Computer Graphics
Coursework 2 Report

Word count: 2048

Contents

Overview of the Visual Scene

My visual scene is a geometric, three-dimensional reimaging of the Kandinsky's composition 8 (Figure 1) through the reconstruction of the left half (Figure 2). This report will detail how I constructed my visual scene (Figure 3), highlighting how each element was implemented and how I reimagined and animated Kandinsky's abstract art within an interactive three-dimensional environment.

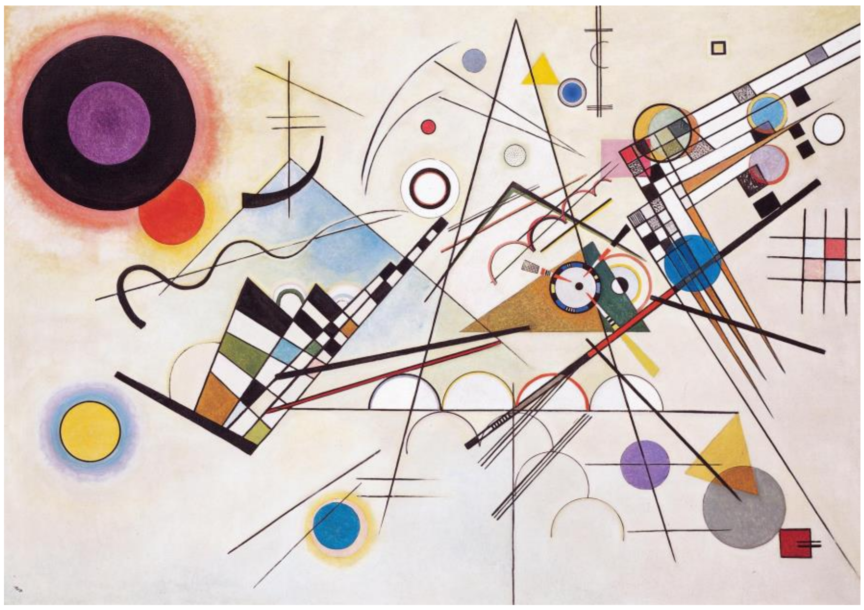


Figure 1 Kandinsky's Composition 8

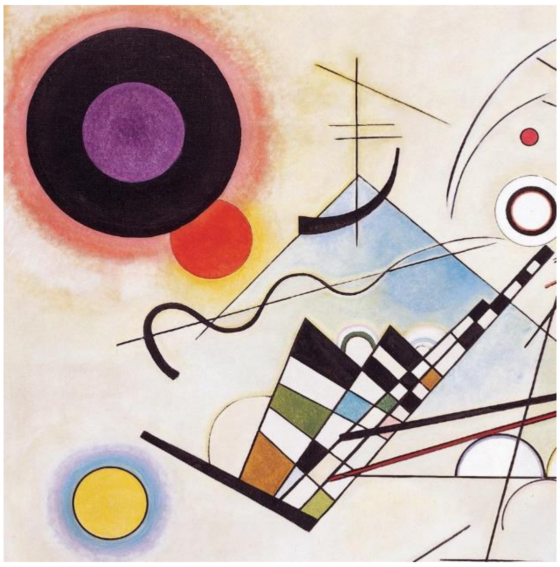


Figure 2 Left half of Kandinsky's Composition 8

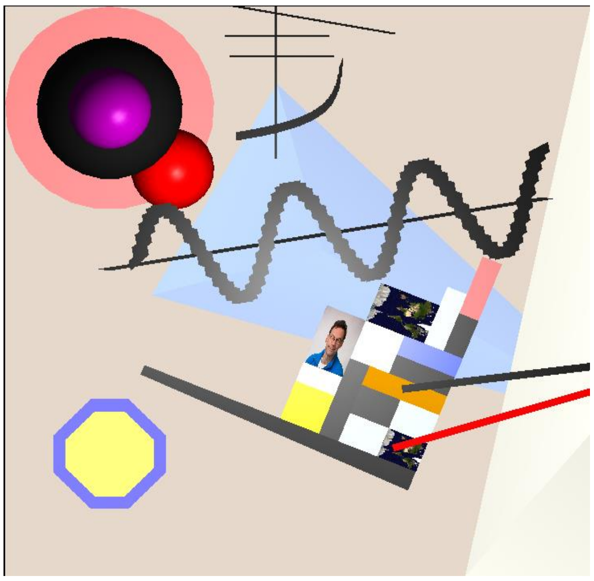
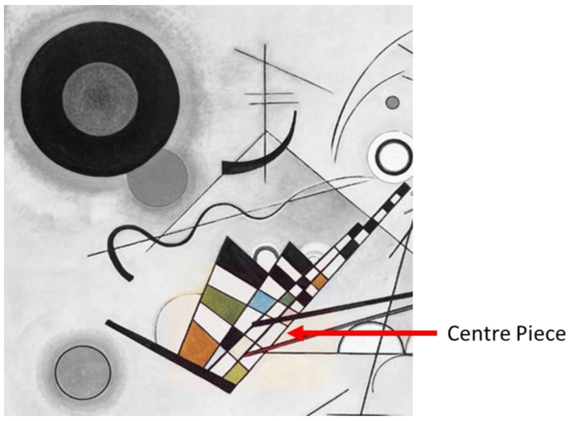


Figure 3 My reimagining of the left half of Kandinsky's composition 8

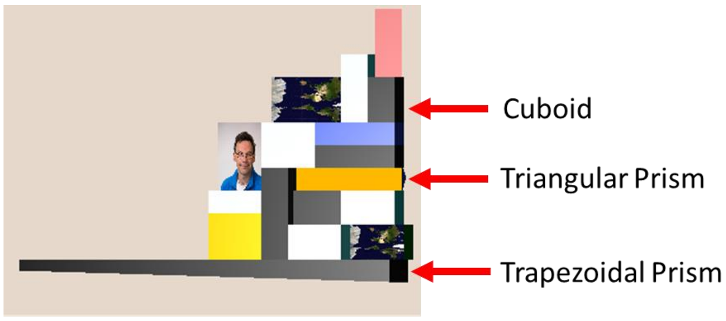
Construction of the Scene

The hierarchical modelling of the centre piece

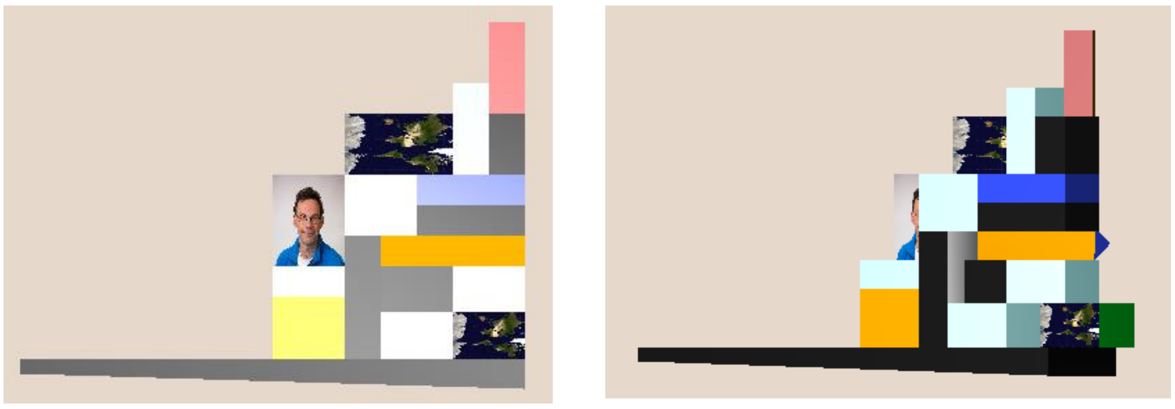


I identified the central object as the centre piece. It consists of a series of quadratics arranged to create a series of steps ending with a slim tip. The quadratics display a range of colours with the most prominent being white and black. In my scene this centre piece was recreated using a series of three-dimensional shapes that have a rectangular or square face, such as cuboids and triangular prisms, providing the object with a more rigid and strict impression. The base, contrastingly, abandons this symmetrical approach through the utilisation of a trapezoidal prism.

The centre piece was then hierarchical modelled using these three shapes. Each shape was manually made using GL_POLYGONS within separate methods. This allowed the shapes to then be called multiple times and individually translated, scaled and rotated to create the holistic centre piece object.

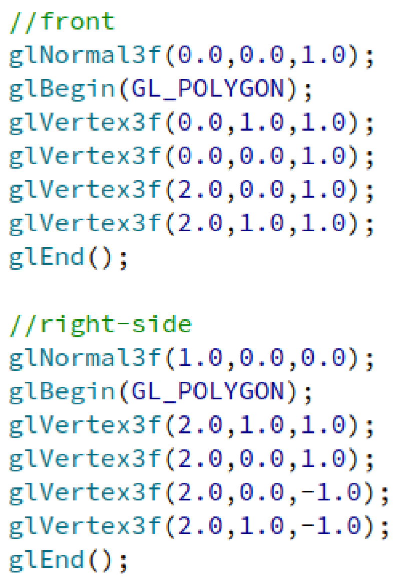


My approach creates a similar image to Kandinsky when viewed head on, however, by providing the different shapes different 'z' coordinates a unique silhouette is formed when viewed at different positions on the x-axis, illustrating my object's three dimensional qualities.



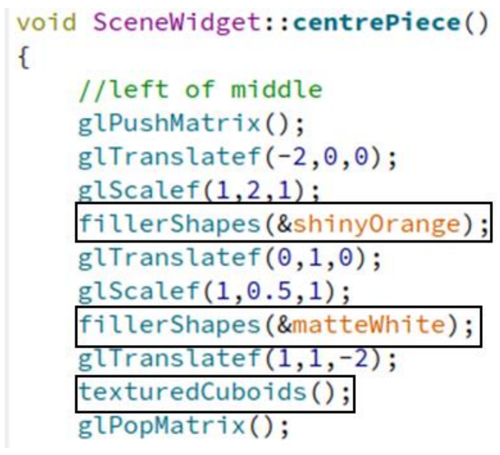
Cuboids

The cuboids were subdivided to serve two purposes, the first were textured cuboids, these were developed to display textures and facilitate some form of animation as outlined later. The second set of cuboids were labelled filler shapes, as they created the centre piece image but did not provide any further animation or purpose. However, their construction is fundamentally identical. They consist of 6 faces, each constructed using GL_POLYGON. As the faces were perpendicular to axis the normal for each face only had to then be a vector parallel to that axis. An example of this is outlined below.



```
//front
glNormal3f(0.0,0.0,1.0);
glBegin(GL_POLYGON);
glVertex3f(0.0,1.0,1.0);
glVertex3f(0.0,0.0,1.0);
glVertex3f(2.0,0.0,1.0);
glVertex3f(2.0,1.0,1.0);
glEnd();

//right-side
glNormal3f(1.0,0.0,0.0);
glBegin(GL_POLYGON);
glVertex3f(2.0,1.0,1.0);
glVertex3f(2.0,0.0,1.0);
glVertex3f(2.0,0.0,-1.0);
glVertex3f(2.0,1.0,-1.0);
glEnd();
```

Within the construction of the centre piece, these cuboids were called and transformed to create various shapes and incorporate different colours as seen below.

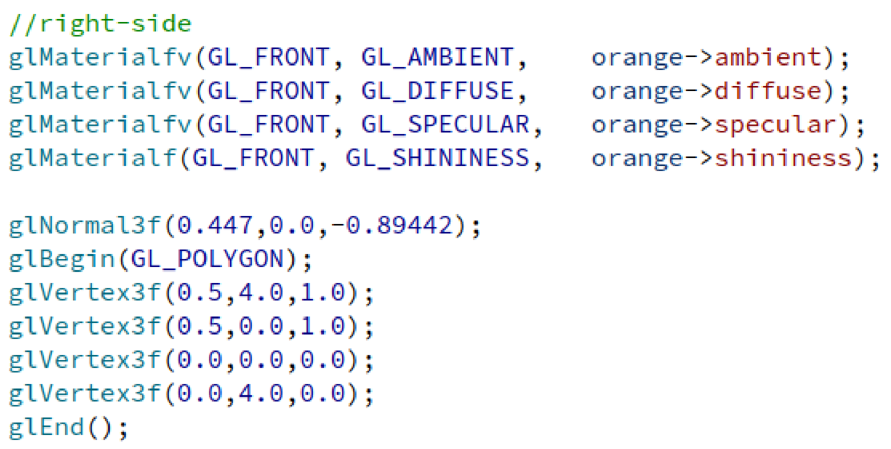


```
void SceneWidget::centrePiece()
{
    //left of middle
    glPushMatrix();
    glTranslatef(-2,0,0);
    glScalef(1,2,1);
    fillerShapes(&shinyOrange);
    glTranslatef(0,1,0);
    glScalef(1,0.5,1);
    fillerShapes(&matteWhite);
    glTranslatef(1,1,-2);
    texturedCuboids();
    glPopMatrix();
```

Triangular Prism

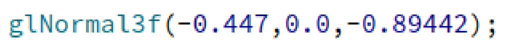
The triangular prism, comparatively, allowed me to maintain the quadratic shape while facilitating different animation possibilities compared to cuboids which I expand upon under ‘elements of animation’.

The triangular prism was constructed using GL_POLYGONS consisting of 5 faces, whereby each of the rectangular faces was developed to have a different material. This shape, however, provided greater complexity when compared to cuboids as due to its triangular base not all of its faces were perpendicular to an axis. As such, the normal for those faces was calculated to be perpendicular to a vector traveling the width of the face. This is outlined in the code extract below.



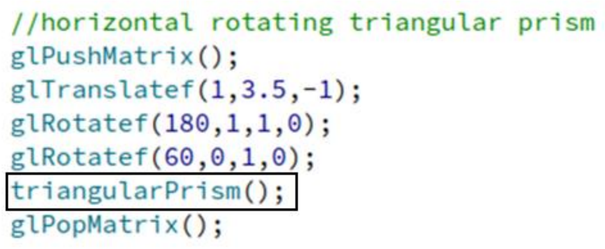
```
//right-side
glMaterialfv(GL_FRONT, GL_AMBIENT,    orange->ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE,    orange->diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR,   orange->specular);
glMaterialf(GL_FRONT, GL_SHININESS,   orange->shininess);

glNormal3f(0.447,0.0,-0.89442);
glBegin(GL_POLYGON);
glVertex3f(0.5,4.0,1.0);
glVertex3f(0.5,0.0,1.0);
glVertex3f(0.0,0.0,0.0);
glVertex3f(0.0,4.0,0.0);
glEnd();
```

When calculating the normal of the left side I simply changed the sign of the vector's x component.



```
glNormal3f(-0.447,0.0,-0.89442);
```

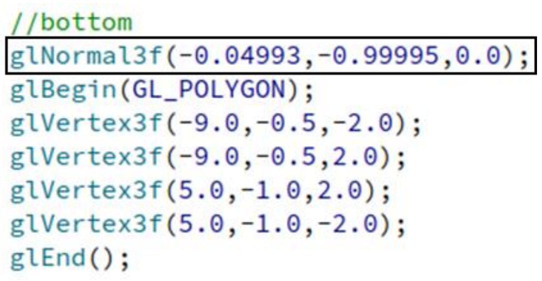
Similar to the cuboids the triangular prisms were transformed to serve different purposes within the centre piece.



```
//horizontal rotating triangular prism
glPushMatrix();
glTranslatef(1,3.5,-1);
glRotatef(180,1,1,0);
glRotatef(60,0,1,0);
triangularPrism();
glPopMatrix();
```

Trapezoidal Prism

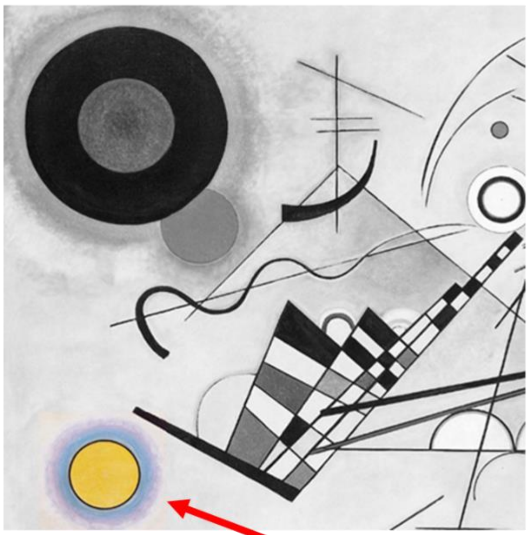
The trapezoidal prism was created using GL_POLYGON, with an approach similar to the cuboids. The shapes only diverge in one aspect and that is the trapezoids slanted side. This necessitated the subsequent face to have a normal using a vector traveling at an angle.

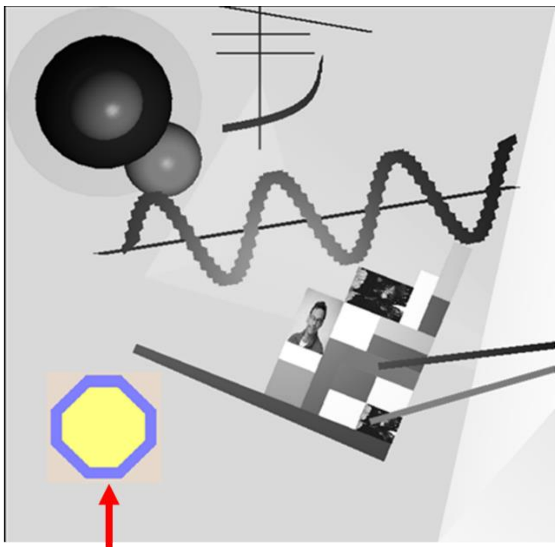


```
//bottom
glNormal3f(-0.04993,-0.99995,0.0);
glBegin(GL_POLYGON);
glVertex3f(-9.0,-0.5,-2.0);
glVertex3f(-9.0,-0.5,2.0);
glVertex3f(5.0,-1.0,2.0);
glVertex3f(5.0,-1.0,-2.0);
glEnd();
```

Constructed convex shapes

To incorporate a convex shape within my scene I reimagined the bottom left circles as an octagonal prism. I felt the symmetry of an octagon would provide similar sense of balance to the scene as the circle did in Kandinsky's composition.





Circle was reimagined as an octagon

The object was constructed using GL_POLYGON and consists of two instances of the octagonal prism shapes layered over one another. The only complexity when constructing the shape was calculating the normal for its angled sides. This resulted in a normal of (-0.5547, -0.5547, 0.0) for its bottom left angled side, comparatively, to calculate the normal of the other sides I manipulated the sign of the x and y components of the vector accordingly.

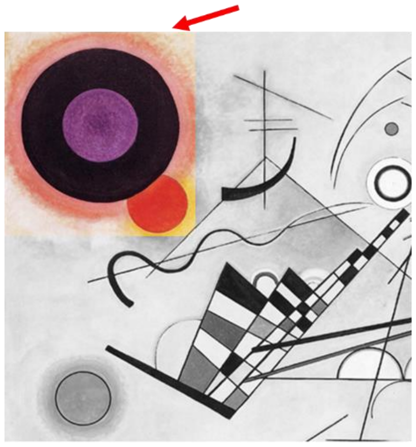
```
//upper right
glNormal3f(0.5547,0.5547,0);
glBegin(GL_POLYGON);
glVertex3f(3.0,4.0,1.0);
glVertex3f(3.0,4.0,0.0);
glVertex3f(1.0,6.0,0.0);
glVertex3f(1.0,6.0,1.0);
glEnd();
```

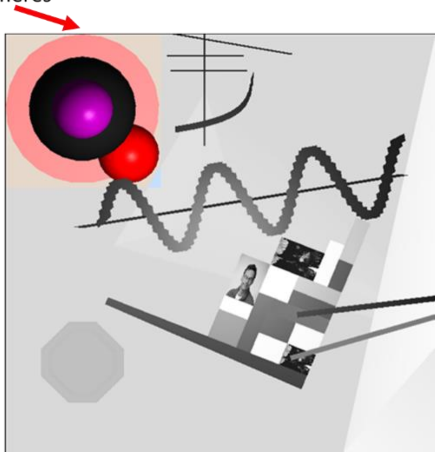
The final layered object was then created using scaling and translations.

```
//octagons
glPushMatrix();
glTranslatef(-9,-8,8);
glScalef(0.9,0.9,0.9);
octagon(&matteBlue);
glPopMatrix();

glPushMatrix();
glTranslatef(-9,-8,9);
glScalef(0.7,0.7,0.7);
octagon(&matteYellow);
glPopMatrix();
```

Application of gluSphere

The circles were reimagined as spheres





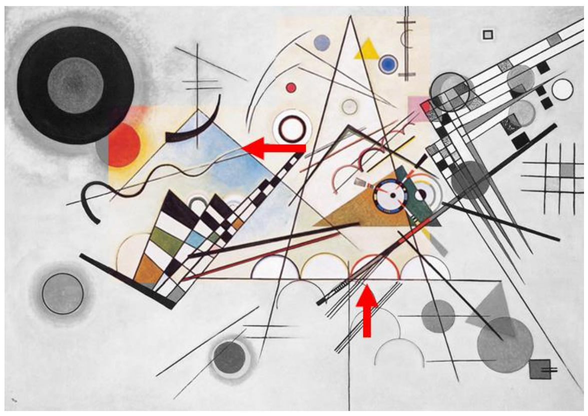
The top left quadrant of Kandinsky's composition 8 presented a series of circles which I reimagined as spheres. To construct these spheres, I utilised glut objects. The glut library handles the calculation of vertices and their normal in a manner far more efficient than what I could implement when modelling spheres resulting in their use in this situation. When constructing the spheres, I used the gluSphere method which allowed me to outline the sphere's radius and the number of subdivisions along and around the z-axis which influences how spherical the sphere appears.

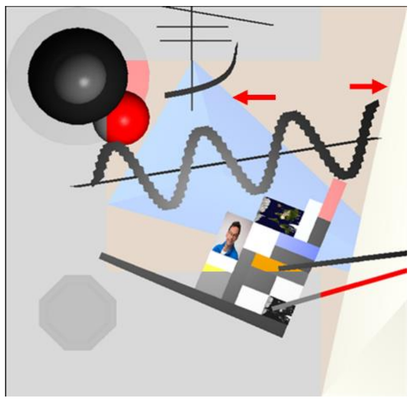
Black, purple and red instances of the sphere were utilised to create the necessary image.

```
rotatingSpheres(&shinyBlack,3.5,0,0,0);

glPushMatrix();
glRotatef(constantRotationAntiClockwise,0,1,0);
rotatingSpheres(&shinyPurple,2,0,0,6);
glPopMatrix();

glPushMatrix();
glRotatef(constantRotationClockwise,1,1,0);
rotatingSpheres(&shinyRed,2,3,-3,-3.75);
glPopMatrix();
```

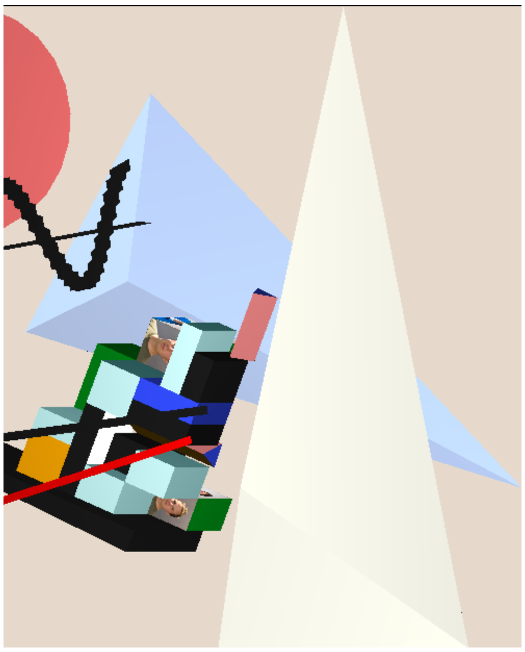
Background elements

Kandinsky's background consisted of a blue and a transparent triangle, which I reimagined as triangular pyramids





The pyramids are most apparent when viewed from different angles as the light and shadows emphasise its three-dimensionality.

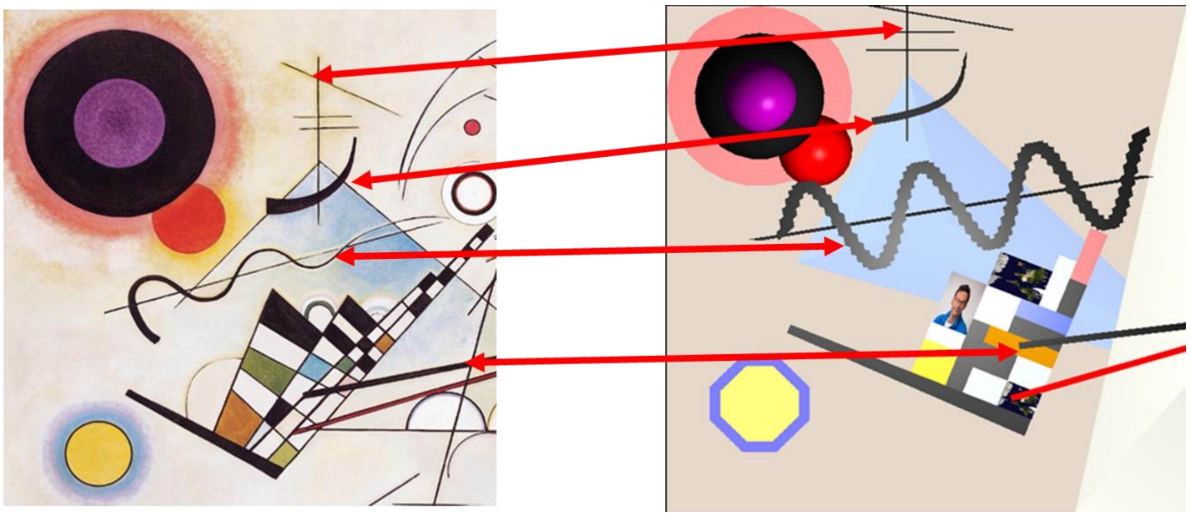


The norm once again had to be calculated as three of the triangle's faces were not perpendicular to any axis.

```
//angled base
glNormal3f(0,0.7,0.7);
glBegin(GL_POLYGON);
glVertex3f(0.0,2.0,2.0);
glVertex3f(-2.0,0.0,0.0);
glVertex3f(2.0,0.0,0.0);
glEnd();

//right
glNormal3f(0.7,0,0.7);
glBegin(GL_POLYGON);
glVertex3f(0.0,8.0,0.0);
glVertex3f(0.0,2.0,2.0);
glVertex3f(2.0,0.0,0.0);
glEnd();

//left
glNormal3f(-0.7,0,0.7);
glBegin(GL_POLYGON);
glVertex3f(0.0,8.0,0.0);
glVertex3f(-2.0,0.0,0.0);
glVertex3f(0.0,2.0,2.0);
glEnd();
```

Foreground elements

The foreground elements consist of a series of slim, flat rectangles which give the illusion of lines, each line was designed to reflect an element of Kandinsky's Composition 8.



In particular, I would like to draw particular attention to the wave I constructed to simulate the shape of the scribble. Similarly, the curve at the base of the 't' shaped lines.

The wave was constructed using a sine wave, emphasising the geometric theme of my reimagining.

```
for (double x=0; x<20; x=x+0.25){
    glNormal3f(0,0,1);
    glBegin(GL_POLYGON);
    double y1=sin(x)*2.5;
    double y2=sin(x+0.25)*2.5;
    glVertex3f(x,y1+0.5,16.);
    glVertex3f(x,y1,16.);
    glVertex3f(x+1,y2,16.);
    glVertex3f(x+1,y2+0.5,16.);
    glEnd();
}
```

Similarly, to create the gradual decrease of the curved lines thickness I constructed it using a logarithmic function.

```
for (double x=0; x<25; x=x+0.25){
    glNormal3f(0,0,1);
    glBegin(GL_POLYGON);
    double y1=log(x);
    double y2=log(x+0.25);
    glVertex3f(x,y1+0.5,16.);
    glVertex3f(x,y1,16.);
    glVertex3f(x+1,y2,16.);
    glVertex3f(x+1,y2+0.5,16.);
    glEnd();
}
```

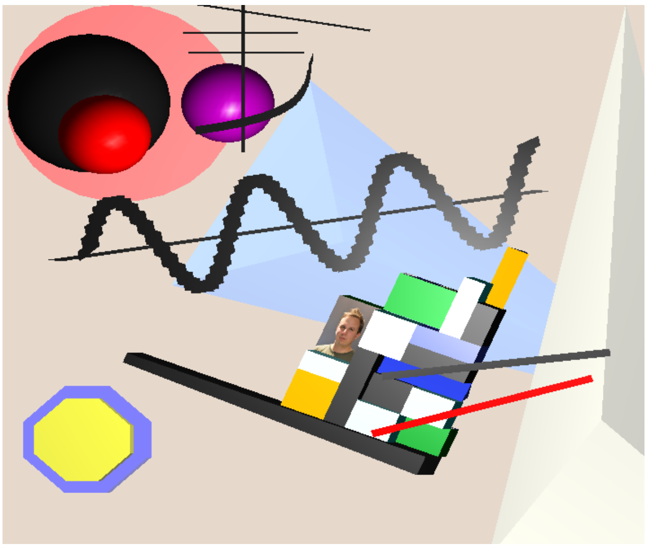
Light and material properties

Lighting was established to focus on the front of the centre piece, the key element of my scene. As such to emphasise the lighting, all materials within my centre piece were highly reflective, which contrasted with the matte materials on the triangular pyramids.

Similarly, due to the shapes being at different depths, some shapes within the centre piece, despite having the same material, contrasted with one another by displaying different hues. This was apparent with the material ‘shinyOrange’; this resulted in the reduction of the material’s specular lighting to 0.1. As before it appeared yellow in one instance and orange in another. However, I avoid eliminating all specular lighting to maintain the shiny effect. Specular lighting was, similarly, reduced on the material ‘shinyBlack’ as it resulted in the black appearing more as a grey, reducing contrast. By comparison, the specular lighting of the purple sphere was greatly increased with the hope that by creating concentrated highlights the objects spherical shape is emphasised.

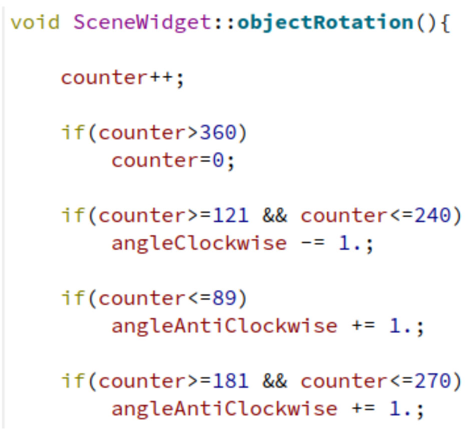
```
static materialStruct shinyBlack = {
                       {0.1f,0.1f,0.1f,1.0f },
                       {0.1f,0.1f,0.1f,1.0f },
                       {0.50f,0.50f,0.50f,0.3f },
                       50.0f};

static materialStruct shinyPurple = {
                       {0.7f,0.0f,0.7f,1.0f },
                       {0.7f,0.0f,0.7f,1.0f },
                       {0.90f,0.90f,0.90f,0.9f },
                       90.0f};
```

The overall impact of lighting is greatly emphasised when viewed at an angle. Specular lighting is made apparent with the centre piece’s various highlights. It is also made apparent by the gradient effect on the wave, resultant from the object moving further away from the light source. The position of the light also creates great contrast with the sides of the shapes as they exhibit considerably darker hues.



Elements of animation

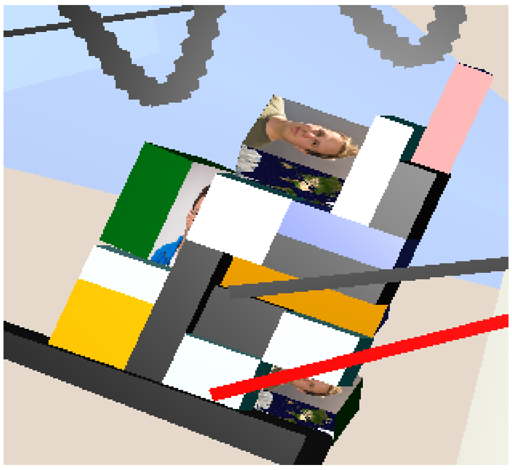
Within the centre piece rotation was implemented as an element of animation. In particular, the triangular prism and the textured cuboids rotated. This was achieved through a timer signal that continuously updated the objects angle of rotation to create the illusion of continuous rotation. The elements rotating incorporated different surfaces to create different visuals when rotating. This allowed for different colour combination and the expression of different surfaces. The rotation itself was adapted to periodically pause to encourage the viewer to appreciate different colour and texture combinations. The method that facilitated this rotation is detailed below.

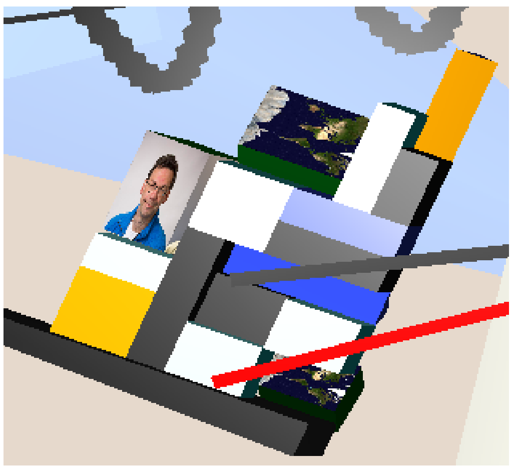


```
void SceneWidget::objectRotation(){

    counter++;

    if(counter>360)
        counter=0;

    if(counter>=121 && counter<=240)
        angleClockwise -= 1.;

    if(counter<=89)
        angleAntiClockwise += 1.;

    if(counter>=181 && counter<=270)
        angleAntiClockwise += 1.;
```

The method had to accommodate both different directions of rotation, as well as degrees of rotation before a new face was shown. For instance, the triangular prism required 30 degrees more than the cuboid for a new face to appear head on.

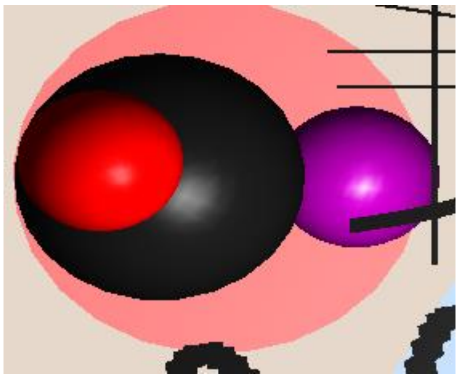
The choice of a triangular prism was motivated by only wanting to display 3 different rotating colours. Comparatively, the cuboid was utilised as it provided the perfect surface area for the textures. When arranging the centre piece, the rotating objects were organised such that different faces start upfront, desynchronising the rotations and creating unique visuals.

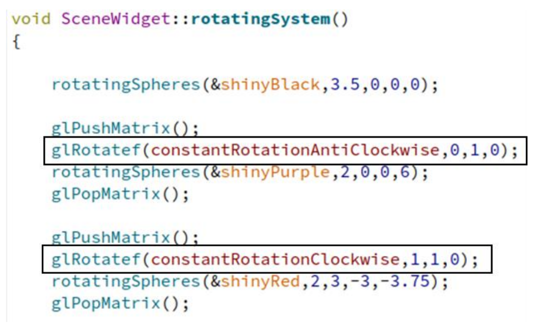




Elements of movement

Movement was realised through the spheres. I envisioned them moving throughout the scene by orbiting the central black sphere. As such, the sphere would move in a circular path, with each sphere traveling at their own prescribed angle and distance from the central sphere. It added dynamism to the scene, that was lacking before. This effect was achieved through a rotation of the orbiting spheres around a central point at the centre of the black sphere.

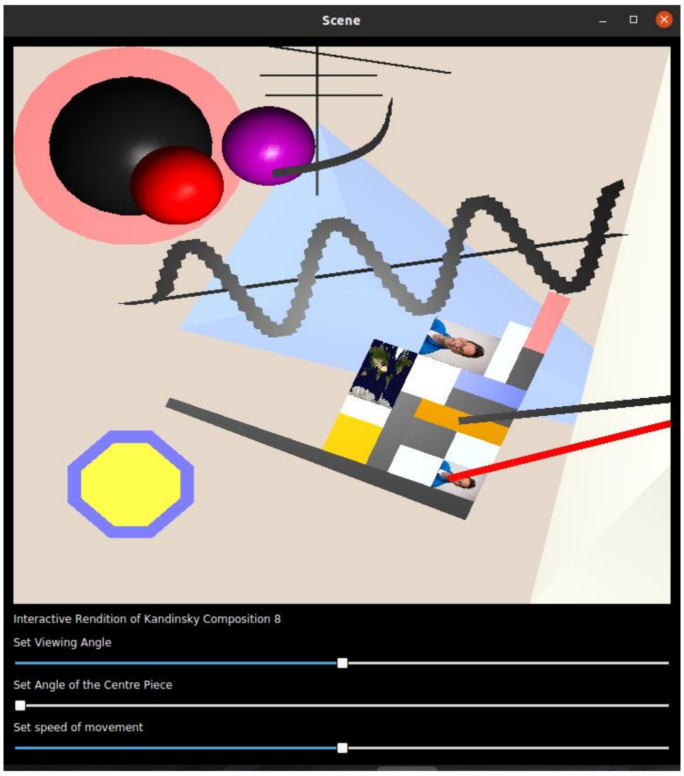




```
void SceneWidget::rotatingSystem()
{

    rotatingSpheres(&shinyBlack,3.5,0,0,0);

    glPushMatrix();
    glRotatef(constantRotationAntiClockwise,0,1,0);
    rotatingSpheres(&shinyPurple,2,0,0,6);
    glPopMatrix();

    glPushMatrix();
    glRotatef(constantRotationClockwise,1,1,0);
    rotatingSpheres(&shinyRed,2,3,-3,-3.75);
    glPopMatrix();
```

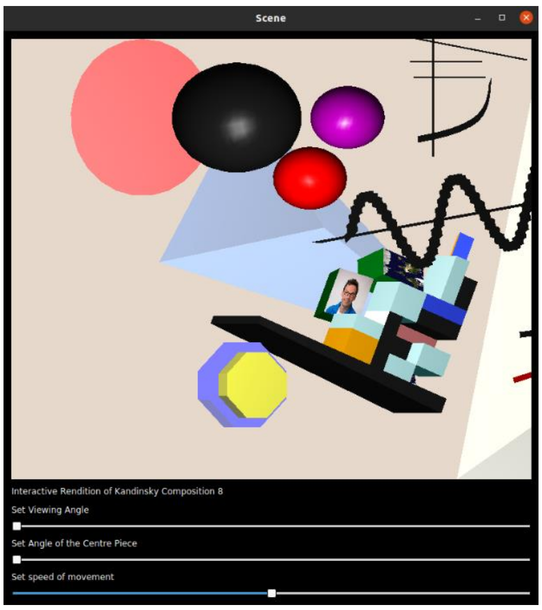
Elements of user interaction

User interaction was facilitated through a series of sliders, each allowing the user user to interact with the scene in a different manner.


Scene
Interactive Rendition of Kandinsky Composition 8
Set Viewing Angle
Set Angle of the Centre Piece
Set speed of movement

Adjustment of viewing angle

Initially the user is presented the scene head on with the x coordinate of gluLookAt at 0. The slider in turn allows the user to manipulate this angle by changing the x coordinates from a range of -10 to 10. This is achieved through a slot that receives the change in value from the slider. In turn it updates the variable representing the x coordinate. This interaction allows the user to better view the three-dimensionality of my scene and reveals intriguing shadows and depth not apparent when viewed head on.

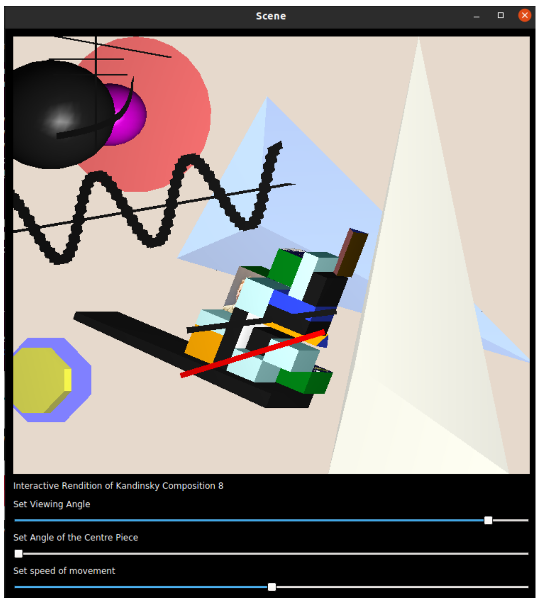


Interactive Rendition of Kandinsky Composition 8

Set Viewing Angle

Set Angle of the Centre Piece

Set speed of movement



Interactive Rendition of Kandinsky Composition 8

Set Viewing Angle

Set Angle of the Centre Piece

Set speed of movement

Signal

```
text[1] = new QLabel();
text[1]->setStyleSheet("QLabel {color : white; }");
text[1]->setText("Set Viewing Angle");
text[1]->setMaximumHeight(20);
windowLayout->addWidget(text[1]);

angle = new QSlider(Qt::Horizontal);
angle ->setRange(-14,14);
angle -> setValue(0);
connect(angle, SIGNAL(valueChanged(int)), scene, SLOT(updateAngle(int)));
windowLayout->addWidget(angle);
```

Slot

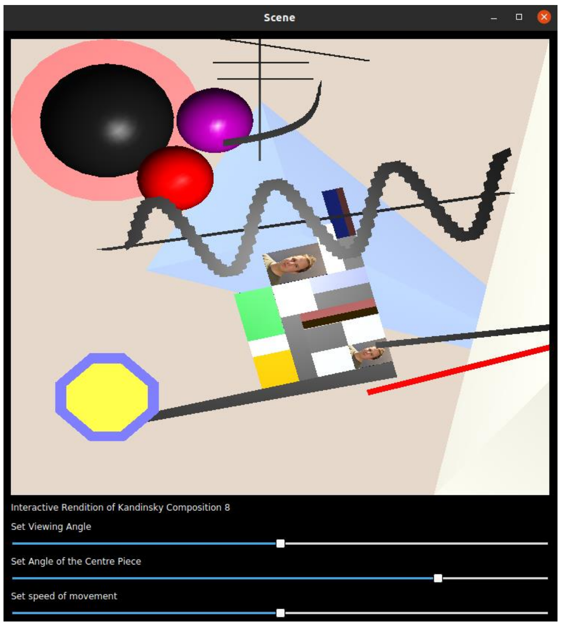
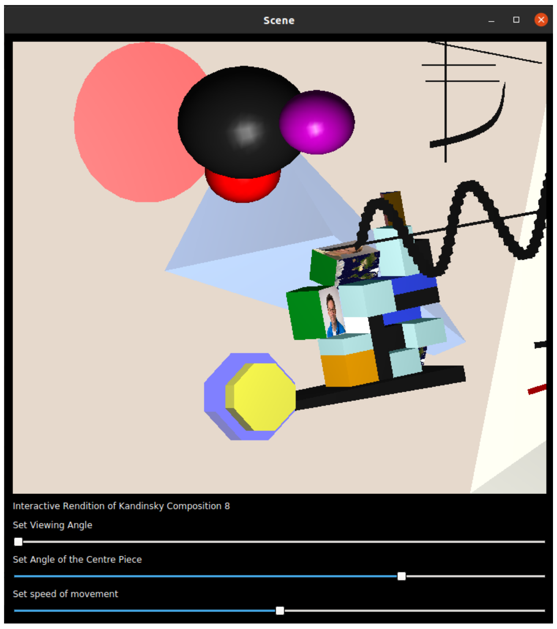
```
void SceneWidget::updateAngle(int input)
{
    viewingAngle = input;
    this->repaint();
}
```

gluLookAt

```
gluLookAt(viewingAngle,0.0,17.0, 0.0,0.0,0.0, 0.0,1.0,0.0);
```

Manipulation of the centre piece

The centre piece itself is located at an angle of -22 degrees to simulate the shape in Kandinsky's composition. In turn this angle is fully adjustable through a slider operating using a similar signal, slot approach. When moving the centre piece, the viewer can better differentiate between foreground and background elements as the centre piece travels through the scene. Moreover, it allows the user to reimagine the scene by reposition the object.


Scene
Interactive Rendition of Kandinsky Composition 8
Set Viewing Angle
Set Angle of the Centre Piece
Set speed of movement

Scene
Interactive Rendition of Kandinsky Composition 8
Set Viewing Angle
Set Angle of the Centre Piece
Set speed of movement

Signal

```
text[2] = new QLabel();
text[2]->setStyleSheet("QLabel {color : white; }");
text[2]->setText("Set Angle of the Centre Piece");
text[2]->setMaximumHeight(20);
windowLayout->addWidget(text[2]);

rotation = new QSlider(Qt::Horizontal);
rotation -> setRange(-22,23);
rotation -> setValue(-22);
connect(rotation, SIGNAL(valueChanged(int)), scene, SLOT(updateRotation(int)));
windowLayout->addWidget(rotation);
```

Slot

```
void SceneWidget::updateRotation(int input)
{
    rotationAngle=input;
    this->repaint();
}
```

How the rotation is implemented

```
//centre piece
glPushMatrix();
glTranslatef(1,-7,6);
glRotatef(rotationAngle,0,0,1);
centrePiece();
glPopMatrix();
```

Control over speed

The speed the spheres are moving, similarly, is fully adjustable with the ability to increase speed as well as reducing the speed to cease movement. This was implemented using a similar slider, signal, slot approach. This is designed to encourage the user to reconsider how simply adjusting movement can influence the serenity of the scene. For an example of this, please refer to my demo as speed cannot be demonstrated using images.

Signal

```
text[3] = new QLabel();
text[3]->setStyleSheet("QLabel {color : white; }");
text[3]->setText("Set speed of movement");
text[3]->setMaximumHeight(20);
windowLayout->addWidget(text[3]);

speed = new QSlider(Qt::Horizontal);
speed ->setRange(0,6);
speed -> setValue(3);
connect(speed, SIGNAL(valueChanged(int)), scene, SLOT(updateSpeed(int)));
windowLayout->addWidget(speed);
```

Slot

```
void SceneWidget::updateSpeed(int input)
{
    rotationSpeed=input;
    this->repaint();
}
```

How the new speed is implemented

```
constantRotationAntiClockwise += rotationSpeed;
constantRotationClockwise -= rotationSpeed;
```

Texture Mapping

To map the provided textures, three things had to be achieved. Firstly, the images had to be loaded and read by the program. Secondly these images then had to be converted into a two-dimensional texture. Thirdly, the texture had to be placed over the necessary surface.

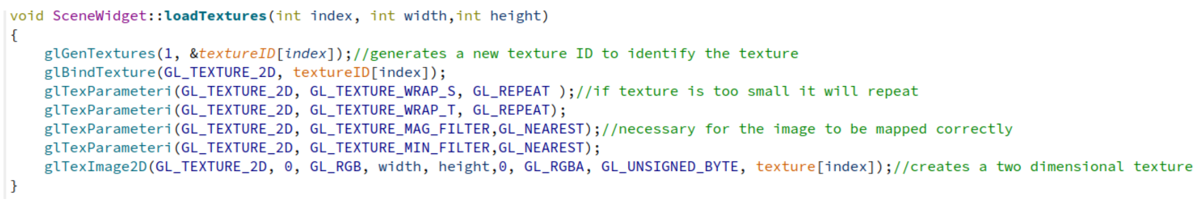
1. Loading the images

The file was read using ifstream and stored using a GluByte array. This allowed me to read the individual RGB values of the ppm image and store them. The size of the array was equivalent to the image's width multiplied by its height multiplied by 4, which is the number of variables necessary to represent each pixel in openGL due to its use of RGBA.

```
int i,j,colour;
for(i=height; i>=0; i--)//loops through the height of the image
{
    for(j=0;j<width;j++)//loops through the width of the image
    {
        //each colour ratio consits of three numbers, rgb, as such for every pixel three numbers are read
        textureFile >> colour;
        texture[index][(i*width+j)*4]=(GLubyte) colour;
        textureFile >> colour;
        texture[index][(i*width+j)*4+1]=(GLubyte) colour;
        textureFile >> colour;
        texture[index][(i*width+j)*4+2]=(GLubyte) colour;
        texture[index][(i*width+j)*4+3]=(GLubyte) 255;//manually includes an alpha chanel missing in ppm for the colours to appear correct
    }
}
textureFile.close();
```

2. Creating the texture

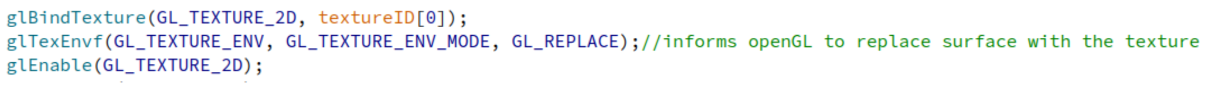
This necessitated the generation of a unique identifier for the texture using glGenTextures. Comparatively, for the image to appear as desired certain parameters had to be outlined using glTexParamteri. Ultimately, the texture was created using glTexImage2D.



```
void SceneWidget::loadTextures(int index, int width,int height)
{
    glGenTextures(1, &textureID[index]);//generates a new texture ID to identify the texture
    glBindTexture(GL_TEXTURE_2D, textureID[index]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );//if texture is too small it will repeat
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,GL_NEAREST);//necessary for the image to be mapped correctly
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height,0, GL_RGBA, GL_UNSIGNED_BYTE, texture[index]);//creates a two dimensional texture
}
```

3. Applying the texture

The texture was then applied using its texture ID.



```
glBindTexture(GL_TEXTURE_2D, textureID[0]);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);//informs openGL to replace surface with the texture
glEnable(GL_TEXTURE_2D);
```