Week 06:
Viewing and Projection
Part 2

CS-537: Interactive Computer Graphics
Dr. Chloe LeGendre
Department of Computer Science

Some materials from the companion slides of Angel and Shreiner, “Interactive Computer Graphics, A Top-Down Approach with WebGL.”

Objectives

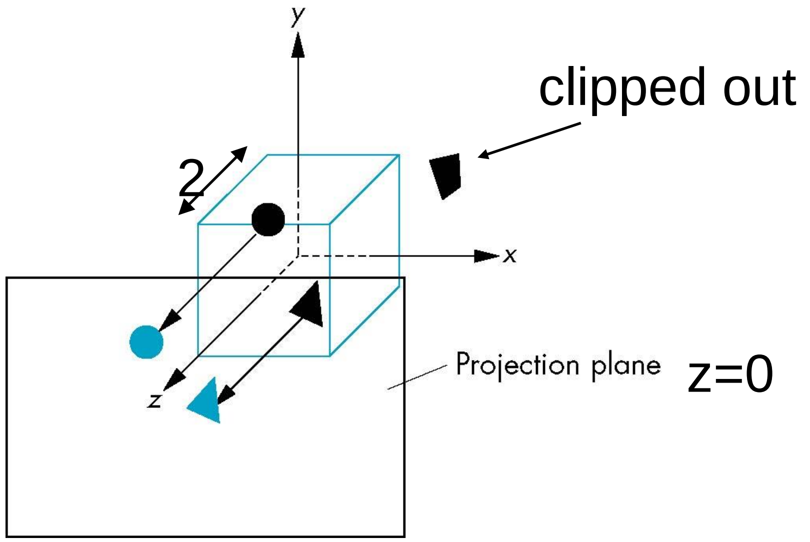
- Introduce computer viewing and projection
- Introduce mathematics of projection
- Describe WebGL viewing and projection functions in class library MV.js
- Introduce projection normalization

Computer Viewing

- There are three aspects of the viewing process, all of which are implemented in the computer graphics pipeline:
  - Positioning the camera
    - Setting the model-view matrix
  - Selecting a lens
    - Setting the projection matrix
  - Clipping
    - Setting the view volume (anything outside will not be rendered)

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
- WebGL 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
  - The default projection is orthographic



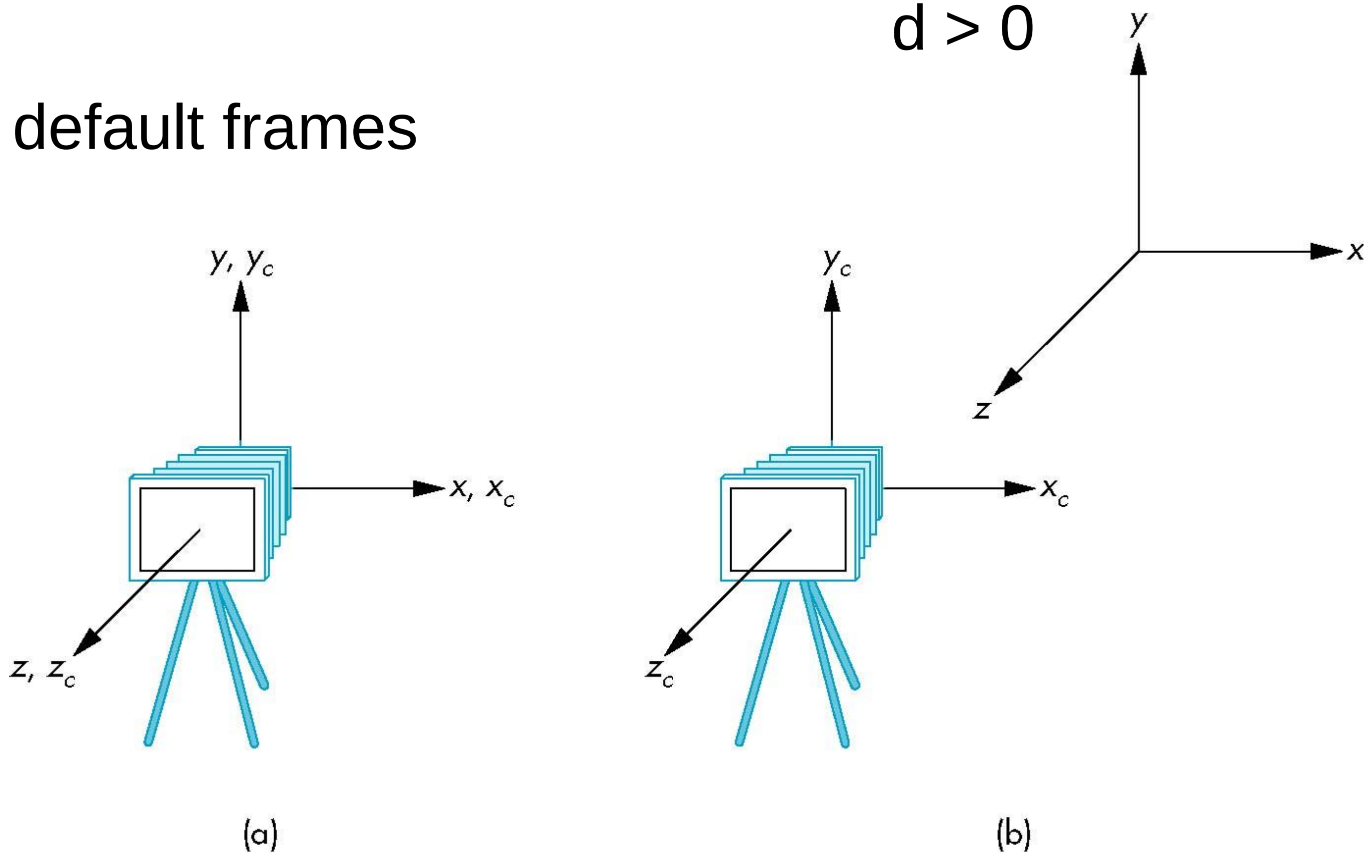
Moving the Camera Frame

- If we want to visualize objects 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 views are equivalent and are determined by the **model-view matrix**
  - Want a translation (`translate(0.0,0.0,-d);`) using the MV.js function
  - `d > 0`

Example: Moving Camera Back from Origin

frames after translation by –d
d > 0

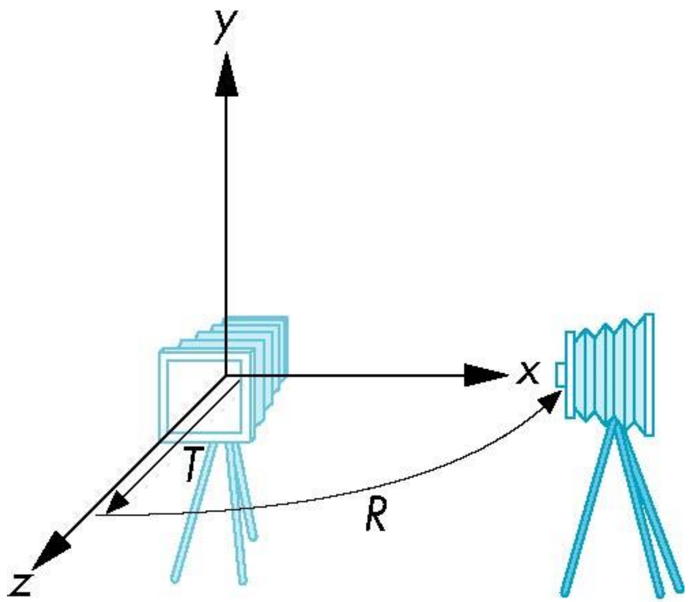
default frames



CS-537: Interactive Computer Graphics

Moving the Camera Frame (II)

- 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 C = TR



WebGL Code for Moving the Camera Frame

- 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););
```

Forming the model-view matrix with a built-in function

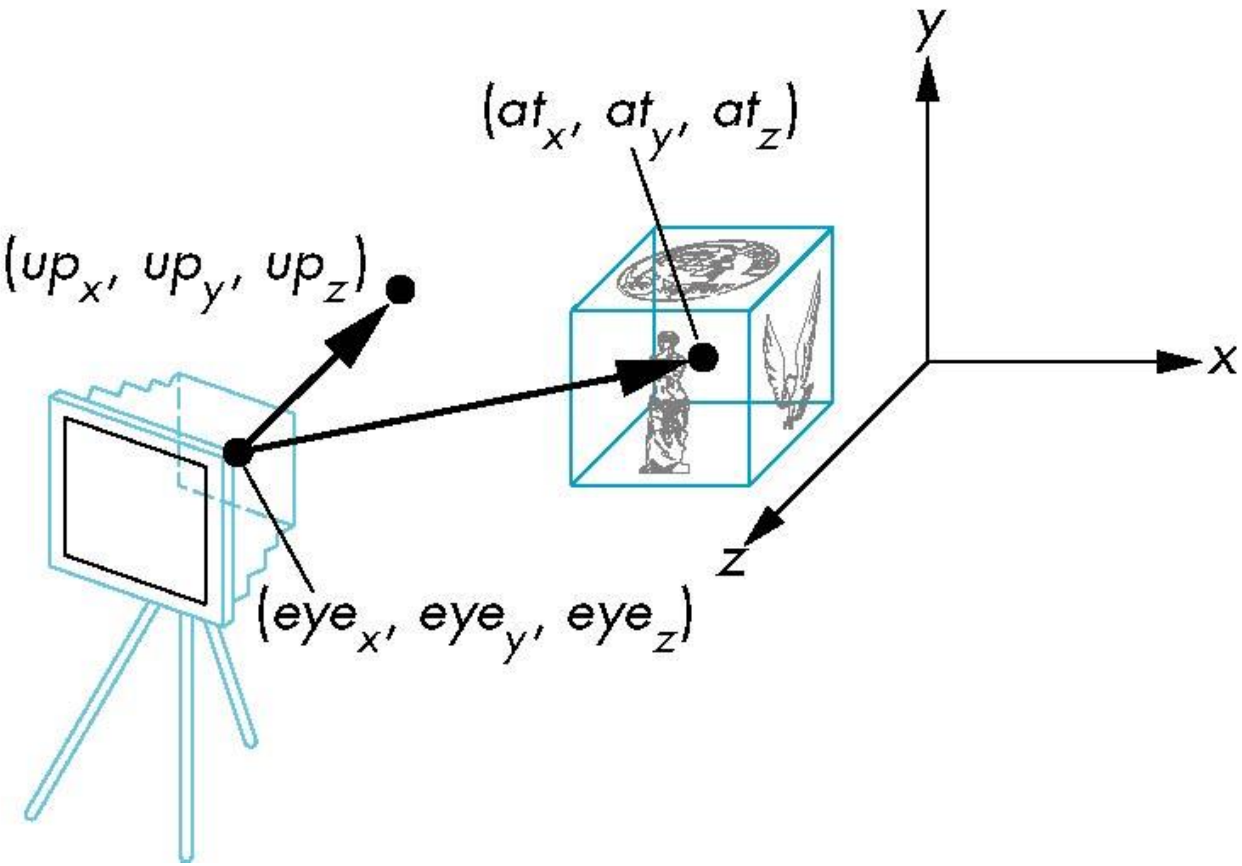
- Some graphics libraries contain the function `lookAt(eye, at, up)` to form the required model-view matrix through a simple interface
- Note the need for setting an "up" direction (see next slide)
- Implemented as lookAt(…) in MV.js
  - Can concatenate with modeling transformations
- Example: isometric 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);
```

- Other viewing APIs exist for setting the model-view matrix, but we'll use this one

The lookAt Function (in MV.js)

```
lookAt(eye, at, up)
```



Eye = point that is camera center of projection

At = point that the camera is looking at

Up = direction vector that orients what is "up" in the final image

Projections and Normalization

- The default projection in the eye (camera) frame is orthographic
- For a point (x,y,z) within the default view volume, the projection is:

$$x_p = x$$

$$y_p = y$$

$$z_p = 0$$

- 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

Homogenous Coordinate Representation

- Default orthographic projection:

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

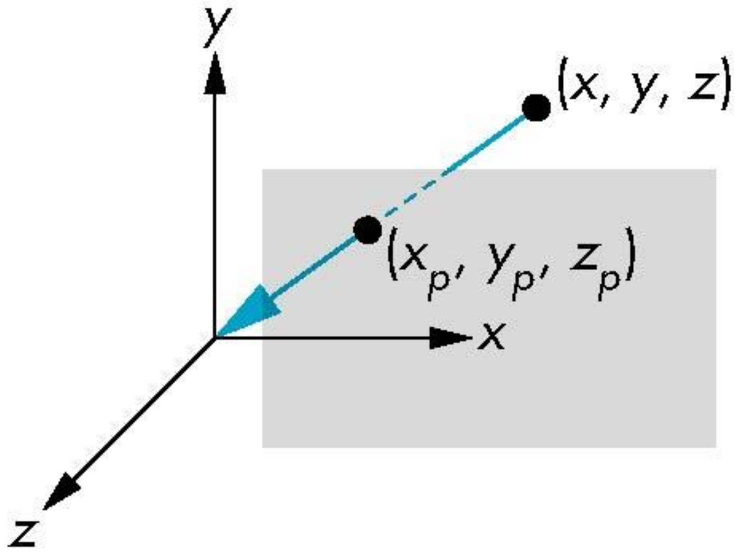
$$\mathbf{p}_p = \mathbf{M}\mathbf{p}$$

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

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

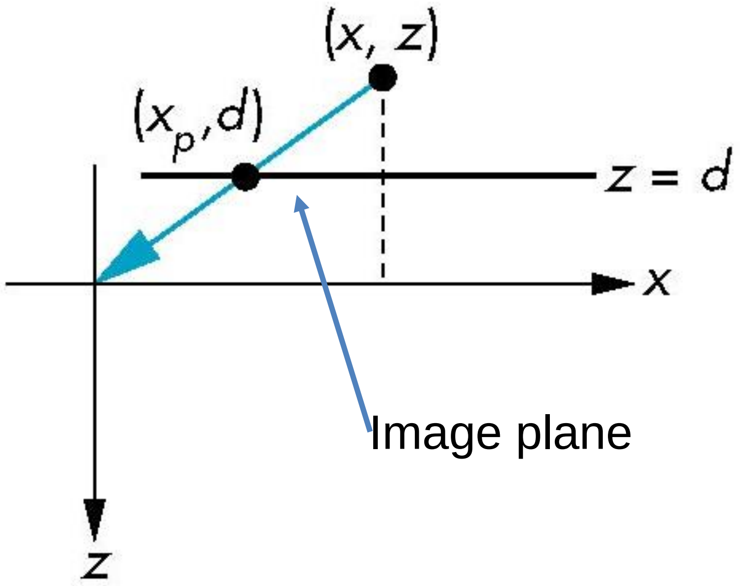
Simple Perspective

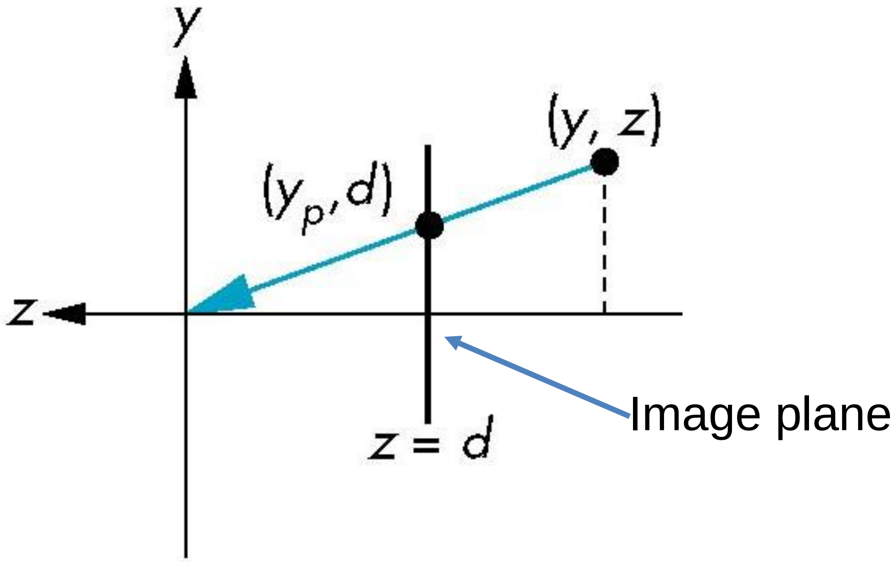
- Center of projection at the origin
- Projection plane $z = d$, $d < 0$



Perspective Equations

- Consider top and side views





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

Homogenous Coordinate Form

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

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

Perspective Division

- However, the last row element $w \neq 1$, so we must divide all rows by $w = (z/d)$ to return from homogeneous coordinates
- This *perspective division* yields

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

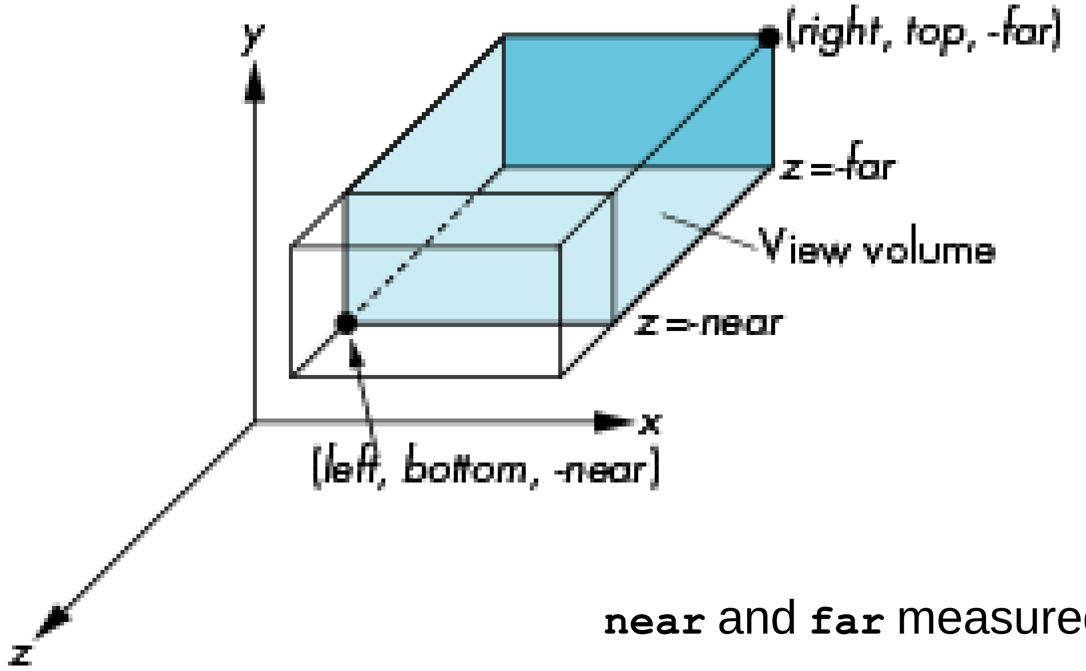
(the desired perspective equations)

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

WebGL Orthographic Viewing

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

(function in Common/MV.js)

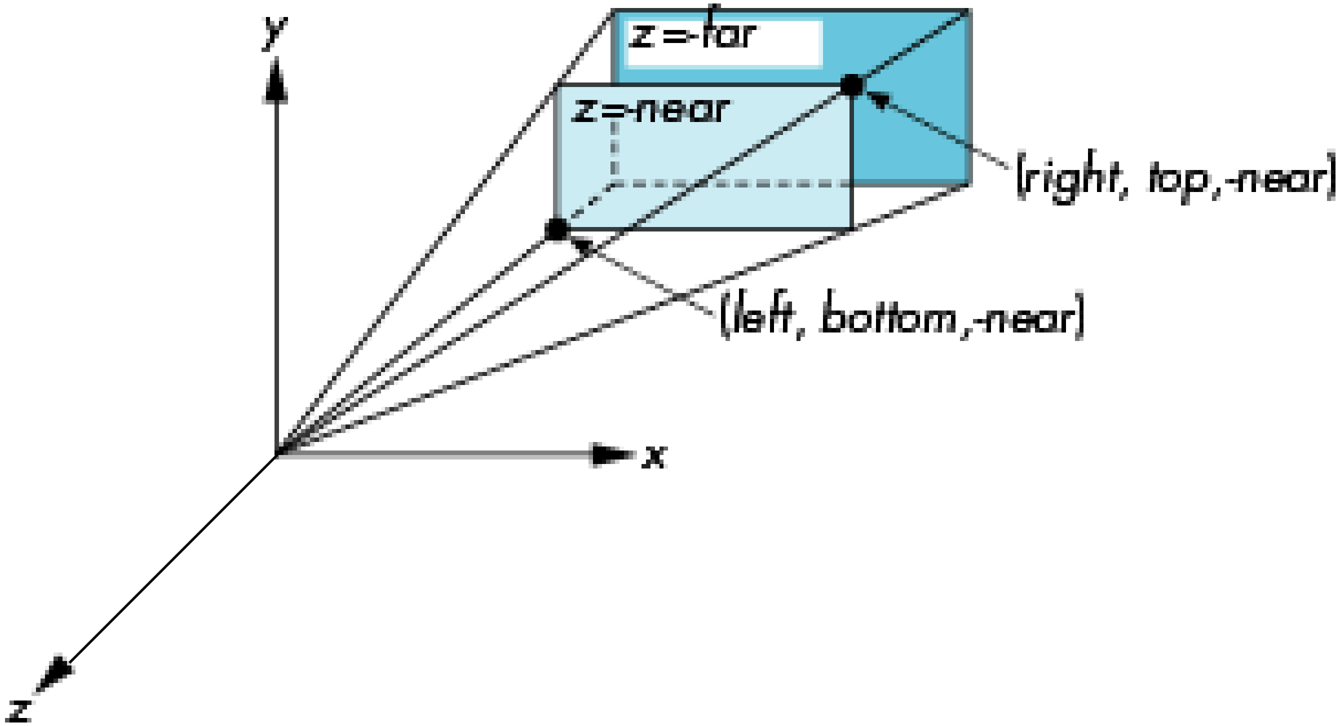


`near` and **`far`** measured <u>from</u> camera

WebGL Perspective Viewing

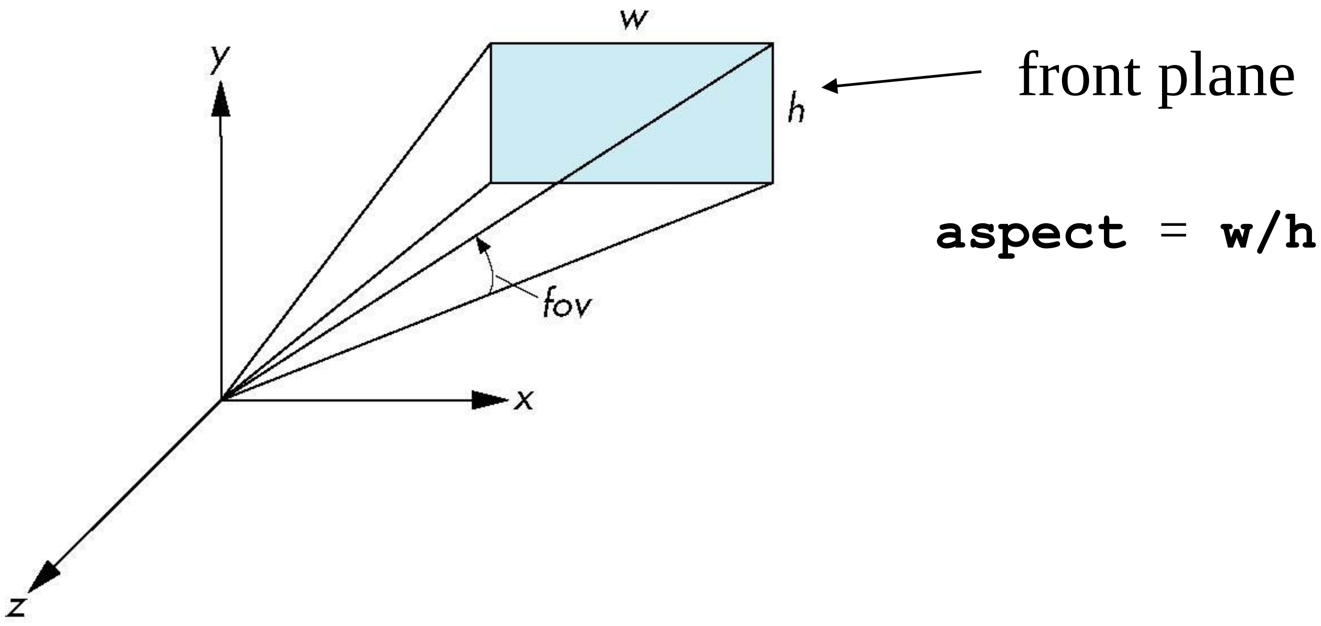
```
frustum(left,right,bottom,top,near,far)
```

(function in Common/MV.js)



Using Field of View

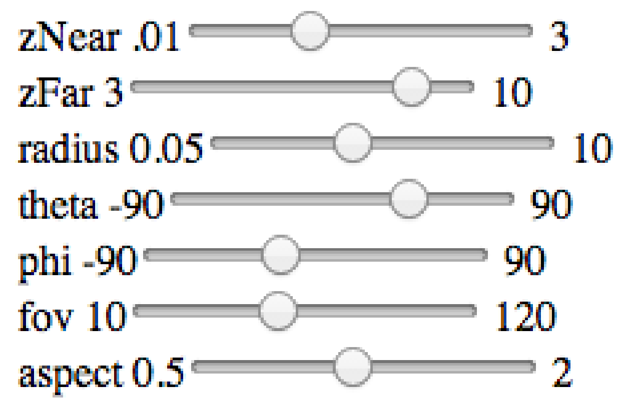
- With **`frustum(…)`** it is often difficult to get the desired view
- **`perpective(fovy, aspect, near, far)`** in MV.js often provides a better interface. (fovy = Field of View, Y direction)

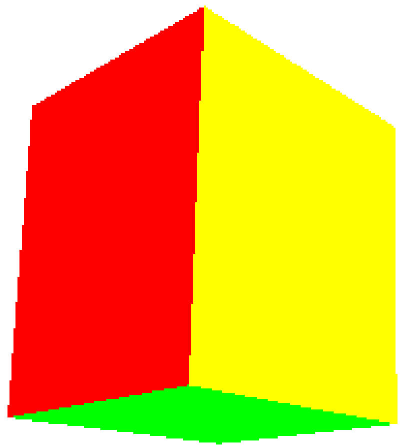


`aspect = w/h`

Computing Projection Matrices

- Compute in JavaScript file, send to vertex shader with gl.uniformMatrix4fv
- Dynamic: update in render() or within shader
- Exercise: Check examples in Ch 5 programming examples!





Example render() function

```
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( modelViewMatrix, false,
          flatten(modelViewMatrix) );
    gl.uniformMatrix4fv( projectionMatrixLoc, false,
         flatten(projectionMatrix) );
    gl.drawArrays( gl.TRIANGLES, 0, NumVertices );
    requestAnimFrame(render);
}
```

function from Common/MV.js

Example corresponding vertex shader

```
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 Normalization

- Rather than derive a different projection matrix for each type of projection (parallel or perspective), 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

model-view transformation → projection transformation → perspective division → clipping → projection

perspective division: $4D \rightarrow 3D$

clipping: against default cube

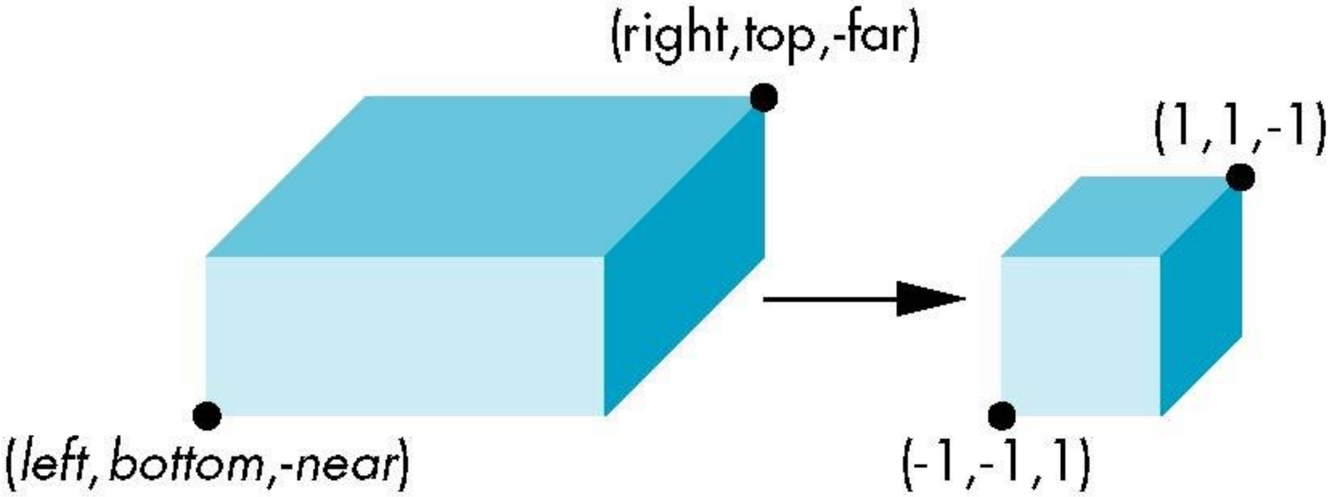
projection: $3D \rightarrow 2D$

- We stay in four-dimensional homogeneous coordinates through both the model-view and projection transformations
  - Both these transformations are nonsingular
  - Default to identity matrices (orthographic view)
- **Normalization** lets us clip against simple cube regardless of type of projection
- Delay final projection until end
  - Important for hidden-surface removal to retain depth information for as long as possible

Orthographic Normalization

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

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



- A similar process applies for perspective normalization