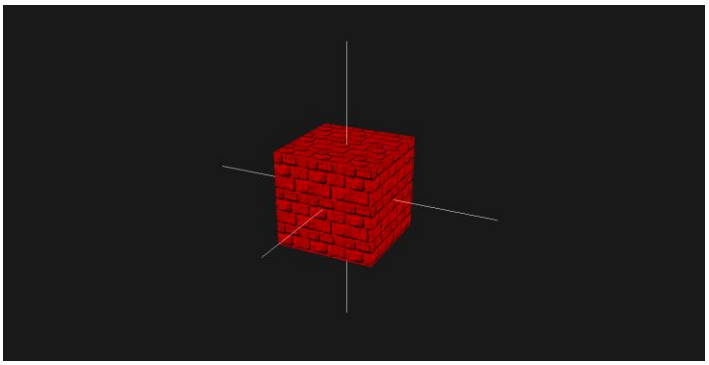
CM20219

Coursework Report

Part 1

Requirement 1 & 2 - Draw a simple cube with coordinate system axes as lines

To draw a cube in OpenGL we need to draw 6 square or 12 triangle faces. We can do this by using the primitives given to us in the OpenGL API. In my code we use the GL_QUAD primitive to draw each faces by assigning 4 vertices of the square face, which OpenGL with connect in the Primitive Assembly part of the fixed-function pipeline that is used in the version of OpenGL in which this code has been written.



```
glBegin(GL_QUADS);
glColor4f(1.0f,0.0f,0.0f,alpha);
glVertex3f( 1.0f, 1.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, 1.0f);
glVertex3f(-1.0f,-1.0f, 1.0f);
glVertex3f( 1.0f,-1.0f, 1.0f);
```

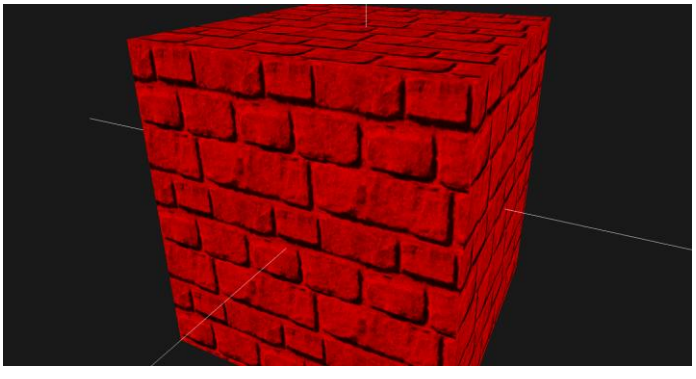
As I don't store the coordinates of the cube in an array , it means that it complicates the matrix transformations that I need to apply to the cube for the latter requirements.

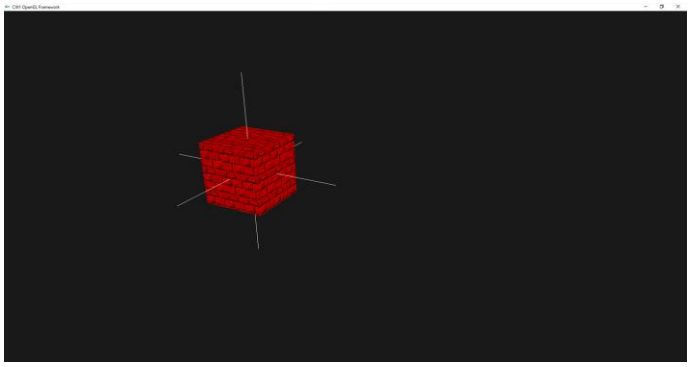
To draw the coordinate axes we use the GL_LINES primitive which need two points to connect.

```
glBegin(GL_LINES);
alpha = 1.0f;
glColor4f(1.0f, 1.0f, 1.0f,alpha);
//X-axis
glVertex3f(-3.0f,0.0f,0.0f);
glVertex3f(3.0f, 0.0f, 0.0f);
//Y-axis
glVertex3f(0.0f, -3.0f, 0.0f);
glVertex3f(0.0f, 3.0f, 0.0f);
```

We need to draw these axes in a different stack of the Model View Matrix so the axes aren't rotated in Req 4 at the same time as the cube. We can change the current stack of the Model View Matrix by using the functions glPushMatrix(); and glPopMatrix(); , which come in handy when we only want to effect certain parts of the objects we draw.

Requirement 3 - Rotate, zoom, and pan the camera





To do any camera manipulation we need to change edit the variables in the gluLookAt function.

```
gluLookAt(eye[0], eye[1], eye[2],
          center[0], center[1], center[2],
          up[0], up[1], up[2]);
```

This function creates a 5x5 viewing matrix from the following 3 vectors, Eye , Center and Up. The eye vector represents the point in the global reference, where you want to be looking from. The center vector represents the points at which the camera is focusing towards, and finally the up V=vector determines the "up" direction of the camera, it used so we can to create a cross product between the eye and center vectors.

$$M = \begin{bmatrix} s & 0 & s & 1 & s \\ 2 & 0 & u & 0 & u \\ 1 & u & 2 & 0 & -f \\ 0 & -f & 1 & -f & 2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

f =(centerX - eyeX centerY - eyeY centerZ - eyeZ)^2

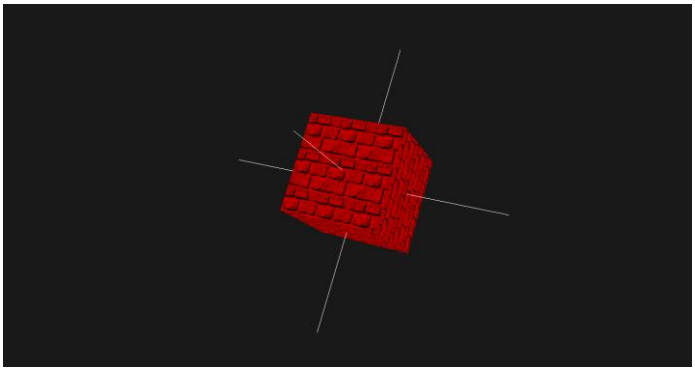
s = f * UP^2

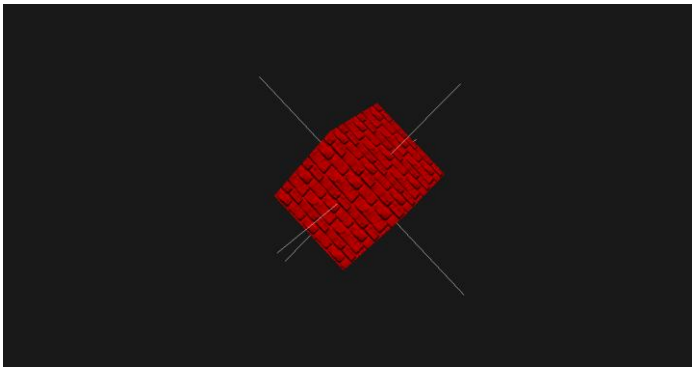
u = ss x f

With this matrix we can get the same result as gluLookAt by using.

```
glMultMatrixf(M);
glTranslated(-eyex, -eyey, -eyez);
```

Rotating the camera is standard matrix multiplication between rotational matrices and the coordinates of the camera. We Also need to do the same to the Up vector of the camera so it looks like the cube and axis are rotating and to avoid odd looking results.





The Rotational matrices we use to rotate the Camera are the following Euler Angle Matrices

$$R_x(\vartheta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\vartheta) & -sin(\vartheta) \\ 0 & sin(\vartheta) & cos(\vartheta) \end{bmatrix} \quad R_y(\vartheta) = \begin{bmatrix} cos(\vartheta) & 0 & sin(\vartheta) \\ 0 & 1 & 0 \\ -sin(\vartheta) & 0 & cos(\vartheta) \end{bmatrix} \quad R_z(\vartheta) = \begin{bmatrix} cos(\vartheta) & -sin(\vartheta) & 0 \\ sin(\vartheta) & cos(\vartheta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

a consequence of using these Euler angles to rotating the camera in this manner, is that it can cause Gimbal Lock, which is when the we lose one of the degrees of motion in a 3D space it occurs when 2 of the axes are parallel to each other causing them to lock.

```
tempEx = eye[0] * Rx[0][0] + eye[1] * Rx[0][1] + eye[2] * Rx[0][2] + h*Rx[0][3];
tempEy = eye[0] * Rx[1][0] + eye[1] * Rx[1][1] + eye[2] * Rx[1][2] + h*Rx[1][3];
tempEz = eye[0] * Rx[2][0] + eye[1] * Rx[2][1] + eye[2] * Rx[2][2] + h*Rx[2][3];
h = eye[0] * Rx[3][0] + eye[1] * Rx[3][1] + eye[2] * Rx[3][2] + h*Rx[3][3];

tempx = up[0] * Rx[0][0] + up[1] * Rx[0][1] + up[2] * Rx[0][2] + up[3] * Rx[0][3];
tempy = up[0] * Rx[1][0] + up[1] * Rx[1][1] + up[2] * Rx[1][2] + up[3] * Rx[1][3];
tempz = up[0] * Rx[2][0] + up[1] * Rx[2][1] + up[2] * Rx[2][2] + up[3] * Rx[2][3];
tempw = up[0] * Rx[3][0] + up[1] * Rx[3][1] + up[2] * Rx[3][2] + up[3] * Rx[3][3];
```

The code above is the calculation of the new camera position by Rotating it by ϑ around the X-axis. the *h/tempw* variable is the homogeneous variable used to make sure the vector used to represent the camera position stays visible when we set the camera position. Before we set the Camera position we need to divide by this variable to make sure it is 1 in the vector , as show below.

```
eye[0] = tempEx / h;
eye[1] = tempEy / h;
eye[2] = tempEz / h;

up[0] = tempx / tempw;
up[1] = tempy / tempw;
up[2] = tempz / tempw;
up[3] = tempw / tempw;
```

To zoom the camera toward and away from the cube we need to increase and decrease the camera coordinates respectfully to themselves.

```
eye[0]= eye[0]/1.01f;
eye[1]= eye[1]/1.01f;
eye[2]= eye[2]/1.01f;

eye[0]=eye[0] * 1.01f;
eye[1]=eye[1] * 1.01f;
eye[2]=eye[2] * 1.01f;
```

This causes the camera to move to the camera closer or further away from the cube when needed, along the normal vector of the camera.

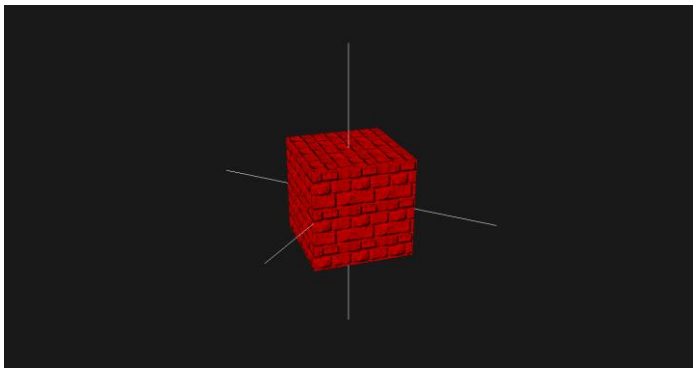
To pan the camera along the x-axis all we need to do is the change the variable representing the x coordinate of the, eye and center positions of the camera at the same rate.

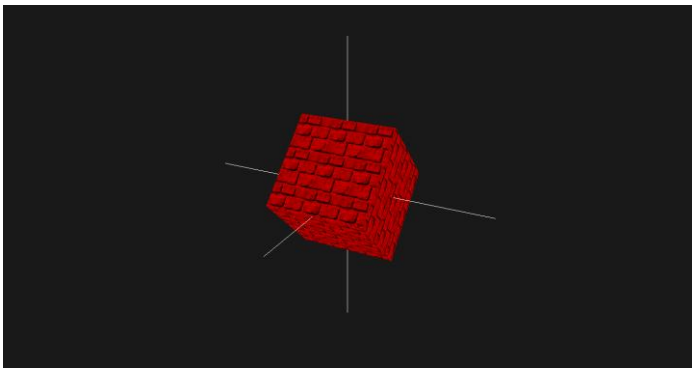
```
//pan right
eye[0] = eye[0] + 0.1f;
center[0] = center[0] + 0.1f;

//pan left
eye[0] = eye[0] - 0.1f;
center[0] = center[0] - 0.1f;
```

Given extra time on this requirement I would implement a way to move the camera along the vector perpendicular to the vector point to the center of the cube, as this would give you a greater degree of motion in the camera alternatively it could be possible to pan along both the z and y axis's.

<u>Requirement 4 - Rotate cube</u>





To rotate cube we use the glRotate function offered by OpenGL, this function takes an angle and a vector as parameters.

It rotates all the points in the current stack by the angle given around the point.

It does this by forming a rotational matrix like the rotational matrices used in Req 3, and multiplies it against all the vector positions defined in the current stack of the ModelView Matrix . In my code we use glRotate() three times to rotate about the x , y and z axis individually .

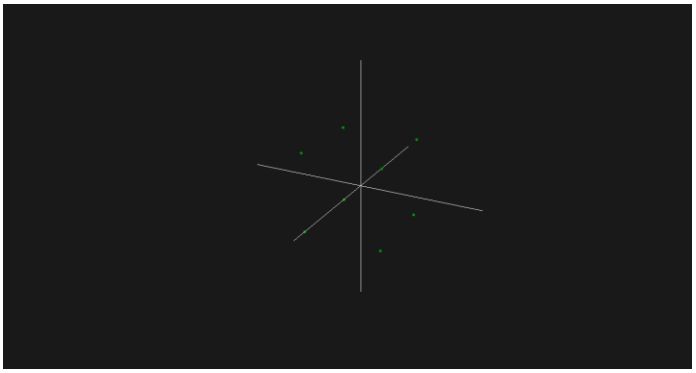
```
glRotatef(RotateY, 0.0f, 1.0f, 0.0f);
glRotatef(RotateX, 1.0f, 0.0f, 0.0f);
glRotatef(RotateZ, 0.0f, 0.0f, 1.0f);
```

Requirement 5 - Different render modes

Drawing the cube as a combination of point and line is a simple task compared to a Phong lighting model.

To draw the cube as a combination of 8 points we use glBegin(GL_POINTS) which draws single points at the vector position given. We use Vector3f to define the 8 vertices of the cube.

```
 glBegin(GL_POINTS);
glColor4f(0.0f,1.0f,0.0f,alpha);
glVertex3f( 1.0f, 1.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, 1.0f);
glVertex3f(-1.0f,-1.0f, 1.0f);
glVertex3f( 1.0f,-1.0f, 1.0f);
glVertex3f(1.0f, 1.0f, -1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);
glVertex3f(1.0f, -1.0f, -1.0f);
glEnd();
```



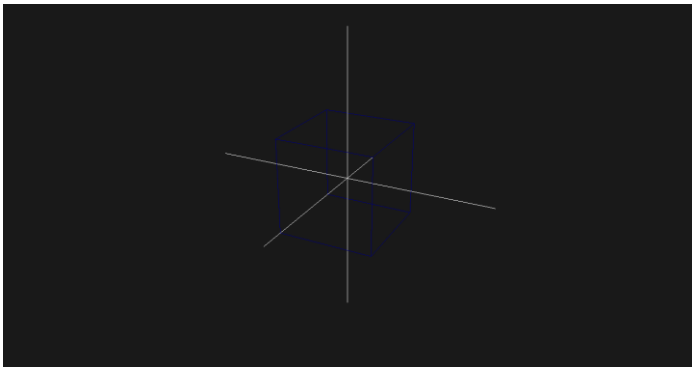
Drawing the cube as lines is every similar but we use GL_LINES instead to draw lines between two defined points, we do this to draw 4 lines between the 4 points of each face of the cube.

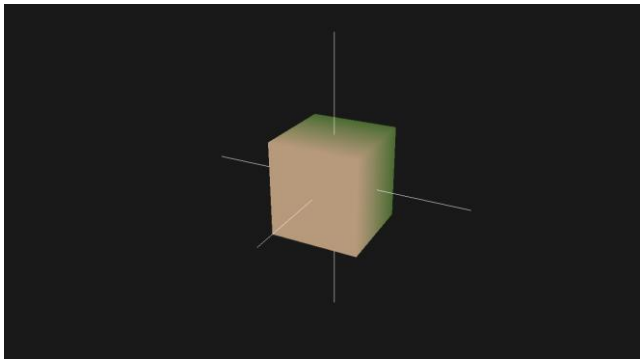
```
glBegin(GL_LINES);
//Front
glVertex3f( -1.0f, 1.0f,1.0f);
glVertex3f( -1.0f, -1.0f,1.0f);

glVertex3f( -1.0f, 1.0f,1.0f);
glVertex3f( 1.0f, 1.0f,1.0f);

glVertex3f( -1.0f, -1.0f,1.0f);
glVertex3f( 1.0f, -1.0f,1.0f);

glVertex3f( 1.0f, -1.0f,1.0f);
glVertex3f( 1.0f, 1.0f,1.0f);
```





Lighting in OpenGL uses the Phong Illumination Model to represent the reflection of lighting on models.

Phong Illumination combines the diffuse Lambertian, specular and ambient reflection of light and object. We end up using the equation

$$I = k_a i_a + k_d i_d(\mathbf{L} \cdot \mathbf{N}) + k_s i_s(\mathbf{R} \cdot \mathbf{V})^{\alpha}$$

to calculate the reflection effect. Where k_a,k_d,k_s are the reflective values corresponding for the ambient, diffuse and specular colours of the object, i_a,i_d,i_s are the values corresponding to the ambient, diffuse and specular colour of the light and α is the shininess of material. L is the light source vector, N is the Normal to the object ,R is the Reflected vector and, V is the Viewing Ray.

To do this is OpenGL need to define either face normal's for flat per face lighting or vertex normal's to calculate smooth shading, in my code i have done the latter as it produces a nicer effect. By doing this we are technically estimating the cube as a sphere, which gets better results depending on the number of faces in the 3d polygon drawn.

```
//Front
glBegin(GL_QUADS);
glNormal3f(1.0f, 0.0f, 1.0f);    glVertex3f(1.0f, 1.0f, 1.0f);
glNormal3f(-1.0f, 0.0f, 1.0f);   glVertex3f(-1.0f, 1.0f, 1.0f);
glNormal3f(-1.0f, 0.0f, 1.0f);  glVertex3f(-1.0f, -1.0f, 1.0f);
glNormal3f(1.0f, 0.0f, 1.0f);   glVertex3f(1.0f, -1.0f, 1.0f);
```

We also need to define the ambient , diffuse and specular colours of both the light and material of the cube. To do this we define some colours

```
GLfloat light_amb[] = { 0.2f,0.5f ,0.2f, 1.0f };
GLfloat light_diff[] = { 0.0f,1.0f ,0.0f, 1.0f };
GLfloat light_spec[] = { 1.0f,1.0f ,1.0f, 1.0f };
GLfloat refl_amb[] = { 0.0f,0.5f,0.0f,1.0f };
GLfloat refl_diff[] = { 0.0f,1.0f,0.0f,1.0f };
GLfloat refl_spec[] = { 1.0f,1.0f,1.0f,1.0f };

glLightfv(GL_LIGHT1, GL_AMBIENT, light_amb);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light_diff);
glLightfv(GL_LIGHT1, GL_SPECULAR, light_spec);
```

and then assign them to the Light and Material effects properties of the enabled light and material respectfully which we need to enable before setting these properties and defining the normal/vertices pairs. When we enable the light need to define the position the light will be emitting from. With all these defined OpenGL can now calculate the I (Intensity) in the Phong Illumination Model.

```
GLfloat light_position[] = { 0.0f,1.0f,1.5f,0.0f };
glLightfv(GL_LIGHT1, GL_POSITION, light_position);
```

To improve on the lighting I would fix the bug that cause some of the vertices to change colour incorrectly when you move the cube and camera

<u>Requirement 6 - Texture mapping</u>

Before we can map a texture to the cube we need to load a bmp image file into OpenGL, to do this we need to read the file and assign the RGB values of each pixel to an Array, which is then Binded to a texture using glGenTextures() and glBindTexture() functions. After the texture data has been bounded to a OpenGL texture we can change its Parameters using the glTexParameteri() functions,

where we can assign filtering, wrapping and mip-mapping to the texture. In my code we use the Linear filtering method, which calculates the colour of the texel ((U,V) coordinates) we look at the other surrounding texels and then mixes the colour depending on the distances to each center coordinate. By doing this we shouldn't get any hard edges around texture.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

We also need to set what happens to the texture after we use texel coordinates outside of 0 to 1. As each texture in OpenGL has coordinates between 0 and 1 for each axis of the texture. In my code we set this to GL_REPEAT so we get a continuous text on each face of the cube.

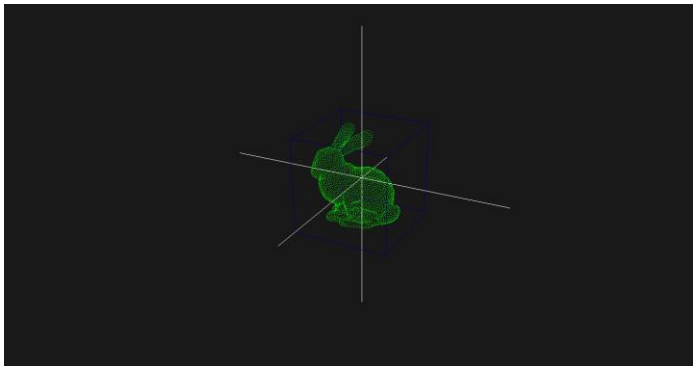
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

After we have assigned the setting of the cube it is now safe to map the texel coordinates to the vertices of the cube, this is done in a similar way to setting the normal's of the cube's vertices, but instead we use glTextCoord2f(U, V), before each vertex we want to assign them to .

```
glBegin(GL_QUADS);
glColor4f(1.0f,0.0f,0.0f,alpha);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,-1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,-1.0f, 1.0f);
```

Part 2

Requirement 7 - Load a mesh model



We loaded the mesh into OpenGL by storing all the vertices - lines in the mesh file in a vector of Eigen::Vector3d's, and the face Index- lines in another vector of Eigen::Vector3d's. After this to draw the mesh we use the GL_POINTS primitive and iterate over the vector of all vertices in the mesh.

```
glBegin(GL_POINTS);
glPointSize(1);
glColor4f(0.0f, 1.0f, 0.0f, alpha);
//Draws all the points in the mesh
for (unsigned int i = 0; i < vertices.size(); i++) {
        glVertex3f(vertices[i](0), vertices[i](1), vertices[i](2));
}
glEnd();
```

To scale and translate the mesh into the cube we need apply the following Transformation Matrix to all points in the vertices vector.

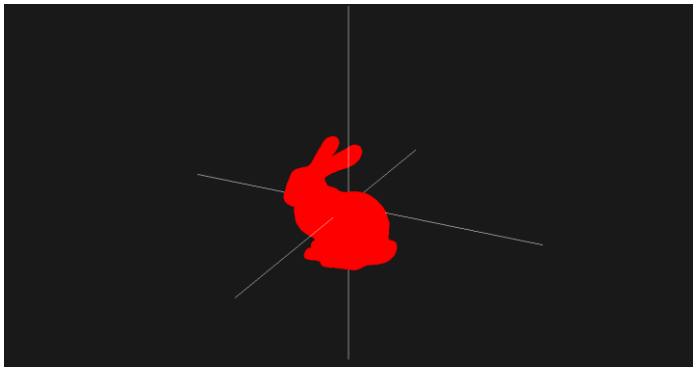
$$T = \begin{bmatrix} 0.45 & 0 & 0 & -0.3 \\ 0 & 0.45 & 0 & -0.2 \\ 0 & 0 & 0.45 & 0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It would be nicer if I centred the mesh onto the 0,0,0 coordinate and then scaled the mesh to fit exactly in the cube, as it would produce the nice results when we rotate the mesh around the axes for requirement 8.

These draw call also have to be defined in a different stack to the cube as we don't want to be rotating the mesh the same way we do the in part 1 of the coursework.

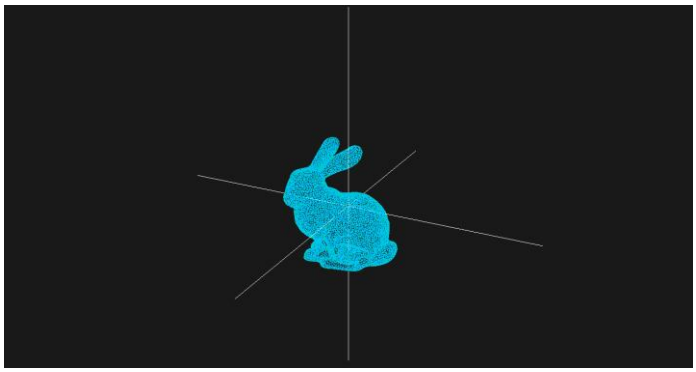
Requirement 8 - Rotate mesh, render mesh in different mode

To draw the mesh in different modes we use the GL_TRIANGLES and GL_LINES primitives, when we use these we need to connect the points defined in each face Index, so we don't draw arbitrary triangles that would not form the 3d representation of the mesh.



```
glBegin(GL_TRIANGLES);
glColor4f(1.0f, 0.0f, 0.0f, alpha);
for (unsigned int i = 0; i < faceIndices.size(); i++) {
        Eigen::Vector3d vertexIndex = faceIndices[i];
        glVertex3f(vertices[vertexIndex(0) - 1](0), vertices[vertexIndex(0) - 1](1),
vertices[vertexIndex(0) - 1](2));
        glVertex3f(vertices[vertexIndex(1) - 1](0), vertices[vertexIndex(1) - 1](1),
vertices[vertexIndex(1) - 1](2));
        glVertex3f(vertices[vertexIndex(2) - 1](0), vertices[vertexIndex(2) - 1](1),
vertices[vertexIndex(2) - 1](2));
}
glEnd();
```

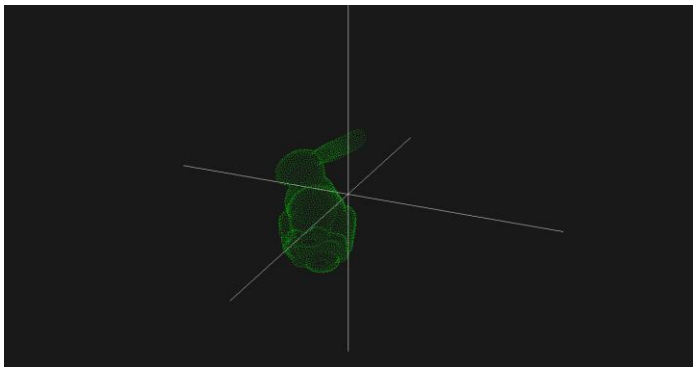
The method I use to draw the connecting lines in the mesh , could be greatly improved, as it currently has the possibility to draw the same line multiple time, this greatly impacts the performance when displaying the mesh like this .



To rotate the mesh we use Matrix transformations, we use the same one's defined for rotating the Camera, the only difference is that we can now use Eigen::Matrix to represent them instead of GLfloat arrays, this makes the code for each rotation a lot shorter and easier to follow.

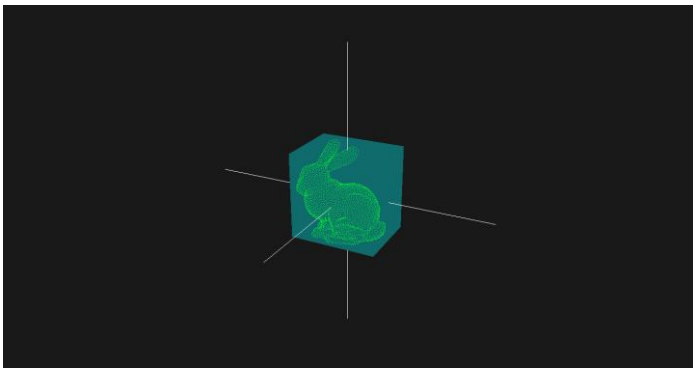
We need to do matrix transformation for each point as we need to know the exact points that are been drawn to the screen so we can do the latter Requirements of the coursework and glRotate doesn't let us do this

To Improve on the rotation of the mesh I would like to make it so that you can see the continuous updates to the mesh, not just the first and last positions from when you press and release the key.



Requirement 9 - Compute axis-aligned bounding box (abb) of the mesh

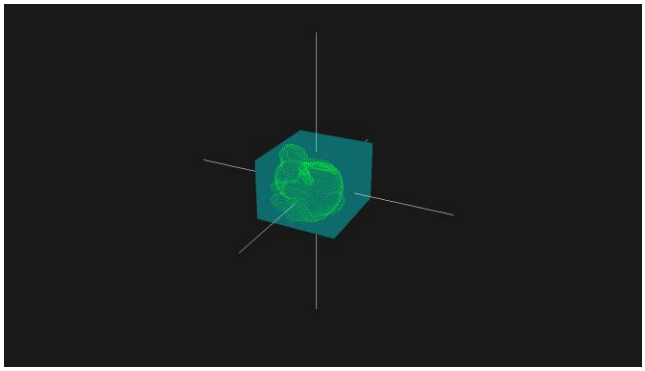
To calculate an Axis-aligned bounding box we need to calculate the minimum and maximum of the mesh in each axis, we need to recalculate this whenever we update the position or rotation of the mesh.



We then draw it as a rectangular box around the mesh, the same method as drawing a cube but we replace all the 1 with the max value in the axis and -1 with the min point in the axis.

```
abbMin = vertices[0];
abbMax = vertices[0];
for (unsigned int i = 0; i < vertices.size(); i++) {
        if (vertices[i](0) < abbMin(0))
        abbMin(0) = vertices[i](0); //Min X
        if (vertices[i](1) < abbMin(1))
        abbMin(1) = vertices[i](1);      //Min y
        if (vertices[i](2) < abbMin(2))
        abbMin(2) = vertices[i](2); //Min z
        if (vertices[i](0) > abbMax(0))
        abbMax(0) = vertices[i](0); //Max x
        if (vertices[i](1) > abbMax(1))
        abbMax(1) = vertices[i](1); //Max y
        if (vertices[i](2) > abbMax(2))
        abbMax(2) = vertices[i](2); //Max Z
}
```

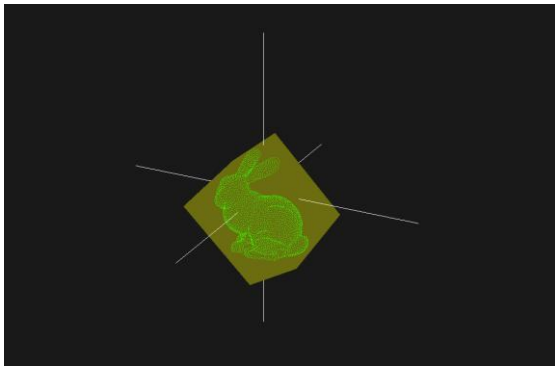
```
glBegin(GL_QUADS);
glColor4f(0.0f, 1.0f, 1.0f, alpha);
//Back
glVertex3f(abbMax(0), abbMin(1),abbMin(2));
glVertex3f(abbMax(0), abbMax(1),abbMin(2));
glVertex3f(abbMin(0), abbMax(1),abbMin(2));
glVertex3f(abbMin(0), abbMin(1),abbMin(2));
```



To draw the Abb as a transparent cube we need to define the colour of the rectangular box as a Color4f(), this is the same as Color3f but the last variable being the transparency of the colour with 0 being completely transparent and 1 being opaque. We also need to define all the colours draw in an order that draws the most opaque first and most transparent last.

Requirement 10 - Compute oriented bounding box (obb) of the mesh

To compute an oriented bounding box for the mesh we need to Principle Component Analysis.



The First step is the calculate the covariance of the points in the mesh, which can then be used to generate the Covariance Matrix.

$$\operatorname{cov}(X, Y) = \sum_{i=1}^{N} \frac{(x_i - \bar{x})(y_i - \bar{y})}{N}.$$

$$\text{CovMat} = \begin{bmatrix} \text{Cov}(X,X) & \text{Cov}(X,Y) & \text{Cov}(X,Z) \\ \text{Cov}(X,Y) & \text{Cov}(Y,Y) & \text{Cov}(Y,Z) \\ \text{Cov}(Z,X) & \text{Cov}(Z,Y) & \text{Cov}(Z,Z) \end{bmatrix}$$

From the covariance matrix we can calculate the Eigen Vector and Eigen Values of it by using eigenvalue decomposition. As CovMat is symmetric we can represent is as

CoveMat = $U\Lambda U^T$ where U is the orthonormal matrix of Eigenvectors of the covariance matrix. In my code we calculate the U^T by

```
Eigen::EigenSolver<Eigen::MatrixXd> covEigen(covarianceMatrix);
eigV = covEigen.eigenvectors().real();
```

We can use these Eigen Vector as a new reference frame for the mesh, To translate all the points to this new reference frame we do the following calculation.

$$p' = U^T(p-\mu)$$

where p' is the new point in the reference frame, p is the old point and μ is the mean of all the points in the mesh.

```
Eigen::Vector3d newPos = vertices[i] - meanVector;
Eigen::Vector3d newVert = V * newPos;
```

We can now calculate the min and max points in the new reference frame by the same method as the abb. The final step is to translate these min/max points back into the global reference frame, we can do this by the equation.

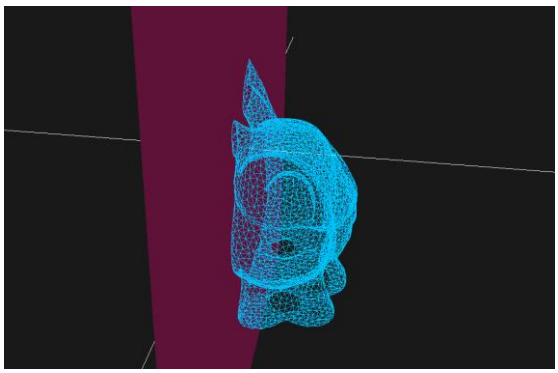
$$p = (U^T)^{-1} p' + \mu$$

```
for (unsigned int i = 0; i < 8; i++) {
        obbVertices[i] = (V.inverse() * obbVertices[i]) + meanVector;
}
```

We can now use these points to draw the desired obb. With extra time I would try to improve the efficiency of calculating this bounding box as I currently use lots of variables that could be reduced to make the program more memory efficient .

Requirement 11 - Cut-away view of the mesh

To calculate a cut away view of the mesh we need to consider the 4 following conditions:

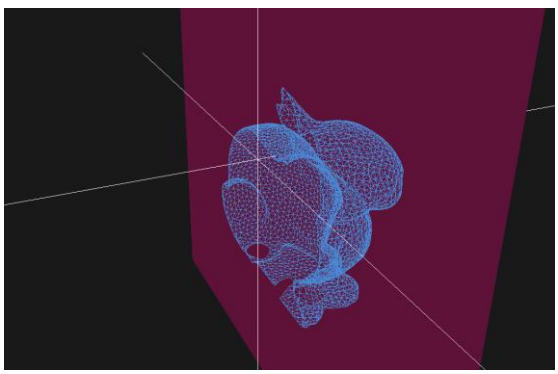


1. **All points in triangle are in front of plane**

This happens when the Z coordinate of the vertices is greater than or equal to the Z coordinate of the Intersecting plane, when this occurs we can draw all the points

2. **All points in triangle are behind the plane**

This happens when the Z coordinate of the vertices is less than to the Z coordinate of the Intersecting plane. When this occurs draw none of the points



3. Two points is in front of plane

Draw the two points in front of the plane and then, we need to use Linear Interpolation between the 2 points in front and the 1 point behind the plane to calculate the positions of the intersecting points on the plane. With these points we can then draw line between them forming and Quad or 2 triangles

```
//Linear Interpolation
double x = v1(0) + ((v1(0) - v2(0)) * ((planeZ - v1(2)) / (v1(2) - v2(2))));
double y = v1(1) + ((v1(1) - v2(1)) * ((planeZ - v1(2)) / (v1(2) - v2(2))));
```

4. One point in front of the plane

Draw the single point that is in front of the plane, Then by Linear interpolation calculate the 2 intersecting points on the plane. We can then draw a new triangle with these 3 points.

It seem that this method of clipping the triangles on the plane can leaves some of them unclipped, so we don't get a totally flat surface on the intersecting plane, but this could be caused by the incorrect ordering of drawing the new points.

Keyboard Controls & Other Info

Render Modes

Cube	Mesh	Bounding Box
v -vertices	m-faces	Z -abb
L -edges	n-vertices	x-obb
f -faces	b-lines	c- none
p -lighting	j-disable	
k- disable		

Rotating, camera controls & other key bindings

w,s - object around X a,d - object around q,e -object around Z

Shift + button -Camera rotate , Mouse 1/2 - zoom in/out , left arrow -pan left , right arrow -pan right

r -reset cube and camera

i - toggle intersecting plane ,Up arrow - move plane along Z axis, Down arrow -move plane back along Z axis

Note - If you want to rotate the cube without having to press the button continuously comment the RotateMesh() functions on lines 1045,1049,1053,1057,1061,1065.