Introduction to Computer Graphics

Viewing

葉奕成 I-Cheng (Garrett) Yeh

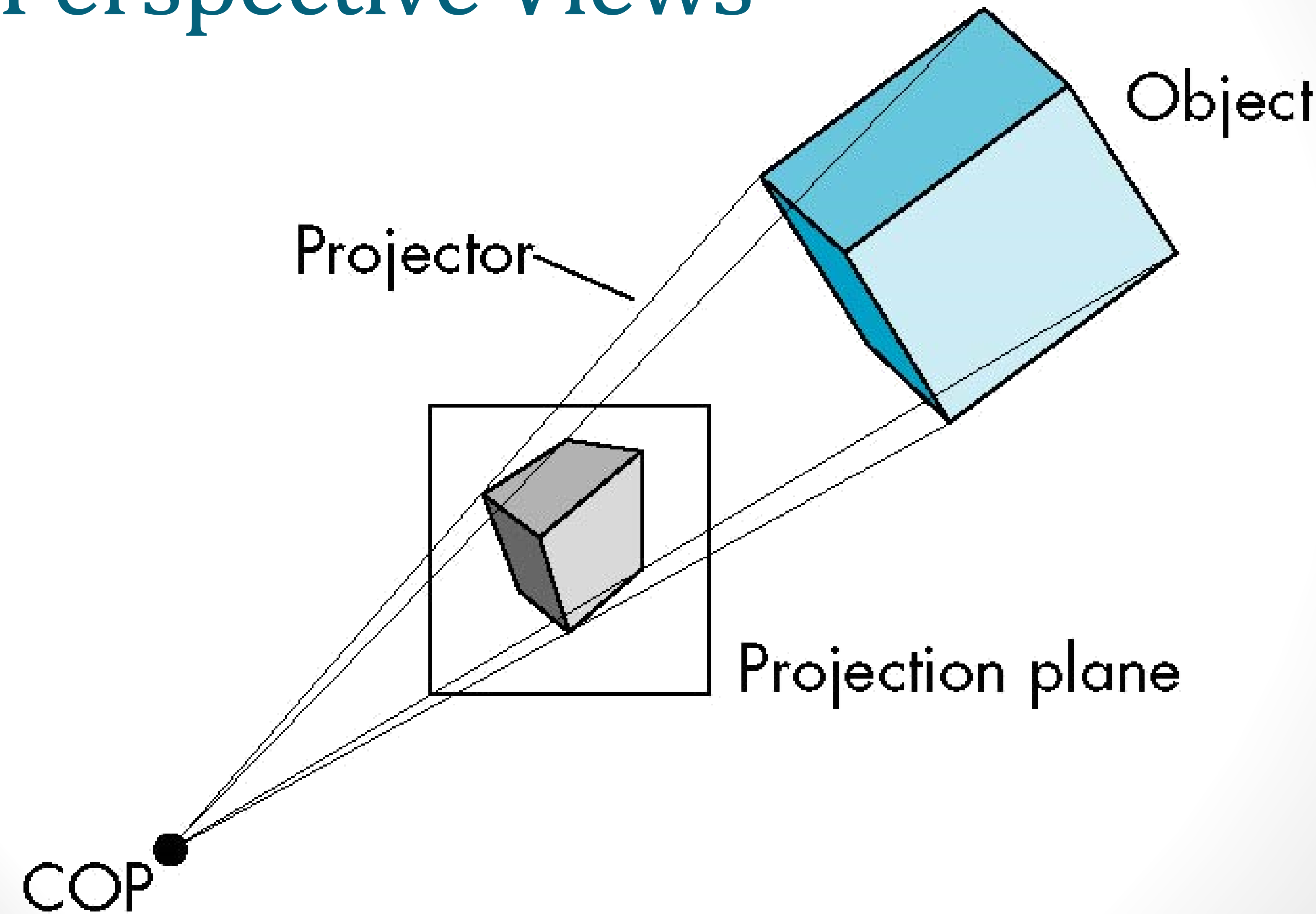
Classical Viewing

- Viewing requires three basic elements
  - One or more objects
  - A viewer with a projection surface
  - Projectors that go from the object(s) to the projection surface
- Classical views are based on the relationship among these elements
  - The viewer picks up the object and orients it how she would like to see it
- Each object is assumed to constructed from flat *principal faces*
  - Buildings, polyhedron, manufactured objects

Similarities between Classical and Computer Viewing

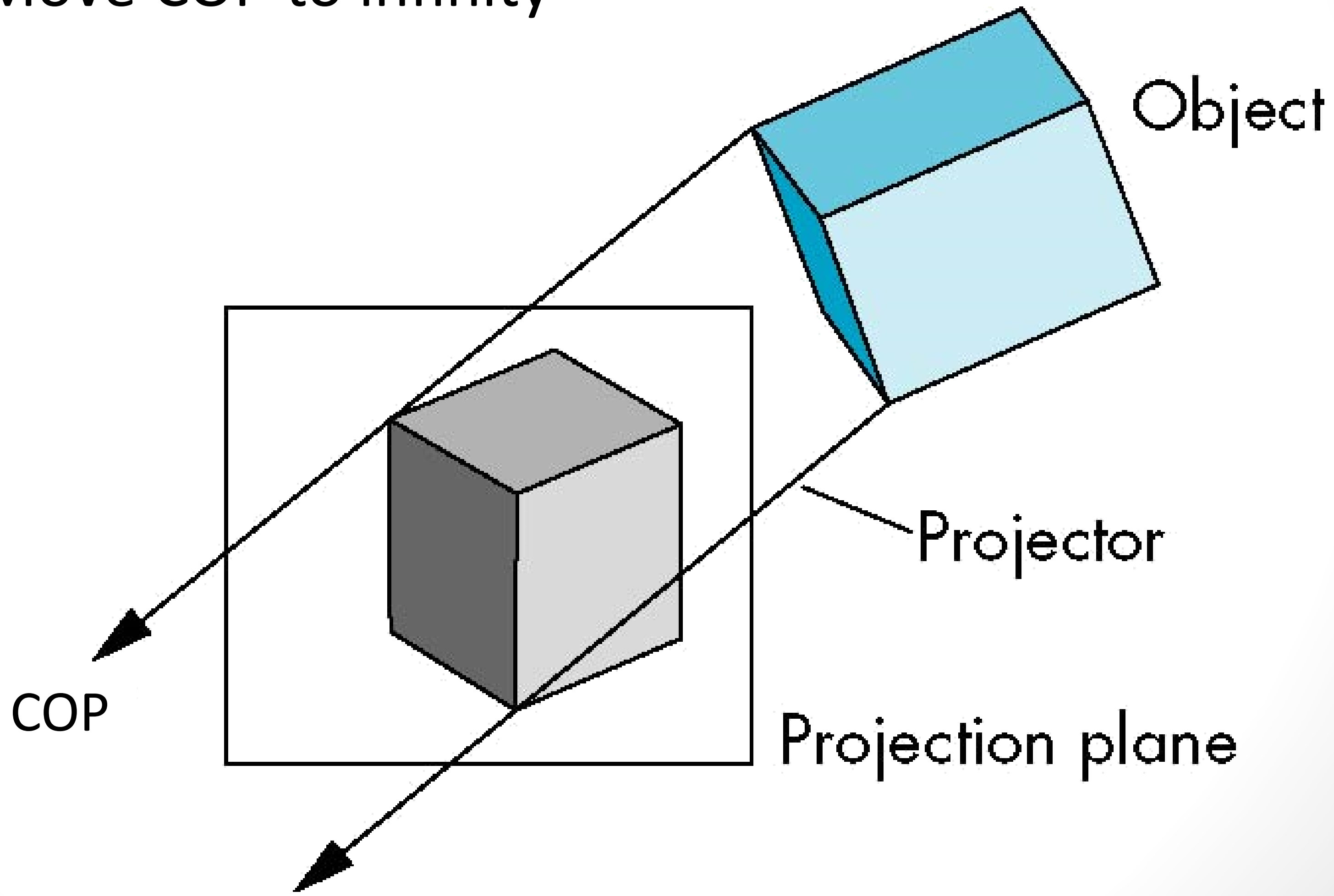
- Basic elements are the same
  - Objects, a viewer, projectors, a projection plane.
- Projectors meet at the COP.
  - COP
    - Center of the lens in the camera(eye)
    - Origin of the camera frame
- Projection surface – a plane
- Projectors – straight lines

Computer Viewing (type 1) - Perspective Views



Computer Viewing (type 2) - Parallel (Orthographic) Views

- Move COP to infinity

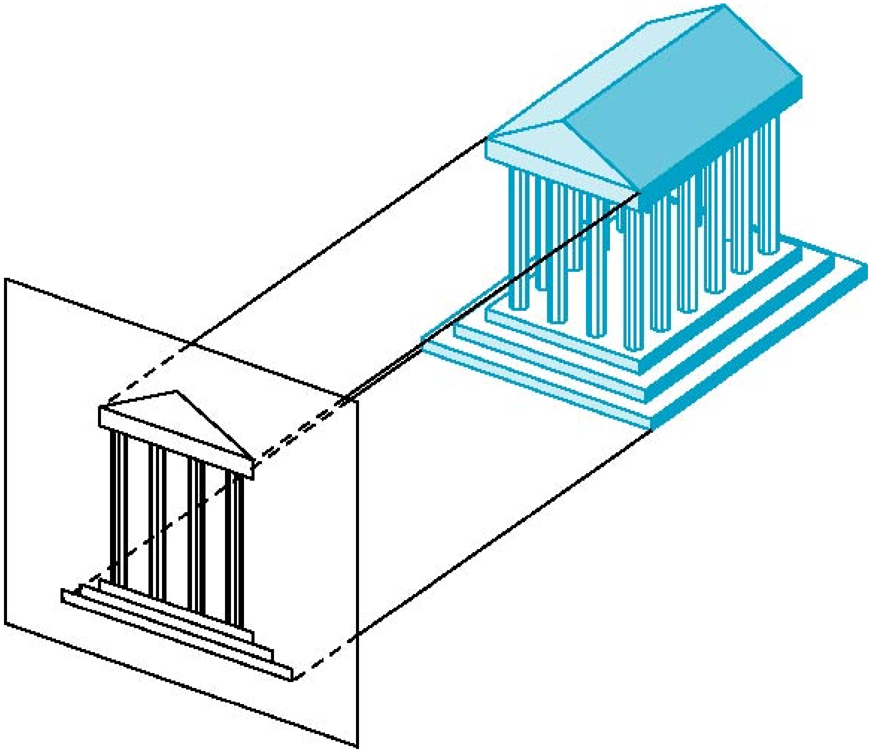


Planar Geometric Projections

- Standard projections project onto a plane
- Projectors are lines that either
  - converge at a center of projection
  - are parallel
- Such projections preserve (straight) lines
  - but not necessarily angles
- Non-planar projections are needed for applications such as map construction
  - The preservation of lines are no guarantee

Orthographic Projection

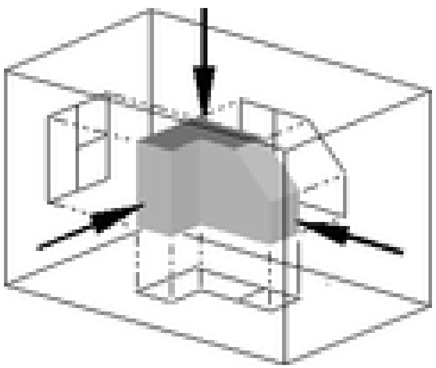
Projectors are orthogonal to projection surface



Multi-view Orthographic Projection

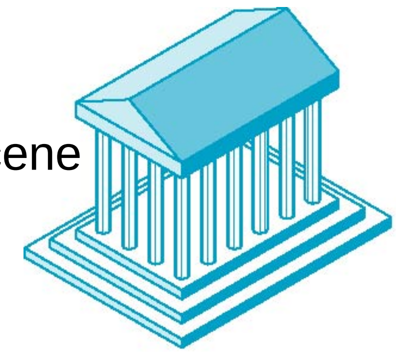
- Projection plane parallel or orthogonal to principal façade

in CAD and architecture, we often display at least three views



third-angle projection

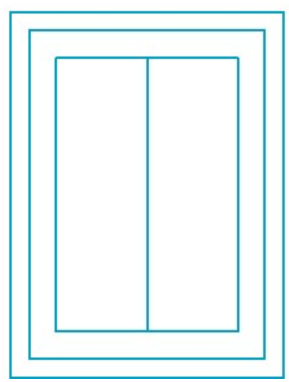
3d scene

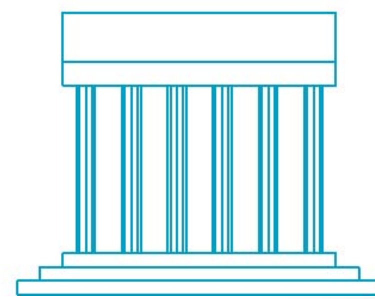


principal façade

front

top





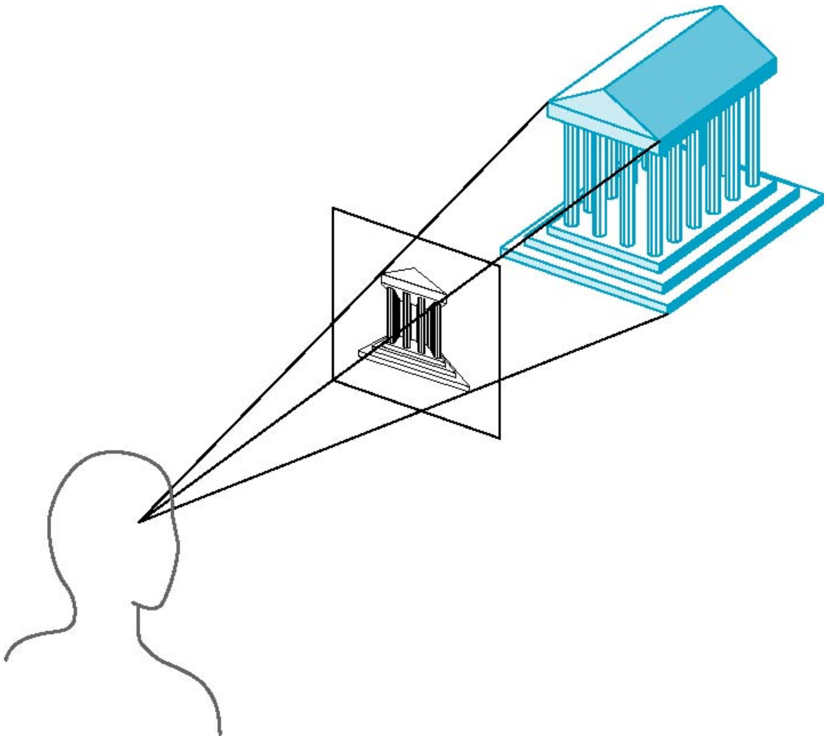
side

Advantages and Disadvantages

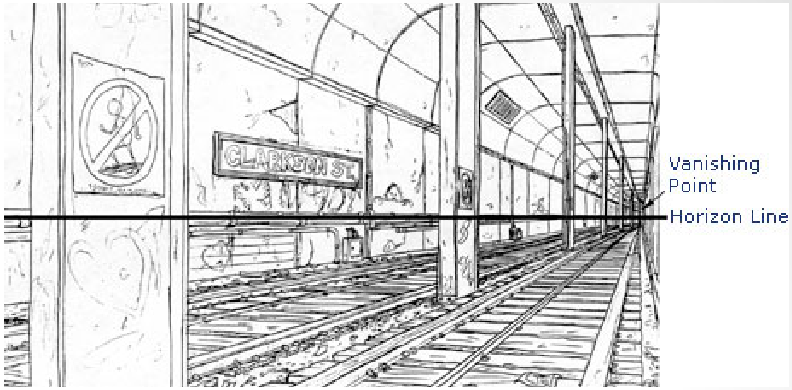
- Preserves both distances and angles
  - Shapes preserved
  - Can be used for measurements
    - Building plans
    - Manuals
- Cannot see what object really looks like because many surfaces hidden from view

Perspective Projection

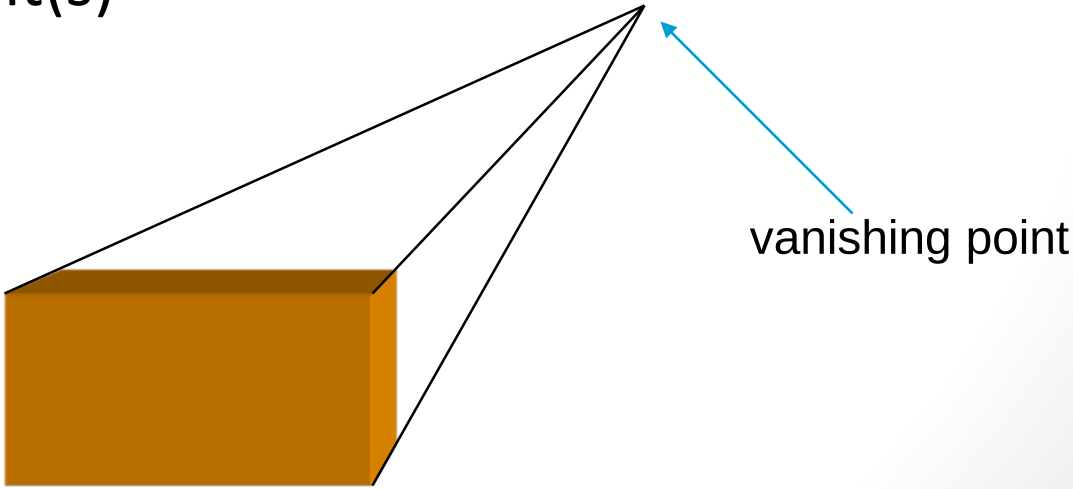
Projectors converge at center of projection



Vanishing Points



- Parallel lines (not parallel to the projection plane) on the object converge at a single point in the projection (the *vanishing point*)
- Drawing simple perspectives by hand uses these vanishing point(s)



One-Point Perspective

- One principal face parallel to projection plane
- One vanishing point for cube

Advantages and Disadvantages

- Objects further from viewer are projected smaller than the same sized objects closer to the viewer => Looks realistic
- Equal distances along a line are not projected into equal distances (*non-uniform foreshortening*)
- Angles preserved only in planes parallel to the projection plane
- More difficult to construct by hand than parallel projections (but not more difficult by computer)

Viewing with a computer

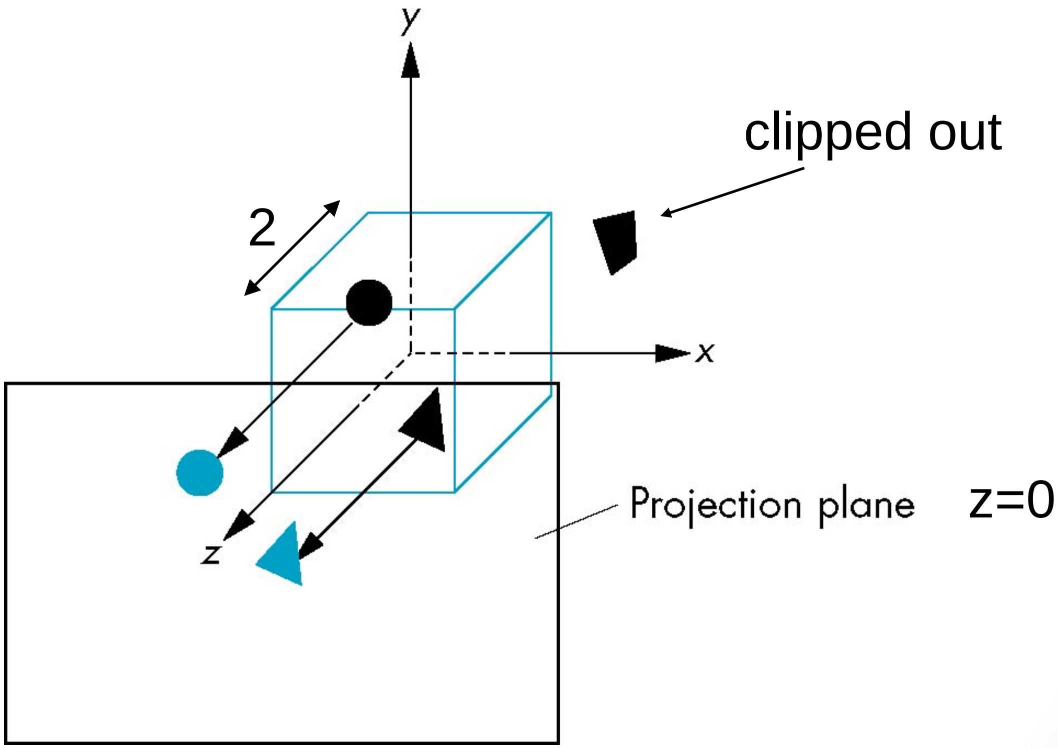
- There are three aspects of the viewing process, all of which are implemented in the pipeline,
  - Positioning the camera
    - Setting the model-view matrix
  - Selecting a lens
    - Setting the projection matrix
  - Clipping
    - Setting the view volume

The OpenGL Camera

- In OpenGL, initially the world and camera frames are the same
  - Default model-view matrix is an identity
- The camera is located at origin and points in the negative z direction
- OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin
  - Default projection matrix is an identity
  - Default projection is orthographic

Default Projection

Default projection is orthographic



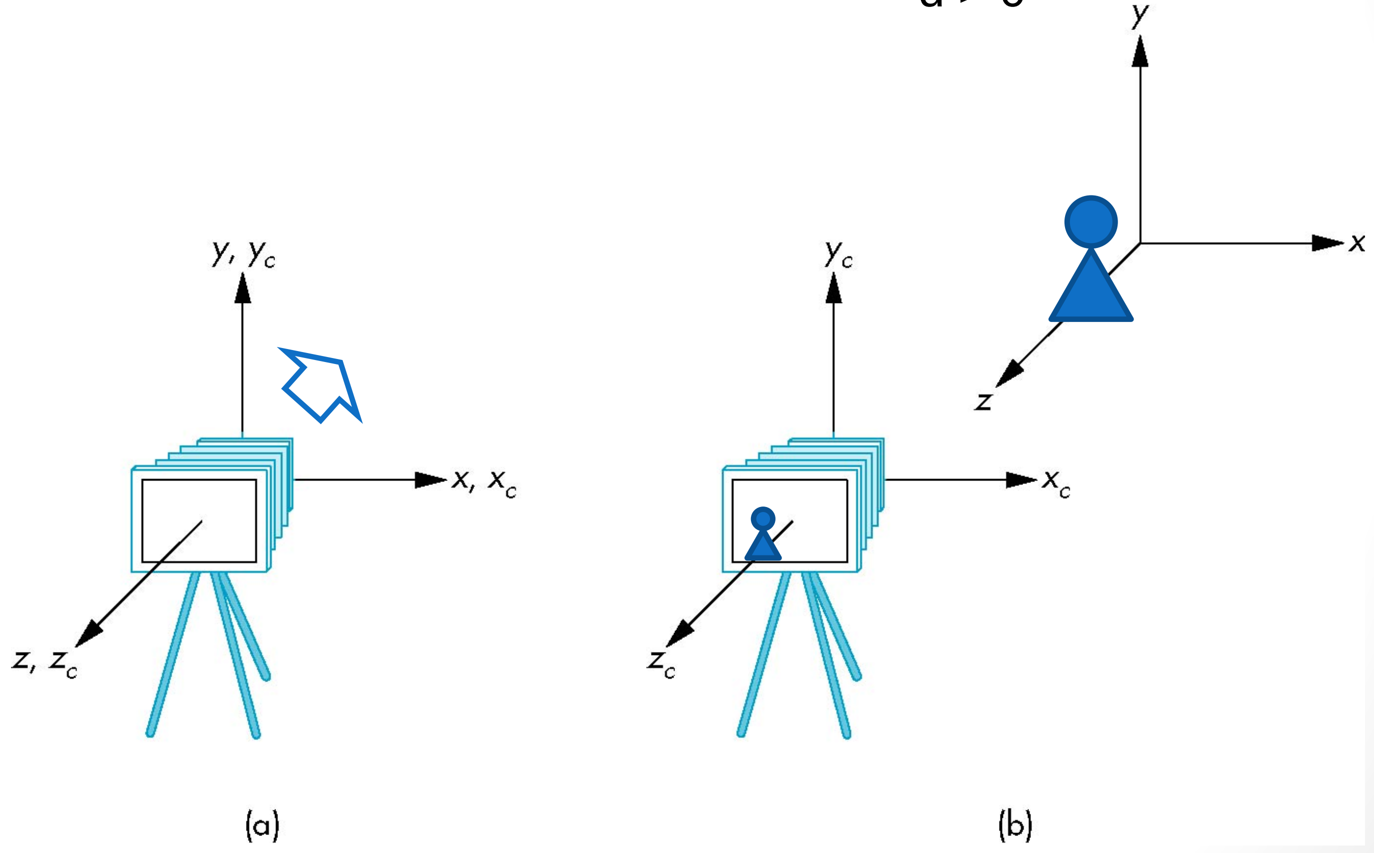
Positioning the Camera

- If we want to visualize object with both positive and negative z values 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 and are determined by the model-view matrix
  - Want a translation (**`glTranslatef(0.0,0.0,-d);`**)
  - **`d > 0`**

Moving Camera back from Origin

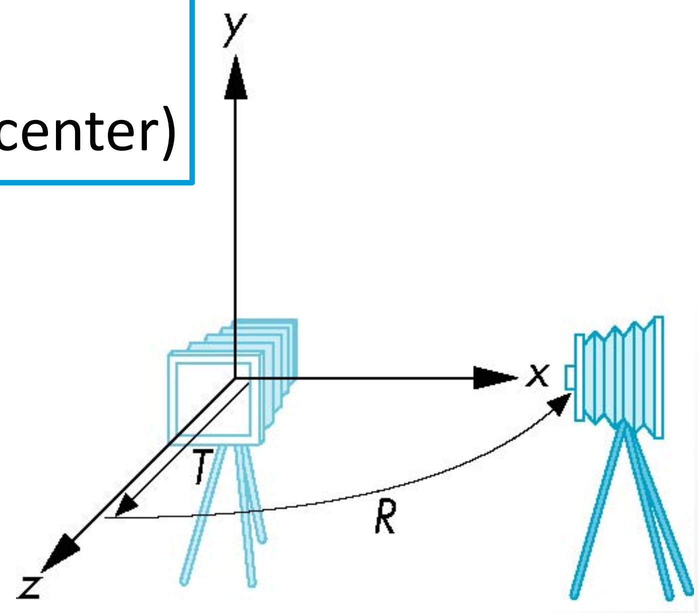
default frames

frames after translation by –d
$d > 0$



Moving the Camera

- We can move the camera to any desired position by a sequence of rotations and translations
- Example: side view
  - Move it away from origin
  - Rotate the camera (use o as center)
  - Model-view matrix C = TR



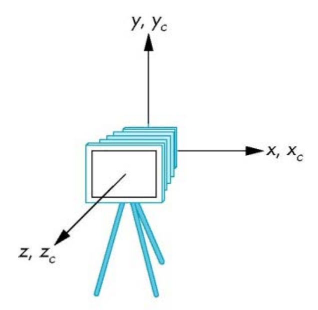
OpenGL code

- Remember that last transformation specified is first to be applied

```
glMatrixMode(GL_MODELVIEW)
glLoadIdentity();
glTranslatef(0.0, 0.0, -d);
glRotatef(-90.0, 0.0, 1.0, 0.0);
DrawTriangle(a, b, c);
```

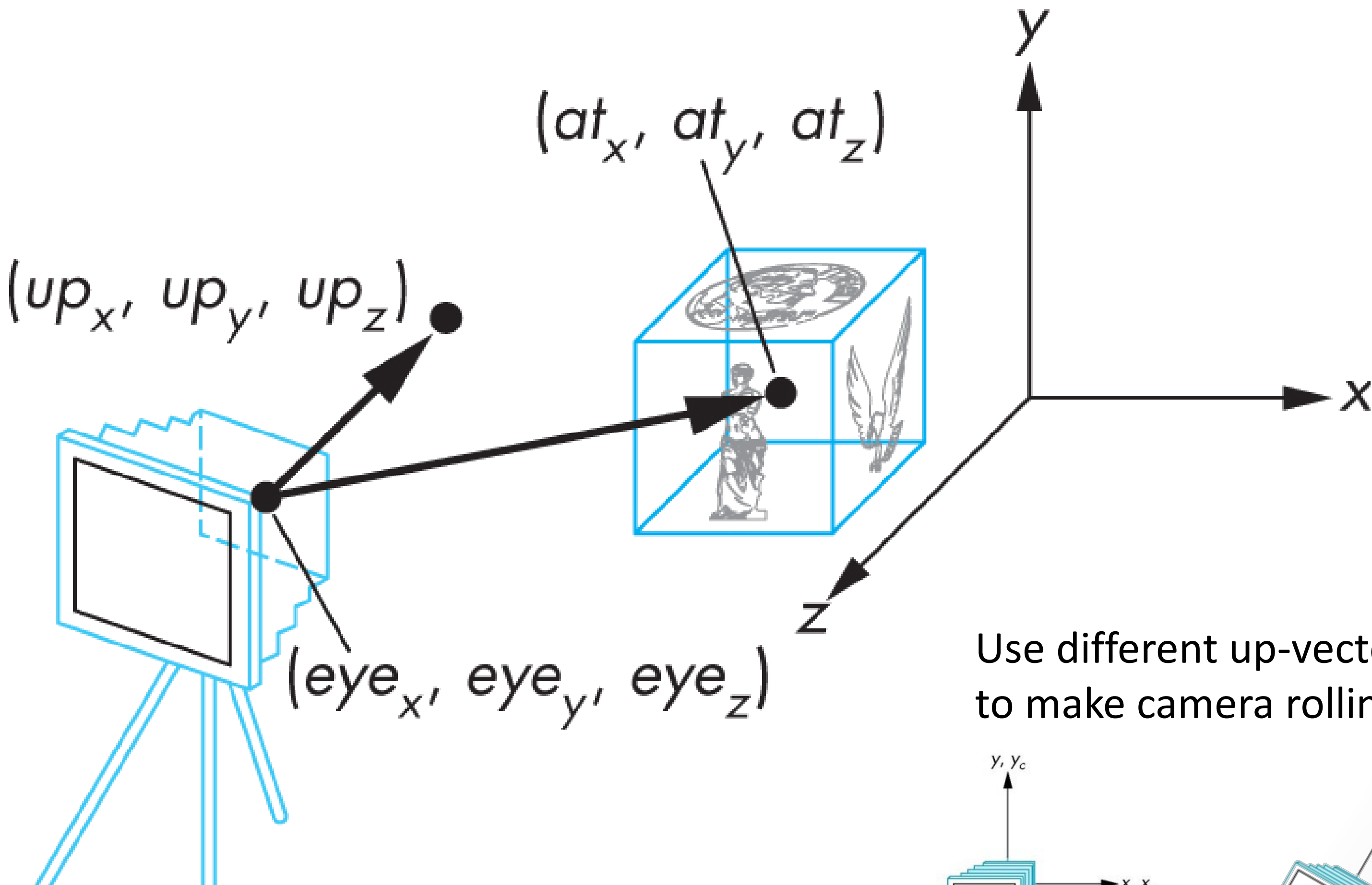
The LookAt Function

- The GLU library contains the function glLookAt to from the required modelview matrix through a simple interface
- Note the need for setting an up direction
- Still need to initialize
  - Can concatenate with modeling transformations
- Example: isometric view of cube aligned with axies

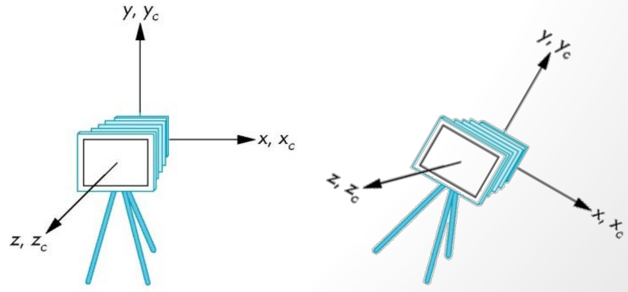



```
glMatrixMode(GL_MODELVIEW):
glLoadIdentity();
gluLookAt(1.0,1.0,1.0,0.0,0.0,0.0,0.0,1.0,0.0);
```

gluLookAt(eyex, eyey, eyez, atx, aty, atz, upx, upy, upz)

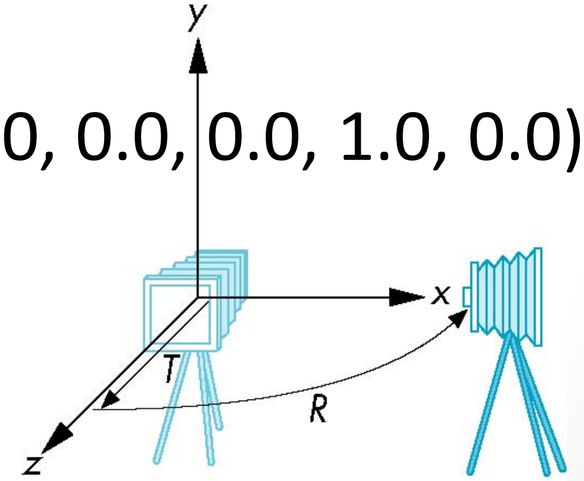


Use different up-vector to make camera rolling



Live demo

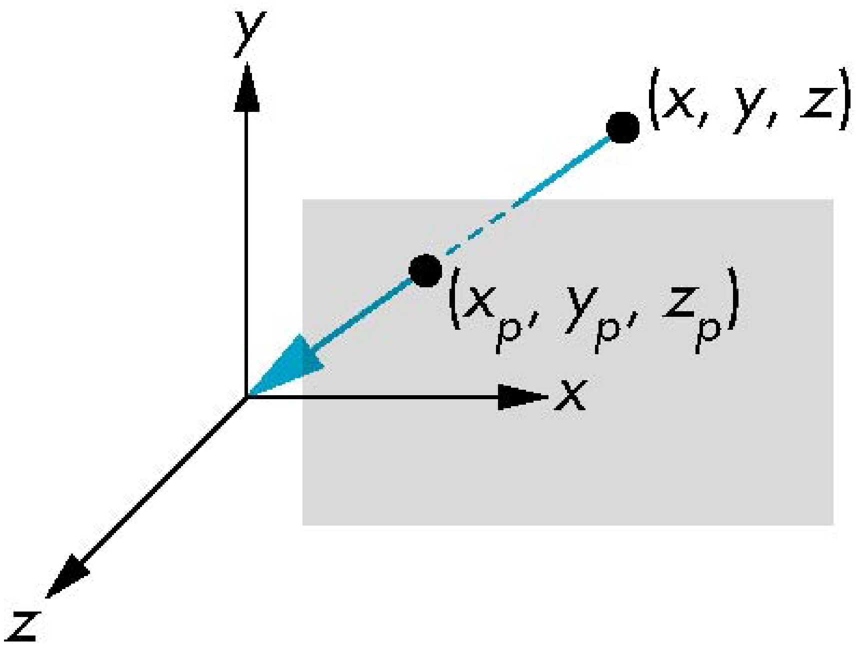
- gluLookAt(0.0, 0.0, 50.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  - = glTranslatef(0.0, 0.0, -50.0);
- gluLookAt(0.0, 50.0, 50.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  - = glRotatef(45.0, 1.0, 0.0, 0.0);
  - + glTranslatef(0.0, -50.0, -50.0);
- gluLookAt(50.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  - = glTranslatef(0.0, 0.0, -50.0);
  - + glRotatef(-90, 0.0, 1.0, 0.0);



- Let's check the example "car_02+testCamera.cpp"

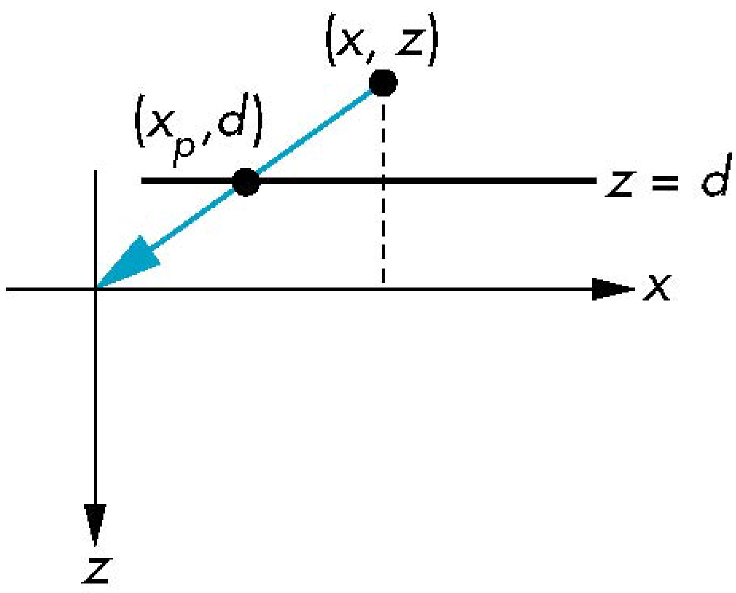
Perspective Projections

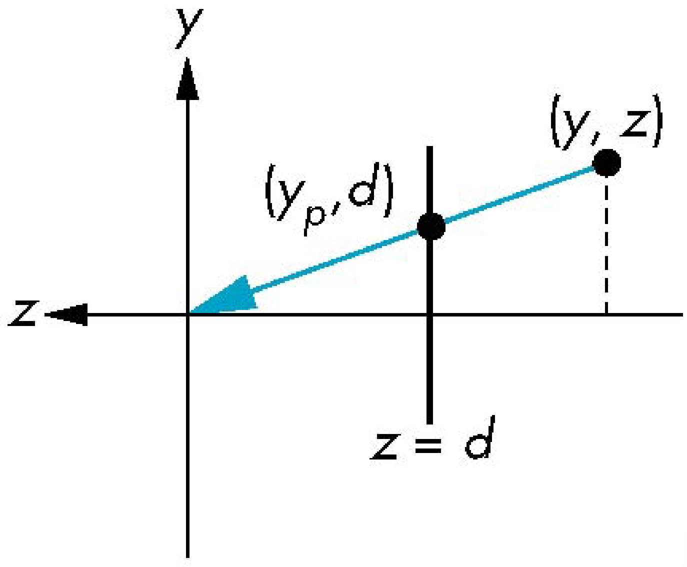
- Center of projection at the origin
- Projection plane $z_p = d, d < 0$



Perspective Equations

Consider top and side views





$x_p = \dfrac{x}{z/d}$ $\qquad y_p = \dfrac{y}{z/d}$ $\qquad z_p = d$

Homogeneous Coordinate Form

consider $\mathbf{p} = \mathbf{Mq}$ where

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

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

Perspective Division

- However $w \neq 1$, so we must divide by w to return from homogeneous coordinates
- This *perspective division* yields the desired perspective equations

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

- Projection pipeline

Orthographic Projections

- The default projection in the eye (camera) frame is orthographic
- For points within the default view volume

$$x_p = x$$
$$y_p = y$$
$$z_p = 0$$

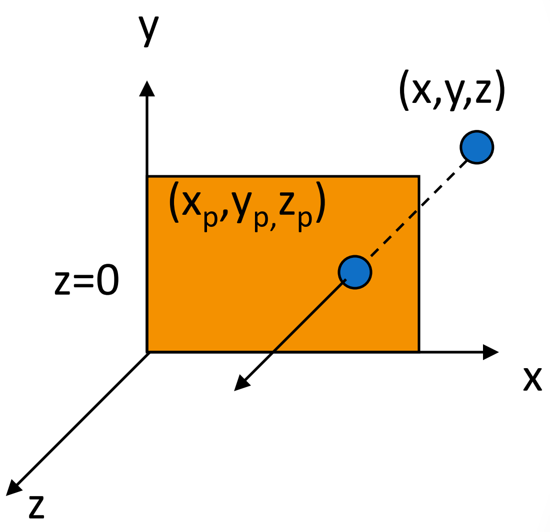
- Most graphics systems use *view normalization*
  - All other views are converted to the default view by transformations that determine the projection matrix
  - Allows use of the same pipeline for all views

Homogeneous Coordinate Representation

$x_p = x$
$y_p = y$
$z_p = 0$
$w_p = 1$

$\mathbf{p}_p = \mathbf{M}\mathbf{p}$

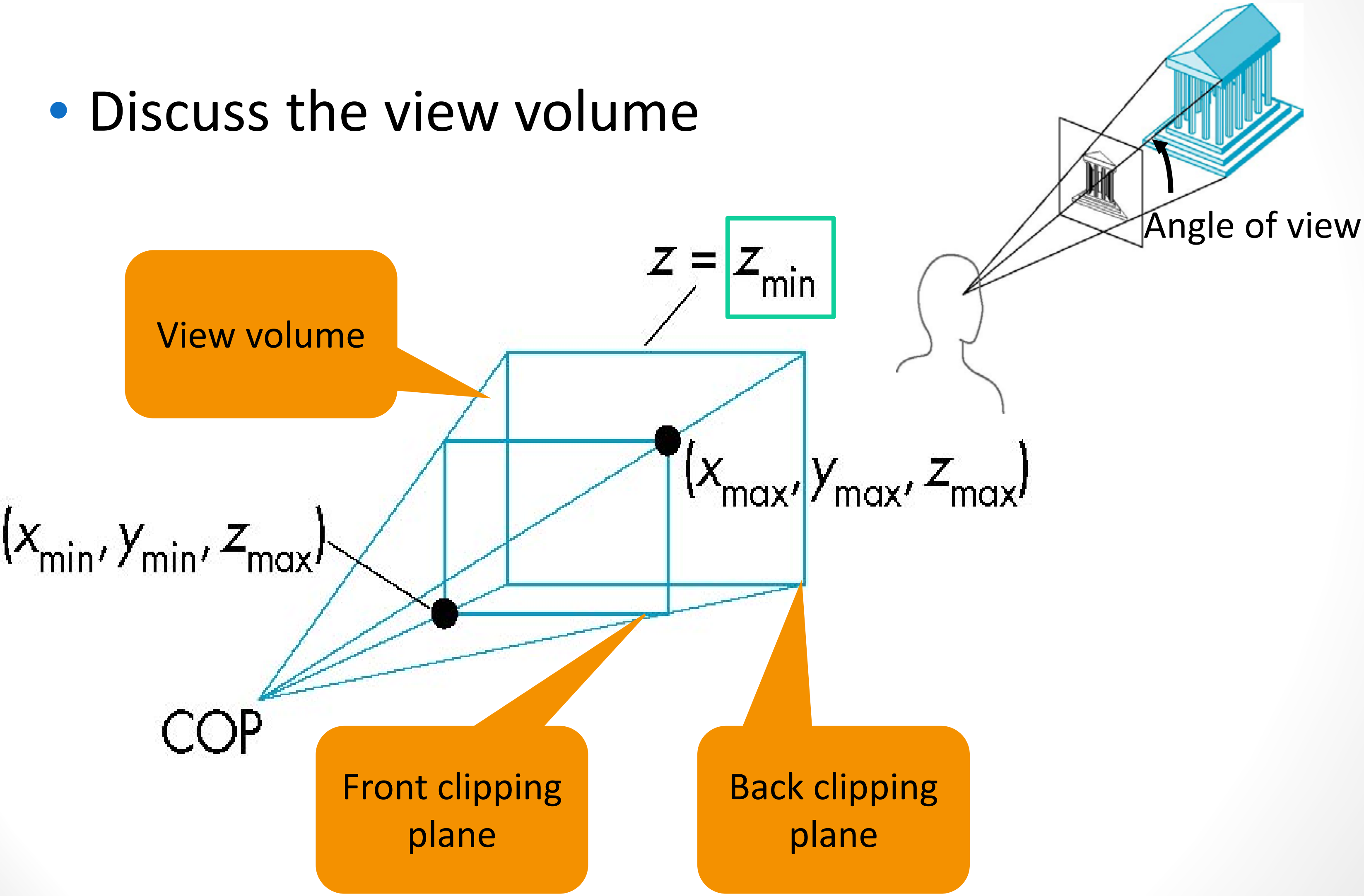
$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



In practice, we can let **M** = **I** and set the *z* term to zero later

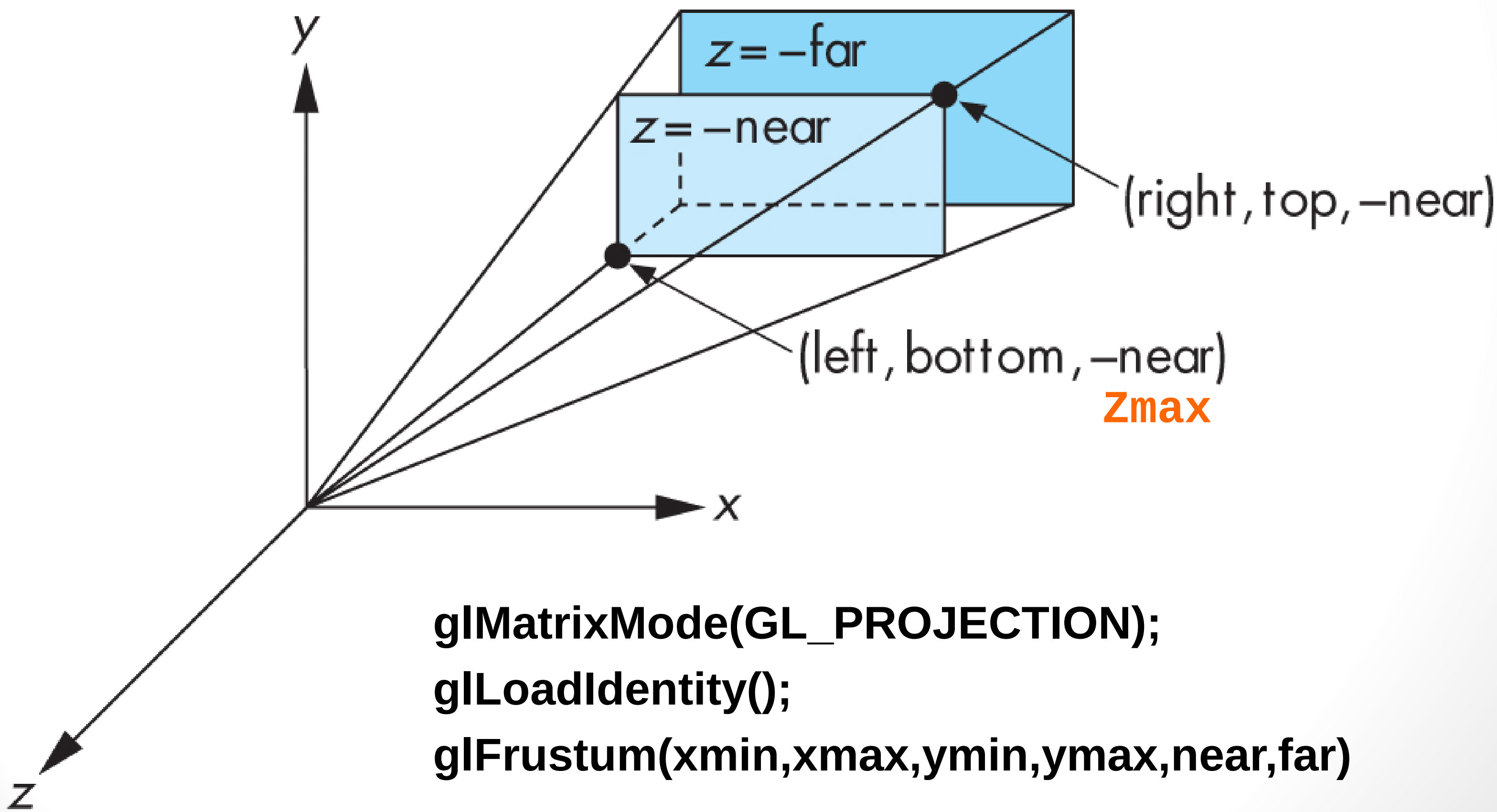
Projections in OpenGL

- Discuss the view volume



OpenGL Perspective

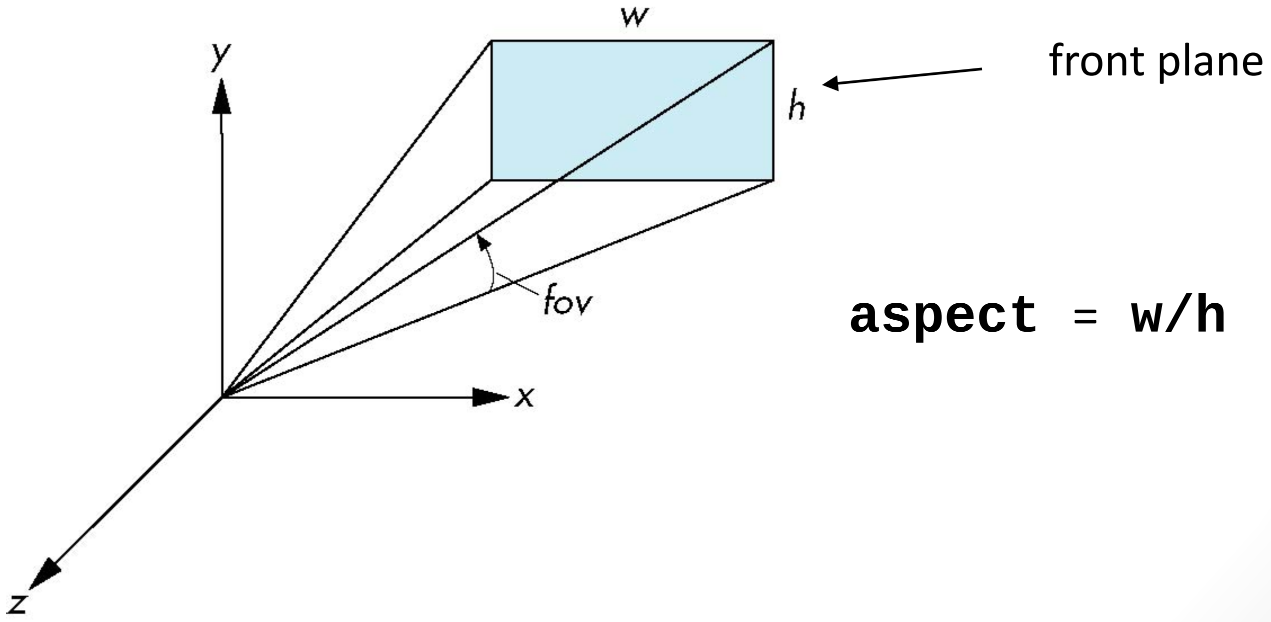
glFrustum(Xmin,Xmax,Ymin,Ymax,near,far)



glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(xmin,xmax,ymin,ymax,near,far)

Using Field of View @ Y-axis

- With **`glFrustum`**, it is often difficult to get the desired view
- **`gluPerpective(fovy, aspect, near, far)`** provides a better interface

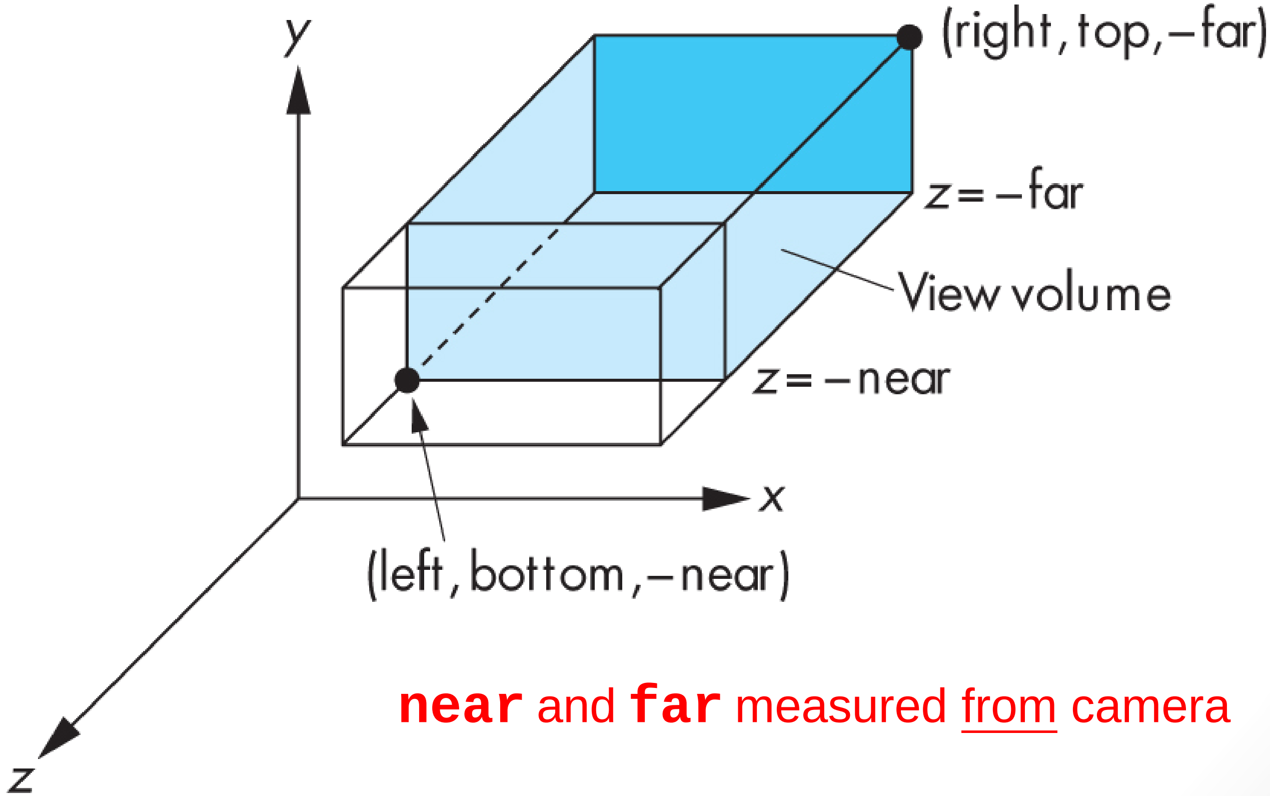


`aspect = w/h`

Orthographic Viewing in OpenGL

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

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



`near` and **`far`** measured <u>from</u> camera

Hidden-Surface Removal

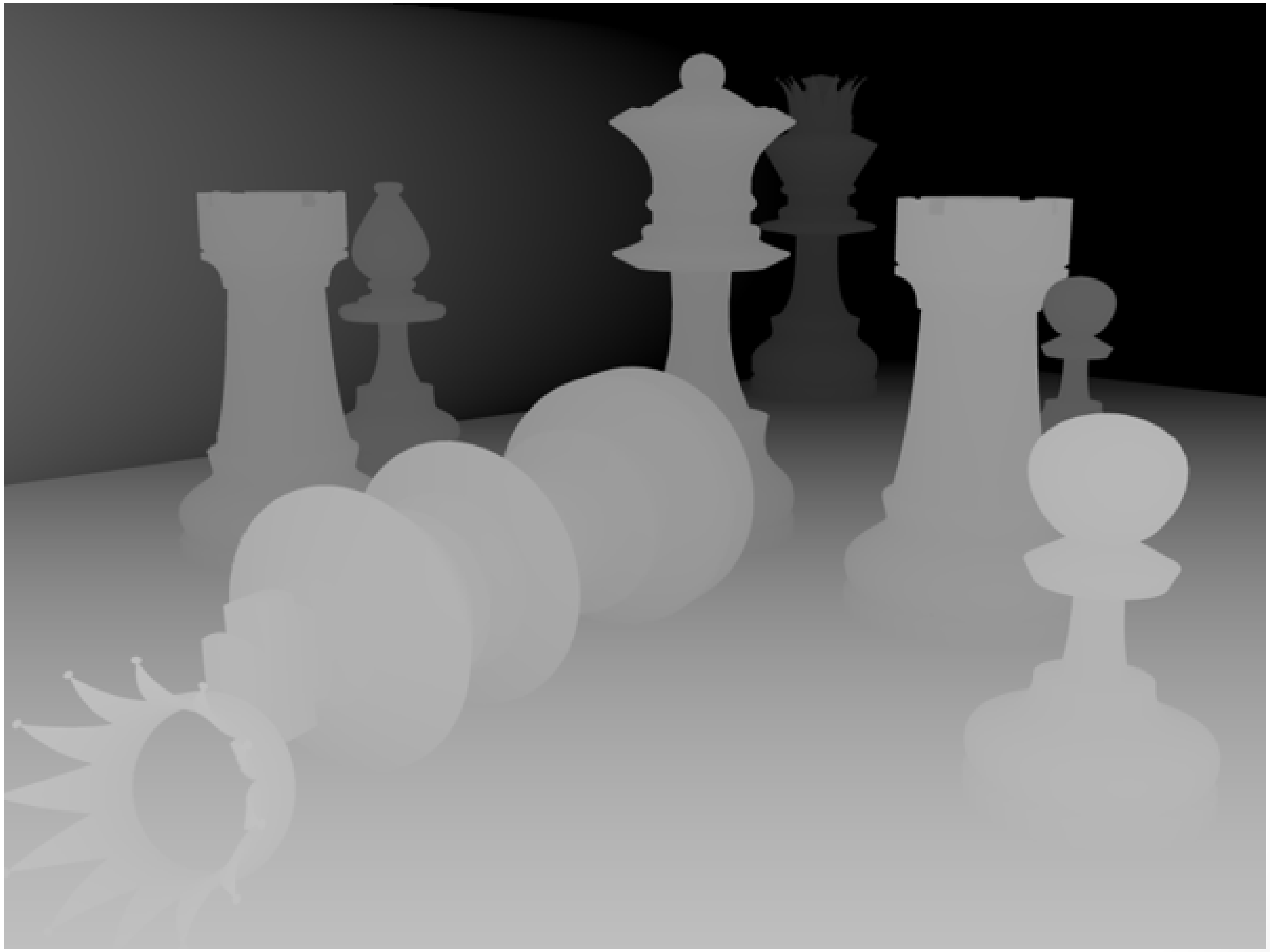
- Find which surfaces are visible
- Two classes
  - Object-space algorithm
    - Attempt to order the surfaces of the objects in the scene such that drawing surfaces in a particular order provides the correct image.
  - Image-space algorithm
    - Seek to determine the relationship among object points on each projector.

Z-buffer Algorithm

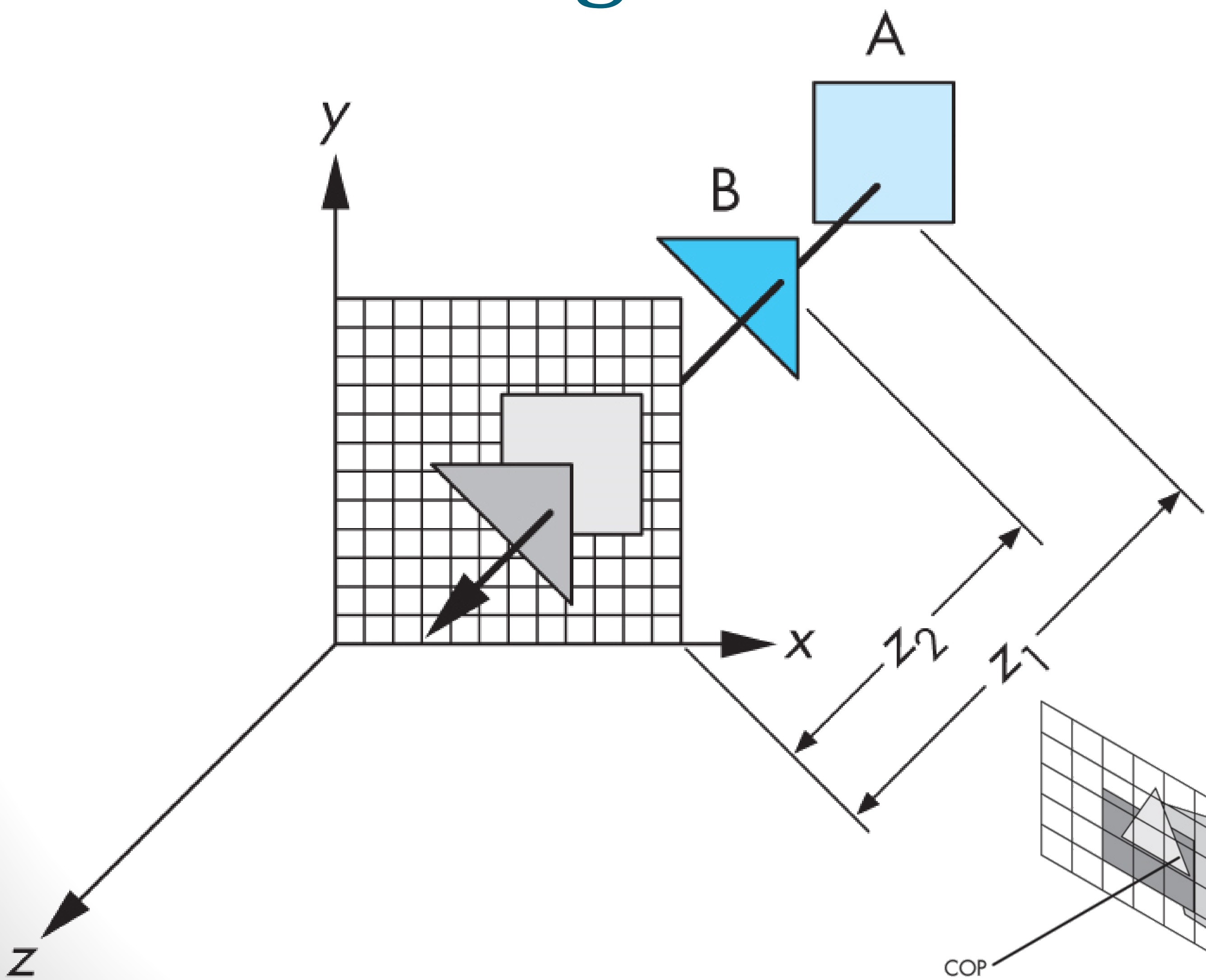
- Keep track of the distance from COP to the closest point on each projector, then we can update this information as successive polygons are projected and filled.
- A z-buffer to store the necessary depth information as polygons are rasterized.

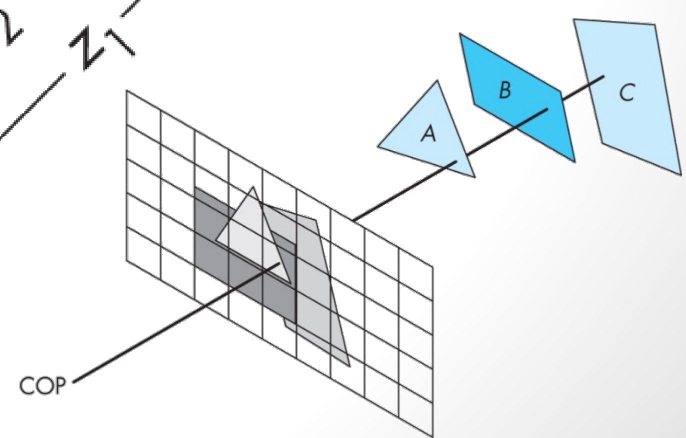
information

Visualization of the depth map

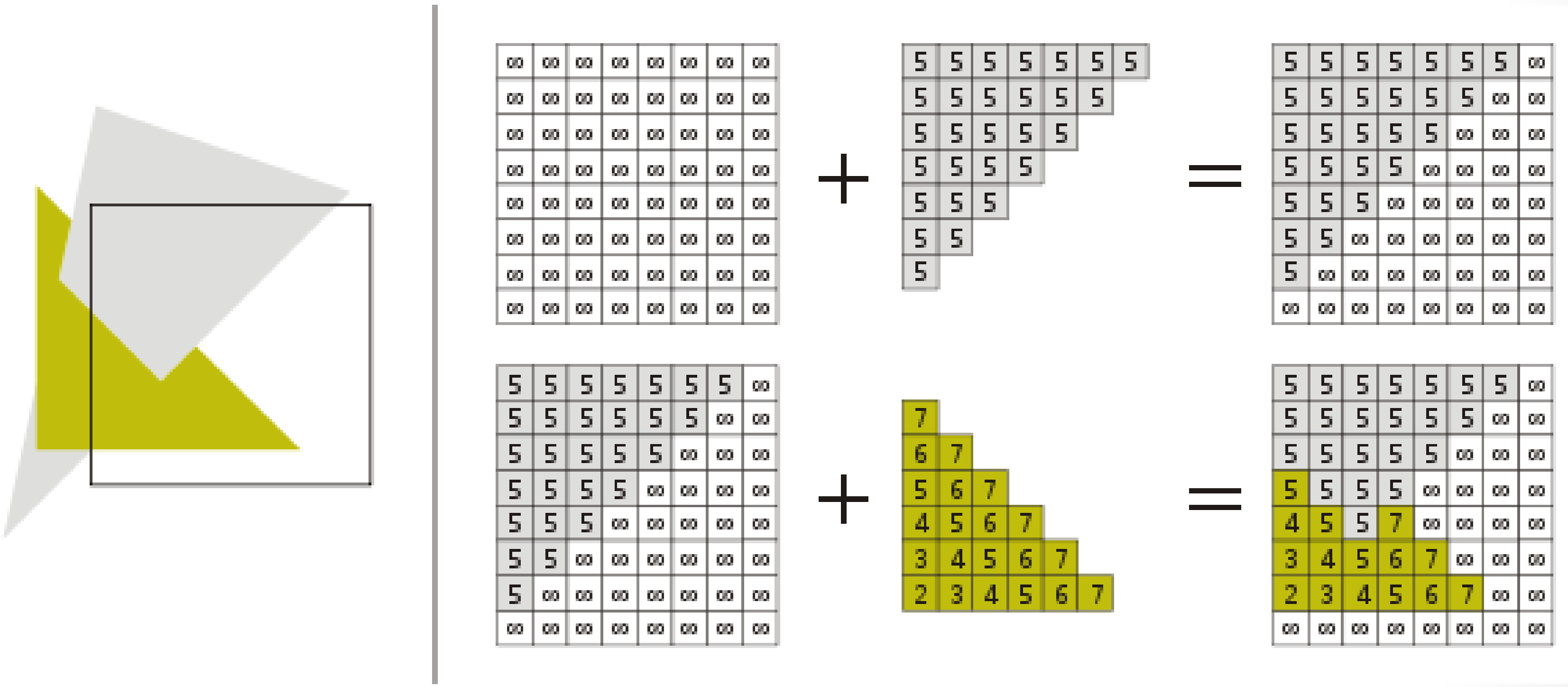


Z-buffer Hidden-Surface Removal Algorithm





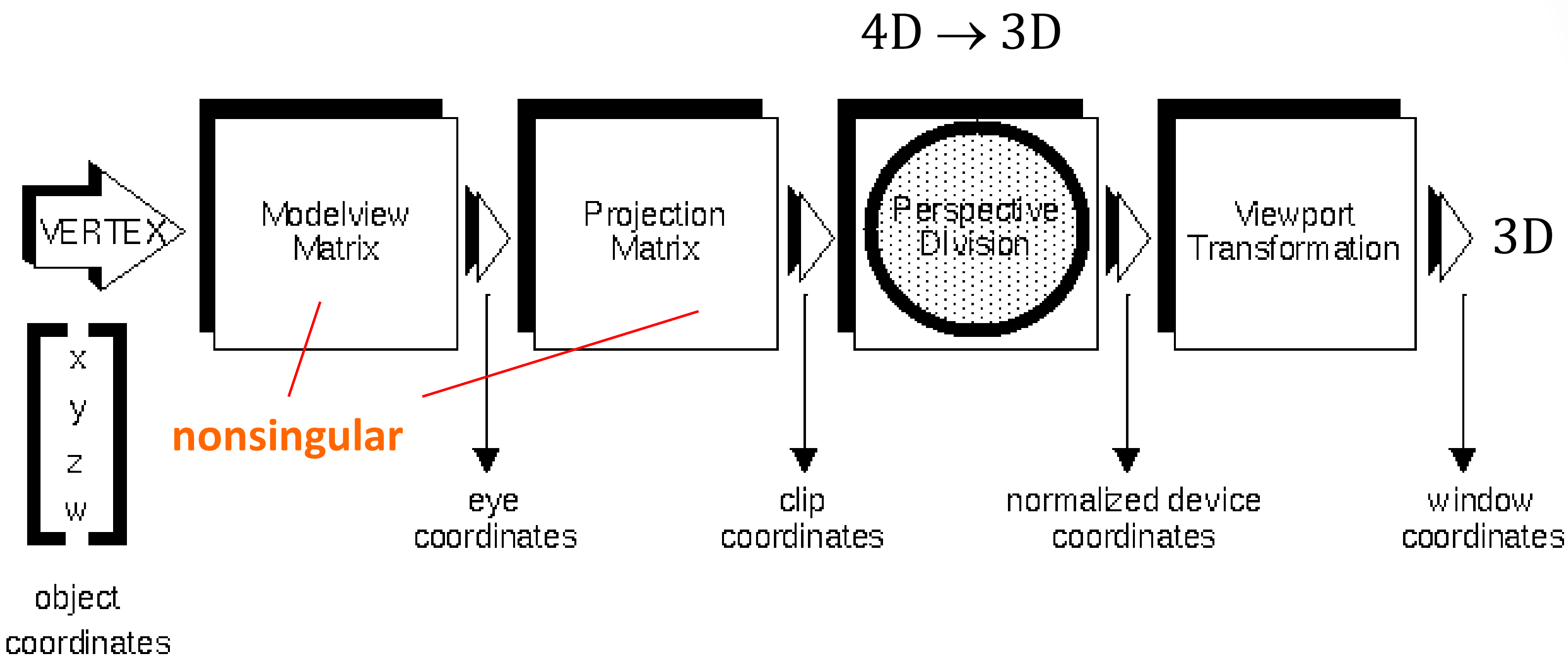
Z-buffer Hidden-Surface Removal Algorithm



OpenGL Z-buffer

- glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  - Setup the buffer before create the window
- glEnable(GL_DEPTH_TEST);
  - Enable depth test
- glClear(GL_DEPTH_BUFFER_BIT)
  - Clean the buffer before use
  - Default value = 1.0 (infinite)

Pipeline View



nonsingular transformation
A linear transformation which has an inverse; equivalently, it has null space kernel consisting only of the zero vector.

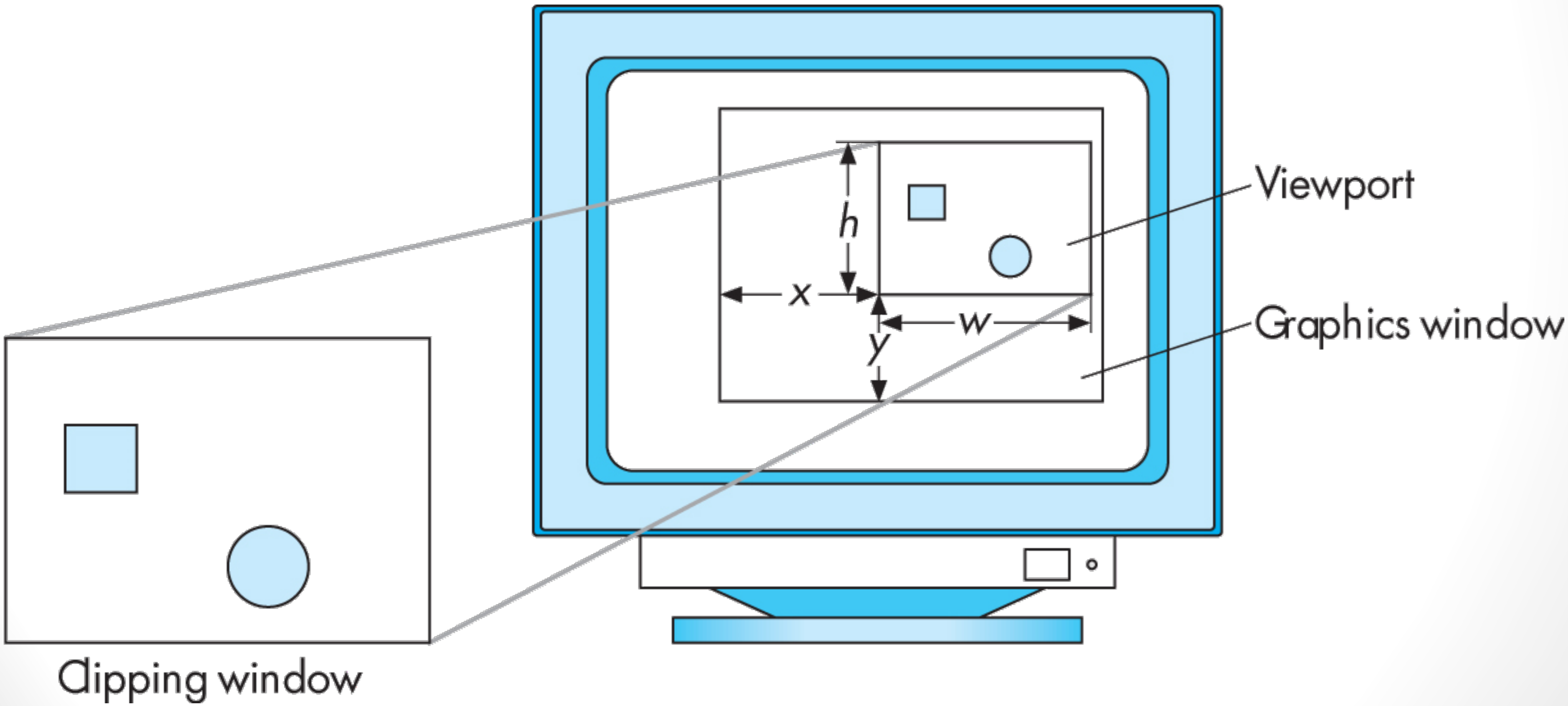
"A square matrix that is not invertible is called **singular** or **degenerate**"

Notes

- We stay in four-dimensional homogeneous coordinates through both the **ModelView** and **Projection** transformations
  - Both these transformations are nonsingular
  - Default to identity matrices (= orthogonal view)
- Normalization lets us clip against simple cube regardless of type of projection
- Window coordinates are still 3D
  - Important for hidden-surface removal to retain depth information as long as possible

Viewport Transformation

- glViewport(x, y, w, h);



Display callback

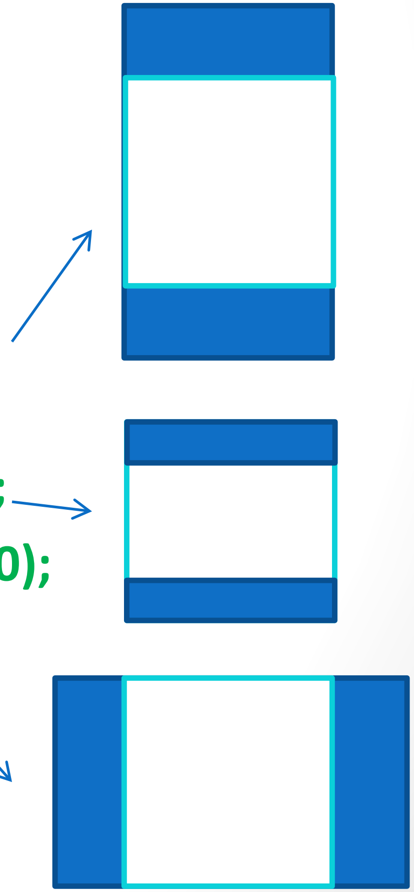
```
void display(void)
{
 glclear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFER_BIT);
 glLoadIdentity();
 gluLookAt(viewer[0],viewer[1], viewer[2], 0.0,0.0,0.0, 0.0,1.0,0.0);

 glRotate(theta[0], 1.0, 0.0, 0.0);
 glRotate(theta[1], 0.0, 1.0, 0.0);
 glRotate(theta[2], 0.0, 0.0, 1.0);

 colorcube();
 glFlush();
 glSwapBuffers();
}
```

Reshape callback

```
void myReshape(int w, int h)
{
  glViewport(0,0,w,h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if(w<=h) {
    glFrustum(-2.0,2.0, -2.0*h/w, 2.0*h/w, 2.0, 20.0);
  }  else {
    glFrustum(-2.0,2.0, -2.0*w/h, 2.0*w/h, 2.0, 20.0);
    //glFrustum(-2.0*w/h,2.0*w/h, -2.0, 2.0, 2.0, 20.0);
  }
  glMatrixMode(GL_MODELVIEW);
}
```



Parallel Normalization

- Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthographic projections with the default view volume
- This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping

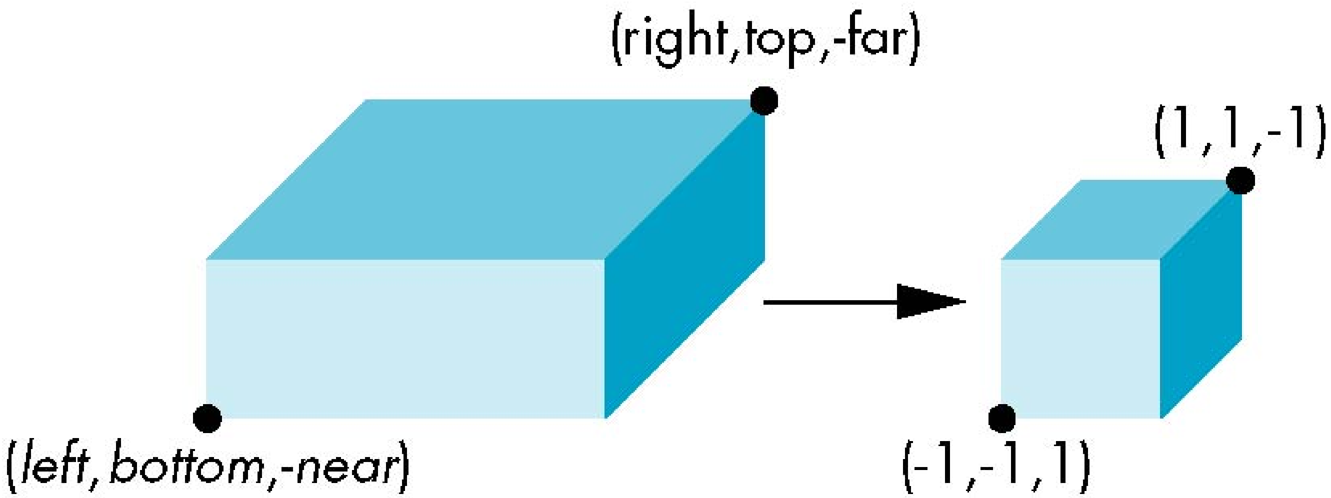
Why do we do it this way?

- Normalization allows for a single pipeline for both perspective and orthogonal viewing
- We keep in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
- The clipping process has been simplified.

*Normalization

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

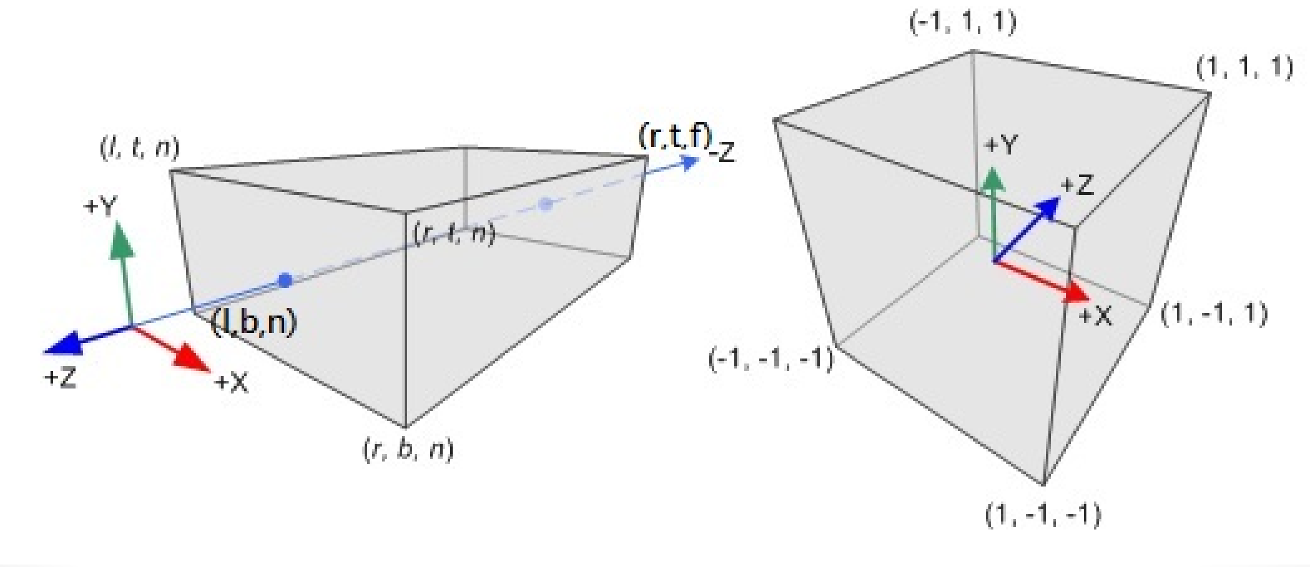
normalization $\Rightarrow$ find transformation to convert specified clipping volume to default



Orthographic Projection

- Map orthographic view volume to the canonical view volume (Axis-aligned box)

- (l,b,n) = (left, bottom, near) , (r,t,f) = (right, top, far)
  - [l, r] x [b, t] x [n,f] → [-1, 1]x[-1, 1]x[-1, 1]
    - n < 0, f < 0, Be care of the difference with OpenGL/DirectX SPEC



Orthographic Matrix

- Two steps
  - Move center to origin
    - $T(-(\text{left} + \text{right})/2, -(\text{bottom} + \text{top})/2, (\text{near} + \text{far})/2))$
  - Scale to have sides of length 2
    - $S\left(2/(\text{righ} - \text{left}), 2/(\text{top} - \text{bottom}), -2/(\text{far} - \text{near})\right)$

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

Final Projection

- Set $z = 0$
- Equivalent to the homogeneous coordinate transformation

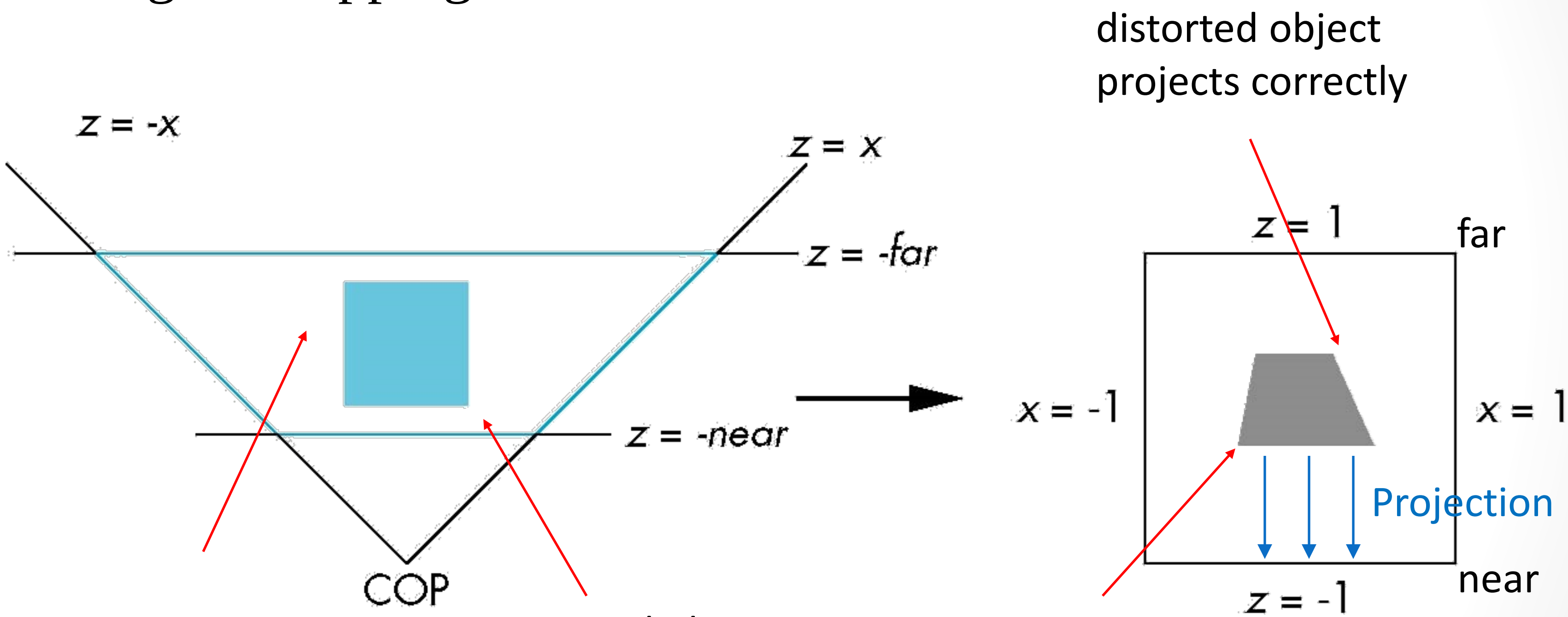
$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Hence, general orthogonal projection in 4D is

$$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{ST}$$

Normalization Transformation (in the Perspective Projection)

original clipping volume



new clipping volume

SUGGESTION!

OR

OBJECTION?

Let's stop here,

TAKE A BREAK