3D Practical 3 – First Person Shooter

In this practical we are going to complete a first person shooter game based on FPShooter3D from chapter 24 of Davison's "Killer Game Programming". You will learn how to use 2D images to create simple effects and how to run short animations in separate threads. The end product will be a working, but simple game.

FPShooter3D

Copy the folder **'K:\CSC9N6\Practicals\FPShooter3D'** (or from \\wsv\CSCU9N6\code\3D\) to a sensible folder on your 'H' drive. Now create an Eclipse or BlueJ project in your copy of this folder, add all the files in the folder to your project and set the file **FPShooter3D.java** as the 'main' file. Note that this application makes use of a utility model loader class contained in **portfolio.jar** in this folder, so this must be accessible in your compilation and runtime Java classpath:

- If you are using BlueJ, then add this file to your project via the Tools->Preferences->Libraries menu
- If you are using Eclipse then add this file via the Libraries tab when creating your project from existing source, or later by changing your Build path – please ask if you are unsure about how to do this.)

Tour of the code

Build and run the initial application. Instead of a sphere, now there is a robot figure standing in the middle of the checkered floor. If you play with the arrow keys you will discover that you can move the user's viewpoint around. But that is all you can do at the moment. What we are going to do is add a gun to the user's viewpoint so that you can fire at the robot, and an explosion will occur if you hit it!

> Note that a finished version is available in **'K:\CSC9N6\Practicals\FPShooter3D-fin'** if you want to see what it will do.

Firstly, you need to take a look at how the application is structured and some of the existing code.

1. Most of the work in creating this game world is in WrapFPShooter3D.java. So open this file in your IDE so you can look at the code. The basic scene is by now quite familiar to you and is constructed, as before, by the createSceneGraph() method. The one major difference is that instead of a sphere, the code reads a complex model specification (a robot) from a file, creating the appropriate geometry and appearance and adding this model to the scene graph. All this is handled by the PropManager class. We will not look at this in detail now, but you may like to examine the PropManager code in your own time. For our purposes, PropManager creates a top level transform group which controls the position of the robot, and that is all that the game code needs to know about.
2. The other new thing done in createSceneGraph() is to initialise user keyboard controls, which is carried out by a call to the initUserControls() method, which is passed a vector specifying the location of the robot. Take a look at the initUserControls() method. It does a number of things: (i) adds a 2D image of a

gun to the view platform, (ii) sets up ammunition for the gun (AmmoManager class) and (iii) adds a keyboard behaviour (custom KeyBehavior class).

3. Now take a look at KeyBehavior.java. This is a slightly more complex keyboard behaviour than the one you constructed in the previous practical. It is also different in that it extends the ViewPlatformBehavior class, because the key controls are to allow movement of the viewer position, rather than changing the position of an object in the scene. This is done by altering the transform in the transform group targetTG of the ViewPlatformBehavior class. This KeyBehavior class is complete and we will not modify it. For now, just have a look through the code and note the different key controls that have been implemented (you will need to know them so you can play the game!)
4. The code for setting up the gun and the ammunition however, is not complete. The rest of this practical will take care of this.

Adding the gun

Instead of a complex 3D model of a gun, we will add a 2D image of a hand holding a gun to the view platform, so that it looks like you (the user) are holding the gun, ready to fire.

5. Go back to WrapFPShooter3D.java and find the gunHand() method. This is called by initUserControls() to create the gun, but the current code is incomplete. What we need to do is create a Shape3D object that contains a simple square geometry which is textured with the gun image. This Shape3D object is then set as a child of the PlatformGeometry object, pg. Note that a PlatformGeometry is actually a transform group added to the view platform, so that the visible object it contains moves along with the viewer position. So do the following steps to achieve this…
6. Using the recipe given in the lecture on textures, add code to gunHand() to create a QuadArray with both vertex and texture coordinates. Geometry coordinates that will place the QuadArray nicely just in front of the viewer position are: (-0.1, -0.3, -0.7), (0.1, -0.3, -0.7), (0.1, -0.1, -0.7), (-0.1, -0.1, -0.7) Texture coordinates should map the four corners of the image to these vertices. Trial-and-error will quickly lead you to the correct ordering (because if you get it wrong the gun will be sideways or upside down! But you will need to complete the next few steps before you can check this.)
7. Create a Shape3D object and add the QuadArray to it using its setGeometry() method.
8. Again using the simple recipe provided in the texture lecture, load the gun image using a TextureLoader object and then set the texture into an Appearance object using its setTexture() method. Note that the file name is available in the GUN_PIC variable.
9. Finally, add the appearance to your Shape3D object using its setAppearance() method. Then add your Shape3D object as a child of the PlatformGeometry, pg.

10. Now you can compile and run the application to see if you have managed to correctly create the gun hand. You might have to revisit the texture coordinates if things do not look quite right!
11. One thing you will notice is the nasty black background around the gun hand. This is because by default the texture colours are being used exclusively as the pixels colours where the texture appears. The way to fix this is to ask for texture colours to be blended with other colours (resulting in black being replaced by other colours.) For this we need a TextureAttributes object. Add the following code before the line of code that adds the appearance to the Shape3D, replacing “app” with whatever you have called your appearance object:

```
TransparencyAttributes tra = new TransparencyAttributes();
tra.setTransparencyMode( TransparencyAttributes.BLENDED );
app.setTransparencyAttributes( tra );
```

Creating ammunition

Now we need to create something for the gun to fire! Ammunition for the gun to fire is set up in the AmmoManager class. The ammunition are instances of the LaserShot class. The actual firing of a bullet (or laser) is handled in a separate thread by the AnimBeam class. We will examine the code in these three classes to see how it works, and make changes to LaserShot to actually create a piece of ammunition.

12. Firstly open AmmoManager.java in your editor, and take a look at the code. The constructor firstly loads a series of 2D images for the “explosion” if you manage to hit the robot with a shot. It then creates a pool of LaserShot objects, which is our ammunition. These objects are all added to the top-level branch group of the content scene graph, sceneBG. The fireBeam() method is called by the KeyBehavior when you type the “f” key. It checks through the available ammunition to find a bullet that is not currently in flight, which is then fired. The loadImages() method uses a TextureLoader to load the series of images for the explosion, which is stored as an array of ImageComponent2D objects. This will be used by the LaserShot ammunition. All this code is complete, so we do not need to make any changes here.
13. Now open LaserShot.java and the fun begins! Take a look at the comments at the top of this file to see details of the scene graph that we need to construct to create a LaserShot. It consists of 2 transform groups (for placing ammunition in the gun and for firing it at the robot), a Switch (for making ammunition visible when required), and the laser beam (or bullet) and explosion visible objects. This scene graph is created in the makeBeam() method, which currently is incomplete.
14. The opening code in makeBeam() creates the first transform group that orients and places the ammunition in the gun. The ammunition itself is a thin, red cylinder. The Cylinder object “beam” has already been declared globally. Add code to makeBeam() to construct “beam” as a cylinder 0.05 in diameter and 0.5 in length, with a red appearance (as provided by a call to makeRedApp()).

15. Following your beam construction, add this peculiar line of code, which will enable the location of the beam to be read in global scene coordinates for calculation as to whether the beam hits the robot or not:

    ```
    beam.setCapability(Node.ALLOW_LOCAL_TO_VWORLD_READ);
    ```

16. The class ImageCsSeries has been provided to display a sequence of 2D images as an animation on a QuadArray geometry (essentially the same as sprite animation in 2D). You might like to examine the code in ImageCsSeries.java. In makeBeam(), construct the explosion object, explShape (already declared globally), which is of type ImageCsSeries. The explosion should be centred at 0 and have a size of 2.0 (see the ImageCsSeries constructor.) The image sequence is in the array, exploIms.
17. Global variables for the Switch object (beamSW) and the transform group for firing the beam (beamTG) have also been declared. You now need to add code to makeBeam() to construct these objects. Using a basic constructor (no parameters required), construct beamSW and then give it Switch.ALLOW_SWITCH_WRITE capabilities. The transform group beamTG has been constructed, but you need to give it both read and write capabilities.
18. Add beam as a child of beamDir and then add beamDir and explShape as children of beamSW. Make the beam and the explosion initially invisible by calling the setWhichChild() method of beamSW with the Switch.CHILD_NONE parameter value. Finally, add beamSW as a child to beamTG.
19. Compile and run your code, which should now be complete. Move around using the keyboard controls and fire the gun until you manage to hit the robot!

Firing the ammunition

What remains is to examine the code that is responsible for animating the firing of a LaserShot. You will discover that instead of using a time-based behaviour, the animation is carried out using threads, in the same way that such animations are done in 2D. Behaviours could be used, but it is sometimes convenient to simply use a separate thread.

20. Firstly look at the “shot” methods provided in LaserShot. The first of these is requestFiring() which actually initiates the animation of a laser beam by starting an AnimBeam thread if this laser beam is not already firing. This method is called by the AmmoManager fireBeam() method in response to the user pressing the “f” (fire) key.
21. A quick look at AnimBeam.java shows that all AnimBeam does when it is asked to run is call the LaserShot moveBeam() method.
22. moveBeam() does all the hard work of actually moving the cylinder that is the laser beam towards the target. The algorithm moves the cylinder along the increment vector (which in this case is directly away from the viewer along the z axis) until it has either travelled a predefined distance or has hit the robot. If it hits the robot then the explosion is initiated.

3D Practical 3 – First Person Shooter

23. Movement is carried out by the doMove() method, which is essentially the same as the basic method used in the previous practical.
24. closeToTarget() has the hard job of calculating whether the beam has hit the target (robot). The location of the robot was passed to LaserShot in its constructor (targetVec). Here the beam's current location is converted into world 3D coordinates and the distance of the beam to the targetVec is calculated.
25. The remaining two methods, showBeam() and showExplosion(), use the Switch object to make the beam and the explosion either visible or invisible, as required.

If you have completed the required code and thoroughly examined the provided code, then you should have a good understanding of how to construct an animated 3D world involving user interaction and transient events, such as the firing of a gun. Much of the code here can be used directly or adapted to create much more complex applications.