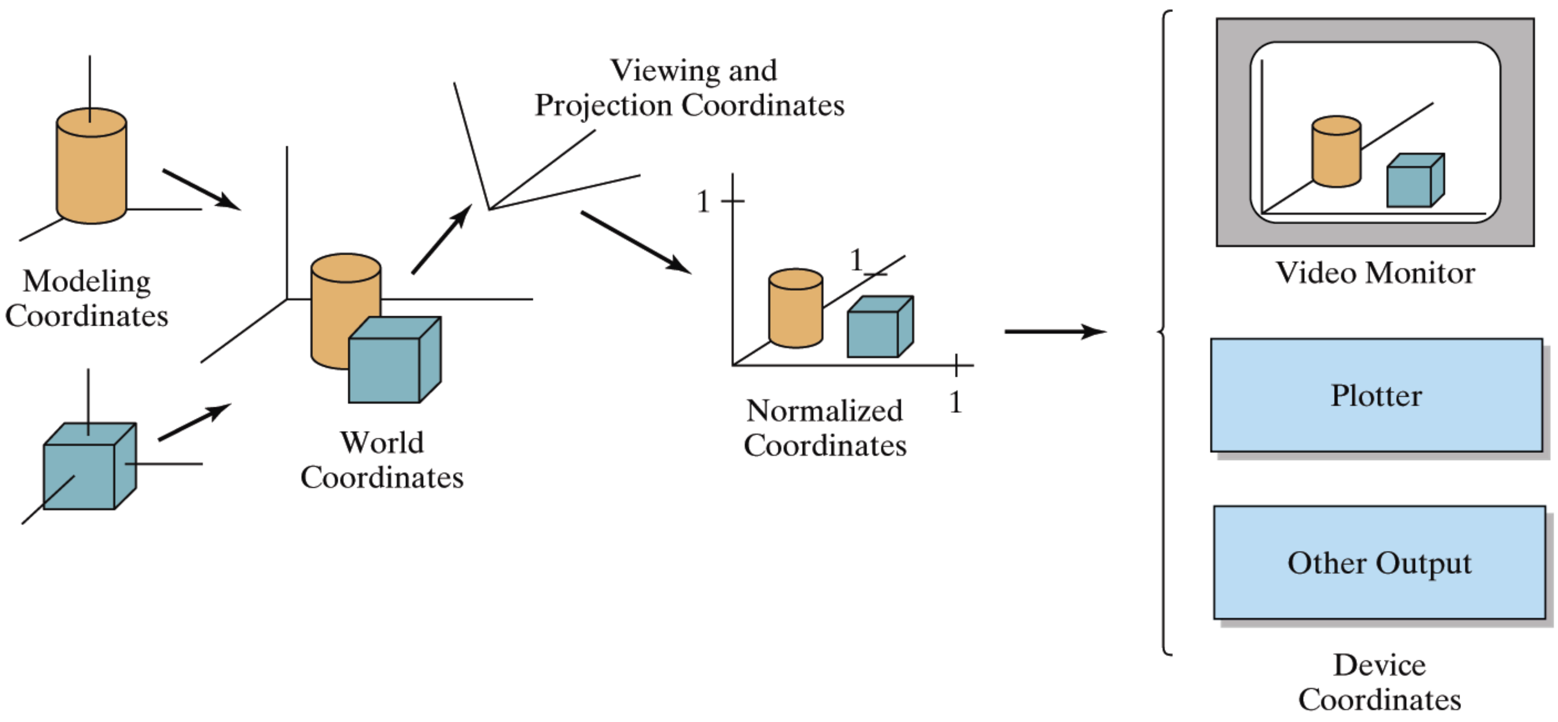
Model View Matrix and its implementation

Sang Il Park

OpenGL Geometric Transformations (old style)

- **`glMatrixMode(GL_MODELVIEW);`**



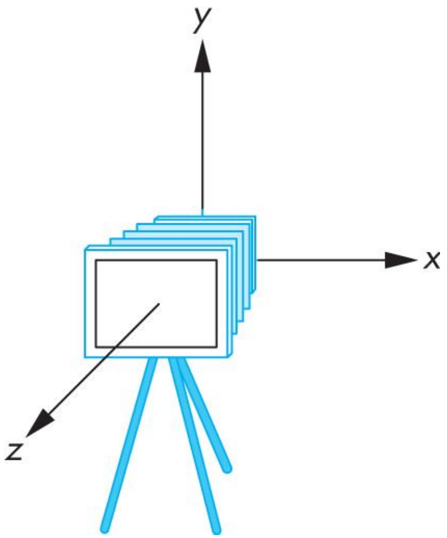
Topics of Model View Matrix

- Local to World Coordinate Transform
- Camera Positioning

Model View Matrix II: Camera Positioning

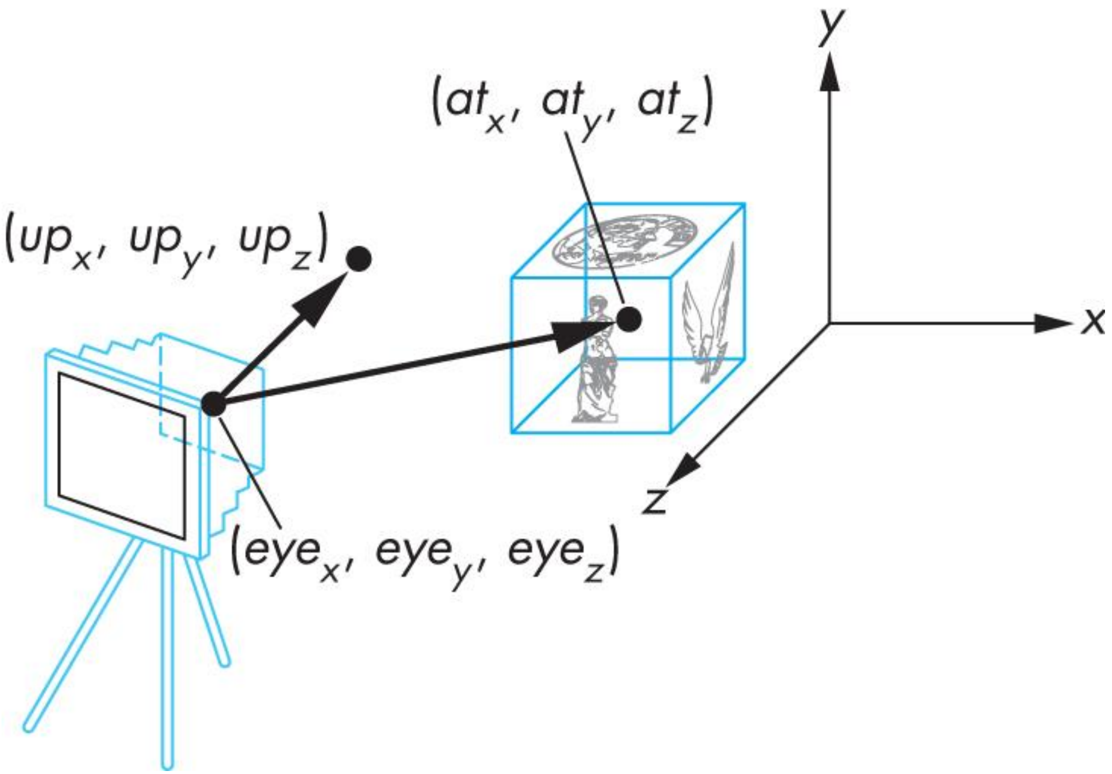
Transform Camera = Transform Scene

- Camera position is identified with a frame
- Either move and rotate the objects
- Or move and rotate the camera
- Initially, camera at origin, pointing in negative z-direction



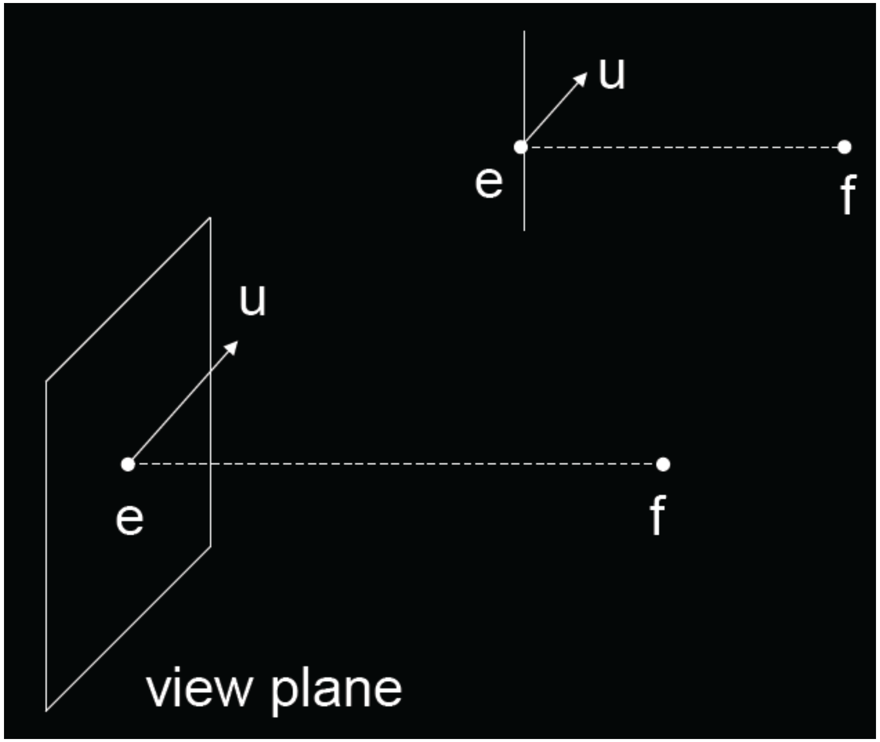
The Look-At Function

- Convenient way to position camera with three parameters:



The Look-At Function (OLD Style)

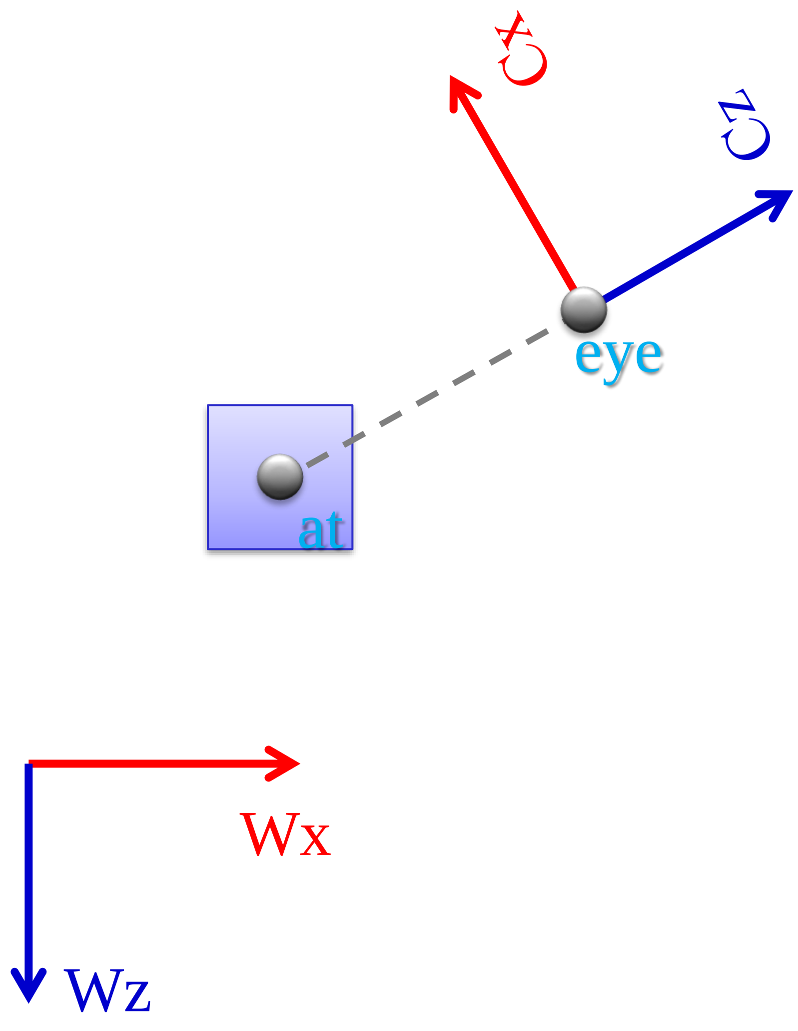
- `gluLookAt (ex,ey,ez, fx,fy,fz, ux,uy,uz);`
- e = eye point (eye)
- f = focus point (at)
- u = up vector (up)



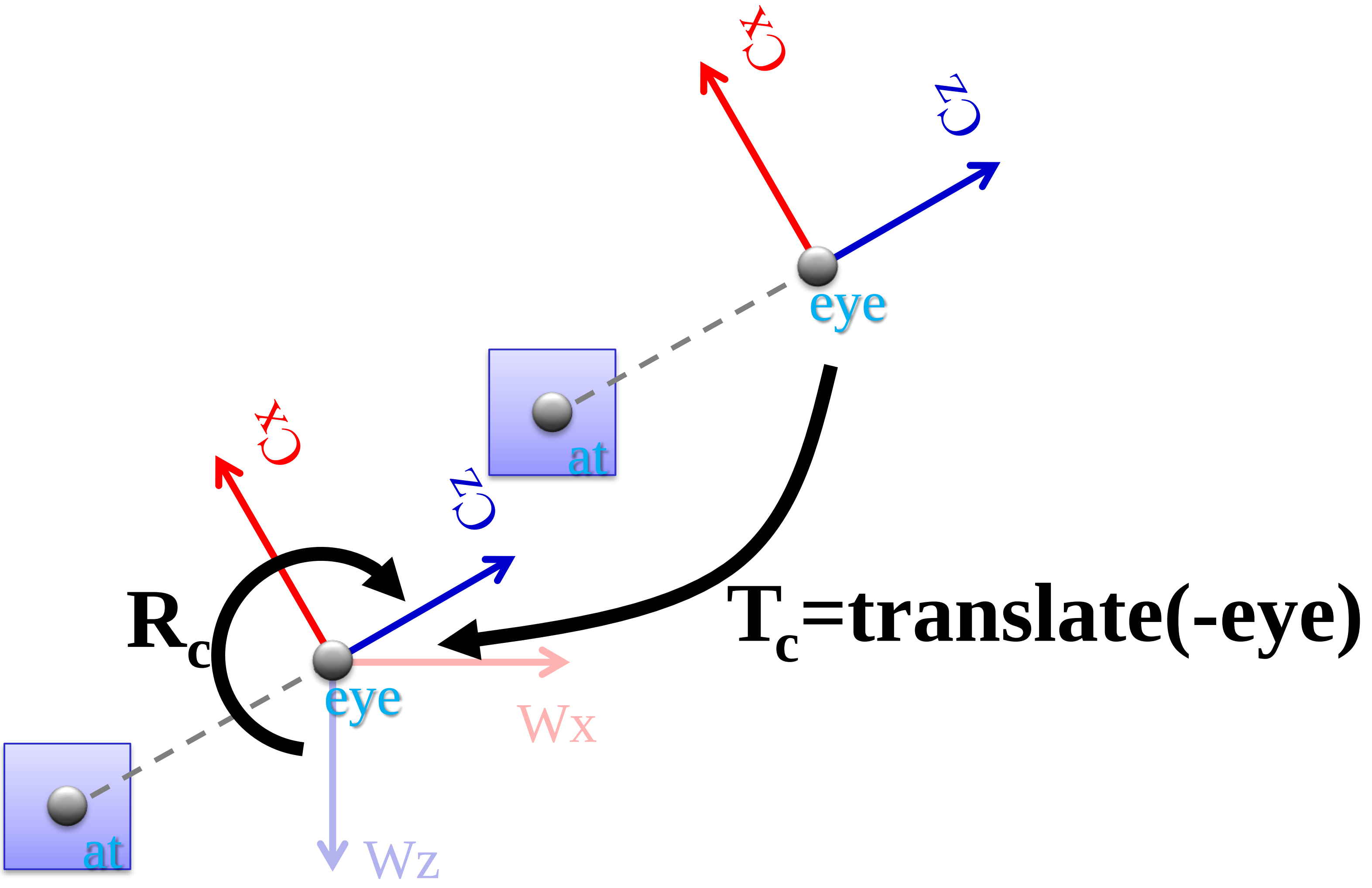
Old OpenGL code

```
void display()
{
      glClear (GL_COLOR_BUFFER_BIT |
            GL_DEPTH_BUFFER_BIT);
      glMatrixMode (GL_MODELVIEW);
      glLoadIdentity ();
      gluLookAt (ex,ey,ez,fx,fy,fz,ux,uy,uz);
      ...
      renderObjects();
      glutSwapBuffers();
}
```

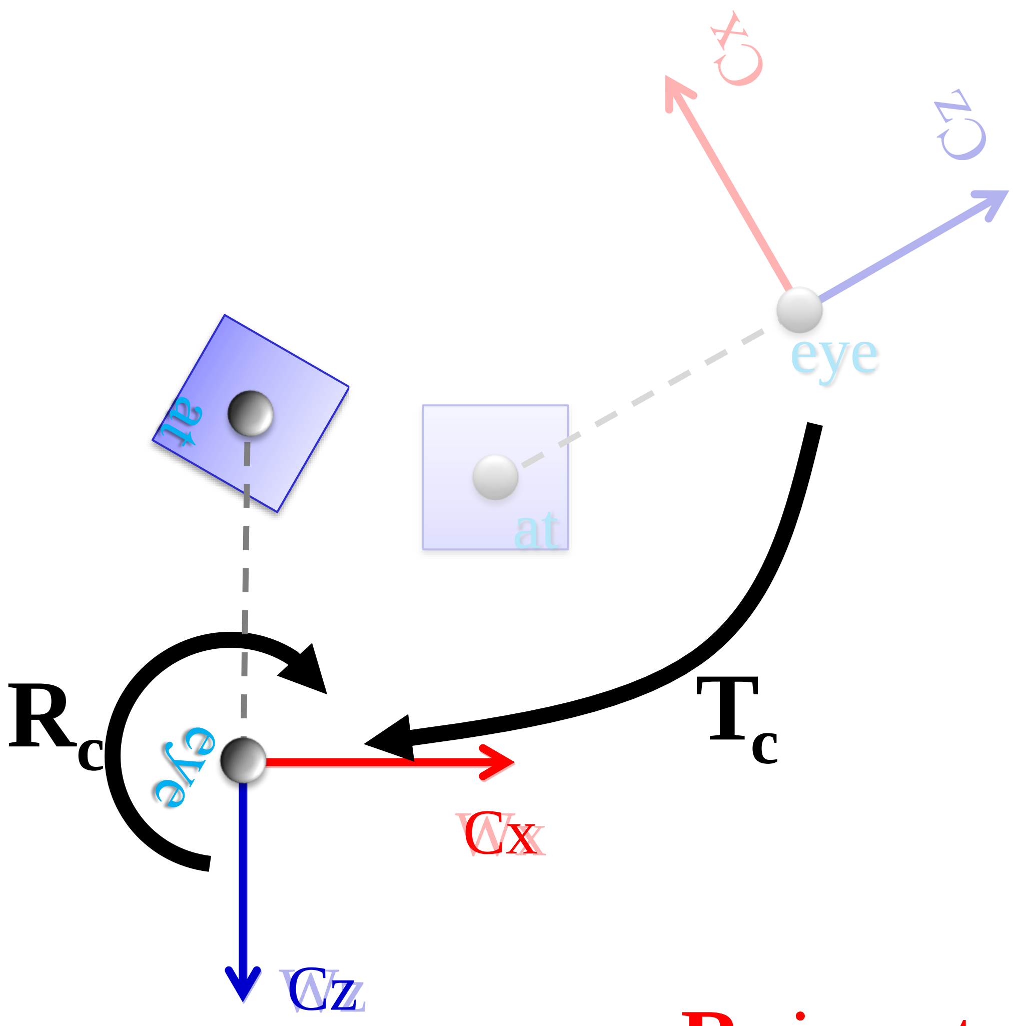
How to Compute?



2D case: Translate + Rotation



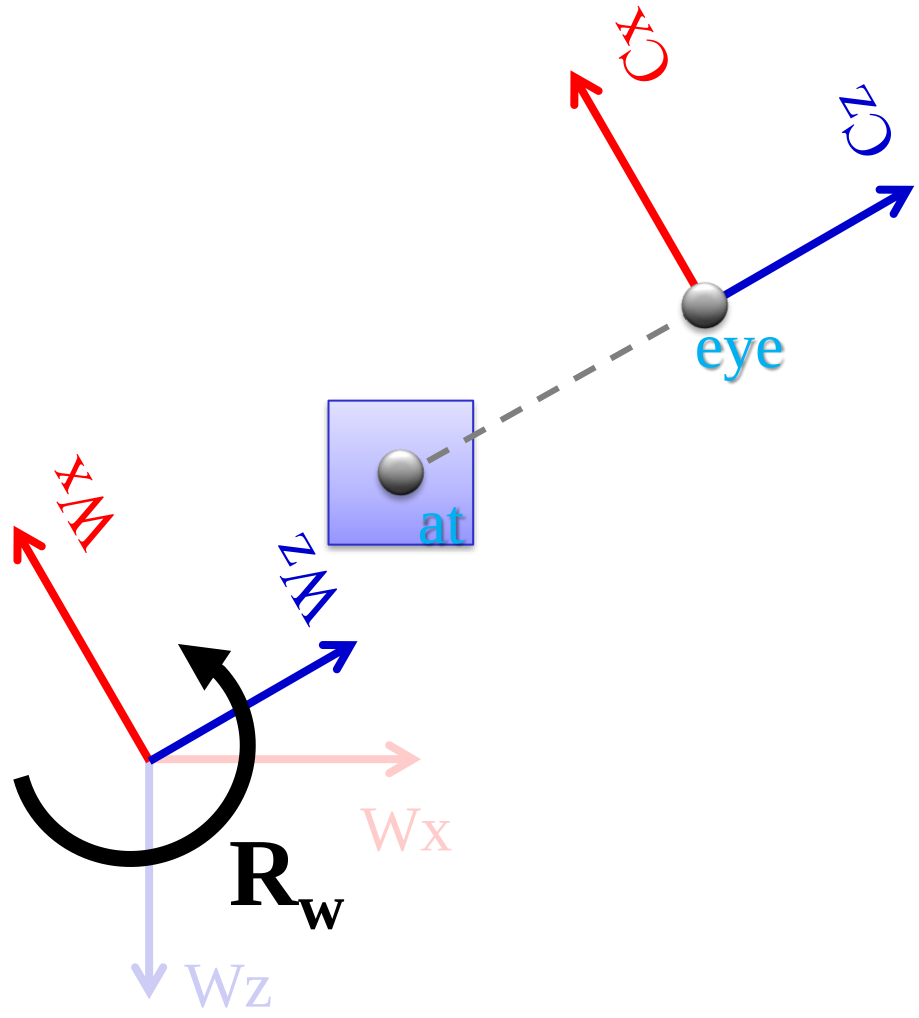
2D case: Translate + Rotation



$$V = R_c T_c$$

R_c is not easy to compute!

Instead, Think about inverse transform:



$$R_c = (R_w)^{-1}$$
$$= (R_w)^T$$

Implementing the Look-At Function

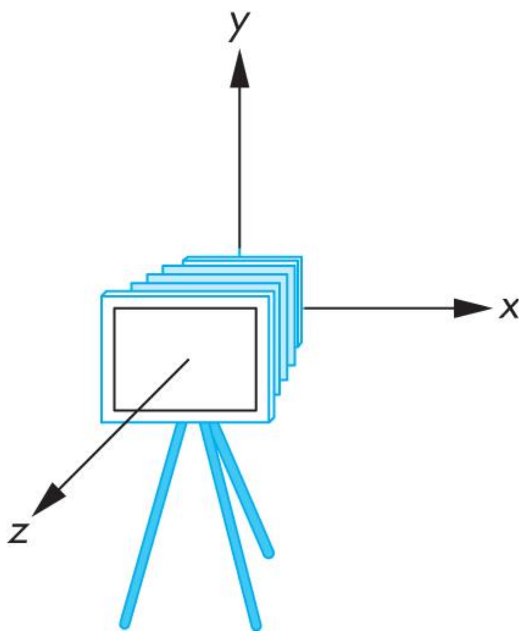
Plan:

1. Transform world frame to camera frame
   - Compose a rotation R with translation T
   - $W = T\,R$

2. Invert W to obtain viewing transformation V
   - $V = W^{-1} = (T\,R)^{-1} = R^{-1}\,T^{-1}$
   - Derive R, then T, then $R^{-1}\,T^{-1}$

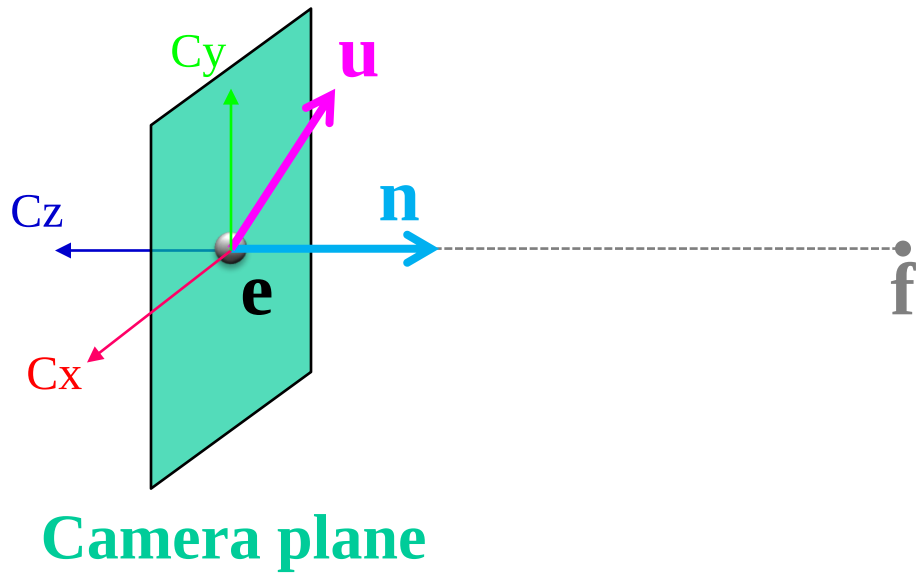
Viewing in OpenGL

- Remember:
  camera is pointing in the negative z direction



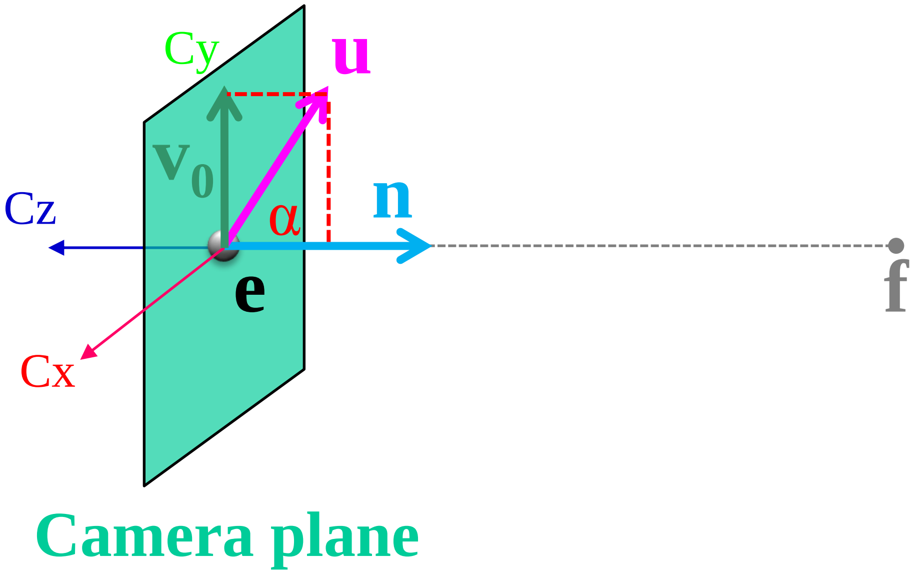
World Frame to Camera Frame I

- Camera points in negative z direction
- n = (f – e) / |f – e| is unit normal to view plane
- Therefore, R_w maps $[0\ 0\ {-1}]^T$ to $[nx\ ny\ nz]^T$



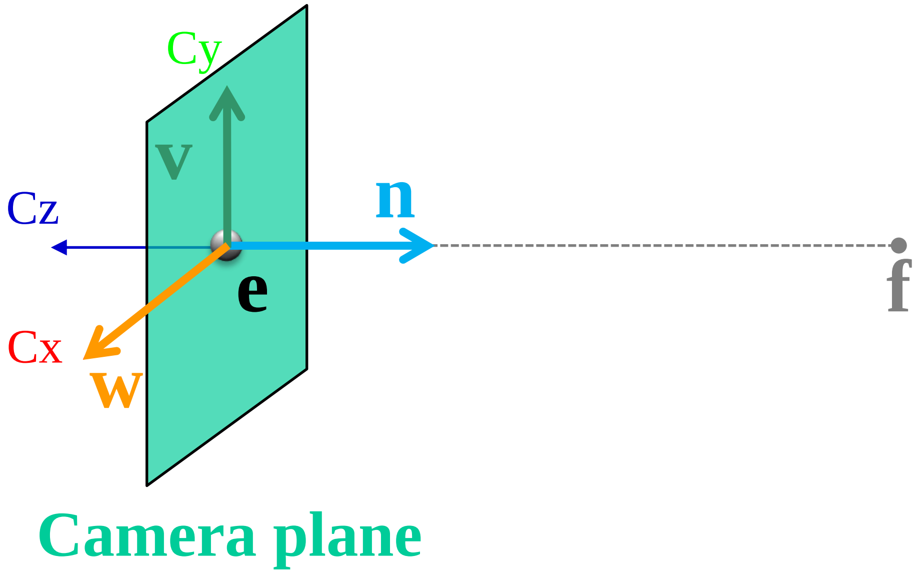
World Frame to Camera Frame II

- R_w maps $[0,1,0]^T$ to projection of u onto view plane
- This projection v equals:
  - $\alpha = (u \cdot n) / |n| = u \cdot n$
  - $v_0 = u - \alpha\, n$
  - $v = v_0 / |v_0|$



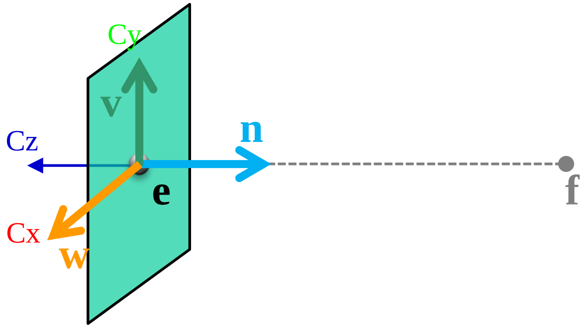
World Frame to Camera Frame III

- Set w to be orthogonal to n and v
- w = n x v
- (w, v, -n) is right-handed



Summary of Rotation

- **gluLookAt (ex,ey,ez, fx,fy,fz, ux,uy,uz);**
- n = (f – e) / |f – e|
- v = (u – (u · n) n) / |u – (u · n) n|
- w = n x v



- Rotation must map:
  - (1,0,0) to w
  - (0,1,0) to v
  - (0,0,-1) to n

$$\mathbf{R}_w = \begin{bmatrix} w_x & v_x & -n_x & 0 \\ w_y & v_y & -n_y & 0 \\ w_z & v_z & -n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

World Frame to Camera Frame IV

- Translation of origin to $e = [e_x\ e_y\ e_z\ 1]^T$

$$\mathbf{T}_w = \begin{bmatrix} 1 & 0 & 0 & e_x \\ 0 & 1 & 0 & e_y \\ 0 & 0 & 1 & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Camera Frame to Rendering Frame

- $V = W^{-1} = (T_w R_w)^{-1} = R_w^{-1} T_w^{-1}$
- R is rotation, so $R_w^{-1} = R_w^T$

$$\mathbf{R}^{-1} = \begin{bmatrix} w_x & w_y & w_z & 0 \\ v_x & v_y & v_z & 0 \\ -n_x & -n_y & -n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- T is translation, so T^{-1} negates displacement

$$\mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Putting All Together

- Calculate $V = R_w^{-1} T_w^{-1}$

$$\mathbf{V} = \mathbf{R}^{-1}\mathbf{T}^{-1} = \begin{bmatrix} w_x & w_y & w_z & -w_x e_x - w_y e_y - w_z e_z \\ v_x & v_y & v_z & -v_x e_x - v_y e_y - v_z e_z \\ -n_x & -n_y & -n_z & n_x e_x + n_y e_y + n_z e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- This is different from book [Angel, Ch. 4.3.2]
- There, u, v, n are right-handed (here: u, v, -n)

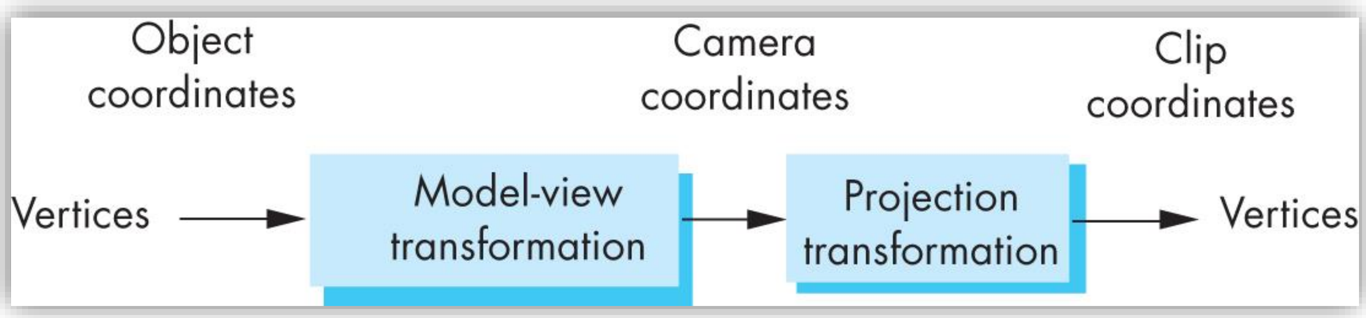
Projection Matrix
and its implementation

Topics

- Simple Parallel Projections
- Simple Perspective Projections

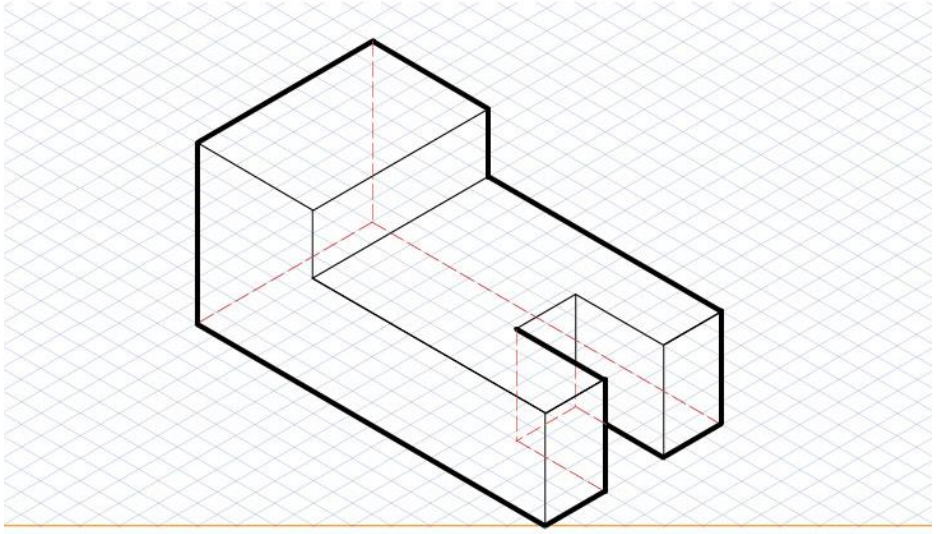
Projection Matrices

- Recall geometric pipeline



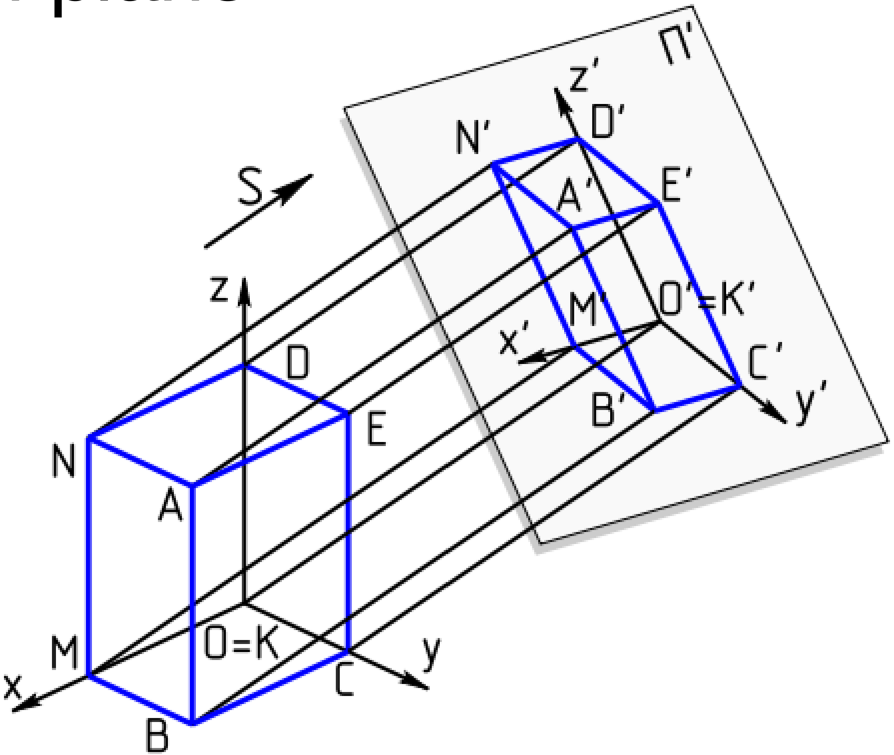
- Projection takes 3D to 2D
- Projections are not invertible
- Projections also described by 4x4 matrix
- Homogenous coordinates crucial
- Parallel and perspective projections

Simple Parallel Projection



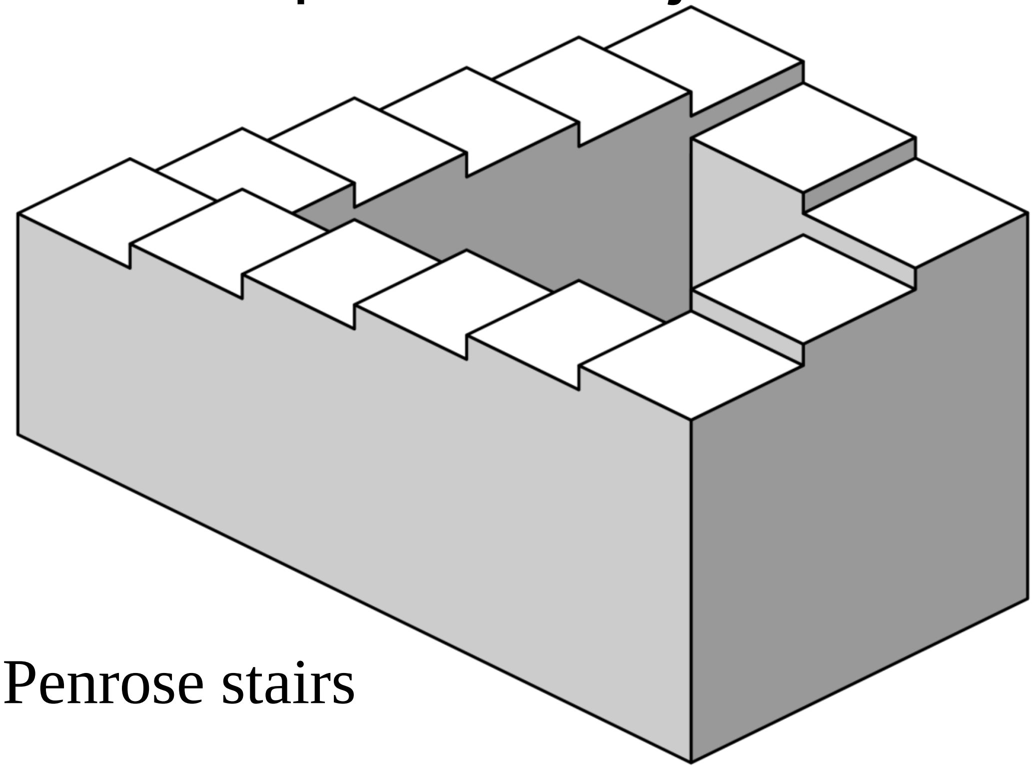
Parallel Projection

- Project 3D object to 2D via parallel lines
- The lines are not necessarily orthogonal to projection plane



Parallel Projection

- Problem: objects far away do not appear smaller
- Can lead to “impossible objects” :

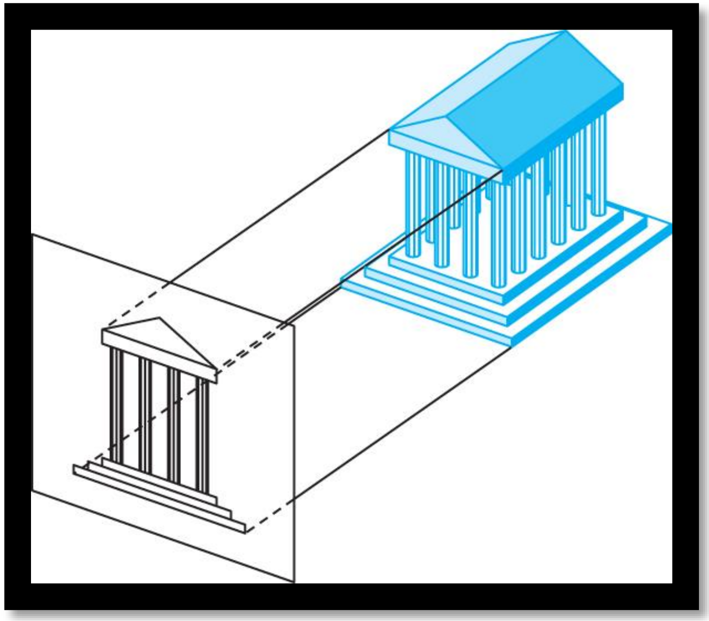


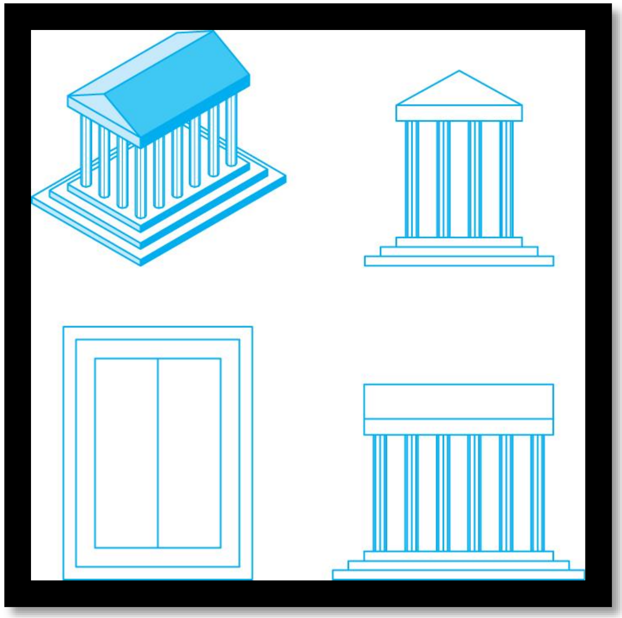
Penrose stairs

- Echochrome: https://www.youtube.com/watch?v=QfICeBtVy8U

Orthographic Projection

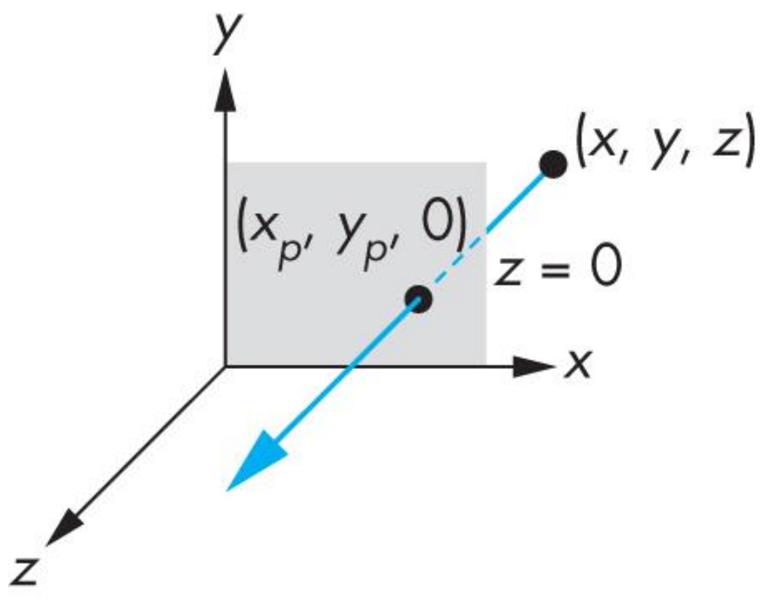
- A special kind of parallel projection: projectors perpendicular to projection plane
- Simple, but not realistic
- Used in blueprints (multiview projections)





Simple Orthographic Projection Matrix

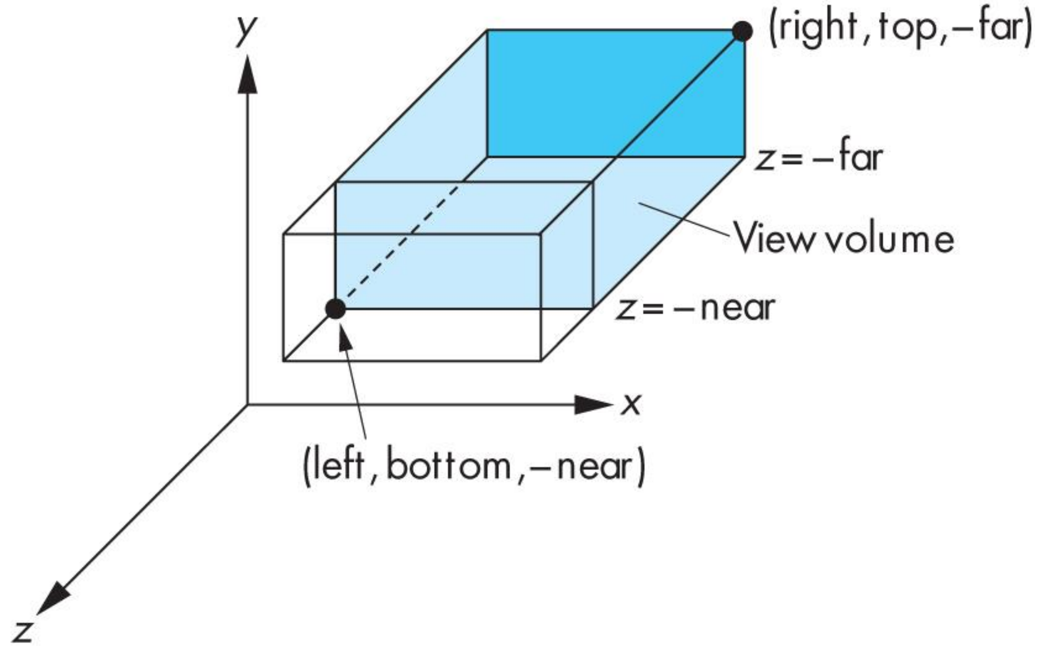
- Project onto z = 0
- $x_p = x$, $y_p = y$, $z_p = 0$
- In homogenous coordinates



$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

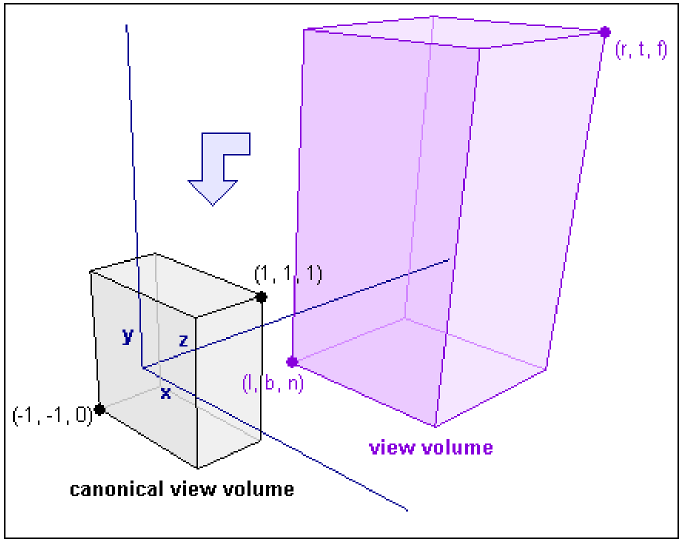
Orthographic Viewing in Old OpenGL

- glOrtho(xmin, xmax, ymin, ymax, near, far)



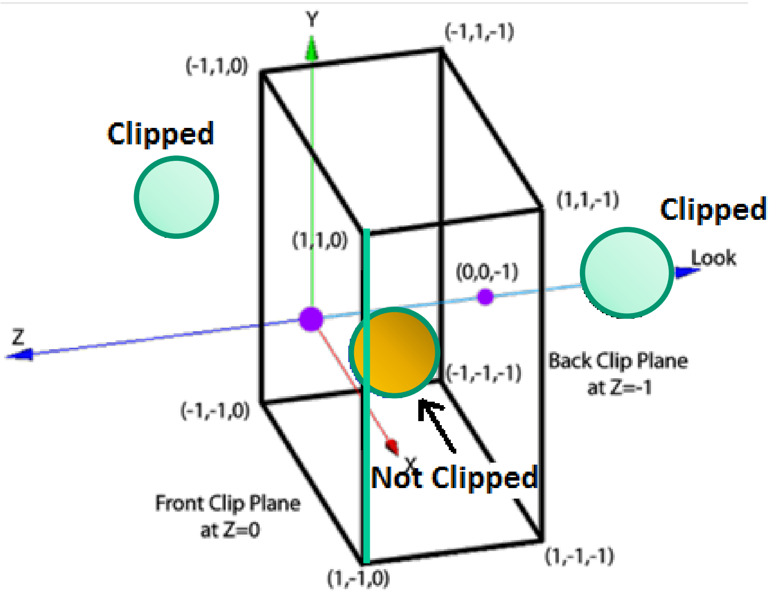
The Normalized view volume

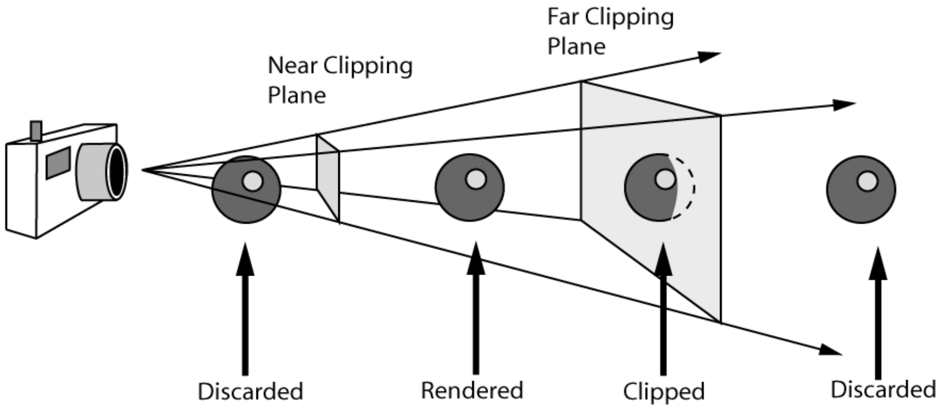
- How exactly do we take contents of an arbitrary view volume and project them to a 2D surface?
- Arbitrary view volume is too complex…
- Reduce it to a simpler problem! The **Normalized view volume!**
- Can also be called the *standard* or *unit* or *canonical* view volume



Clipping against the normalized view volume

- After applying normalizing transformation to all vertices in scene, anything that falls outside the bounds of the planes x = (-1,1), y = (-1, 1) and z = (0, -1), is clipped. Primitives that intersect the view volume must be partially clipped
- Most graphics packages such as OpenGL will do this step for you

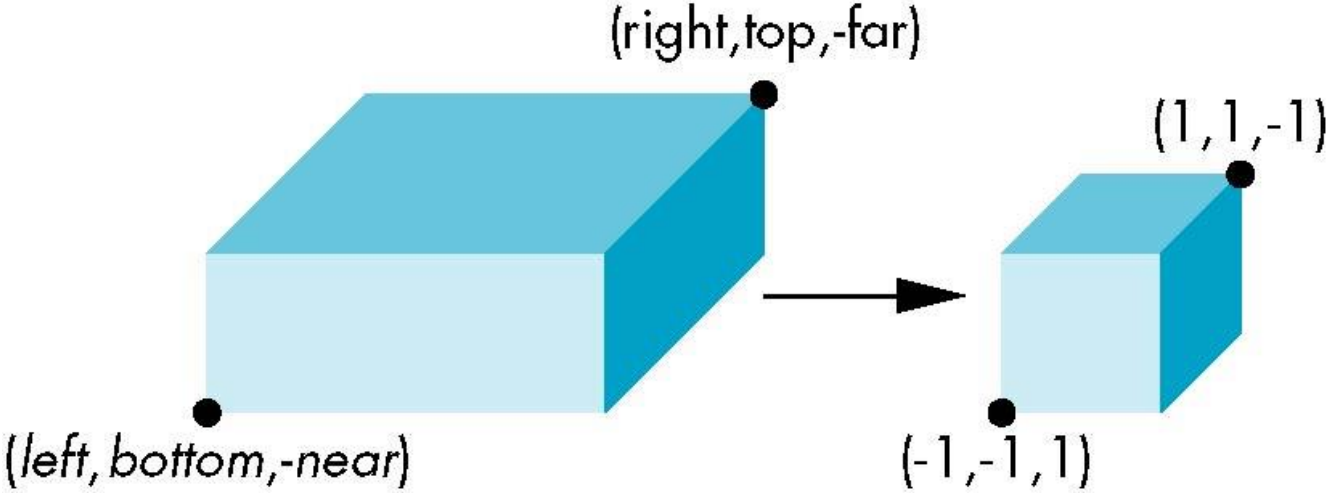




Orthogonal Normalization I

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

normalization $\Rightarrow$ find transformation to convert specified clipping volume to default



Orthogonal Matrix I

- Two steps
  - Move center to origin

    T(-(left+right)/2, -(bottom+top)/2, (near+far)/2)
  - Scale to have sides of length 2

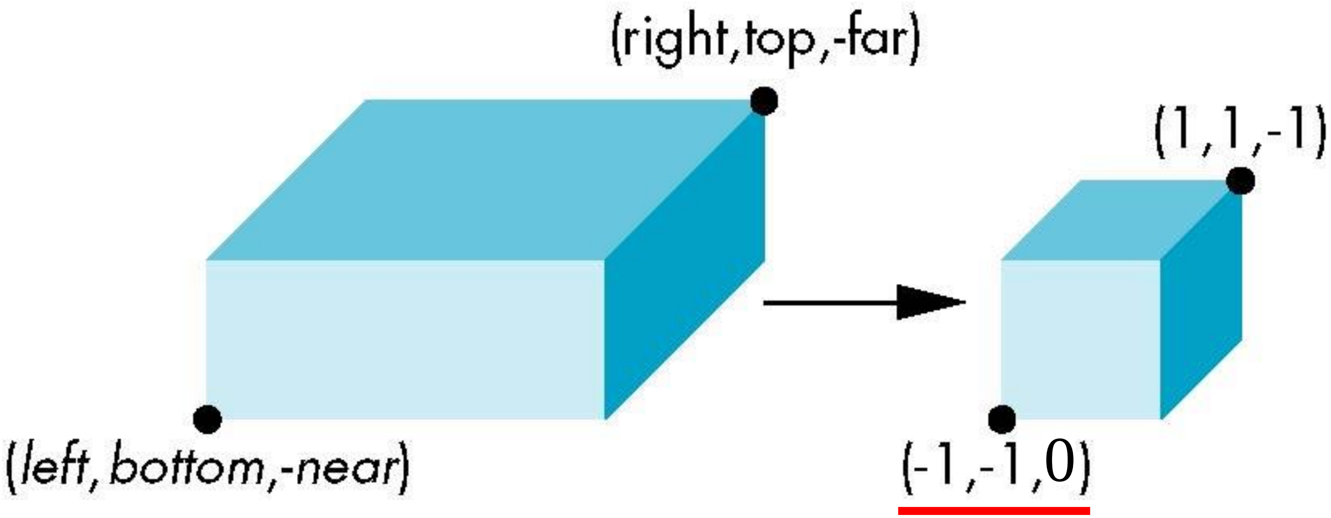
    S(2/(left-right),2/(top-bottom),2/(far-near))

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

Orthogonal Normalization II

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

normalization ⇒ find transformation to convert specified clipping volume to default



Orthogonal Matrix II

- Two steps
  - Move center to origin

    Translate(-(left+right)/2, -(bottom+top)/2, near)

  - Scale to have sides of length 2 for x, y, and length 1 for z

    Scale(2/(left-right),2/(top-bottom),1/(far-near))

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