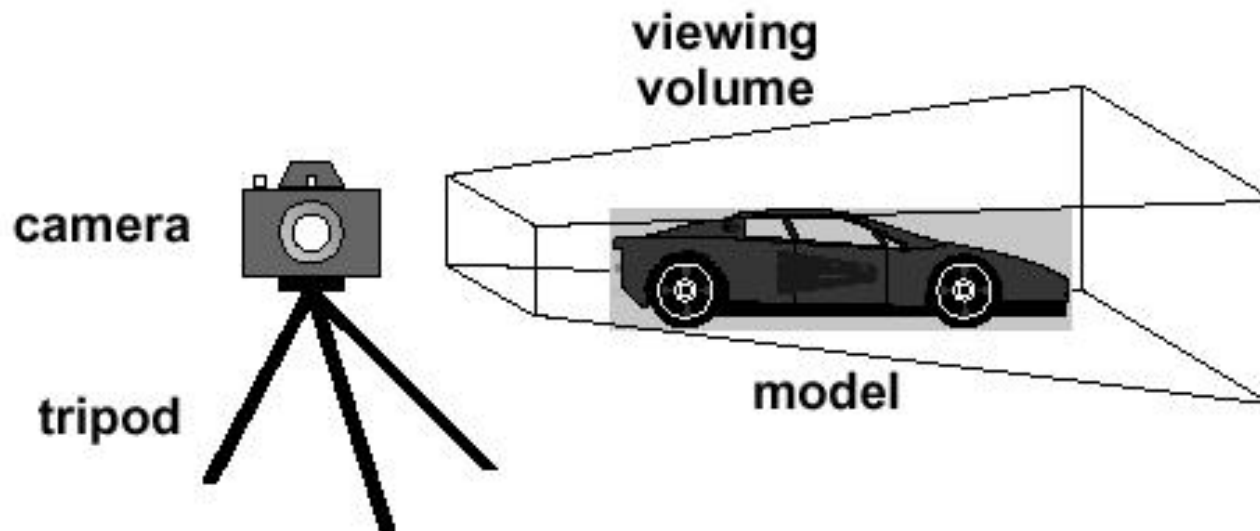


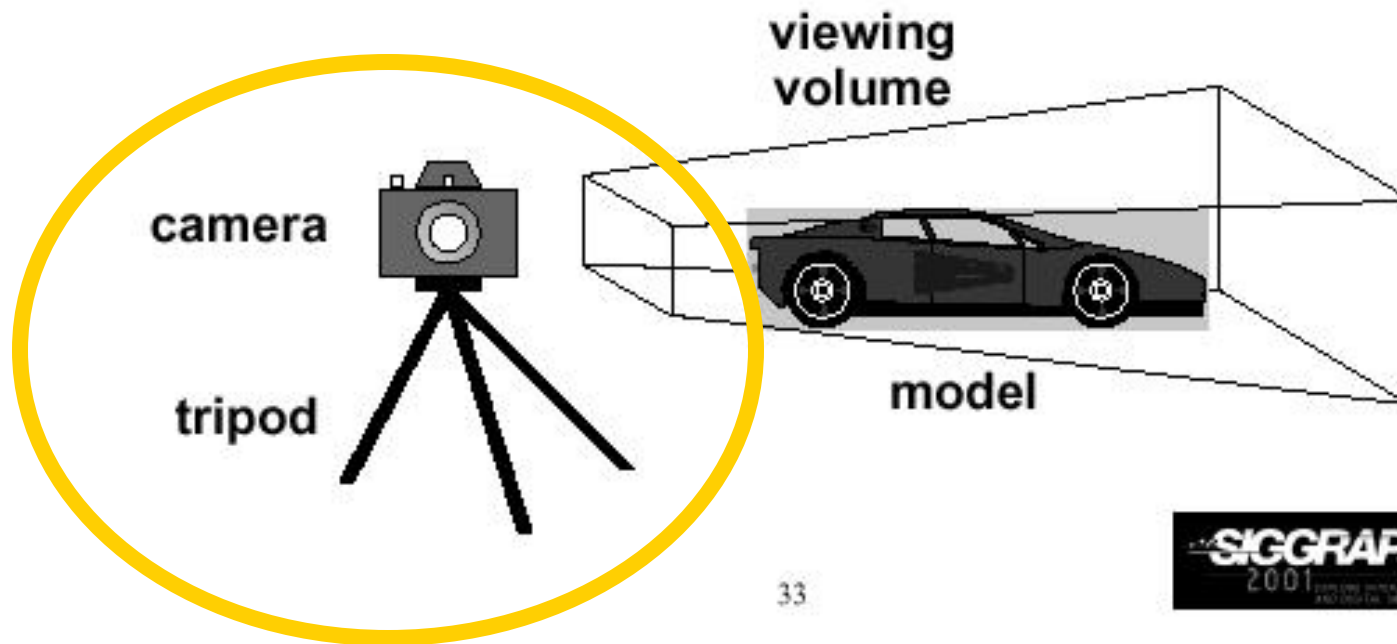
Introduction to 3D viewing

- 3D is just like taking a photograph!



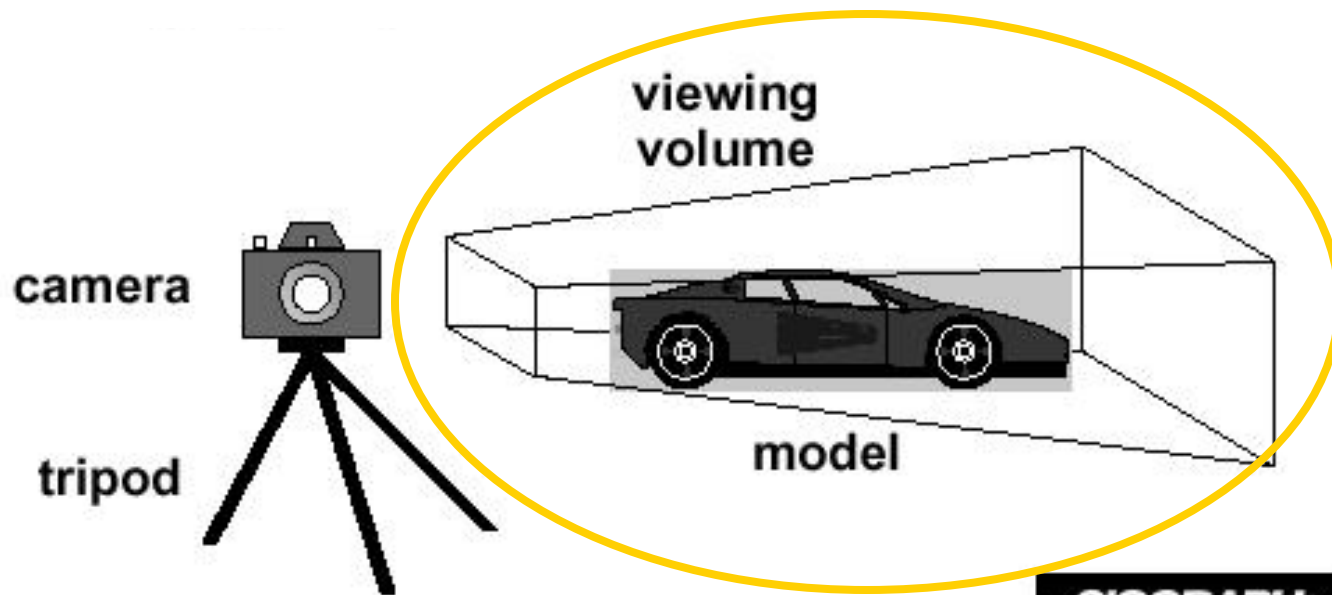
Viewing Transformation

- Position and orient your camera



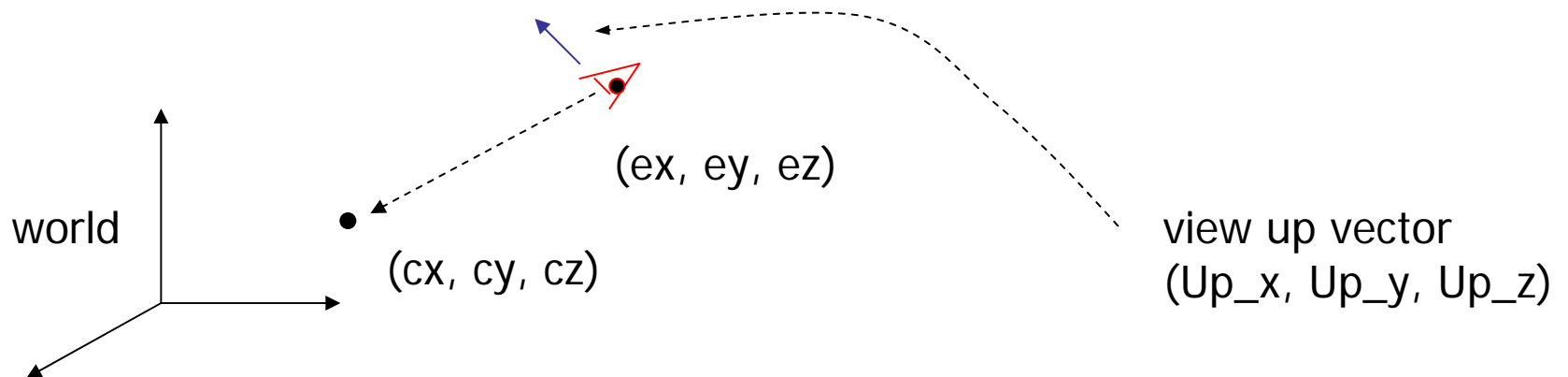
Projection Transformation

- Control the “lens” of the camera
- Project the object from 3D world to 2D screen



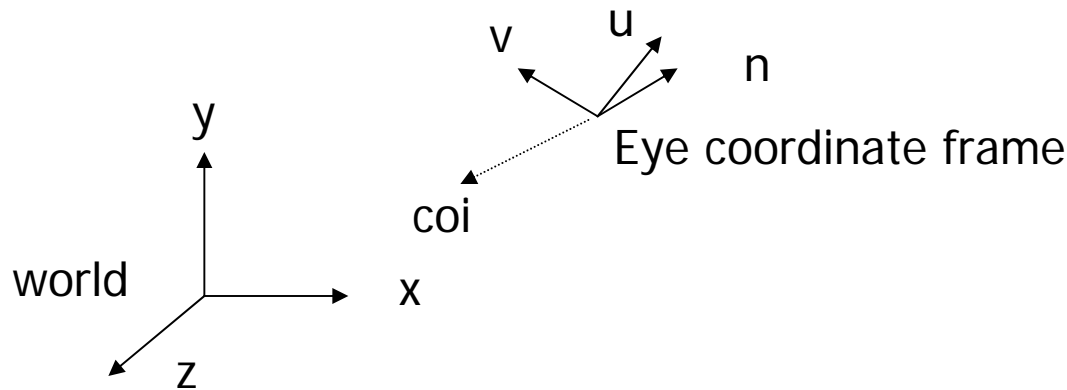
Viewing Transformation (2)

- Important camera parameters to specify
 - Camera (eye) position (E_x, E_y, E_z) in world coordinate system
 - Center of interest (coi) (c_x, c_y, c_z)
 - Orientation (which way is up?) View-up vector (Up_x, Up_y, Up_z)



Viewing Transformation (3)

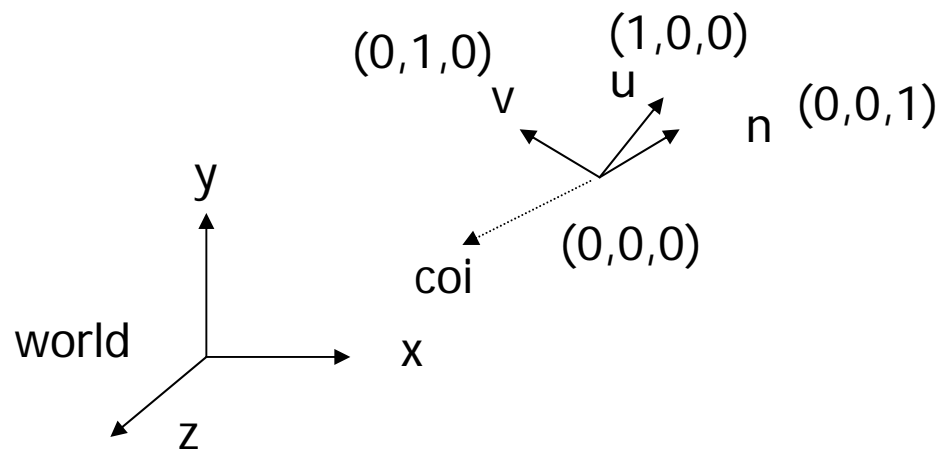
- Transformation?
 - Form a camera (eye) coordinate frame



- Transform objects from world to eye space

Viewing Transformation (4)

- Eye space?



- Transform to eye space can simplify many downstream operations (such as projection) in the pipeline



Viewing Transformation (5)

- In OpenGL:



- gluLookAt (Ex, Ey, Ez, cx, cy, cz,
Up_x, Up_y, Up_z)
- The view up vector is usually (0,1,0)
- Remember to set the OpenGL matrix mode to **GL_MODELVIEW** first



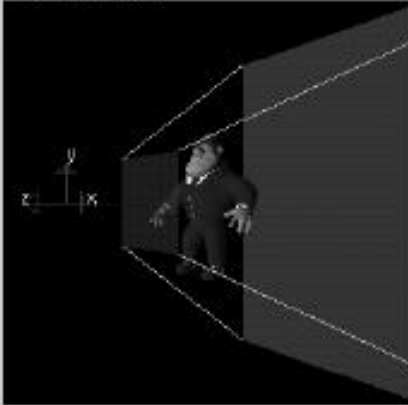
Viewing Transformation (6)

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);
    display_all(); // your display routine
}
```


Demo

Projection

World-space view



Screen-space view



Command manipulation window

```
fovy aspect zNear zFar  
gluPerspective( 60.0 , 1.00 , 1.0 , 10.0 );  
gluLookAt( 0.00 , 0.00 , 2.00 , ← eye  
          0.00 , 0.00 , 0.00 , ← center  
          0.00 , 1.00 , 0.00 ); ← up
```

Click on the arguments and move the mouse to modify values.

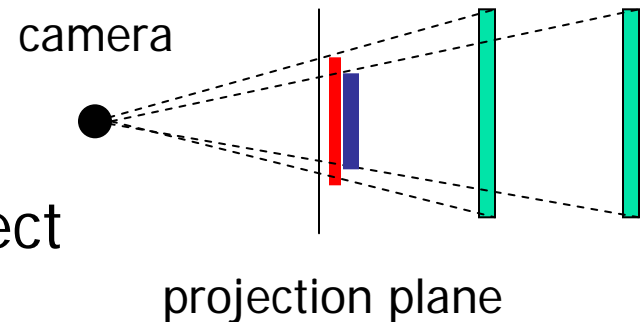


Projection Transformation

- Important things to control
 - Perspective or Orthographic
 - Field of view and image aspect ratio
 - Near and far clipping planes

Perspective Projection

- Characterized by **object foreshortening**
 - Objects appear to be larger if they are closer to the camera
 - This is what happens in the real world
- Need:
 - Projection center
 - Projection plane
- Projection: Connecting the object to the projection center





Orthographic Projection

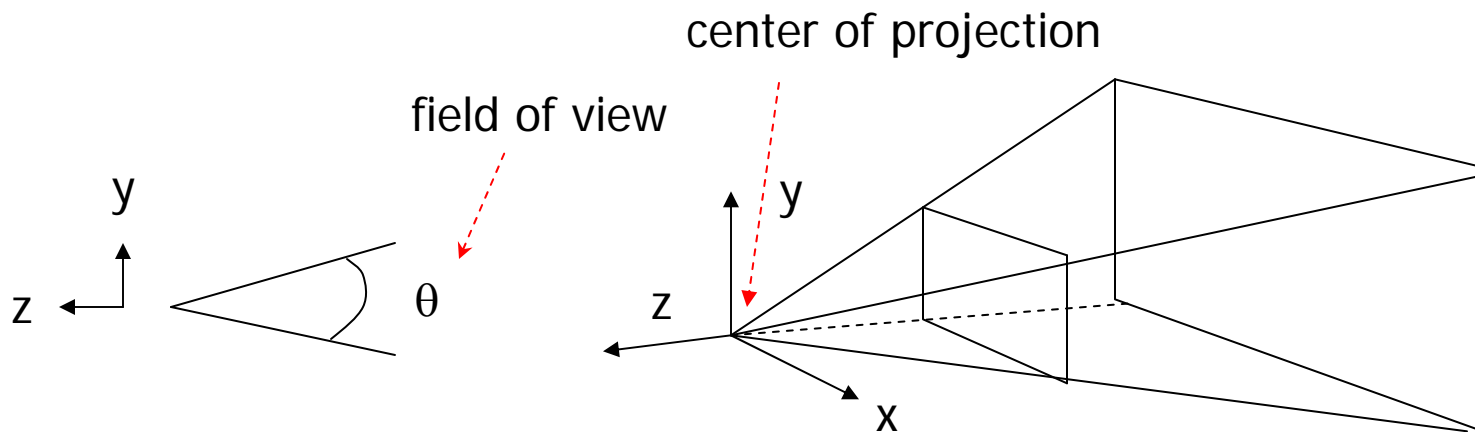
- No foreshortening effect – distance from camera does not matter
- The projection center is at infinite



- Projection calculation – just drop z coordinates

Field of View

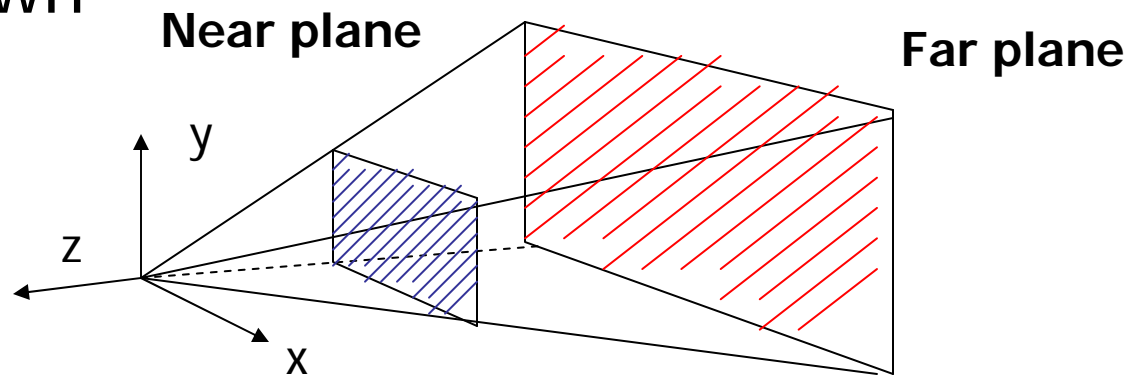
- Determine how much of the world is taken into the picture



- The larger is the field view, the smaller is the object projection size

Near and Far Clipping Planes

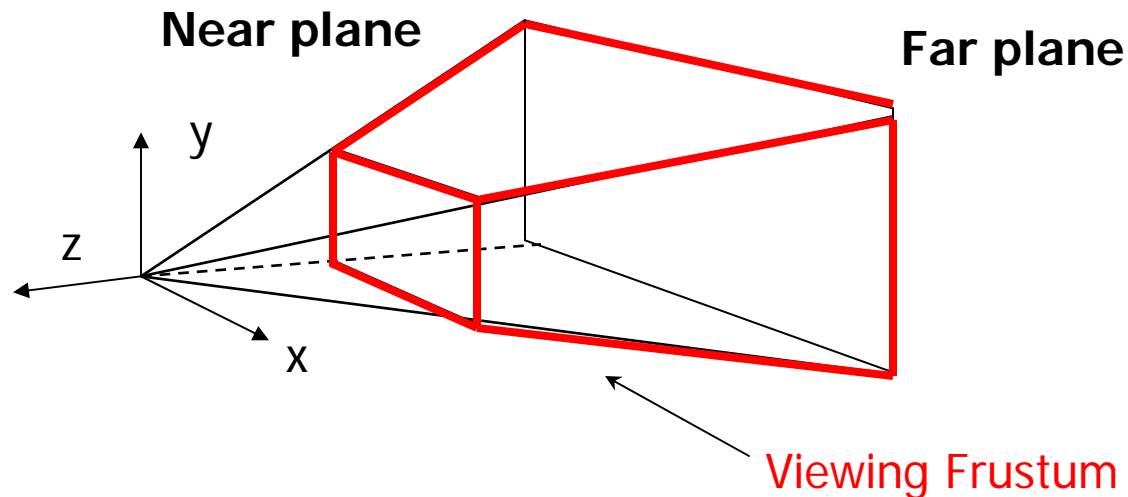
- Only objects between near and far planes are drawn



- Near plane + far plane + field of view =
Viewing Frustum

Viewing Frustum

- 3D counterpart of 2D world clip window



- Objects outside the frustum are clipped



Projection Transformation

- In OpenGL:

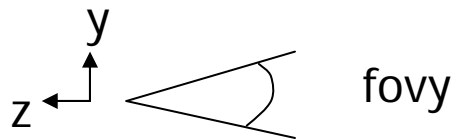


- Set the matrix mode to `GL_PROJECTION`
- Perspective projection: use
 - `gluPerspective(fovy, aspect, near, far)` **or**
 - `glFrustum(left, right, bottom, top, near, far)`
- Orthographic:
 - `glOrtho(left, right, bottom, top, near, far)`

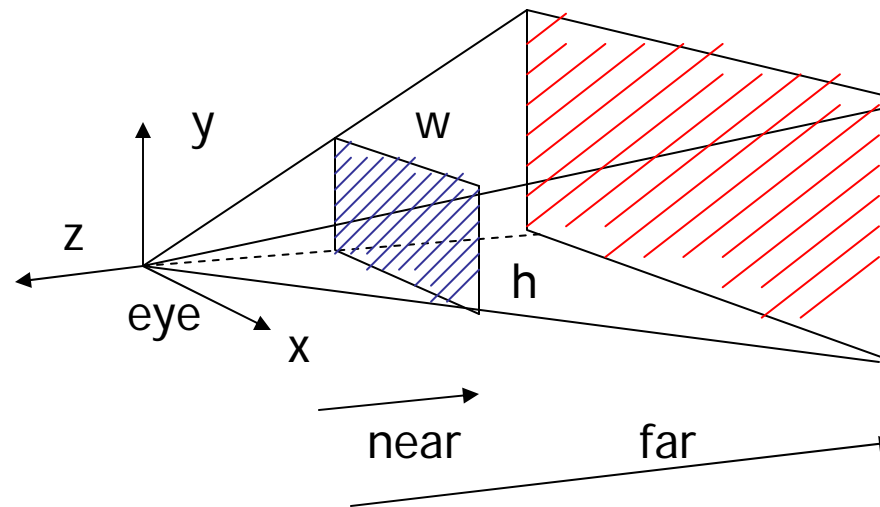


`gluPerspective(fovy, aspect, near, far)`

- Aspect ratio is used to calculate the window width



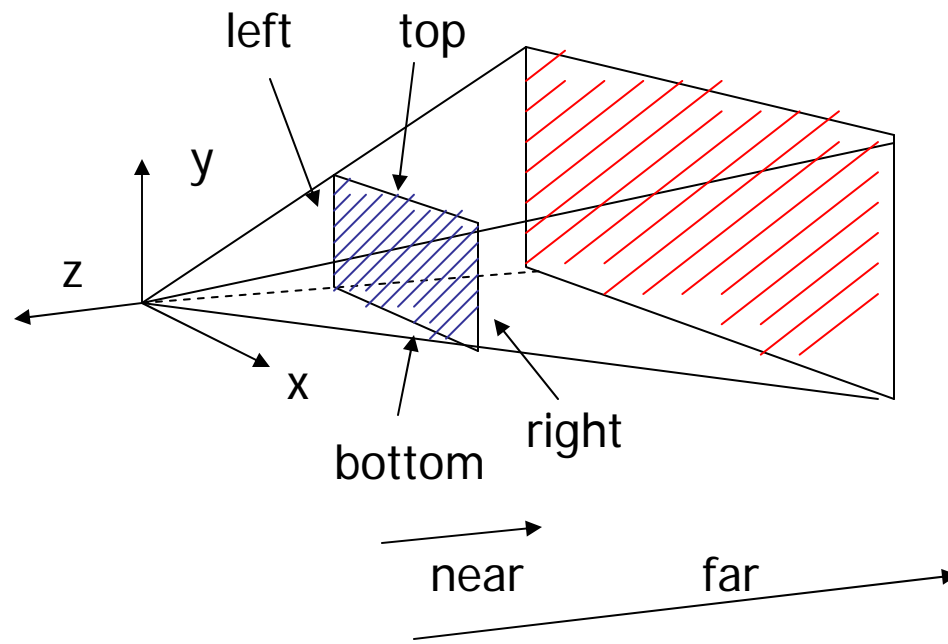
$$\text{Aspect} = w / h$$





`glFrustum(left, right, bottom, top, near, far)`

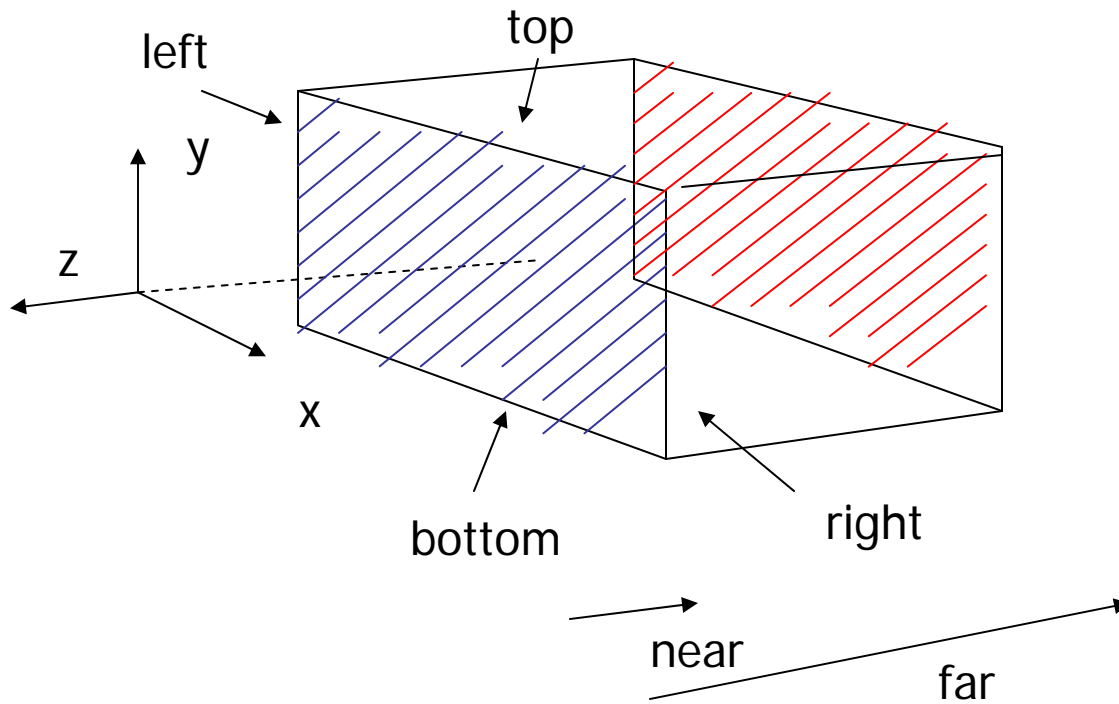
- Or You can use this function in place of `gluPerspective()`





`glOrtho(left, right, bottom, top, near, far)`

- For orthographic projection





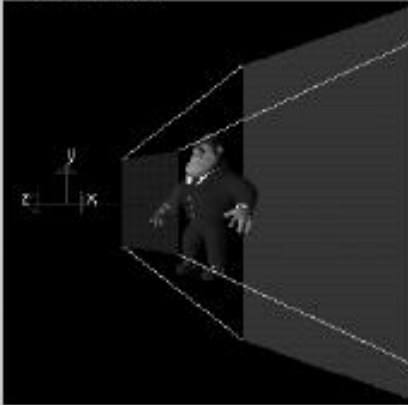
Projection Transformation

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(fove, aspect, near, far);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);
    display_all();    // your display routine
}
```


Demo

Projection

World-space view



Screen-space view

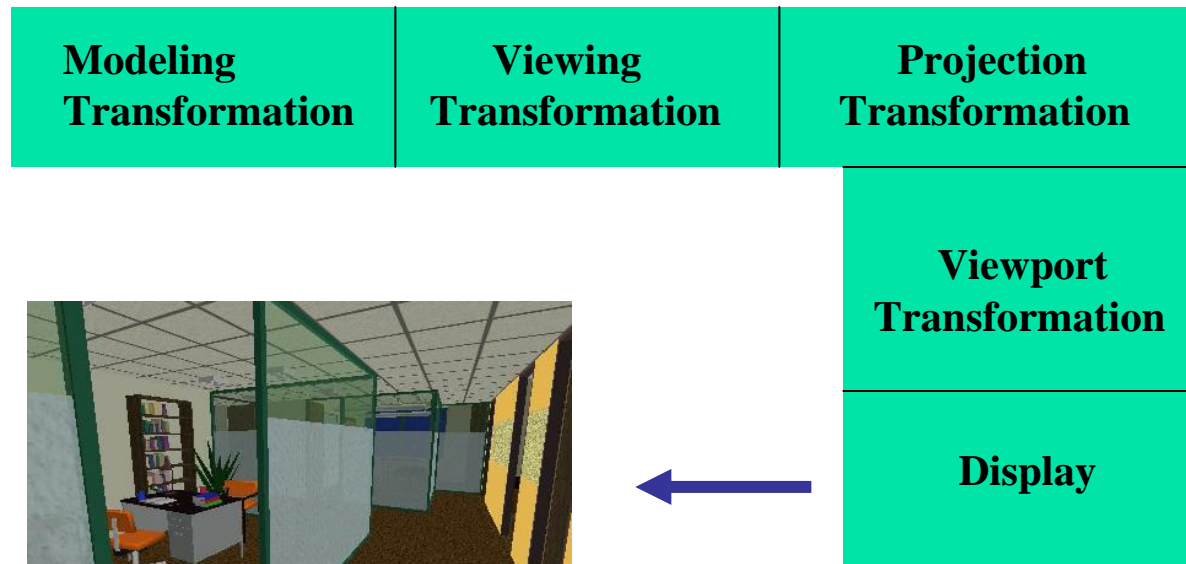
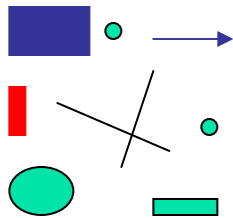


Command manipulation window

```
fovy aspect zNear zFar  
gluPerspective( 60.0 , 1.00 , 1.0 , 10.0 );  
gluLookAt( 0.00 , 0.00 , 2.00 , ← eye  
          0.00 , 0.00 , 0.00 , ← center  
          0.00 , 1.00 , 0.00 ); ← up
```

Click on the arguments and move the mouse to modify values.

3D viewing under the hood





3D viewing under the hood

Topics of Interest:

- Viewing transformation
- Projection transformation

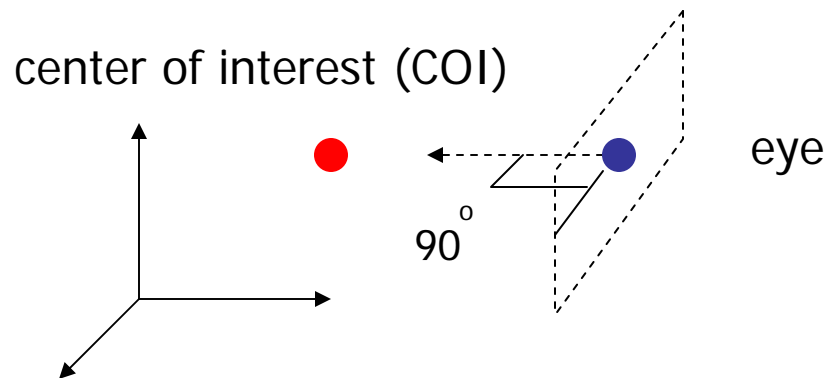


Viewing Transformation

- Transform the object from world to eye space
 - Construct an eye space coordinate frame
 - Construct a matrix to perform the coordinate transformation
 - Flexible Camera Control

Eye Coordinate Frame

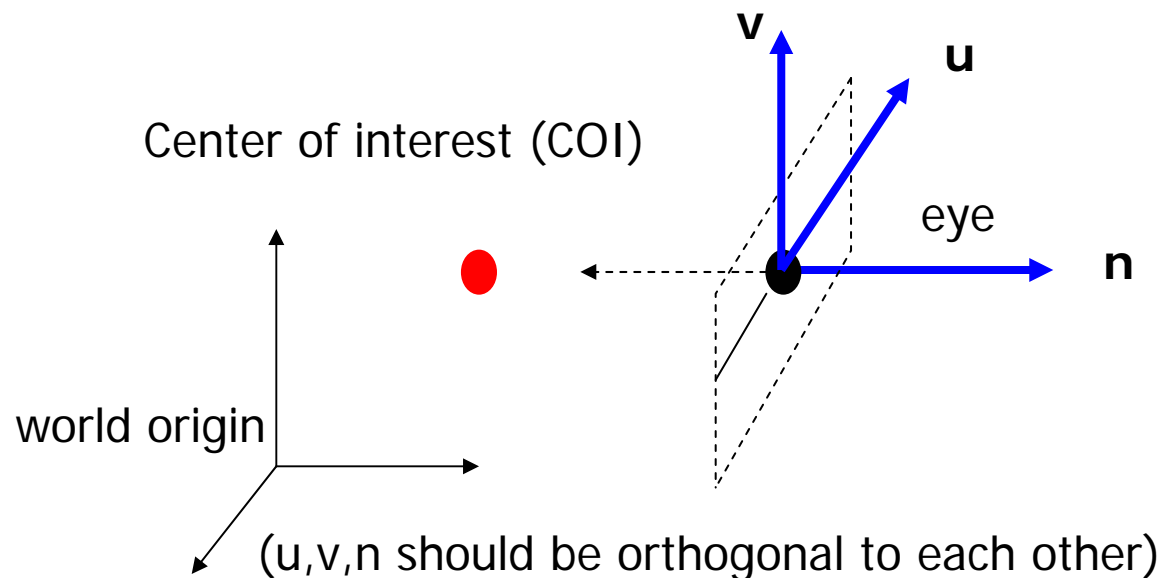
- Known: eye position, center of interest, view-up vector
- To find out: new origin and three basis vectors



Assumption: the direction of view is orthogonal to the view plane (the plane that objects will be projected onto)

Eye Coordinate Frame (2)

- Origin: **eye position** (that was easy)
- Three basis vectors: one is the normal vector (**n**) of the viewing plane, the other two are the ones (**u** and **v**) that span the viewing plane



n is pointing away from the world because we use right hand coordinate system

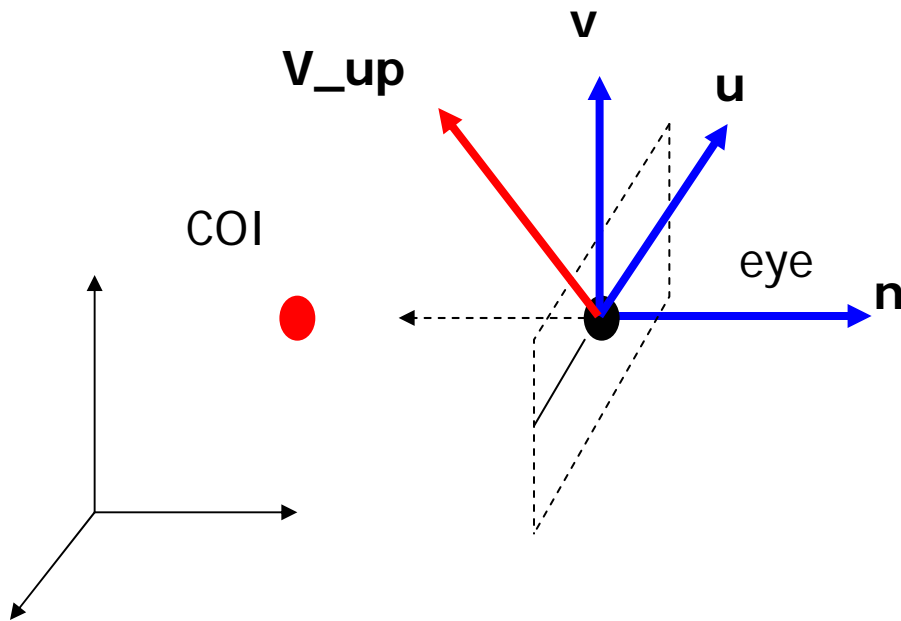
$$\mathbf{N} = \text{eye} - \text{COI}$$

$$\mathbf{n} = \mathbf{N} / |\mathbf{N}|$$

Remember **u,v,n** should be all unit vectors

Eye Coordinate Frame (3)

- How about u and v?



We can get u first -

u is a vector that is perpendicular to the plane spanned by N and view up vector (V_{up})

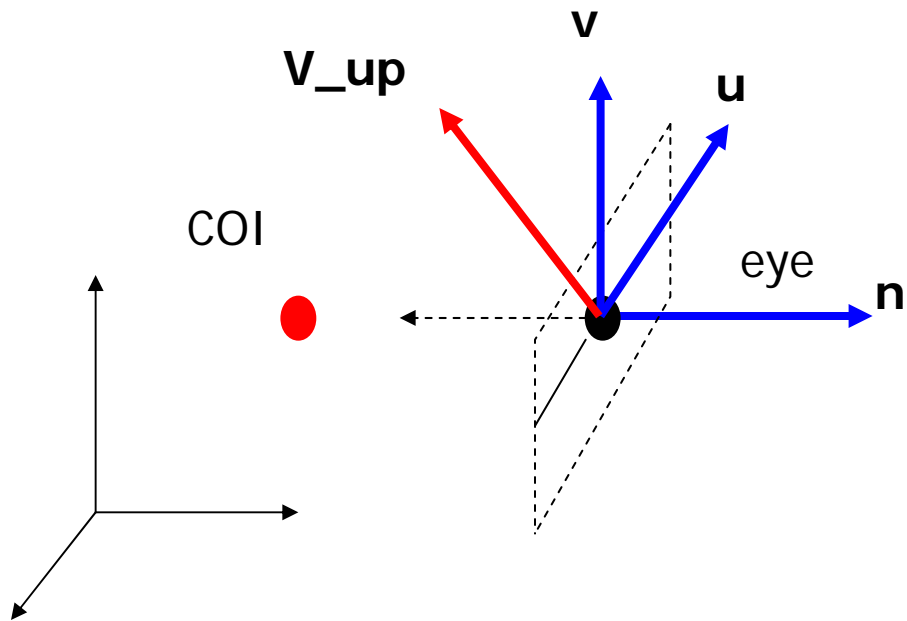
$$U = V_{up} \times n$$

$$u = U / |U|$$

Eye Coordinate Frame (4)

- How about v ?

Knowing n and u , getting v is easy



$$v = n \times u$$

v is already normalized

Eye Coordinate Frame (5)

- Put it all together

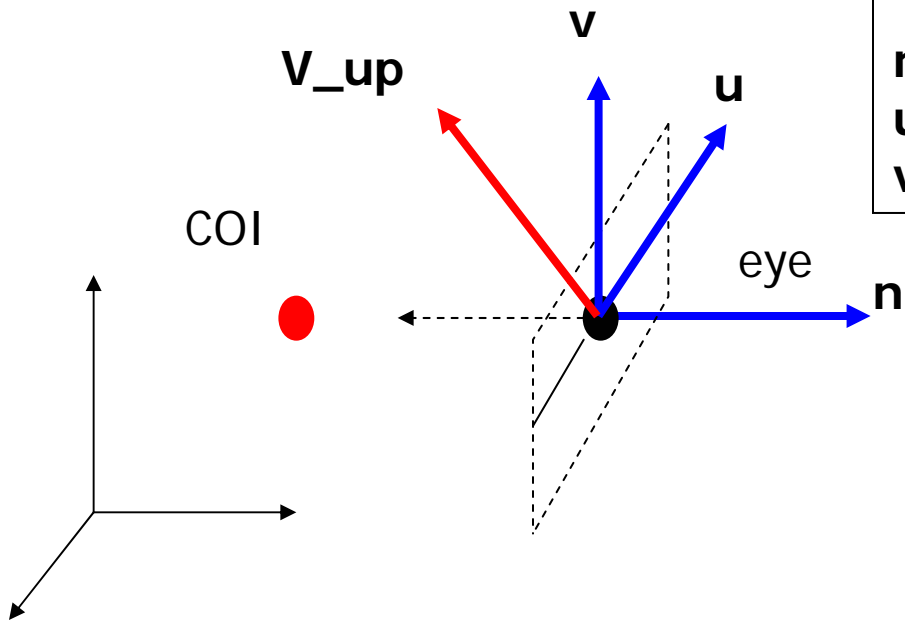
Eye space **origin**: $(\text{Eye.x}, \text{Eye.y}, \text{Eye.z})$

Basis vectors:

$$\mathbf{n} = (\text{eye} - \text{COI}) / |\text{eye} - \text{COI}|$$

$$\mathbf{u} = (\mathbf{V_up} \times \mathbf{n}) / |\mathbf{V_up} \times \mathbf{n}|$$

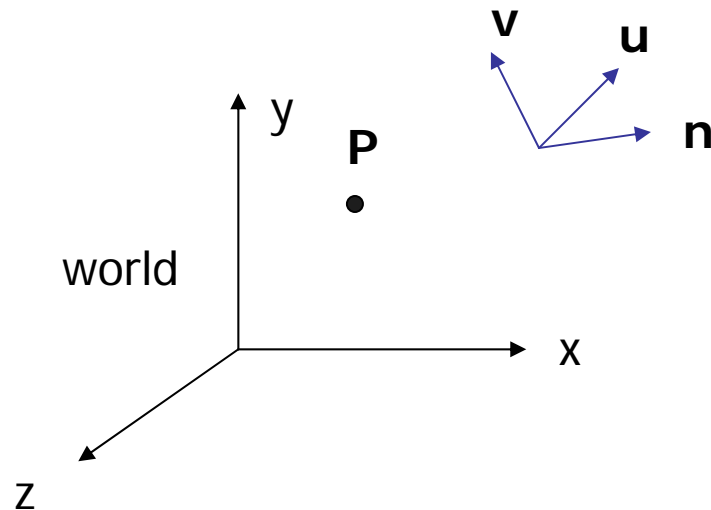
$$\mathbf{v} = \mathbf{n} \times \mathbf{u}$$



World to Eye Transformation

- Transformation matrix (M_{w2e}) ?

$$P' = M_{w2e} \times P$$

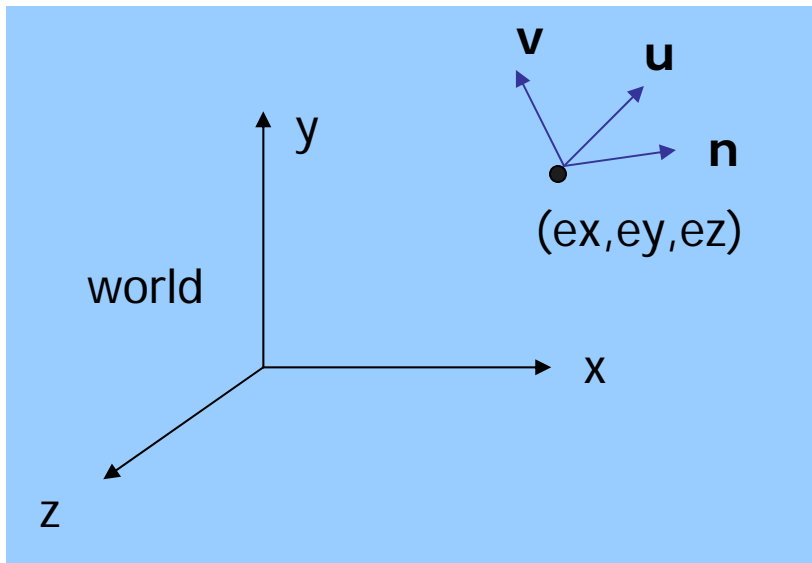


1. Come up with the transformation sequence to move eye coordinate frame to the world

2. And then apply this sequence to the point P in a reverse order

World to Eye Transformation

- Rotate the eye frame so that it will be “aligned” with the world frame
- Translate $(-ex, -ey, -ez)$



Rotation:

$$\begin{vmatrix} ux & uy & uz & 0 \\ vx & vy & vz & 0 \\ nx & ny & nz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

How to verify the rotation matrix?

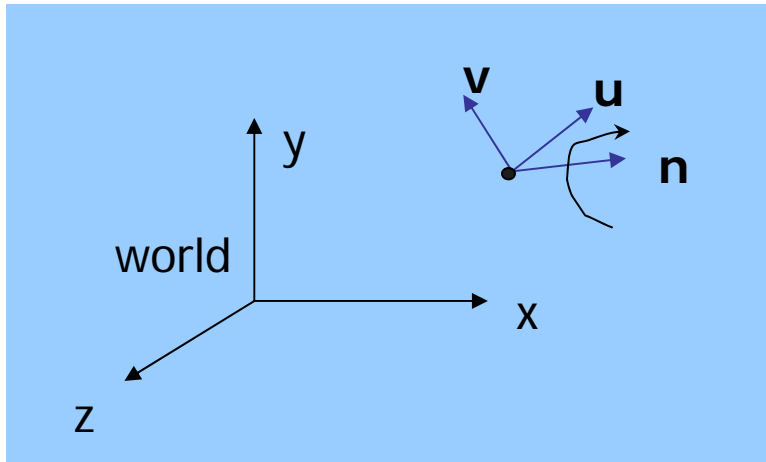
Translation:

$$\begin{vmatrix} 1 & 0 & 0 & -ex \\ 0 & 1 & 0 & -ey \\ 0 & 0 & 1 & -ez \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

World to Eye Transformation (3)

- Head tilt: Rotate your head by δ
- Just rotate the object about the eye space z axis - δ

$$M_{w2e} = \begin{vmatrix} \cos(-\delta) & -\sin(-\delta) & 0 & 0 \\ \sin(-\delta) & \cos(-\delta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



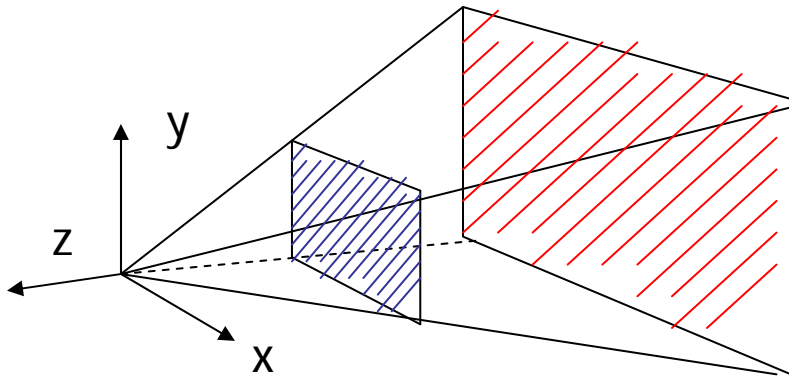
Why $-\delta$?

When you rotate your head by δ , it is like rotate the object by $-\delta$

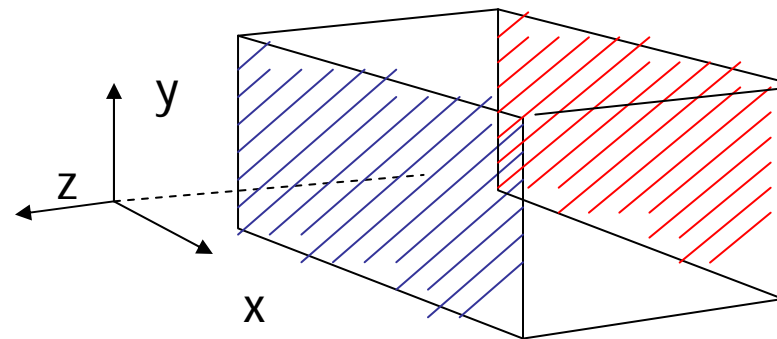


Projection Transformation

- Projection – map the object from 3D space to 2D screen



Perspective: `gluPerspective()`



Parallel: `glOrtho()`

Parallel Projection

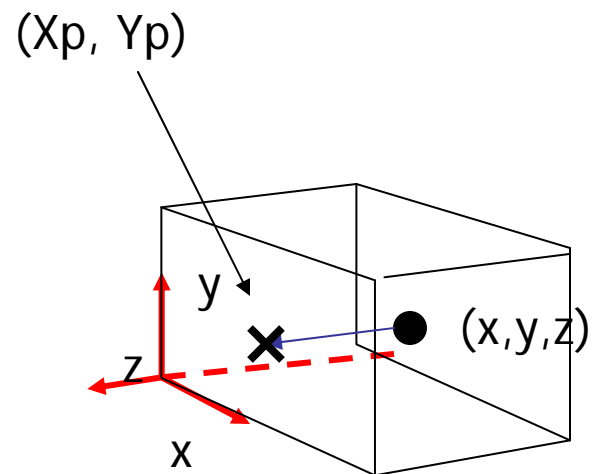
- After transforming the object to the eye space, parallel projection is relative easy – we could just drop the Z

$$X_p = x$$

$$Y_p = y$$

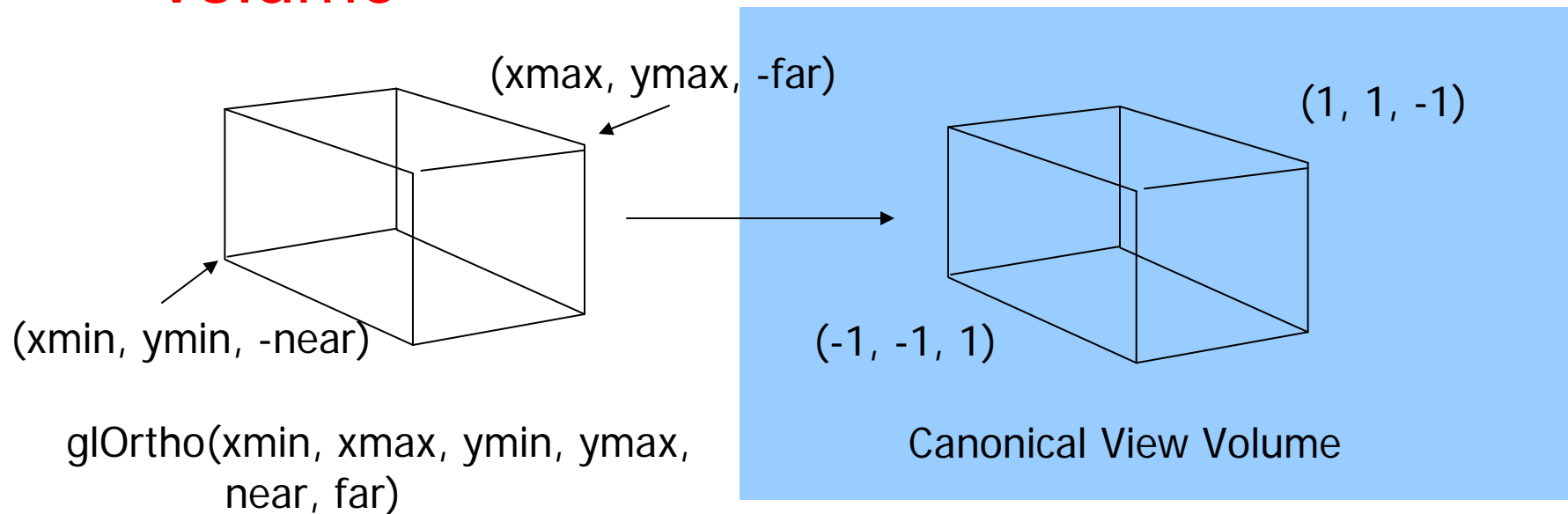
$$Z_p = -d$$

- We actually want to keep Z
– why?



Parallel Projection (2)

- OpenGL maps (projects) everything in the visible volume into a **canonical view volume**





Parallel Projection (3)

- Transformation sequence:

1. Translation (M1): (-near = zmax, -far = zmin)

$$-(x_{\max}+x_{\min})/2, -(y_{\max}+y_{\min})/2, -(z_{\max}+z_{\min})/2$$

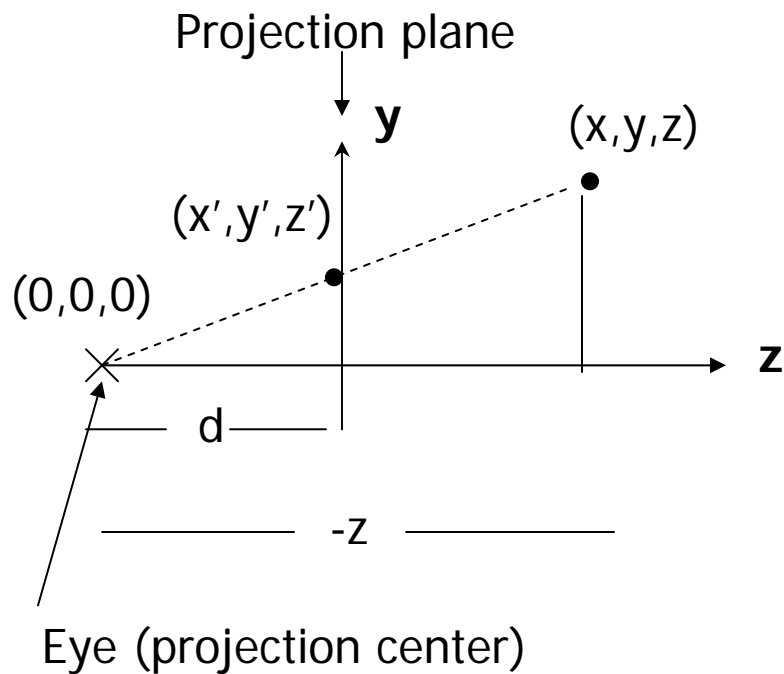
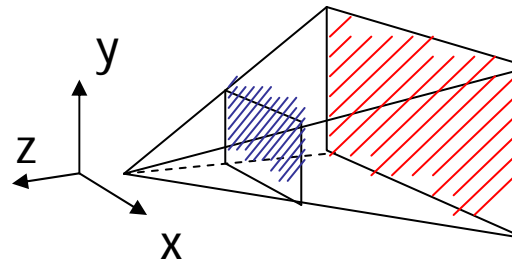
2. Scaling (M2):

$$2/(x_{\max}-x_{\min}), 2/(y_{\max}-y_{\min}), 2/(z_{\max}-z_{\min})$$

$$M2 \times M1 = \begin{vmatrix} 2/(x_{\max}-x_{\min}) & 0 & 0 & -(x_{\max}+x_{\min})/(x_{\max}-x_{\min}) \\ 0 & 2/(y_{\max}-y_{\min}) & 0 & -(y_{\max}+y_{\min})/(y_{\max}-y_{\min}) \\ 0 & 0 & 2/(z_{\max}-z_{\min}) & -(z_{\max}+z_{\min})/(z_{\max}-z_{\min}) \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Perspective Projection

- Side view:



Based on similar triangle:

$$\frac{y}{y'} = \frac{-z}{d}$$

➔ $Y' = y \times \frac{d}{-z}$



Perspective Projection (2)

- Same for x . So we have:

$$x' = x \times d / -z$$

$$y' = y \times d / -z$$

$$z' = -d$$

- Put in a matrix form:

$$\begin{array}{l} x' \\ y' \\ z' \\ w \end{array} = \begin{array}{c} \left| \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & (1/-d) & 0 \end{array} \right| \left| \begin{array}{c} x \\ y \\ z \\ 1 \end{array} \right| \end{array}$$

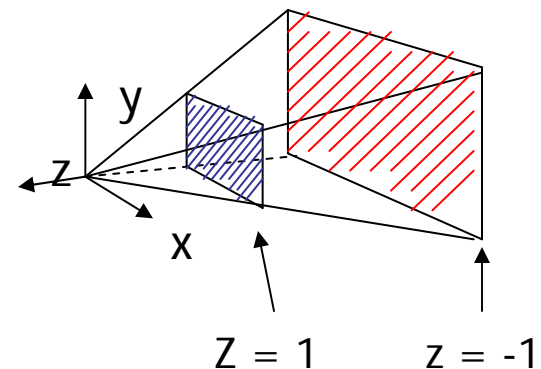
- OpenGL assume $d = 1$, i.e. the image plane is at $z = -1$

Perspective Projection (3)

- We are not done yet. We want to somewhat keep the z information so that we can perform depth comparison
- Use pseudo depth – OpenGL maps the near plane to 1, and far plane to -1
- Need to modify the projection matrix: solve a and b

$$\begin{array}{c|ccc|c} x' & 1 & 0 & 0 & 0 & x \\ y' & 0 & 1 & 0 & 0 & y \\ z' & 0 & 0 & a & b & z \\ w & 0 & 0 & (1/-d) & 0 & 1 \end{array}$$

How to solve a and b?



Perspective Projection (4)

- Solve a and b

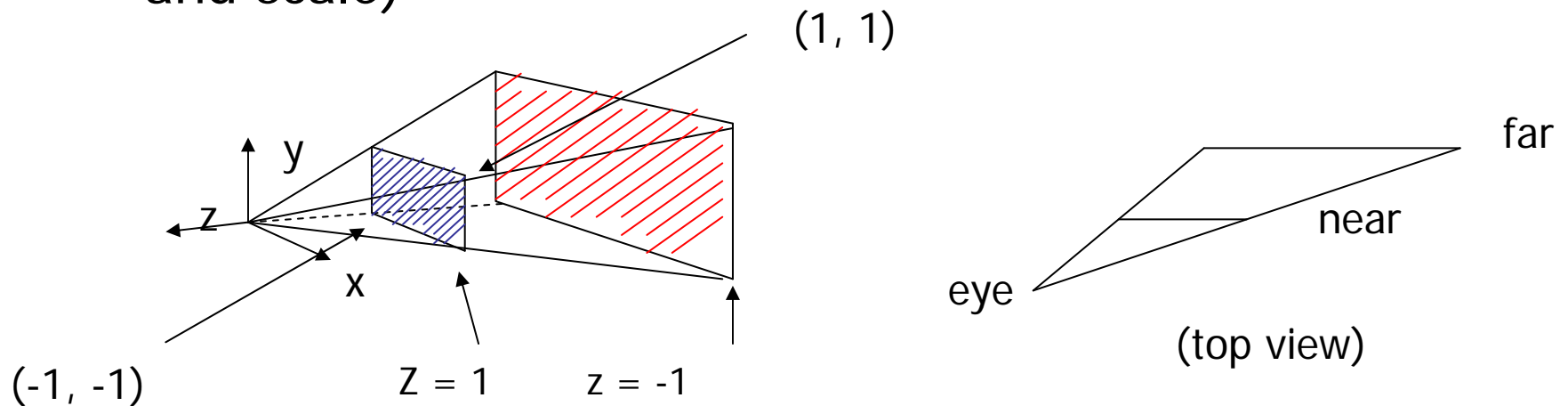
$$\begin{array}{c|c} \begin{array}{c} x' \\ y' \\ z' \\ w \end{array} & = & \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & (1/-d) & 0 \end{array} & \begin{array}{c|c} \begin{array}{c} x \\ y \\ z \\ 1 \end{array} & \end{array} \end{array}$$

- $(0,0,1)^T = M \times (0,0,-near)^T$
 $(0,0,-1)^T = M \times (0,0,-far)^T$
- M

- $a = -(far+near)/(far-near)$
 $b = (-2 \times far \times near) / (far-near)$
- Verify this!

Perspective Projection (5)

- Not done yet. OpenGL also normalizes the x and y ranges of the viewing frustum to $[-1, 1]$ (translate and scale)



- And takes care the case that eye is not at the center of the view volume (shear)



Perspective Projection (6)

- Final Projection Matrix:

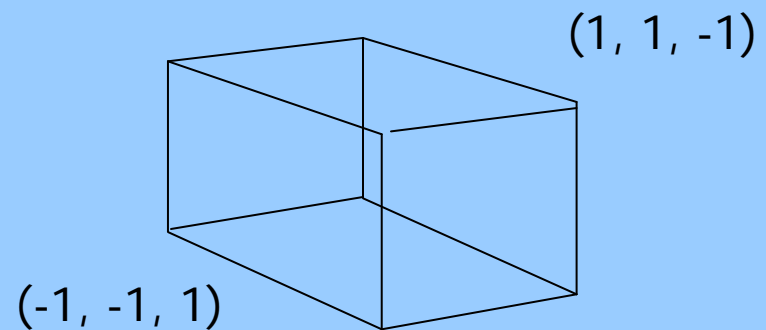
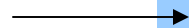
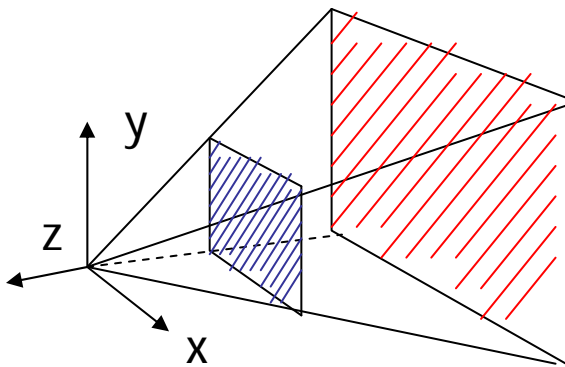
$$\begin{array}{l} x' \\ y' \\ z' \\ w' \end{array} = \begin{array}{c} \left| \begin{array}{cccc|c} 2N/(x_{\max}-x_{\min}) & 0 & (x_{\max}+x_{\min})/(x_{\max}-x_{\min}) & 0 & x \\ 0 & 2N/(y_{\max}-y_{\min}) & (y_{\max}+y_{\min})/(y_{\max}-y_{\min}) & 0 & y \\ 0 & 0 & -(F+N)/(F-N) & -2F*N/(F-N) & z \\ 0 & 0 & -1 & 0 & 1 \end{array} \right. \end{array}$$



`glFrustum(xmin, xmax, ymin, ymax, N, F)` N = near plane, F = far plane

Perspective Projection (7)

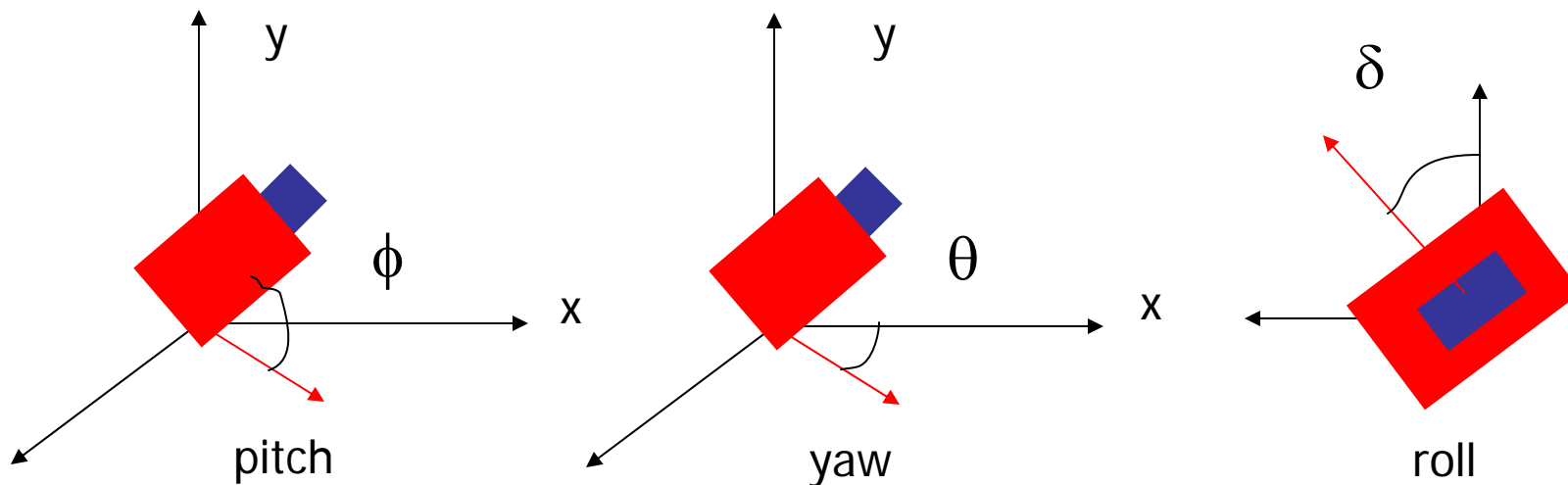
- After perspective projection, the viewing frustum is also projected into a canonical view volume (like in parallel projection)



Canonical View Volume

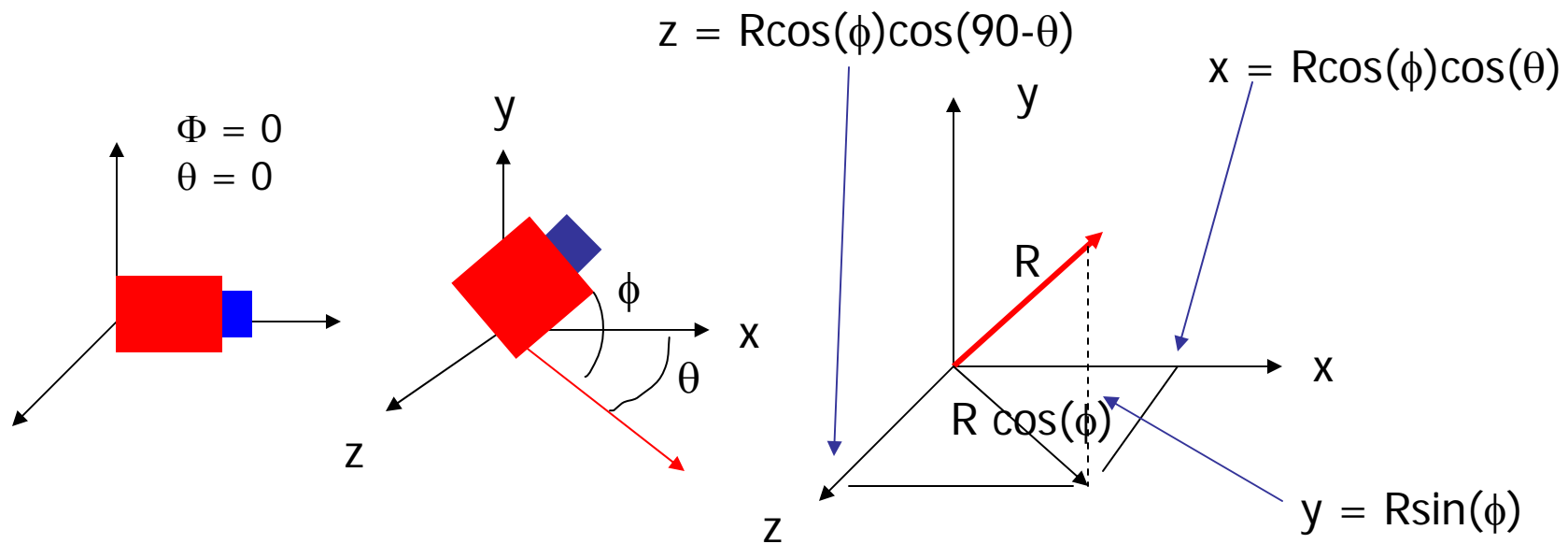
Flexible Camera Control

- Instead of provide COI, it is possible to just give camera orientation
- Just like control a airplane's orientation



Flexible Camera Control

- How to compute the viewing vector (x,y,z) from pitch(ϕ) and yaw(θ) ?





Flexible Camera Control

- `gluLookAt()` does not let you to control pitch and yaw
- you need to compute/maintain the vector by yourself
- And then calculate $COI = Eye + (x,y,z)$ before you can call `gluLookAt()`.