



Other Camera Controls

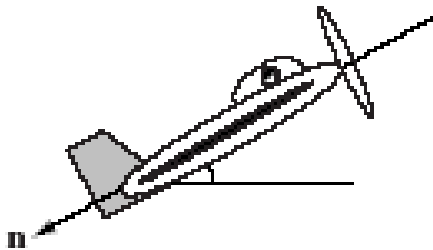
- The LookAt function is only for positioning camera
- Other ways to specify camera position/movement
 - Yaw, pitch, roll
 - Elevation, azimuth, twist
 - Direction angles



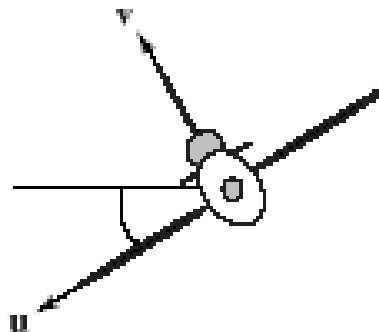
Flexible Camera Control

- Sometimes, we want camera to move
- Like controlling an airplane's orientation
- Adopt aviation terms:
 - **Pitch:** nose up-down
 - **Roll:** roll body of plane
 - **Yaw:** move nose side to side

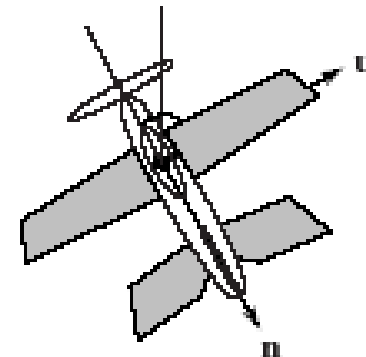
a) pitch



b) roll



c) yaw

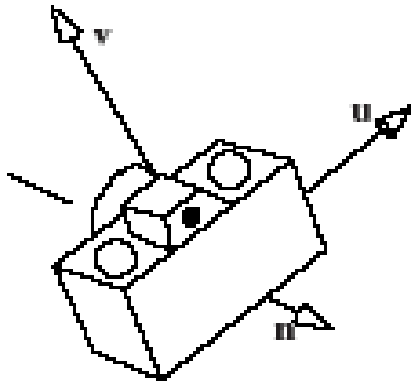




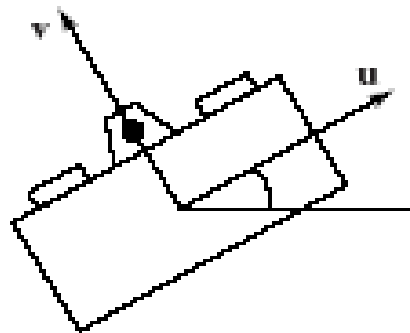
Yaw, Pitch and Roll Applied to Camera

- Similarly, yaw, pitch, roll with a camera

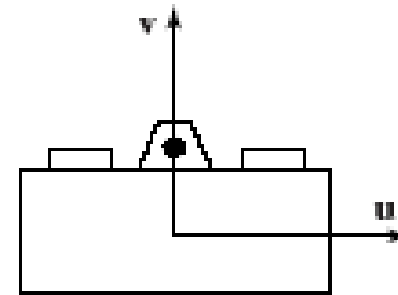
a) camera orientation



b) with roll



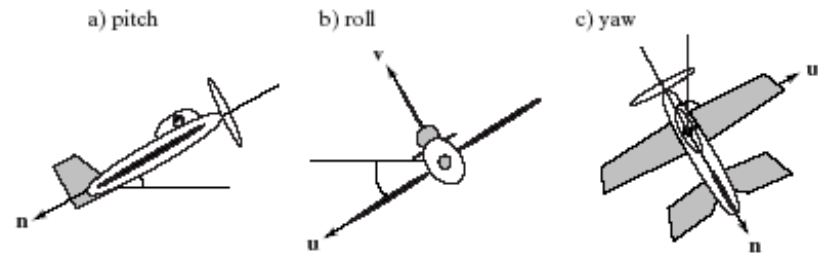
c) no roll



Flexible Camera Control

- Create a camera class

```
class Camera
  private:
    Point3 eye;
    Vector3 u, v, n;... etc
```



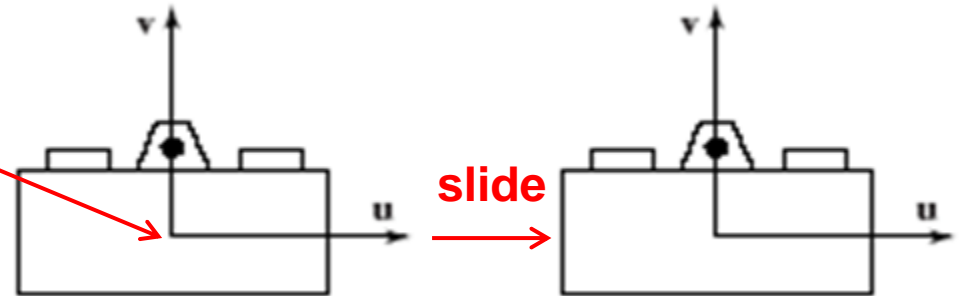
- Camera functions to specify pitch, roll, yaw. E.g

```
cam.slide(-1, 0, -2); // slide camera forward -2 and left -1
cam.roll(30); // roll camera 30 degrees
cam.yaw(40); // yaw it 40 degrees
cam.pitch(20); // pitch it 20 degrees
```



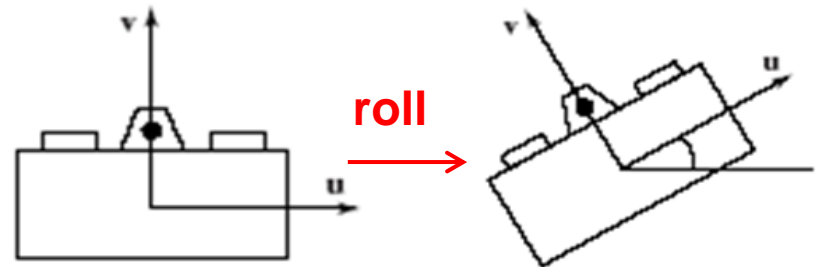
Recall: Final LookAt Matrix

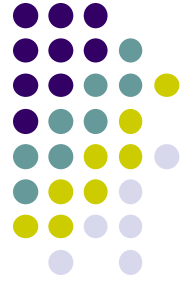
- Slide along u, v or n
- Changes eye position
- Changes these components



$$\begin{array}{|c|} \hline \begin{array}{ccc} ux & uy & uz \\ vx & vy & vz \\ nx & ny & nz \\ 0 & 0 & 0 \end{array} \\ \hline \end{array}
 \begin{array}{|c|} \hline \begin{array}{c} -\mathbf{e} \cdot \mathbf{u} \\ -\mathbf{e} \cdot \mathbf{v} \\ -\mathbf{e} \cdot \mathbf{n} \\ 1 \end{array} \\ \hline \end{array}$$

- Pitch, yaw, roll rotates u, v or n
- Changes u, v or n





Implementing Flexible Camera Control

- Camera class: maintains current (u,v,n) and eye position

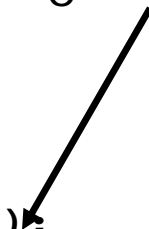
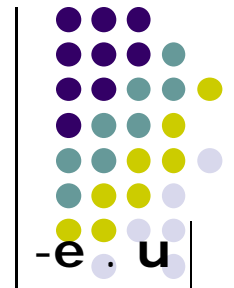
```
class Camera
private:
    Point3 eye;
    Vector3 u, v, n;... etc
```

- User inputs desired roll, pitch, yaw angle or slide
 1. Roll, pitch, yaw: calculate modified vector (u' , v' , n')
 2. Slide: Calculate new eye position
 3. Update lookAt matrix, Load it into CTM

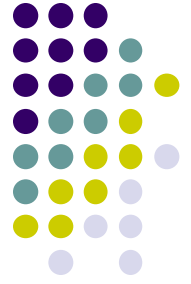
Load Matrix into CTM

```
void Camera::setModelViewMatrix(void)
{ // load modelview matrix with camera values
  mat4 m;
  Vector3 eVec(eye.x, eye.y, eye.z); // eye as vector
  m[0] = u.x; m[4] = u.y; m[8] = u.z; m[12] = -dot(eVec,u);
  m[1] = v.x; m[5] = v.y; m[9] = v.z; m[13] = -dot(eVec,v);
  m[2] = n.x; m[6] = n.y; m[10] = n.z; m[14] = -dot(eVec,n);
  m[3] = 0; m[7] = 0; m[11] = 0; m[15] = 1.0;
  CTM = m; // Finally, load matrix m into CTM Matrix
}
```

$$\begin{array}{ccc|c} ux & uy & uz & -e \cdot u \\ vx & vy & vz & -e \cdot v \\ nx & ny & nz & -e \cdot n \\ 0 & 0 & 0 & 1 \end{array}$$



- Slide changes **eVec**,
- roll, pitch, yaw, change **u, v, n**
- Call setModelViewMatrix after slide, roll, pitch or yaw



Example: Camera Slide

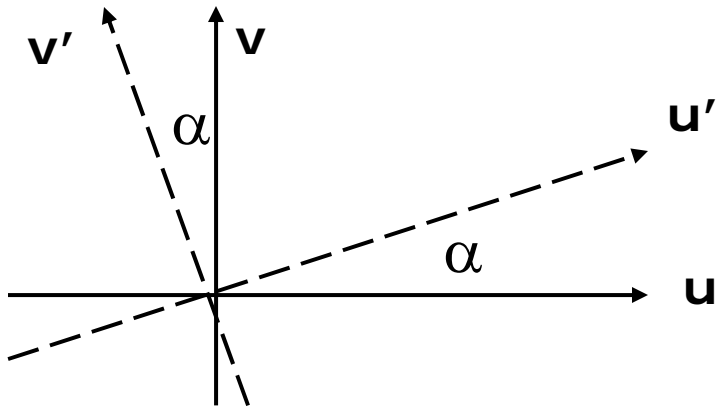
- User changes eye by delU , delV or delN
- $\text{eye} = \text{eye} + \text{changes}(\text{delU}, \text{delV}, \text{delN})$
- Note: function below combines all slides into one

E.g moving camera by D along its u axis = $\text{eye} + D\mathbf{u}$

```
void camera::slide(float delU, float delV, float delN)
{
    eye.x += delU*u.x + delV*v.x + delN*n.x;
    eye.y += delU*u.y + delV*v.y + delN*n.y;
    eye.z += delU*u.z + delV*v.z + delN*n.z;
    setModelViewMatrix( );
}
```




Example: Camera Roll



$$\mathbf{u}' = \cos(\alpha)\mathbf{u} + \sin(\alpha)\mathbf{v}$$

$$\mathbf{v}' = -\sin(\alpha)\mathbf{u} + \cos(\alpha)\mathbf{v}$$

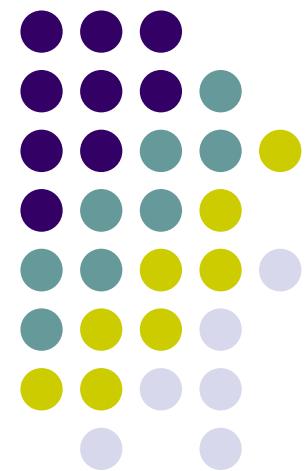
```
void Camera::roll(float angle)
{ // roll the camera through angle degrees
  float cs = cos(3.142/180 * angle);
  float sn = sin(3.142/180 * angle);
  Vector3 t = u; // remember old u
  u.set(cs*t.x - sn*v.x, cs*t.y - sn*v.y, cs*t.z - sn*v.z);
  v.set(sn*t.x + cs*v.x, sn*t.y + cs*v.y, sn*t.z + cs*v.z)
  setModelViewMatrix( );
}
```

Computer Graphics (CS 543)

Lecture 5 (part 1): Projection (Part I)

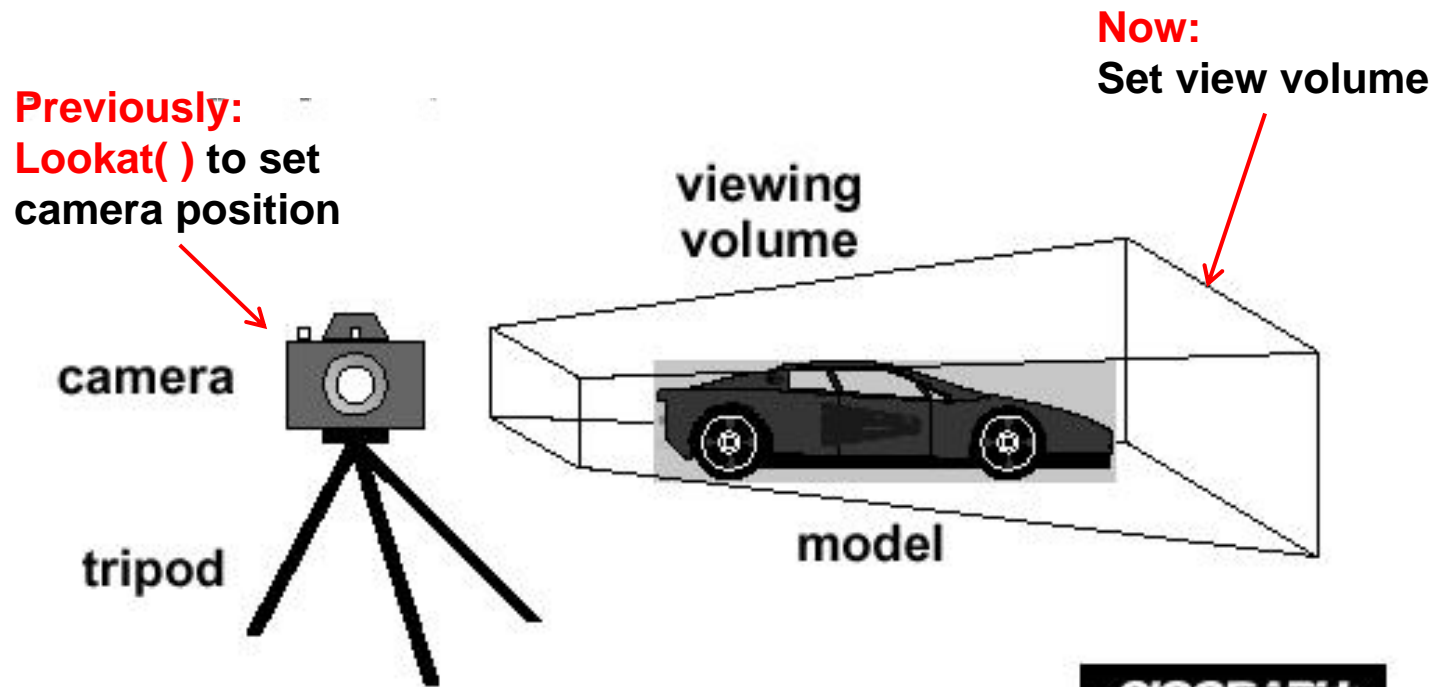
Prof Emmanuel Agu

*Computer Science Dept.
Worcester Polytechnic Institute (WPI)*

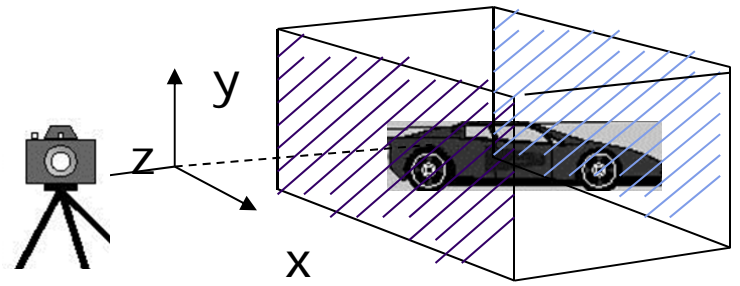




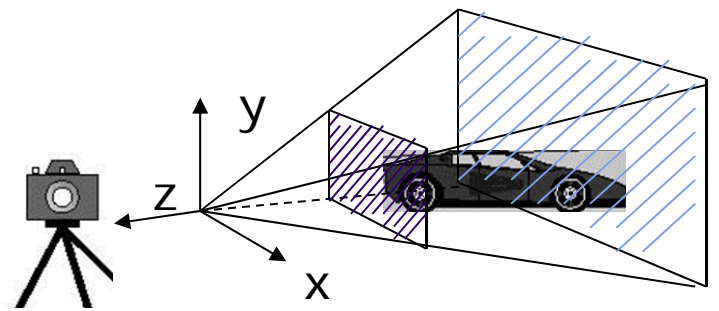
Recall: 3D Viewing and View Volume



Recall: Different View Volume Shapes



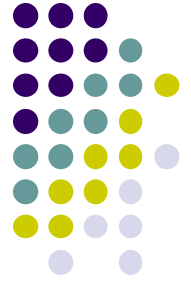
Orthogonal view volume
(no foreshortening)



Perspective view volume
(exhibits foreshortening)

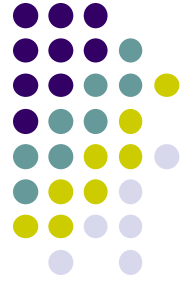


- Different view volume => different look
- **Foreshortening?** Near objects bigger



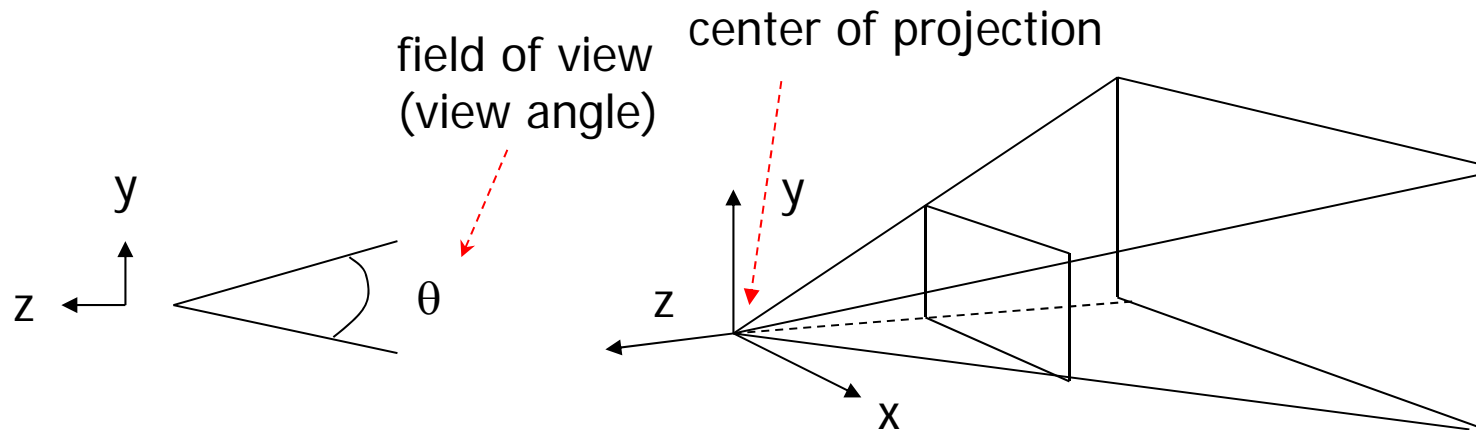
View Volume Parameters

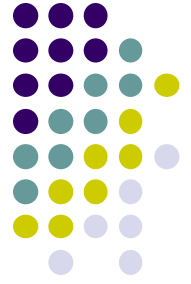
- Need to set view volume parameters
 - **Projection type:** perspective, orthographic, etc.
 - Field of view and aspect ratio
 - Near and far clipping planes



Field of View

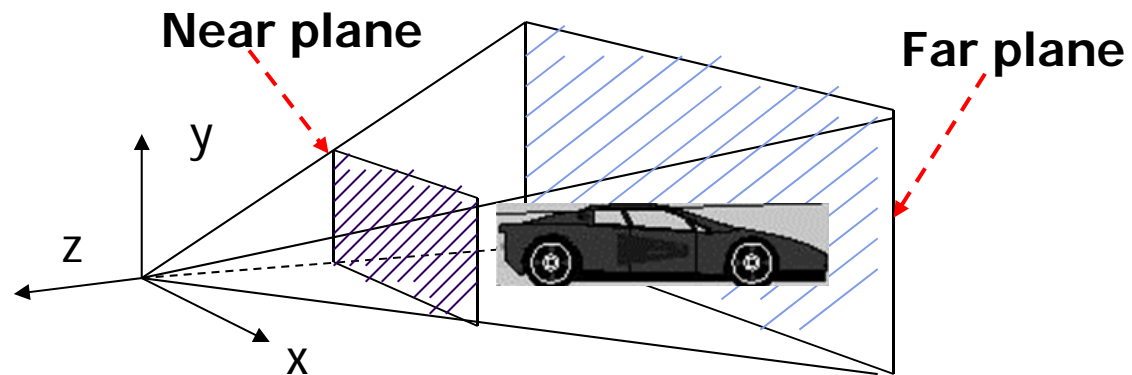
- View volume parameter
- Determines how much of world in picture (vertically)
- Larger field of view = smaller objects drawn





Near and Far Clipping Planes

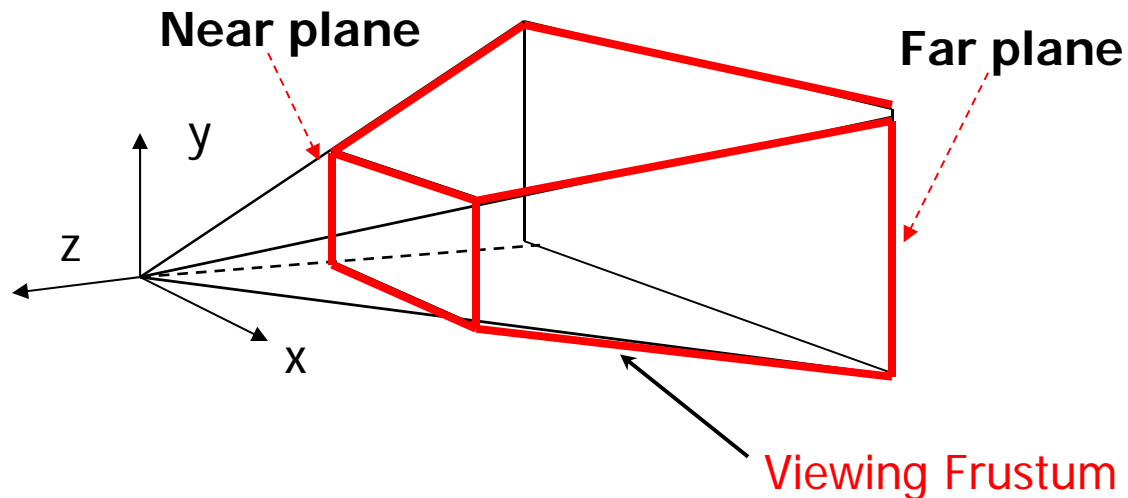
- Only objects between near and far planes drawn





Viewing Frustum

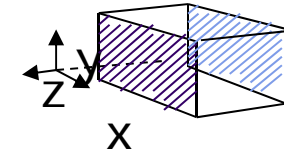
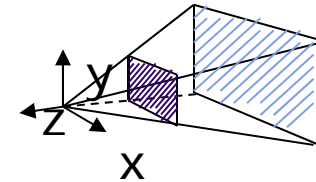
- Near plane + far plane + field of view = **Viewing Frustum**
- Objects outside the frustum are clipped





Setting up View Volume/Projection Type

- Previous OpenGL projection commands **deprecated!!**
- Perspective view volume/projection:
 - `gluPerspective(fovy, aspect, near, far)` or
 - `glFrustum(left, right, bottom, top, near, far)`
- Orthographic:
 - `glOrtho(left, right, bottom, top, near, far)`
- Useful functions, so we implement similar in `mat.h`:
 - `Perspective(fovy, aspect, near, far)` or
 - `Frustum(left, right, bottom, top, near, far)`
 - `Ortho(left, right, bottom, top, near, far)`

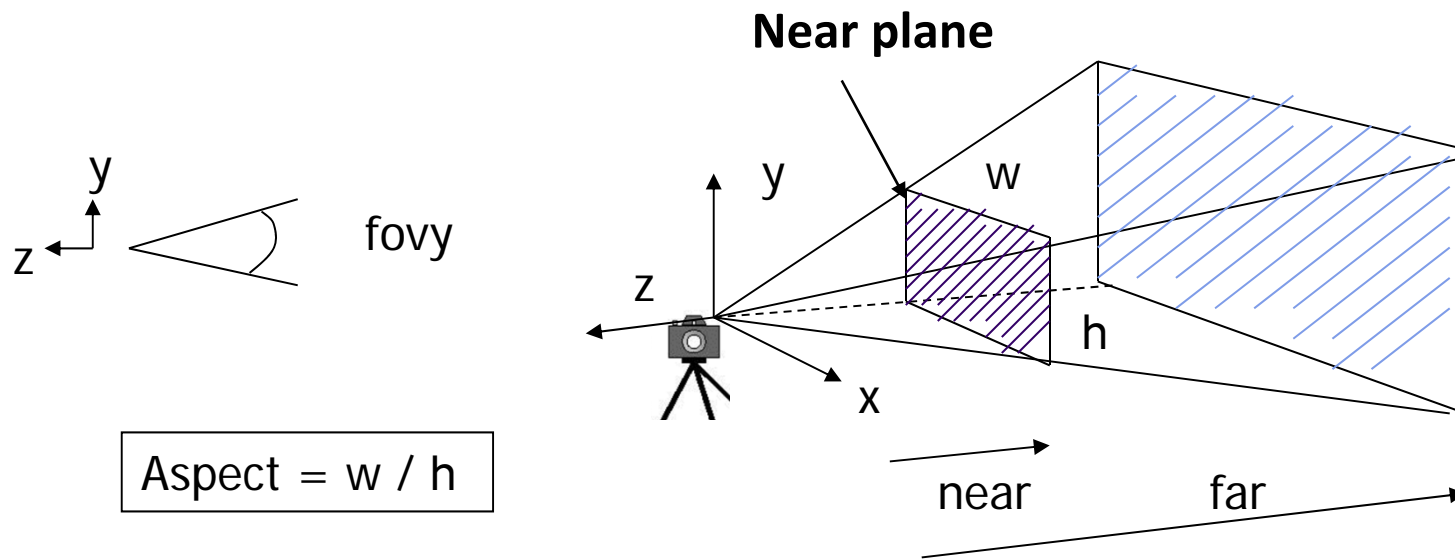


What are these arguments? Next!



Perspective(fovy, aspect, near, far)

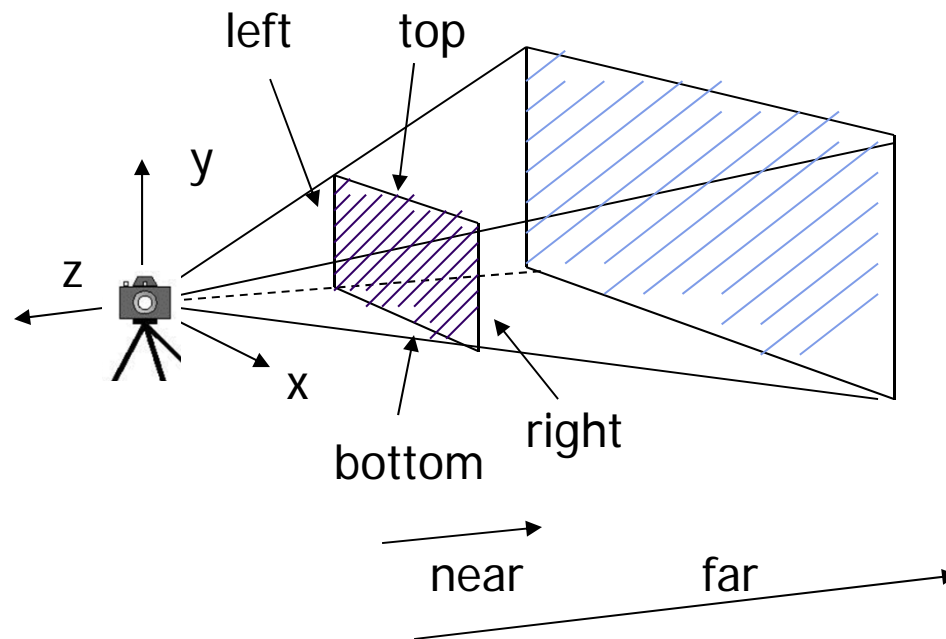
- Aspect ratio used to calculate window width





Frustum(left, right, bottom, top, near, far)

- Can use **Frustum()** in place of **Perspective()**
- Same view volume shape, different **arguments**

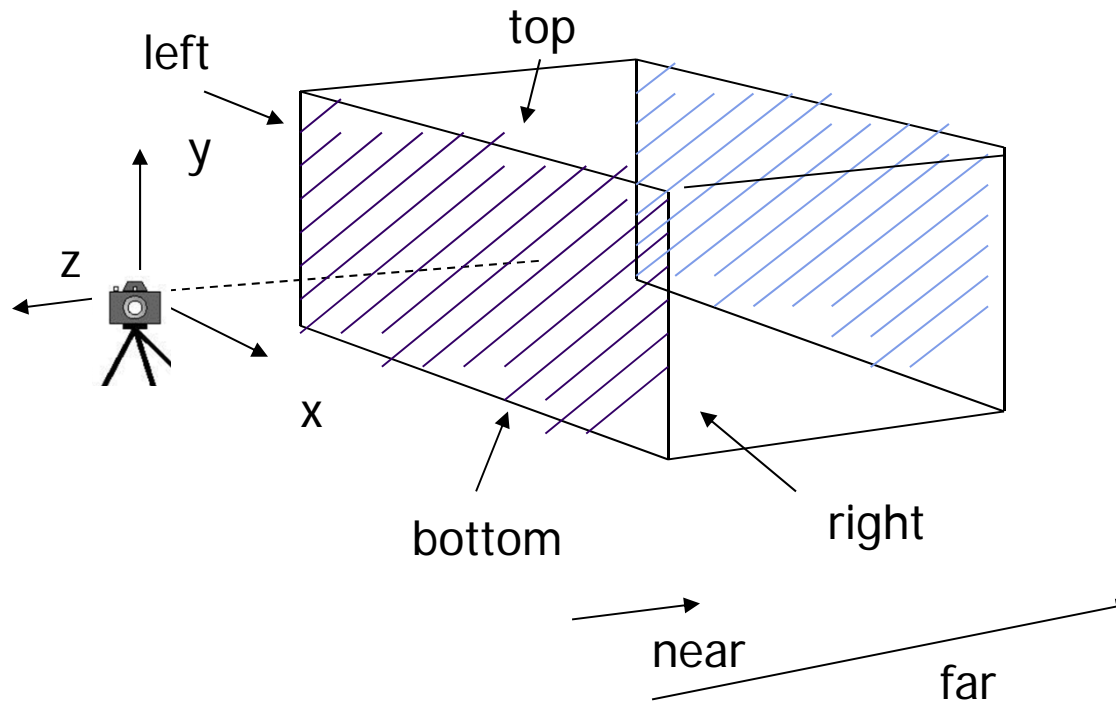


near and **far** measured from **camera**



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

- For orthographic projection



near and **far** measured from **camera**

Example Usage: Setting View Volume/Projection Type



```
void display()
{
    // clear screen
    glClear(GL_COLOR_BUFFER_BIT);
    .....
    // Set up camera position
    LookAt(0,0,1,0,0,0,0,1,0);

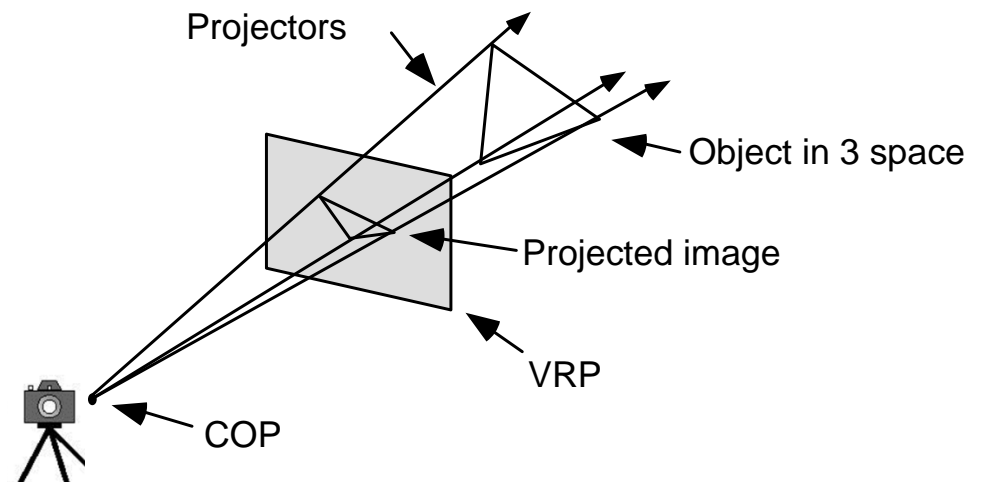
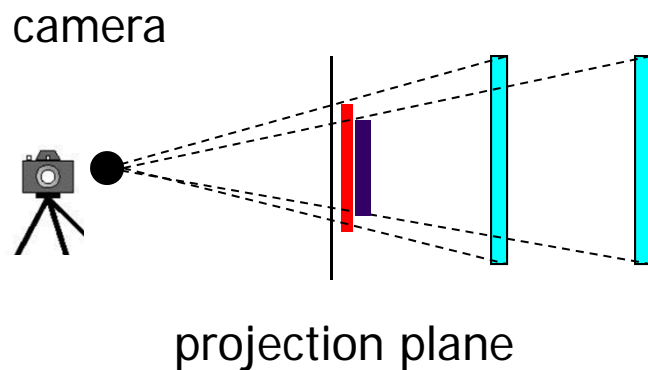
    .....
    // set up perspective transformation
    Perspective(fovy, aspect, near, far);

    .....
    // draw something
    display_all();    // your display routine
}
```



Perspective Projection

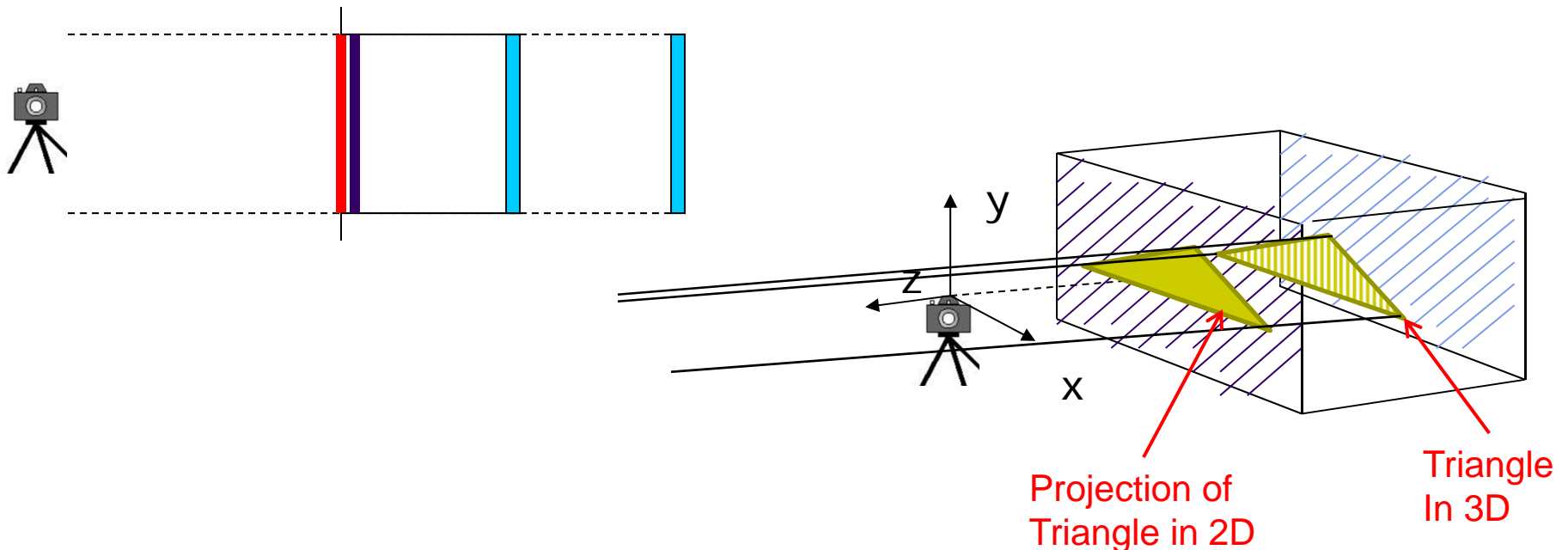
- After setting view volume, then projection transformation
- Projection?
 - **Classic:** Converts 3D object to corresponding 2D on screen
 - How? Draw line from object to projection center
 - Calculate where each cuts projection plane





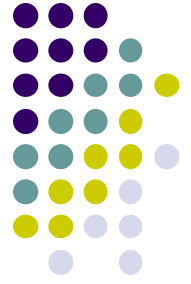
Orthographic Projection

- How? Draw parallel lines from each object vertex
- The projection center is at infinite
- In short, use (x,y) coordinates, just drop z coordinates



Demo

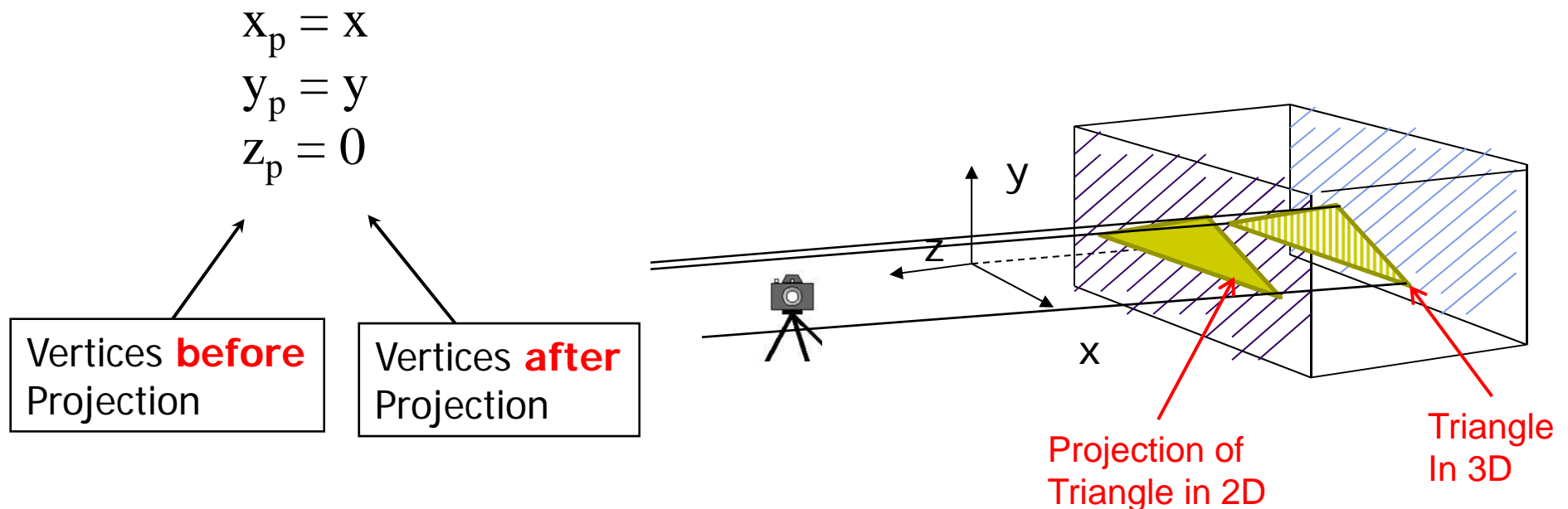
- Nate Robbins demo on projection





Default View Volume/Projection?

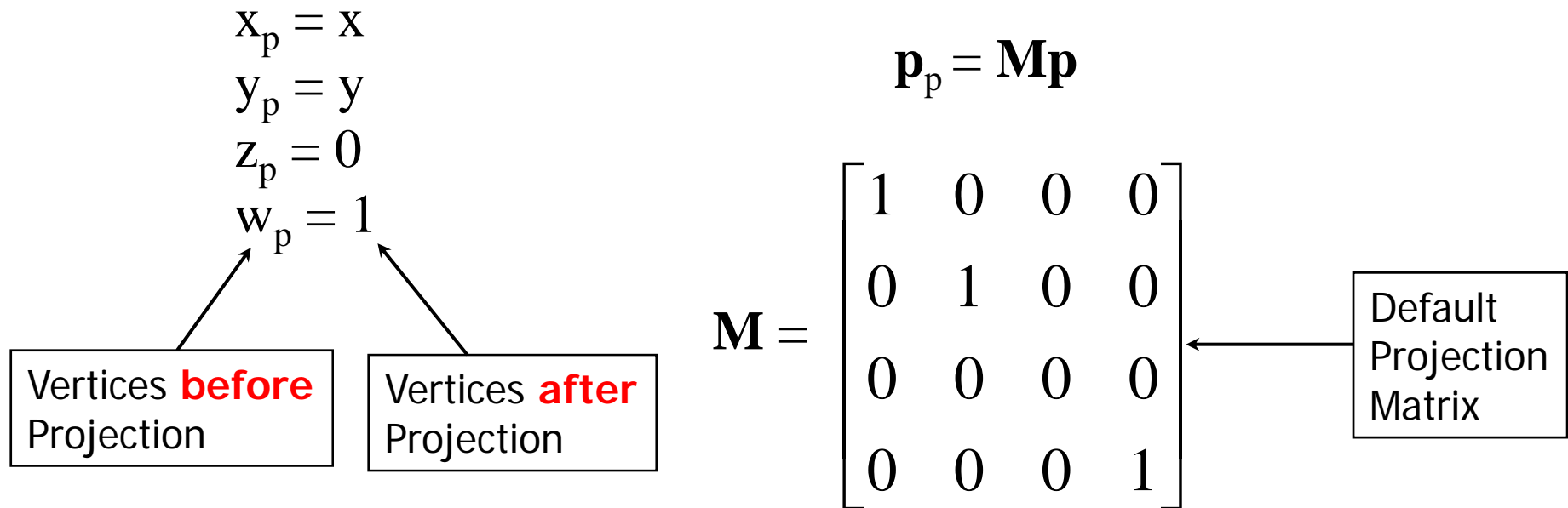
- What if you user does not set up projection?
- Default OpenGL projection is orthogonal (Ortho());
- To project points within default view volume



Homogeneous Coordinate Representation



default orthographic projection

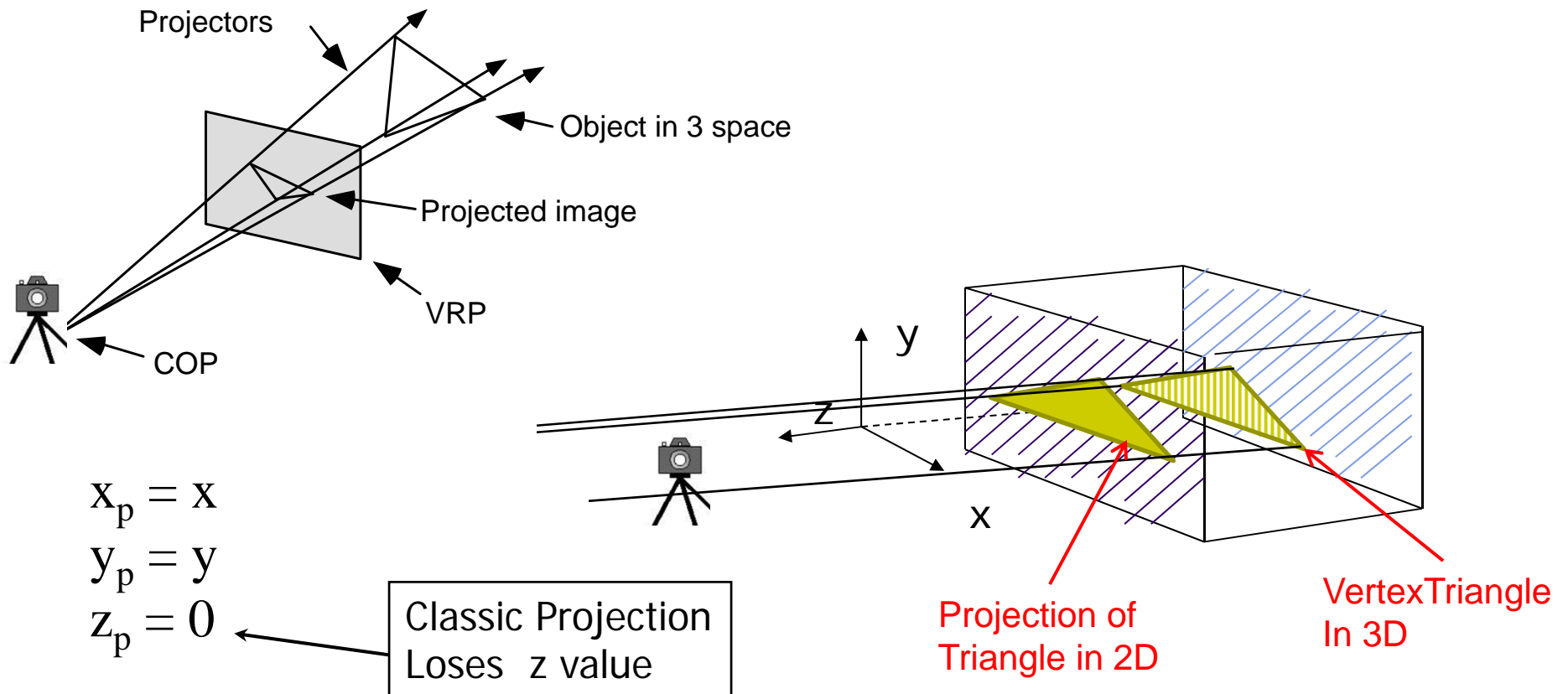


In practice, can let $\mathbf{M} = \mathbf{I}$, set the z term to zero later



The Problem with Classic Projection

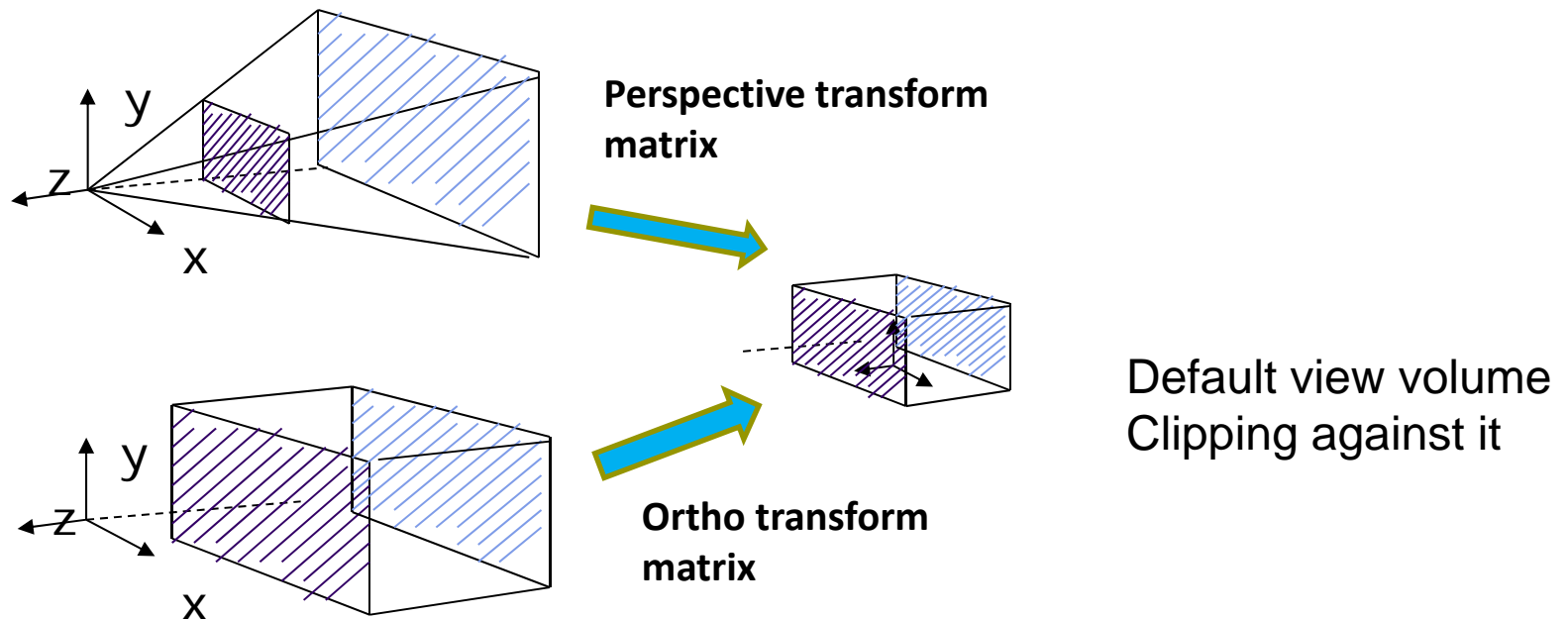
- Keeps (x,y) coordinates for drawing, drops z
- We may need z. Why?





Normalization: Keeps z Value

- Most graphics systems use *view normalization*
- **Normalization:** convert all other projection types to orthogonal projections with the *default view volume*



References



- Interactive Computer Graphics (6th edition), Angel and Shreiner
- Computer Graphics using OpenGL (3rd edition), Hill and Kelley