the wave animation immediately when the `greet` method is called on the client, the client-side script could be designed so that greets are queued up so that the next greet will not commence until the previous greet has completed. This could be achieved by passing a callback into `model.Wave()` so that the scripts get notified when the current action has completed. See the Client Python API for `ActionQueuer.__call__` for information on how you can use action callbacks.

- Currently the Greeter entity simply plays the Wave action. It would be nice if the entity looked towards you while it is greeting you. A head tracker can be created by using the `BigWorld.Tracker` class coupled with the `BigWorld.TrackerNodeInfo` class.

- The player can cause the Greeter to spam greetings if they quickly move in and out of the trap radius. To avoid this problem, the cell part of the entity should keep track of recent greets (associate an entity ID with a time stamp). It should only re-greet a player if some time has elapsed since the previous greeting. This list should be added as a new property in `Greeter.def`, and additional logic placed in `Greeter.onEnterTrap`.