COMP3811: Computer Graphics
Coursework 2 Report
Interactive Animated Scenes within OpenGL

Ben Hulse

Project Overview

The program I created, when run, puts the user in a small, dark room in the first person perspective. The user can move around the room with the WASD keys and can look around by holding left click on the window and moving the mouse. This gives the user standard and intuitive first person controls, that is used in almost every first person game.
Three of the walls are made of brick, and the fourth is a map of the world. The floor and ceiling are a wooden texture. In the middle of the room is a table on top of which is a Robot with two axis of movement, the base can spin and the arm can move up and down. This robots movement is automatically spinning, but can be toggled to user control by pressing N, at which point the robot is controlled with the arrow keys.
The room is being lit by a flickering torch being held by the robot. Because the user can control the robot they can move the torch to light up different areas of the room, and inspect different areas of the map.

Project Breakdown

The Lighting and Materials

For the scene lighting I am using the default GL_LIGHTING from OpenGL, and Phong shading. For the materials I first created a struct to store the four components of a material, so that I could easily store and use the material properties.
Within the main CW2Widget class I also predefined some const materials, the values of which that I got from http://www.it.hiof.no/~borres/j3d/explain/light/p-materials.html and changed to better suit the scene.

When I draw something with a texture, I will use the matte white material so that the texture isn't changed by the material underneath.

For the torch light, I wanted to have the light be flickering, as if it was a flame torch. To do this I store the light properties as a material struct, and also store a modifier, that will be multiplied by all of the elements of the material to change the intensity; this way over time the intensity modifier can be changed randomly and it will be reflected in the intensity of the light, which creates the flicker effect. I also set the light's attenuation to quadratic with a factor of 2, this was to make the light drop off more severe so areas further from the torch are darker.

The Drawables - Room

To create the room, first I defined a function to draw a single 2D plane in 3D space, between given coordinates, aligned with the coordinate plane (x, y or z) also given.

```
void plane(GLfloat corners[2][2], int plane, GLfloat depth, GLfloat
resolution);
```

Plane can have 3 possible values (0, 1, 2) each standing for the x, y and z axis respectively. *Resolution* can be any whole number > 0, and relates to how many quads the plane will be drawn as.

This function would then draw the 2D rectangle on the plane: *plane* = *depth*. As an example if *plane* = 0 and *depth* = 5, then the rectangle would be drawn on x = 5. With the coordinates for the other two dimensions being given in the array *corners.*

When the function draws the plane in 3D space, instead of drawing one big plane, it tessellates the plane into lots of smaller quads. The higher the value of *resolution* the more, quads are drawn. This is because the default OpenGL lighting is a vertex shader, so by adding more vertices to the shape, the lighting is smoother over the plane. This was done by interpolating over the coordinates of the wall in two dimensions in a nested for loop, and drawing quads using these interpolated values.

From this function I created a second function to draw the plane with a texture on it.

```
void texturedPlane(GLfloat corners[2][2], int plane, GLfloat depth,
GLfloat resolution, int tex_i);
```

The function is almost identical to the original plane function, however when it defined vertices at the interpolated world coordinates, it also defines texture coordinates for that vertex that are interpolated. Essentially each small quad will have a small part of the texture on it. The texture that is put on the plane is the texture stored in the index *tex_i* in the textures array, which is an attribute of the widget class.

To create the walls of the room I called the texturedPlane() function for each wall, each time giving the *tex_i* value of the texture needed. Three fo them are drawn with the brick texture, and the fourth is drawn using the “earth.ppm” texture.

The “brick.ppm” and “wood.ppm” textures were from http://www.cadhatch.com/ and are royalty free under the license described here: http://www.cadhatch.com/terms-of-use/4588167704.

The Drawables - Table

The table was created using the aforementioned plane() function with a light brown material. I used a set of functions for the different parts of the table, for example tableLeg() would create a table leg at (0,0,0). This function is used by the main table() function by first translating to where the leg should be, then calling the tableLeg() function.

The Drawables - Robot

To draw the robot I used hierarchical modeling, wherein each part of the robot is responsible for calling the next part of the robot, so that the positioning of each part is reliant on the previous. This means that moving the arm of the robot, will also move the torch at the end of the arm, which allows for the two axis, user controlled movement.

Parts of the robot are drawn using gluCylinder and gluDisk, which requires quadric objects to be defined as a member of the widget for each of these shapes. Before drawing the base, glRotatef() is used so that the robot is rotated by the amount stored in the robotAngle[0] variable. By not undoing this rotation until after the robot is finished being drawn, the entire robot is turned but his amount. Parts of the base are also made out of planes, both textured and with a shiny red material. The side of the robot base is where the “Moi.ppm” texture is used. The function drawing the robot base will then call a function to draw the arm, and that will then call a function drawing the torch. Because I wanted the light source to be where the torch is, as well as having it draw the torch, I had the robotTorch() function place the light in the scene. This means that when the user moves the robot, which in turn moves the torch, they are also moving the light source for the scene.

Because the quadric objects are dynamically allocated they are also freed in the destructor of the widget to prevent memory leaks.

User Control - Camera

One element of user interactivity I decided to incorporate was an intuitive first person movement system. By doing this it allows the user to walk around the room using the WASD keys and to look around by holding left click on the window and moving the mouse.

To handle user input I used the event's that are a part of the QWidget class. By having a function to handle key press, key release, mouse press, mouse release and mouse move I can properly process all of the inputs the user can make. For the key press handling I find the key that is being pressed, then check if it needs to be handles immediately (to quit, open a menu or toggle the robot movement) and do so, then finally I store all other key presses in a hash table, where the key is the key in question and the value is whether or not it is being pressed. I also implemented the key release event handler so that when any key is released, it's value in the hash table is set to false. By doing this I now have a hash table that can act as a look-up for any key, and say whether or not it is being pressed/held at any given time.

I did a similar thing for mouse presses, where I have a bool to store whether the mouse is currently being pressed, that is set to true by the mouse press event handler, and false by the mouse release event handler. The final event handler was the mouse move handler, this is used to rotate the camera, however it only does so if the mouse button is currently being held.

To implement the movement I defined an array of floats, that would store the current position of the camera, and an array of float to store the angle of the camera. Its worth noting only Pitch and yaw are stored as roll is not needed, this avoids any gimble locking. On every call to update (which is done 60 times per second) I call a function 'handleKeys()', which checks if any of the WASD keys are being pressed; if they are, I use trigonometry and the angle of the camera to change the camera position x and z values to move the camera in the correct direction. Because the camera stays at the same level, the movement is only reliant on the yaw of the camera which means the trigonometry is relatively simple. When the camera is placed using gluLookAt(), the camera is placed at the coordinates stored in the camera position array. I also had to make sure that the x and z of the camera position stay between the walls of the room, because the room size is fixed I did this with simple hard coded if checks.

To implement the camera turning first I had to hide the mouse when the window is being clicked on, I did this using Qt's QCursor class. Once the mouse was hidden I could put it in the center of the window. Now that the mouse cursor is in the center of the window, in the handler for the mouse move event I could calculate how far from the center the mouse has moved, change the pitch and yaw stored for the camera proportionally to how far the cursor has moved, and then reset the cursor to the middle of the window. To translate the desired pitch and yaw of the camera to actual camera positioning I had to again use trigonometry with the camera position and angles, to calculate a point in space that is in the direction I want the camera to face, relative to its position. Now that I have this point I can tell the camera to look at at in the gluLookAt() call, meaning the camera is now in a custom position and facing a custom direction.

User Control - Robot

Because the scene is so dark, the map on the wall is hard to see, this is why I implemented the ability to spin the robot, both automatically and manually.

By default the robot is animates to spin around, therefore moving the torch around the room continually and lighting up different areas. However if the user presses the N key the robot becomes user controllable using the arrow keys, this allows them to rotate the base and move the arm up and down, that way they can illuminate specific parts of the map to loo at.

The automatic spinning is simply done by adding one to the angle every update and using modulo to keep it between 0 and 360. And it is toggled by the key press event handler.

The manual movement is handled in the ‘handleKeys()’ function after the camera movement. It is just a simple check for each key, and change the robotAngle variables accordingly.

The actual movement of the robot is handled by rotating the base and arm before placing drawing them. Because of this the arm is also moved by the base rotating as they are hierarchically modeled.

User Control - Torch Light

The final piece of user control was the ability top change the properties of the light emitted by the torch. This is done by a separate widget, CW2MaterialWindow, that uses standard QtEditLine objects to take a set of user inputs as text.

After creating the widget for taking the input, it is initialized by the main widget with no parent, that was it acts as it’s own window. The main widget then connects the signal given off by the material widget when the ‘done’ button to its own updateTorch() slot, which receives the values to set the torch to. This way when the ‘done’ button is pressed on the material window, the values that are input are sent back to the main widget and the torch material struct is updated to have those values. The window can be shown by pressing the M key, when it is shown it is passed the current torch material values so that they are already filled in when the window appears.

With this the user is able to change the colour, intensity and light properties of the torch.