Learning Objectives

- Students completing this lecture will be able to
  - Explain the following terms: view normalization, perspective division
  - Write WebGL code to implement desired viewing

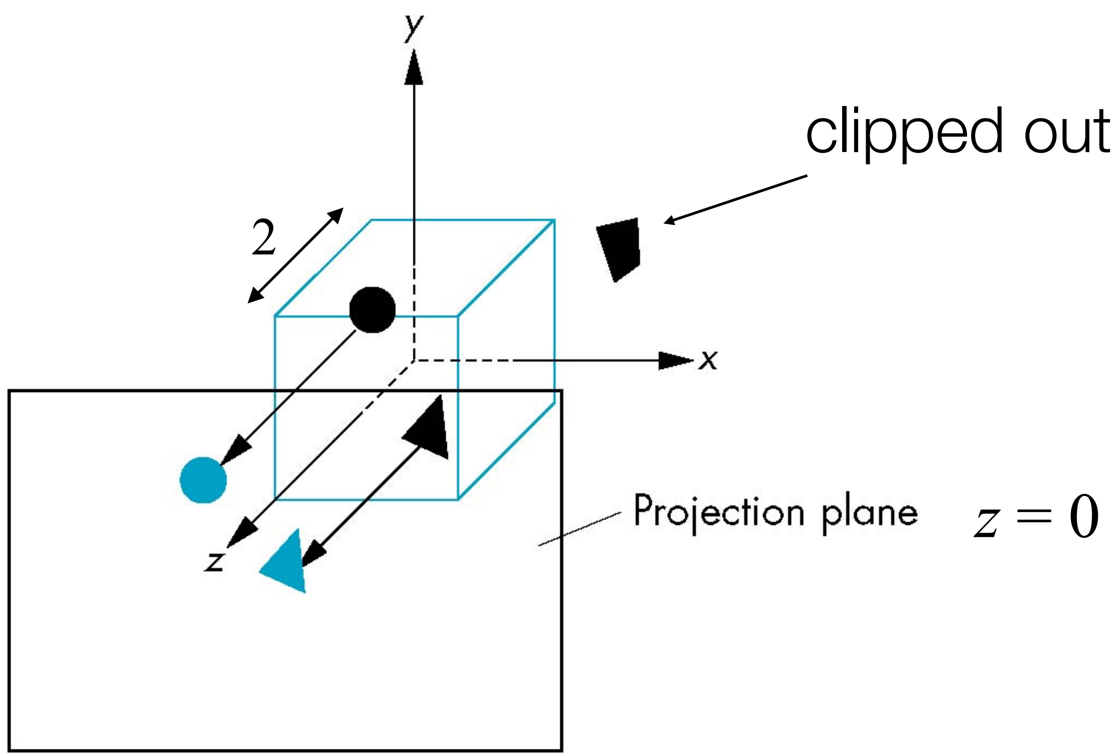
WebGL Camera

The WebGL Camera

- In WebGL, initially the object 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

- Default projection is orthogonal



Moving the Camera Frame

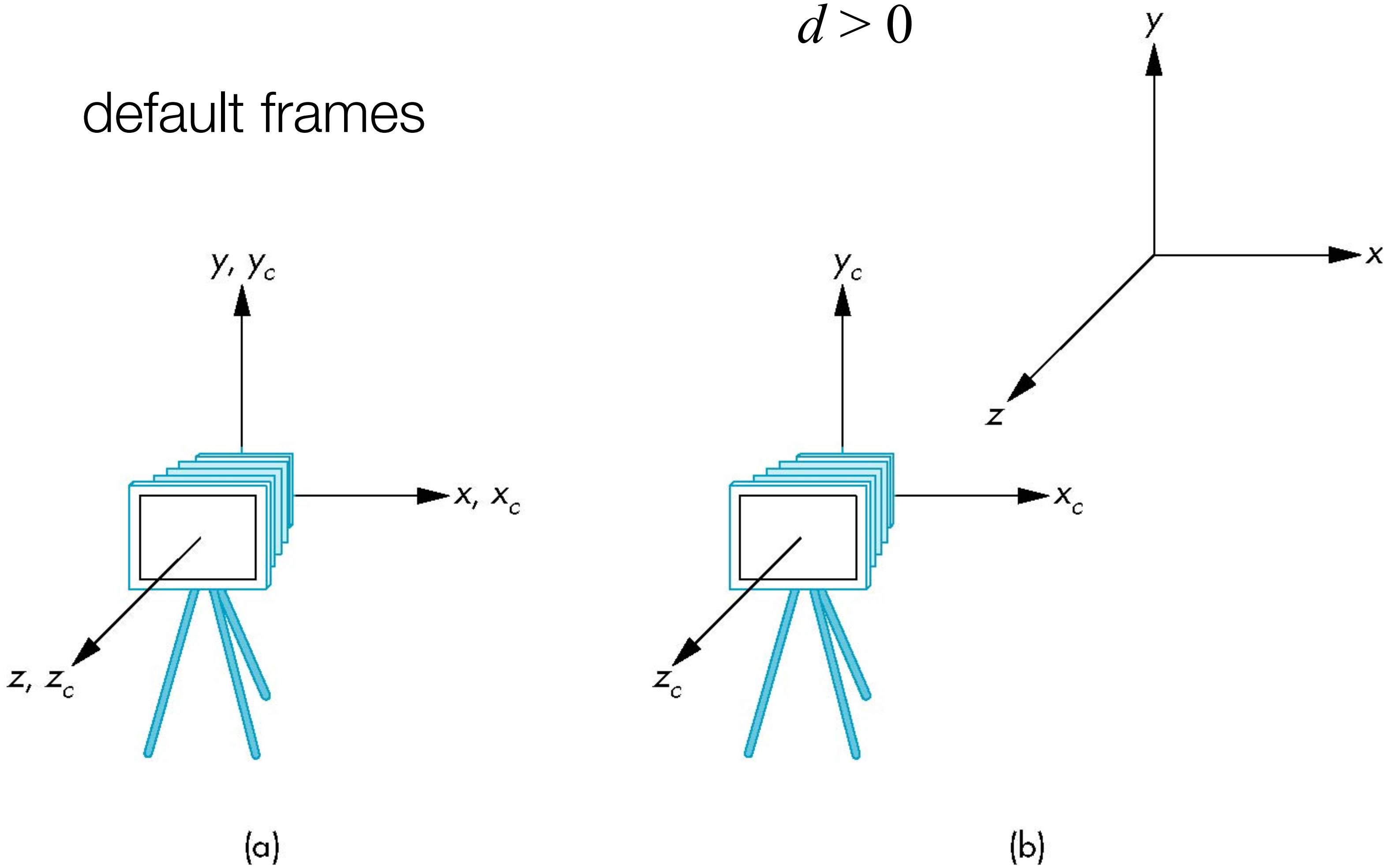
- If we want to visualize object with both positive and negative z values we can either
  - Move the camera in the positive z direction
    - Translate the camera frame
  - Move the objects in the negative z direction
    - Translate the world frame
- Both of these views are equivalent and are determined by the model-view matrix
  - Want a translation (`translate(0.0,0.0,-d);`)
  - `d` > 0

Moving Camera back from Origin

frames after translation by $-d$

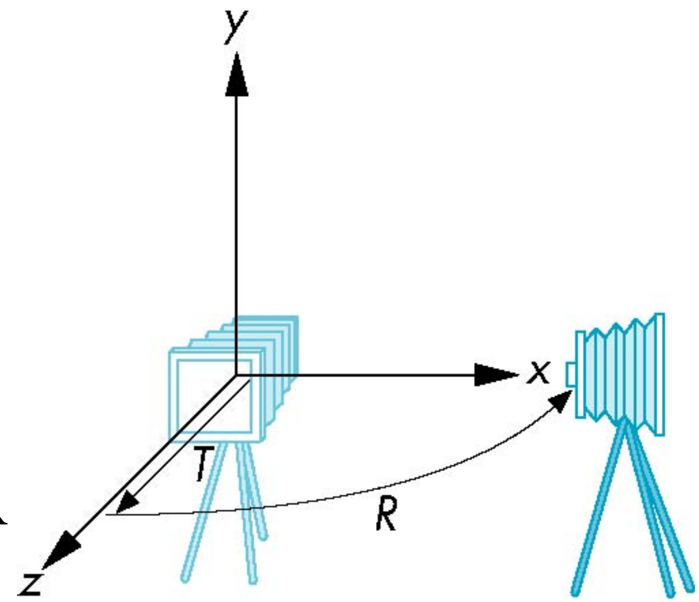
$d > 0$

default frames



Moving the Camera

- We can move the camera to any desired position by a sequence of rotations and translations
- Example: side view
  - Rotate the camera
  - Move it away from origin
  - Model-view matrix $\mathbf{C} = \mathbf{TR}$



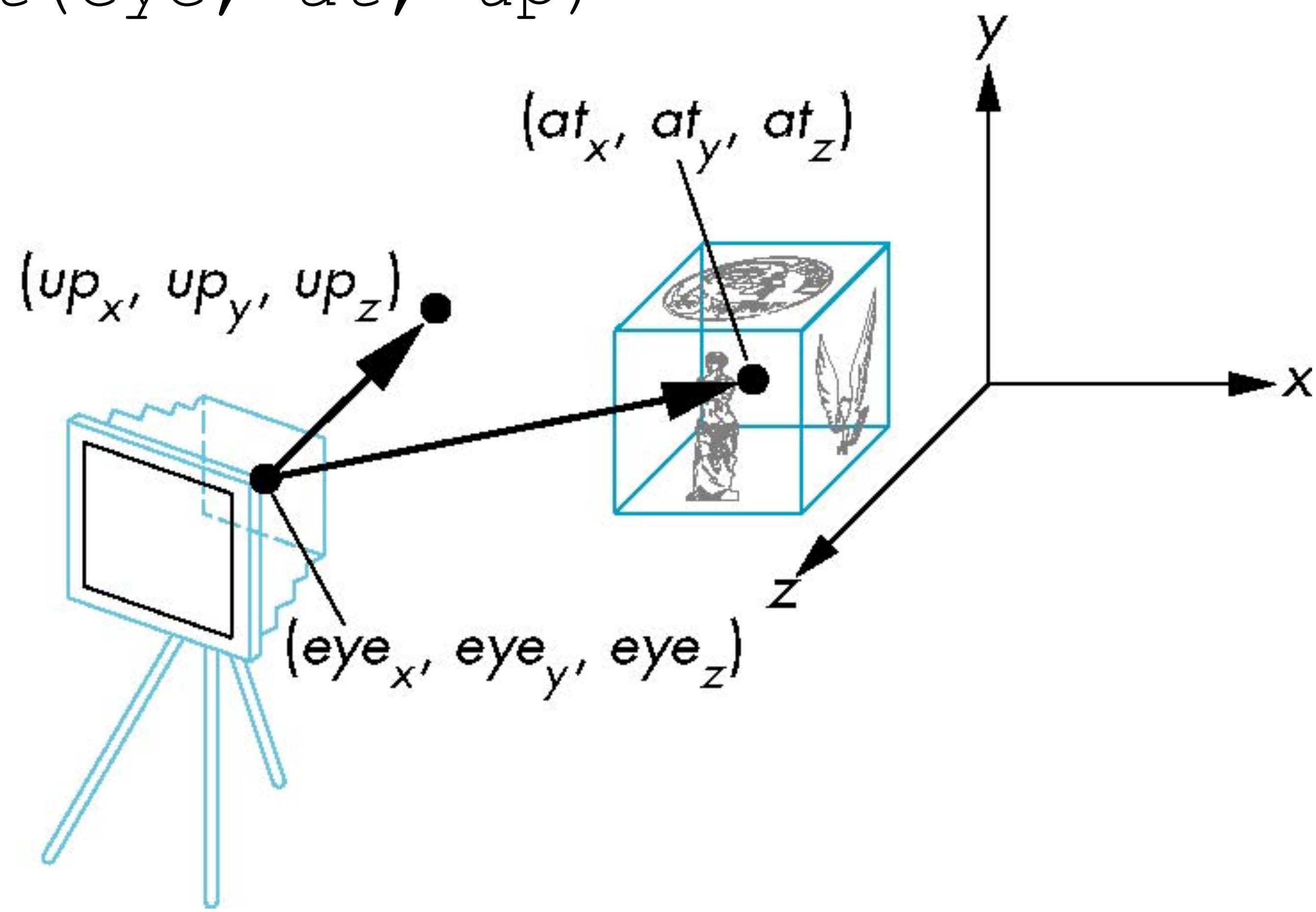
WebGL code

- Remember that last transformation specified is first to be applied

```
// Using MV.js

var t = translate (0.0, 0.0, -d);
var ry = rotateY(90.0);
var m = mult(t, ry);

// or

var m = mult(translate (0.0, 0.0, -d),
         rotateY(90.0););
```

The `lookAt` Function

`lookAt(eye, at, up)`



The `lookAt` Function

- The GLU library contained the function `gluLookAt` to form the required modelview matrix through a simple interface
- Note the need for setting an `up` direction
- Replaced by `lookAt()` in `MV.js`
  - Can concatenate with modeling transformations
- Example: isometric view of cube aligned with axes

```
var eye = vec3(1.0, 1.0, 1.0);
var at = vec3(0.0, 0.0, 0.0);
var up = vec3(0.0, 1.0, 0.0);
var mv = lookAt(eye, at, up);
```

WebGL Projection

Projections and Normalization

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

$$
\begin{aligned}
x_p &= x \\
y_p &= y \\
z_p &= 0
\end{aligned}
$$

- 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

default orthographic projection

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

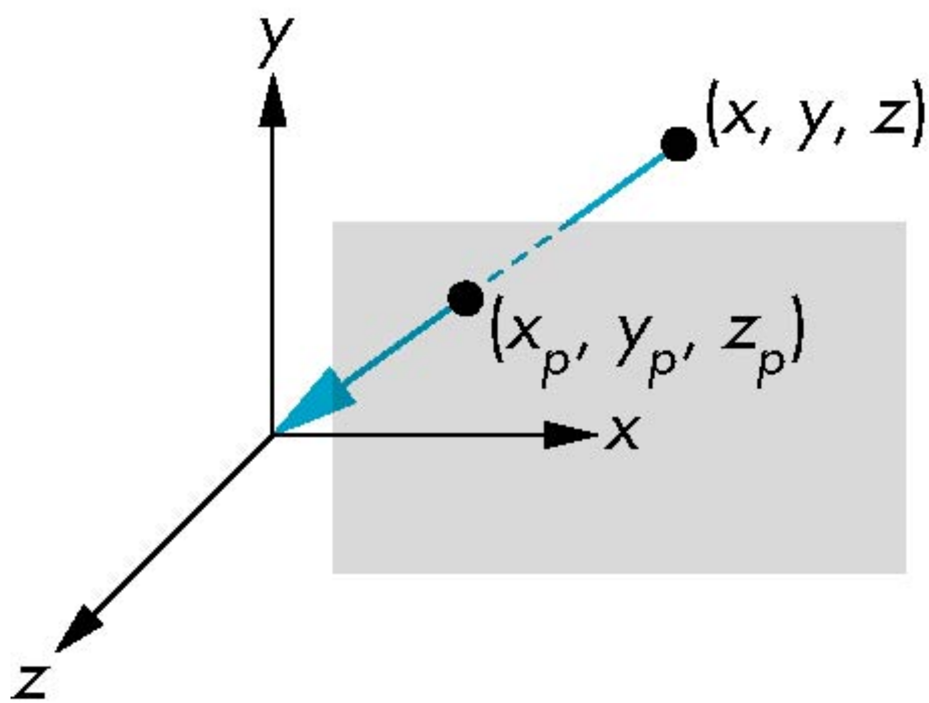
$$\mathbf{p}_{\mathrm{p}} = \mathbf{M}\mathbf{p}$$

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

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

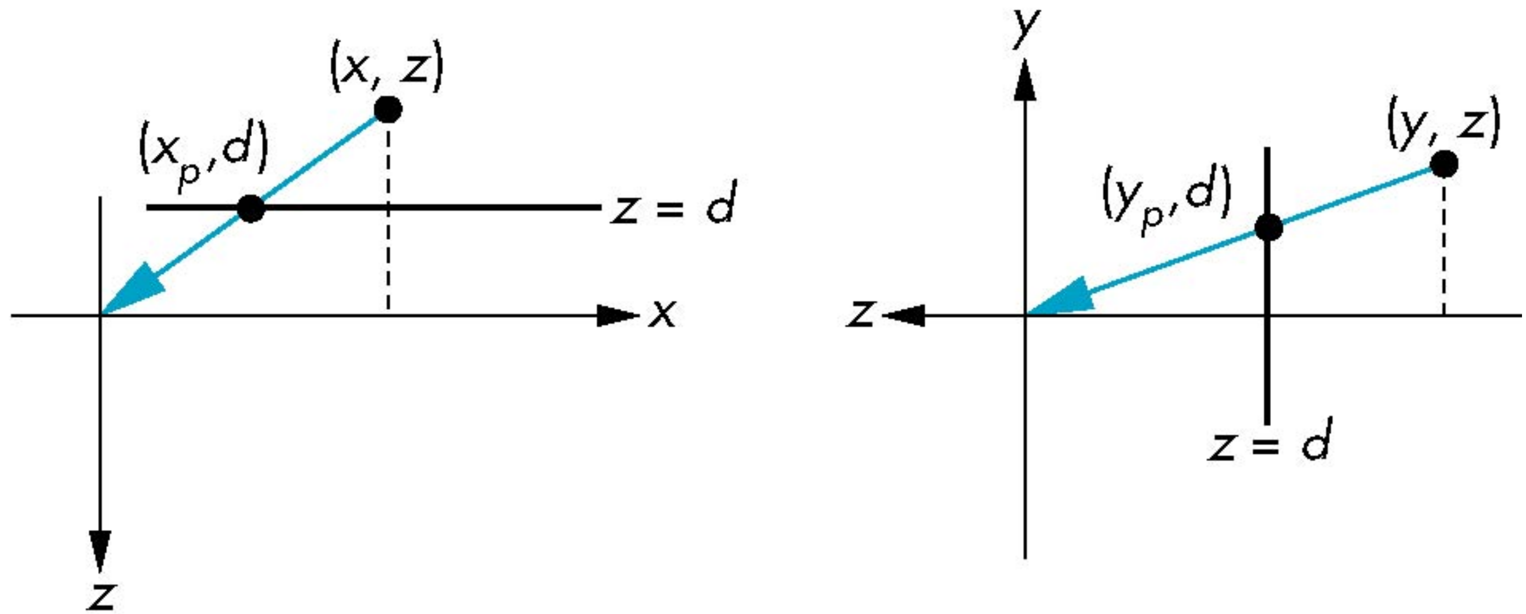
Simple Perspective

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



Perspective Equations

Consider top (*x*-*z* plane) and side (*y*-*z* plane) views



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

Homogeneous Coordinate Form

consider $\mathbf{q} = \mathbf{M}\mathbf{p}$ where $\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$

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

Perspective Division

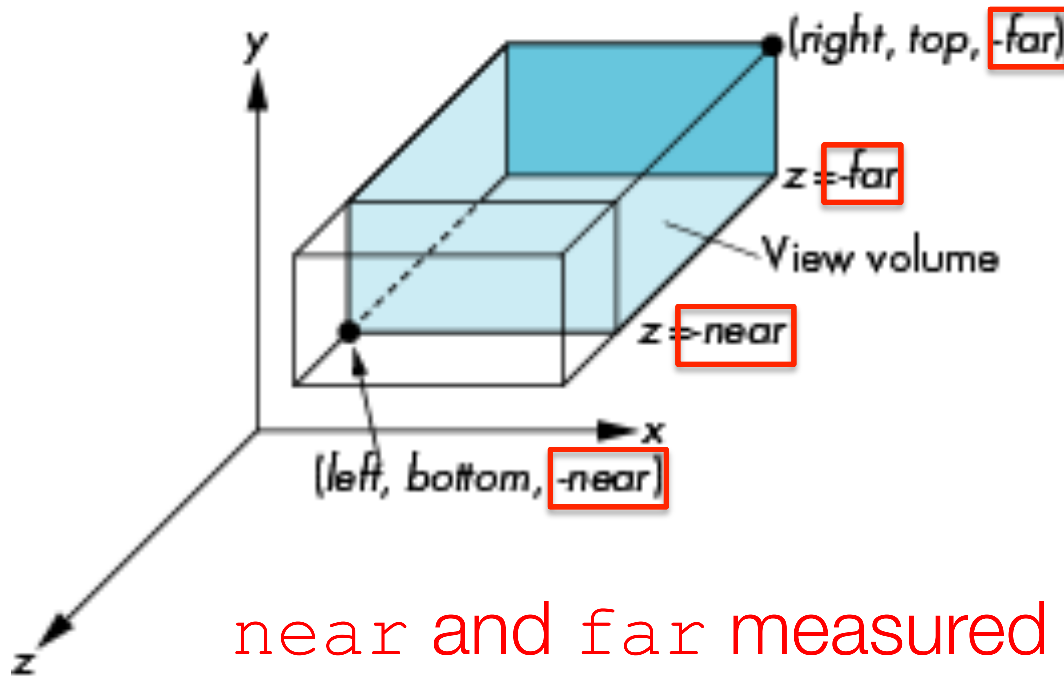
- However $w \neq 1$, so we must divide by w to return from homogeneous coordinates
- This perspective division yields the desired perspective equations

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

- We will consider the corresponding clipping volume with `MV.js` functions

WebGL Orthogonal Viewing

```
ortho(left,right,bottom,top,near,far);
```

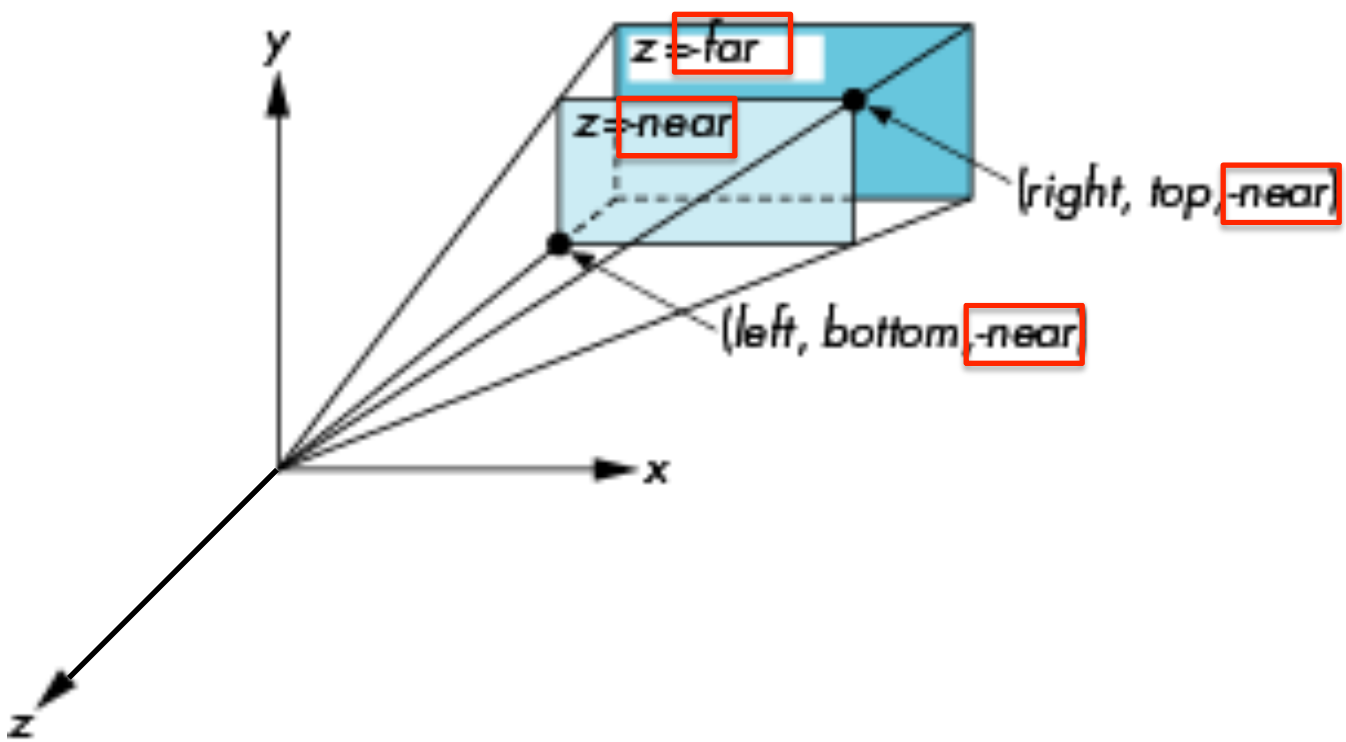


`near` and `far` measured <u>from</u> camera, negative if behind the camera, note that the function does z-flipping!

WebGL Perspective Viewing

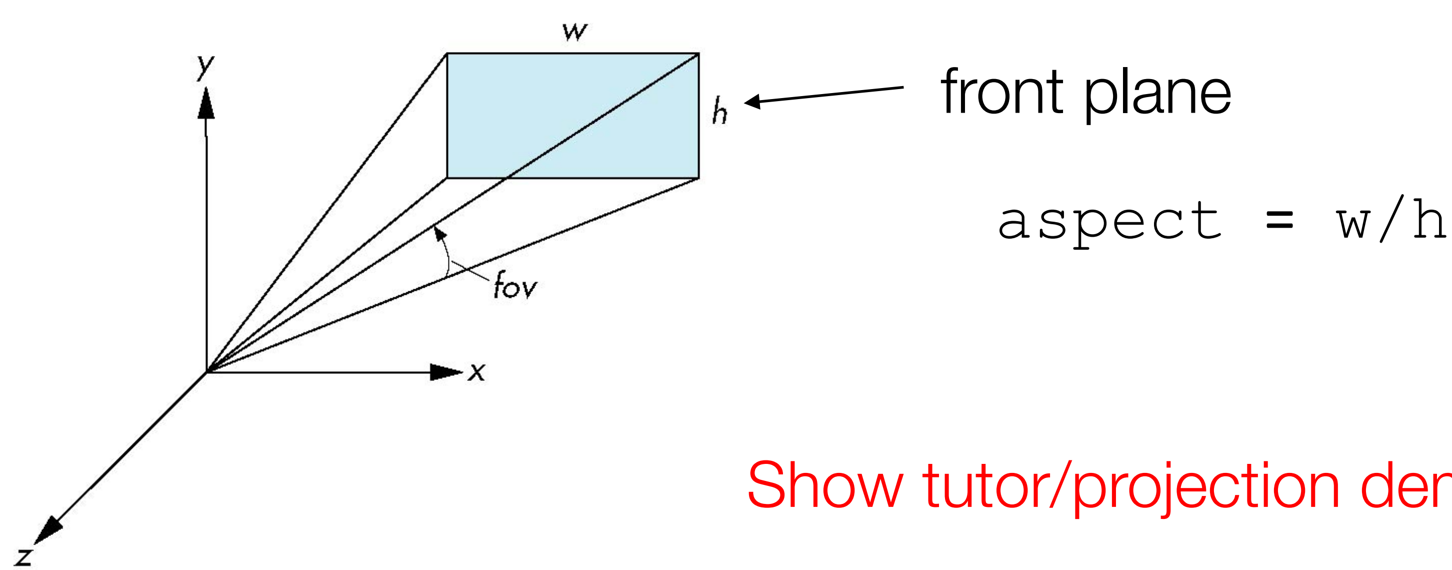
`frustum(left,right,bottom,top,near,far);`

- `near` and `far` must be positive, i.e., the camera is in front of both `near` and `far` planes



Using Field of View

- With `frustum` it is often difficult to get the desired view
- `perpective(fovy, aspect, near, far);`
- Often provides a better interface
  - `fovy` field of view angle in the y direction
  - `near` and `far` must be positive



Show tutor/projection demo

Example – WebGL Code

```
var render = function(){
  gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
   eye = vec3(radius*Math.sin(theta)*Math.cos(phi),
              radius*Math.sin(theta)*Math.sin(phi),
              radius*Math.cos(theta));
  modelViewMatrix = lookAt(eye, at, up);
  projectionMatrix = perspective(fovy, aspect, near,
                                  far);
  gl.uniformMatrix4fv( modelViewMatrixLoc, false,
                       flatten(modelViewMatrix) );
  gl.uniformMatrix4fv( projectionMatrixLoc, false,
                       flatten(projectionMatrix) );
  gl.drawArrays( gl.TRIANGLES, 0, NumVertices );
  requestAnimFrame(render);
}
```

Example – Vertex Shader Code

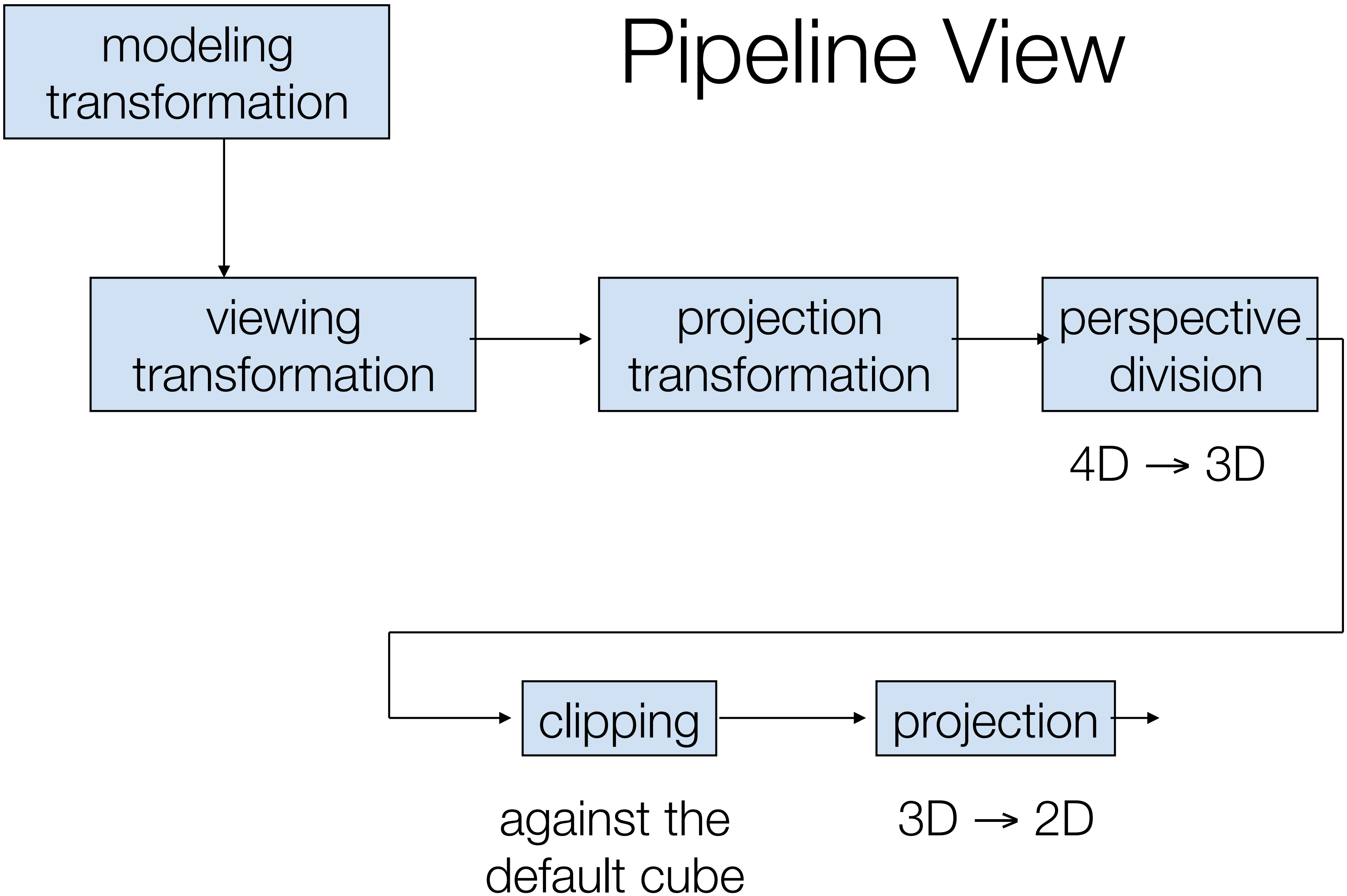
```
attribute vec4 vPosition;
attribute vec4 vColor;
varying vec4 fColor;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main() {
    gl_Position =
        projectionMatrix*modelViewMatrix*vPosition;
    fColor = vColor;
}
```

Projection Matrices

View Normalization

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

Pipeline View



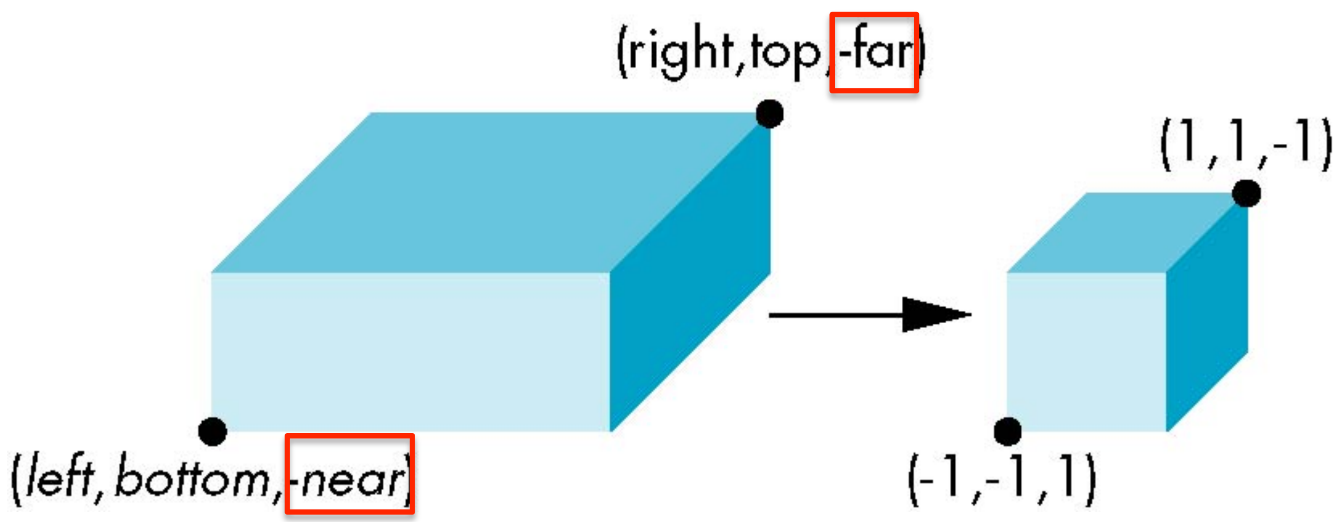
Notes

- We stay in four-dimensional homogeneous coordinates through both the modelview and projection transformations
- Normalization lets us clip against simple cube regardless of type of projection
- Delay final projection until the end (important for hidden-surface removal to retain depth information as long as possible)

Orthogonal Normalization

```
ortho(left,right,bottom,top,near,far);
```

normalization ⇒ find transformation to convert specified clipping volume to default

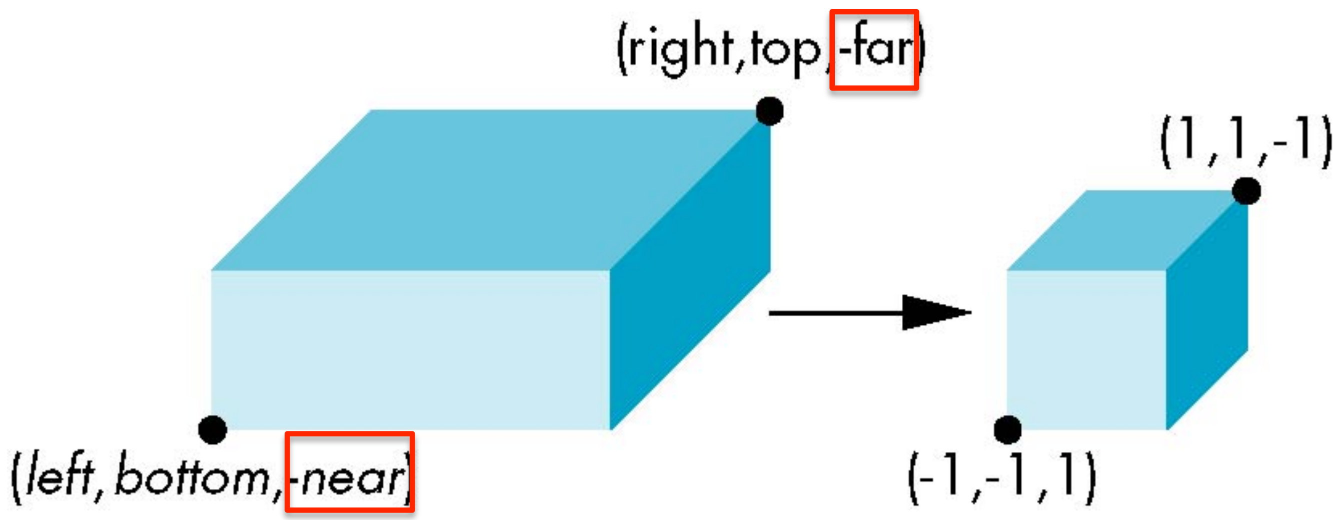


Orthogonal Matrix (1)

- Two steps
  - Move center to origin

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

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



Orthogonal Matrix (2)

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

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

Check implementation in `MV.js`

Orthogonal Matrix (3)

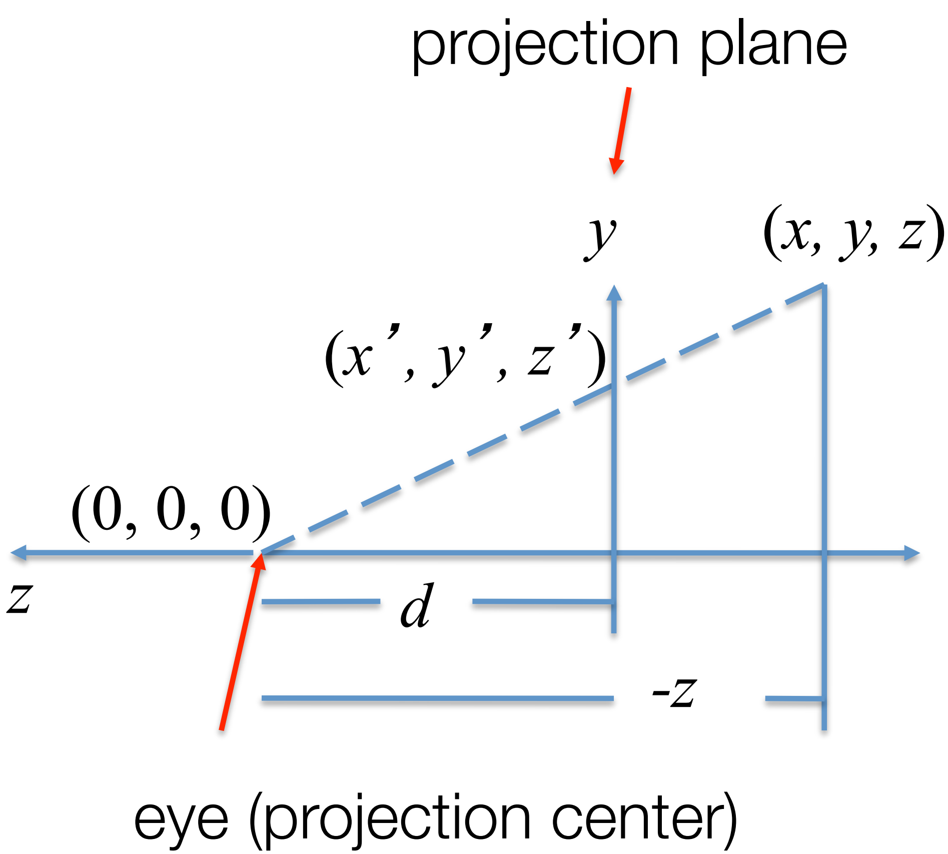
$$\mathbf{P} = \mathbf{ST} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & 0 \\ 0 & \frac{2}{top-bottom} & 0 & 0 \\ 0 & 0 & -\frac{2}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{right+left}{2} \\ 0 & 1 & 0 & -\frac{top+bottom}{2} \\ 0 & 0 & 1 & \frac{far+near}{2}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

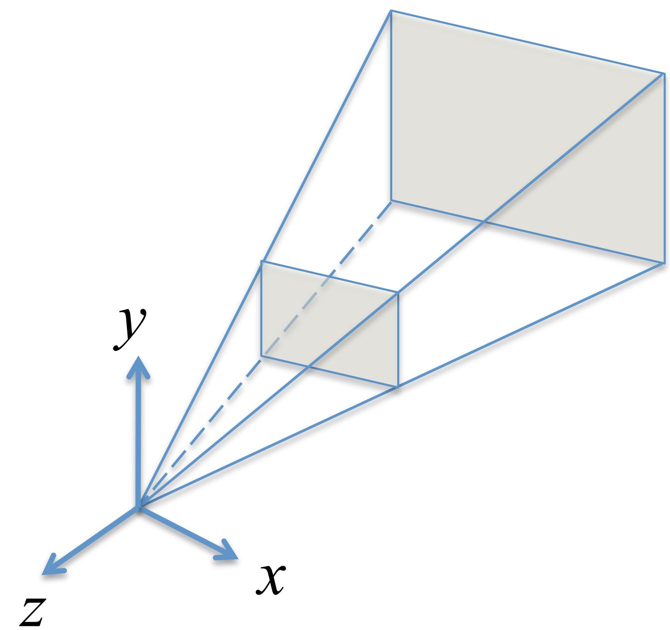
$$= \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Verify for ortho(-1, 1, -1, 1, -1, 1)

Perspective Projection (1)

- Side view





$$y/y' = -z/d$$

$$y' = y \times d/-z$$

Perspective Projection (2)

- Same for x. So we have:

  $x' = x \times d/-z$

  $y' = y \times d/-z$

  $z' = -d$

- Put into a matrix form:

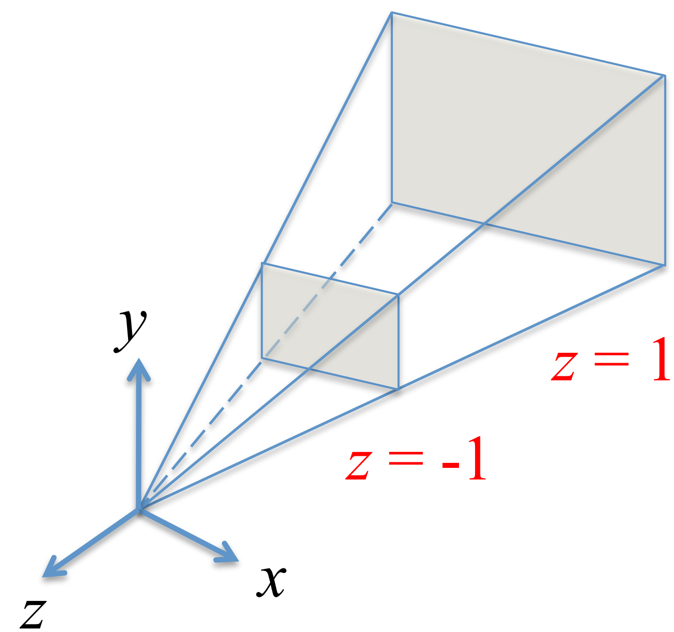
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & (1/-d) & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- OpenGL assume $d = 1$, i.e., the image plane is at $z = -1$

Perspective Projection (3)

- We are not done yet. We want to somewhat keep the z information so that we can perform depth comparison
- Use pseudo depth – OpenGL maps the near plane to -1, and far plane to +1
- Need to modify the projection matrix: solve a and b

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \boxed{a} & \boxed{b} \\ 0 & 0 & (1/-d) & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



How to solve a and b?

Perspective Projection (4)

- Solve a and b

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & (1/-d) & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\underset{\mathbf{M}}{\Uparrow}$$

- $d = 1$
- $(0, 0, -1)^T = \mathbf{M}\ (0, 0, -\text{near})^T$
- $(0, 0, +1)^T = \mathbf{M}\ (0, 0, -\text{far})^T$

- a = -(far+near) / (far-near)
- b = (-2 x far x near) / (far-near)

Verify this!

- $\boxed{(0, 0, -1)^T} = \mathbf{M}\,\boxed{(0, 0, -\text{near})^T}$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \boxed{\begin{bmatrix} 0 \\ 0 \\ -near \\ 1 \end{bmatrix}} = \boxed{\begin{bmatrix} 0 \\ 0 \\ -a \times near + b \\ near \end{bmatrix}}$$

$$\frac{-a \times near + b}{near} = -1$$

Verify this!

- $\boxed{(0,\ 0,\ +1)^{T}} = \mathbf{M}\ \boxed{(0,\ 0,\ \text{-far})^{T}}$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \boxed{\begin{bmatrix} 0 \\ 0 \\ -far \\ 1 \end{bmatrix}} = \boxed{\begin{bmatrix} 0 \\ 0 \\ -a \times far + b \\ far \end{bmatrix}}$$

$$\frac{-a \times far + b}{far} = +1$$

Verify this!

$$\frac{-a \times near + b}{near} = -1$$

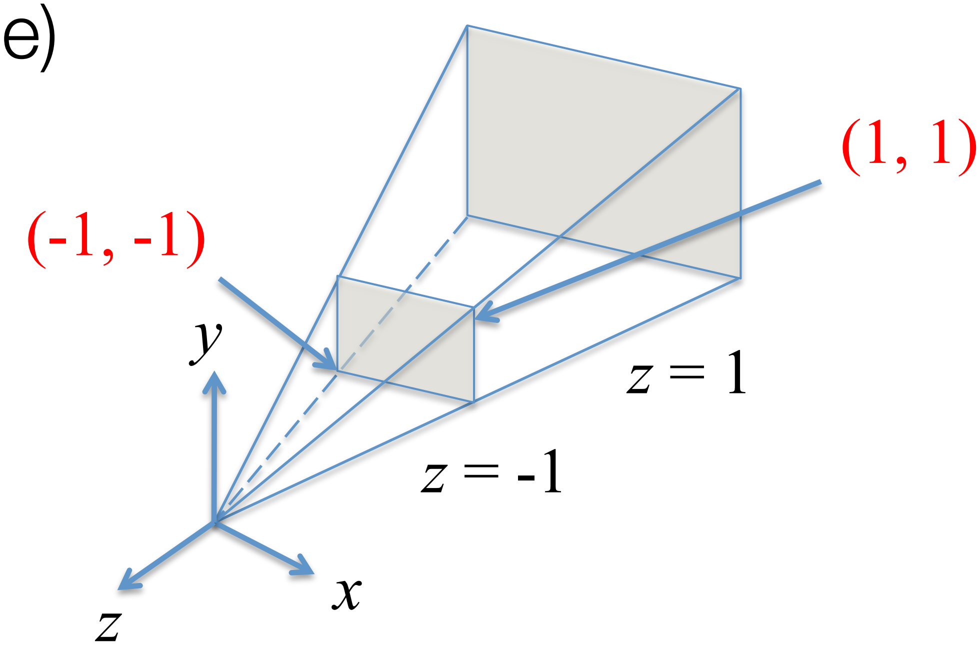
$$\frac{-a \times far + b}{far} = +1$$

$$a = \frac{-(far + near)}{far - near}$$

$$b = \frac{-(2 \times far \times near)}{far - near}$$

Perspective Projection (5)

- Not done yet. OpenGL also normalizes the x and y ranges of the viewing frustum to [-1, 1] (translate and scale)



- And takes care the case that eye is not at the center of the view volume (shear)

OpenGL Perspective Matrix

- The normalization in `frustum` requires an initial shear to form a right viewing pyramid, followed by a scaling to get the normalized perspective volume. Finally, the perspective matrix results in needing only a final orthogonal transformation

$$\mathbf{P} = \mathbf{M}\mathbf{S}\mathbf{H}$$

our previously defined perspective matrix

shear and scale

Final Projection Matrix

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2 \times near}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2 \times near}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & \frac{-(far + near)}{far - near} & \frac{-2 \times far \times near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

`frustum(left,right,bottom,top,near,far);`

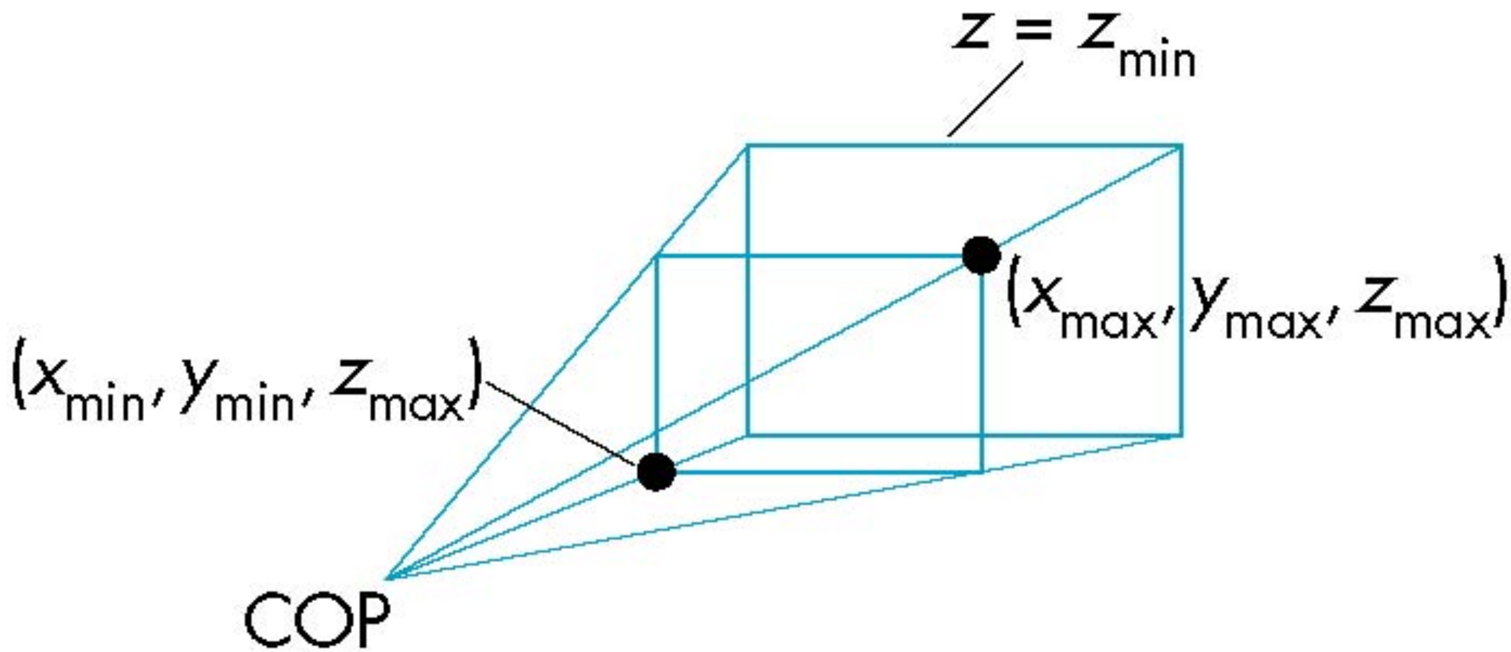
`near` and `far` must be positive

- After perspective projection, the viewing frustum is also project into a canonical view volume (like in orthogonal projection)

Check implementation in `MV.js`

WebGL Perspective

- `frustum` allows for an unsymmetric viewing frustum (although `perspective` does not)



Why do we do it this way?

- Normalization allows for a single pipeline for both perspective and orthogonal viewing
- We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
- We simplify clipping