Orthographic projections

Isometric

In isometric all distances along the major axes are the same

Oblique
Orthographic Oblique using a shear

- Move $x$ and $y$ proportional to the depth

$$H_x(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
To generate Oblique perspective

\[
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & \cot\theta & 0 \\
0 & 1 & \cot(\varphi) & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]

\(\cot(\theta) = \frac{dx}{dz}\)

\(\cot(\phi) = \frac{dy}{dz}\)
Orthographic projection matrix

The effect is to scale in to a $[-1,+1]$ cube.

$$P = \begin{bmatrix}
\frac{2}{\text{right-left}} & 0 & 0 & -\frac{\text{right-left}}{\text{far}+\text{near}} \\
0 & \frac{2}{\text{top-bottom}} & 0 & -\frac{\text{top-bottom}}{\text{far}+\text{near}} \\
0 & 0 & \frac{\text{far}+\text{near}}{\text{far}+\text{near}} & 1
\end{bmatrix}$$

Note – the mapping into a viewport is actually done in two stages

First we map into a cube. Then we map into the viewport.

Reason:Clipping
The Z buffer algorithm

- Problem, how do we make sure that nearer objects hide further objects?
- A reason why we need to preserve depth after projection.
- The z-buffer contains depth information about an image for every pixel that is rendered between the near and the far clipping plane.
- For each pixel, about to be rendered.
  - If closer than z-buffer depth draw rgb and z value.
  - Else, discard.

It works because the image has already been transformed.
Perspective. How to get the scaling
Brunelleschi’s window
Perspective transformation - simplified

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1/d & 0
\end{bmatrix} \begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]

\[w = z/d\]

- Final step, divide \(x\) and \(y\) by \(w\)

Where \(P\) is the projective transformation matrix (above)

\[M = PS\]

\(S\) is the Scene to Viewport mapping
Two specifications

**glFrustum** (Left, Right, Bottom, Top, Near, Far);

**gluPerspective** (fov, aspect, near far);
The Lookat function from a viewpoint (vp) to a scene point (sp)

```c
gluLookat(vpX, vpY, vpZ, spX, spY, spZ, upX, upY, upZ);
```

Three ways of doing it.

Using gluLookat, or using translate and rotate, or Using change in basis.
Change in basis

- Express one set of axes in the coordinates of another.
- In a simple form this is a rotation.

[Diagram showing coordinate systems X, Y, U, V with point (x,y) and its coordinates (u,v) expressed in the X,Y system.]
• How to get a rotation to view vector
• Use basis method

1) Take difference
2) Normalize
3) Put into matrix form
To deal with the Up Vector in 3D

- Purpose, Rotate view so that some defined up in the scene is aligned with the vertical direction on the screen.
- Calculate view vector $\mathbf{v}$ (difference + normalize)
- Given up vector $\mathbf{u}$ use cross products
- Side ways vector of view coords $\mathbf{s} = \mathbf{u} \times \mathbf{v}$
- New up vector $\mathbf{n} = \mathbf{u} \times \mathbf{s}$

Useful fact: the Transpose of a rotation matrix is its inverse.
How to fly

- Keep moving the viewpoint in the direction of travel
- Rotate according to the view direction.

```c
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glPushMatrix();
    glRotatef(viewAngle*180.0/3.141592, 0,1,0);
    glTranslatef(-eyeX, -eyeY, -eyeZ); // to origin
    scene();
glPopMatrix();
glutSwapBuffers();

df = my*0.01;  // the speed in the forward direction (my from mouse)
viewAngle += mx*0.0001;  // Azimuth in radians
dz = cos(viewAngle)*df;  // dx = sin(viewAngle)*df;
eyeX += dx;  eyeZ -= dz;
```
The outcode function returns an outcode as follows:

- **< Xmin**
- **< Ymin**
- **> Ymax**
- **> Xmax**

Here are the corresponding outcodes:

<table>
<thead>
<tr>
<th>1001</th>
<th>1000</th>
<th>1010</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>0101</td>
<td>0100</td>
<td>0110</td>
</tr>
</tbody>
</table>

**Clipping Cohen Sutherland**
Let \( o_1 \) be the outcode of \( p_1 \)

**Case A:** \( o_1 = o_2 = 0 \); Both inside, no clipping needed

**Case B:** \((o_1 \& o_2) = 0\); all bits same \(\rightarrow\) same region \(\rightarrow\) discard

**Case C:** \((o_1 \text{ XOR } o_1) > 0\); some bits different \(\rightarrow\) different regions, may need clipping