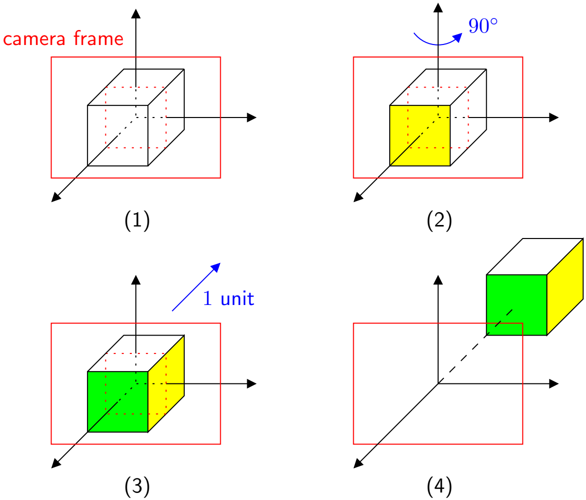
Viewing

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

Look at Objects in a Different Angle

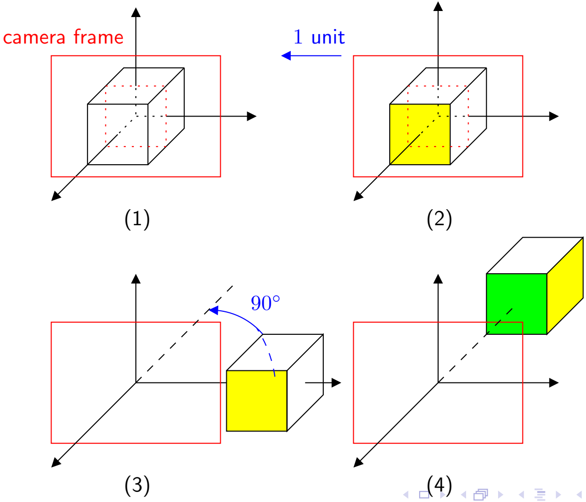
- Recall that by default, the camera frame is at the origin and look into the direction of -z-axis.
- Suppose we want to look at the left side of a cube where its center of mass is at the origin with the distance of 1 unit away from its center of mass. We can achieve this in one of two ways:
    1. Rotate the cube about y-axis for 90 degrees and move the cube 1 unit in the direction of the -z-axis, or
    2. Move the camera to the point (-1, 0, 0) and rotate the object from about y-axis for 90 degree using the point (-1, 0, 0) as the fixed point

- Rotate the cube about y-axis for 90 degrees and move the cube 1 unit in the direction of the -z-axis, or



Look at Objects in a Different Angle

- Move the camera to the point (-1, 0, 0) and rotate the object from about y-axis for 90 degree using the point (-1, 0, 0) as the fixed point



Look at Objects in a Different Angle

- Note that we achieve the same result in both case by applying two transformation:
  - First case: Rotate and Translate $\mathbf{TR}$
  - Second case: Translate and Rotate $\mathbf{RT}$
- In both case, rotations are identical:
  - We rotate about y-axis for 90° using the origin as the fixed point
- However, the translations are not the same:
  - In the first case, we push the cube along the z-axis
  - In the second case, we move the frame along the -x-axis

Camera Frame

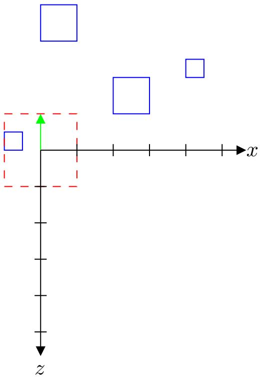
- You can think of viewing as
  1. rotate objects to the desired angle, and
  2. move the object to the desired location

  or

  1. move your camera in the object frame to a desired location, and
  2. point the camera into the desired direction.
- Note that a camera frame is a frame which can be defined by three vectors (axes) u, v, and n.
- What we need is to convert all vertices in object frame (defined by axes x, y, and z) into the camera frame.
- This is the same as change in coordinate discussed in Chapter 3

Camera Frame

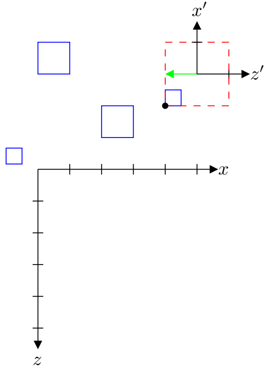
- Consider the top view of a world frame



- We only see a small cube on the left because of the OpenGL canonical view volume
  - $x = \pm 1$, $y = \pm 1$, and $z = \pm 1$

Camera Frame

- Suppose we want to look at this world from the point $(5, 0, -3)$ into the negative x direction

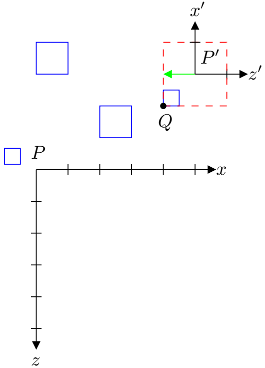


- We should only see a small cube on the left because it is the only object in the OpenGL canonical view volume
- The bottom left of the cube is at $(4, 0, -2)$ in the world frame but it is at $(-1, 0, -1)$ in the camera frame

Camera Frame

- We need to transform all vertices in the world frame into the camera frame
- After the transformation, any objects lie inside $x = \pm 1$, $y = \pm 1$, and $z = \pm 1$ can be seen
- We need to use change of frame discussed in Chapter 3

Camera Frame



- From the above picture, we have

$$
\begin{aligned}
x' &= (0)x + (0)y + (-1)z \\
y' &= (0)x + (1)y + (0)z \\
z' &= (1)x + (0)y + (0)z \\
P' &= (5)x + (0)y + (-3)z + P
\end{aligned}
$$

Change of Frame

- This is equivalent to the following matrix representation:

$$\begin{bmatrix} x' \\ y' \\ z' \\ P' \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 5 & 0 & -3 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ P \end{bmatrix}$$

- In other words,

$$\begin{bmatrix} x' \\ y' \\ z' \\ P' \end{bmatrix} = \mathbf{M} \begin{bmatrix} x \\ y \\ z \\ P \end{bmatrix} \quad \text{where} \quad \mathbf{M} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 5 & 0 & -3 & 1 \end{bmatrix}$$

Change of Frame

- A point Q can be represented in x, y, and z basis vectors as

$$
\begin{aligned}
Q &= ax + by + bz + P \\
&= \begin{bmatrix} a & b & c & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ P \end{bmatrix}
\end{aligned}
$$

- Similarly, the same point Q can be represented in x', y', and z' basis vector as

$$
\begin{aligned}
Q &= a'x' + b'y' + b'z' + P' \\
&= \begin{bmatrix} a' & b' & c' & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \\ P' \end{bmatrix}
\end{aligned}
$$

Change of Frame

- So, we have

$$\begin{bmatrix} a' & b' & c' & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \\ P' \end{bmatrix} = \begin{bmatrix} a & b & c & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ P \end{bmatrix}$$

$$\begin{bmatrix} a' & b' & c' & 1 \end{bmatrix} \mathbf{M} \begin{bmatrix} x \\ y \\ z \\ P \end{bmatrix} = \begin{bmatrix} a & b & c & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ P \end{bmatrix}$$

$$\begin{bmatrix} a' & b' & c' & 1 \end{bmatrix} \mathbf{M} = \begin{bmatrix} a & b & c & 1 \end{bmatrix}$$

$$\mathbf{M}^T \begin{bmatrix} a' \\ b' \\ c' \\ 1 \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}$$

$$(\mathbf{M}^T)^{-1}\mathbf{M}^T \begin{bmatrix} a' \\ b' \\ c' \\ 1 \end{bmatrix} = (\mathbf{M}^T)^{-1} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} a' \\ b' \\ c' \\ 1 \end{bmatrix} = (\mathbf{M}^T)^{-1} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}$$

Change of Frame

- The matrix $(\mathbf{M}^T)^{-1}$ is a transformation matrix that changes a representation in world frame to a representation in camera frame
- From previous example,

$$\mathbf{M} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 5 & 0 & -3 & 1 \end{bmatrix} \rightsquigarrow (\mathbf{M}^T)^{-1} = \begin{bmatrix} 0 & 0 & -1 & -3 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & -5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- The point $Q = (4, 0, -2)$ in the world frame becomes

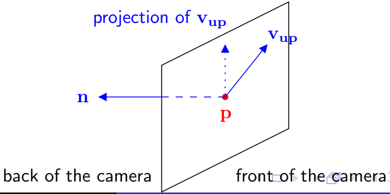
$$\begin{bmatrix} 0 & 0 & -1 & -3 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & -5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 0 \\ -2 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

- However, a camera frame is not easy to define
  - Need to find x', y', and z' axes according to the world frame

Camera Frame

- Traditionally a camera frame is defined by three elements:
  1. **View Reference Point (VRP)** specifies the position of the camera in the object frame
  2. **View-Plane Normal (VPN)** is a vector that specifies the orientation of the back of the camera as plane's normal
  3. **View-Up Vector (VUP)** vector, where it projection to the view plane specifies the up direction of the camera
- We need to derive the camera frame from VRP ($\mathbf{p}$), VPN ($\mathbf{n}$), and VUP ($\mathbf{v_{up}}$) where

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}, \quad \mathbf{n} = \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix}, \quad \text{and } \mathbf{v_{up}} = \begin{bmatrix} v_{up_x} \\ v_{up_y} \\ v_{up_z} \\ 0 \end{bmatrix}$$



Camera Frame

- We can consider the VRP as the origin of the camera frame
- Since n is the plane's normal, we can use this as an axis of the camera frame
- We need two more orthogonal vectors u and v to define the frame which can be derived from $\mathbf{n}$ and $\mathbf{v_{up}}$.
- Again, our goal is to change from the original x, y, z axes to u, v, n axes.
- We need a **model-view matrix** $\mathbf{V}$ that can be used to change a coordinate of $\mathbf{p}$ in object frame (x, y, z) to a coordinate $\mathbf{p}'$ in camera frame (u, v, n)

$$\mathbf{p}' = \mathbf{V}\mathbf{p}$$

Camera Frame

- Change of origin can be handled by the translation matrix $\mathbf{T}$
- Thus, our model-view matrix becomes

$$\mathbf{V} = \mathbf{T}_1\mathbf{R} = \mathbf{R}\mathbf{T}_2$$

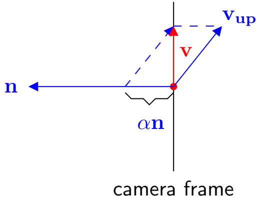
  where $\mathbf{R}$ is a rotation matrix
- Note that the rotation matrix $\mathbf{R}$ will be the same in both case but not the translation matrices $\mathbf{T}$ are not the same
- We are going to consider the second case:
  1. move your camera in the object frame to a desired location, and
  2. point the camera into the desired direction.

Camera Frame

- Lets $\mathbf{v}$ be the projection of $\mathbf{v_{up}}$ to the view plane
- Since v will be another axis, $\mathbf{v}$ must be orthogonal to $\mathbf{n}$. According to the dot-product, we have

$$\mathbf{n} \cdot \mathbf{v} = 0$$

- Consider the situation below



- $\mathbf{v}$ is a linear combination of $\mathbf{n}$ and $\mathbf{v_{up}}$:

$$\mathbf{v} = \alpha\mathbf{n} + \mathbf{v_{up}}$$

Camera Frame

- We have

$$\begin{aligned} \mathbf{v} &= \alpha\mathbf{n} + \mathbf{v_{up}} \\ \mathbf{v}\cdot\mathbf{n} &= (\alpha\mathbf{n} + \mathbf{v_{up}})\cdot\mathbf{n} \\ 0 &= \alpha\mathbf{n}\cdot\mathbf{n} + \mathbf{v_{up}}\cdot\mathbf{n} \\ -\mathbf{v_{up}}\cdot\mathbf{n} &= \alpha\mathbf{n}\cdot\mathbf{n} \\ -\frac{\mathbf{v_{up}}\cdot\mathbf{n}}{\mathbf{n}\cdot\mathbf{n}} &= \alpha \end{aligned}$$

- Therefore

$$\mathbf{v} = \mathbf{v_{up}} - \frac{\mathbf{v_{up}}\cdot\mathbf{n}}{\mathbf{n}\cdot\mathbf{n}}\mathbf{n}$$

- For the third axis $(\mathbf{u})$ can be obtained by the cross-product

$$\mathbf{u} = \mathbf{v}\times\mathbf{n}$$

Camera Frame

- Now we have three axes that define the camera frame

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \\ 0 \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}, \quad \text{and } \mathbf{n} = \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix}$$

- Note that these vectors are not always be unit vectors. We need to normalize them:

$$\mathbf{u}' = \frac{\mathbf{u}}{|\mathbf{u}|} = \frac{1}{|\mathbf{u}|} \begin{bmatrix} u_x \\ u_y \\ u_z \\ 0 \end{bmatrix} = \begin{bmatrix} u'_x \\ u'_y \\ u'_z \\ 0 \end{bmatrix}$$

$$\mathbf{v}' = \frac{\mathbf{v}}{|\mathbf{v}|} = \frac{1}{|\mathbf{v}|} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix} = \begin{bmatrix} v'_x \\ v'_y \\ v'_z \\ 0 \end{bmatrix}$$

$$\mathbf{n}' = \frac{\mathbf{n}}{|\mathbf{n}|} = \frac{1}{|\mathbf{n}|} \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix} = \begin{bmatrix} n'_x \\ n'_y \\ n'_z \\ 0 \end{bmatrix}$$

Camera Frame

- From a Chapter 3 slide, in homogeneous coordinate, we have

$$
\begin{aligned}
u' &= u'_x x + u'_y y + u'_z z \\
v' &= v'_x x + v'_y y + v'_z z \\
n' &= n'_x x + n'_y y + n'_z z \\
Q_0 &= P_0
\end{aligned}
$$

  which is equivalent to

$$
\begin{bmatrix} u' \\ v' \\ n' \\ 0 \end{bmatrix} = \begin{bmatrix} u'_x & u'_y & u'_z & 0 \\ v'_x & v'_y & v'_z & 0 \\ n'_x & n'_y & n'_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \mathbf{M} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}
$$

  where

$$
\mathbf{M} = \begin{bmatrix} u'_x & u'_y & u'_z & 0 \\ v'_x & v'_y & v'_z & 0 \\ n'_x & n'_y & n'_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

- Note that we do not include translation of the origin, it will be handle by a translation matrix

Camera Frame

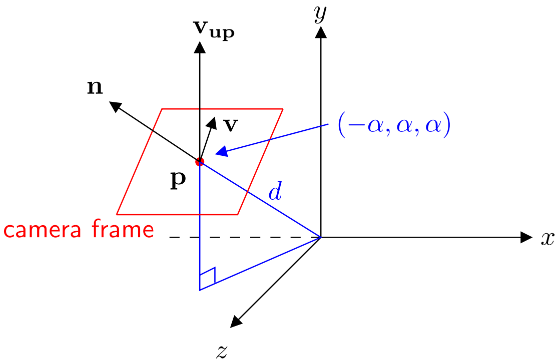
- Recall that $(\mathbf{M}^T)^{-1}$ is a matrix that take us from object frame to camera frame
- Thus we have
$$\mathbf{R} = (\mathbf{M}^T)^{-1}$$
- Since $\mathbf{M}^T$ is a rotation matrix, as discussed in previous chapter, its inverse is simply its transpose. Therefore

$$\mathbf{R} = (\mathbf{M}^T)^{-1} = (\mathbf{M}^T)^T = \mathbf{M} = \begin{bmatrix} u'_x & u'_y & u'_z & 0 \\ v'_x & v'_y & v'_z & 0 \\ n'_x & n'_y & n'_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Example

Problem

Suppose we want to move the camera to a point $(-\alpha, \alpha, \alpha)$ such that the distance from the center of the camera frame to the origin is d. Then orient the camera to point to the origin. What would be the view-model matrix?



Example

- First we need to find the view reference point by simply solve $\sqrt{(-\alpha)^2 + \alpha^2 + \alpha^2} = d$ which gives us $\alpha = \frac{1}{\sqrt{3}}d$. Thus we have

$$\mathbf{p} = \begin{bmatrix} -d/\sqrt{3} \\ d/\sqrt{3} \\ d/\sqrt{3} \\ 0 \end{bmatrix}$$

- Since $\mathbf{n}$ is in the same direction from the origin to the VRP, $\mathbf{n}$ can simply be a vector:

$$\mathbf{n} = \begin{bmatrix} -1 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

- We use y-axis as a reference to the up direction of the camera, therefore the VUP can simply be

$$\mathbf{v_{up}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Example

- Recall that

$$\begin{aligned}
\mathbf{v} &= \mathbf{v_{up}} - \frac{\mathbf{v_{up}} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}}\mathbf{n} \\
&= \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} - \frac{(0)(-1)+(1)(1)+(0)(1)+(0)(0)}{(-1)(-1)+(1)(1)+(1)(1)+(0)(0)} \begin{bmatrix} -1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} - \frac{1}{3} \begin{bmatrix} -1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 2/3 \\ -1/3 \\ 0 \end{bmatrix}
\end{aligned}$$

- Note that $|\mathbf{v}| = \sqrt{(1/3)^2 + (2/3)^2 + (-1/3)^2} = \sqrt{6}/3$, thus

$$\mathbf{v}' = \frac{1}{\sqrt{6}/3} \begin{bmatrix} 1/3 \\ 2/3 \\ -1/3 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{6} \\ 2/\sqrt{6} \\ -1/\sqrt{6} \\ 0 \end{bmatrix}$$

Example

- Recall that $\mathbf{u} = \mathbf{v} \times \mathbf{n}$ and

$$\mathbf{v} \times \mathbf{n} = \begin{bmatrix} v_y n_z - v_z n_y \\ v_z n_x - v_x n_z \\ v_x n_y - v_y n_x \\ 0 \end{bmatrix} = \begin{bmatrix} (2/3)(1) - (-1/3)(1) \\ (-1/3)(-1) - (1/3)(1) \\ (1/3)(1) - (2/3)(-1) \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \mathbf{u}$$

- $|\mathbf{u}| = \sqrt{1^2 + 0^2 + 1^2} = \sqrt{2}$, thus

$$\mathbf{u}' = \frac{\mathbf{u}}{|\mathbf{u}|} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} \\ 0 \\ 1/\sqrt{2} \\ 0 \end{bmatrix}$$

- Note that $|\mathbf{n}| = \sqrt{1^2 + 1^2 + 1^2} = \sqrt{3}$, thus

$$\mathbf{n}' = \frac{\mathbf{n}}{|\mathbf{n}|} = \frac{1}{\sqrt{3}} \begin{bmatrix} -1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1/\sqrt{3} \\ 1/\sqrt{3} \\ 1/\sqrt{3} \\ 0 \end{bmatrix}$$

Example

- This is the same as change of frame where we change from the original bases x, y, and z to a new bases u, v, and n where

$$
\begin{aligned}
u' &= \frac{1}{\sqrt{2}}x + (0)y + \frac{1}{\sqrt{2}}z \\
v' &= \frac{1}{\sqrt{6}}x + \frac{2}{\sqrt{6}}y - \frac{1}{\sqrt{6}}z \\
n' &= -\frac{1}{\sqrt{3}}x + \frac{1}{\sqrt{3}}y + \frac{1}{\sqrt{3}}z \\
Q_0 &= -\frac{d}{\sqrt{3}}x + \frac{d}{\sqrt{3}}y + \frac{d}{\sqrt{3}}z + P_0
\end{aligned}
$$

which is equivalent to

$$
\begin{bmatrix} u \\ v \\ n \\ Q_0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & 0 & 1/\sqrt{2} & 0 \\ 1/\sqrt{6} & 2/\sqrt{6} & -1/\sqrt{6} & 0 \\ -1/\sqrt{3} & 1/\sqrt{3} & 1/\sqrt{3} & 0 \\ -d/\sqrt{3} & d/\sqrt{3} & d/\sqrt{3} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ P_0 \end{bmatrix} = \mathbf{M} \begin{bmatrix} x \\ y \\ z \\ P_0 \end{bmatrix}
$$

where

$$
\mathbf{M} = \begin{bmatrix} 1/\sqrt{2} & 0 & 1/\sqrt{2} & 0 \\ 1/\sqrt{6} & 2/\sqrt{6} & -1/\sqrt{6} & 0 \\ -1/\sqrt{3} & 1/\sqrt{3} & 1/\sqrt{3} & 0 \\ -d/\sqrt{3} & d/\sqrt{3} & d/\sqrt{3} & 1 \end{bmatrix}
$$

Example

- Recall that $(\mathbf{M}^T)^{-1}$ is a matrix that translate points in the bases x, y, z to the bases u', v', n'
- From the matrix $\mathbf{M}$ we have

$$\mathbf{M}^T = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{6} & -1/\sqrt{3} & -d/\sqrt{3} \\ 0 & 2/\sqrt{6} & 1/\sqrt{3} & d/\sqrt{3} \\ 1/\sqrt{2} & -1/\sqrt{6} & 1/\sqrt{3} & d/\sqrt{3} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- The augmented matrix of $\mathbf{M}^T$ is as follows:

$$\left[\begin{array}{cccc|cccc} 1/\sqrt{2} & 1/\sqrt{6} & -1/\sqrt{3} & -d/\sqrt{3} & 1 & 0 & 0 & 0 \\ 0 & 2/\sqrt{6} & 1/\sqrt{3} & d/\sqrt{3} & 0 & 1 & 0 & 0 \\ 1/\sqrt{2} & -1/\sqrt{6} & 1/\sqrt{3} & d/\sqrt{3} & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

- Subtract row 3 by row 1:

$$\left[\begin{array}{cccc|cccc} 1/\sqrt{2} & 1/\sqrt{6} & -1/\sqrt{3} & -d/\sqrt{3} & 1 & 0 & 0 & 0 \\ 0 & 2/\sqrt{6} & 1/\sqrt{3} & d/\sqrt{3} & 0 & 1 & 0 & 0 \\ 0 & -2/\sqrt{6} & 2/\sqrt{3} & 2d/\sqrt{3} & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

Example

- Subtract row 1 by (1/2) times row 2:

$$\left[\begin{array}{cccc|cccc} 1/\sqrt{2} & 0 & -\sqrt{3}/2 & -d\sqrt{3}/2 & 1 & -1/2 & 0 & 0 \\ 0 & 2/\sqrt{6} & 1/\sqrt{3} & d/\sqrt{3} & 0 & 1 & 0 & 0 \\ 0 & -2/\sqrt{6} & 2/\sqrt{3} & 2d/\sqrt{3} & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

- Add row 3 by row 2:

$$\left[\begin{array}{cccc|cccc} 1/\sqrt{2} & 0 & -\sqrt{3}/2 & -d\sqrt{3}/2 & 1 & -1/2 & 0 & 0 \\ 0 & 2/\sqrt{6} & 1/\sqrt{3} & d/\sqrt{3} & 0 & 1 & 0 & 0 \\ 0 & 0 & \sqrt{3} & d\sqrt{3} & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

- Add row 1 by (1/2) times row 3:

$$\left[\begin{array}{cccc|cccc} 1/\sqrt{2} & 0 & 0 & 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 2/\sqrt{6} & 1/\sqrt{3} & d/\sqrt{3} & 0 & 1 & 0 & 0 \\ 0 & 0 & \sqrt{3} & d\sqrt{3} & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

Example

- Subtract row 2 by $(1/3)$ times row 3:

$$\left[\begin{array}{cccc|cccc} 1/\sqrt{2} & 0 & 0 & 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 2/\sqrt{6} & 0 & 0 & 1/3 & 2/3 & -1/3 & 0 \\ 0 & 0 & \sqrt{3} & d\sqrt{3} & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

- Subtract row 3 by $d\sqrt{3}$ times row 4:

$$\left[\begin{array}{cccc|cccc} 1/\sqrt{2} & 0 & 0 & 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 2/\sqrt{6} & 0 & 0 & 1/3 & 2/3 & -1/3 & 0 \\ 0 & 0 & \sqrt{3} & 0 & -1 & 1 & 1 & -d\sqrt{3} \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

- Multiply row 1 by $\sqrt{2}$, multiply row 2 by $\sqrt{6}/2$, and multiply row 3 by $1/\sqrt{3}$:

$$\left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & \sqrt{2}/2 & 0 & \sqrt{2}/2 & 0 \\ 0 & 1 & 0 & 0 & \sqrt{6}/6 & \sqrt{6}/3 & -\sqrt{6}/6 & 0 \\ 0 & 0 & 1 & 0 & -\sqrt{3}/3 & \sqrt{3}/3 & \sqrt{3}/3 & -d \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

Example

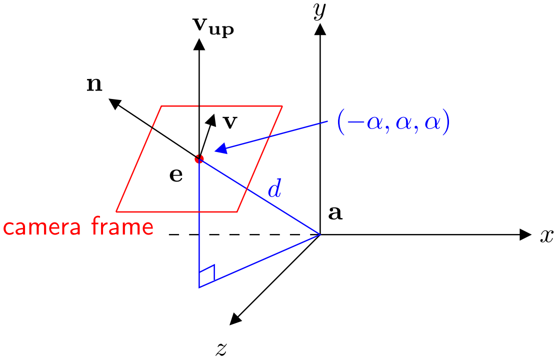
- We have

$$(\mathbf{M}^T)^{-1} = \begin{bmatrix} \sqrt{2}/2 & 0 & \sqrt{2}/2 & 0 \\ \sqrt{6}/6 & \sqrt{6}/3 & -\sqrt{6}/6 & 0 \\ -\sqrt{3}/3 & \sqrt{3}/3 & \sqrt{3}/3 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- The matrix $(\mathbf{M}^T)^{-1}$ can be used to change coordinates of vertices defined in the bases x, y, z to coordinates in bases u, v, n.

Look At

- It may be simpler to visualize the way we look at the object



- Our eyes are at the point $\mathbf{e}$ and look at the point $\mathbf{a}$ with the up direction is indicated by $\mathbf{v_{up}}$ vector.
- We can use the same idea to construct the model-view matrix

Look At

- The view-plane normal $\mathbf{vpn}$ is simply the vector $\mathbf{e} - \mathbf{a}$ which can be normalized to

$$\mathbf{n} = \frac{\mathbf{vpn}}{|\mathbf{vpn}|}$$

- The normalized $\mathbf{u}$ can be calculated by

$$\mathbf{u} = \frac{\mathbf{v_{up}} \times \mathbf{n}}{|\mathbf{v_{up}} \times \mathbf{n}|}$$

- Similarly, the normalized $\mathbf{v}$ vector can be calculated by

$$\mathbf{v} = \frac{\mathbf{n} \times \mathbf{u}}{|\mathbf{n} \times \mathbf{u}|}$$

- Finally, the view-reference point (VRP) is simply the eye point $\mathbf{e}$
- Use $\mathbf{u}$, $\mathbf{v}$, $\mathbf{n}$, and $\mathbf{p}$ to construct the model-view matrix as discussed earlier

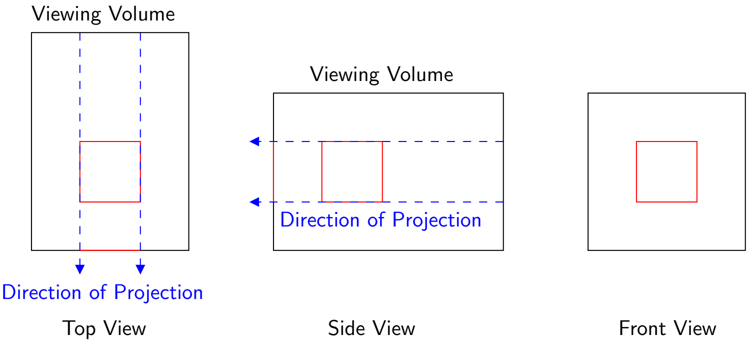
Look At

- You should implement a function named `look_at` that returns a model-view matrix based on camera's position and orientation:

```
mat4 look_at(GLfloat eyex, GLfloat eyey, GLfloat eyez,
             GLfloat atx, GLfloat aty, GLfloat atz,
             GLfloat upx, GLfloat upy, GLfloat upz);
```

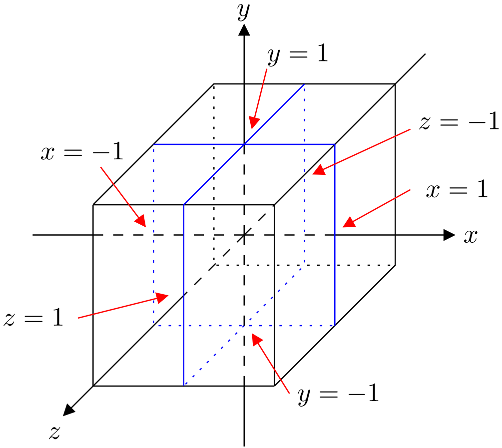
Parallel Projections

- Orthogonal Projection



Canonical View Volume

- OpenGL **canonical view volume** is a cube defined by the planes $x = \pm 1$, $y = \pm 1$, and $z = \pm 1$.

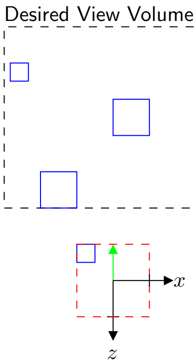


Canonical View Volume

- OpenGL canonical view volume can be defined by six variables:
  - `left = -1.0`
  - `right = 1.0`
  - `bottom = -1.0`
  - `top = 1.0`
  - `near = 1.0`
  - `far = -1.0`
- **Note** that textbook uses `near = -1.0` and `far = 1.0`
  - It is up to you how you want to define `near` and `far`
  - Your implementation must follow your definitions of `near` and `far`
  - For our discussion, in case of canonical view volume, `near` will be 1.0 and `far` will be -1.0

Orthogonal-Projection Matrices

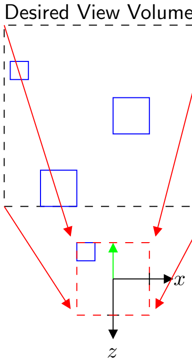
- Generally we want to define our own viewing volume as shown below:



- This is the same as zooming in and out

Orthogonal-Projection Matrices

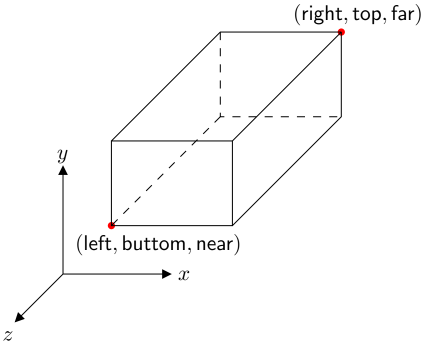
- A general idea is to translate and scale our desired view volume to fit into the OpenGL canonical view volume



- But first, we need a way to define a desired view volume

Orthogonal-Projection Matrices

- Generally we want to define our own viewing volume as shown below:



- The viewing volume above is defined by two points (left, bottom, near) and (right, top, far).

Orthogonal-Projection Matrices

- All variables simply associated with actual camera frame coordinates (x, y, z)
- Our goal is to fit user-defined view volume into the OpenGL canonical view volume
- This can be achieve by
  1. Translate the center of mass of the user-defined view volume to the center of mass of canonical view volume (origin)
  2. Scale the user-defined view volume to fit into the OpenGL canonical view volume

Translating the Center of Mass

- Let (x, y, z) be the center of mass of the user-defined viewing volume. We have

$$x = \text{left} + \frac{\text{right} - \text{left}}{2} = \frac{2\text{left} + \text{right} - \text{left}}{2} = \frac{\text{right} + \text{left}}{2}$$

$$y = \text{bottom} + \frac{\text{top} - \text{bottom}}{2} = \frac{2\text{bottom} + \text{top} - \text{bottom}}{2} = \frac{\text{top} + \text{bottom}}{2}$$

$$z = \texttt{far} + \frac{\texttt{near} - \texttt{far}}{2} + \frac{2\text{far} + \text{near} - \text{far}}{2} = \frac{\texttt{near} + \texttt{far}}{2}$$

- Thus, the translation matrix $\mathbf{T} = \mathbf{T}(-x, -y, -z)$ is as follows:

$$\mathbf{T} = \mathbf{T}(-x, -y, -z) = \begin{bmatrix} 1 & 0 & 0 & -\frac{\text{right}+\text{left}}{2} \\ 0 & 1 & 0 & -\frac{\text{top}+\text{bottom}}{2} \\ 0 & 0 & 1 & -\frac{\text{near}+\text{far}}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling the View Volume

- Next, we need to scale it to fit into the OpenGL viewing volume
- The width of the OpenGL viewing volume is 2.0 and the width of the user-define viewing volume is right - left
- Thus, we need to scale it by the factor of $\frac{2}{\text{right}-\text{left}}$
- Same for height and depth which are $\frac{2}{\text{top}-\text{bottom}}$ and $\frac{2}{\text{near}-\text{far}}$, respectively.
- Thus, the scaling matrix $\mathbf{S}$ is as follows:

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

Orthogonal-Projection Matrices

- Since we need to translate and then scale, our projection matrix $\mathbf{N} = \mathbf{ST}$

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

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

- It is a good idea to have a function named `ortho` that returns a orthogonal-projection matrix $\mathbf{N}$

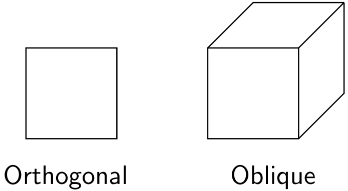
```
mat4 ortho(GLfloat left, GLfloat right,
           GLfloat bottom, GLfloat top,
           GLfloat near, GLfloat far);
```

- OpenGL original orthogonal-projection matrix is

```
ortho(-1.0, 1.0, -1.0, 1.0, 1.0, -1.0);
```

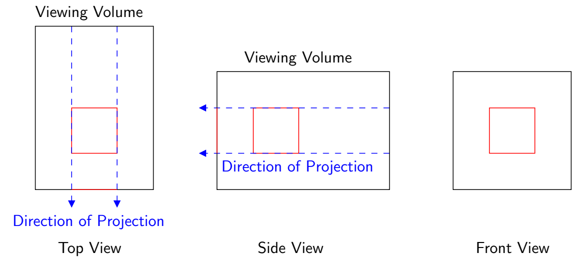
Oblique Projection

- In orthogonal projection, if we look directly at the front of the cube we have no idea that it is a cube

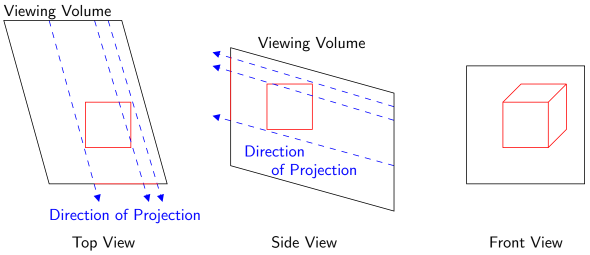


- Oblique projection is very useful (we use it all the time)
- Generally, our view is orthogonal to the projection plane
- Oblique effect can be seen if we look at the projection plane in a angle
    - Angle that the projector make with the projection plane

Oblique Projection



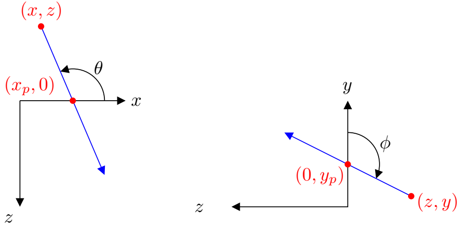
Orthogonal Projection



Oblique Projection

Oblique Projection

- Consider the top view and the side view of a point (x, y, z)



- We need to find a formula to translate the point (x, z) to $(x_p, 0)$ and from (z, y) to $(0, y_p)$

$$\tan\theta = \frac{z}{x_p - x} \text{ and } \tan\phi = \frac{z}{y_p - y}$$

therefore,

$$x_p = x + z\cot\theta \text{ and } y_p = y + z\cot\phi$$

- Note that the value of z in the above picture is less than 0 which corresponds to the value of $\tan\theta$ and $\tan\phi$ when θ and ϕ are greater than 90° and $1/\tan\theta = \cot\theta$.

Oblique Projection

- The value of x and y are changed based on the value of z and angles which is equivalent to

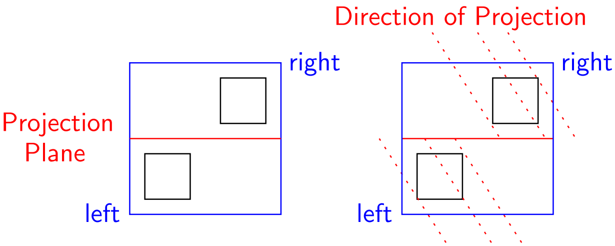
$$\mathbf{P} = \begin{bmatrix} 1 & 0 & \cot\theta & 0 \\ 0 & 1 & \cot\phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which is equivalent to

$$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{H}(\theta,\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \cot\theta & 0 \\ 0 & 1 & \cot\phi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Oblique Projection

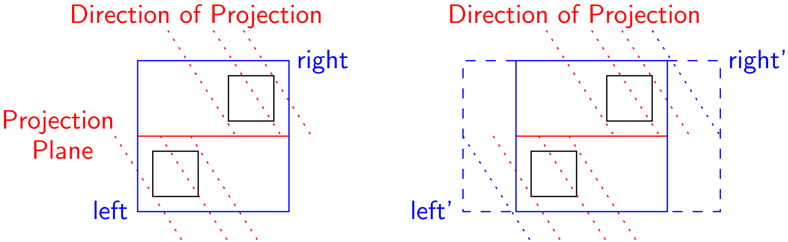
- Let's consider the top view of two cubes in a view volume:



- From the above situation:
  - we can only see half of the front of the cube in the front.
  - we can only see half of the right side of the cube in the back.
- We need to change the view volume a little bit

Oblique Projection

- The original view volume can be changed by the same shear



- From the above picture

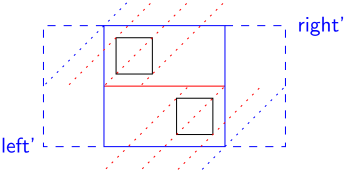
$$\text{left}' = \text{left} + \text{near} \cot \theta$$
$$\text{right}' = \text{right} + \text{far} \cot \theta$$

- Similarly,

$$\text{bottom}' = \text{bottom} + \text{near} \cot \phi$$
$$\text{top}' = \text{top} + \text{far} \cot \phi$$

Oblique Projection

- However, near' and far' is different if the direction of projection $\theta < 90$ change as shown below



- In the above case, left' depends on the value of far and right' depends on the value of near

$$\text{left}' = \text{left} + \text{far} \cot\theta$$

$$\text{right}' = \text{right} + \text{near} \cot\theta$$

- Similarly,

$$\text{bottom}' = \text{bottom} + \text{far} \cot\phi$$

$$\text{top}' = \text{top} + \text{near} \cot\phi$$

Model View and Projection Matrices in OpenGL

- Similarly to transformation matrix, model view and projection matrices will be sent to the vertex shader as two uniform variables of type `mat4`

```
#version 130

in vec4 vPosition;
in vec4 vColor;
out vec4 color;

uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;

void main()
{
    color = vColor;
    gl_Position = projection_matrix * model_view_matrix * vPosition;
}
```

- **Note** that we apply the model view matrix first

Model View and Projection Matrices in OpenGL

- In the `init()` function, we need to locate those variables in the vertex shader:

```
model_view_location = glGetUniformLocation(program,
                                            "model_view_matrix");
projection_location = glGetUniformLocation(program,
                                            "projection_matrix");
```

  where `model_view_location` and `projection_location` are global variables of type `GLuint`

- As usual, simply send both matrices to the vertex shader before drawing:

```
    glClear(...);

    glUniformMatrix4fv(model_view_location, 1, GL_FALSE,
                       (GLfloat *) &model_view);
    glUniformMatrix4fv(projection_location, 1, GL_FALSE,
                       (GLfloat *) &projection);

    glDrawArrays(...);
    glutSwapBuffers();
```

Model View and Projection Matrices in OpenGL

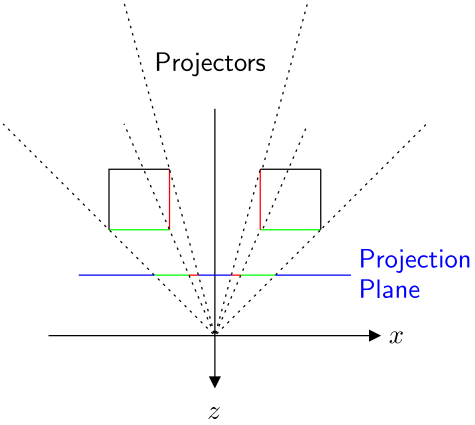
- The model view matrix is the matrix generated by the `look_at()` function

```
mat4 look_at(GLfloat eyex, GLfloat eyey, GLfloat eyez,
             GLfloat atx, GLfloat aty, GLfloat atz,
             GLfloat upx, GLfloat upy, GLfloat upz);
```

- In case of orthographic projection, the projection matrix is the matrix generated by the `ortho()` function

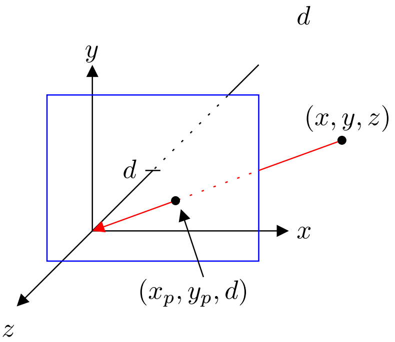
```
mat4 ortho(GLfloat left, GLfloat right,
           GLfloat bottom, GLfloat top,
           GLfloat near, GLfloat far);
```

- Consider the situation where the center of projection is at the origin:



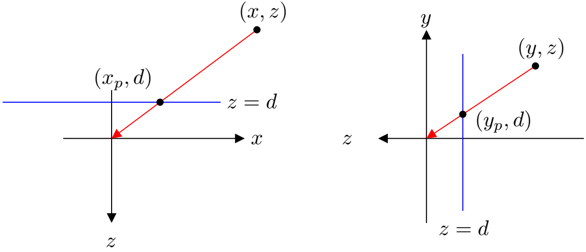
Simple Perspective Projection

- All projectors pass through the origin
- Projection plane is at the plane $z = d$ where $d < 0$
- A point (x, y, z) is projected to the point (x_p, y_p, d)



Simple Perspective Projection

- Consider the top view and the side view



- From above, we have

$$\frac{x}{z} = \frac{x_p}{d} \quad \rightsquigarrow \quad x_p = \frac{x}{z/d}$$

and

$$\frac{y}{z} = \frac{y_p}{d} \quad \rightsquigarrow \quad y_p = \frac{y}{z/d}$$

- Note that the value of x_p and y_p depend on z. The further the object from the origin, the smaller it appears

Simple Perspective Projection

- All points are moved to the plane $z = d$.
  - We lost all z values which causes problem with hidden-surface removal.
- Instead of representing a point by $(x, y, z, 1)$, use (wx, wy, wz, w) instead.
  - To obtain the original point, simply divided by w $(w \neq 0)$
- Consider the matrix $\mathbf{M}$ and a point $\mathbf{p}$ as follows:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \qquad \mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

  we have

$$\mathbf{Mp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \mathbf{q}$$

- Note that the fourth component of $\mathbf{q}$ is not 1.

Simple Perspective Projection

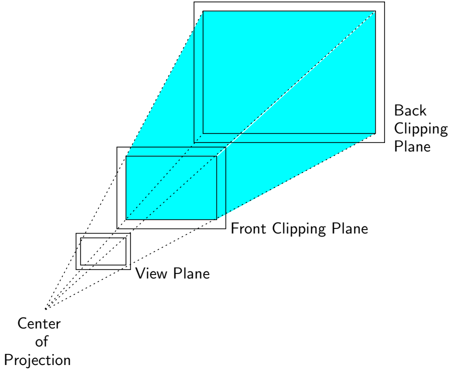
- But if we multiply $\mathbf{q}$ by $\frac{1}{z/d}$, we have

$$\frac{1}{z/d}\mathbf{q} = \frac{1}{z/d}\begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ d \\ 1 \end{bmatrix}$$

  which is what we want as discussed earlier.
- By using this method, we need to perform **perspective division** at the end to obtain the correct representation of points.
- **Note** that OpenGL already support homogeneous coordinates where the fourth component are not 1

Viewing Volume

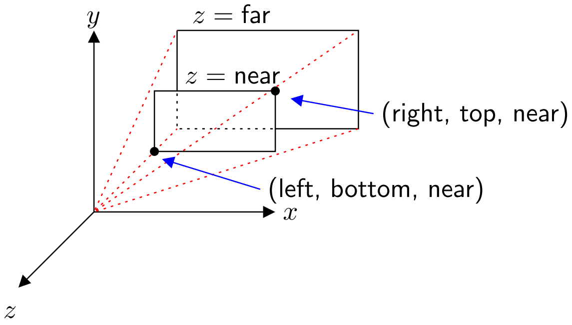
- Consider a viewing volume in a perspective projection



- The above viewing volume is a frustum
- We want a way to specify a viewing volume as a frustum

Specification of a Frustum

- Consider a viewing volume below:

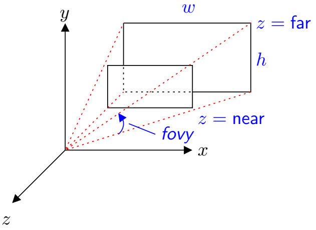


- We can specify a frustum using six values, left, right, bottom, top, near, and far.
- It is a good idea to have a function that returns a perspective model-view matrix defined in a form of a frustum:

```
mat4 frustum(GLfloat left, GLfloat right, GLfloat bottom,
             GLfloat top, GLfloat near, GLfloat far);
```

Field of View

- Often time, we define the specification of perspective viewing as a field of view



- This field of view can be defined by four variables:

```
mat4 perspective(GLfloat fovy, GLfloat aspect,
                 GLfloat near, GLfloat far);
```

  where
  - `fovy` is an angle between top and bottom planes
  - `aspect` is a ratio between the width and the height of view volume
- It is pretty straightforward to convert field of view into frustum.

Perspective Normalization

- Recall that our simple perspective-projection matrix

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

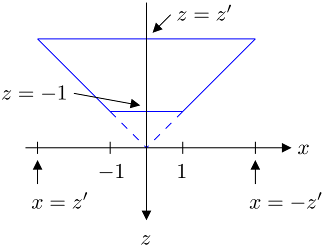
- If the projection plane is at $z = -1$ $(d = -1)$ we have:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

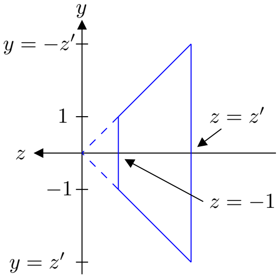
- We need to define a clipping volume

Perspective Normalization

- Let's consider a simple clipping volume as shown below:



Top View



Side View

- The left and the right planes of the frustum is defined by $x = z$ and $x = -z$
- Similarly, the top and the bottom planes is defined by $y = -z$ and $y = z$

Perspective Normalization

- Recall that a point (x, y, z) is projected to a new point (x_p, y_p, d) where

$$x_p = \frac{x}{z/d} \text{ and } y_p = \frac{y}{z/d}$$

- Since $d = -1$, we have $x_p = -x/z$ and $y_p = -y/z$
- Any point (x, y, z) where $-z \leq x \leq z$ and $-z \leq y \leq z$ get projected to $(x_p, y_p, -1)$ where $-1 \leq x_p \leq 1$ and $-1 \leq y_p \leq 1$ (canonical view volume)
- But we have to deal with near and far

Perspective Projection Matrices

- Consider the matrix

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- This matrix transform a point $p = (x, y, z, 1)$ to and new point $q = (x', y', z', w')$

$$\mathbf{Np} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \alpha z + \beta \\ -z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{q}$$

- Multiply $\mathbf{q}$ by $1/w'$ we have

$$\frac{1}{w'}\mathbf{q} = -\frac{1}{z} \begin{bmatrix} x \\ y \\ \alpha z + \beta \\ -z \end{bmatrix} = \begin{bmatrix} -x/z \\ -y/z \\ -(\alpha + \beta/z) \\ 1 \end{bmatrix} = \begin{bmatrix} x'' \\ y'' \\ z'' \\ 1 \end{bmatrix}$$

Perspective Projection Matrices

- Note that x'' and y'' are what we need
- However, we need to find the right value for α and β such that
  - For a point where $z = \text{near}$, it is changed to $z'' = 1$, and
  - for a point where $z = \text{far}$, it is changed to $z'' = -1$

  to fit in OpenGL canonical view volume
- So, we have

$$1 = -(\alpha + \frac{\beta}{\text{near}}) \text{ and } -1 = -(\alpha + \frac{\beta}{\text{far}})$$

- Let's solve the above equations to find the value of α and β in terms of near and far

$$\begin{aligned} 1 &= -(\alpha + \frac{\beta}{\text{near}}) \\ &= -\alpha - \frac{\beta}{\text{near}} \\ \alpha &= -1 - \frac{\beta}{\text{near}} \end{aligned}$$

Perspective Projection Matrices

- Substitute the value of α

$$
\begin{aligned}
-1 &= -((-1 - \frac{\beta}{\text{near}}) + \frac{\beta}{\text{far}}) \\
&= 1 + \frac{\beta}{\text{near}} - \frac{\beta}{\text{far}} \\
&= 1 + \beta(\frac{1}{\text{near}} - \frac{1}{\text{far}}) \\
&= 1 + \beta\frac{\text{far} - \text{near}}{\text{near} \times \text{far}} \\
-2 &= \beta\frac{\text{far} - \text{near}}{\text{near} \times \text{far}} \\
\beta &= -\frac{2 \times \text{near} \times \text{far}}{\text{far} - \text{near}}
\end{aligned}
$$

- Recall the value of α

$$
\begin{aligned}
\alpha &= -1 - \frac{\beta}{\text{near}} \\
&= -1 + \frac{2 \times \text{near} \times \text{far}}{(\text{far} - \text{near}) \times \text{near}} \\
&= -1 + \frac{2 \times \text{far}}{\text{far} - \text{near}} = \frac{-\text{far} + \text{near} + (2 \times \text{far})}{\text{far} - \text{near}} = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}
\end{aligned}
$$

Perspective Projection Matrix

- Thus, our matrix $\mathbf{N}$ becomes

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{near+far}{far-near} & -\frac{2\times near\times far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- This matrix should take any points at $z = near$ to $z = 1.0$ and $z = far$ to $z = -1.0$ (canonical view volume)

Perspective Projection Matrix

- Let's check for a point $(x, y, near)$:

$$\mathbf{N}\begin{bmatrix} x \\ y \\ near \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{near+far}{far-near} & -\frac{2\times near\times far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \\ near \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} x \\ y \\ \frac{near+far}{far-near}near - \frac{2\times near\times far}{far-near} \\ -near \end{bmatrix}$$

$$= \begin{bmatrix} x \\ y \\ \frac{(near\times near)+(near\times far)-2\times near\times far}{far-near} \\ -near \end{bmatrix}$$

$$= \begin{bmatrix} x \\ y \\ \frac{(near\times near)-(near\times far)}{far-near} \\ -near \end{bmatrix} = \begin{bmatrix} x \\ y \\ \frac{near(near-far)}{far-near} \\ -near \end{bmatrix} = \begin{bmatrix} x \\ y \\ -near \\ -near \end{bmatrix}$$

- Once you divide by $-near$, you get the z component to be 1.0 as expected.

Perspective Projection Matrix

- Let's check for a point (x, y, far):

$$\mathbf{N}\begin{bmatrix} x \\ y \\ far \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{near+far}{far-near} & -\frac{2\times near\times far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \\ far \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} x \\ y \\ \frac{near+far}{far-near}far - \frac{2\times near\times far}{far-near} \\ -far \end{bmatrix}$$

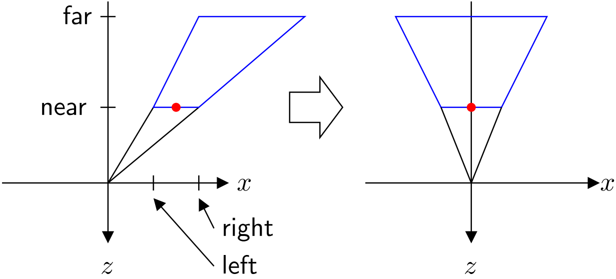
$$= \begin{bmatrix} x \\ y \\ \frac{(near\times far)+(far\times far)-2\times near\times far}{far-near} \\ -far \end{bmatrix}$$

$$= \begin{bmatrix} x \\ y \\ \frac{(far\times far)-(near\times far)}{far-near} \\ -far \end{bmatrix} = \begin{bmatrix} x \\ y \\ \frac{far(far-near)}{far-near} \\ -far \end{bmatrix} = \begin{bmatrix} x \\ y \\ far \\ -far \end{bmatrix}$$

- Once you divide by $-far$, you get the z component to be -1.0 as expected.

OpenGL Perspective Transformation

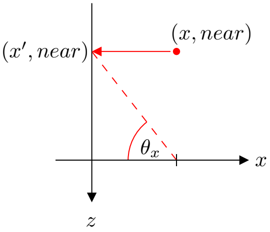
- A view volume in a form of a frustum does not have to be a symmetric frustum
- We need to shear it into a symmetric one (see the top view of a frustum)



- This can be done by shearing the center of the near plane of the frustum to $x = 0$
- Similarly, we need to shear the center of the near plane of the frustum to $y = 0$

OpenGL Perspective Transformation

- Consider the top view



- From the image above, $z_0 = 0$, we have

$$\tan\theta_x = \frac{z - z_0}{x' - x} = \frac{near - 0}{x' - x}$$

- If $x = \frac{left+right}{2}$ and $x' = 0$, we have

$$\tan\theta_x = \frac{near}{0 - \frac{left+right}{2}} = \frac{-2 \times near}{left + right} \rightsquigarrow \cot\theta_x = \frac{left + right}{-2 \times near}$$

- Thus, we have

$$\theta_x = \cot^{-1}\left(\frac{left + right}{-2 \times near}\right)$$

OpenGL Perspective Transformation

- Similarly,
$$\phi_y = \cot^{-1}\left(\frac{bottom + top}{-2 \times near}\right)$$
- The required shear matrix is

$$\mathbf{H}_z(\theta_x, \phi_y) = \mathbf{H}_z\left(\cot^{-1}\left(\frac{left + right}{-2 \times near}\right), \cot^{-1}\left(\frac{bottom + top}{-2 \times near}\right)\right)$$
$$= \begin{bmatrix} 1 & 0 & \frac{left+right}{-2\times near} & 0 \\ 0 & 1 & \frac{bottom+top}{-2\times near} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

OpenGL Perspective Transformation

- The point $(left, y, near)$ is moved to

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

$$
= \begin{bmatrix} left + \frac{left+right}{-2} \\ y + \frac{bottom+top}{-2} \\ near \\ 1 \end{bmatrix}
$$

$$
= \begin{bmatrix} \frac{-2left+left+right}{-2} \\ y + \frac{bottom+top}{-2} \\ near \\ 1 \end{bmatrix}
$$

$$
= \begin{bmatrix} \frac{right-left}{-2} \\ y + \frac{bottom+top}{-2} \\ near \\ 1 \end{bmatrix}
$$

OpenGL Perspective Transformation

- The point $(right, y, near)$ is moved to

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

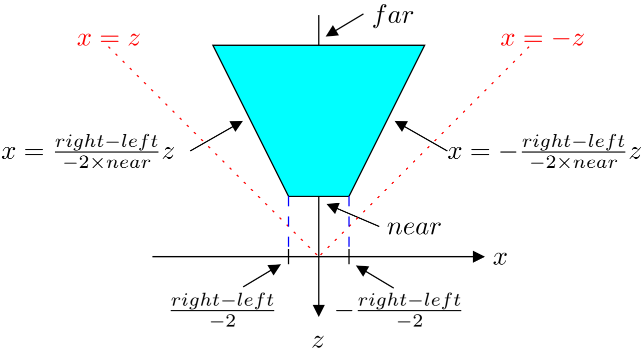
$$= \begin{bmatrix} right + \frac{left+right}{-2} \\ y + \frac{bottom+top}{-2} \\ near \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{-2right+left+right}{-2} \\ y + \frac{bottom+top}{-2} \\ near \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} -\frac{right-left}{-2} \\ y + \frac{bottom+top}{-2} \\ near \\ 1 \end{bmatrix}$$

OpenGL Perspective Transformation

- Let's look at the top view of the frustum after shearing:



- We need to scale the frustum so that its sides are $x = \pm z$
- Similarly, the sides of the frustum are $y = \pm\frac{top-bottom}{-2\times near}$ which must be scaled to $y = \pm z$.

OpenGL Perspective Transformation

- This corresponds to the scaling matrix

$$\mathbf{S}(\frac{-2 \times near}{right - left}, \frac{-2 \times near}{top - bottom}, 1) = \begin{bmatrix} \frac{-2 \times near}{right - left} & 0 & 0 & 0 \\ 0 & \frac{-2 \times near}{top - bottom} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Recall that we have to perform the following:
  - shear the frustum $(\mathbf{H})$
  - scale the frustum $(\mathbf{S})$
  - fit the frustum into canonical view volume $(\mathbf{N})$

  in that order
- Thus, our desired perspective projection matrix $\mathbf{P} = \mathbf{NSH}$

OpenGL Perspective Transformation

- Our final perspective projection matrix

$$\mathbf{P}' = \mathbf{SH} = \begin{bmatrix} \frac{-2\times near}{right-left} & 0 & 0 & 0 \\ 0 & \frac{-2\times near}{top-bottom} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \frac{left+right}{-2\times near} & 0 \\ 0 & 1 & \frac{bottom+top}{-2\times near} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

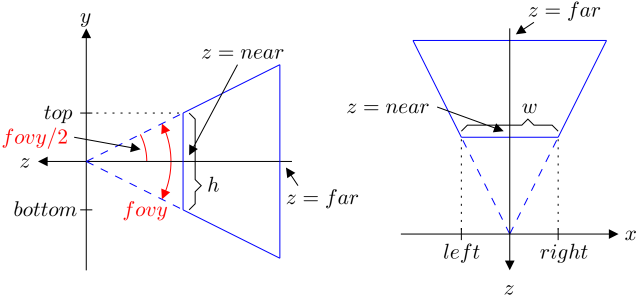
$$= \begin{bmatrix} \frac{-2\times near}{right-left} & 0 & \frac{left+right}{right-left} & 0 \\ 0 & \frac{-2\times near}{top-bottom} & \frac{bottom+top}{top-bottom} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

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

OpenGL Perspective Transformation

- As mentioned earlier, a perspective projection matrix defined by a field of view can be converted into a frustum.
- Consider a side view of a frustum defined by a field of view:



- From above image, we can conclude the following:
  - $bottom = -top$ and also $left = -right$
  - $top = near \times \tan(fovy/2)$
  - $right = top/aspect$

OpenGL Perspective Transformation

- Thus, our perspective projection matrix becomes:

$$\mathbf{P} = \mathbf{NSH} = \begin{bmatrix} -\frac{near}{right} & 0 & 0 & 0 \\ 0 & -\frac{near}{top} & 0 & 0 \\ 0 & 0 & \frac{near+far}{far-near} & -\frac{2\times near\times far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} -\frac{\cot(fovy/2)}{aspect} & 0 & 0 & 0 \\ 0 & -\cot(fovy/2) & 0 & 0 \\ 0 & 0 & \frac{near+far}{far-near} & -\frac{2\times near\times far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

OpenGL Perspective Transformation

- Note that we need to incorporate our model view and projection matrices into our vertex shader
- Thus, our current vertex shader becomes:

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform mat4 model_view;
uniform mat4 projection;
void main()
{
  gl_Position = projection * model_view * vPosition / vPosition.w;
  color = vColor;
}
```

OpenGL Hidden Surface Removal

- Enable depth buffer and hidden-surface removal:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glEnable(GL_DEPTH_TEST);
```

- Need to clear depth buffer before rendering:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

- Tell OpenGL not to render a surface if its normal points away from the viewer:

```
glEnable(GL_CULL_FACE);
```