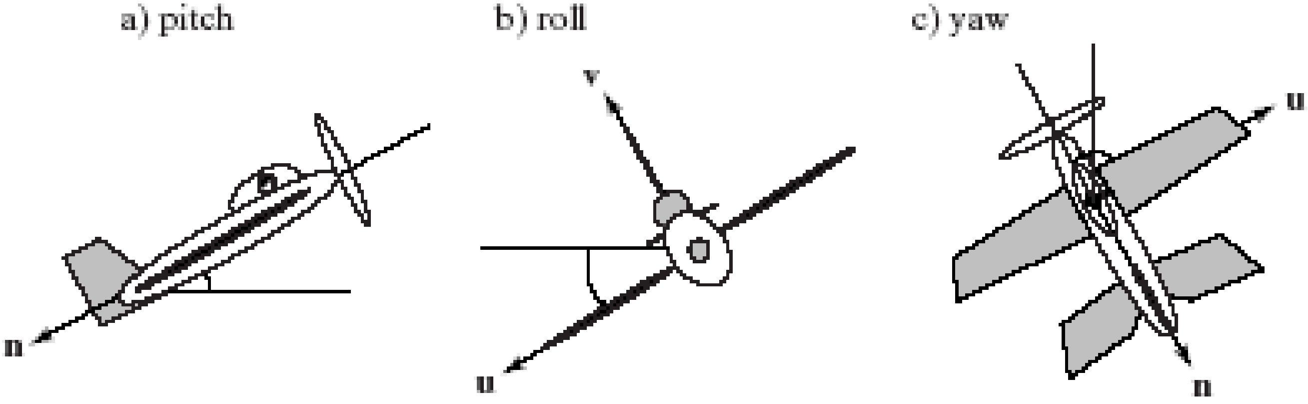
Other Camera Controls

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

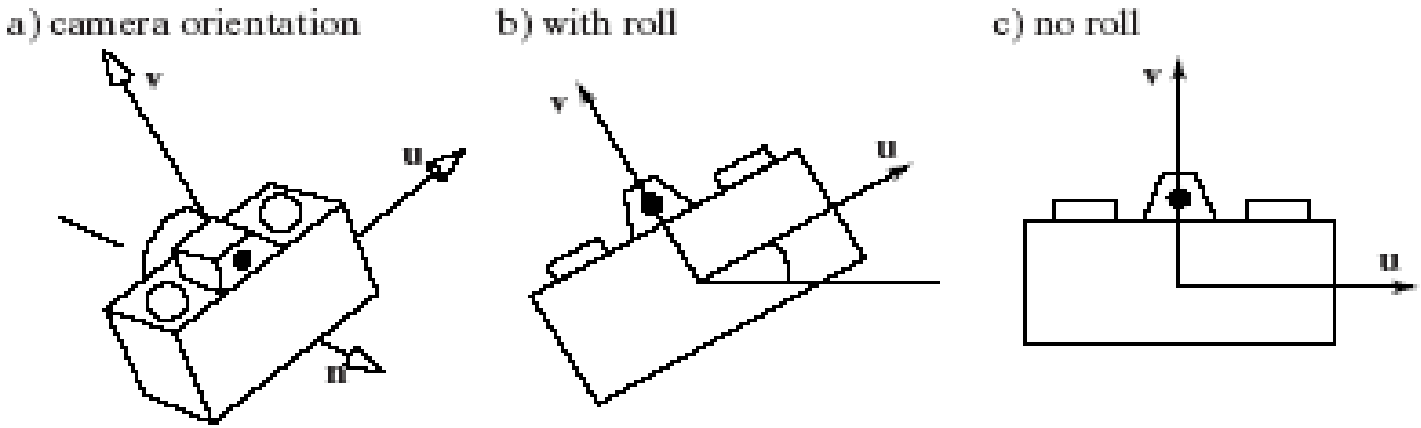
Flexible Camera Control

- 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



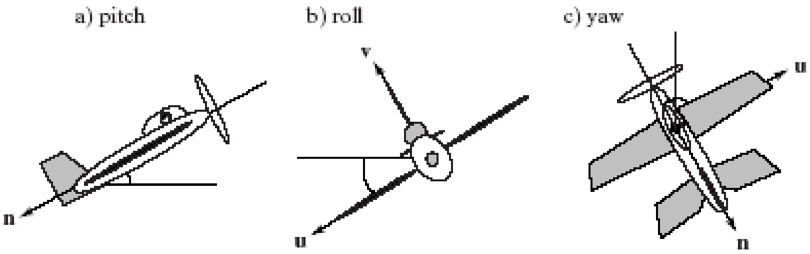
Yaw, Pitch and Roll Applied to Camera

- Similarly, yaw, pitch, roll with a camera



Flexible Camera Control

- Create a **camera** class



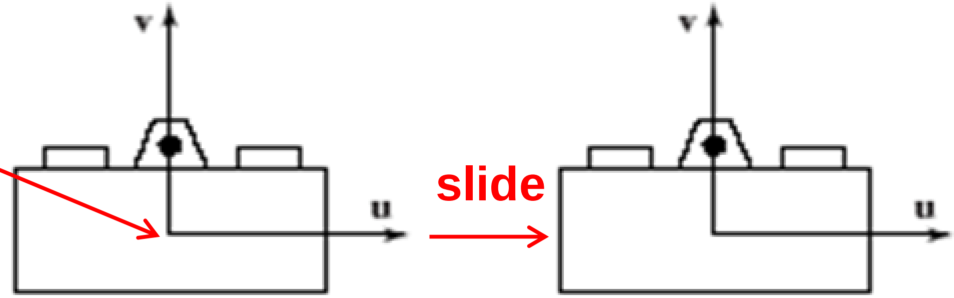

```
class Camera
   private:
        Point3 eye;
        Vector3 u, v, n;…. etc
```

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

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

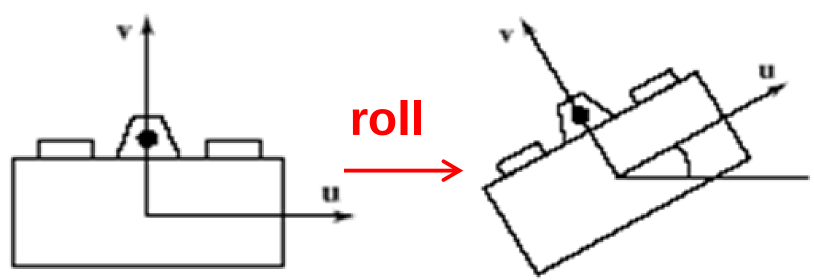
Recall: Final LookAt Matrix

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



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

- 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

Example: Camera Slide

- Recall: the axes are unit vectors
- User changes eye by delU, delV or delN
- eye = eye + changes (delU, delV, delN)
- Note: function below combines all slides into one

  E.g moving camera by ***D*** along its u axis = **eye + *D*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( );
}
```

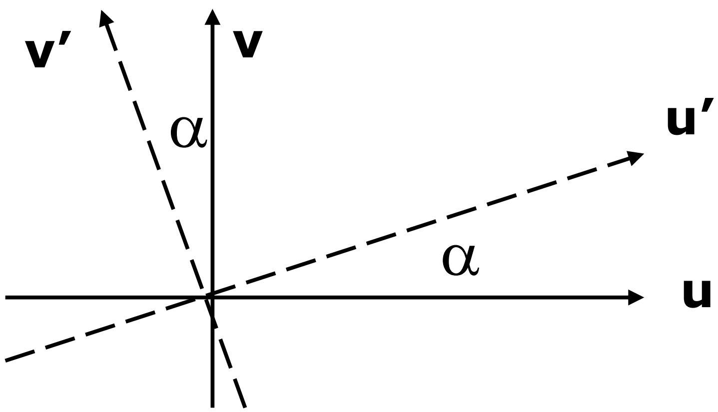
Load Matrix into CTM

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

```
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
}
```

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

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 4731)
Lecture 14: Projection (Part I)

Prof Emmanuel Agu

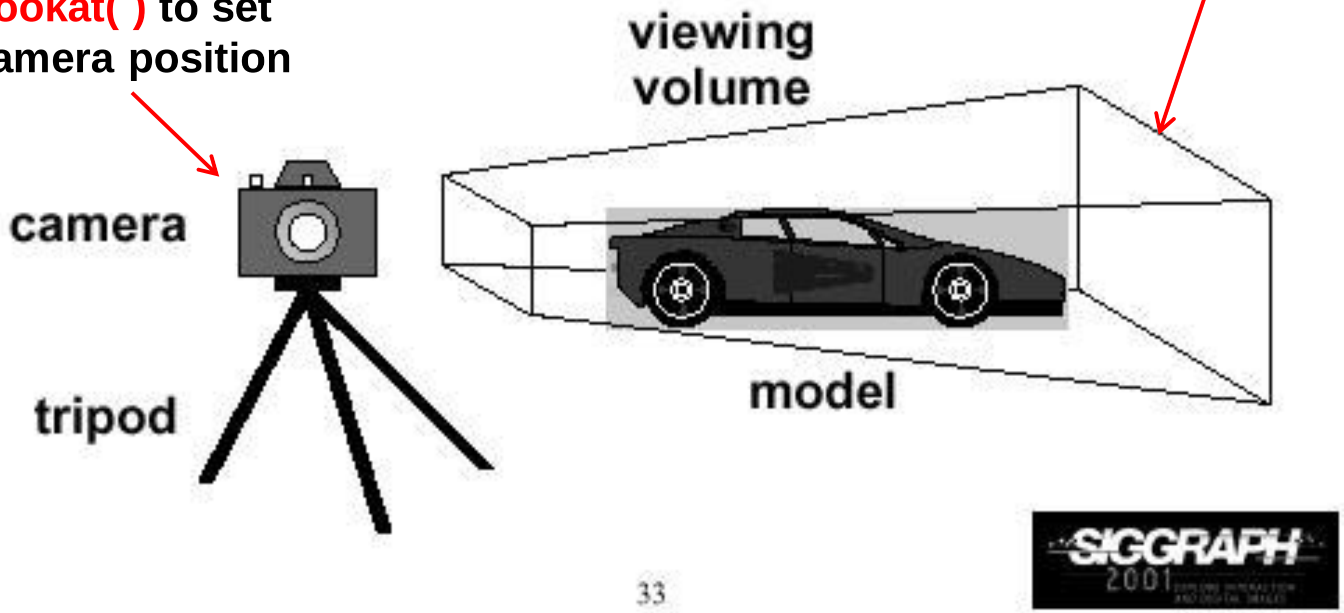
Computer Science Dept.

Worcester Polytechnic Institute (WPI)

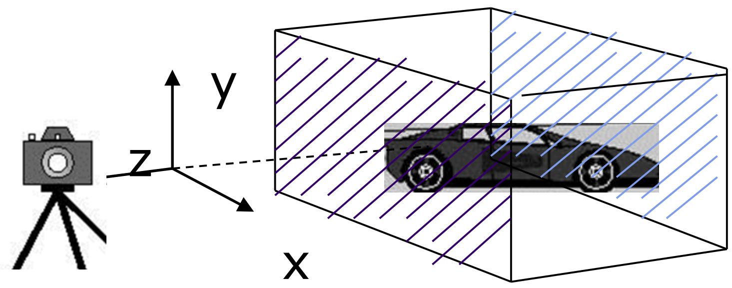
Recall: 3D Viewing and View Volume

Now:
Set view volume

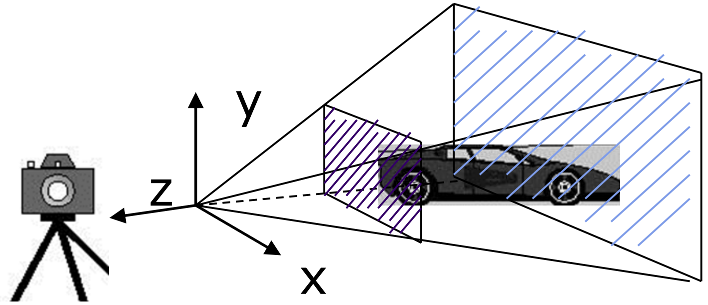
Previously:
Lookat( ) to set camera position



Recall: Different View Volume Shapes



Orthogonal view volume
(no foreshortening)



Perspective view volume
(exhibits foreshortening)

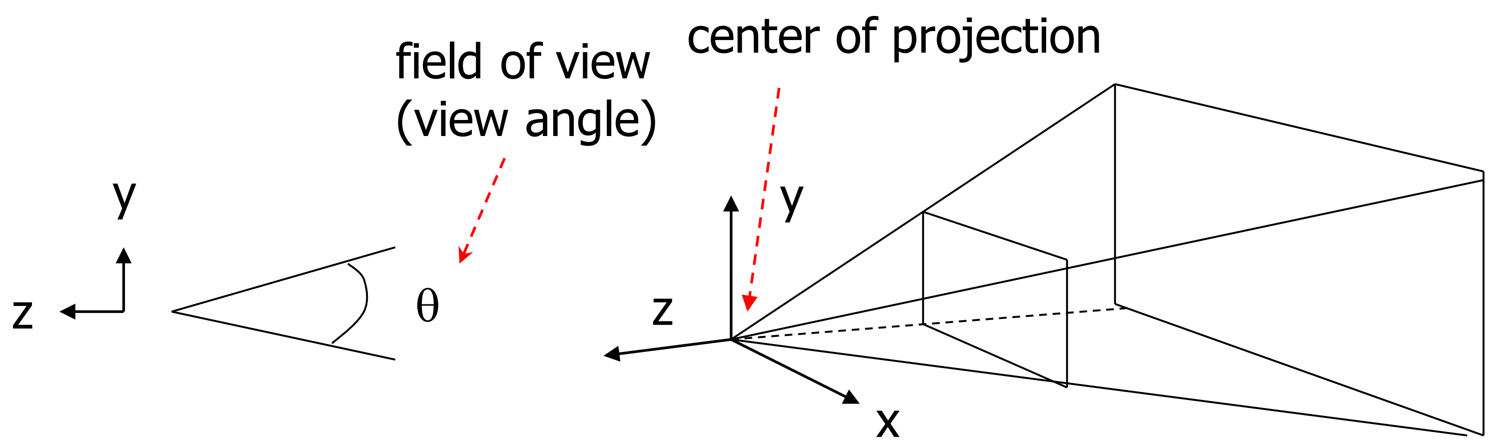
- Different view volume => different look
- **Foreshortening?** Near objects bigger

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

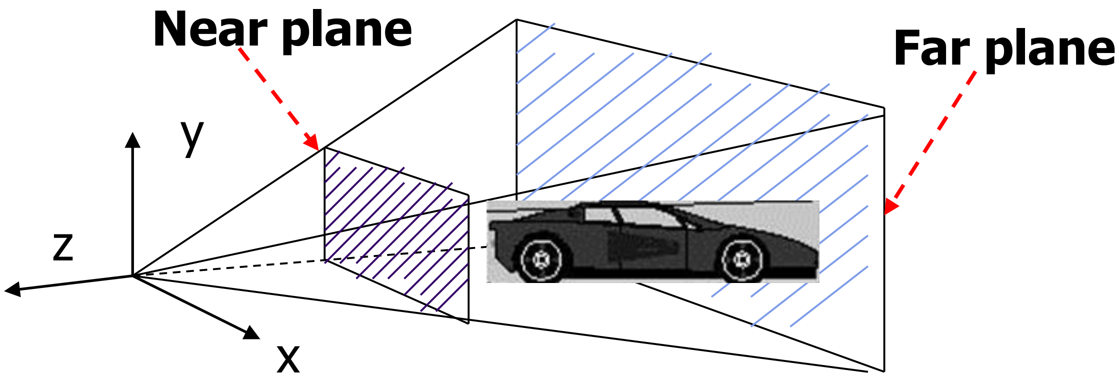
Field of View

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



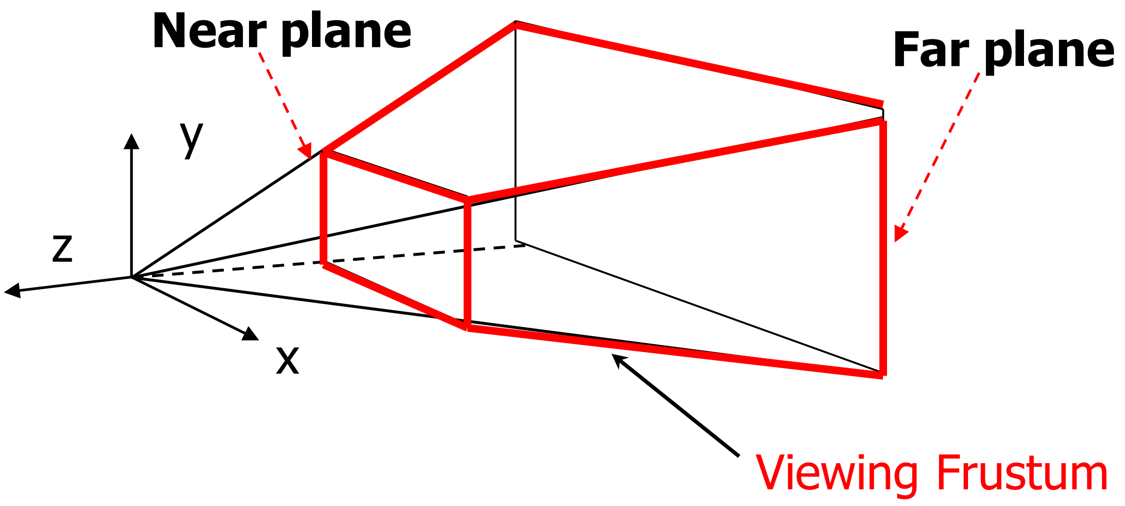
Near and Far Clipping Planes

- Only objects between near and far planes drawn



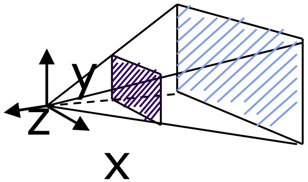
Viewing Frustrum

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

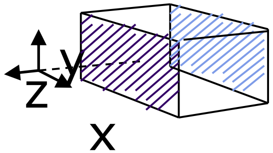


Setting up View Volume/Projection Type

- 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)

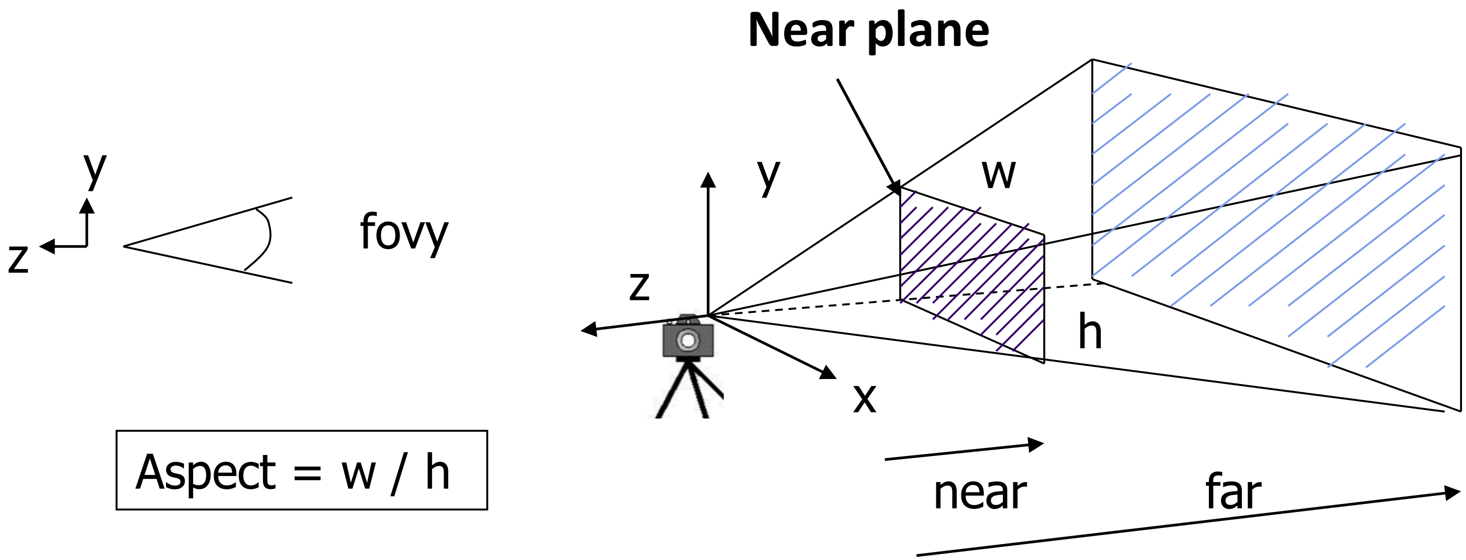


- Useful functions, so we implement similar in `mat.h`:
    - **Perspective**(fovy, aspect, near, far) **or**
    - **Frustum**(left, right, bottom, top, near, far)
    - **Ortho**(left, right, bottom, top, near, far)

What are these arguments? Next!

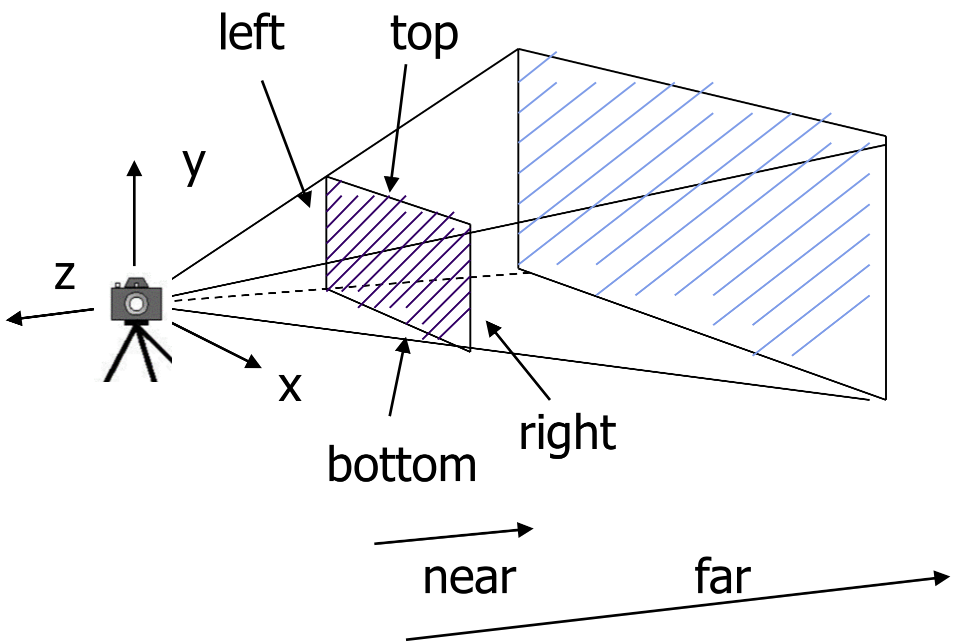
Perspective(fovy, aspect, near, far)

- Aspect ratio used to calculate window width



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

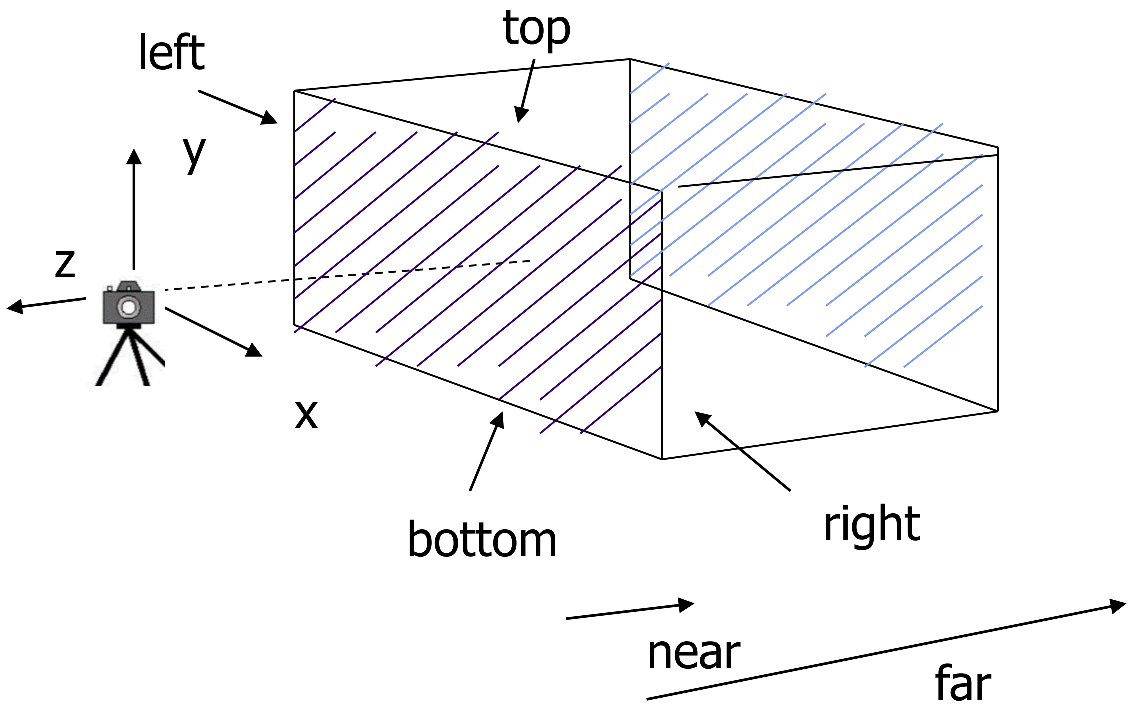
- Can use **Frustrum( )** in place of **Perspective**()
- Same view volume **shape**, different **arguments**



`near` and `far` measured from camera

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

- For orthographic projection



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

Example Usage:
Setting View Volume/Projection Type

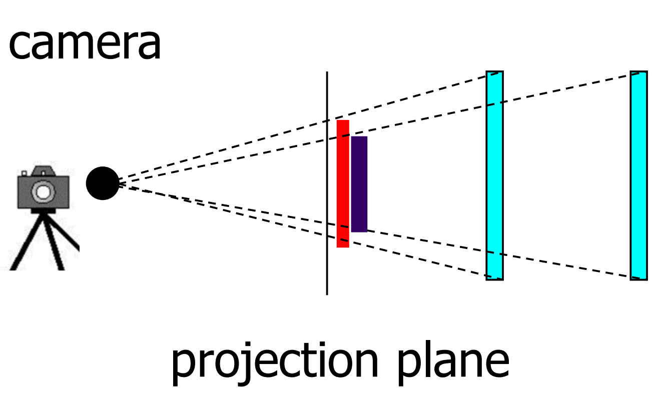
```
void display()
{      // clear screen
       glClear(GL_COLOR_BUFFER_BIT);
       ……….. .
       // Set up camera position
       LookAt(0,0,1,0,0,0,0,1,0);
              eye    at    up
       ……….. .
       // set up perspective transformation
       Perspective(fovy, aspect, near, far);

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

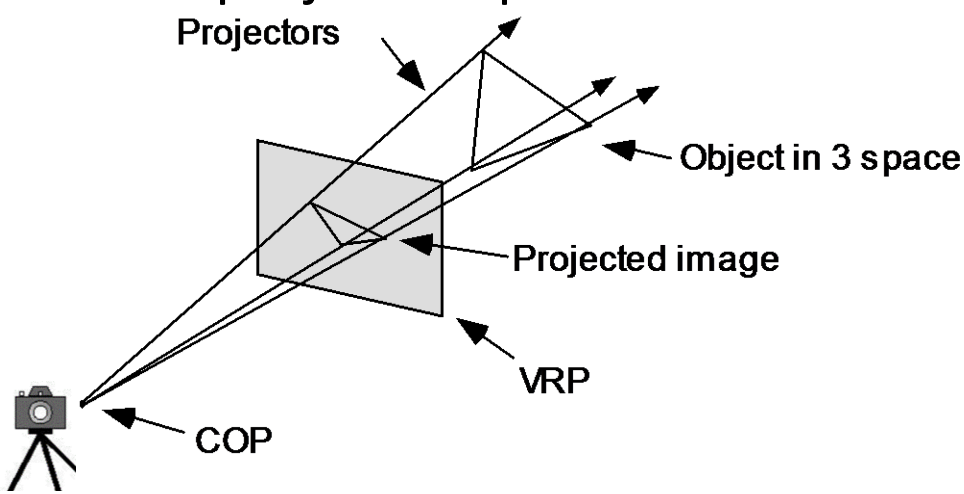
Demo

- Nate Robbins demo on projection

Perspective Projection

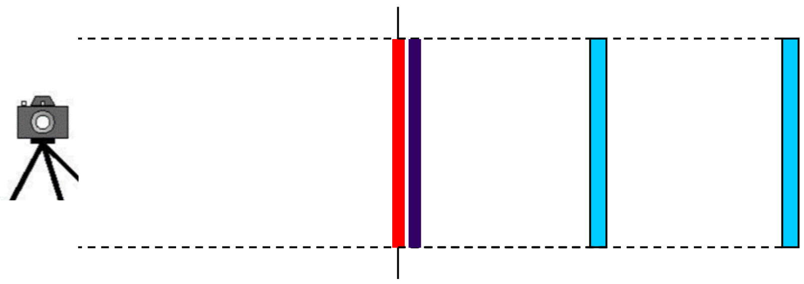
- After setting view volume, then projection transformation
- Projection?
  - **Classic:** Converts 3D object to corresponding 2D on screen
  - How? Draw line from object to projection center
  - Calculate where each intersects projection plane

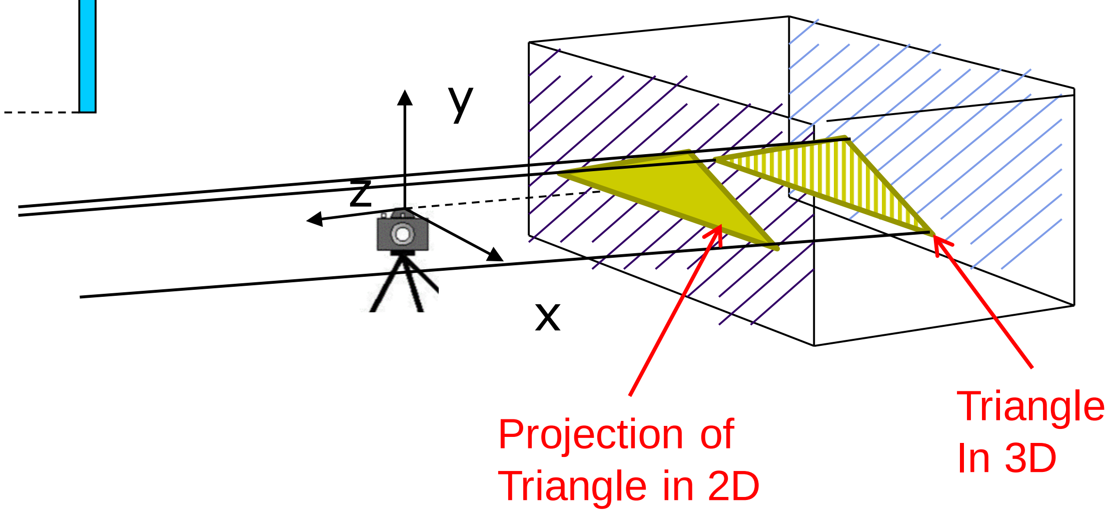




Orthographic Projection

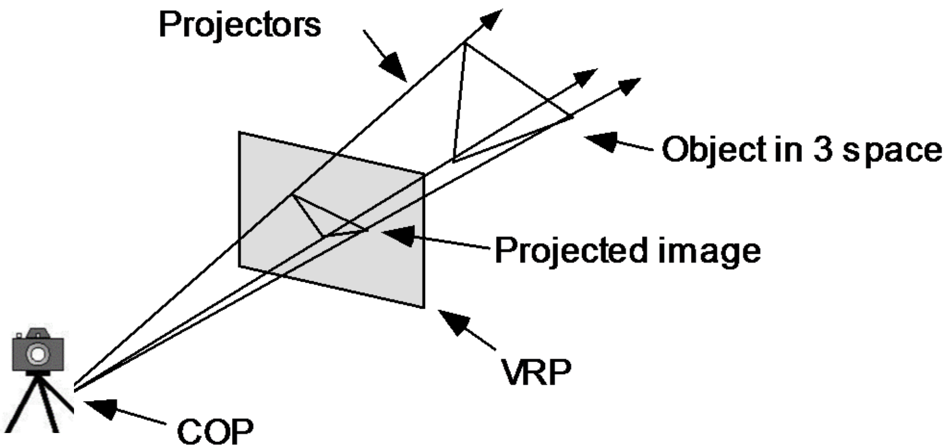
- How? Draw parallel lines from each object vertex
- The projection center is at infinite
- In short, use (x,y) coordinates, just drop z coordinates

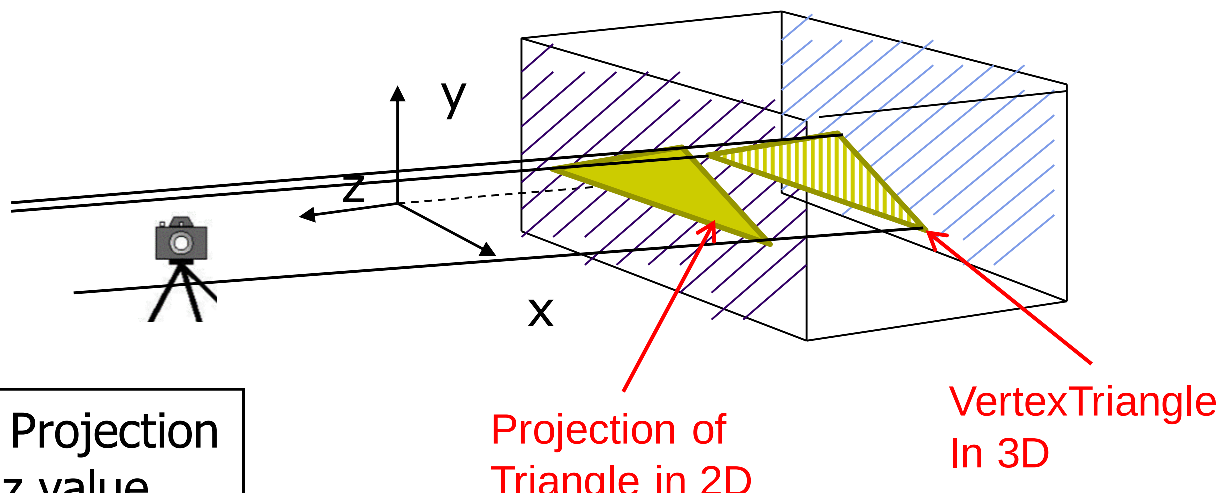




The Problem with Classic Projection

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



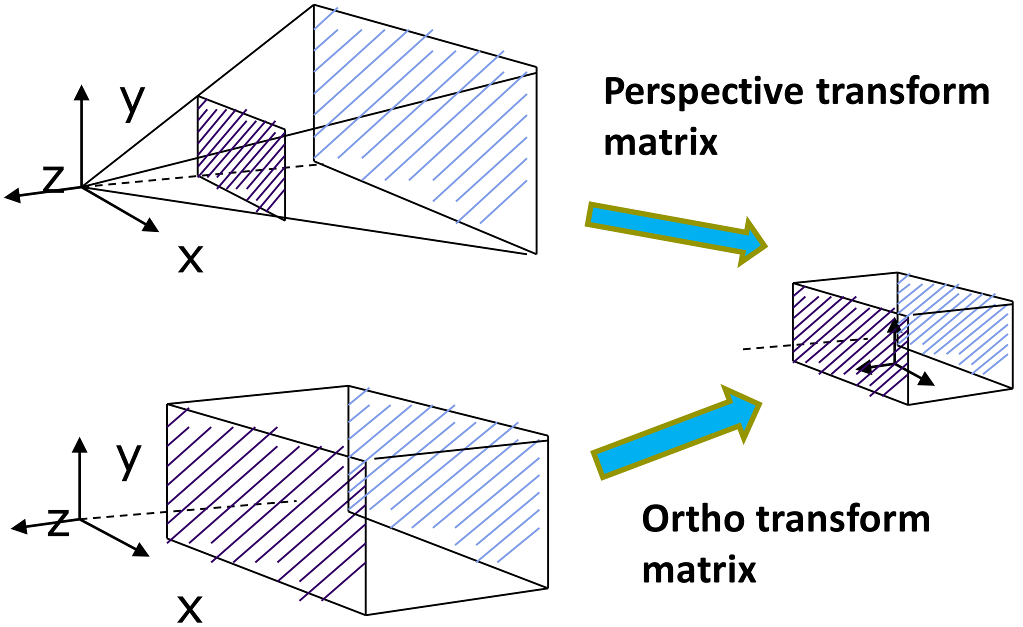


$$x_p = x$$
$$y_p = y$$
$$z_p = 0$$

Classic Projection
Loses z value

Normalization: Keeps z Value

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



Default view volume
Clipping against it

Computer Graphics (CS 4731)
Lecture 15: Projection (Part 2): Derivation

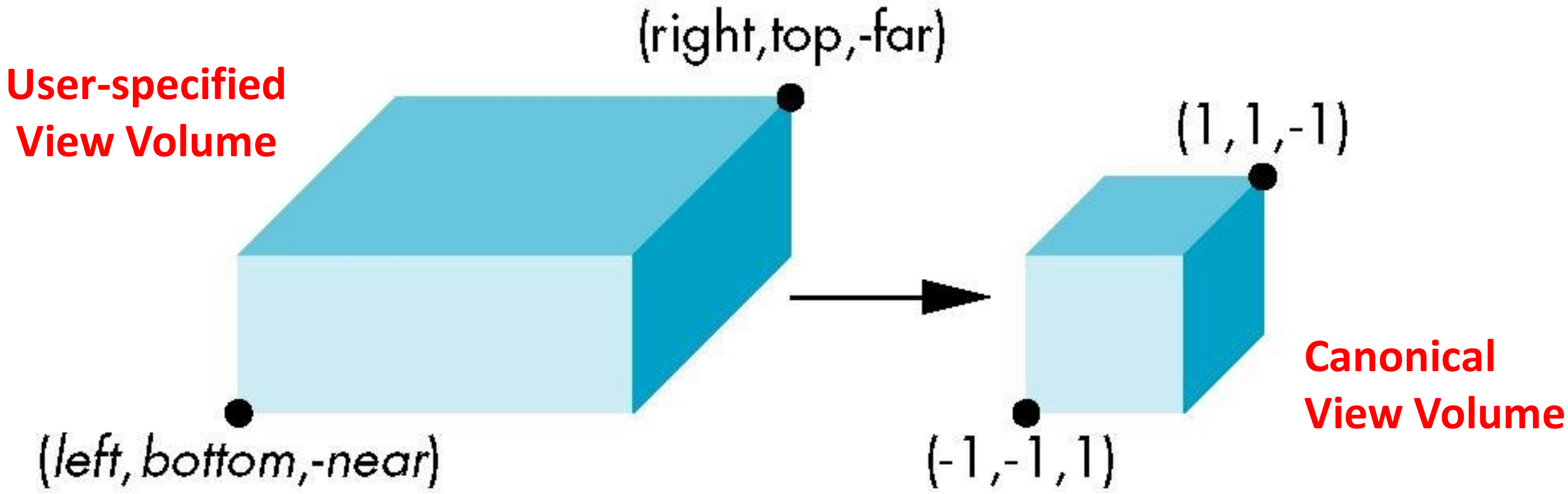
Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

Parallel Projection

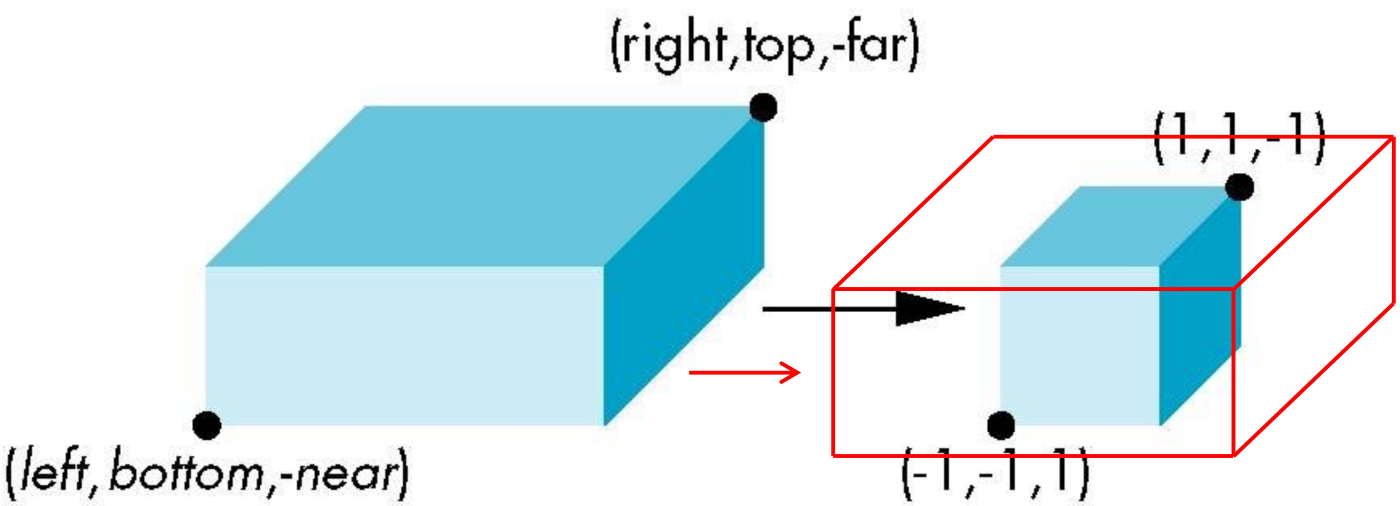
- **normalization** ⇒ find 4x4 matrix to transform **user-specified view volume** to **canonical view volume (cube)**



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

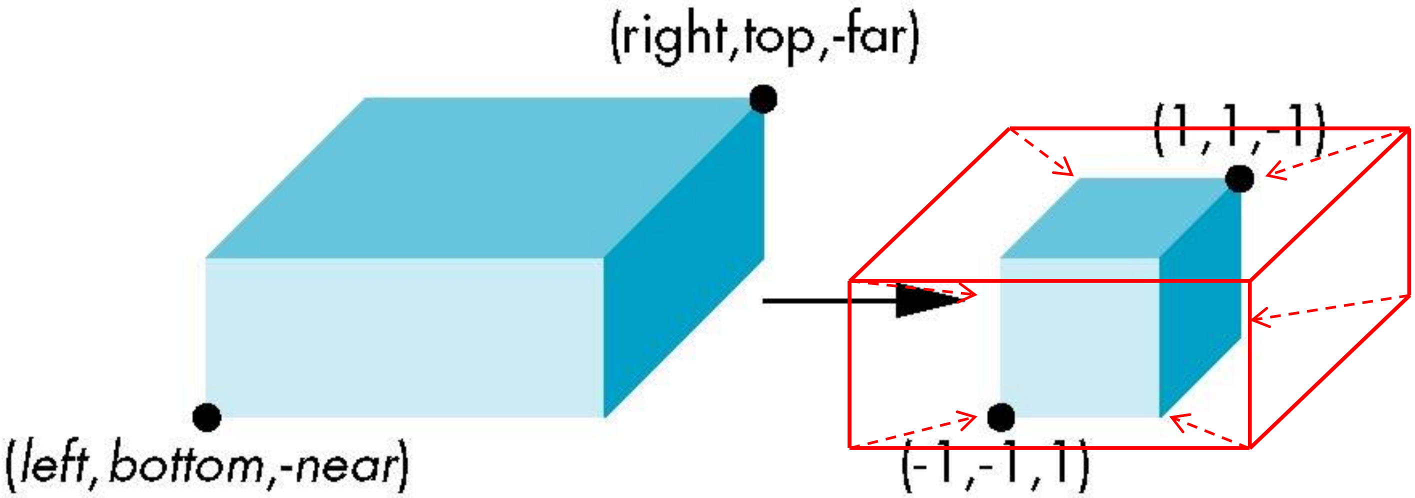
Parallel Projection: Ortho

- Parallel projection: 2 parts
  1. **Translation:** centers view volume at origin



Parallel Projection: Ortho

2. **Scaling:** reduces user-selected cuboid to canonical cube (dimension 2, centered at origin)

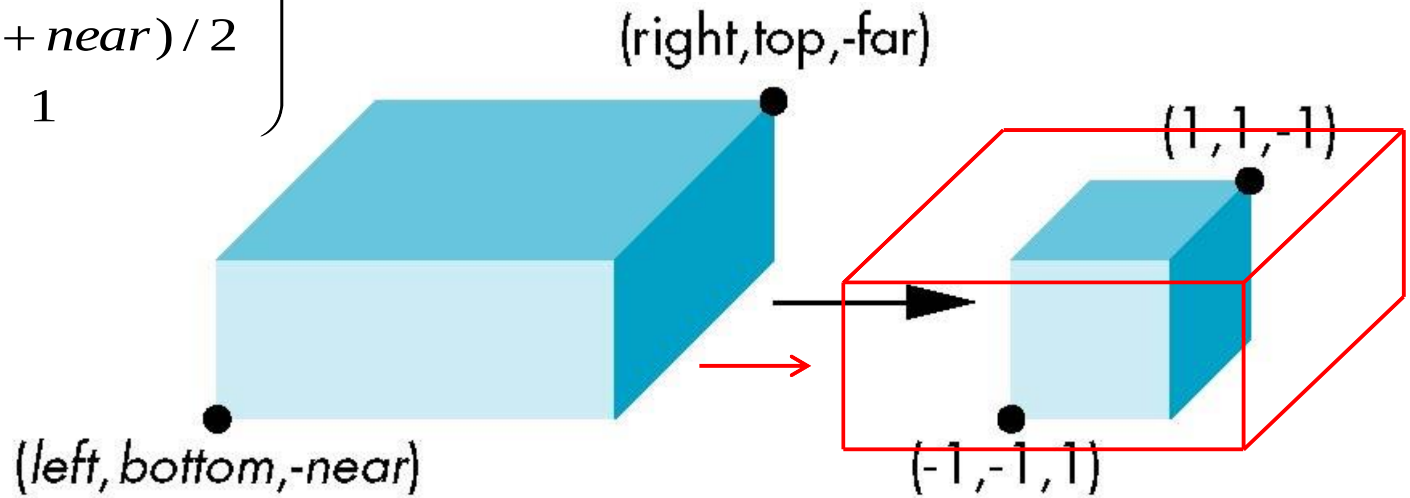


Parallel Projection: Ortho

- Translation lines up midpoints: E.g. midpoint of x = (right + left)/2
- Thus translation factors:

  -(right + left)/2, -(top + bottom)/2, -(far+near)/2
- Translation matrix:

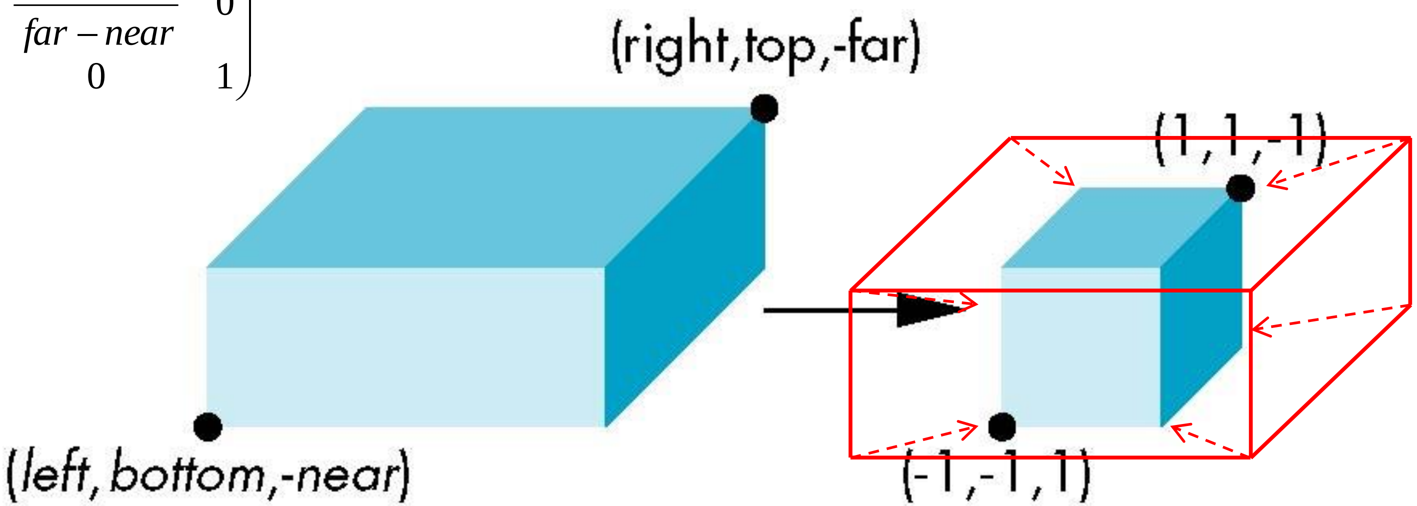
$$\begin{pmatrix} 1 & 0 & 0 & -(right+left)/2 \\ 0 & 1 & 0 & -(top+bottom)/2 \\ 0 & 0 & 1 & -(far+near)/2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Parallel Projection: Ortho

- Scaling factor: ratio of ortho view volume to cube dimensions
- Scaling factors: 2/(right - left), 2/(top - bottom), 2/(far - near)
- Scaling Matrix M2:

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & 0 \\ 0 & \frac{2}{top-bottom} & 0 & 0 \\ 0 & 0 & \frac{2}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Parallel Projection: Ortho

Concatenating **Translation** x **Scaling**, we get Ortho Projection matrix

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & 0 \\ 0 & \frac{2}{top-bottom} & 0 & 0 \\ 0 & 0 & \frac{2}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & -(right+left)/2 \\ 0 & 1 & 0 & -(top+bottom)/2 \\ 0 & 0 & 1 & -(far+near)/2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

Final Ortho Projection

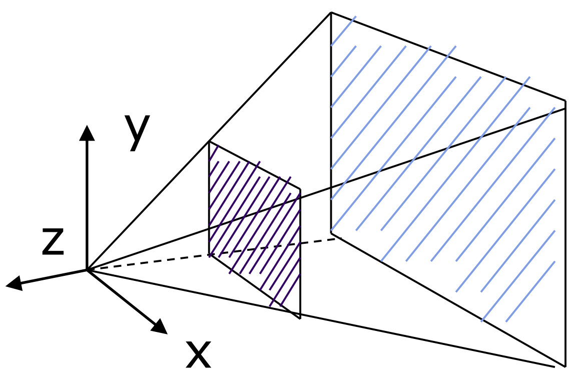
- Set $z = 0$
- Equivalent to the homogeneous coordinate transformation

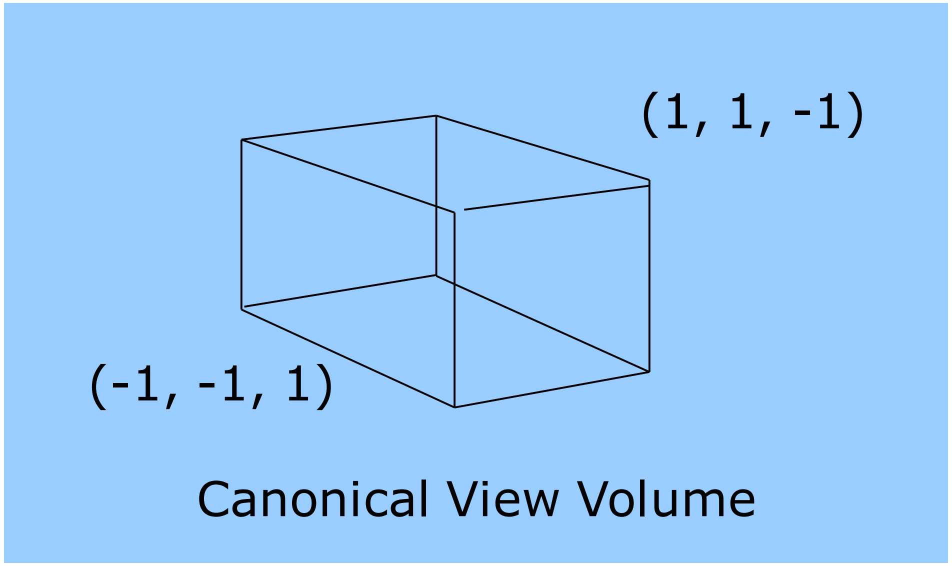
$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Hence, general orthogonal projection in 4D is
$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{ST}$

Perspective Transformation

- We want to transform viewing frustum volume into canonical view volume





Perspective Projection Matrix

Derivation skipped!

$$\begin{pmatrix} \frac{2N}{x\max - x\min} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2N}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{-(F+N)}{F-N} & \frac{-2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

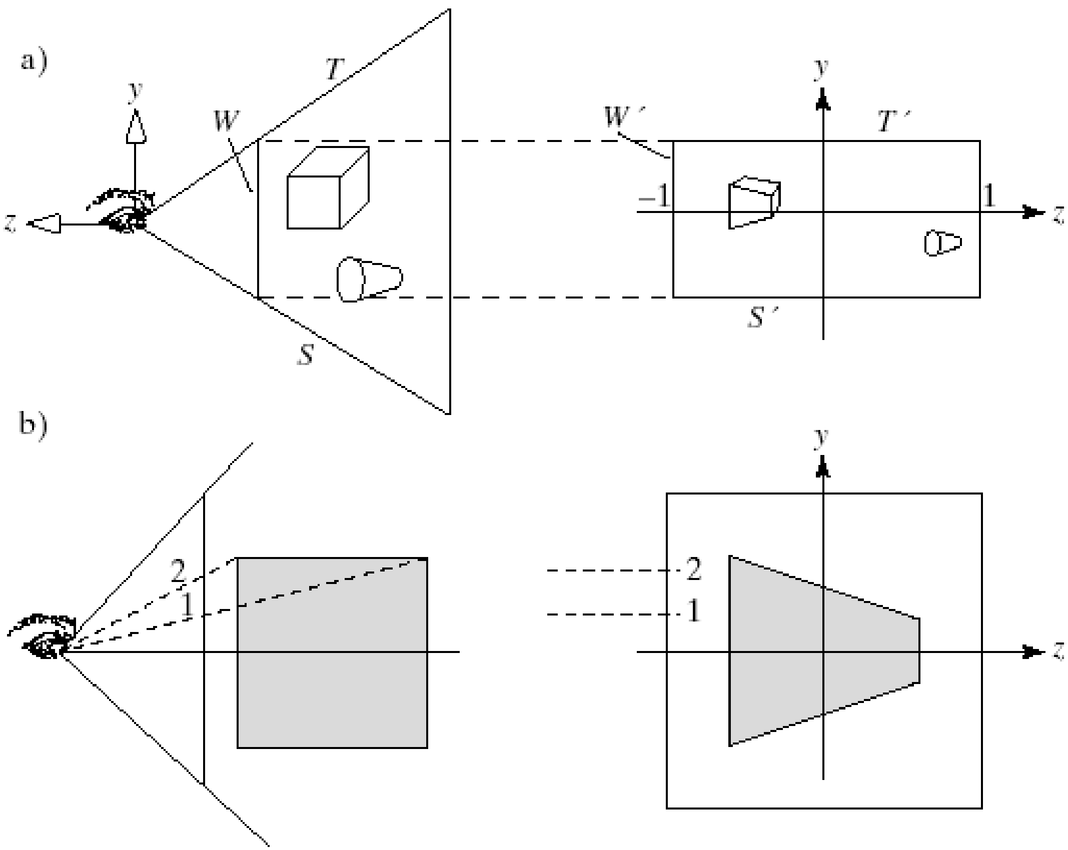
Final Perspective Transform Matrix

glFrustum(left, right, bottom, top, N, F) N = near plane, F = far plane

Geometric Nature of Perspective Transform

a) Lines through eye map into lines parallel to z axis after transform

b) Lines perpendicular to z axis map to lines perp to z axis after transform



Implementation

- Set modelview and projection matrices in application program
- Pass matrices to shader

Build 4x4 projection matrix

```
void display( ){
     .....
     model_view = LookAt(eye, at, up);
     projection = Ortho(left, right, bottom,top, near, far);

     // pass model_view and projection matrices to shader
     glUniformMatrix4fv(matrix_loc, 1, GL_TRUE, model_view);
     glUniformMatrix4fv(projection_loc, 1, GL_TRUE, projection);
  .....
}
```

Implementation

- And the corresponding shader

```
in vec4 vPosition;
in vec4 vColor;
Out vec4 color;
uniform mat4 model_view;
Uniform mat4 projection;

void main( )
{
   gl_Position = projection*model_view*vPosition;
   color = vColor;
}
```

References

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