Week 7

Computer Viewing is *moving the camera* to a desired position

There are three aspects of the viewing process, all of which are implemented in the pipeline:

1. **Positioning the camera**
    - Setting the model-view matrix, which defines:
        - Where the camera is
        - Where it is looking at
2. **Selecting an appropriate lens** *(Setting the projection matrix)*
3. **Clipping**
    - We *set the view volume*
    - We decide what is visible to the camera and what isn't
    - The camera cannot see forever, it is clipped at the far end, and at the sides, top, bottom

The OpenGL Camera

In OpenGL, initially the object and camera frames are the same

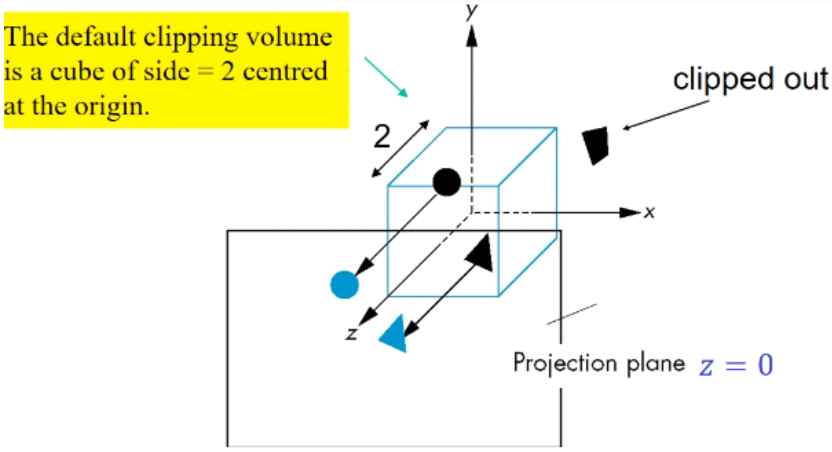
The <u>default model-view matrix</u> is an identity

The **camera** is *located at the origin* and *points in the negative z direction*

OpenGL also specifies a <u>default view volume</u> that is a **cube with sides of length 2** centered at the origin

- The cube goes from -1 to 1 in x, y and z dimensions
- The *default projection matrix* is an identity

The Default Projection

	The <u>default projection is orthogonal</u> - this means that *objects that are far will appear the same* as objects that are close This is unrealistic The projection plane is at z=0

Moving the Camera Frame

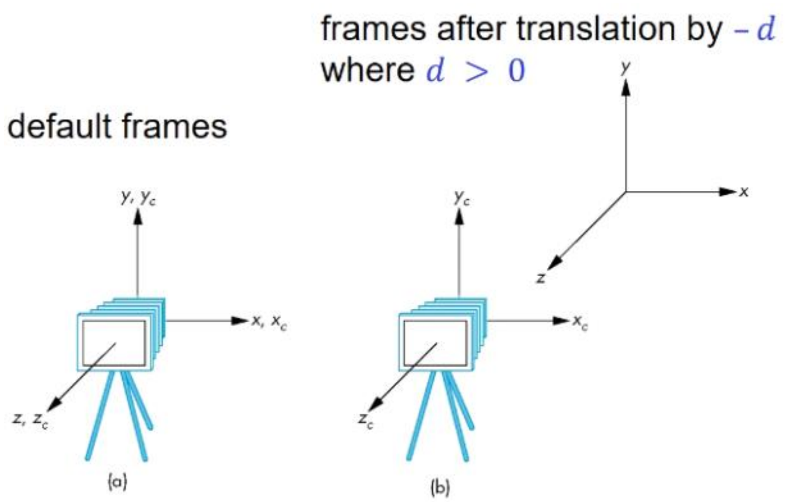
If <u>we want to visualize objects that have both positive and negative z values</u> we can either:

- *Move the camera* in the positive z direction (Translate the **camera frame**)
- **Move the objects** in the negative z direction (Translate the **world frame**)

Both of these views are equivalent

- Moving the camera (frame) **is the same as** moving the objects (world frame)
- Both are determined by the model-view matrix
  - We do a translation on Z axis = `Translate (0.0, 0.0, -d);`
  - We move the objects in the `-z` direction.

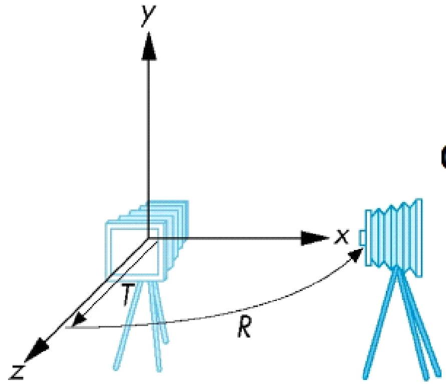
Moving Camera Back From Origin



Before and after translation

The camera is now moved back and may be able to see things that were behind it previously

Moving the Camera

 d	We can move the camera to any desired position by a sequence of rotations and translations **Example:** Side view at the +x axis looking the origin 1. Rotate the camera 2. Translate away from origin • Model-view matrix (C) ○ $C = TR$ ○ The rightmost matrix is the first applied

OpenGL Code

The rightmost matrix is the first applied

`// Using mat.h`
 	
`mat4 t = Translate (0.0, 0.0, -d);`
`mat4 ry = RotateY(90.0);`
`mat4 m = t*ry;`
	We rotate 90 degrees
	
	We apply rotation first (multiply last)
	
	m is the model view matrix

The LookAt() Function

The GLU library contains the function **gluLookAt()** which can be used to form the required model-view matrix.

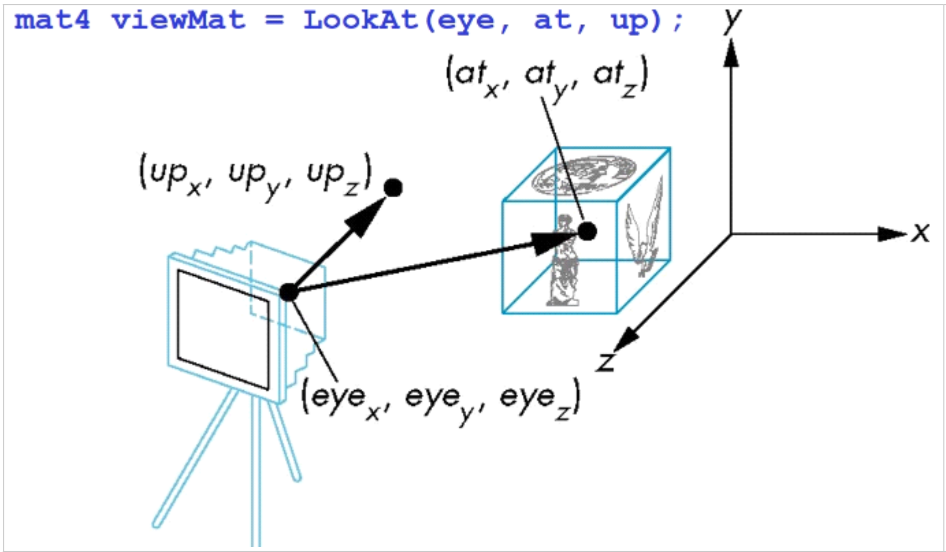
```
void gluLookAt(eyeX, eyeY, eyeZ, centreX, centreY, centreZ, upX, upY, upZ)
```

- We need to define the eye (camera) position, the center (fixation point) (where camera looks), and an up direction.
- All are of type GLdouble.

Alternatively, we can use **LookAt() defined in mat.h**

- The function returns a mat4 matrix.
- Can concatenate with modeling transformations
- Uses GLfloat
- Example:
  - `mat4 mv = LookAt(vec4 eye, vec4 at, vec4 up);`
  - Eye = xyz of eye + scaling number
  - Returns model view matrix

Diagram

	Example use of LookAt function

Default Orthographic Projection

The default projection in the eye (camera) frame is orthogonal

For a point $\mathbf{p} = (x, y, z, 1)^{\mathrm{T}}$ within the default view volume, it is projected to $\mathbf{p}_p = (xp, yp, zp, wp)^{\mathrm{T}}$, where $x_p = x, \; yp = y, \; zp = 0, \; wp = 1$ i.e., we can define $\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ and we can then write $\mathbf{P}_p = \mathbf{Mp}$	This is default model view matrix. X, Y, Z, W is along diagonal By default, everything is placed onto camera plane (z=0) with no scaling (w=1)

In practice, we can let $\boldsymbol{M} = \boldsymbol{I}$ and set $\boldsymbol{z}$ term to 0 later

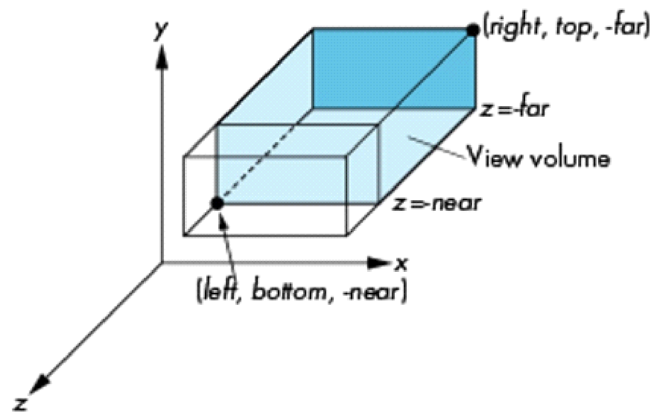
Orthogonal Viewing

The OpenGL orthogonal viewing function:

- `void glOrtho(left, right, bottom, top, near, far);`
- Uses GLdouble

Alternatively, we can use Ortho() defined in mat.h:

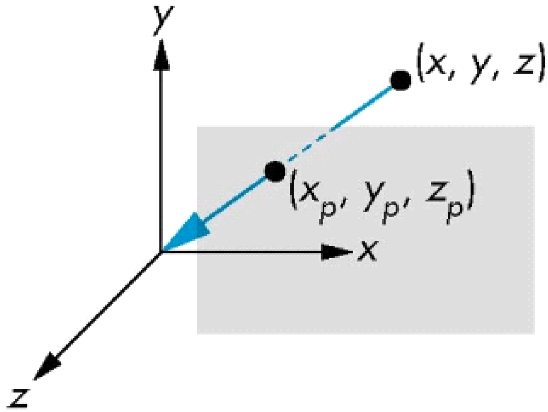
- `mat4 Ortho(left, right , bottom, top , near , far);`
- Returns matrix so can be combined with other matrices
- Uses GLfloat



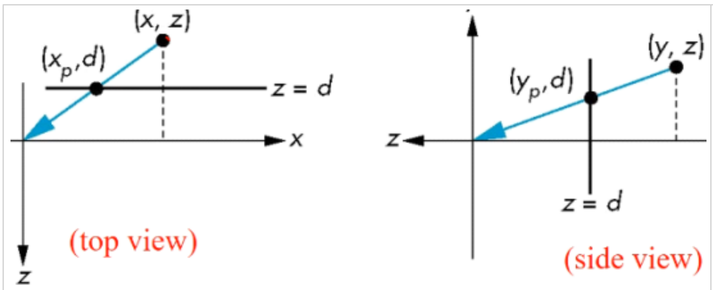
Parameters for Ortho functions are shown here

- Left, Right
- Bottom, Top
- Near, Far (measured from camera)

Simple Perspective

	In **orthographic projection**, the *camera's focal length is infinite* (the camera lens is at infinity). However, in perspective projection, the camera's **focal length d is finite** A simple perspective projection: • Center of projection is at the origin • Projection plane $z = d$, where d is negative

Top and Side Views



$$\frac{x_p}{d} = \frac{x}{z} \qquad \frac{y_p}{d} = \frac{y}{z}$$

$$x_p = \frac{x}{z/d} \qquad \text{i.e., } y_p = \frac{y}{z/d} \qquad z_p = d$$

Every point is drawn in a straight line towards the center of projection, and wherever it *crosses the imaging plane* is **the location of its projection**

Recall the OpenGL synthetic camera model in an earlier lecture

Matrix

$$\mathbf{M}p \quad = \quad q$$

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

q is the projected point
w is the perspective division

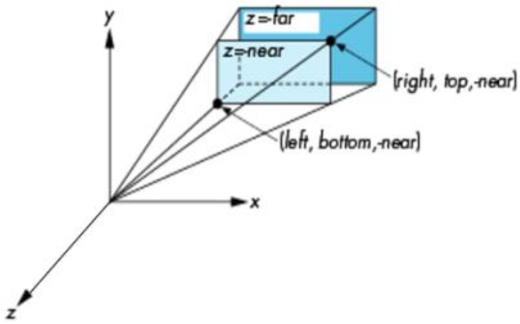
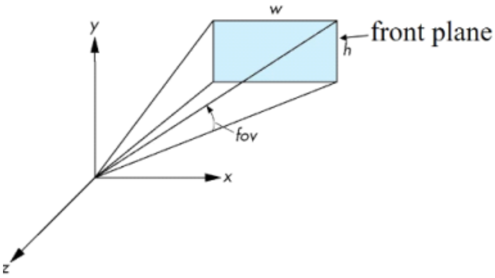
However, since $\boldsymbol{w} = \boldsymbol{z/d} \neq \mathbf{1}$, so we must divide by w to return back to inhomogeneous coordinates.

The **perspective division** yields the desired perspective equations:

$$x_p = \frac{x}{z/d} \quad y_d = \frac{y}{z/d} \quad z_p = d$$

Perspective Viewing

To define a perspective transformation matrix for the camera, we can use:

	**The Frustum function** in *mat.h* • `mat4 Frustum(left, right, bottom, top, near, far)` • Near is negative • Uses GLfloat • Possibly difficult to get the desired view. • Can be combined with other matrices
	**The Perspective function** • `mat4 Perspective(fovy, aspect, near, far)` • Uses GLfloat • Better interface • `Aspect` = w/ h • `Fovy` is an angle in degrees • `Near,far` = distance from viewer to near/far clipping plane • Has equivalent `gluPerspective(...)`