Summary and Inception

This scene is based on the allegory of Plato's cave. The allegory tells the story of prisoners who lived their entire life chained to the wall of a cave, force to look at the opposing wall. Behind the prisoners sat a large fire that projected the shadows of the various objects that passed in front of it onto the opposing wall. Given that the prisoner had spent the entirety of their life in the cave, the prisoners incorrectly mistook the projected shadows for the real objects that they represented and began naming the shadows. One day a prisoner is freed from his restraints and is able to move around the room. They see a fire which blinds them, making it hard for them to perceive the actual objects. Slowly their eyes begin to readjust, and they leave the cave. Once again, they are blinded by the sunlight but slowly their eyes begin to once again readjust. They see the real world filled with actual objects. The prisoner gains a new understanding of reality and the realisation shattered his prior notions causing a paradigm shift. Upon returning to the cave the freed prisoner is no longer able to communicate effectively with the chained prisoners due to each of them holding such different conceptions of reality. The chained prisoners then perceive the freed prisoner deranged from his journey.

The allegory isn't supposed to be taken at face value presents an argument that reality cannot easily be determined via the five senses alone. It also serves to highlight the how individual can become trapped by their ignorance and the nature of truth.

The reason why I decided to recreate this story for this coursework is primarily because its emphasise on shadows. The basic structure of the scene is also well documented in the text and through various illustrations of it throughout time. There is a distinct inside (the cave) and outside (the world outside of the cave), this meant that I would be able to implement different types of lighting and experiment with terrain generation. It also allowed me to imbue my passion for philosophy with my computer science course. The usage of the computer model within the application, as well as the program qua a program also calls attention to the nature of reality via more model philosophical ideas such as simulation theory. To elaborate, the cave is connected to the exit through a tunnel. A computer terminal sits in the tunnel that displays the outside world. The prisoner would be tempted to view the terminal screen as a new reality after entering the tunnel. Once the prisoner has left tunnel they are then confronted with the real world. In fact, all of this takes place within a simulation (the program). The program, existing in the real world, is a false reality with several other false realities embedded in it. This therefore begs the question 'is this reality real?'

Explanation of the design choices

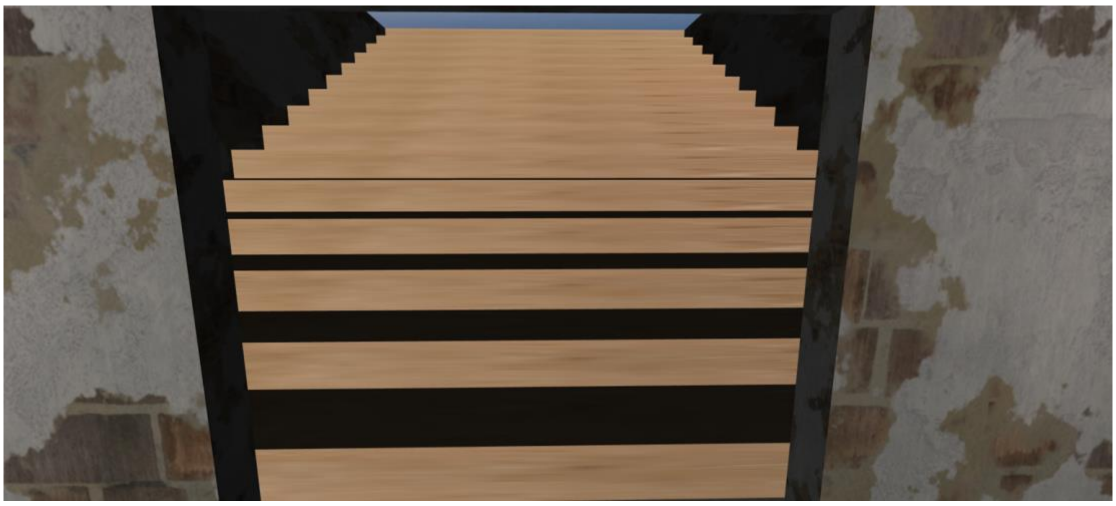
The project is implemented using Qt and OpenGL. Qt is used to facilitate the User Interface. A window is created that has various QWidgets as children. A new class was created named SceneWidget, that inherits QWidget, this is where the bulk of the program is implemented and where OpenGL is used to render the scene. To further break down and encapsulate the code, the SceneWidget contains a pointer to a "ShapeCreator" object. The ShapeCreator, in turn, has a pointer to a "TextureCreator" object. This structure allows for classes to access other variables and methods via it's 'children'. For example, if an index of a texture needed to be used by the SceneWidget widget it can be accessed using sceneWidget→shapeCreator→textureCreator->index.

Coding Style and Commenting

The code has been broken down into four main classes. The program has used ideas from object oriented programming and event driven programming. Commenting has been used throughout the entire project so I, or the marker, can follow the flow of the program.

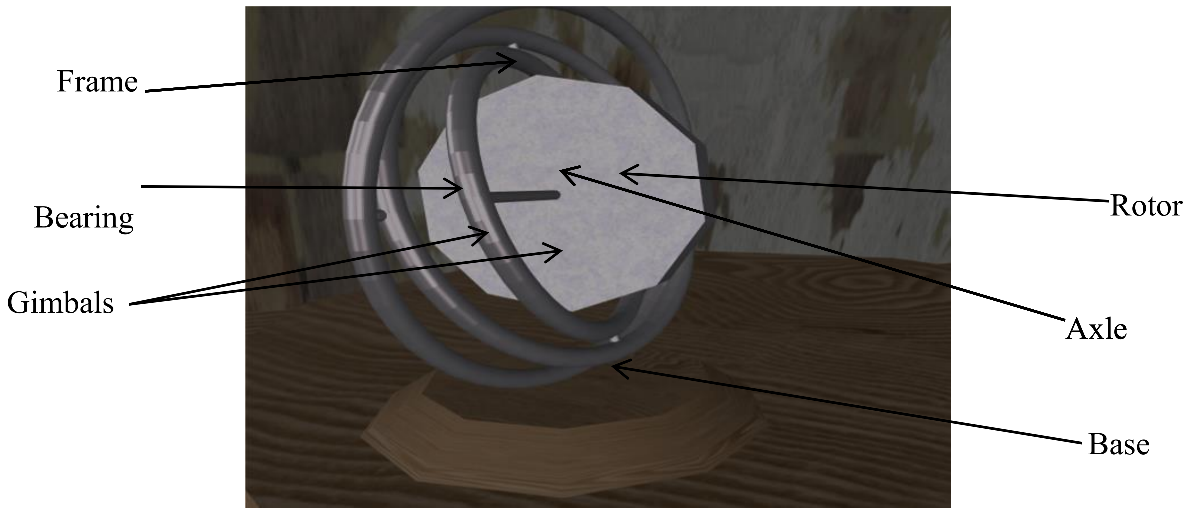
Instancing

Instancing has been used throughout the project. The ShapeCreator class contains multiple methods that create various objects. The scope of these shapes ranges from cuboids, spheres, and cylinders to more complex objects such as a gyroscope, terrain and trees. An example of instancing simple objects like the cuboids can be seen below in the image of the stairs:



The stairs are constructed from 15 cuboids. After each cuboid is rendered, the next one is rendered after translating by the height and length of the previous step.

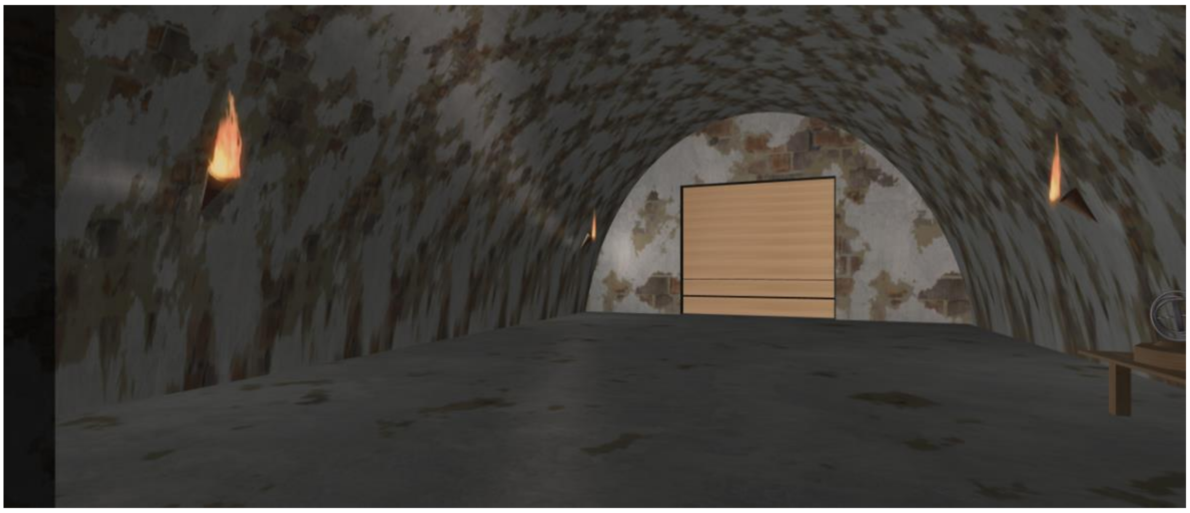
Perhaps a more intricate example of instancing occurs when drawing the gyroscope. The image below shows this object:



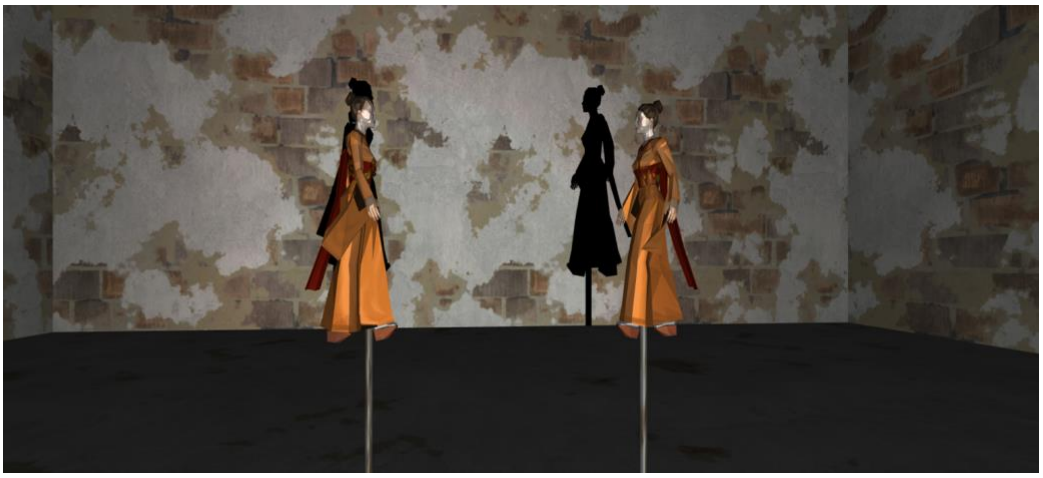
The gyroscope is composed from tori and cylinders. A torus is used to create the frame, likewise the two gimbals are made from two scaled down tori. The bearings (of which there are four) and created using cylinders. In the same manner the axle, that runs through the inner gimbal, is made of a longer cylinder. The rotor is also a cylinder, however this time it is drawn using a smaller number of stacks and slices. This is to make it more obvious that it is, in fact, spinning. The base is also a cylinder (the call is to the same function), however this time the radii of the top and base a specified to be different make it into a truncated cone.

This example is used to demonstrate how more complex models can be built up through instancing simpler shapes – in this case tori and cylinders.

Another example of instancing is the three torches that are mounted to the wall of the tunnel. The model for the torch is composed of a cone (a cylinder with the base's radius set to 0) and a plane that is animated by changing the texture with each frame of the program.



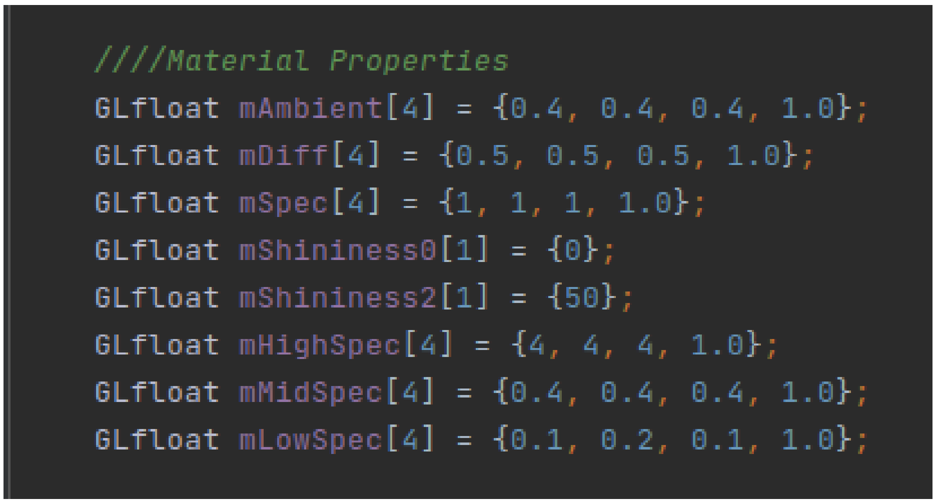
The last example of instancing I will give is of the two figurines. The models this time are imported from an OBJ file (I will go into more depth about this later in the report) and a cylinder is added to the base of the model. There vertex, UV and normal data of the OBJ is saved to memory once. The model is then instanced by calling the drawGeisha() method twice. Interestingly the shadows are just the same models redrawn after the colour has been set to black and a matrix projection is applied to them.



Specular and Diffusive lighting

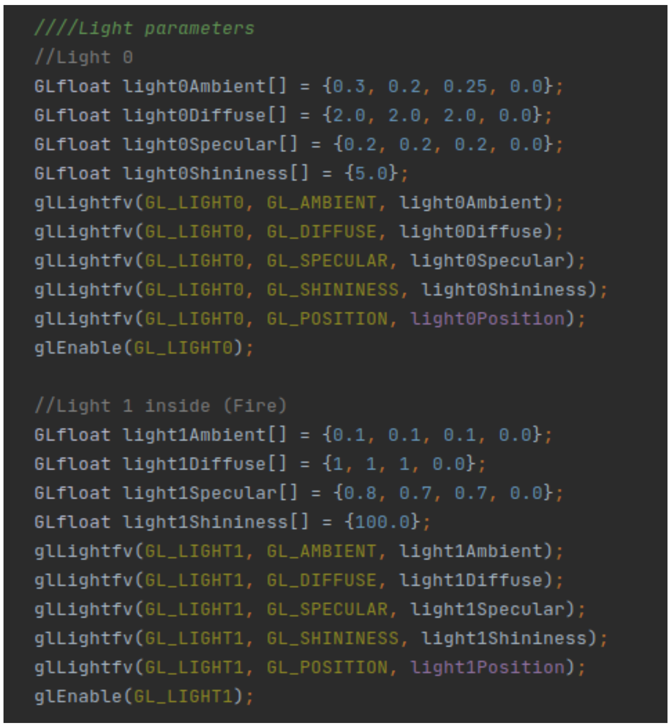
There are 5 lights within the scene, the fire, three torches, and the sun. Each light has ambient, diffusive, and specular properties. Material properties have been set so the objects interact with the lighting and can display the ambient, diffusive, and specular contributions. The lights' positions are set to where the emitting light source would be in the scene i.e., the position of the fire texture, the positions of the torches, and the position of where the sun sits in the sky box. The positions are described using homogenous coordinates. The last parameter of the lights' positions, w, is set to 1 for each of the lights, this implies that the lights are positioned in the x, y, and z coordinates specified and the diffuse and specular calculations are based on the location of the light with respect to the gluLookAt coordinates.

Since the lighting calculation are based on vertex positions and normals, most objects – such as walls – were tessellated so that each polygon's lighting was calculated to be slightly different. This allows flat planes to appear to be lit smoothly and resembled real lighting more accurately. It also stopped objects from looking particularly flat.



```
////Material Properties
GLfloat mAmbient[4] = {0.4, 0.4, 0.4, 1.0};
GLfloat mDiff[4] = {0.5, 0.5, 0.5, 1.0};
GLfloat mSpec[4] = {1, 1, 1, 1.0};
GLfloat mShininess0[1] = {0};
GLfloat mShininess2[1] = {50};
GLfloat mHighSpec[4] = {4, 4, 4, 1.0};
GLfloat mMidSpec[4] = {0.4, 0.4, 0.4, 1.0};
GLfloat mLowSpec[4] = {0.1, 0.2, 0.1, 1.0};
```

Initialising material properties

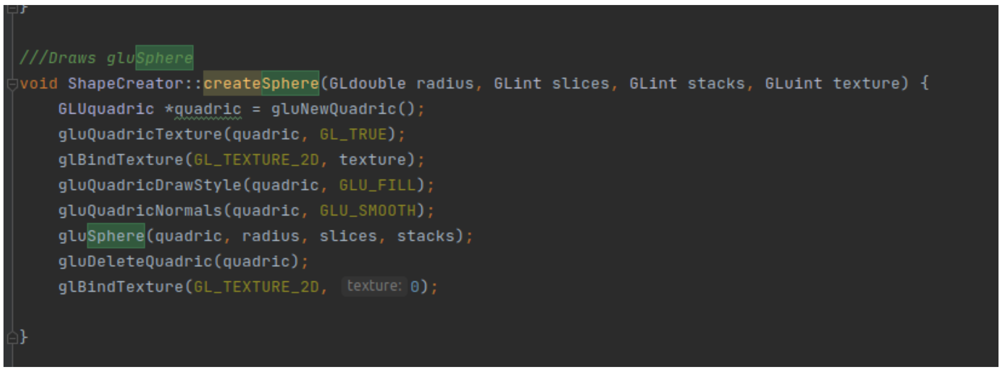


```
////Light parameters
//Light 0
GLfloat light0Ambient[] = {0.3, 0.2, 0.25, 0.0};
GLfloat light0Diffuse[] = {2.0, 2.0, 2.0, 0.0};
GLfloat light0Specular[] = {0.2, 0.2, 0.2, 0.0};
GLfloat light0Shininess[] = {5.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, light0Ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light0Diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light0Specular);
glLightfv(GL_LIGHT0, GL_SHININESS, light0Shininess);
glLightfv(GL_LIGHT0, GL_POSITION, light0Position);
glEnable(GL_LIGHT0);

//Light 1 inside (Fire)
GLfloat light1Ambient[] = {0.1, 0.1, 0.1, 0.0};
GLfloat light1Diffuse[] = {1, 1, 1, 0.0};
GLfloat light1Specular[] = {0.8, 0.7, 0.7, 0.0};
GLfloat light1Shininess[] = {100.0};
glLightfv(GL_LIGHT1, GL_AMBIENT, light1Ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light1Diffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, light1Specular);
glLightfv(GL_LIGHT1, GL_SHININESS, light1Shininess);
glLightfv(GL_LIGHT1, GL_POSITION, light1Position);
glEnable(GL_LIGHT1);
```

Setting lighting parameters for lights 0 and 1

Glut objects

Glut objects have been used throughout the scene. The glut object used are gluCylinder, gluDisk, and gluSphere. A smaller wrapper has been written for each Glut object so they can easily be created and textured. This means that a GLUquadric does not have to be manually called every time a Glut object is needed. It also means that the memory can be handled efficiently, guaranteeing that for every GLUquadric initialised a respective gluDeleteQuadric is called. This layer of abstraction may seem simple, but as the complexity of the scene increases methods like these ones make handling the program much easier. Two separate methods were created that create cylinders – createCylinder and createEdgeCylinder. The createCylinder method creates a glut cylinder, by default a glut cylinder has no top or base. This may be useful when only the curved portion of the cylinder is needed. By doing this the number of vertices in the scene is slightly reduced. The createEdgeCylinder method draws a cylinder in a similar fashion to createCylinder, however this time two disks are placed on either side to add a base and top to it. The glut disk has the number of slices and stacks and the glut cylinder guaranteeing that the disk will fit on the cylinder. As mentioned before this reduces the number of manual calls to gluCylinder and gluDisk and stops any errors caused by the shapes being transformed incorrectly. Usages of the glut objects can be seen in the trees (a sphere, and a cylinder), the gyroscope (its base is a edge cylinder and bearings are cylinders), the poles for the figurines (a cylinder), and the cones that make up the torches (a cylinder with a base of 0 and a top that is larger). The glut objects are textured by calling the gluQuadricTexture method – which automatically creates UV coordinates for the object and then binding the appropriate texture. The method gluQuadricNormals was used to generate the normals for each of the glut objects.



```
///Draws gluSphere
void ShapeCreator::createSphere(GLdouble radius, GLint slices, GLint stacks, GLuint texture) {
    GLUquadric *quadric = gluNewQuadric();
    gluQuadricTexture(quadric, GL_TRUE);
    glBindTexture(GL_TEXTURE_2D, texture);
    gluQuadricDrawStyle(quadric, GLU_FILL);
    gluQuadricNormals(quadric, GLU_SMOOTH);
    gluSphere(quadric, radius, slices, stacks);
    gluDeleteQuadric(quadric);
    glBindTexture(GL_TEXTURE_2D, texture: 0);

}
```

Creating a sphere using Glut Objects

Explanation of Design

As mentioned before the program is split into four main classes, the window, the SceneWidget, the ShapeCreator, and the TextureCreator. The ShapeCreator class aims to create both simple objects and complex objects that are composed of the simple objects. The objects can then be placed in the scene accordingly. This allows the shapes to be created with reference object coordinates and then the objects to be placed with respect to their world coordinates. This bottom up approach is what allowed me to create a fairly complex scene.

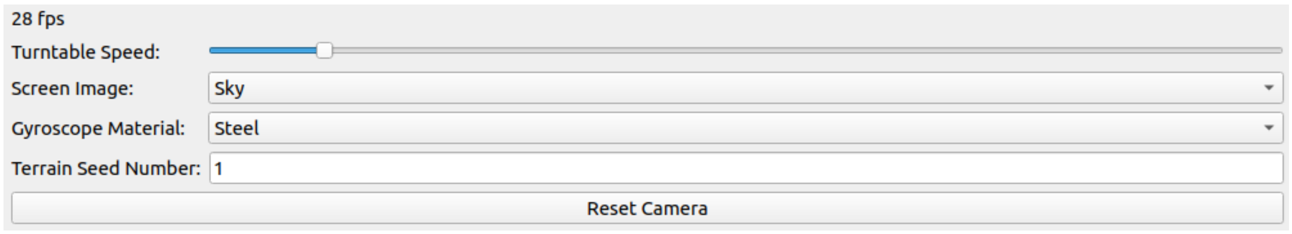
The main room of the cave was the first object created. Initially it was a scaled up cuboid. Tessellation was added to allow the diffusive and specular lighting to become more apparent (the lighting is calculated for each shape, so each wall is therefore composed of many smaller rectangles). A door was then added to one of the walls by using an if statement that specified where the tessellated rectangle should not be drawn. This doorway was then fleshed out by duplicating the wall containing the door and offsetting it by two. Three planes were then drawn connecting the two walls to create a 3 dimensional doorway. This same process was applied to the north wall (in the -Z direction) to create the fireplace that surrounds the fire. Objects – that were defined in the ShapeCreator class – were then placed in the room. These objects are the two figurines and the fire.

The fire is composed of 10 textured planes place close to one other. Fire was particularly difficult to create in OpenGL, particle systems can be used but look fairly unrealistic. I also struggled to find a adequate fire animation under Creative Commons. To combat this, I created a fire animation in Adobe After Effects – a software that I am familiar with – using primarily fractal noise and turbulent displacement. I selected 20 consecutive frames that I liked and rendered then onto the ten textured planes according to what frame the program is on. This created a fire that loops every 20 frames. The After Effect composition was rendered to a PNG sequence. This means that each resulting image supports transparency. This is carried across to OpenGL by enabling blender after loading the images with an alpha channel.

Various elements of the object oriented programming paradigm were used to manage the structure of the program like classes, inheritance, and encapsulation. Additionally, some elements of the event driven programming paradigm were used to handle user input such as keyboard presses and mouse clicks performed on the user interface that handles camera movement and QWidget interaction.

50% - 60% Band

Elements of User Interaction



28 fps

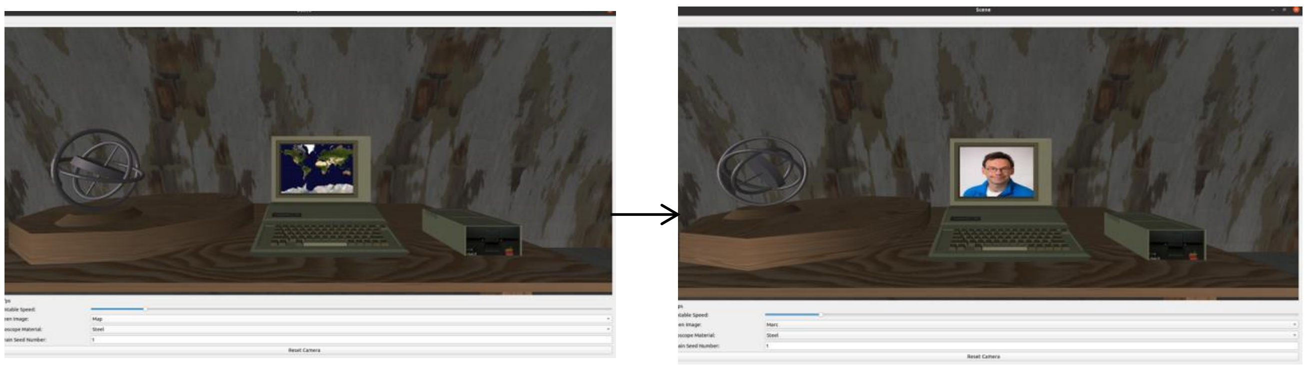
Turntable Speed:

Screen Image: Sky

Gyroscope Material: Steel

Terrain Seed Number: 1

Various elements of user interaction are used. The top QSlider is a user adjustable slider that can be used to set the speed of rotation for the turntable that the gyroscope sits upon



ntable Speed:

een Image: Map

roscope Material: Steel

rain Seed Number: 1

ntable Speed:

een Image: Marc

roscope Material: Steel

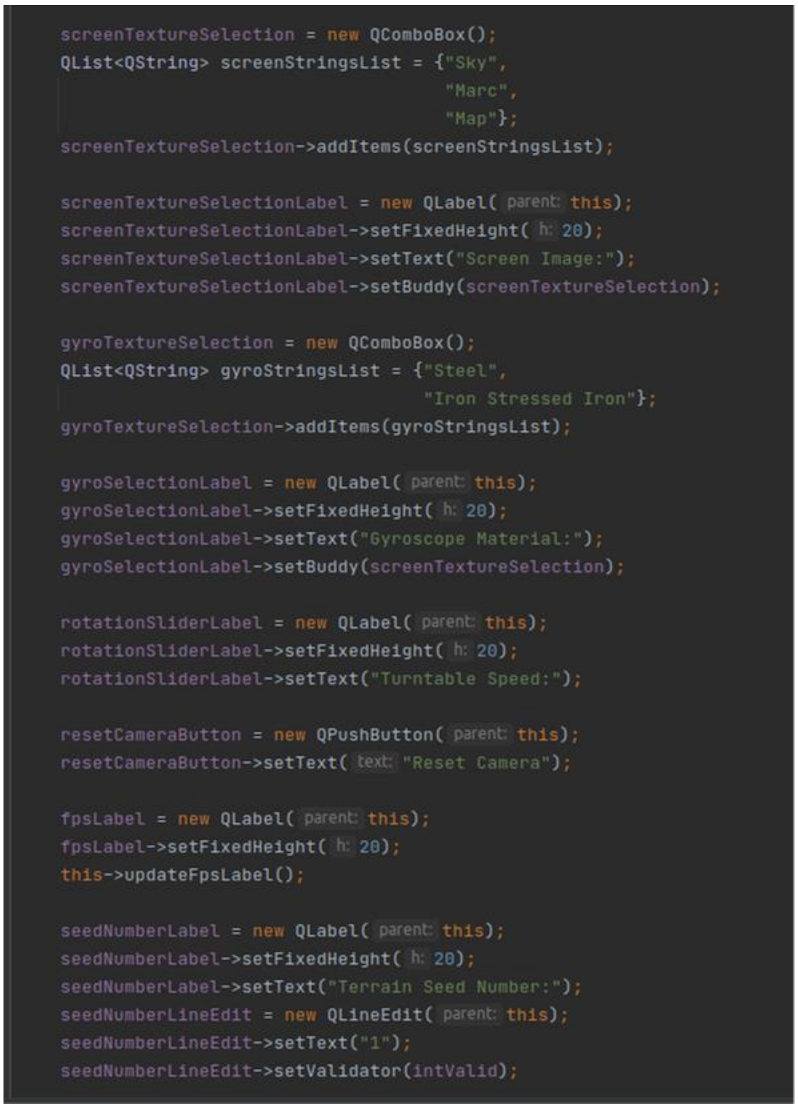
rain Seed Number: 1

The second element is a QComboBox that allows users to select 3 options. Upon clicking an option the textured plane that sits in front of the PC – that acts as a screen – is changed. Here I have included the options to selected the compulsory textures for this coursework.

Likewise the 3rd element is another QComboBox that allows users to selected textures used for the gyroscope.

The 4th element is the random seed used to generate the procedural terrain. The seed is used by srand() to set the various random heights at different points of the terrain and set the trees' x, y, and z coordinates. Users can edit this seed within the program during run time. This will recall the heightsGenerated method and render a different terrain. Since srand() uses pseudo randomness, prior seeds can be reused to return to a user's preferred terrain.

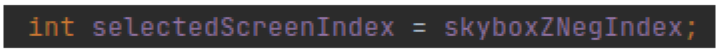
The "Reset Camera" button can be used by the user to place the camera back to its starting position.



```
screenTextureSelection = new QComboBox();
QList<QString> screenStringsList = {"Sky",
                                    "Marc",
                                    "Map"};
screenTextureSelection->addItems(screenStringsList);

screenTextureSelectionLabel = new QLabel( parent: this);
screenTextureSelectionLabel->setFixedHeight( h: 20);
screenTextureSelectionLabel->setText("Screen Image:");
screenTextureSelectionLabel->setBuddy(screenTextureSelection);

gyroTextureSelection = new QComboBox();
QList<QString> gyroStringsList = {"Steel",
                                  "Iron Stressed Iron"};
gyroTextureSelection->addItems(gyroStringsList);

gyroSelectionLabel = new QLabel( parent: this);
gyroSelectionLabel->setFixedHeight( h: 20);
gyroSelectionLabel->setText("Gyroscope Material:");
gyroSelectionLabel->setBuddy(screenTextureSelection);

rotationSliderLabel = new QLabel( parent: this);
rotationSliderLabel->setFixedHeight( h: 20);
rotationSliderLabel->setText("Turntable Speed:");

resetCameraButton = new QPushButton( parent: this);
resetCameraButton->setText( text: "Reset Camera");

fpsLabel = new QLabel( parent: this);
fpsLabel->setFixedHeight( h: 20);
this->updateFpsLabel();

seedNumberLabel = new QLabel( parent: this);
seedNumberLabel->setFixedHeight( h: 20);
seedNumberLabel->setText("Terrain Seed Number:");
seedNumberLineEdit = new QLineEdit( parent: this);
seedNumberLineEdit->setText("1");
seedNumberLineEdit->setValidator(intValid);
```

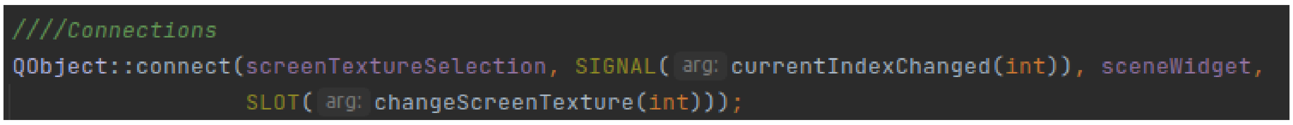
Creation of QWidgets

All of these elements of user interaction were implemented using the same basic concept. First a connection is established between a QWidget and the SceneWidget. The connection allows an event listener to wait for the QWidget to emit a signal. Once the signal is emitted the relevant slot is called in the SceneWidget which handles the event.



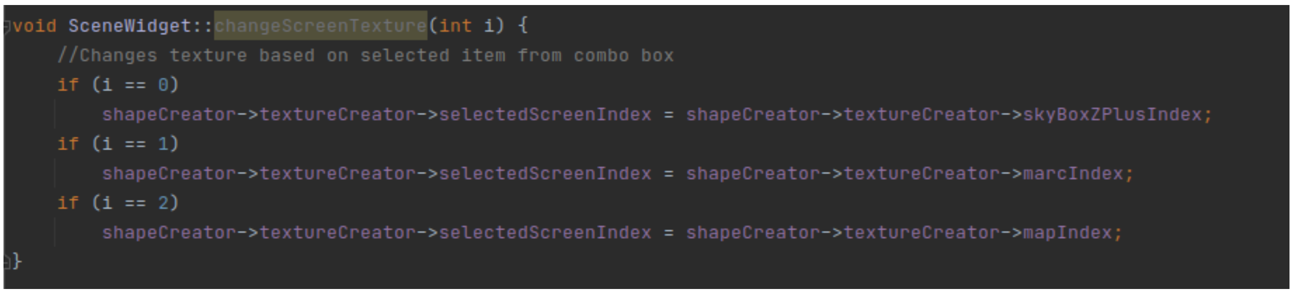
```
int selectedScreenIndex = skyboxZNegIndex;
```

For example, the PC screen's texture is bound using a variable named selectedScreenIndex that is defined in the TextureCreator class.



```
////Connections
QObject::connect(screenTextureSelection, SIGNAL( arg: currentIndexChanged(int)), sceneWidget,
                 SLOT( arg: changeScreenTexture(int)));
```

Initially this variable is set to be equal to the first sky texture. Within the window class, a connection is established using QObject::connect(). This adds the event listener between the combo box's signal – currentIndexChanged() – and the sceneWidget's slot – changeScreenTexture(). The index is then passed to changeScreenTexture().



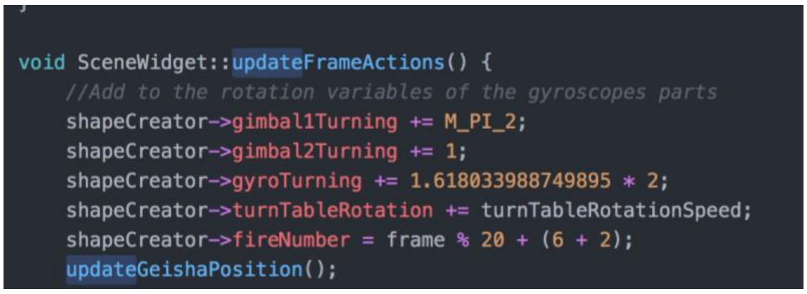
```
void SceneWidget::changeScreenTexture(int i) {
    //Changes texture based on selected item from combo box
    if (i == 0)
        shapeCreator->textureCreator->selectedScreenIndex = shapeCreator->textureCreator->skyBoxZPlusIndex;
    if (i == 1)
        shapeCreator->textureCreator->selectedScreenIndex = shapeCreator->textureCreator->marcIndex;
    if (i == 2)
        shapeCreator->textureCreator->selectedScreenIndex = shapeCreator->textureCreator->mapIndex;
}
```

The changeScreenTexture slot then handles the event by changing the selectedScreenIndex variable to the relevant texture index based on the input from the QComboBox.

60%-70% Band
Elements of animation
Animation has been implemented to control the position of the figurines. The 2 figurines move along the x axis until they reach the defined limit. Upon reaching this limit they turn 180 degrees and move in the other direction until reaching the other limit and then turn. This process is repeated indefinitely. The animation process is defined for one figure and the other figurine is drawn using the same coordinate only with the x coordinate negated. Additionally, the second figurines rotation is 180 less than the first.

Animation is also seen in the turntable and gyroscope. The turntable spins along its y axis indefinitely by updating the value by which is rotated by every frame. Likewise, the gyroscope spins upon 3 different axes as gyroscopes do. The outer gimbal spins upon the x axis, the inner gimbal spins upon its y axis and the rotor spins on its z axis. Like the turntable, the rotation variable for the gyroscope is updated each frame.

The fire is a plane with a sequence of PNG images being updated each frame. This gives the illusion of movement, similar to a film, however the polygon is entirety static.



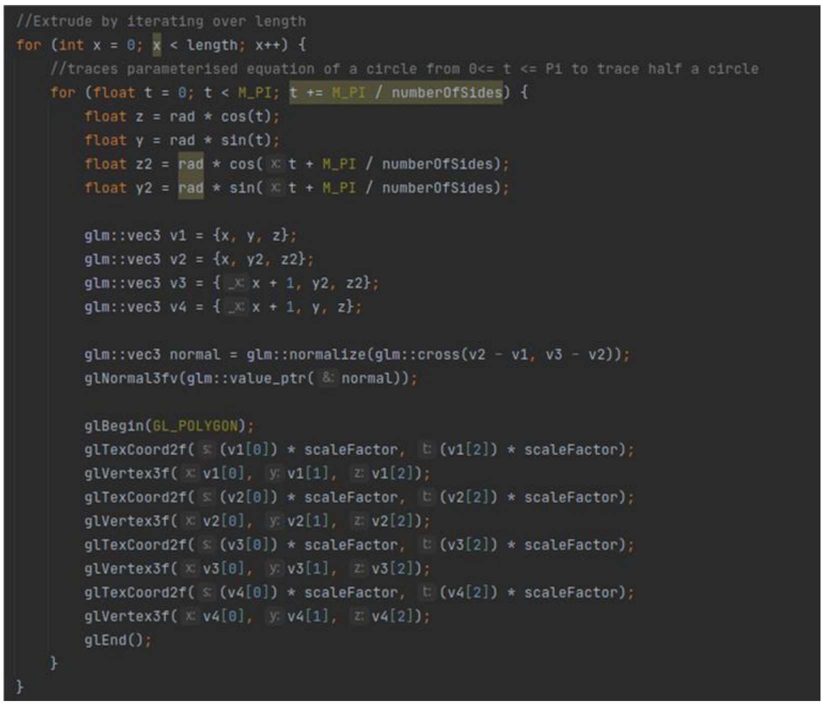
```
void SceneWidget::updateFrameActions() {
    //Add to the rotation variables of the gyroscopes parts
    shapeCreator->gimbal1Turning += M_PI_2;
    shapeCreator->gimbal2Turning += 1;
    shapeCreator->gyroTurning += 1.618033988749895 * 2;
    shapeCreator->turnTableRotation += turnTableRotationSpeed;
    shapeCreator->fireNumber = frame % 20 + (6 + 2);
    updateGeishaPosition();
```

Updating the rotation variables

Convex objects

<u>Semi-cylinder</u>

Three convex objects were created within this project. The first is a semi-cylinder which is created by tracing the parametric coordinates of a semicircle (using y and z coordinates where z = r * cos(t) and y = r * sin(t) and 0 <= t <= π) and extruding it along its x axis. Firstly, a 3 dimensional arch is drawn. Each arch consists of 100 planes. The parametric coordinates are traced with the x coordinate for the second and fourth vertices for each plane being extended by 1. The arch is then extruded by its length by repeatedly drawing consecutive arches to produce a semi-cylinder. The semi-cylinder is used to draw the tunnel leading out of the cave.



```
//Extrude by iterating over length
for (int x = 0; x < length; x++) {
    //traces parameterised equation of a circle from 0<= t <= Pi to trace half a circle
    for (float t = 0; t < M_PI; t += M_PI / numberOfSides) {
        float z = rad * cos(t);
        float y = rad * sin(t);
        float z2 = rad * cos( x: t + M_PI / numberOfSides);
        float y2 = rad * sin( x: t + M_PI / numberOfSides);

        glm::vec3 v1 = {x, y, z};
        glm::vec3 v2 = {x, y2, z2};
        glm::vec3 v3 = { _x: x + 1, y2, z2};
        glm::vec3 v4 = { _x: x + 1, y, z};

        glm::vec3 normal = glm::normalize(glm::cross(v2 - v1, v3 - v2));
        glNormal3fv(glm::value_ptr( &: normal));

        glBegin(GL_POLYGON);
        glTexCoord2f( s: (v1[0]) * scaleFactor,  t: (v1[2]) * scaleFactor);
        glVertex3f( x: v1[0],  y: v1[1],  z: v1[2]);
        glTexCoord2f( s: (v2[0]) * scaleFactor,  t: (v2[2]) * scaleFactor);
        glVertex3f( x: v2[0],  y: v2[1],  z: v2[2]);
        glTexCoord2f( s: (v3[0]) * scaleFactor,  t: (v3[2]) * scaleFactor);
        glVertex3f( x: v3[0],  y: v3[1],  z: v3[2]);
        glTexCoord2f( s: (v4[0]) * scaleFactor,  t: (v4[2]) * scaleFactor);
        glVertex3f( x: v4[0],  y: v4[1],  z: v4[2]);
        glEnd();
    }
}
```

Drawing the semi-cylinder

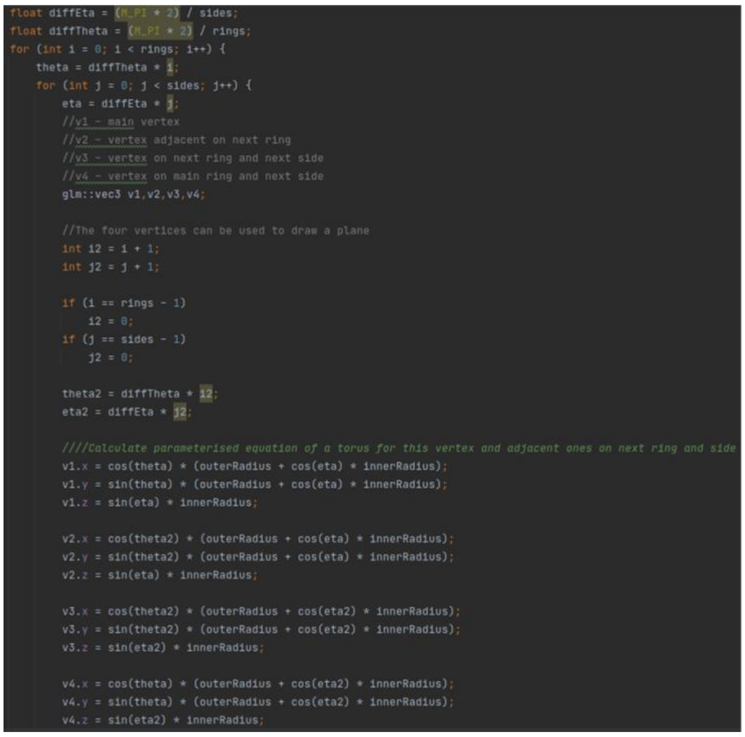
Torus

The parametric equation for a torus is

$x = \cos(\theta) * (c + \cos(\eta) * a)$
$y = \sin(\theta) * (c + \cos(\eta) * a)$
$z = \sin(\eta) * a$

with $0 <= \theta <= 2\pi$ and $0 <= \eta <= 2\pi$ and c being the distance from the central point to the centre of the outer circular and a being the radius of the outer circle.

Using this information, the number of sides and rings was passed to the function to dictate how many times η and θ should be iterated through. The four vertices of each plane were then drawn using information about the current η and θ and well and the next one.



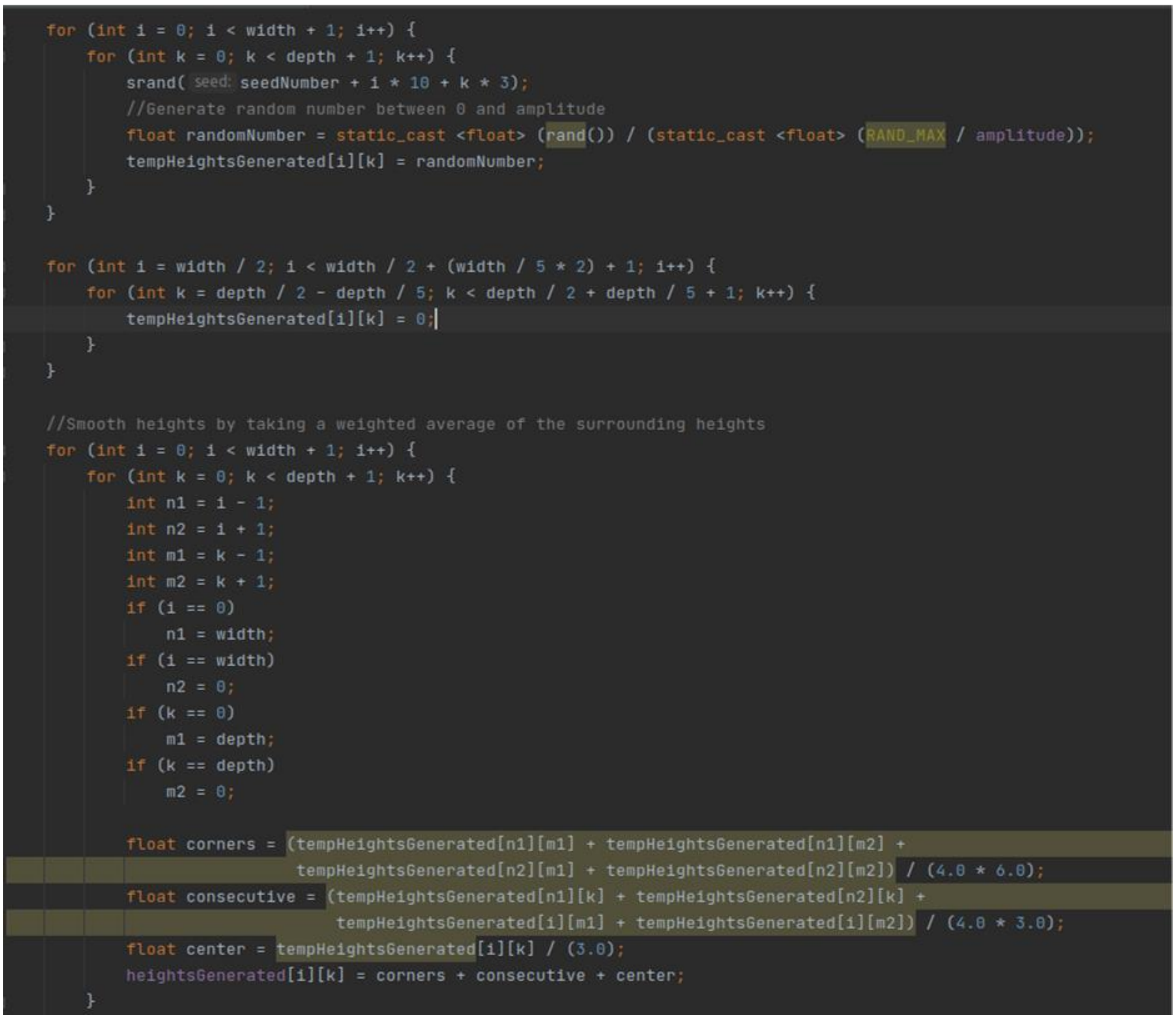
```
float diffEta = (M_PI * 2) / sides;
float diffTheta = (M_PI * 2) / rings;
for (int i = 0; i < rings; i++) {
    theta = diffTheta * i;
    for (int j = 0; j < sides; j++) {
        eta = diffEta * j;
        //v1 - main vertex
        //v2 - vertex adjacent on next ring
        //v3 - vertex on next ring and next side
        //v4 - vertex on main ring and next side
        glm::vec3 v1,v2,v3,v4;

        //The four vertices can be used to draw a plane
        int i2 = i + 1;
        int j2 = j + 1;

        if (i == rings - 1)
            i2 = 0;
        if (j == sides - 1)
            j2 = 0;

        theta2 = diffTheta * i2;
        eta2 = diffEta * j2;

        ////Calculate parameterised equation of a torus for this vertex and adjacent ones on next ring and side
        v1.x = cos(theta) * (outerRadius + cos(eta) * innerRadius);
        v1.y = sin(theta) * (outerRadius + cos(eta) * innerRadius);
        v1.z = sin(eta) * innerRadius;

        v2.x = cos(theta2) * (outerRadius + cos(eta) * innerRadius);
        v2.y = sin(theta2) * (outerRadius + cos(eta) * innerRadius);
        v2.z = sin(eta) * innerRadius;

        v3.x = cos(theta2) * (outerRadius + cos(eta2) * innerRadius);
        v3.y = sin(theta2) * (outerRadius + cos(eta2) * innerRadius);
        v3.z = sin(eta2) * innerRadius;

        v4.x = cos(theta) * (outerRadius + cos(eta2) * innerRadius);
        v4.y = sin(theta) * (outerRadius + cos(eta2) * innerRadius);
        v4.z = sin(eta2) * innerRadius;
```

Calculating the coordinates of the vertices of the torus

Terrain

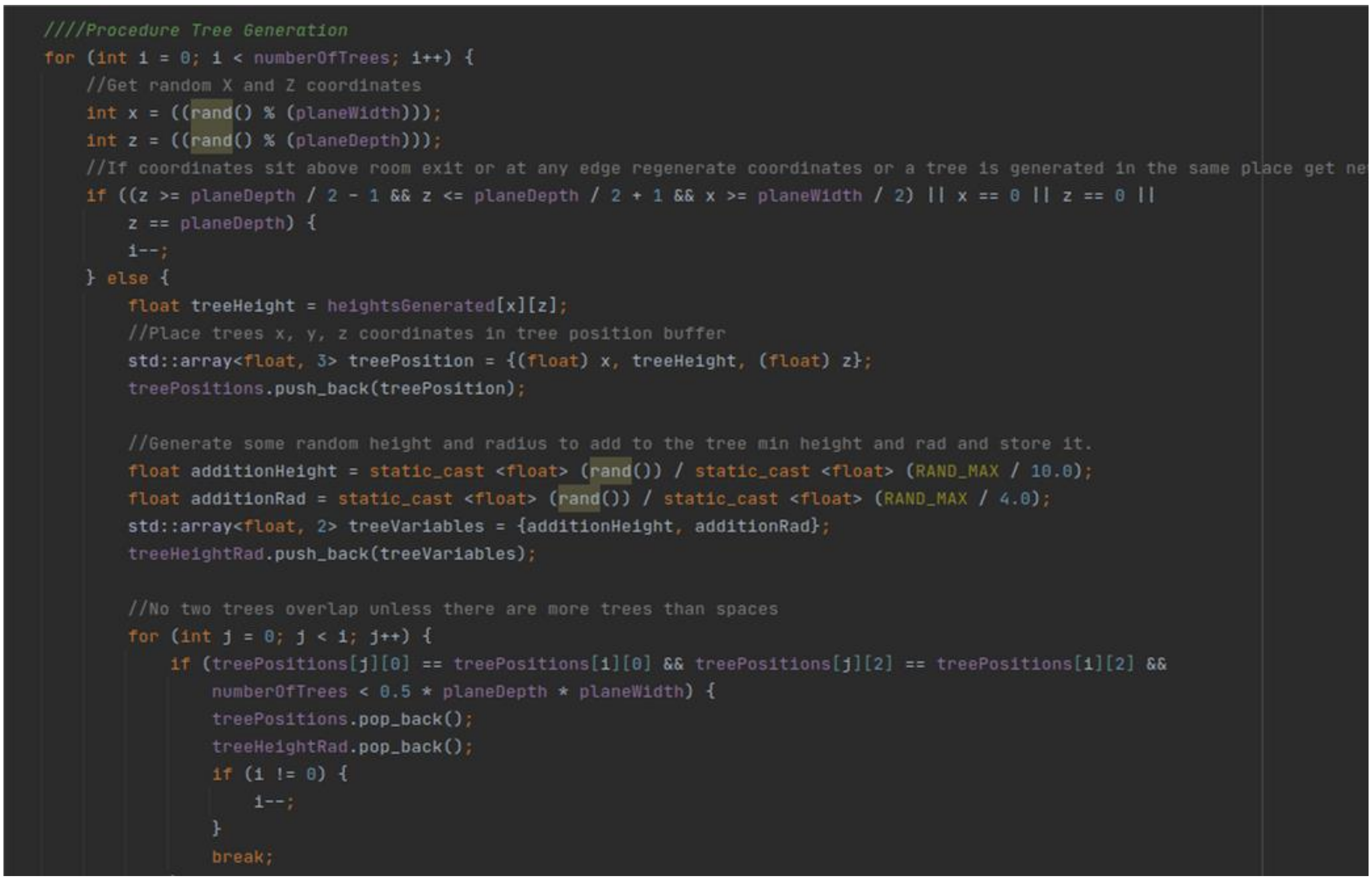
The terrain can be thought of as a convex object. It is composed of triangles that are places next to one another. It is generated by adding a number of random height values for different points across a x and z plane. The points are then smoothed out by taking a weighted averaged of its surrounding points. Triangles are then rendered, filling the space in between each of points where the heights generated. To work out the height of each triangle its height is interpolated based on how many far it is between each point and the height values of the points its between. The interpolation is done using a normalised cos function (that is to say a cos function that is normalised to return a value between 0 and 1) to interpolate how much of each points value should be used. Between the four points, the height is interpolated in the x axis twice between points that have the same z value. These temporary height values are interpolated using the same process to interpolate in the z axis. Using this method, every point's interpolated height can be determined between the 4 points in a smooth fashion.

The number of unique vertices that exist between any 4 points is defined as the product of its x and z tessellation factor i.e., how many times the points are interpolated between in both the x and z directions. In my program the x and z tessellation factors are both set to 20. The number of initial height values is the product of the plane width and plane depth. In my program both the plane width and depth are set to 10. This means that there are 100 heights which are interpolated between in 400 places. At each occurrence two triangles are drawn. This means that the terrain consists of 100 * 400 * 6 (240,000) vertices. If we define the plane width as X, the plane depth as Z, the x axis tessellation factor as x and the z axis tessellation factor as z then this method is O(X * Z * x * z). The method is rather inefficient and does not scale well, it is also likely to be the bottleneck of my program. The method could be improved upon by using a VBO and currently multiple vertices are being reusing instead of being indexed. Despite this I am happy with the results and I find it to be visually impressive.



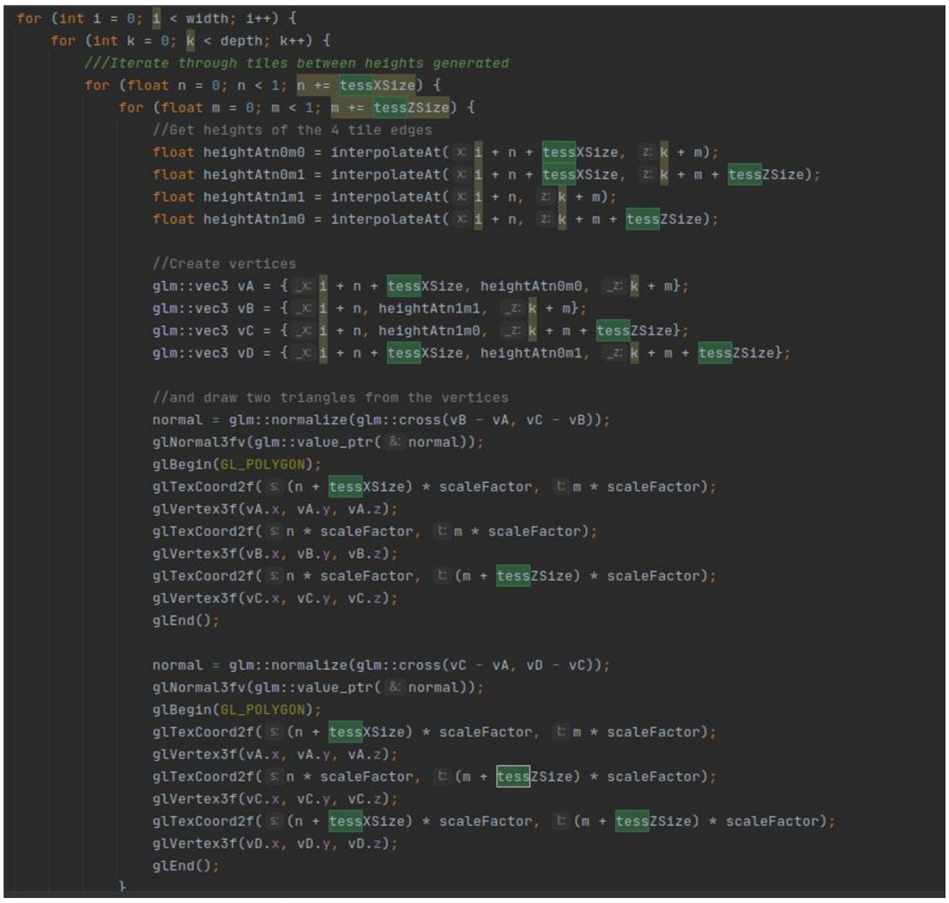
```
for (int i = 0; i < width + 1; i++) {
    for (int k = 0; k < depth + 1; k++) {
        srand( seed: seedNumber + i * 10 + k * 3);
        //Generate random number between 0 and amplitude
        float randomNumber = static_cast <float> (rand()) / (static_cast <float> (RAND_MAX / amplitude));
        tempHeightsGenerated[i][k] = randomNumber;
    }
}

for (int i = width / 2; i < width / 2 + (width / 5 * 2) + 1; i++) {
    for (int k = depth / 2 - depth / 5; k < depth / 2 + depth / 5 + 1; k++) {
        tempHeightsGenerated[i][k] = 0;
    }
}

//Smooth heights by taking a weighted average of the surrounding heights
for (int i = 0; i < width + 1; i++) {
    for (int k = 0; k < depth + 1; k++) {
        int n1 = i - 1;
        int n2 = i + 1;
        int m1 = k - 1;
        int m2 = k + 1;
        if (i == 0)
            n1 = width;
        if (i == width)
            n2 = 0;
        if (k == 0)
            m1 = depth;
        if (k == depth)
            m2 = 0;

        float corners = (tempHeightsGenerated[n1][m1] + tempHeightsGenerated[n1][m2] +
                         tempHeightsGenerated[n2][m1] + tempHeightsGenerated[n2][m2]) / (4.0 * 6.0);
        float consecutive = (tempHeightsGenerated[n1][k] + tempHeightsGenerated[n2][k] +
                             tempHeightsGenerated[i][m1] + tempHeightsGenerated[i][m2]) / (4.0 * 3.0);
        float center = tempHeightsGenerated[i][k] / (3.0);
        heightsGenerated[i][k] = corners + consecutive + center;
    }
```

Height Generation



```
////Procedure Tree Generation
for (int i = 0; i < numberOfTrees; i++) {
    //Get random X and Z coordinates
    int x = ((rand() % (planeWidth)));
    int z = ((rand() % (planeDepth)));
    //If coordinates sit above room exit or at any edge regenerate coordinates or a tree is generated in the same place get ne
    if ((z >= planeDepth / 2 - 1 && z <= planeDepth / 2 + 1 && x >= planeWidth / 2) || x == 0 || z == 0 ||
        z == planeDepth) {
        i--;
    } else {
        float treeHeight = heightsGenerated[x][z];
        //Place trees x, y, z coordinates in tree position buffer
        std::array<float, 3> treePosition = {(float) x, treeHeight, (float) z};
        treePositions.push_back(treePosition);

        //Generate some random height and radius to add to the tree min height and rad and store it.
        float additionHeight = static_cast <float> (rand()) / static_cast <float> (RAND_MAX / 10.0);
        float additionRad = static_cast <float> (rand()) / static_cast <float> (RAND_MAX / 4.0);
        std::array<float, 2> treeVariables = {additionHeight, additionRad};
        treeHeightRad.push_back(treeVariables);

        //No two trees overlap unless there are more trees than spaces
        for (int j = 0; j < i; j++) {
            if (treePositions[j][0] == treePositions[i][0] && treePositions[j][2] == treePositions[i][2] &&
                numberOfTrees < 0.5 * planeDepth * planeWidth) {
                treePositions.pop_back();
                treeHeightRad.pop_back();
                if (i != 0) {
                    i--;
                }
                break;
```

Tree Heights and positions



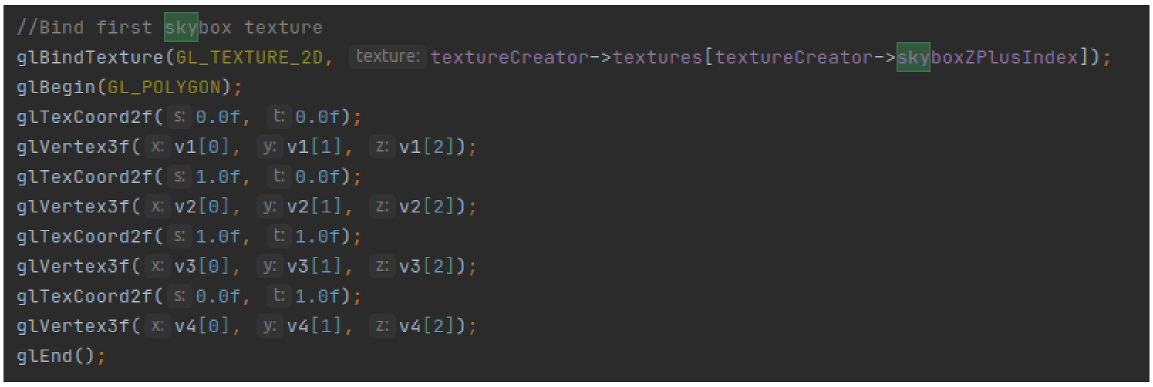
```
for (int i = 0; i < width; i++) {
    for (int k = 0; k < depth; k++) {
        ///Iterate through tiles between heights generated
        for (float n = 0; n < 1; n += tessXSize) {
            for (float m = 0; m < 1; m += tessZSize) {
                //Get heights of the 4 tile edges
                float heightAtn0m0 = interpolateAt( x: i + n + tessXSize,  z: k + m);
                float heightAtn0m1 = interpolateAt( x: i + n + tessXSize,  z: k + m + tessZSize);
                float heightAtn1m1 = interpolateAt( x: i + n,  z: k + m);
                float heightAtn1m0 = interpolateAt( x: i + n,  z: k + m + tessZSize);

                //Create vertices
                glm::vec3 vA = { _x: i + n + tessXSize, heightAtn0m0,  _z: k + m};
                glm::vec3 vB = { _x: i + n, heightAtn1m1,  _z: k + m};
                glm::vec3 vC = { _x: i + n, heightAtn1m0,  _z: k + m + tessZSize};
                glm::vec3 vD = { _x: i + n + tessXSize, heightAtn0m1,  _z: k + m + tessZSize};

                //and draw two triangles from the vertices
                normal = glm::normalize(glm::cross(vB - vA, vC - vB));
                glNormal3fv(glm::value_ptr( &: normal));
                glBegin(GL_POLYGON);
                glTexCoord2f( s: (n + tessXSize) * scaleFactor,  t: m * scaleFactor);
                glVertex3f(vA.x, vA.y, vA.z);
                glTexCoord2f( s: n * scaleFactor,  t: m * scaleFactor);
                glVertex3f(vB.x, vB.y, vB.z);
                glTexCoord2f( s: n * scaleFactor,  t: (m + tessZSize) * scaleFactor);
                glVertex3f(vC.x, vC.y, vC.z);
                glEnd();

                normal = glm::normalize(glm::cross(vC - vA, vD - vC));
                glNormal3fv(glm::value_ptr( &: normal));
                glBegin(GL_POLYGON);
                glTexCoord2f( s: (n + tessXSize) * scaleFactor,  t: m * scaleFactor);
                glVertex3f(vA.x, vA.y, vA.z);
                glTexCoord2f( s: n * scaleFactor,  t: (m + tessZSize) * scaleFactor);
                glVertex3f(vC.x, vC.y, vC.z);
                glTexCoord2f( s: (n + tessXSize) * scaleFactor,  t: (m + tessZSize) * scaleFactor);
                glVertex3f(vD.x, vD.y, vD.z);
                glEnd();
            }
```

Drawing the Vertices for the terrain

Texture mapping

Images were loaded into the program using functionality provided by Qt. The QImage() function allows for an easy way to load images into memory by referencing their path. The images were stored as Qimages. Each image was evaluated to see if it was a PNG. If it was 4 channels were used to support alpha based transparency, otherwise 3 channels were used (just RGB). glGenTextures was used so OpenGL fixed pipeline could specify how much space in memory to allocate for textures. Rather than looping over images pixel by pixel to obtain each pixels' RGB(A) information, a simpler approach was used to convert the QImages by using Qt built in convertToFormat() function. Strangely this imported the images invert upon their y axis, so the function mirrored was called beforehand. Once this was done each image was stored in an element within an array of Gluints.

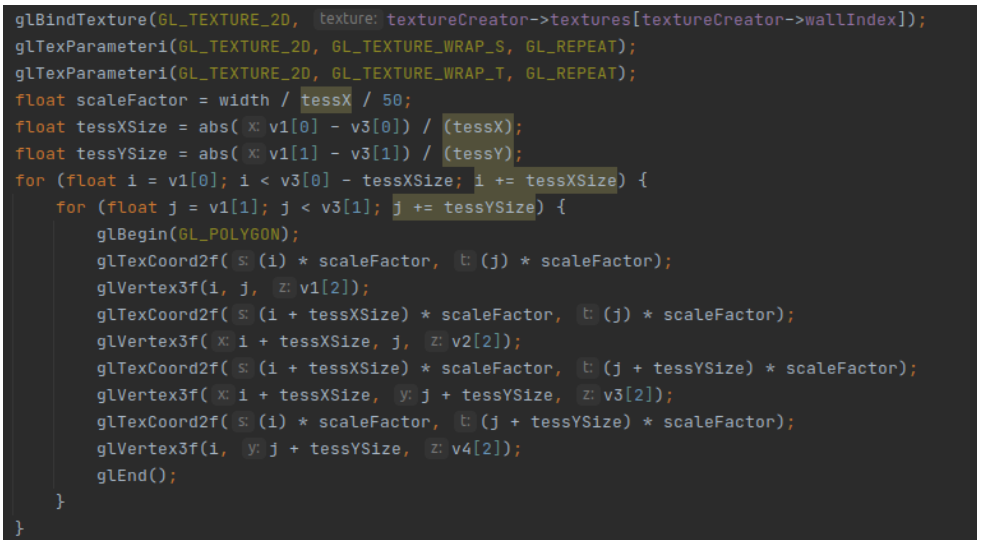
Texture mapping was applied to all objects within my scene. Various textures were sourced from online (see README.txt for more details) and were all used under the creative commons license. Simple objects like the untessellated cube that was used for the sky were fairly simple to texture. The relevant textures where bound and the correct UV coordinates were supplied to each polygon. These textures were clipped to the edge of each polygon by using texture parameters.



```
//Bind first skybox texture
glBindTexture(GL_TEXTURE_2D, texture: textureCreator->textures[textureCreator->skyboxZPlusIndex]);
glBegin(GL_POLYGON);
glTexCoord2f( s: 0.0f,  t: 0.0f);
glVertex3f( x: v1[0],  y: v1[1],  z: v1[2]);
glTexCoord2f( s: 1.0f,  t: 0.0f);
glVertex3f( x: v2[0],  y: v2[1],  z: v2[2]);
glTexCoord2f( s: 1.0f,  t: 1.0f);
glVertex3f( x: v3[0],  y: v3[1],  z: v3[2]);
glTexCoord2f( s: 0.0f,  t: 1.0f);
glVertex3f( x: v4[0],  y: v4[1],  z: v4[2]);
glEnd();
```

One side of the sky box being drawn

Tessellated objected such as walls or the terrain required some additional mathematics to calculate the UV coordinates with respect to where the texture corresponded to the shape and the shape was broken down into smaller tiles. Seamless textures were used and repeated to cause a wallpaper-like looping effect.



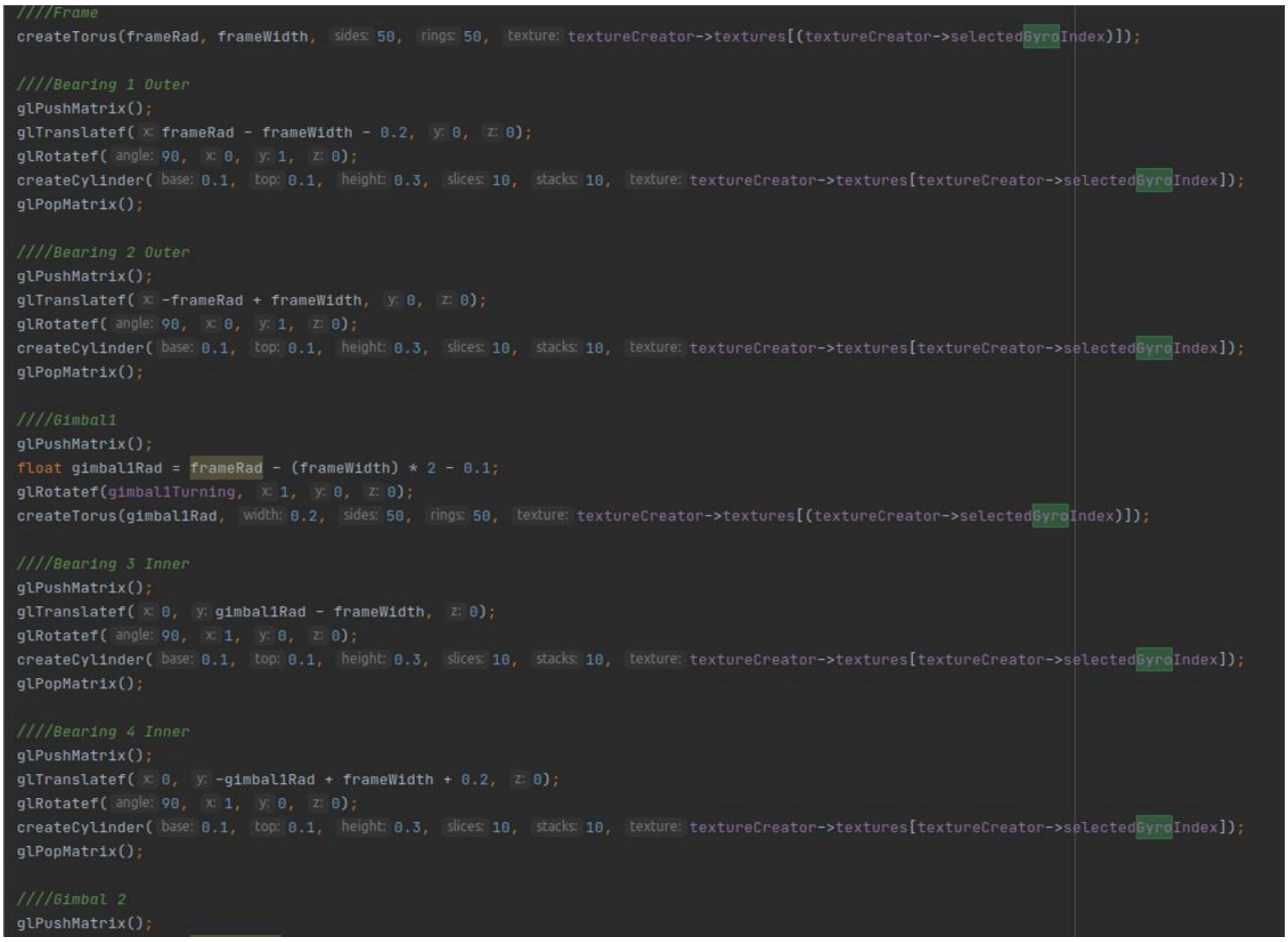
```
glBindTexture(GL_TEXTURE_2D, texture: textureCreator->textures[textureCreator->wallIndex]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
float scaleFactor = width / tessX / 50;
float tessXSize = abs( x: v1[0] - v3[0]) / (tessX);
float tessYSize = abs( x: v1[1] - v3[1]) / (tessY);
for (float i = v1[0]; i < v3[0] - tessXSize; i += tessXSize) {
    for (float j = v1[1]; j < v3[1]; j += tessYSize) {
        glBegin(GL_POLYGON);
        glTexCoord2f( s: (i) * scaleFactor,  t: (j) * scaleFactor);
        glVertex3f(i, j,  z: v1[2]);
        glTexCoord2f( s: (i + tessXSize) * scaleFactor,  t: (j) * scaleFactor);
        glVertex3f( x: i + tessXSize, j,  z: v2[2]);
        glTexCoord2f( s: (i + tessXSize) * scaleFactor,  t: (j + tessYSize) * scaleFactor);
        glVertex3f( x: i + tessXSize,  y: j + tessYSize,  z: v3[2]);
        glTexCoord2f( s: (i) * scaleFactor,  t: (j + tessYSize) * scaleFactor);
        glVertex3f(i,  y: j + tessYSize,  z: v4[2]);
        glEnd();
    }
}
```

One tessellated wall being textured

70%-100% Band

Hierarchical modelling

Hierarchical modelling was used in the gyroscope. It is composed of 3 tori and 7 cylinders. The pushMatrix and popMatrix functions were used repeatedly to handle transformations especially when certain parts needed to be placed relative to other parts. The gyroscope is animated, as statement before, to show movement in its parts. It is placed on a turntable and rotated upon an axis. This demonstrates that the parts move relative to one another and the hierarchical modelling was implemented correctly. The user can also change to rotation speed of the turntable and even bring to a stop. The user can also change the texture applied to the gyroscope.



```
////Frame
createTorus(frameRad, frameWidth,  sides: 50,  rings: 50,  texture: textureCreator->textures[(textureCreator->selectedGyroIndex)]);

////Bearing 1 Outer
glPushMatrix();
glTranslatef( x: frameRad - frameWidth - 0.2,  y: 0,  z: 0);
glRotatef( angle: 90,  x: 0,  y: 1,  z: 0);
createCylinder( base: 0.1,  top: 0.1,  height: 0.3,  slices: 10,  stacks: 10,  texture: textureCreator->textures[textureCreator->selectedGyroIndex]);
glPopMatrix();

////Bearing 2 Outer
glPushMatrix();
glTranslatef( x: -frameRad + frameWidth,  y: 0,  z: 0);
glRotatef( angle: 90,  x: 0,  y: 1,  z: 0);
createCylinder( base: 0.1,  top: 0.1,  height: 0.3,  slices: 10,  stacks: 10,  texture: textureCreator->textures[textureCreator->selectedGyroIndex]);
glPopMatrix();

////Gimbal1
glPushMatrix();
float gimbal1Rad = frameRad - (frameWidth) * 2 - 0.1;
glRotatef(gimbal1Turning,  x: 1,  y: 0,  z: 0);
createTorus(gimbal1Rad,  width: 0.2,  sides: 50,  rings: 50,  texture: textureCreator->textures[(textureCreator->selectedGyroIndex)]);

////Bearing 3 Inner
glPushMatrix();
glTranslatef( x: 0,  y: gimbal1Rad - frameWidth,  z: 0);
glRotatef( angle: 90,  x: 1,  y: 0,  z: 0);
createCylinder( base: 0.1,  top: 0.1,  height: 0.3,  slices: 10,  stacks: 10,  texture: textureCreator->textures[textureCreator->selectedGyroIndex]);
glPopMatrix();

////Bearing 4 Inner
glPushMatrix();
glTranslatef( x: 0,  y: -gimbal1Rad + frameWidth + 0.2,  z: 0);
glRotatef( angle: 90,  x: 1,  y: 0,  z: 0);
createCylinder( base: 0.1,  top: 0.1,  height: 0.3,  slices: 10,  stacks: 10,  texture: textureCreator->textures[textureCreator->selectedGyroIndex]);
glPopMatrix();

////Gimbal 2
glPushMatrix();
```

Drawing various parts of the gyroscope relative to each other using glPushMatrix and glPopMatrix

Additional features outside of coursework specification

Importing models

After attempting to design some figurines myself, and being dissatisfied by there realism, I decided to import a model to OpenGL. A blender model was found online licensed under the Creative Commons license. To pre-process the model – making it suitable for usage in OpenGL – I triangulated the blender model and exported it as an OBJ file in blender. The OBJ file was exported with a list of vertex points, a list UV coordinates and related texture file information, a list of normals, and a list of faces composed of 3 indices of vertices, 3 indices of UV coordinates and 3 indices of normals. Since the model was triangulated, each face was made up of 3 vertices. The file was opened and looped through. The vertex data was stored in a vector of arrays sized 3, the UV data was stored in a vector of arrays sized 2, and the normal data was stored in a vector of arrays sized 3. Each face is read from the file. A face is composed of 3 points in the form of 9 indices (three points each with one vertex, UV and normal value). The indices of the vertices, UVs, normals are recorded separately. The object file is not necessarily in order meaning all the vertex, UV, normal, and face data must be loaded before the vertices, UVs, and normals can be indexed. Once all the data has been obtained, a loop is created equal to the number of faces.

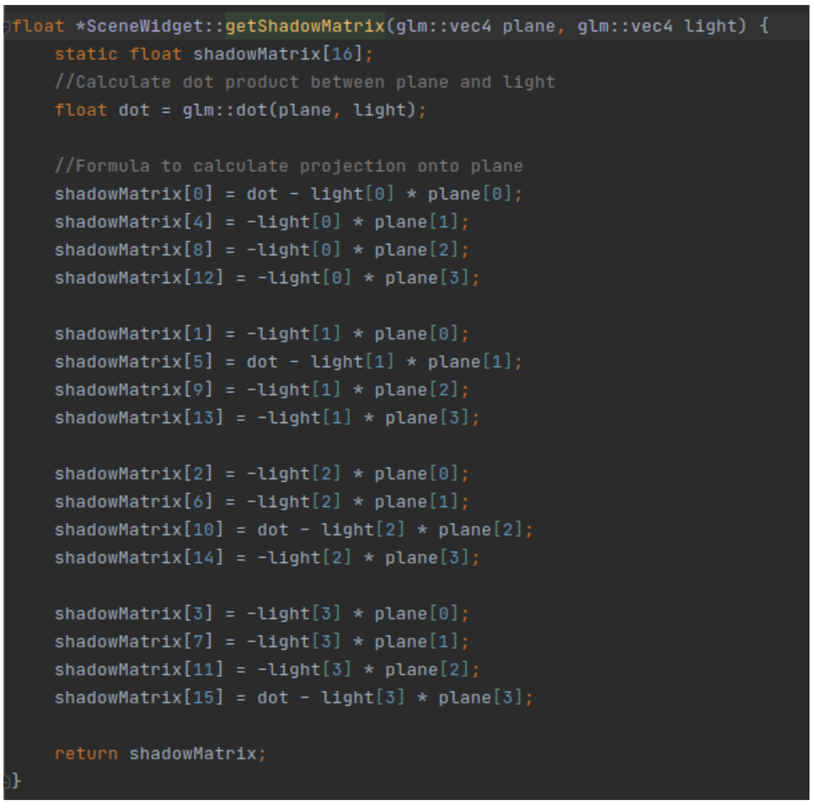
Version control

The version control software, git, was used to manage the progress of this project. Git also allowed to work across different machines easily and ensure that none of my progress was lost. A link to the repository can be found here.

Procedural Generation

As mentioned earlier, the terrain is generated from a seed. Additionally, the tree heights and sizes of the crowns are generated using some element of randomness. This aspect allows the terrain and tree to be generated procedurally and allows for more diversity to be generated by the program rather than me.

Shadows
To draw the shadows a matrix projection is used to calculate how the shadows should be projected onto a plane. A plane can be expressed in the form ax + by + cz + d = 0. A lights position is expressed in homogenous coordinates meaning the two can be dotted together to obtain the dot product. This can then be used to create a projection matrix:



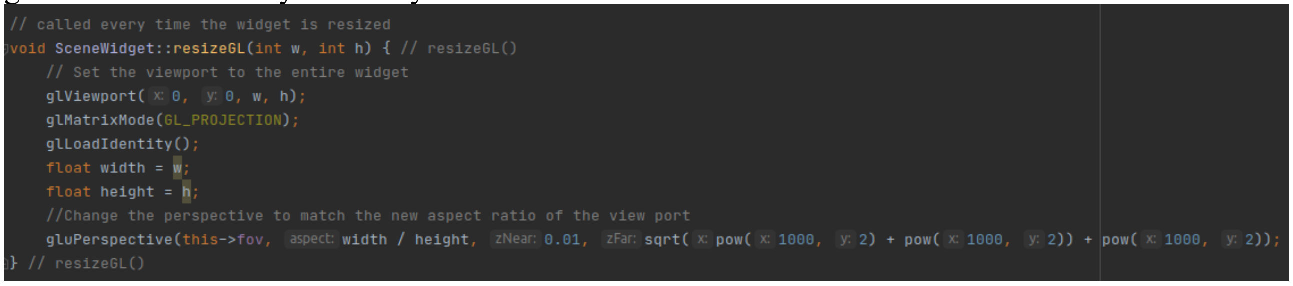
```
float *SceneWidget::getShadowMatrix(glm::vec4 plane, glm::vec4 light) {
    static float shadowMatrix[16];
    //Calculate dot product between plane and light
    float dot = glm::dot(plane, light);

    //Formula to calculate projection onto plane
    shadowMatrix[0] = dot - light[0] * plane[0];
    shadowMatrix[4] = -light[0] * plane[1];
    shadowMatrix[8] = -light[0] * plane[2];
    shadowMatrix[12] = -light[0] * plane[3];

    shadowMatrix[1] = -light[1] * plane[0];
    shadowMatrix[5] = dot - light[1] * plane[1];
    shadowMatrix[9] = -light[1] * plane[2];
    shadowMatrix[13] = -light[1] * plane[3];

    shadowMatrix[2] = -light[2] * plane[0];
    shadowMatrix[6] = -light[2] * plane[1];
    shadowMatrix[10] = dot - light[2] * plane[2];
    shadowMatrix[14] = -light[2] * plane[3];

    shadowMatrix[3] = -light[3] * plane[0];
    shadowMatrix[7] = -light[3] * plane[1];
    shadowMatrix[11] = -light[3] * plane[2];
    shadowMatrix[15] = dot - light[3] * plane[3];

    return shadowMatrix;
}
```

This will project the object draw after the projection matrix has been applied from the perceptive of the light’s position onto the plane.

Blending
As mentioned before, some images were PNG file which allow for transparency to be accounted for. In facilitate transparency in OpenGL blending must be enabled. This was done to allow the background of the fire texture to be seen through. Additionally, the transparent objects were drawn last to the blending worked correctly.

Perspective

glPerspective was used to define the viewing frustum. A fixed field of view was set to 60, with the aspect ratio of the frustum being equal to that of the ratio of the aspect ratio of the viewing window. This way consistent perspective is maintain regardless of if the window is resized. A small value (0.01) was set to as the near clipping value and sqrt(1000^2+1000^2+1000^2) was set as the far clipping plane. This way the entire sky box – which size is 1000 x 1000 x 1000 – and all objects that are greater than 0.01 away are always in the frustum.

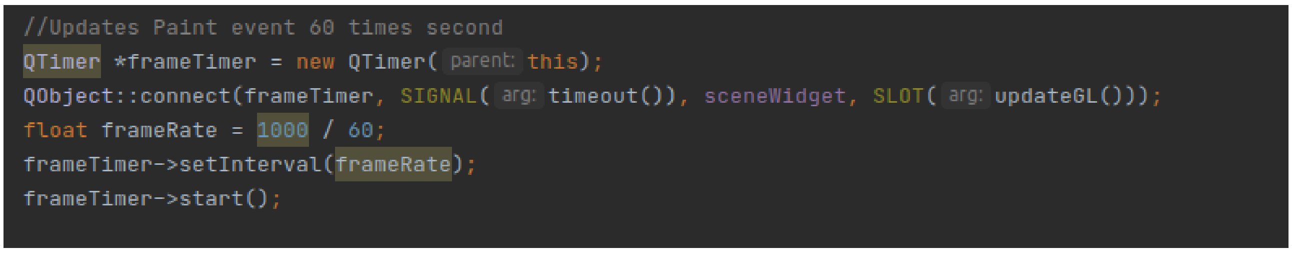


```
// called every time the widget is resized
void SceneWidget::resizeGL(int w, int h) { // resizeGL()
    // Set the viewport to the entire widget
    glViewport( x: 0,  y: 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    float width = w;
    float height = h;
    //Change the perspective to match the new aspect ratio of the view port
    gluPerspective(this->fov,  aspect: width / height,  zNear: 0.01,  zFar: sqrt( x: pow( x: 1000,  y: 2) + pow( x: 1000,  y: 2)) + pow( x: 1000,  y: 2));
} // resizeGL()
```

The resizing event changing the aspect ratio of the viewing frustum

Frame Rates

A QTimer was used to cap the maximum number of frames at 60 frames per a second (fps). This was done by setting a timer to call updateGL() at 1/60 second intervals. On my personal computer 27 fps is averaged however on DEC10 computers roughly 10 fps is achieved. The fps directly correlates with the computational power of a computer. A video of the program running on my personal computer is provided it demonstrates its performance at a higher frame rate. With this being said there is little difference in functionality between running it on DEC10 and on my personal PC but most updates of variables are applied upon frame updates so the program will run proportionally slower.



```
//Updates Paint event 60 times second
QTimer *frameTimer = new QTimer( parent: this);
QObject::connect(frameTimer, SIGNAL( arg: timeout()), sceneWidget, SLOT( arg: updateGL()));
float frameRate = 1000 / 60;
frameTimer->setInterval(frameRate);
frameTimer->start();
```

Implementation of fps capping

Notes for Compiling on DEC10

Qt 5.12.8 was used on my personal computer. As DEC10 does not have this version installed it is preferable to compile using Qt 5.3.1 or Qt 5.13.0 by opening the terminal and running:

```
$ module add qt
$ qmake
$ make clean
$ make
$ ./Scene
```

On my PC version 4.6 of OpenGL was used, however no functionality from OpenGL >3 was used. This means that the program can be compiled easily on DEC10, which uses OpenGL 2.