This project is broken down into two parts. In Part one, we will demonstrate different types of projections used in computer graphics and how to program them using C++ and OpenGL. Part two will go through different types of illusions and how to make them using Blender.

The code for the project can be found here:
https://github.com/lukeleontowich/Math3200Projections
The demo video can be found here:
https://youtu.be/1FFek7C9fhU

Section 1: Projections

The projections that will be discussed in this project will be orthogonal, oblique and perspective projections. But before we get into the projections lets go through some concepts of computer graphic. In computer graphics all points must be projected onto a 2D window. The x axis is horizontal, the y axis is the vertical and the z axis is the depth. The matrices that are discussed are all 4x4 matrices and all points are in $\Re^4$. The points are (x, y, z, w) where (x, y, z) is the point in 3D space and w is homogeneous coordinate.

There are three types of matrices that are needed to take a point in a local geometric shape to be translated to the screen. These matrices are the model, view and projection matrix. The picture can give a brief knowledge of the purpose of each matrix. The model matrix essentially takes care of translations, scalings, rotations of a particular object. The view matrix is the camera, it takes care of what we want to look at. Lastly, is the matrix that we are concerned about for this project, that is the projection matrix. This matrix takes care of how points are projected to the screen.

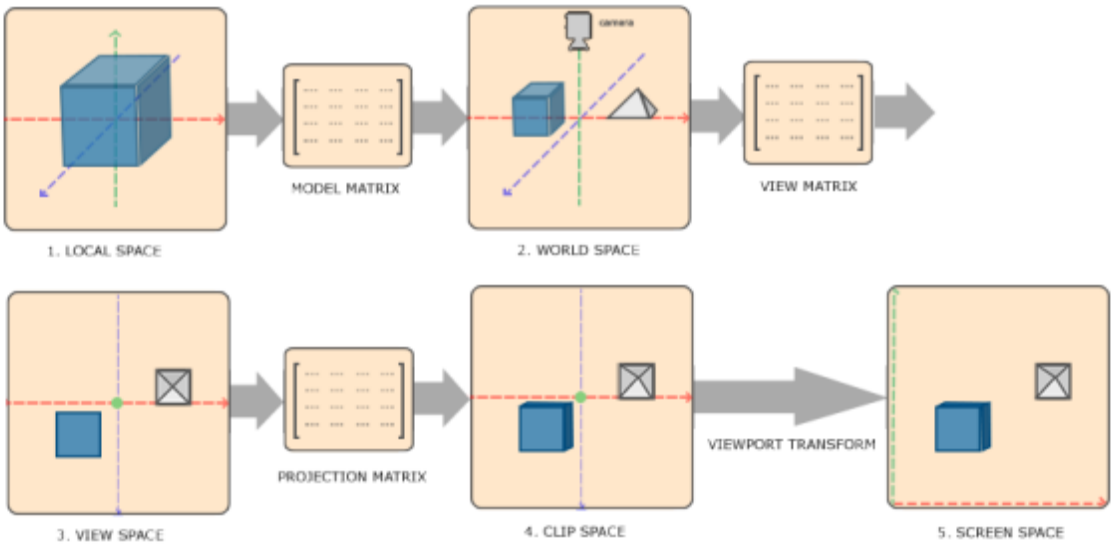

Figure 1.1: Projection Matrix [1]

Parallel Projections

Parallel projections take a point (x, y, z) in 3D space and map it to the computer screen in a parallel line. There are two types of parallel projections that we will go through, Orthogonal and Oblique projections.

Orthogonal Projection

Orthogonal projections take a point (x, y, z) and map to the screen such that the line from (x, y, z) to the screen is orthogonal to the screen. So say the screen is at (x, y, 0) then an orthogonal projection as a function would be f : M (x, y, z, 1) $\rightarrow (\alpha,\ \beta,\ 0,\ 1)$, where x = α, y = β and z = 0. And M is the orthogonal projection matrix.
So a matrix for this particular orthogonal projection would be:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Since:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 0 \\ 1 \end{pmatrix}$$

As you can see our orthogonal projection matrix isn't necessarily an "orthogonal matrix" as we would expect in mathematics. Since $MM^T \neq I$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Now we need to generalize the orthogonal projection. Essentially what we need to do is define our 3D space as a rectangular box and map that into a generalized cube, which is called the canonical view volume. The rectangular box will be defined as a box by the dimensions left, right, top, bottom, near and far or l, r, t, b, -n, -f for simplicity. Note that near and far are negated since the depth of the screen goes in the -z direction. Then the 8 vertices of the rectangular box will be (l, b, -n), (r, b, -n), (r, t, -n), (l, t, -n), (l, b, -f), (r, b, -f), (r, t, -f), (l, t, -f). The canonical view volume is a cube with corresponding vertices: (-1, -1, 1), (1, -1, 1), (1, 1, 1), (-1, 1, 1), (-1, -1, -1), (1, -1, -1), (1, 1, -1), (-1, 1, -1).

Now we need to set up our function. What we want to do is take the centre of the rectangular box and translate it to the centre of the canonical view. Then we are going to scale so that the sides have a length 2.

So let's set up the translation first. The centre of the rectangular box is {(r+l / 2), (t+b / 2), -(f+n) / 2, 1} and that needs to be translated to (0, 0, 0, 1). So then we can see that we need to subtract the centre by the vector {-(l+r) / 2, -(t+b) / 2, -(n+f) / 2, 1}. This can be written as a translation matrix:

$$\begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & \frac{-(b+t)}{2} \\ 0 & 0 & 1 & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Then check to see if this works by using our centre of the rectangular box:

$$\begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & \frac{-(b+t)}{2} \\ 0 & 0 & 1 & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{l+r}{2} \\ \frac{b+t}{2} \\ -\frac{f+n}{2} \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Now we need the scaling matrix. So let's show this by example. Say we have the vertex (r, t, -f) on our rectangular box. We need this point to map to (1, 1, -1) in our canonical view. So first we apply the transformation:

$$\begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & \frac{-(b+t)}{2} \\ 0 & 0 & 1 & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r \\ t \\ -f \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{r-l}{2} \\ \frac{t-b}{2} \\ \frac{-f+n}{2} \\ 1 \end{pmatrix}$$

Now we need to scale the x coordinate by 2 / (r - l), the y coordinate by 2 / (t - b) and the z coordinate by 2 / (f - n). This can be done with the matrix:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

So then putting our two matrices together gives:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & \frac{-(b+t)}{2} \\ 0 & 0 & 1 & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

So this is our orthographic projection matrix.

In the OpenGL program, the code for the Orthogonal projection is:

```
glm::mat4 OrthogonalProjection::createProjection() {
    //  initialize matrix to identity matrix
    glm::mat4 m(1.0f);

    // explanation of matrix in report
    m[0][0] = 2 / (right - left);
    m[1][1] = 2 / (top - bottom);
    m[2][2] = 2 / (far - near);

    m[3][0] = -(right + left) / (right - left);
    m[3][1] = -(top + bottom) / (top - bottom);
    m[3][2] = (far + near) / (far - near);

    return m;
}
```

Oblique Projection

An oblique projection is a parallel projection but the line from a point in 3D space to the screen is at an angle to the screen. Using Geogebra I will attempt to show what I mean by oblique projection:

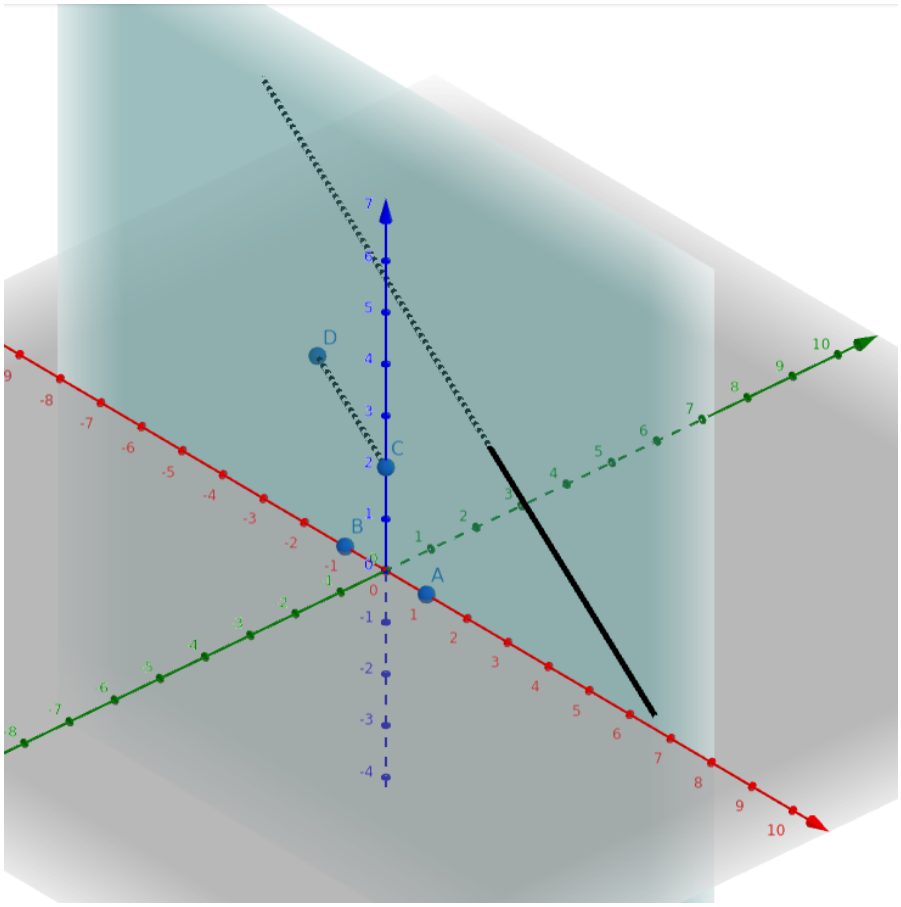

Figure 1.2: Oblique Projection [2]

The blue plane is the screen, and DC is a segment from 3D space to the screen. The other line is parallel to DC. In an oblique projection every point in 3D space is mapped in a line parallel to DC.

From a top-down view we have:

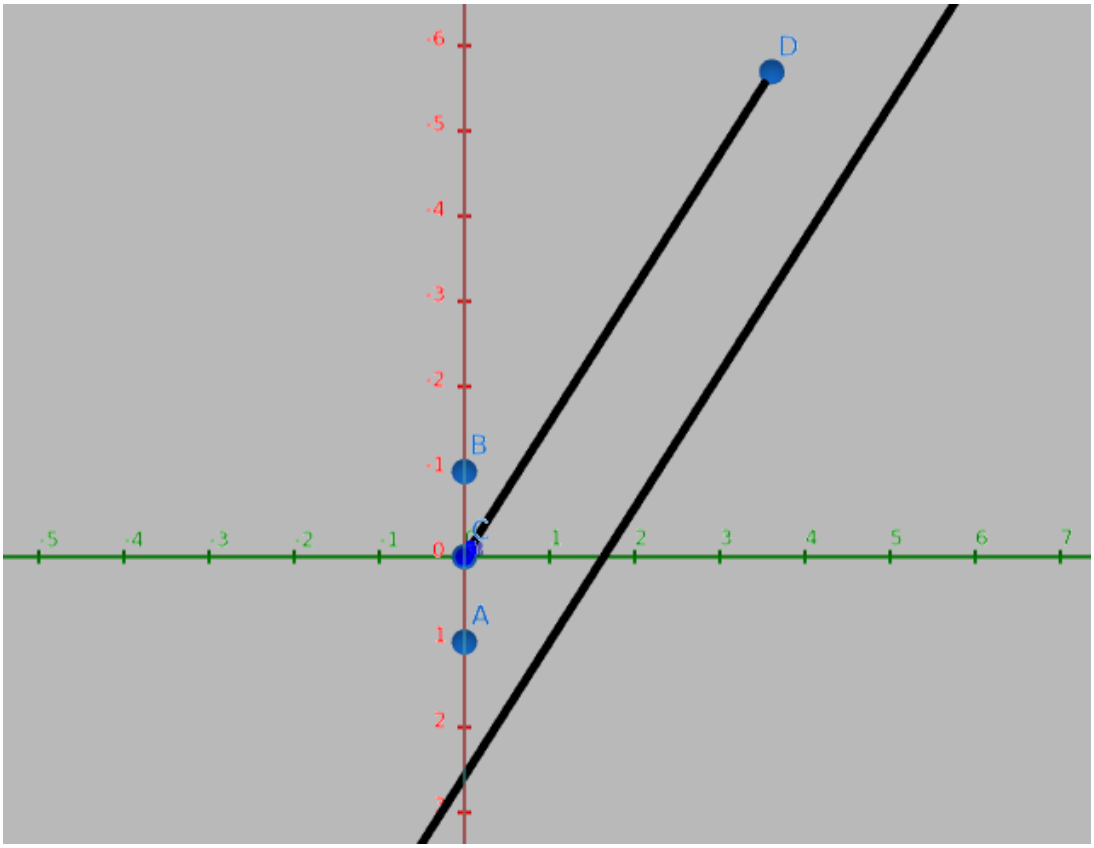

Figure 1.3: Top-Down Oblique Projection [2]

So the red line is where the blue screen was (z-axis), and the green line is x-axis. We can see that there is an angle θbetween the line DC and the z-axis. So to solve from x_s (x on screen) we have D(x, z) then $tan\theta = z / (x_s - x) \Rightarrow x_s - x = z / tan\theta \Rightarrow x_s = z\,cot\theta + x$

Looking from the side we get:

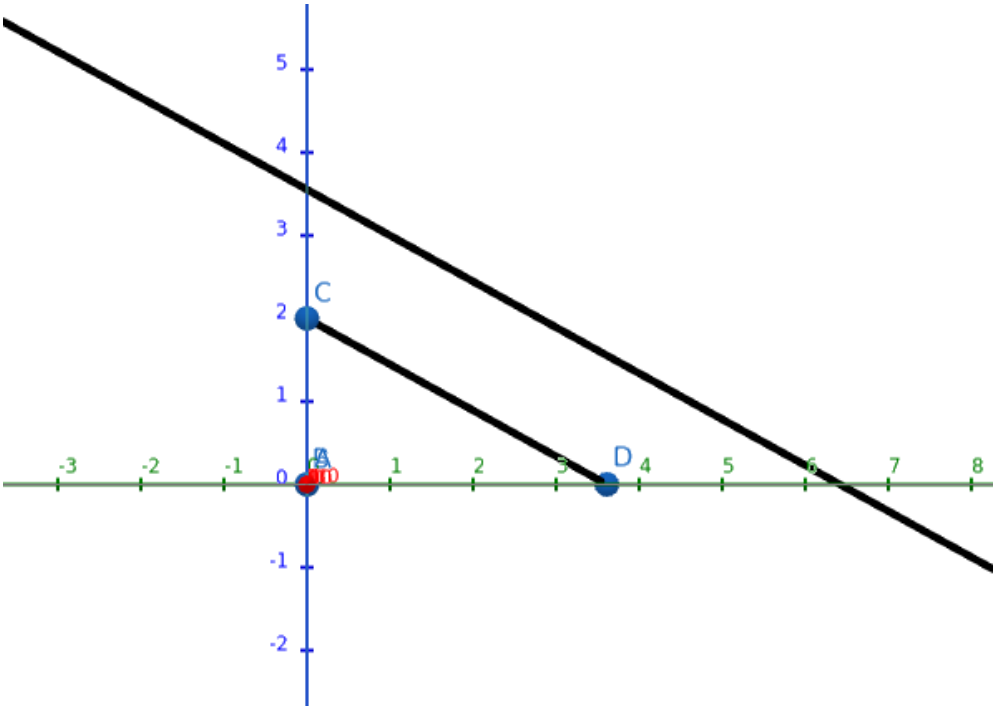

Figure 1.4: Side View Oblique Projection [2]

Where the blue line is the z-axis and the green line is the y-axis. We see again that there is angle ϕbetween DC and the z-axis. So like as above we can solve for $y_s = zcot\phi + y$.

Now to make the oblique matrix. Say the screen is at Z = 0 then we need a matrix M such that $M(x, y, z, 1) = (\alpha, \beta, \gamma, 1)$ where $\alpha = zcot\theta + x$, $\beta = ycot\phi + y$, $\gamma = 0$.
The matrix to do this is:

$$\begin{pmatrix} 1 & 0 & \cot(\theta) & 0 \\ 0 & 1 & \cot(\varphi) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + z\cot(\theta) \\ y + z\cot(\varphi) \\ 0 \\ 1 \end{pmatrix}$$

However we need to rewrite this matrix M as a product of two matrices and call it A.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Then

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & \cot(\theta) & 0 \\ 0 & 1 & \cot(\varphi) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + z\cot(\theta) \\ y + z\cot(\varphi) \\ 0 \\ 1 \end{pmatrix}$$

We needed that step because we still need the translation and scaling matrix that we constructed in the orthographic projection:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & \frac{-(b+t)}{2} \\ 0 & 0 & 1 & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

But we need to sandwich that between A and M to get:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & \frac{-(b+t)}{2} \\ 0 & 0 & 1 & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & \cot(\theta) & 0 \\ 0 & 1 & \cot(\varphi) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Which gives the matrix:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & \frac{2\cot(\theta)}{r-l} & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & \frac{2\cot(\varphi)}{t-b} & -\frac{t+b}{t-b} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

But we need to change our left, right, bottom and top a bit. We need to apply the matrix

$$\begin{pmatrix} 1 & 0 & \cot(\theta) & 0 \\ 0 & 1 & \cot(\varphi) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

To the 8 vertices of our rectangular box to get the new left, right, bottom and top. For example for the vertex (left, bottom, -near, 1) we get the following:

$$\begin{pmatrix} 1 & 0 & \cot(\theta) & 0 \\ 0 & 1 & \cot(\varphi) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} l \\ b \\ -n \\ 1 \end{pmatrix} = \begin{pmatrix} -n\cot(\theta) + l \\ b - n\cot(\varphi) \\ 0 \\ 1 \end{pmatrix}$$

So then we would get:

$left_{new} = left_{old} - near * cot\theta$

$$right_{new} = right_{old} - near * cot\theta$$
$$bottom_{new} = bottom_{old} - near * cot\phi$$
$$top_{new} = top_{old} - near * cot\phi$$

So for the code, we initialize the parameters in our ObliqueProjections constructor:

```
ObliqueProjection::ObliqueProjection(const float& left_old, const float& right_old,
                                     const float& bottom_old, const float& top_old,
                                     const float& near_old, const float& far_old,
                                     const float& atheta, const float& aphi) {
    //  No calculations need for angles, near and far. Just copy the values coming in.
    theta = atheta;
    phi = aphi;

    near = near_old;
    far = far_old;

    //  left, right, bottom, top are calculated based on the report
    //  Note that cmath doesn't have cot so using property cot(x) = 1 / tan(x)
    left = left_old - near * (1.0f / tan(glm::radians(theta)));
    right = right_old - near * (1.0f / tan(glm::radians(theta)));

    bottom = bottom_old - near * (1.0f / tan(glm::radians(phi)));
    top = top_old - near * (1.0f / tan(glm::radians(phi)));
}
```

Then the code for the Oblique matrix

$$\begin{pmatrix} \frac{2}{r-l} & 0 & \frac{2\cot(\theta)}{r-l} & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & \frac{2\cot(\varphi)}{t-b} & -\frac{t+b}{t-b} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Is:

```
glm::mat4 ObliqueProjection::createProjection() {
    //  initialize matrix to identity matrix
    glm::mat4 m(1.0f);

    //  See report for matrix details
    m[0][0] = 2.0f / (right - left);
    m[2][0] = 2.0f / (tan(glm::radians(theta)) * (right - left));
    m[3][0] = -(right + left) / (right - left);
    m[1][1] = 2.0f / (top - bottom);
    m[2][1] = 2.0f / (tan(glm::radians(phi)) * (top - bottom));
    m[3][1] = -(top +bottom) / (top - bottom);
    m[2][2] = 0.0f;

    return m;
}
```

Perspective Projections

Perspective projections are real life. What I mean by that, is that we view the world as a perspective projection. This is obviously important to computer graphics, since we work with perspective projections we are trying to model the world as we see it.

To set up the projection matrix, we need to again map our 3D world into a generalized cube, but this to we a truncated pyramid as our original 3D space, instead of the rectangular box that we used for parallel projections. Perhaps it's easier to visualize as a picture:

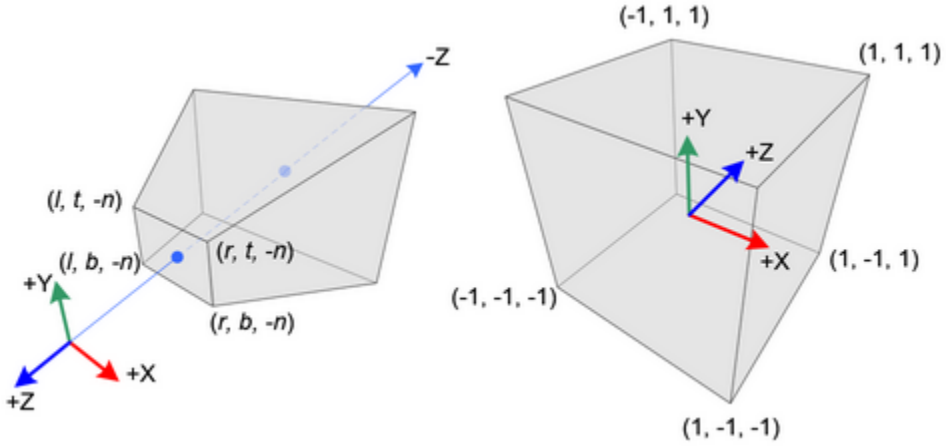

Figure 1.5: Perspective Projections [3]

So first we need to see how a point in the 3D point maps to the screen given by (l, t, -n), (r, t -n), (l, b, -n), (r, b, -n).

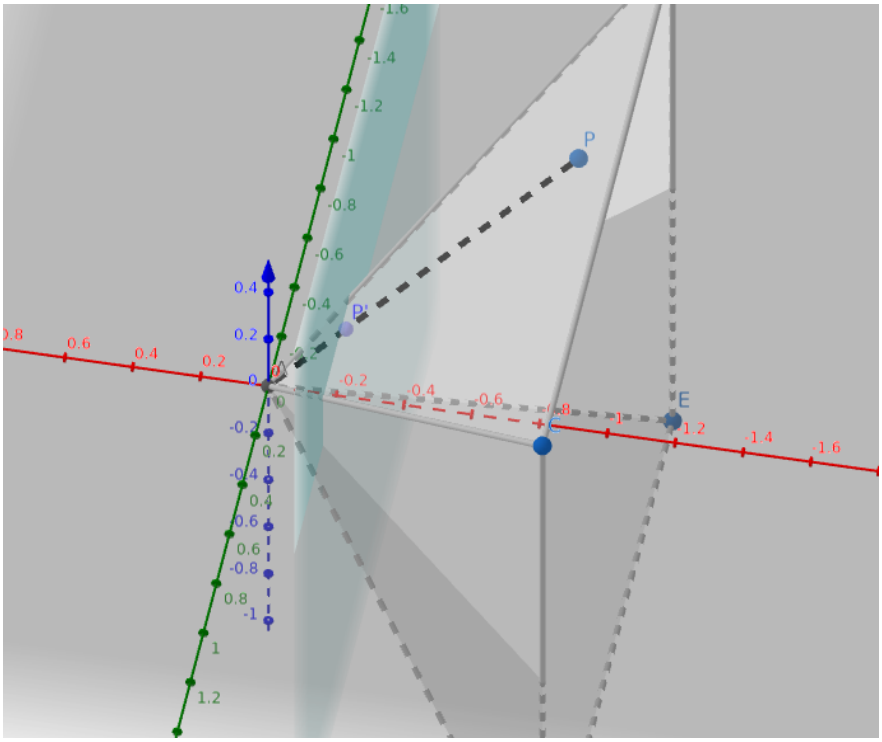

Figure 1.6: Perspective [4]

The grey pyramid is our 3D space and the blue plane is the screen. As you can see, there is a line segment from P to O. The intersection of PO and the blue plane is P', the point mapped to the screen.
So then the top down view gives:

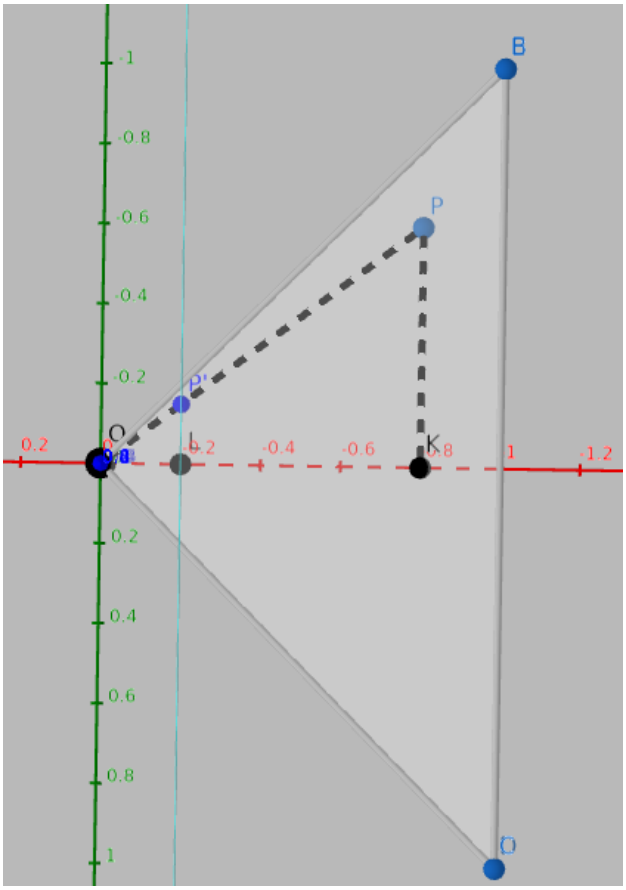

Figure 1.7: Top-Down Perspective [4]

P is some point (x, y, z) but right now since we projecting from the top (xz projection) we have P (x, z). We know that OL is $|z_s|$ (z screen) and OK is |z| and PK is |x|. So by Thales theorem we have: $OK/\,OL \;=\; PK/\,P'L \;\Rightarrow |z| \,/\, |z_s| \;=\; |x| \,/\, |x_s| \;\Rightarrow |x_s| \;=\; |z_s| \,/\, (|z||x|)$

But the plane is at -near (-n) so then we have:

$x_s = -\,nx \,/\, z$

The yz projection will give similar results:

$y_s = -\,ny \,/\, z$

So now we need to map the coordinates in the pyramid to the canonical view, like in the parallel projections. So from figure 5 we have (l, t, -n) →(-1, 1, -1), (r, t, -n) →(1, 1, -1), (l, b, -n) →(-1, -1, -1), (r, b, -n) →(1, -1, -1)

So for x [l, r] →[-1,1] which is $x_s \rightarrow x_c$ where x_c is the canonical view x. So like in chapter 5.6 (Linear fractional functions) we project a line onto a perpendicular line. Where $x_s \; and \; x_c$ are the axis'.

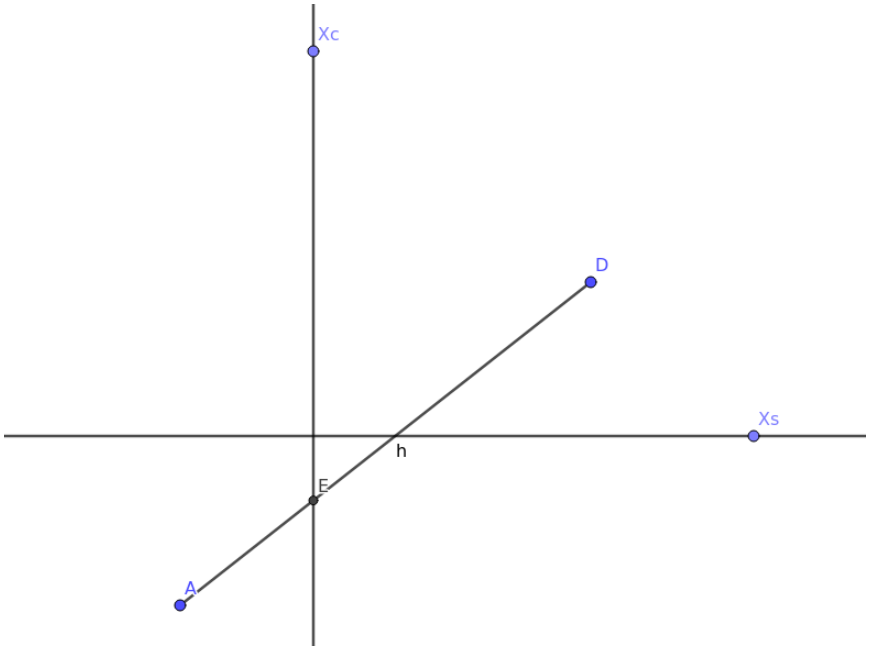

Figure 1.8: Line Projected on Perpendicular Line [5]

So A is at (l, -1) and D is at (r, 1). So the equation of the line would be:

$x_c = ((1 - (-1)) / (r - l)) * x_s + k$

Then use the point (r, -1) to solve for k

$-1 = (2 / (r - l)) * r + k$

$k = (2r / (r - l)) - 1$

$k = (2r - (r - l) / (r - l)$

$k = (r + l) / (r - l)$

So the equation is:

$x_c = (2 / (r - l)) * x_s + (r + l) / (r - l)$

$x_c = (2x_s + r + l) / (r - l)$

The math will be very similar for y_c so the equation is:

$y_c = (2y_s + r + l) / (r - l)$

Now back to x_c. Sub in the $x_s = -n / (zx)$.

Then

$x_c = (2(-nx / z) + r + l)) / (r - l)$

$x_c = (-2nx + rz + lz)) / z(r - l)$

$x_c = (-2nx) / (z(r - l)) + (z(r + l)) / (z(r - l))$

$x_c = (- 1 / z) * \{(2n / (r - l)) x + ((r+l) / (r - l)) z\}$

y_c will give a similar equation but with t and b (top and bottom) instead . Let's write both these equations using symbolab so they look cleaner.

$$x_C = \left(\frac{2n}{r-l}x + \frac{(r+l)}{(r-l)}z\right)\left(-\frac{1}{z}\right) \quad \text{and} \quad y_C = \left(\frac{2n}{t-b}y + \frac{(t+b)}{(t-b)}z\right)\left(-\frac{1}{z}\right)$$

We need the homogeneous coordinate w_s to be -z in order to get rid of the (- 1 / z) in order to that we need a matrix like:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -z \end{pmatrix}$$

Then this matrix will give us what we calculated above

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{2nx+z(r+l)}{r-l} \\ \frac{2ny+z(t+b)}{t-b} \\ 0 \\ -z \end{pmatrix}$$

Now we need to find z_c which will be z / w which is z_s / w_s well we know $w_s = -z$. To figure out the rest of it we need to add two unknown variables j and k to the matrix.

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & j & k \\ 0 & 0 & -1 & 0 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{2nx+z(r+l)}{r-l} \\ \frac{2ny+z(t+b)}{t-b} \\ jz+k \\ -z \end{pmatrix}$$

So $z_s = jz + k$ then $z_c = (jz + k) / -z$

So now we use the fact that we can map $(z, z_c) \rightarrow (-n, -1)$ and $(z, z_c) \rightarrow (-f, 1)$

So then we get two equations:

1) $-1 = (-jn + k) / -n \Rightarrow n = jn - k$
2) $1 = (-jf + k) / f \Rightarrow f = -jf + k$

Let's solve this by setting up a system of equations and then let symbolab do all the work for us!

Reduce matrix to reduced row echelon form $\begin{pmatrix} 1 & \cdots & b \\ 0 & \ddots & \vdots \\ 0 & 0 & 1 \end{pmatrix}$: $\begin{pmatrix} 1 & 0 & \frac{f+n}{-f+n} \\ 0 & 1 & \frac{2nf}{-f+n} \end{pmatrix}$

So then j = (f + n) / (n - f)
And k = (2nf) / (n - f)

So our matrix is:

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{(r+l)}{(r-l)} & 0 \\ 0 & \frac{2n}{t-b} & \frac{(t+b)}{(t-b)} & 0 \\ 0 & 0 & \frac{(f+n)}{(n-f)} & \frac{2nf}{(n-f)} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Then the code for the matrix is:

```
glm::mat4 PerspectiveProjection::createProjection() {
    //  Initialize matrix to identity matrix
    glm::mat4 m(1.0f);

    m[0][0] = (2.0f * near) / (right - left);
    m[2][0] = (right + left) / (right - left);
    m[1][1] = (2.0f * near) / (top - bottom);
    m[2][1] = (top + bottom) / (top -bottom);
    m[2][2] = (far + near) / (near - far);
    m[3][2] = (2.0f * far * near) / (near - far);
    m[2][3] = -1.0f;

    return m;
}
```

Section 2: Optical Illusions in Blender

Optical Illusions

To demonstrate some of the unique properties of projective geometry, we decided to research a field of geometry which is very common to many people: perception geometry or “optical illusions”. We recently had the opportunity to visit the Museum of Illusions in Toronto, Ontario, where we discovered the effects of perspective in geometry and how important distance, size, and angles are to how we perceive objects.

Figure 2.1: Ceiling Walking Illusion
The furniture in this room was nailed to the ceiling and the walls were designed upside-down, so it looks like you are walking on the ceiling. [6]

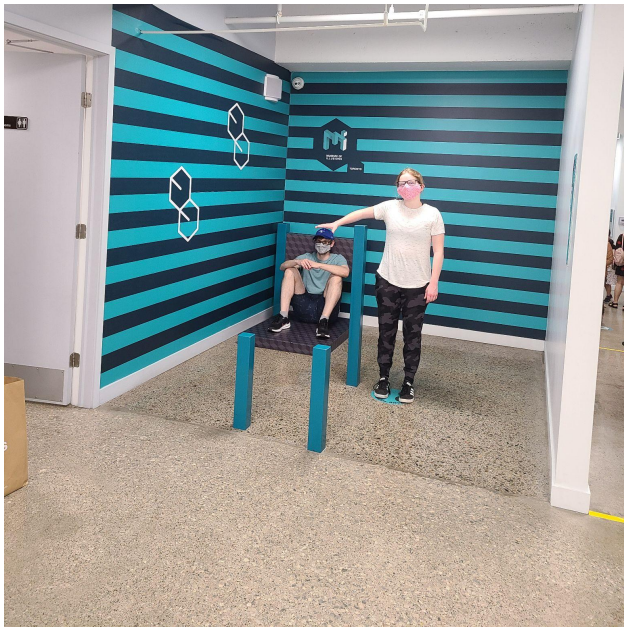

Figure 2.2: Tiny Person Illusion
The “chair” in this image is not actually a chair. If you look closely, you’ll notice that the two font pegs are ahead of the seat of the chair and the seat of the chair is on the ground. Taking this picture at a certain angle gives the perception that the person in the chair is smaller than the person standing closer to the camera. [7]

The illusions at the museum are examples of projective geometry. The objects in the illusions are 3D when seen in person, but taking a photo maps the 3D space to a 2D plane, and lining up the perspective of the camera just right can make some interesting pictures. In the next few sections, we will research some common optical illusions that we implemented in Blender.

Penrose Triangle

This optical illusion was named after Lionel Penrose, a psychiatrist, and his son, Roger Penrose, an award-winning mathematician.

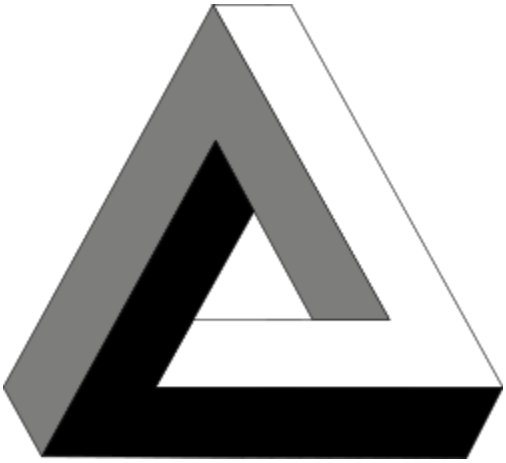

Figure 2.3: The Penrose Triangle [8]

The triangle looks impossible because the faces of the object do not connect in a way that the human eye perceives to be correct. However, the object in 3D space isn't a triangle at all, it's three rectangles connected at right angles to each other. This looks like a triangle when we project the image from a certain perspective. We implemented this in Blender to illustrate how this can be done.

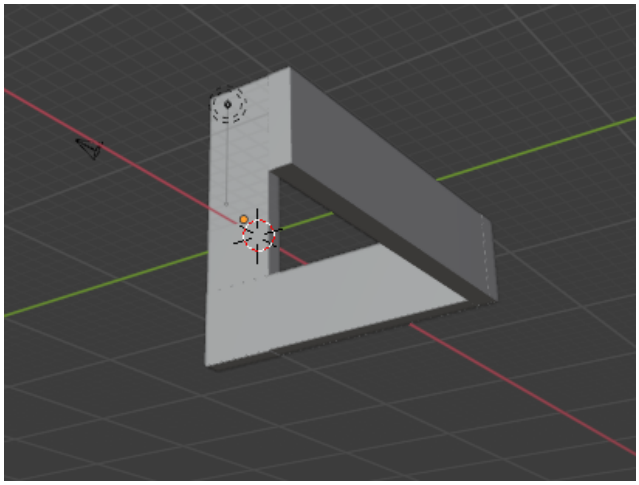

Figure 2.4: Blender Penrose Triangle
When we take a picture from a certain angle, the object looks like a triangle. [9]

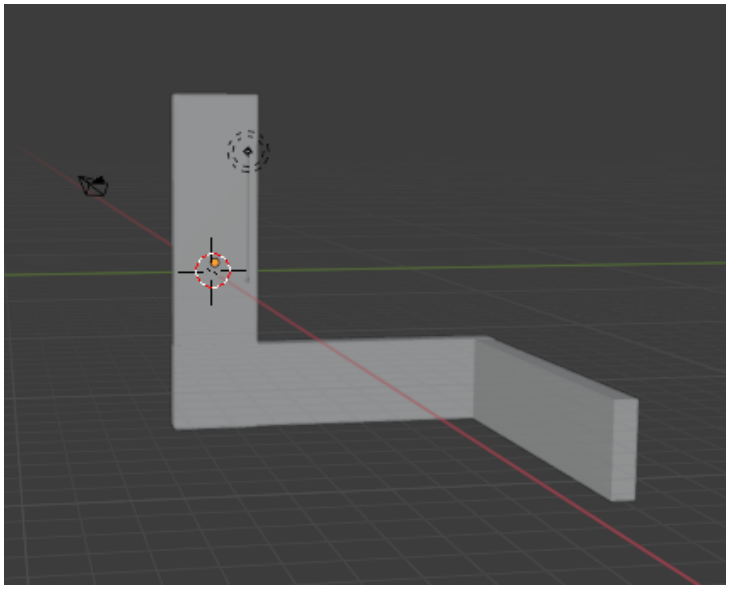

Figure 2.5: Penrose Triangle Exposed When we rotate the shape, we see that it isn't a triangle at all, in fact, two of the rectangles aren't even connected. [9]

There are many variations of the Penrose Triangle, including the Penrose square, pentagon, and other shapes. Perhaps one of the most famous variations is the Penrose Stairs illusion, which was popularized by Maurits Cornelis Escher in his painting *Ascending and Descending*, where people can be seen walking up and down stairs infinitely and never getting any higher or lower.

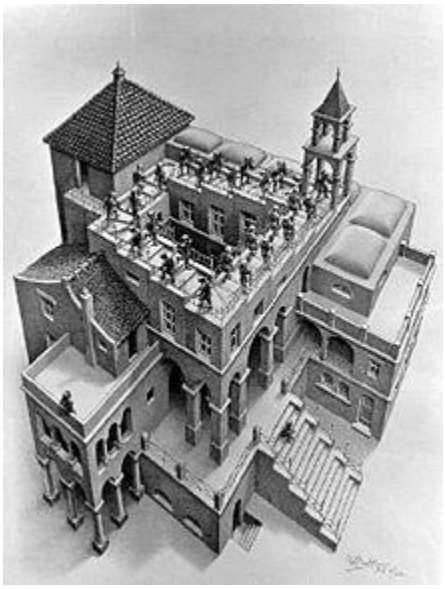

Figure 2.6: Ascending and Descending [10]

The Impossible Four Bar

The Impossible Four Bar may not be as popularized as the Penrose Triangle, but the principles are similar. The illusion here also has to do with the fact that the sides of the object do not connect the way we expect them to, giving the illusion that the shape is twisting when in reality, it isn't even completely connected.

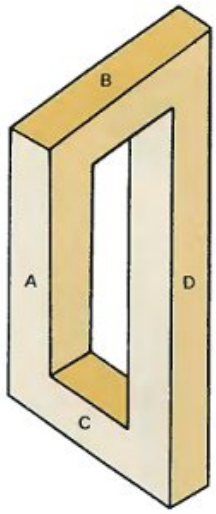

Figure 2.7: The Impossible Four Bar [11]

Making this shape in Blender was much like making the Penrose Triangle, except this time there were four rectangles to connect. The difference here was that side A (Figure 2.7) had to be shorter than side D and side D had to be a bit wider than side A.

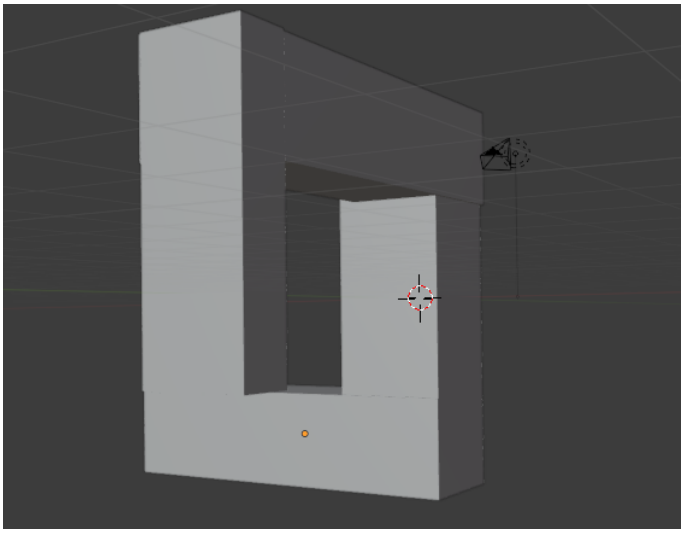

Figure 2.8: Blender Impossible Four Bar Notice that the rectangle on the left is much shorter than the one on the right, but from the right perspective, they look like they are the same length. [12]

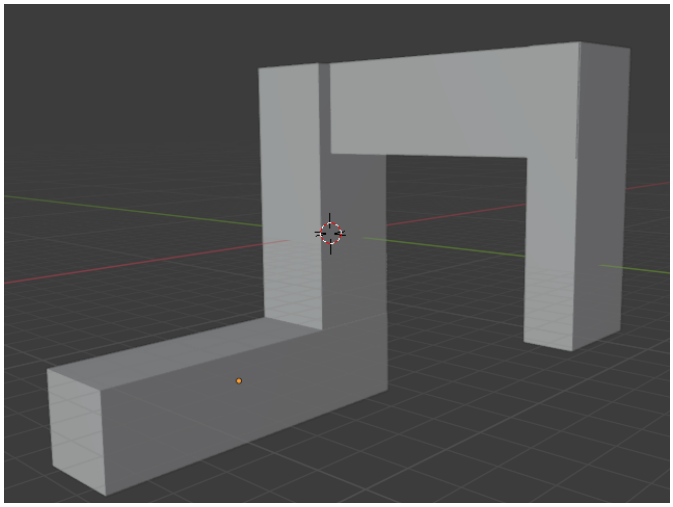

Figure 2.9: The Impossible Four Bar Exposed
Upon rotating the object, it is much easier to see that the rectangles are different sizes. [12]

Ambiguous Cylinder Illusion

The Ambiguous Cylinder Illusion was created by Kokichi Sugihara, a mathematics professor at Meiji University in Japan [13]. This illusion involves an object that looks like a cube from one perspective and a cylinder from another.

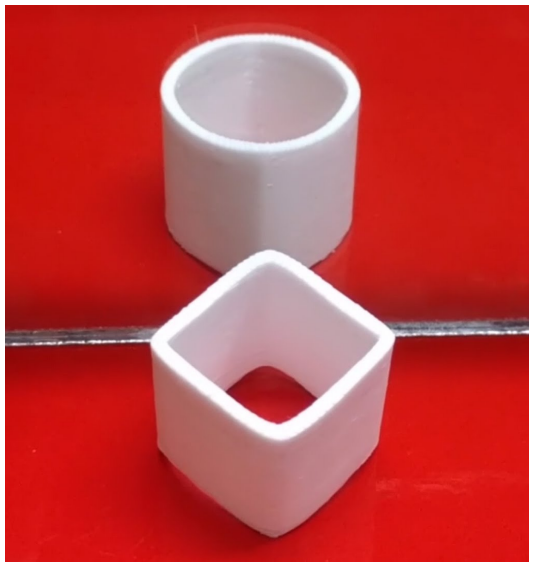

Figure 2.10: The Ambiguous Cylinder
In a mirror, the object looks like a different shape. [13]

This illusion can be made by making two quarters of the top of the shape rounded up and two quarters rounded down. The cut-out in Figure 2.11 below shows how this can be done.

Figure 2.11: Ambiguous Cylinder Cut-Out
Cutting out this shape and folding it into a cylinder makes an ambiguous cylinder. [14]

This shape involved a lot more steps to make in Blender, including using Boolean modifiers to extend the shape up or down and using the draw and smooth functions to fix up some of the rigid surfaces.

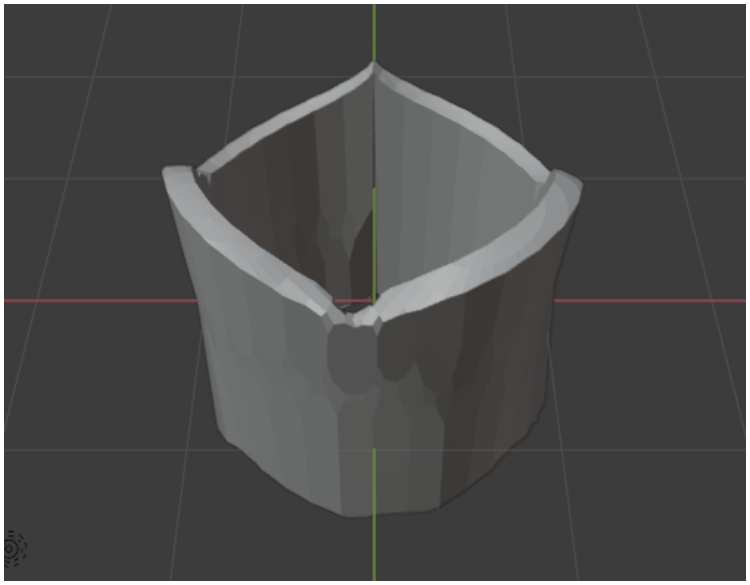

Figure 2.12: Cube
From one perspective, the ambiguous cylinder looks like a cube… [15]

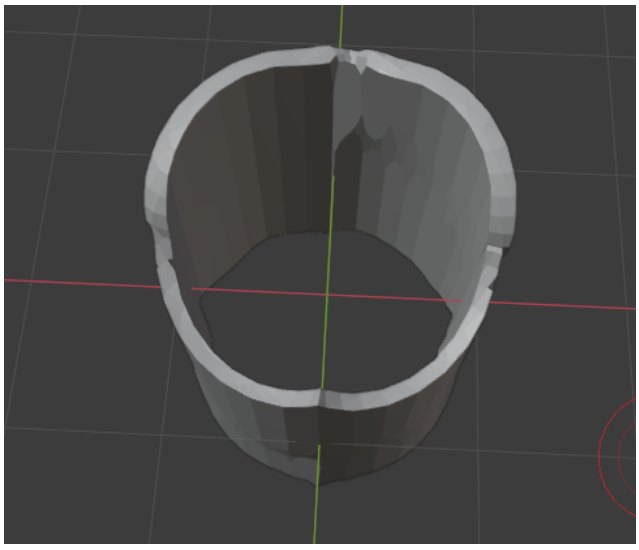

Figure 2.13: Cylinder
…from another perspective, it looks like a cylinder. [15]

References

[1] J. de Vries, *Learn OpenGL - Graphics Programming.* Kendall & Welling, 2020. Accessed on Dec. 7, 2021. [Online]. Available: https://learnopengl.com/book/book_pdf.pdf

[2] L. Leontowich, *Oblique Projection.* Accessed on Dec.7, 2021. [Online]. Available: https://www.geogebra.org/m/n5mj5eua

[3] S.H. Ann, *OpenGL Projection Matrix.* Accessed on: Dec. 7, 2021. [Online]. Available: https://www.songho.ca/opengl/gl_projectionmatrix.html#perspective

[4] L. Leontowich, *Math3200_perspective.* Accessed on Dec.7, 2021. [Online]. Available: https://www.geogebra.org/m/rrbhebvu

[5] L. Leontowich, *math3200_figure8.* Accessed on Dec.7, 2021. [Online]. Available: https://www.geogebra.org/classic/gz8wkbym

[6] B. McDonnell, *Ceiling Walking Illusion.* Sept. 5, 2021, Museum of Illusions: Toronto, Ontario.

[7] B.McDonnell, *Tiny Person Illusion.* Sept. 5, 2021, Museum of Illusions: Toronto, Ontario.

[8] *Penrose Triangle.* Wikipedia. Accessed on Dec.7, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Penrose_triangle

[9] J. McDonnell, *PenroseTriangle.* Blender. Dec. 7, 2021.

[10] M.C. Escher, *Ascending and Descending.* Wikipedia. Accessed on Dec. 7, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Penrose_stairs

[11] V. Alexeev, *The Impossible Four Bar.* Impossible World. Accessed on Dec. 7, 2021. [Online]. Available: https://im-possible.info/english/articles/awif/chapter2.html

[12] J. McDonnell, *opticalRec*. Blender. Dec. 7, 2021.

[13] *Ambiguous Cylinder Illusion By Kokichi Sugihara.* The Kid Should See This. Accessed on: Dec. 7, 2021. [Online]. Available: https://thekidshouldseethis.com/post/ambiguous-cylinder-illusion-by-kokichi-sugihara

[14] D.Richeson, *Make Your Own Impossible Cylinder.* Youtube. Accessed on: Dec. 7, 2021. [Online]. Available: https://www.youtube.com/watch?v=p5VTqUIBBFE

[15] J. McDonnell, *circleSphere*. Blender. Dec. 7, 2021.

[16] *OpenGL.* Wikipedia. Accessed on: Dec. 7, 2021. [Online]. Available: https://en.wikipedia.org/wiki/OpenGL