CSCI1040 **HANDS-ON INTRODUCTION TO PYTHON**

Mini –Project

Game Tools Programming Due: Apr 19, 2014

Python is a robust scripting language that it is now used extensively in all sorts of applications. Gaming is one of the main stream computer applications nowadays, and thus Python is thus also used in some daily tools development. In this project, we will try a simple game tool development which will help in level building.

In this project, we will write a Python program which produce a game world which allows one to precisely control the movement of a camera within.

In this project, you have two paths to choose:

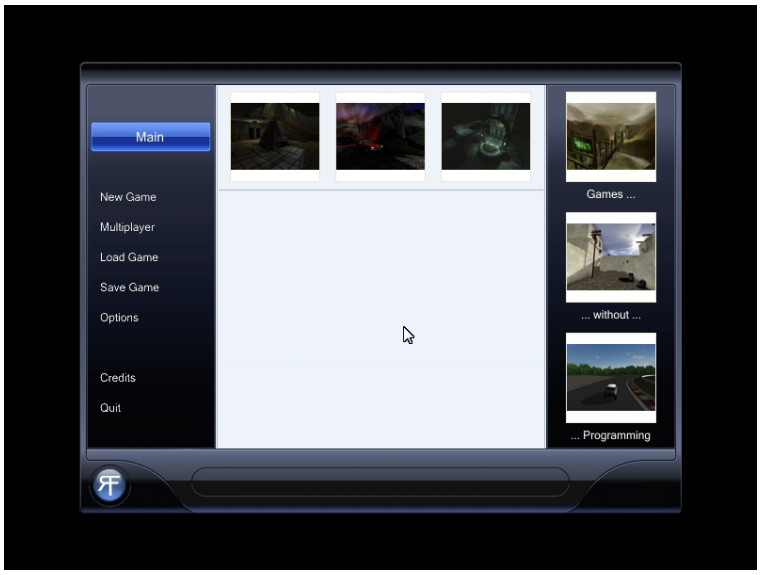
1. Download a game engine package and work on a level file provided. In this case, you are only needed to examine the level file and write the corresponding Python scripts.
2. Learn the working details of the level file and work out your own version of level file, and thus write the Python scripts.

Each way has its own merits and disadvantage. In general, path 2 is more time consuming but you can learn more know-how from it whereas path 1 would let you just focus on Python scripting.

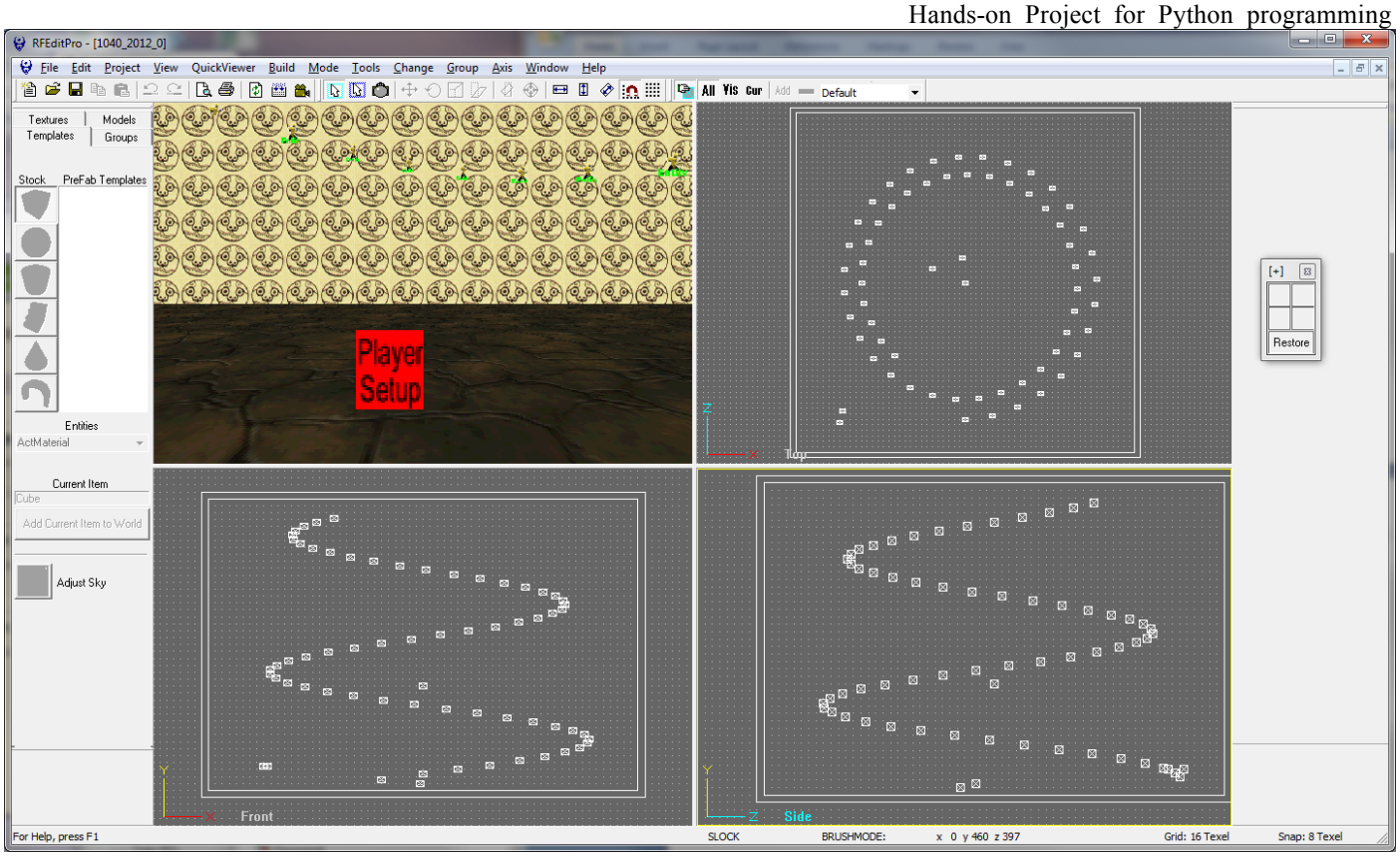
Below we will discuss the two paths in details.

1. Path 1 : Pure Python Scripting

Download the game engine package marked RF_Path1 and unzip it. To test the package, simply execute the "RealityFactory.exe" in the RealityFactory076 folder. If the following screen appears, then the game engine should run properly.

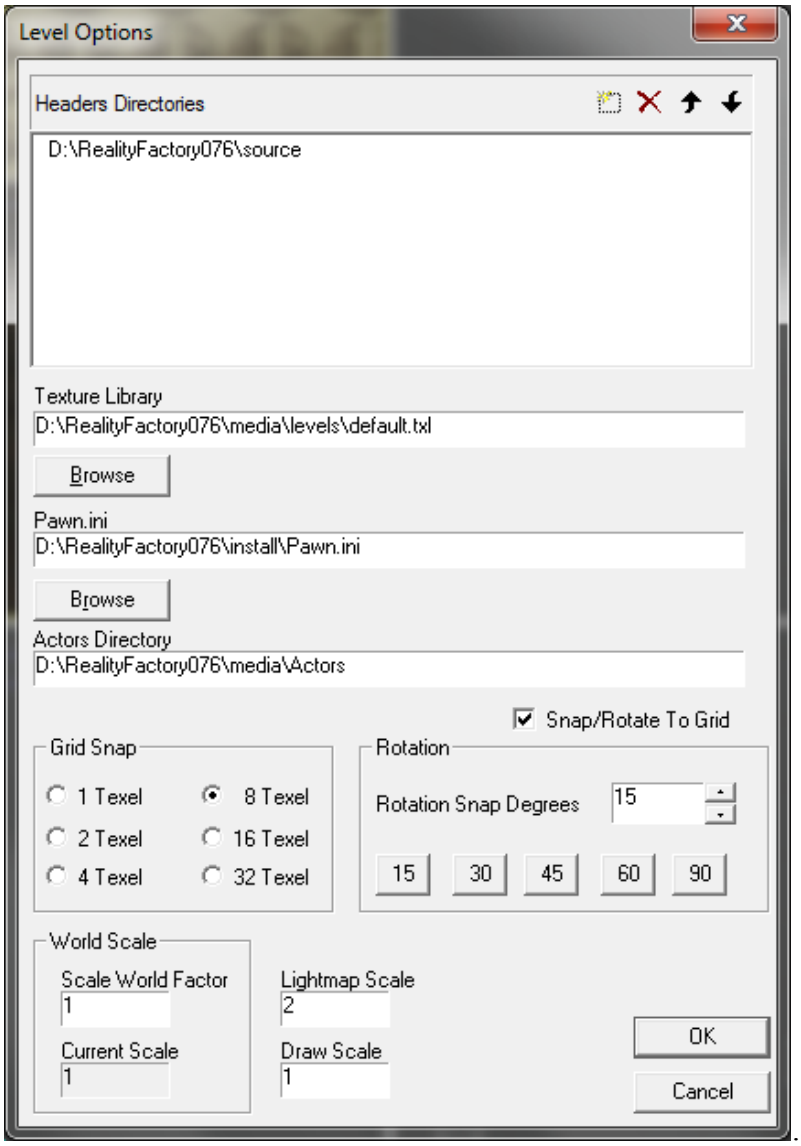


Now close the program by selecting "Quit" and go to the Tools folder under the RealityFactory076 directory. Execute the "RFEditPro.exe" to bring up the level editor.
The level editor is program to let an artist work on the game world. When the editor screen is up, click on the File menu and select the open option. Inside the opened dialog box, pick the "1040_2014_0.3dt" file, which is our pre-baked level.
The screen should be something like this:



After that , click on the “Project” menu, picking the “Level options”. A dialog box similar to that below should show.

Change the directories to that correspond to your installed path for ALL the paths e.g. “D:” change to “C:” if you install the program on C drive.



Now select the Build menu, pick the “Compile” option. Then just click OK to see the controlled camera effect.

In this level, we are using the camera control system to do a fly-through in the game world. The movement of the camera is being controlled precisely by making it to follow some prescribed points within the game world.

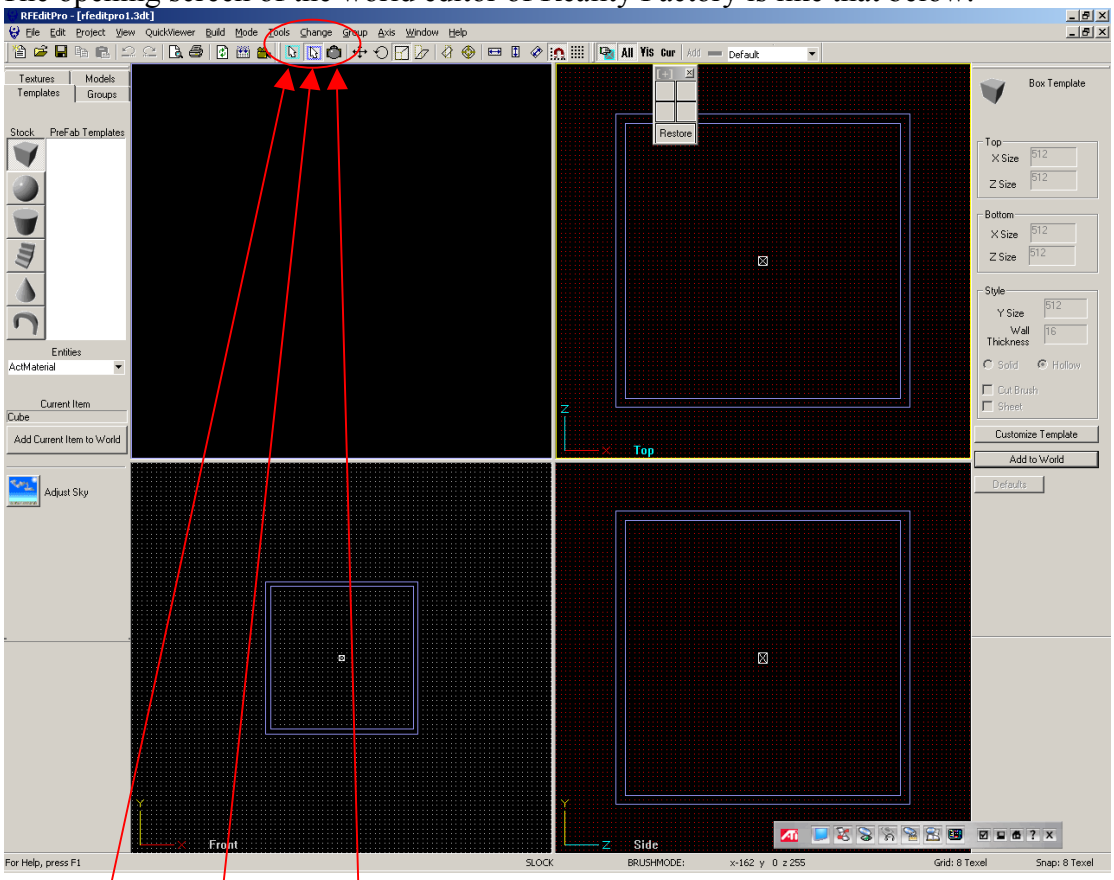
Now skip to section 3 for our Python scripting project.

2. Path 2 : Game Engine Drilling

First we need a software called game engine. Download and install the game engine (labeled "RF_EngineDrill" from our eLearning site and remember also download the patch. After installing the package and do the patching, follow the instructions below to have a taste of the game in running.

World Editor Usage

After installation, you should have a folder called RealityFactory being created in your hard disk, typically under C drive(depends on whether you have specify your own path during installation..
Bring up the world editor of RealityFactory(RF) package by going to the *tools* folder of RF installed directory (typically *C:\realityfactory*) and double click the *rfEditPro* program icon.
The opening screen of the world editor of Reality Factory is like that below:



There are three primary editing modes of the editor:
Selection, Template & Camera.

Camera/Viewing window: navigating inside the created level (top left sub-window) and adjusting the viewing portion in the three views(top, front & side – rest sub-windows)

To switch between these three modes, simply press the buttons as indicated above, or pressing the hot key (***C-camera, B - Selection, T-template***)

For each mode, there are some actions associated with it. We will cover them more in the following sections.

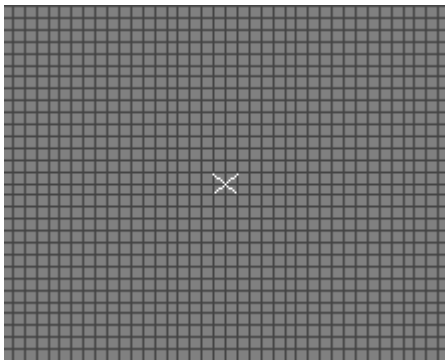
2.1 Basics of Modeling in Polygons

Polygonal models consist of three basic elements: the vertex, the edge, and the polygon.

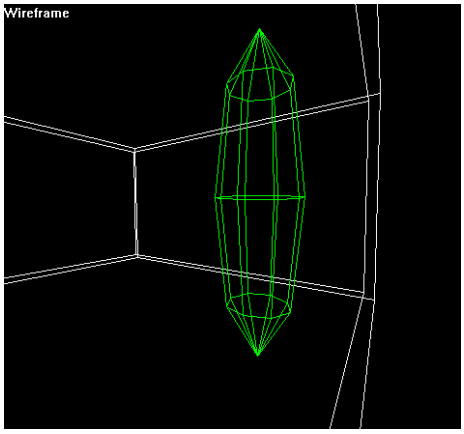
Vertices: a single point in space

Edges: defined by two vertices. When viewing an object in wireframe mode, we are seeing the edges.
Polygons: defined by three or more edges. Actually polygons are planes. Most modeling packages allow an polygon to have more than three sides, but these are generally tessellated to triangles(3-sides polygons) when being rendered to screen.

In 3D game modeling, the models have to be incredibly simple i.e. least number of polygons used in modeling.



Vertex



edges and polygons

2.2 Create a little world

To create a level using RF requires you to understand the concept of *Constructive Solid Geometry*. But first let's have an idea of the components of a level first. Basically a level inside a RF game(or most of the first person view game) consists of many little geometric entities called "*brushes*". A brush refers to some basic geometric primitives or their combinations such as *spheres, cube*, etc. Inside the world editor, we have only two kinds of brushes : *solid* brush and *cut* brush. They refer to the basic operations of CSG as indicated below.

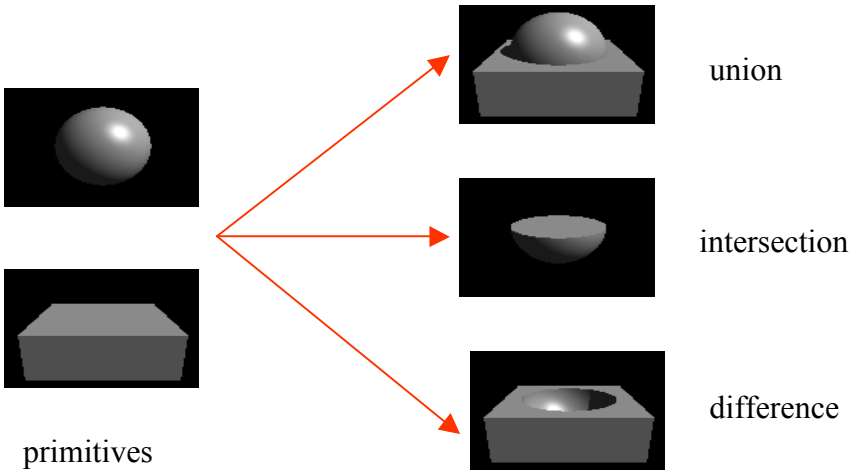


Fig. 1 Possible operations of constructive solid geometry

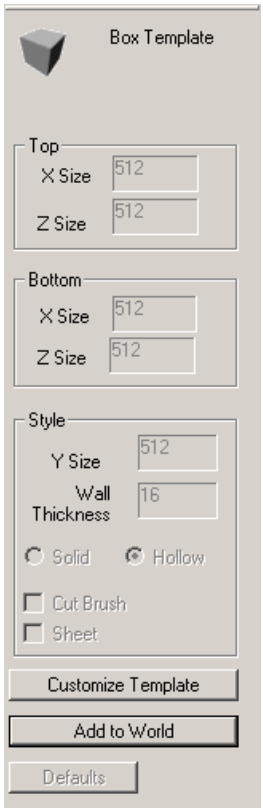
In the *union* operation, the properties of two brushes are solid brushes i.e. they add up together. In the *difference* operation, the sphere assumes the role of a cut brush and the box is a solid brush. (there is *no* correspondents in RF for the intersection operation AFAIK) In this sense, we define solid brush as a volume which has actual existence in a level whereas a cut brush is a tool we used to *carve* or *remove* some volume out from a solid brush. In the example below, we illustrate a simple example how to build a level which consists of a room.

Create a room

Change to template mode by pressing the template button (or pressing hot key ‘t’). Note the right hand side of the editor window now shows the template tab as below:

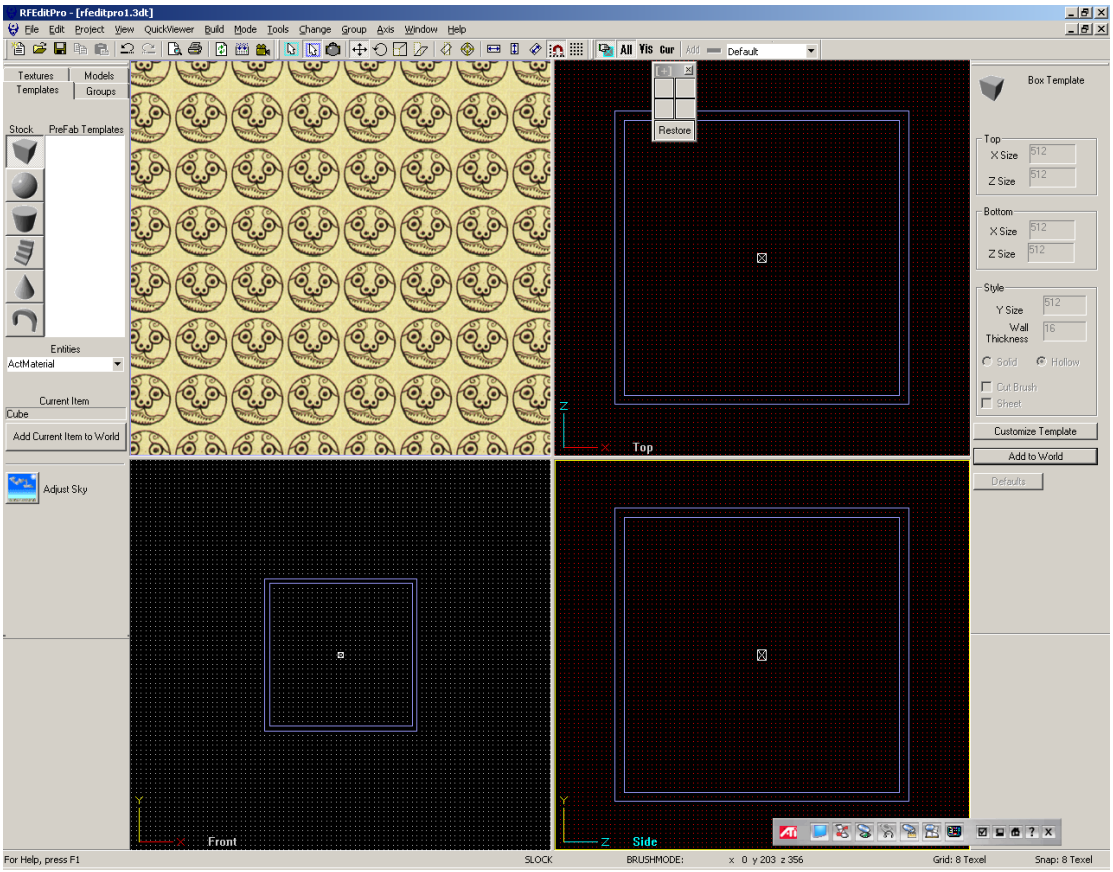
Now click on the box button as indicated.

The description for the box template will appear on the right hand side as below.



This is the configuration box for creating a brush in your world. In the coordinate system of RF,

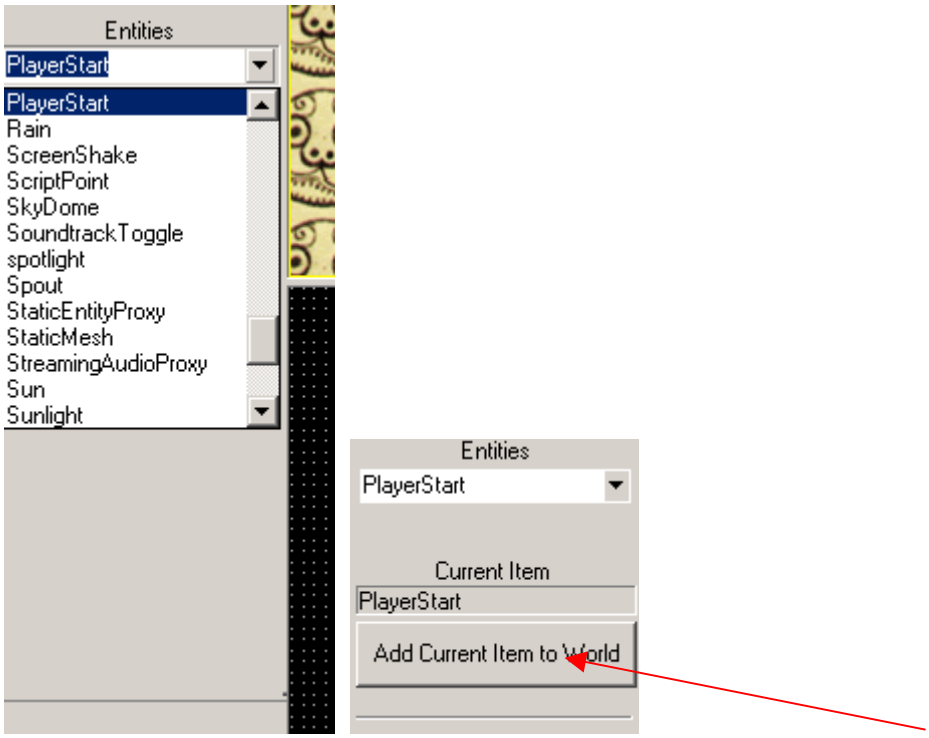
vertical direction is represented by the y-axis. The dialog box allows you to input values which define the size of the brush. You can change the values for the dimensions according to your requirement by pressing the “Customize Template” button and entering corresponding values. For our current purpose the default value is okay. Press the “Add to World” button and you should hear a “flip” sound and the screen should change as below:



Now a blue outline will show in all the three view windows which indicates that a brush has been placed into the level.

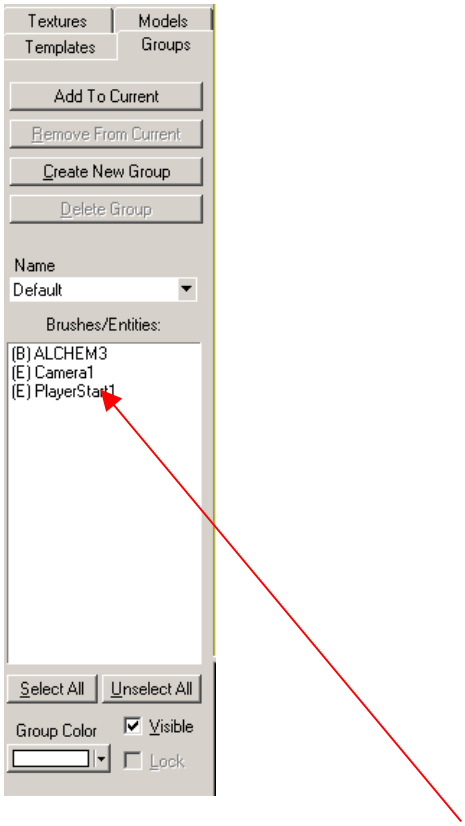
The created room will use default setting i.e. textures etc. for all your created brushes. Now we need to do an extra operation to enable walkthrough.

Change to template mode and click on the template tab window. Click on the drop down menu below the “Entities” and scroll to select the “PlayerStart” entity. This indicate to the engine the starting position of our player.

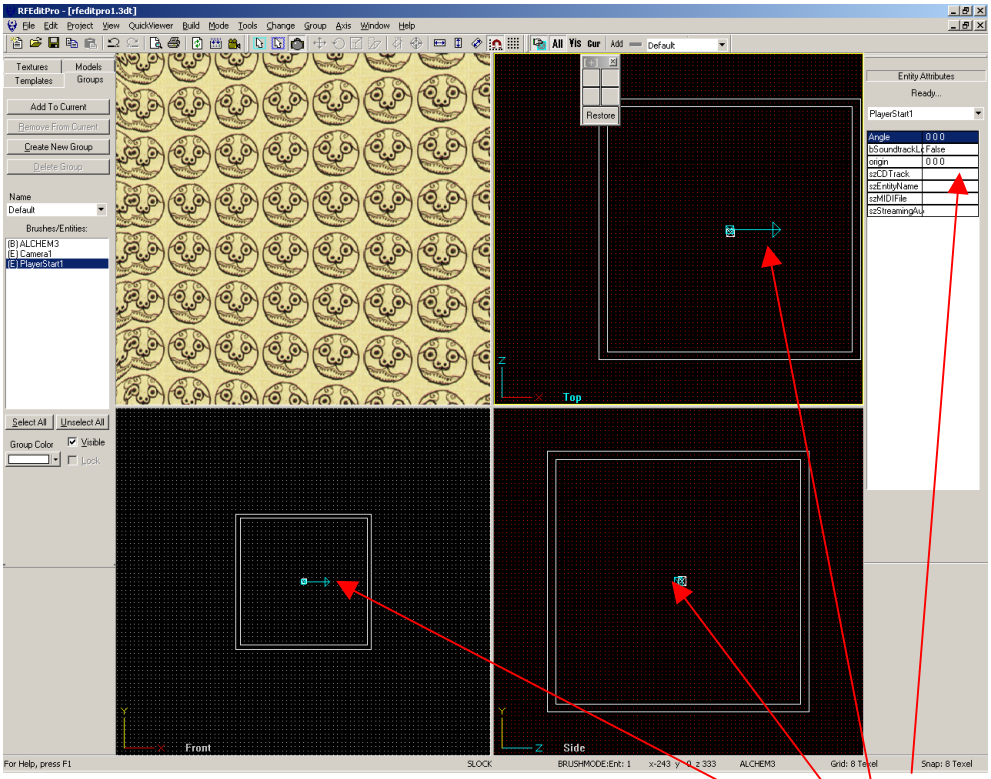


To place the player start position onto our created level, click on the “Add” button You should also hear a sound now, indicating that the entity Ihas been added to the level.

To view the player start position, click on the “Group” tab and the following will display:



This will show a list of entries and brush in our created world. Click on the “PlayerStart” and the three views would be similar to that below.

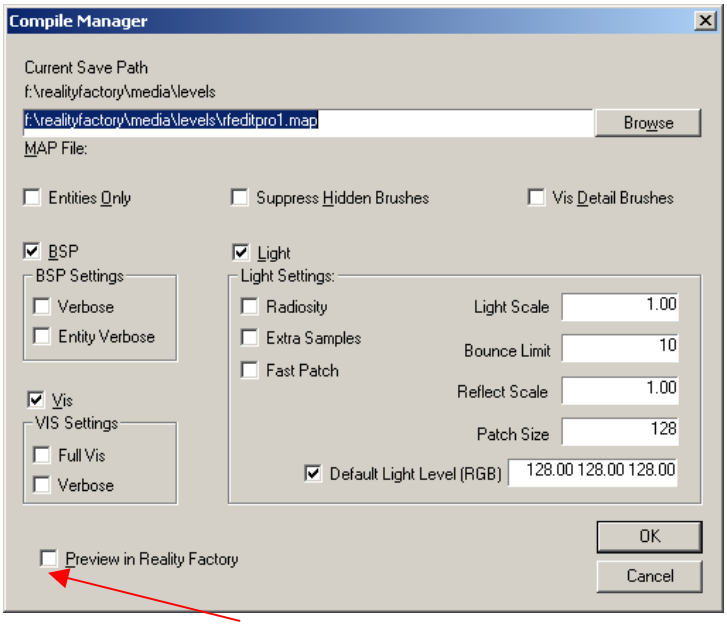


The information below the entry can be edited by clicking on the right hand table.

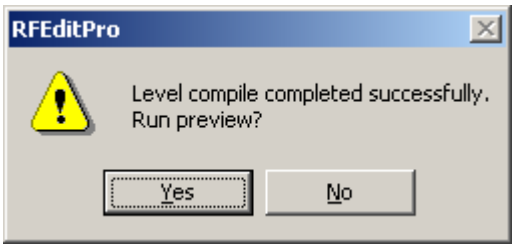
Now the four windows that show the level should change to make an arrow with light blue color inside the level, which means that you have selected that entity(PlayerStart).

Now **repeat** the same procedure to add also the entity called “PlayerSetup” to the level. Note that the newly added entity is also placed in the center (0,0,0).

To start the walkthrough around the level, simply compile the level to generate the required level file. Choose “Build\compile”” get the similar dialog box as follow:



Now check the Preview button. Press the "OK" button and the editor will perform some calculations and the following dialog box will be displayed:



That means the level is compiled successfully. Press "Y" or enter to confirm it.

1. The editor will bring us into the game world. The first picture of your created level would like this:

Move around the level by using the "W,S,A,D" keys (forward, backward, strafe left, strafe right respectively) and mouse to adjust view point.

2.3 Adding Enemies

In this section, we will learn how to add computer controlled characters with prescribed behavior to your game.

The addition of enemies (computer controlled AI in-game character) is performed in the following way:

0. You need to have already built an actor (character) and placed the actor file (with extension .act) into the "RealityFactory\media\actors" directory. Alternatively, you can choose an actor from among the files available inside that directory. For this example, we would use the robot actor, which is already placed inside the 'pawn' subdirectory of actors under the name "robot.act'.
1. Prepare a script file and placed it inside the "*RealityFactory\scripts*" directory.
   In Reality Factory, those computer controlled characters are collectively called 'pawn'. In order to let the Reality Factory system knows your added characters, you have to registering them by editing the "pawn.ini" file inside the "*RealityFactory\install*" directory. Following is the content of the "pawn.ini" file which registered your own character "Virgil" and your enemy "robot":

```
[Virgil]
actorname = Virgil_red.act
actorrotation = -90 180 0
actorscale = 1
fillcolor = 255 255 255
ambientcolor = 255 255 255
subjecttogravity = true
boundingboxanimation = Idle
shadowsize = 30

[robot]
actorname = robot.act
actorrotation = 0 90 0
actorscale = 1.4
fillcolor = 255 255 255
ambientcolor = 255 255 255
subjecttogravity = true
boundingboxanimation = idle
```

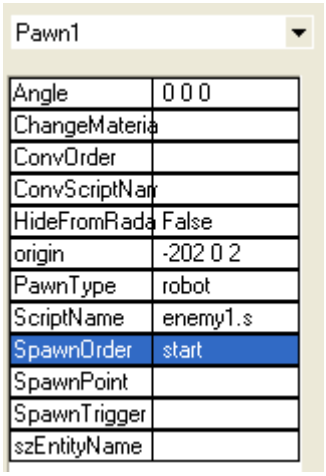
The parameters above are all self explanatory.

2. Now you may add any number of the newly added pawn to your level!

Start the RF editor and build a simple level with the necessary information (please refer to materials in previous RF sections).

Let's add the enemy to our level. Choose the template mode and scroll down the list and choose "Pawn" entity. Add this to your level and place it at some appropriate position, for example, just in front of the player.

Start the entity editor and choose the "pawn" entry you have just added. Change the entries of the pawn into the following:



i.e. you only need to change the Pawn type to “robot”, which is the type you’d registered in the pawn.ini file in step 2. Also you have to add script file “*enemy1.s*” to instruct the RF runtime to load the script file that we have just written.

The scripts in Rf is stored inside a file with suggested extension ‘.s’. So we will first created a text file called ‘*enemy1.s*’ and placed it inside the ‘scripts’ subdirectory of installed RF directory.

In RF, all the scripts inside a script file must be placed within a set of braces ‘{‘ & ‘}’ and each script file consists of collections of functions called ‘orders’. An order is designated by a pair of brackets ‘[‘ & ‘]’ with a C function like body in it. For example, the scripts below illustrate a pawn with an order called ‘start’.

```
{
start[ ()
{
PlayAnimation("idle", true, "");
RestartOrder();
} ]
}
```

Now compile and walkthrough the level, you should be able to see the following screen which show our newly added friend.

2.4 Make it move!

Movement of monsters is easy in RF – you need to specify where the monster should move and then make it move inside your own script file.
First we need to specify where to move. The destination is being specified by a path point called *ScriptPoint* in RF. *ScriptPoint* can be added in RF through the entity template.

Add a *ScriptPoint* and place it somewhere inside the level. The placement of *ScriptPoint* need not be exactly at the place you place the pawn because you can control the pawn to move to the *ScriptPoint*. A little hint is that the orientation of the ScriptPoint should not deviate too much from the stating point (you can also make it happen by adjusting the rotation speed of the pawn). To change the orientation of an entity, select the entity and press the Move/Rotate button, then drag with right mouse button pressed. Also the height of a script point for a pawn should be set at approximately the ground level to avoid strange orientation of the pawn when moving.

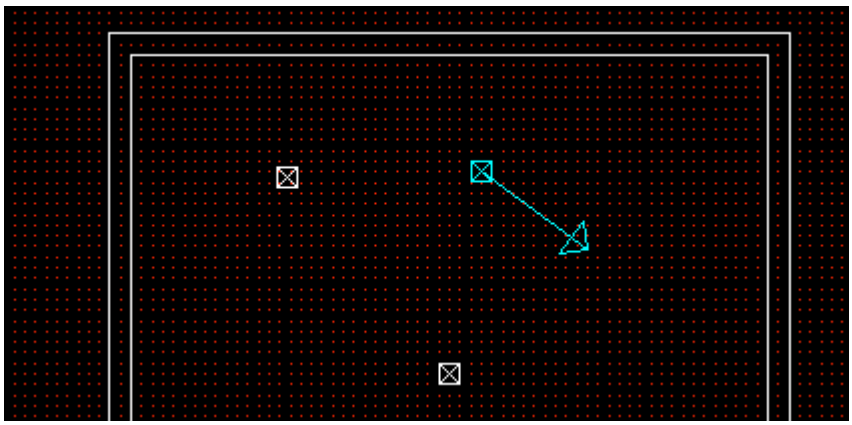
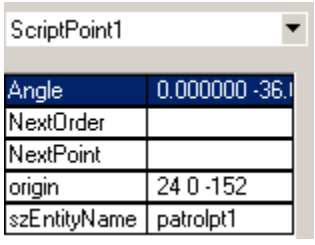
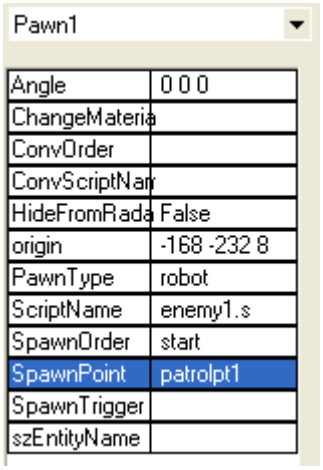


Fig. Rotating the ScriptPoint entity

Now give a name to the entity by adding it to the *szEntityName* entry inside entity editor as shown



Now select the pawn entity and set its *SpawnPoint* entry to the name of our added script point i.e. patrolpt1. Also you need to specify the *SpawnOrder* in this case. The edited entity should like this:



Finally in order to make the pawn move, we use the following functions:
RotateMoveToPoint(Animation, Rotation Speed, Speed, State, Sound);

The above function will rotate the pawn to face the ScriptPoint and move towards it at the same time. First parameter is the motion we choose pawn to perform. The second speed is the translational speed. The State parameter determine whether rotation about X axis be performed (at some instance where we want to twist of orientations).

After the alignment is being performed, the pawn can walk straight to the ScriptPoint by the following function:
MoveToPoint(Animation, Speed, Sound);

The edited script file content is as follow:

```
{
start[ ()
{
Console(true);
PlayAnimation("idle",  true, "");
Delay("idle", 3, "");
RotateMoveToPoint("walk", 100, 20, true, "footsteps\\m1.wav");
MoveToPoint("walk", 50, "footsteps\\m1.wav");
} ]
}
```

Note I add the call Console(true); statement. This will show on the run time screen what kind of action the pawn is doing (to help debugging). The Delay call is just a delay loop to let you see more clearly the transition of scripting statements. In actual situation it can be used to make more accurate timing or for any effect you can think of.

Now compile and run the level to see the pawn in action!

2.5 In-game movies

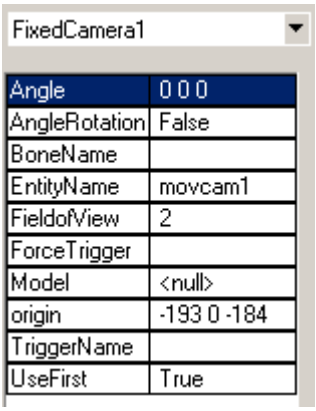
In a lot of 3D games recently, you might have seen that the cut-scene movies are being actually produced using the game engine itself. This came naturally as a result of better rendering quality of latest generation of game engines.

These in-game movies are actually a sequence of scripted events in the game engine's point of view. The most important point is that the camera can move on its own. To do this, we need to configure a camera pawn in the game engine. Firstly let us register a moving camera by adding the following lines in the *pawn.ini* file in *install* directory:

```
[MovCam]
actorname = projectile\proj.act
actorrotation = 0 180 0
actorscale = 1
fillcolor = 255 255 255
ambientcolor = 255 255 255
subjecttogravity = false
boundingboxanimation = nocollide
shadowsize = 0
```

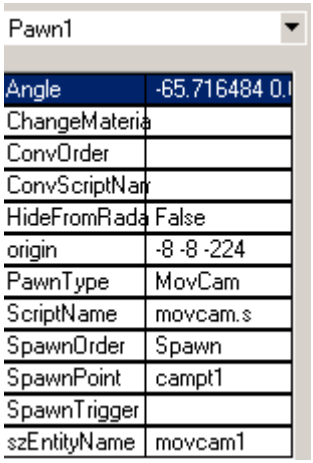
Then we follow the same procedure as in the last tutorial for adding a pawn to the created level, with the changes now being:

I. Create an entity of type FixedCamera and add to the level, and give it a name, say movcam1 and set the *UseFirst* attribute to true as below:



II. Set the *LevelViewPoint* attribute of the *PlayerSetup* entity to 3 to tell the game engine that we use fixed camera for this level.

III. Add a pawn entity and build the connection between it and the previous fixed camera as below:



III. Add a script point called *campt1* to the level so that when the camera will know where to be placed at when being spawned. Finally add a script file called movcam.s to the scripts directory to complete the procedure. You can be just place an empty script just as below will do:

```
{

    GROUP               [camera]    // name of monster group
    // spawn pawn and do setup work

    Spawn[ ()
    {
        Console(true);
    }]
   }
```

Compile the level to see the effect.

2.6 Script camera move

To make the camera move is easy – just script the motion just as monster move (check the section above for details). To do this, the script file can be changed as follow:

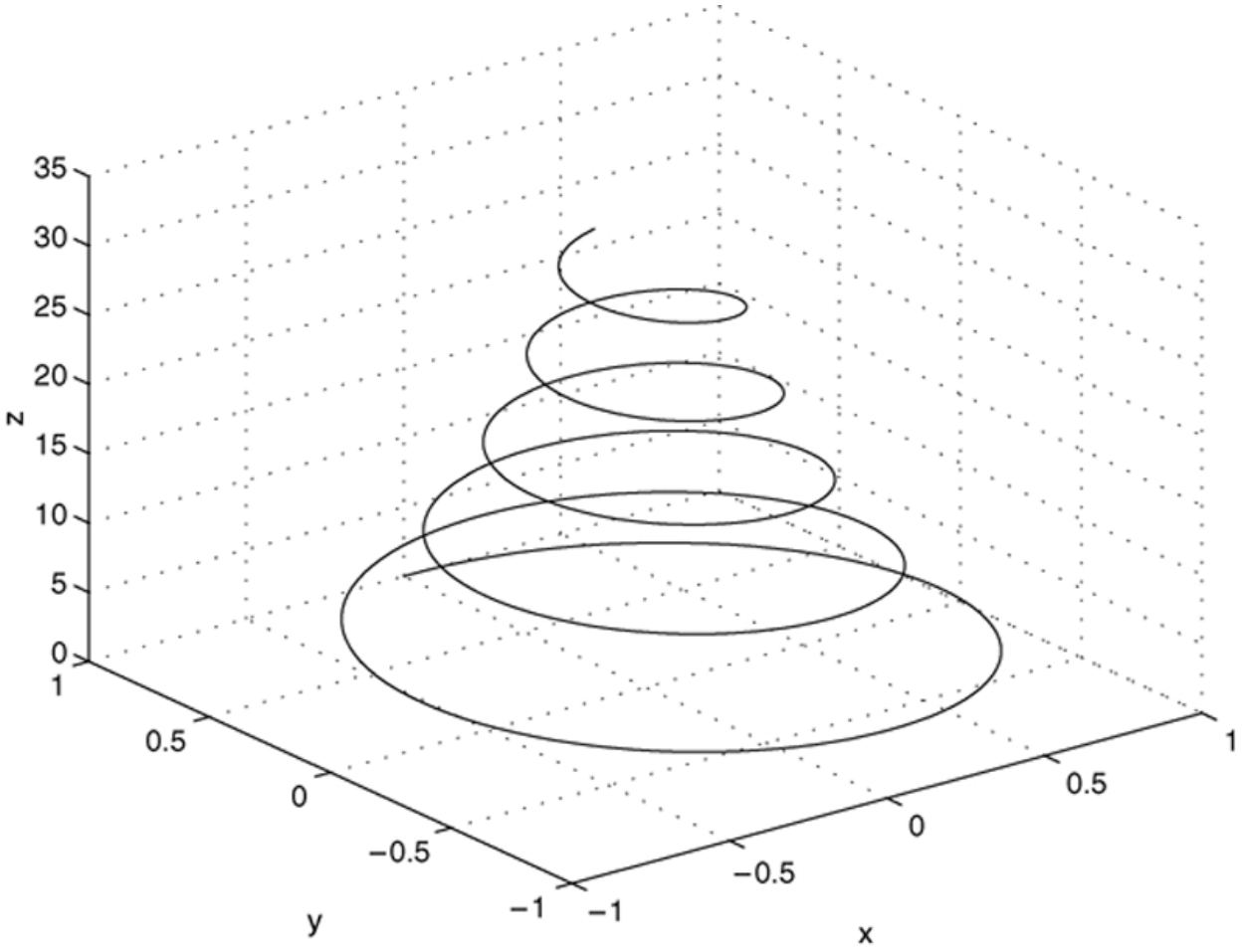
```
{
GROUP               [troop] // name of monster group
YAWSPEED            [40]        // speed of rotation in deg/sec
WALK                [Walk]      // walking animation
WALKSPEED           [100]       // average walking speed
// spawn pawn and do setup work

Spawn[ ()
{
    Console(true);
    RotateToPoint("Idle", YAWSPEED*5, true, ""); // go to point if any specified
    MoveToPoint(WALK, WALKSPEED, "");
    NewOrder("Patrol");
} ]
// walk the beat from point to point
Patrol[ ()
{
    NextPoint();
    RotateMoveToPoint(WALK, YAWSPEED, WALKSPEED, true, "");
    MoveToPoint(WALK, WALKSPEED, "");
    NewOrder("Patrol");
} ]
}
```

3. Python Scripting

Now you should have a taste of the game engine in working. Let's focus on our tasks in this project. Firstly for the level being packed together in Path 1 archive (for those who have tried path 2, you may also unpack the file "1040_2014_0.3DT" from the archive. Remember you need to have the MovCam pawn type and corresponding set up according to path 2 instruction intact so as to enable the script points in this level file) .

For the level file, the path the camera traced out is a spiral as below.



The general equation describing the path is given by

$x = e^{-0.05t} \sin t,$

$y = e^{-0.05t} \cos t,$

$z = t$

Actually the level file is a text file, so that you can open it with a text editor such as WordPad. Just right click on the file and choose WordPad or your favorite editor to open it.

For this file, there are several positions that you need to have special attention at. The first is at line 6

```
NumEntities 25
```

This is the line that tells the level editor how many entities this level has. And note that the ScriptPoints we used to guide the camera belongs to entity, thus the number of ScriptPoint we placed in the level will affect this number.

This number also appear in line 200

```
EntCount 25
```

So if you want to write a program to modify the level file, then you need to locate these two lines and change it.

The ScriptPoint starts at line 299 and ends at line 578, in which each ScriptPoint will be similar to that below

```
CEntity
eStyle 0
eOrigin -9 -186 400 2
eFlags 0
eGroup 0
ePairCount 7
Key Angle Value "0 0 0"
Key classname Value "ScriptPoint"
Key NextOrder Value ""
Key NextPoint Value "campt2"
Key origin Value "-9 -186 400"
Key szEntityName Value "campt1"
Key %name% Value "ScriptPoint1"
End CEntity
```

The lines are just a textual representation of those attributes showing in the editor. Those Key-Value pairs are those attributes which we change in the editor to script the level. Here we are interested in the fields eOrigin (exactly the same as origin Value), NextPoint, , szEntityName and %name.

The ScriptPoint is actually used here to control the movement of the camera in the level. By appropriately creating extra ScriptPoint using a program, we can precisely control the camera movement in the level, for example the spiral movement of the camera in the provided level, which is an almost impossible trajectory using human effort in real life. An additional advantage is that we can tune the trajectory by making changes on parameters of the above equation and view the result almost instantaneously.

Specification

Given the general spiral equation as below:

$$x = K_1 e^{-0.05 * K_2 * t} \sin(K_3 t),$$
$$y = K_1 e^{-0.05 * K_2 * t} \cos(K_3 t),$$
$$z = t * K_4 \qquad (*)$$

Write a Python program which when being run, will generate a world file for RF with the following properties:

1. The program will ask the user how many ScriptPoints he or she would like to generate. Then the program will ask user the four parameters K_1, K_2, K_3 & K_4 of the spiral.
   Prompt message:

```
How many ScriptPoints to generate?
K1?
K2?
K3?
K4?
```

   The values generating our sample level is `400, 0.1, 0.2 & 10` respectively.
2. The program will then automatically generate a text file with name '1040_2014.3dt', with the (suggested, you may choose your own, but the size should be at least this to allow easy experiment with different paths) room dimension x = 1024, z = 1024, y = 800. You are free to choose where this room is being placed i.e. the centre of the room can be or not coinciding with the origin of the world. Obviously picking the room origin same with world origin would make you easier to generate the path.
3. The generated room should have the number of ScriptPoints specified in step 1 according to the equation in (*). The placement of the ScritPoints should be centered in the level, and all ScriptPoints must be within the level to allow a camera walkthrough using the engine. Note that the movement of the camera can be controlled further by making changes to the YAW & WALK speed in the script file. Sometimes you may need to adjust them so as to make the camera correctly traversed all ScirptPoints.
4. The generated ScriptPoints should trace out a clear spiral path. There is no restriction on where should the camera go after the last point. You may simply set it to a non-existed point, or loop back to first point at your own will. Note that sometimes it may not be possible to make the camera returning to first point due to camera orientation and rotation issues.
5. The output file, should be able to read by the RF world editor, and allow successful camera walkthrough after compilation i.e. player Setup and PlayerStart should be written inside the text file.

Submit your Python program into the submission slot under the course eLearning account.