Tutorial

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. 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 standard 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 folder and build the project file by file, to give you a clear understanding of what each file and folder is for.

Note

Until this tutorial is complete, there may be slight inconsistencies between the source code in the accompanying `<res>` tree and the source code presented inline in the tutorial itself. Eventually this will all be made consistent, but for development purposes these issues are not dealt with strictly at the moment.

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 — those issues will be addressed in “Windows/Linux file sharing” on page 15 .

Note

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

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` module 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/res/scripts/common/tutorial.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>` folder in the tutorial folder (*e.g.*, `tutorial/res_client_server`). You can then alter the `paths.xml` and `.bwmachined.conf` settings to point to these stripped trees.

Note

For details on how to configure `paths.xml`, see "Setting up the client executable folder" on page 5, and "A simple space" on page 11. For details on `paths.xml`'s grammar, see the File Grammar Guide's section `paths.xml`.
For details on how to configure `.bwmachined.conf`, see "Starting and connecting to the server" on page 19.

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.

The symbolic constants for each chapter are given in the chapter heading.

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 folder.
- 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` folder in `C:\mf`. Following that convention, we will start our new tutorial project in the same folder, by creating a folder called `tutorial` in `C:\mf`. All resources and scripts specific to this project will be located within this folder.

2.2. Setting up the client executable folder

The client executable for a BigWorld game is independent of the game resources it loads (unless you have customised the client with your own extensions). Since the FantasyDemo executable does not have any such modifications, we can re-use it in our tutorial project.

The client executable typically resides in a folder called `game` in the root of your project. Simply copy the entire folder `fantasydemo/game` into the `tutorial` folder, and then rename `fantasydemo.exe` to `tutorial.exe` — this is not strictly required, but it is recommended to avoid confusing the tutorial executable with the shipped FantasyDemo executable.

Other than the client executable itself, this folder also contains the DLLs that the client needs to run, as well as a file called `paths.xml` which determines the folders from which the client loads its resources. The `paths.xml` file is set up for the FantasyDemo resources, so change it so it points to the tutorial resources instead:

```
<root>
   <Paths>
      <!-- Path defines the root location of any available resources. -->
      <!-- Multiple entries allow for multiple resource locations. -->
      <!-- Path precedence is in order of listing. -->
      <Path>  ../../tutorial/res  </Path>
      <Path>  ../../bigworld/res  </Path>
   </Paths>
</root>
```

Example `tutorial/paths.xml`

2.3. Creating the resources folder

Resource folders for BigWorld games are typically named `res`, therefore you can simply create a folder called `res` in the `tutorial` folder. This top-level resources folder 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` folder. One of the files that must exist in this folder is `entities.xml`, which lists the game entities that will be used — for details on this file, see the Server Programming Guide's section Physical Entity Structure for Scripting in The `entities.xml` file.

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 folder that must exist is `res/scripts/entity_defs`, which contains the `.def` files, with definitions of the properties and methods for each entity — for details on these files, see the Server Programming Guide's section Physical Entity Structure for Scripting, in The definition file.

It might be helpful to think of these definition files as being similar to C/C++ header files — 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>
         <Type>   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 — it defines properties for the entity, but not 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 important thing is to know their current value (the history of changes on the property is less important, and 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 Volatile 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 Data distribution.

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` folders.

Create each of these folders within the `tutorial/res/scripts` folder — your folder 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` folder, 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` (see "`entities.xml`" on page 6 and defined it in `tutorial/scripts/entity_defs/Avatar.def` (see "Defining the Avatar entity type"on page 6). 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 enterWorld( self ):
      pass

class PlayerAvatar( Avatar ):

   def enterWorld( self ):

      Avatar.enterWorld( self )

      # 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, isDown, key, mods ):

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

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

      # Save back the new velocity
      self.physics.velocity = v
```

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

Notice that the script declares two classes: `Avatar` and `PlayerAvatar` — this is 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 folder (*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 )

# 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

# Keyboard event handler
def handleKeyEvent( down, key, mods ):
   return False
```

```
# Mouse event handler
def handleMouseEvent( dx, dy, dz ):
    return False

# Joystick event handler
def handleAxisEvent( axis, value, dTime ):
    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 Overview, in Configuration files, 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 9).

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`

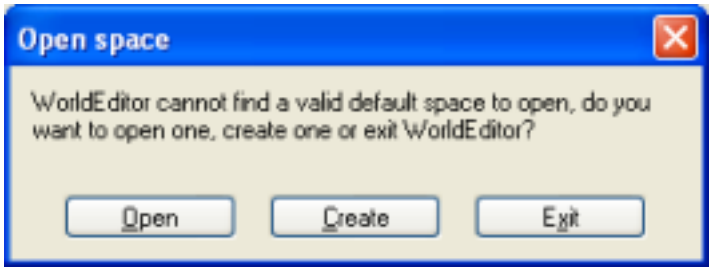
At this stage, the values for the configuration settings expected by the personality script's `init` method have been provided. The only thing still missing for our basic client is the actual space data. The script configuration passes the string `spaces/main` into the personality script as the space 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 up the application, copy `tutorial/game/paths.xml` to `bigworld/tools/worldeditor`, so that the editor is working with the same files and folders that the client loads. However, you will need to add an extra `../` to each path in this file, since `bigworld/tools/worldeditor` is one folder level deeper than `tutorial/game`.

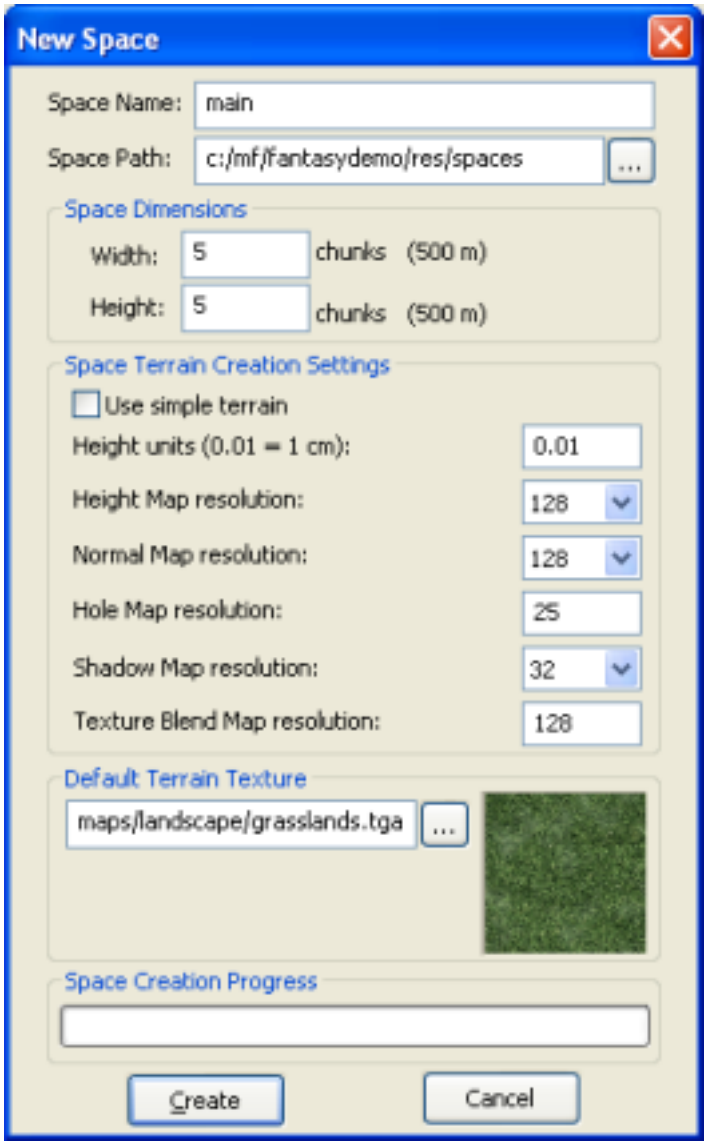
To create a simple space that can be navigated, follow the steps below:

- 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.

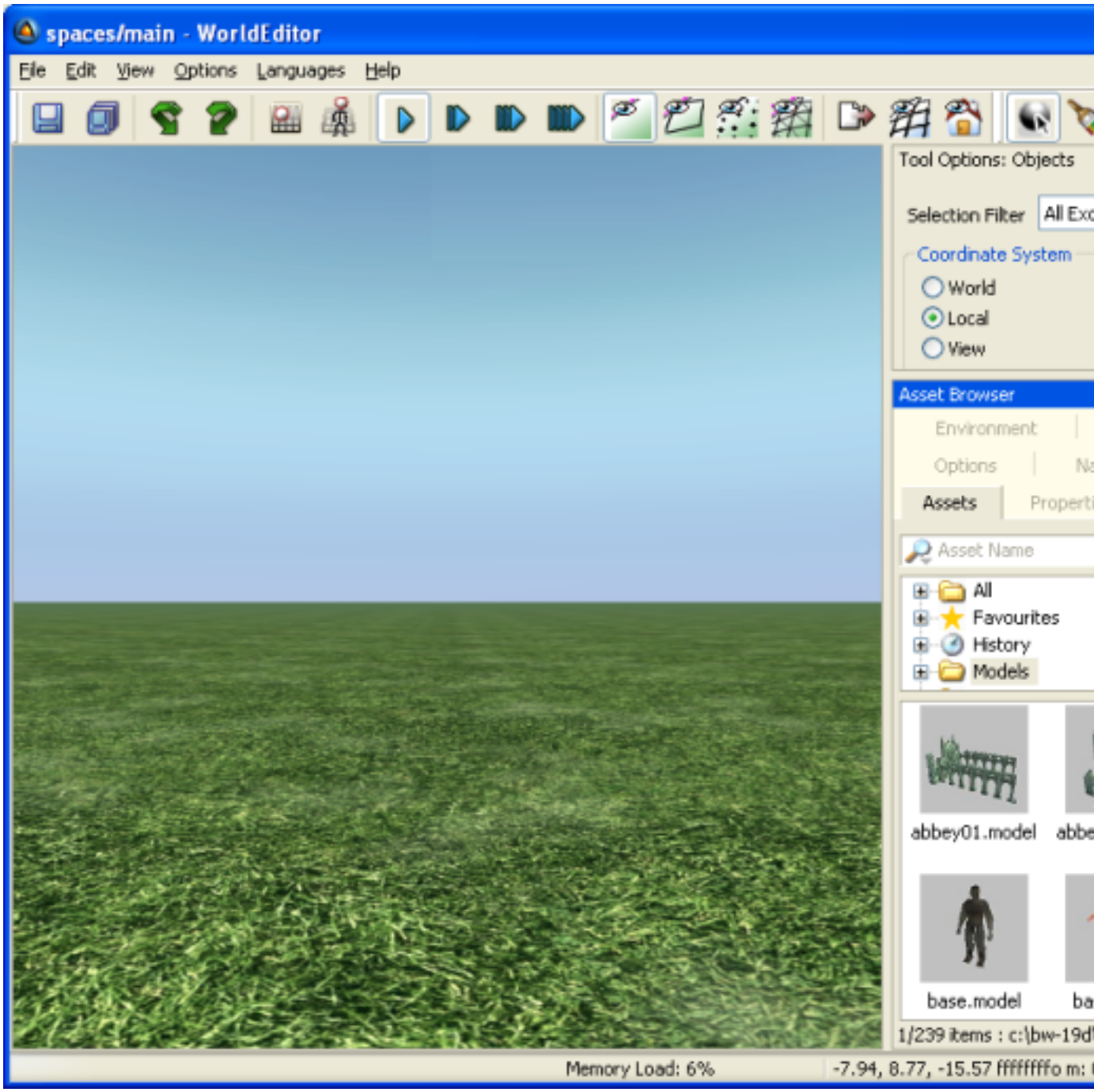


New Space dialog box

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.



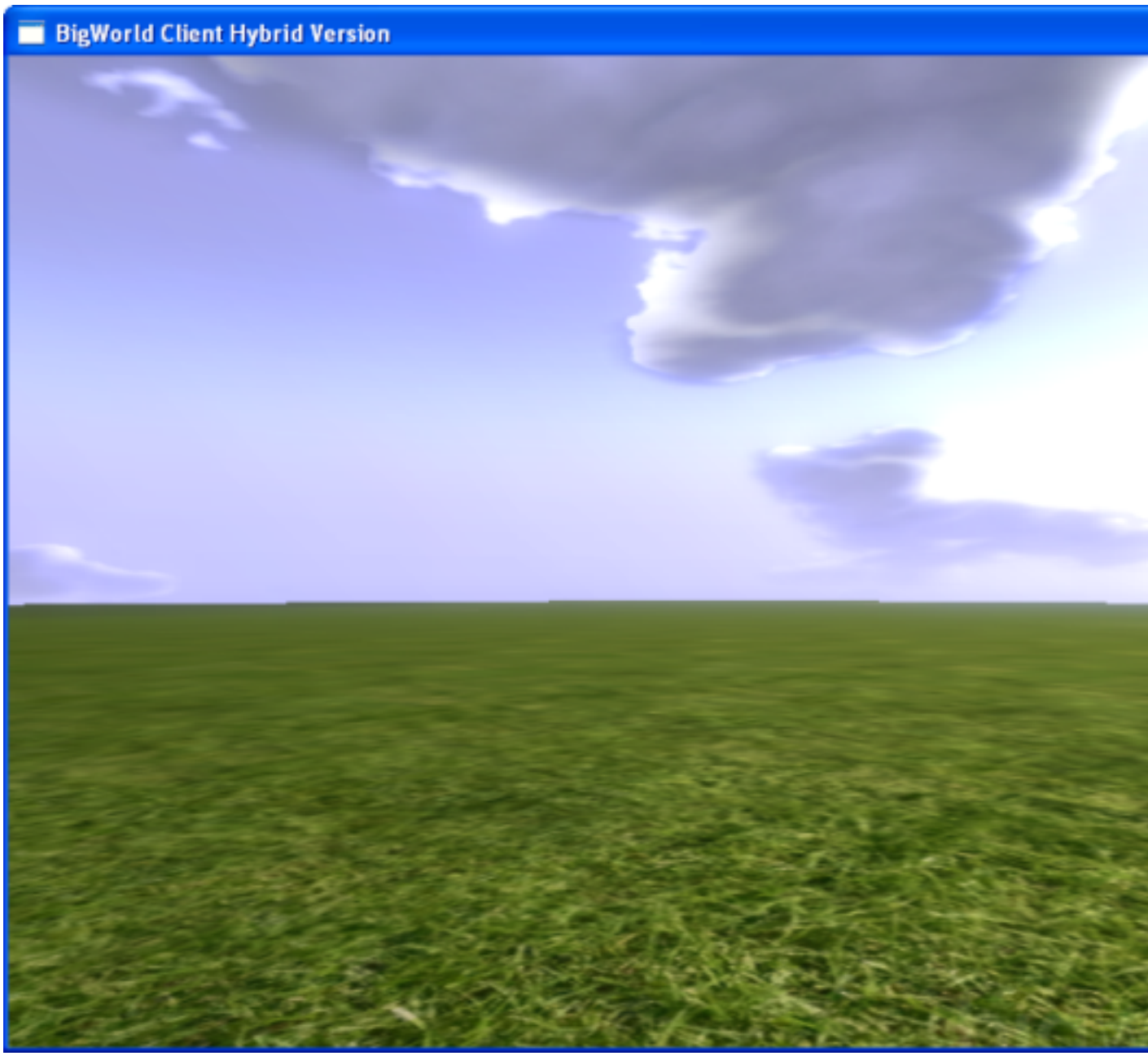
The `main` space

- 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, run `tutorial/game/tutorial.exe`. — 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 5 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. Windows/Linux file sharing

At this point, we need to address the issue of sharing files between our 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), we need 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.

The basic idea is to keep all files on the Windows machine, and export them as a network share that can be mounted by the Linux box. For details on how to do this, see the Client Programming Guide's section Simplified Server Usage.

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

3.2. 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 `onCellAppReady` callback in the CellApp personality script, which we will use to map the geometry for `spaces/main` onto the default space when the first CellApp is ready.

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

```
# Cell bootstrap script
import BigWorld

def onInit( isReload ):
   pass

def onCellAppReady( isFromDB ):
   # Map spaces/main onto the default space (which is always space 1)
   BigWorld.addSpaceGeometryMapping( 1, None, "spaces/main" )
```

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

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 ):
   pass
```

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

Our implementation of the scripts is trivial and provides only stub implementations of a couple of callbacks that we will use later.

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

3.3. 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 folders 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 )

      # Spawn in the default space once the cell entity has been created.
      self.createInDefaultSpace()

   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 simply creates the cell entity in the default space, which is the space we mapped geometry for spaces/main onto in our personality scripts in “The personality script” on page 9 .

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 ):
   pass
```

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

3.4. 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`

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.

3.5. 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`

3.6. Server-side XML configuration

The BigWorld server uses the file `res/server/bw.xml` for configuring options on the various server components (for details, see the Server Operations Guide's section Server Configuration with `bw.xml`).

Typically, this file *includes* the file `bigworld/res/server/defaults.xml` using the `parentFile` tag, which provides sensible defaults for all configurable options. In most cases, you do not need to alter those defaults. For details, see the Server Operations Guide's section Server Configuration with `bw.xml` , in The entry `parentFile`).

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/defaults.xml </parentFile>
   <dbMgr>
      <entityType>      Avatar </entityType>
      <loadUnknown>     true   </loadUnknown>
      <createUnknown>   true   </createUnknown>
      <rememberUnknown> false  </rememberUnknown>
   </dbMgr>
</root>
```

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

3.7. 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 `bwmachined2` 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 Admin 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. Save the following into `$HOME/.bwmachined.conf` (note the leading `.` in the code) on your Linux machine:

```
# Format is $MF_ROOT;$BW_RES_PATH
/home/fred/mf;/home/fred/mf/tutorial/res:/home/fred/mf/bigworld/res
```

Example `$HOME/.bwmachined.conf`

Now you can 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.

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. Basic GUI text console

The client side of our chat console implementation requires a way of entering and displaying chat messages. Fortunately for us, BigWorld already includes a text console component that we can leverage to provide this functionality in a basic form without too much work.

We will implement the chat console functionality in a class called `ChatConsole`, but seeing as this is a client-side only class, and not an entity in the BigWorld sense of the word, we will create a `Helpers` folder in `tutorial/res/scripts/client` where we will store the chat console script and other non-entity class implementations.

Note

An empty new file called `__init__.py` will need to be created in the `Helpers` folder to allow Python to import the modules. This can be done in Linux with the command 'touch'.

Save the following into `tutorial/res/scripts/client/Helpers/ChatConsole.py`:

```
import BigWorld
import GUI
import Keys

import collections

class ChatConsole( object ):
   """
   Leverages the GUI.Console object to provide a basic chat console.
   """

   sInstance = None

   def __init__( self, numVisibleLines = 4 ):
      """
      Create a new ChatConsole with a maximum number of visible lines.  Note
 that you cannot
      construct one of these things until after BWPersonality.init() has been
 called because
      it relies on resource stuff that is initialised prior to init().
      """
      self.numVisibleLines = numVisibleLines
      self.lines = collections.deque()
      self.timerID = None

      self.con = GUI.Console()
```

```
      self.con.editCallback = self.editCallback

      GUI.addRoot( self.con )
      ChatConsole.sInstance = self

   @classmethod
   def instance( cls ):
      """
      Static access to singleton instance.
      """
      if not cls.sInstance:
         cls.sInstance = ChatConsole()
      return cls.sInstance

   def write( self, msg ):
      """
      Append a new line of output to the console.
      """
      self.lines.append( msg )

      # Rotate out the oldest line if the ring is full
      if len( self.lines ) > self.numVisibleLines:
         self.lines.popleft()

      # Redraw all lines in the ring
      self.con.clear()
      for line in self.lines:
         self.con.prints( line + "\n" )

      # Hide the console after 10 seconds
      self.show()
      self.hide( 10 )

   def hide( self, delay = 0 ):
      """
      Hide the console in the specified number of seconds, or now if none
specified.
      """

      # Cancel any outstanding timers
      if self.timerID is not None:
         BigWorld.cancelCallback( self.timerID )
         self.timerID = None

      if delay == 0:
         self.con.visible = False
      else:
         BigWorld.callback( delay, self.hide )

   def show( self ):
      """
      Show the console immediately.
      """
      self.con.visible = True

   def editing( self, val = None ):
      """
      Enter/leave editing mode if val is passed, otherwise return whether in
editing mode.
      """
      if val is None:
         return self.con.editEnable
```

```
      else:
         self.show()
         self.con.editEnable = val

   def editCallback( self, line ):
      """
      Callback for when a line of input is entered.
      """
      # Send the line of input as a chat message
      BigWorld.player().cell.say( line )

      # Display it in our own console
      self.write( "You say: " + line )

      # Stop editing since the user has pressed ENTER
      self.editing( False )

   def handleKeyEvent( self, down, key, mods ):
      if down and key == Keys.KEY_ESCAPE:
         if self.editing():
            self.editing( False )
         else:
            self.hide()
          return True
      else:
         return self.con.handleKeyEvent( (down, key, mods) )
```

Example `tutorial/res/scripts/client/Helpers/ChatConsole.py`

In summary, the implementation provides a chat console with a fixed size buffer for on-screen messages. A lot of the heavy lifting is done by the `ConsoleGUIComponent` object created with `GUI.Console` in the constructor. Some points to note:

- We call `GUI.addRoot` after creating the underlying console object. Without this step, the console will not render.
- The console uses a timed callback, triggered with `BigWorld.callback` to hide the console after 10 seconds of inactivity.
- The definition of `editCallback` refers to a cell method called `say` — this method will be implemented later in this tutorial.

Notice that the class has obviously been implemented with a view to creating it as a singleton, so we need to actually construct it somewhere.

The personality script is usually the repository for global initialisation code and instances, so to create the chat console we need to modify the `init` method in `tutorial/res/scripts/client/BWPersonality.py`:

```
gChatConsole = None

def init( scriptConfig, engineConfig, prefs ):
   ...
   # Create the chat console and make global reference so we do not have to
 keep
   # re-acquiring it on each keypress
   global gChatConsole
   from Helpers import ChatConsole
   gChatConsole = ChatConsole.ChatConsole(
      scriptConfig.readInt( "chat/visibleLines" ) )
```

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

Note

You may be asking, "Why not just construct the ChatConsole object in the global scope, instead of making it None and then creating it later during init()?"
The problem with constructing it in the global scope is that code would be run before init, and you are not allowed to execute any custom code before init, as various sub-systems would not have been initialised yet, and trying to do so would therefore cause problems and probably crash the client.

We have also declared another script configuration option: chat/visibleLines. This option controls how many lines of output will be visible in the chat console at once, so declare a sensible value for it in tutorial/res/scripts_config.xml.

We also need to modify the personality script's key press handler to route input to the ChatConsole's key press handler:

```
# Keyboard event handler
import Keys
def handleKeyEvent( down, key, mods ):

   # If the chat console is in edit mode, let it handle all keypresses
   if gChatConsole.editing():
      return gChatConsole.handleKeyEvent( down, key, mods )

   # If the user hits the ENTER key, we enter chat mode
   if down and key == Keys.KEY_RETURN:
      gChatConsole.editing( True )
      return True

   # Otherwise fall through to the engine
   return False
```

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

This handler implements the behaviour expected from a typical in-game chat system. Pressing Enter makes the system enter the Edit mode, and whilst in it, all keyboard events go to the chat console.

Now that the GUI parts of the chat console functionality are in place, we will implement the entity methods to support routing the messages between the client and the server.

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>
      <!-- Client part of the chat implementation -->
      <say>
```

```
            <Arg> STRING </Arg> <!-- message -->
         </say>
      </ClientMethods>
      <CellMethods>
         <!-- Chat to people within 50 metres -->
         <say>
            <Exposed/>
            <Arg>            STRING </Arg>
            <DetailDistance> 50     </DetailDistance>
         </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 "Basic GUI text console" on page 20) 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:

```
from Helpers import ChatConsole

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

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

When the method is called, it simply gets the `ChatConsole` singleton and calls its `write` method.

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