CS 432 – Interactive Computer Graphics

Lecture 4 – Part 4

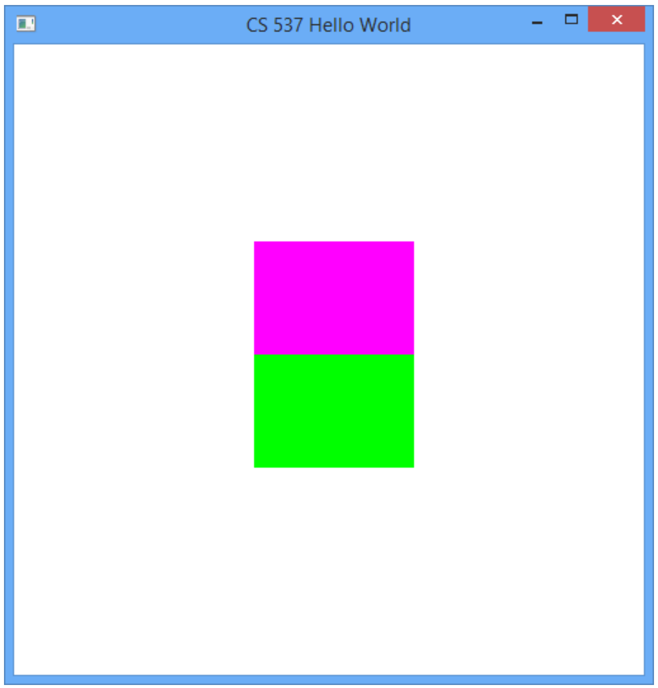
Projection

Projections

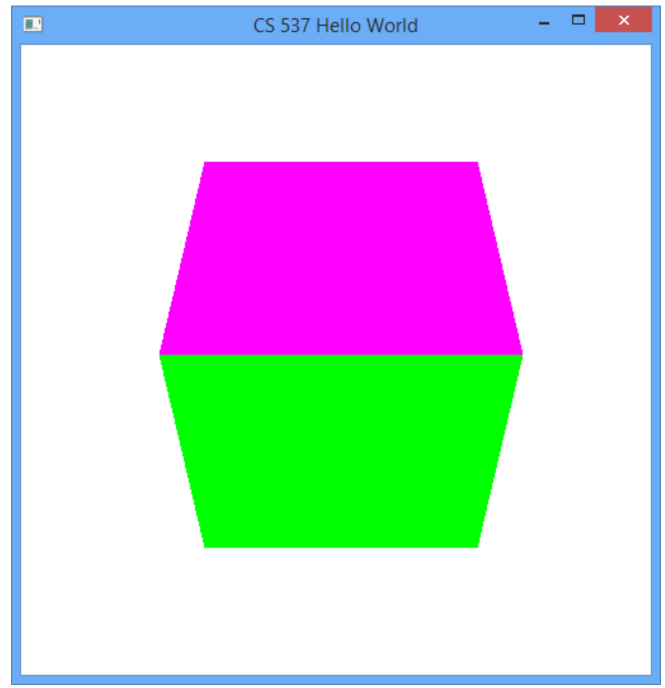
- In addition to how the camera is positioned and oriented we need to specify how things should look on its projection plane
  - This is akin to specifying camera/lens properties
- There are two common ways
  1. Orthographic (parallel)
  2. Perspective

Perspective vs Parallel

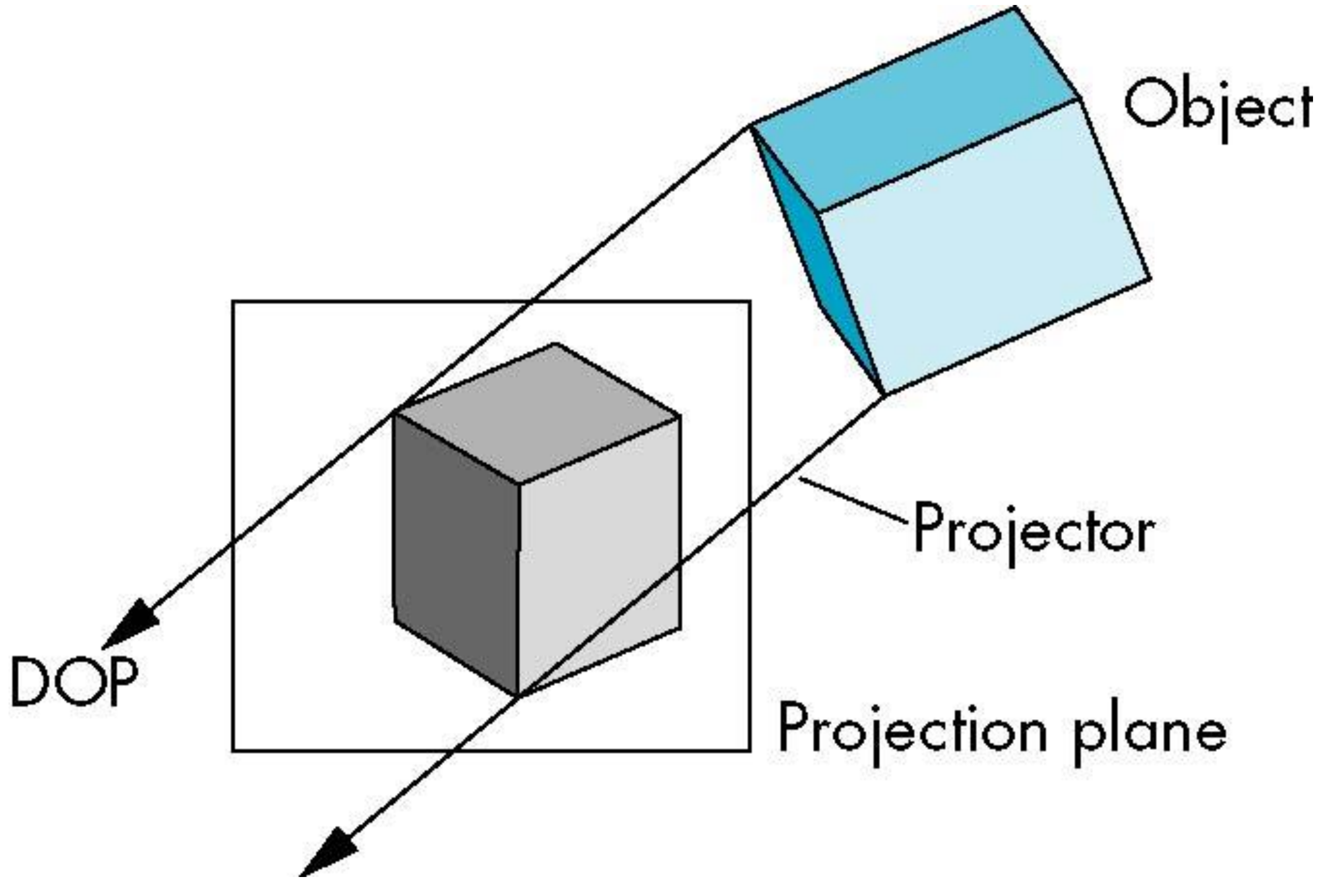
Parallel



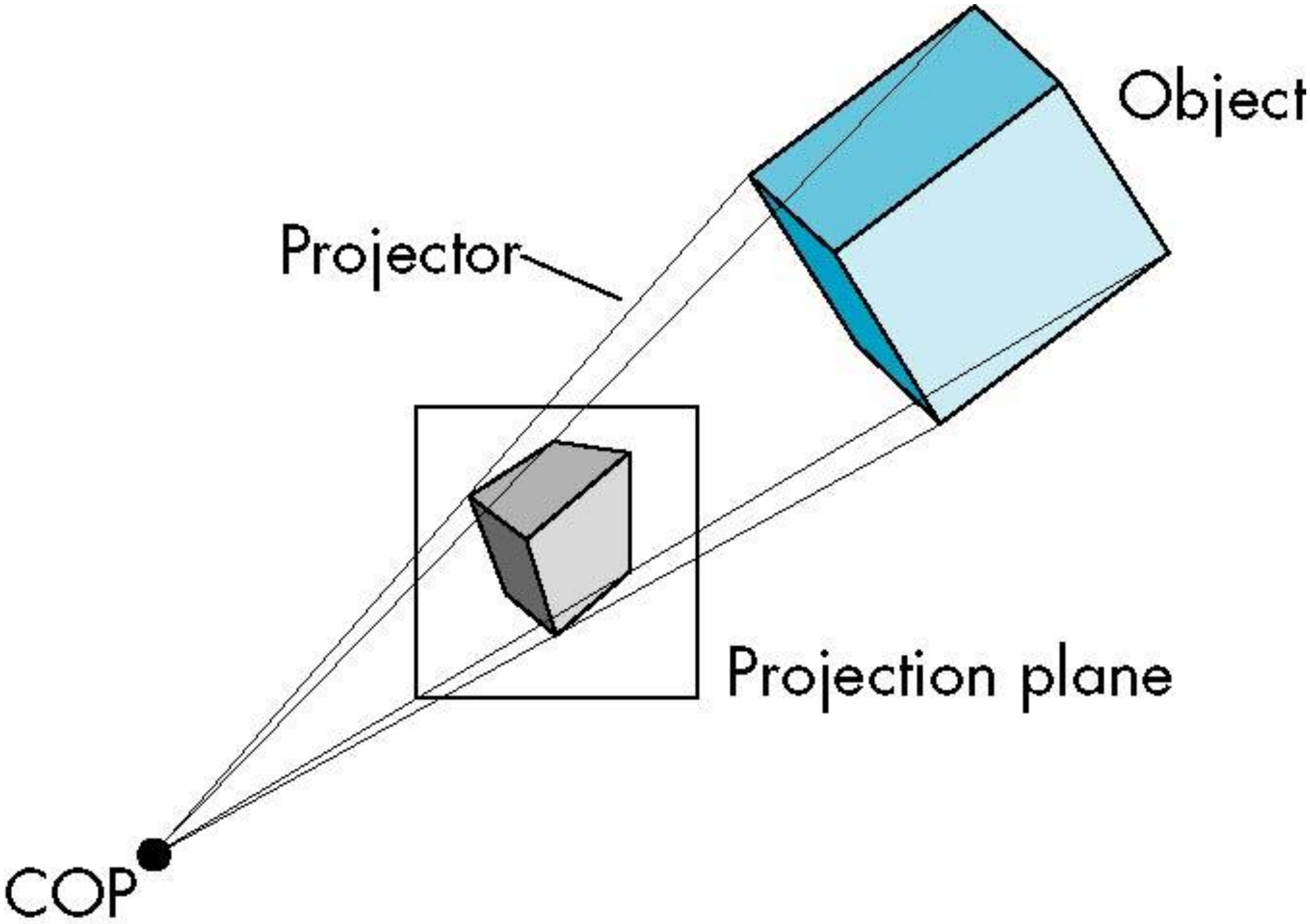
Perspective



Parallel Projection

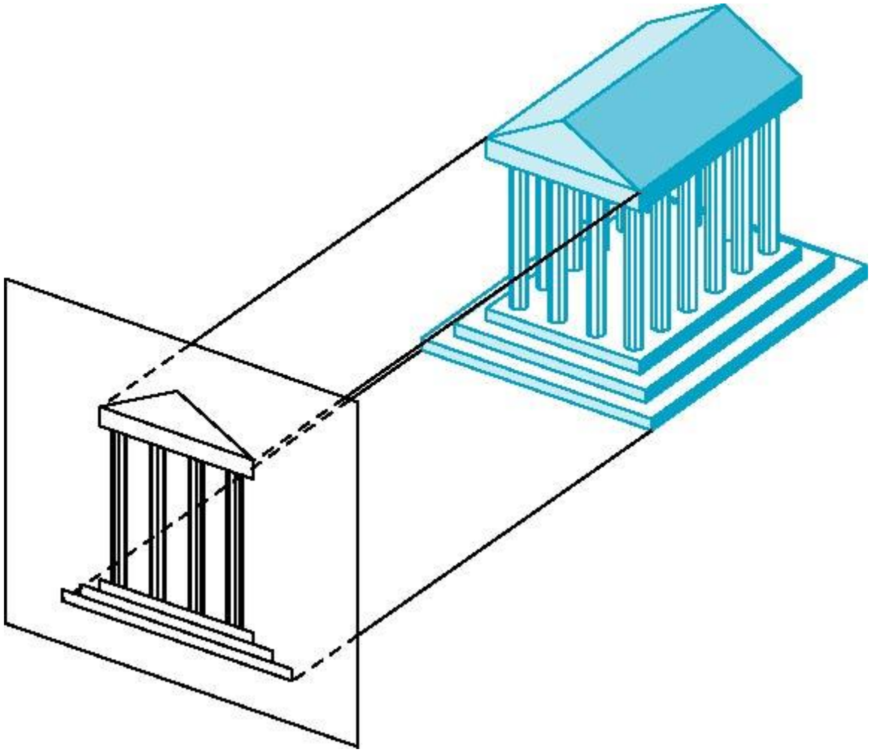


Perspective Projection



Orthographic Projection

- Projectors are *orthogonal* to projection surface

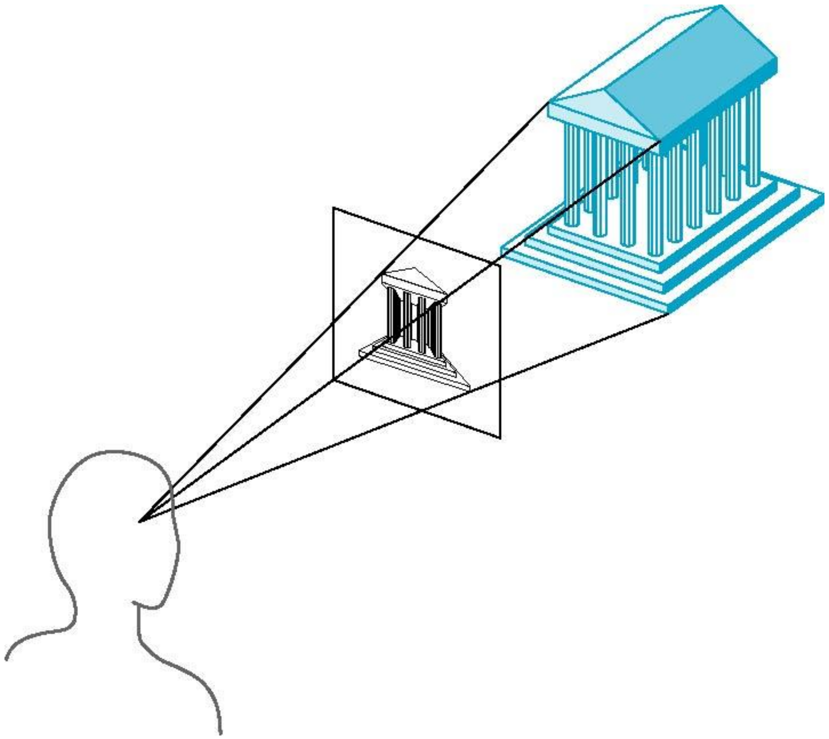


Pros and Cons of Orthographic/Parallel

- Preserves both distance and angles
    - Shapes preserved
    - Can be used for measurements
        - Building plans
        - Manuals
- Not realistic looking
    - Distant objects are as large a near objects

Perspective Projection

- Projectors converge at center of projection

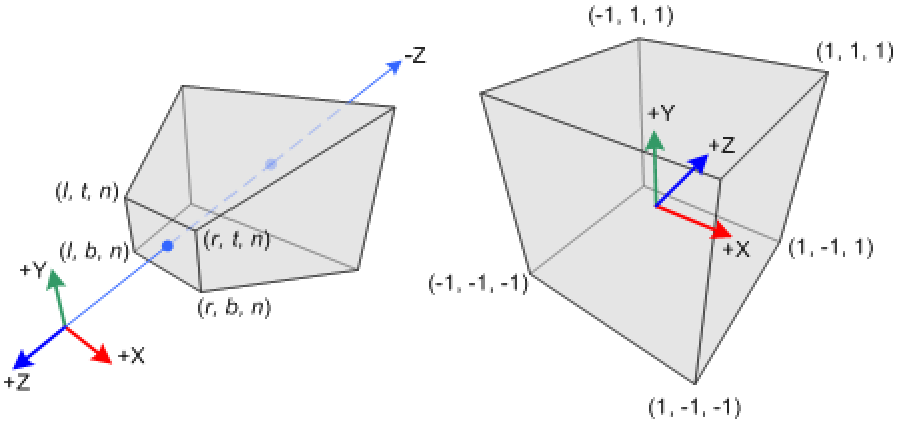


Pros and Cons of Perspective Projection

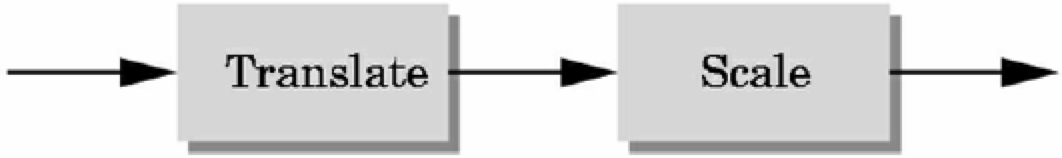
- Objects further from viewer are projected smaller than the same sized objects closer to the viewer
  - Looks realistic
- Equal distances along a line are not projected into equal distances
  - Non-uniform foreshortening

Mapping to Standard Cube

- Regardless of your projection time (orthographic or perspective) we want to map the volume on to a standard 2x2x2 cube
  - This makes clipping and the final projection easy
- This provides what is called ***normalized device coordinates***
  - And it follows the **left-hand-rule**



Orthographic Projection Matrix



Move to center

$$\mathbf{P}=\mathbf{ST}=\begin{bmatrix} \frac{2}{x_{\max}-x_{\min}} & 0 & 0 & -\frac{x_{\max}+x_{\min}}{x_{\max}-x_{\min}} \\ 0 & \frac{2}{y_{\max}-y_{\min}} & 0 & -\frac{y_{\max}+y_{\min}}{y_{\max}-y_{\min}} \\ 0 & 0 & \frac{2}{z_{\max}-z_{\min}} & -\frac{z_{\max}+z_{\min}}{z_{\max}-z_{\min}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

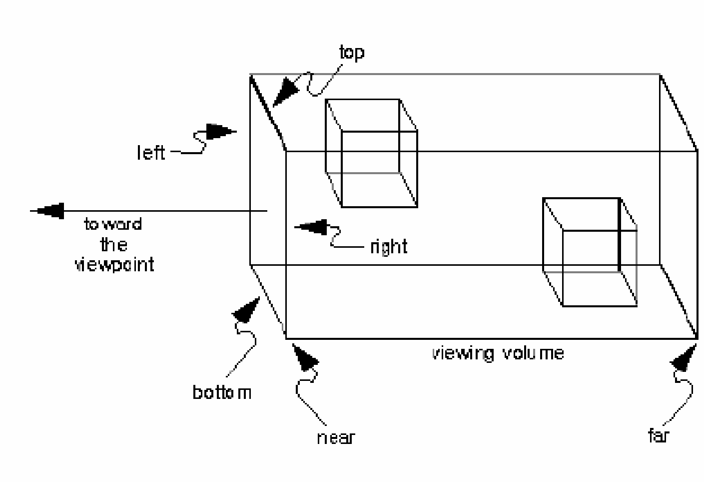
Scale to fit 2x2x2 volume

OpenGL Orthographic Projection

- Fortunately OpenGL provides functions to build these matrices more easily…

  `mat4 glOrtho(left, right, bottom, top, near, far);`

- These parameters are relative to the camera's coordinate system (position and orientation).
- We'll use Angel's Ortho function to get back a projection matrix (in mat.h)
  - `mat4 Ortho(left, right, bottom, top, near, far)`



OpenGL: Orthographic Projection

- Just like the model matrix and the view matrix we can now decide how we want to use our projection matrix depending on the nature of your application.
- One way would be just to send all three matrices individually to the shader and have it do the full multiplication:
- OpenGL code…

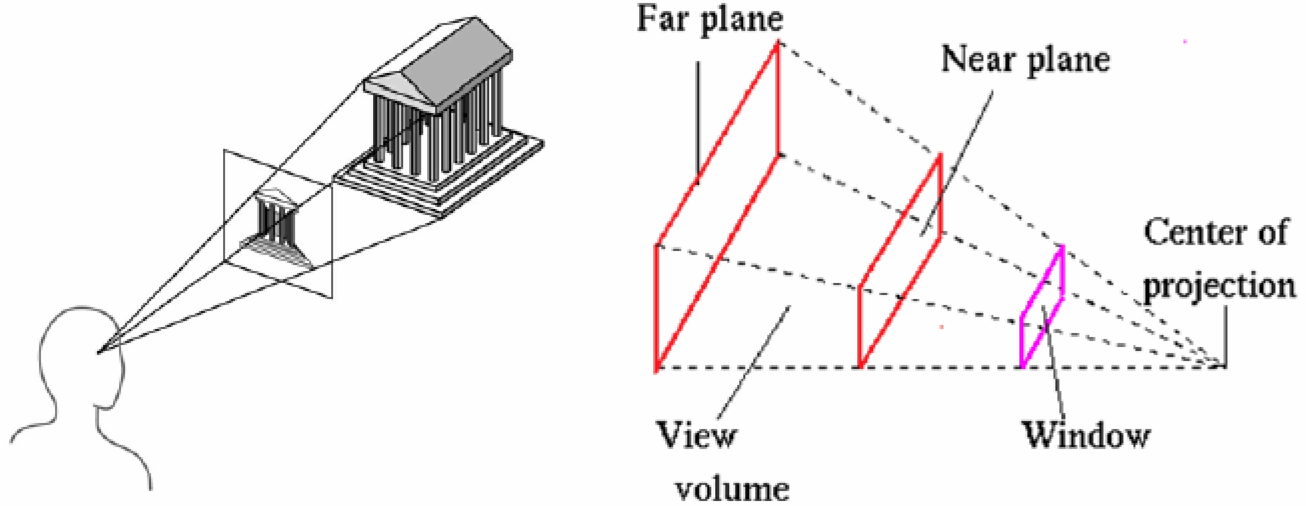
```
mat4 proj = Ortho(left, right, bottom, top, near, far);
GLuint proj_loc =
        glGetUniformLocation(program,"proj_matrix");
glUniformMatrix4fv(proj_loc,1,GL_TRUE proj);
```

- In the vertex shader….

```
gl_Location = proj_matrix*view_matrix*model_matrix*vPosition;
```

Perspective Viewing

- This is similar to the “pin-hole camera”



Math of Perspective Projection

- Given camera reference frame `(VRP, u, v, n)`
- The projection transformation maps 3D points to 2D points in the projection pane
- Standard configuration
  - COP=VRP
  - Projection plane is orthogonal to z-axis, at $z = d$

Simple Perspective Projection

- Let $p = [x, y, z]$ be the point in camera coordinates
- We can do perspective projection using similar triangles and the projection plane located at $z = d$ to get the point $q = [x', y', z']$
  - $x' = xd/z$
  - $y' = yd/z$
  - $z' = d$

Simple Perspective Projection

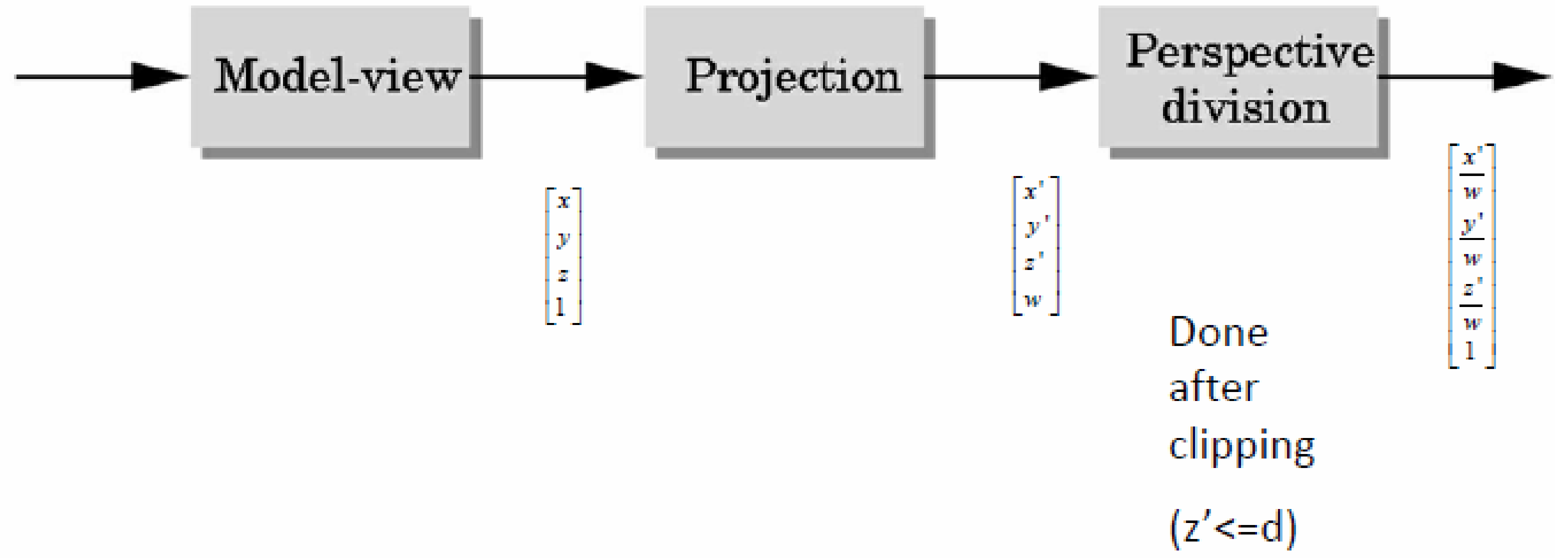
- Doing this perspective projection in homogenous coordinates with a perspective matrix, M_{per} we get:

  $[x'y', z', w']^T = M * [x\ y\ z\ 1]^T$ where

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

- However this results in $w' = z$, when we actually want $w = 1$!
- So we must then divide everything by w'
  - This is called the *perspective division*

Perspective Projections

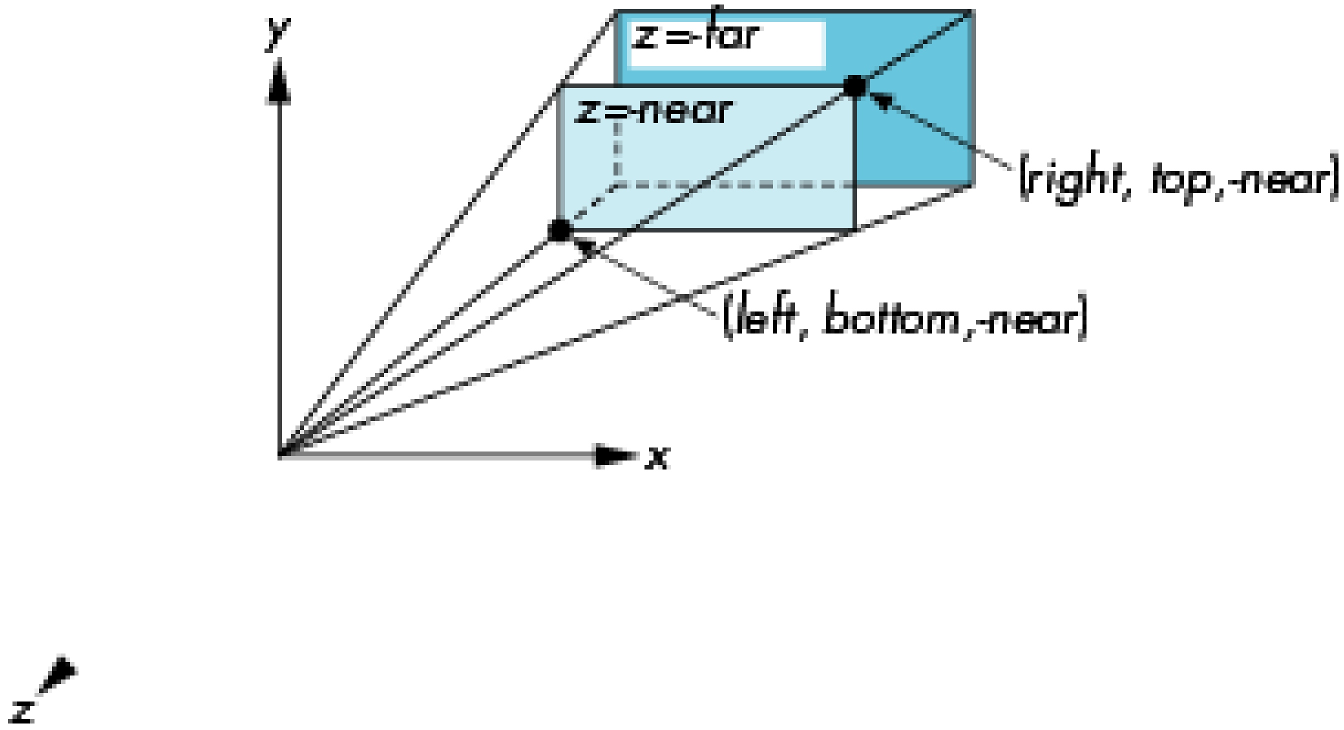


General Perspective Projection

- Like Orthographic projection, we can also specify a bounding volume which must be mapped into the 2x2x2 cube for clipping
- In perspective viewing this volume is referred to as the *frustum*
- We can allow OpenGL to create the final perspective projection matrix one of two ways
  - Explicitly provide the planes that define the frustum
  - Provide the near and far planes, the *field of view,* and the *aspect ratio*
  - Just like with orthographic projection, the parameters are relative to the camera coordinate system.

OpenGL Perspective

- `Frustum(left, right, bottom, top, near, far)`
- Near and far must both be positive, relative to the COP



Frustum (from Angel.h)

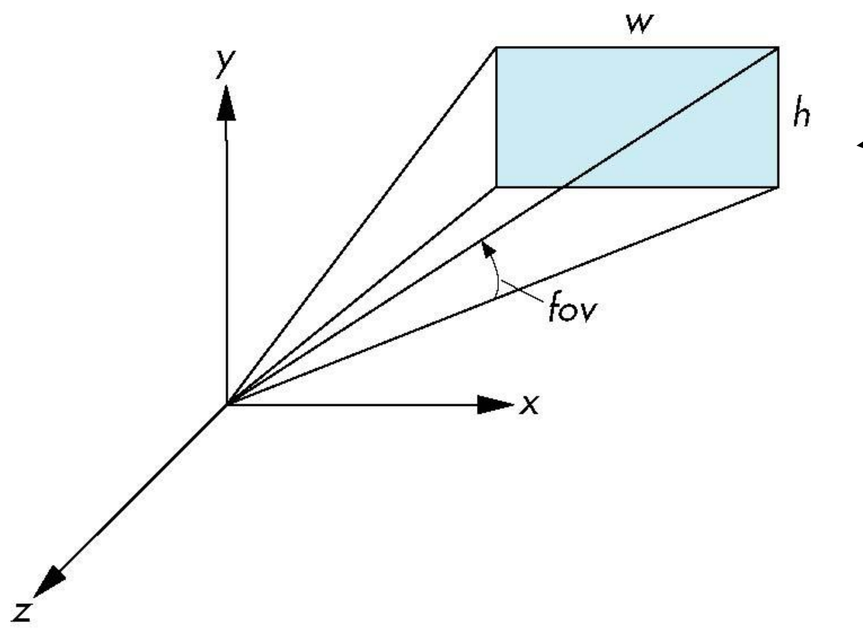
- `projectionMatrix = Frustum(left, right, bottom, top, near, far);`

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

```
mat4 Frustum( const GLfloat left, const GLfloat right,
              const GLfloat bottom, const GLfloat top,
              const GLfloat zNear, const GLfloat zFar )
{
    mat4 c;
    c[0][0] = 2.0*zNear/(right - left);
    c[0][2] = (right + left)/(right - left);
    c[1][1] = 2.0*zNear/(top - bottom);
    c[1][2] = (top + bottom)/(top - bottom);
    c[2][2] = -(zFar + zNear)/(zFar - zNear);
    c[2][3] = -2.0*zFar*zNear/(zFar - zNear);
    c[3][2] = -1.0;
    return c;
}
```

Using Field of View

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



front plane

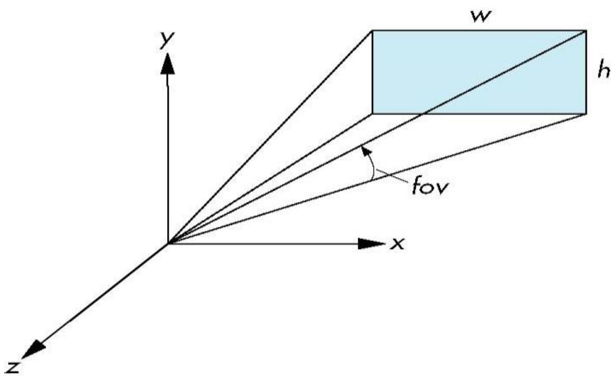
`aspect = w/h`

Using Field of View

- `mat4 Proj = Perspective(fovy, aspect, near, far)`
- `t = tan(fov/2)*zNear; //top`
- `r = top*aspect; //right`

Incorrect in Angel 6th edition
Uses t=tan(fov)*zNear

$$Proj = \begin{bmatrix} n/r & 0 & 0 & 0 \\ 0 & n/t & 0 & 0 \\ 0 & 0 & -(f+n)/(f-n) & -2fn/(f-n) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



front plane

`aspect = w/h`

Perspective (Angel.h)

- `projectionMatrix = Perspective(fovy, aspect, n, f);`

```
mat4 Perspective( const GLfloat fovy, const GLfloat aspect,
                  const GLfloat zNear, const GLfloat zFar)
{
    GLfloat top   = tan(fovy*DegreesToRadians/2) * zNear;
    GLfloat right = top * aspect;

    mat4 c;
    c[0][0] = zNear/right;
    c[1][1] = zNear/top;
    c[2][2] = -(zFar + zNear)/(zFar - zNear);
    c[2][3] = -2.0*zFar*zNear/(zFar - zNear);
    c[3][2] = -1.0;
    return c;
}
```

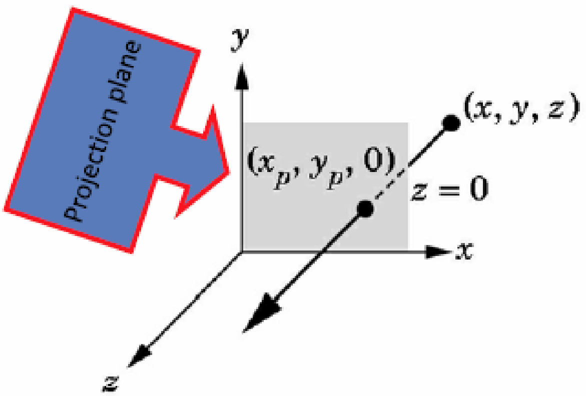
Reshape Function

- Since the Perspective function requires an aspect ratio, it might be a good idea to set this up whenever the window is resized…
- Of course the code below should be better organized using `Camera` and `Drawable` objects…

```
void resize(int w, int h){
    glViewport(0,0,(GLsizei) w, (GLsizei) h);
    projmat = Perspective(65.0, GLfloat(w/h), 1.0, 100.0);
    glUniformMatrix4fv(proj_loc, 1, GL_TRUE, projmat);
}
```

Final Projection

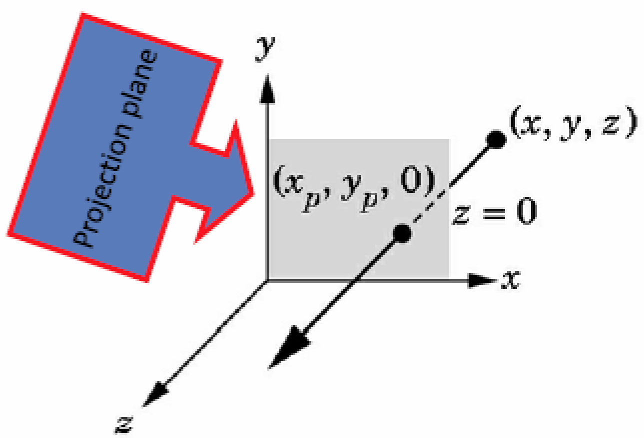
- How that our points are in normalized device coordinates (via a projection matrix) we can perform a simple orthographic projection onto the plane $z = 0$ for our final image
  - Actually this will be done automatically for us



$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \overbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{\text{Projection matrix}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Standard Orthographic Projection

- The point (x, y, z) is projected onto the point $(x, y, 0)$ in the plane $z = 0$



Projection matrix

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

Pipeline View