Introduction to Computer Graphics with WebGL

Week6

Instructor: Hooman Salamat

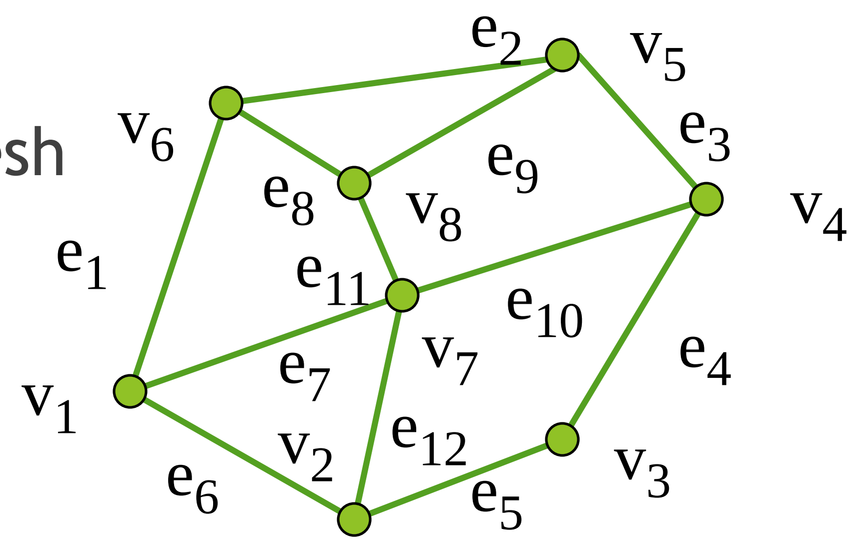
Building Models

Objectives

- Introduce simple data structures for building polygonal models
  - Vertex lists
  - Edge lists

Representing a Mesh

- Consider a mesh



- There are 8 nodes and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges
- Each vertex has a location $v_i = (x_i\ y_i\ z_i)$

Simple Representation

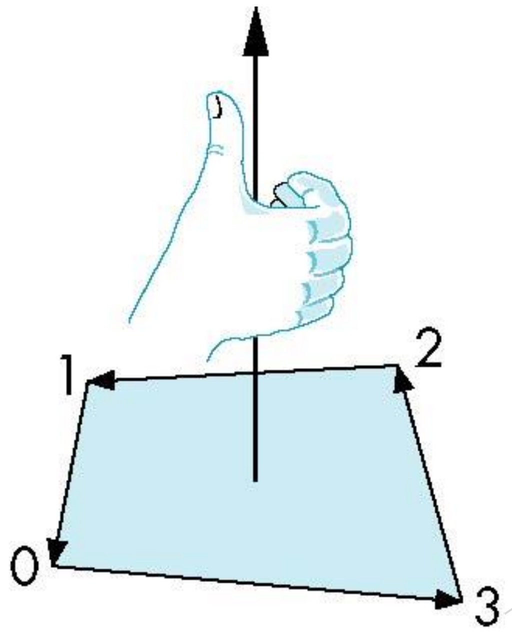
- Define each polygon by the geometric locations of its vertices
- Leads to WebGL code such as

```
vertex.push(vec3(x1, y1, z1));
vertex.push(vec3(x6, y6, z6));
vertex.push(vec3(x7, y7, z7));
```

- Inefficient and unstructured
  - Consider moving a vertex to a new location
  - Must search for all occurrences

Inward and Outward Facing Polygons

- The order $\{v_1, v_6, v_7\}$ and $\{v_6, v_7, v_1\}$ are equivalent in that the same polygon will be rendered by OpenGL but the order $\{v_1, v_7, v_6\}$ is different
- The first two describe *outwardly facing* polygons
- Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal
- OpenGL can treat inward and outward facing polygons differently

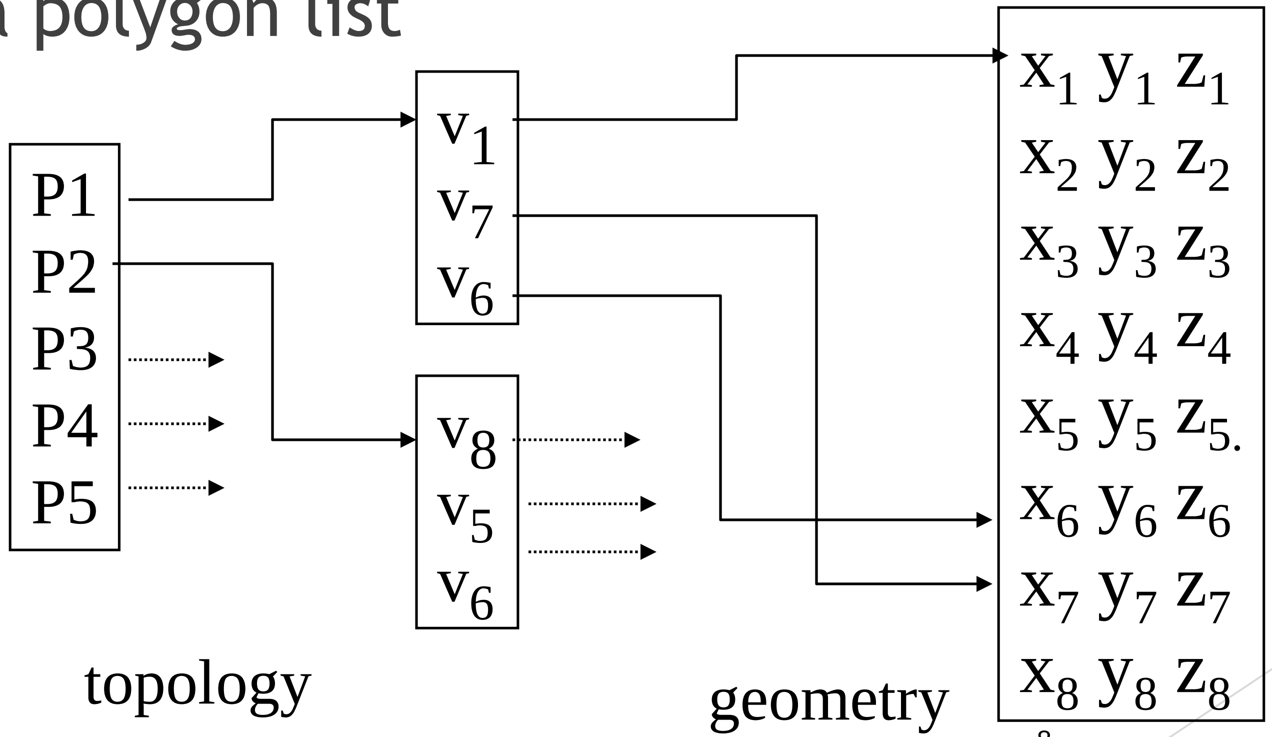


Geometry vs Topology

- Generally it is a good idea to look for data structures that separate the geometry from the topology
  - Geometry: locations of the vertices
  - Topology: organization of the vertices and edges
  - Example: a polygon is an ordered list of vertices with an edge connecting successive pairs of vertices and the last to the first
  - Topology holds even if geometry changes

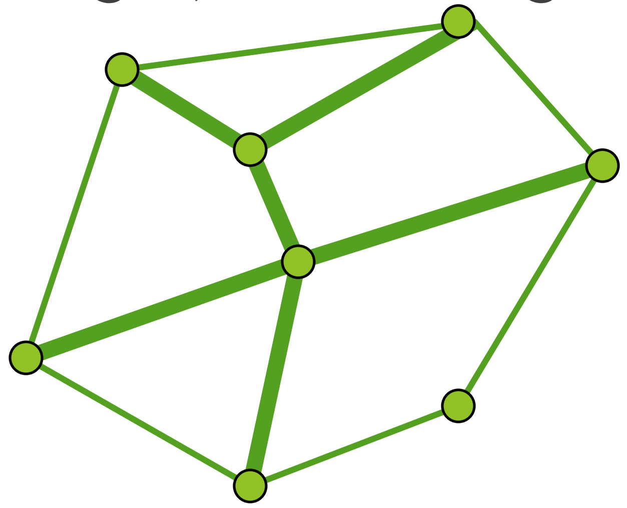
Vertex Lists

- Put the geometry in an array
- Use pointers from the vertices into this array
- Introduce a polygon list



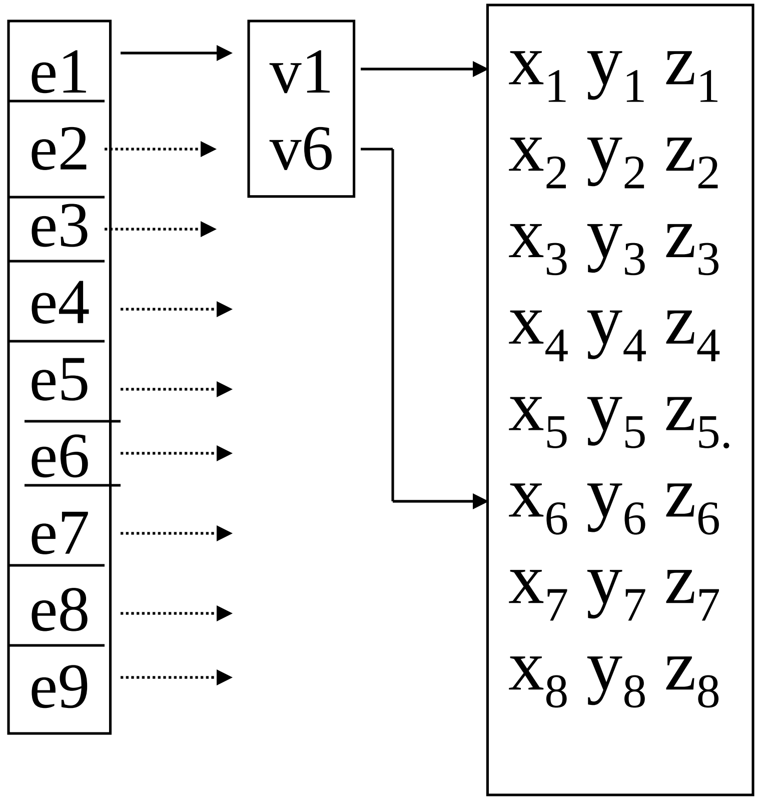
Shared Edges

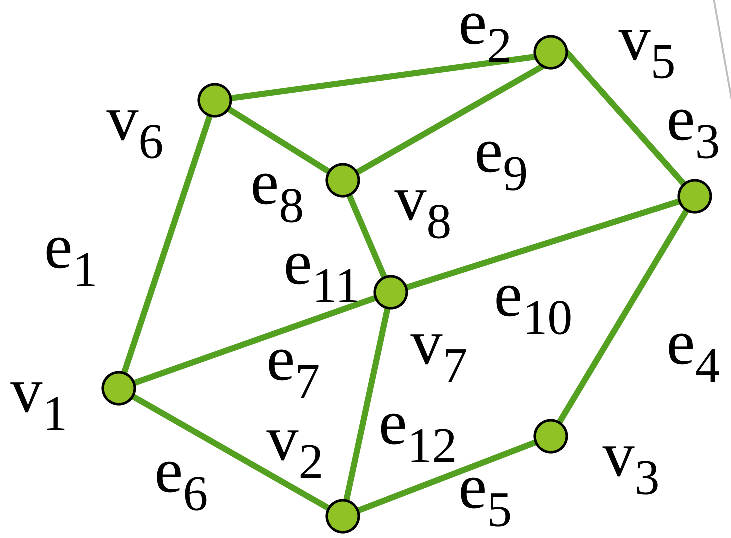
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- Can store mesh by *edge list*

Edge List

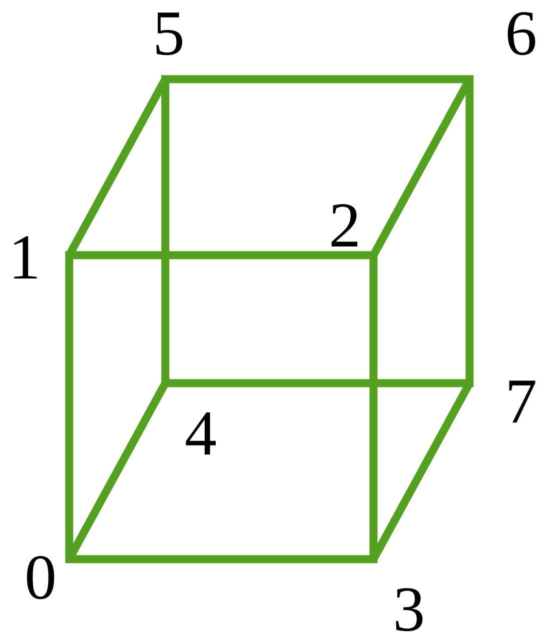




Note polygons are not represented

Draw cube from faces

```
var colorCube( )
{
    quad(0,3,2,1);
    quad(2,3,7,6);
    quad(0,4,7,3);
    quad(1,2,6,5);
    quad(4,5,6,7);
    quad(0,1,5,4);
}
```



The Rotating Cube

Objectives

- Put everything together to display rotating cube
- Two methods of display
  - by arrays
  - by elements

Modeling a Cube

Define global array for vertices

```
var vertices = [
        vec3( -0.5, -0.5,  0.5 ),
        vec3( -0.5,  0.5,  0.5 ),
        vec3(  0.5,  0.5,  0.5 ),
        vec3(  0.5, -0.5,  0.5 ),
        vec3( -0.5, -0.5, -0.5 ),
        vec3( -0.5,  0.5, -0.5 ),
        vec3(  0.5,  0.5, -0.5 ),
        vec3(  0.5, -0.5, -0.5 )
    ];
```

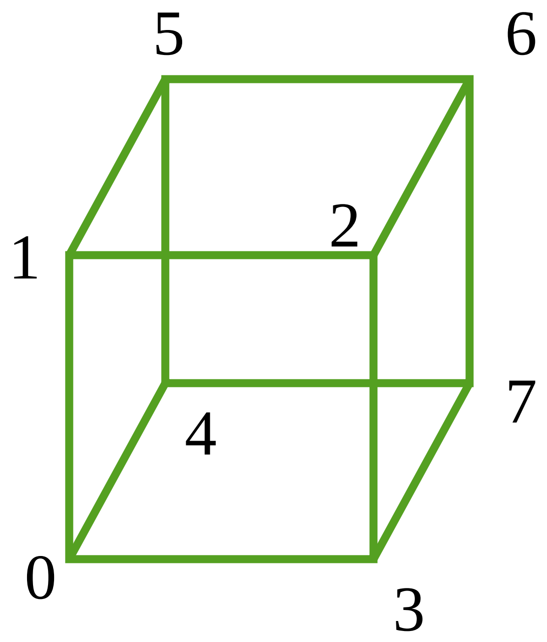
Colors

Define global array for colors

```
var vertexColors = [
        [ 0.0, 0.0, 0.0, 1.0 ],  // black
        [ 1.0, 0.0, 0.0, 1.0 ],  // red
        [ 1.0, 1.0, 0.0, 1.0 ],  // yellow
        [ 0.0, 1.0, 0.0, 1.0 ],  // green
        [ 0.0, 0.0, 1.0, 1.0 ],  // blue
        [ 1.0, 0.0, 1.0, 1.0 ],  // magenta
        [ 0.0, 1.0, 1.0, 1.0 ],  // cyan
        [ 1.0, 1.0, 1.0, 1.0 ]   // white
    ];
```

Draw cube from faces

```
function colorCube( )
{
    quad(0,3,2,1);
    quad(2,3,7,6);
    quad(0,4,7,3);
    quad(1,2,6,5);
    quad(4,5,6,7);
    quad(0,1,5,4);
}
```



Note that vertices are ordered so that
we obtain correct outward facing normals
Each quad generates two triangles

Initialization

```
var canvas, gl;
var numVertices  = 36;
var points = [];
var colors = [];

window.onload = function init(){
    canvas = document.getElementById( "gl-canvas" );
    gl = WebGLUtils.setupWebGL( canvas );

    colorCube();

   gl.viewport( 0, 0, canvas.width, canvas.height );
   gl.clearColor( 1.0, 1.0, 1.0, 1.0 );
   gl.enable(gl.DEPTH_TEST);

// rest of initialization and html file
// same as previous examples
```

The quad Function

Put position and color data for two triangles from a list of indices into the array **`vertices`**

```
var quad(a, b, c, d)
{
   var indices = [ a, b, c, a, c, d ];
   for ( var i = 0; i < indices.length; ++i ) {

      points.push( vertices[indices[i]]);
      colors.push( vertexColors[indices[i]] );

// for solid colored faces use
//colors.push(vertexColors[a]);

 }
}
```

Render Function

```
function render(){
    gl.clear( gl.COLOR_BUFFER_BIT |gl.DEPTH_BUFFER_BIT);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );
}
```

Mapping indices to faces

```
var indices = [
1,0,3,
3,2,1,
2,3,7,
7,6,2,
3,0,4,
4,7,3,
6,5,1,
1,2,6,
4,5,6,
6,7,4,
5,4,0,
0,1,5
];
```

Rendering by Elements

- Send indices to GPU

```
var iBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, iBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
       new Uint8Array(indices), gl.STATIC_DRAW);
```

- Render by elements

```
gl.drawElements( gl.TRIANGLES, numVertices,
    gl.UNSIGNED_BYTE, 0 );
```

- Even more efficient if we use triangle strips or triangle fans

Adding Buttons for Rotation

```
var xAxis = 0;
var yAxis = 1;
var zAxis = 2;
var axis = 0;
var theta = [ 0, 0, 0 ];
var thetaLoc;

document.getElementById( "xButton" ).onclick =
function () {          axis = xAxis;     };
document.getElementById( "yButton" ).onclick =
function () {          axis = yAxis;     };
document.getElementById( "zButton" ).onclick =
function () {          axis = zAxis;     };
```

Render Function

```
function render(){
    gl.clear( gl.COLOR_BUFFER_BIT |gl.DEPTH_BUFFER_BIT);
    theta[axis] += 2.0;
    gl.uniform3fv(thetaLoc, theta);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );
}
```

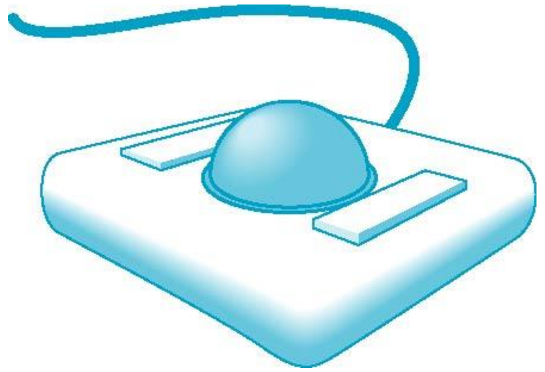
The Virtual Trackball

Objectives

- Introduces the use of graphical (virtual) devices that can be created using WebGL
- Reinforce the benefit of not using direction angles and Euler angles
- Makes use of transformations
- Leads to reusable code that will be helpful later

Physical Trackball

- The trackball is an “upside down” mouse

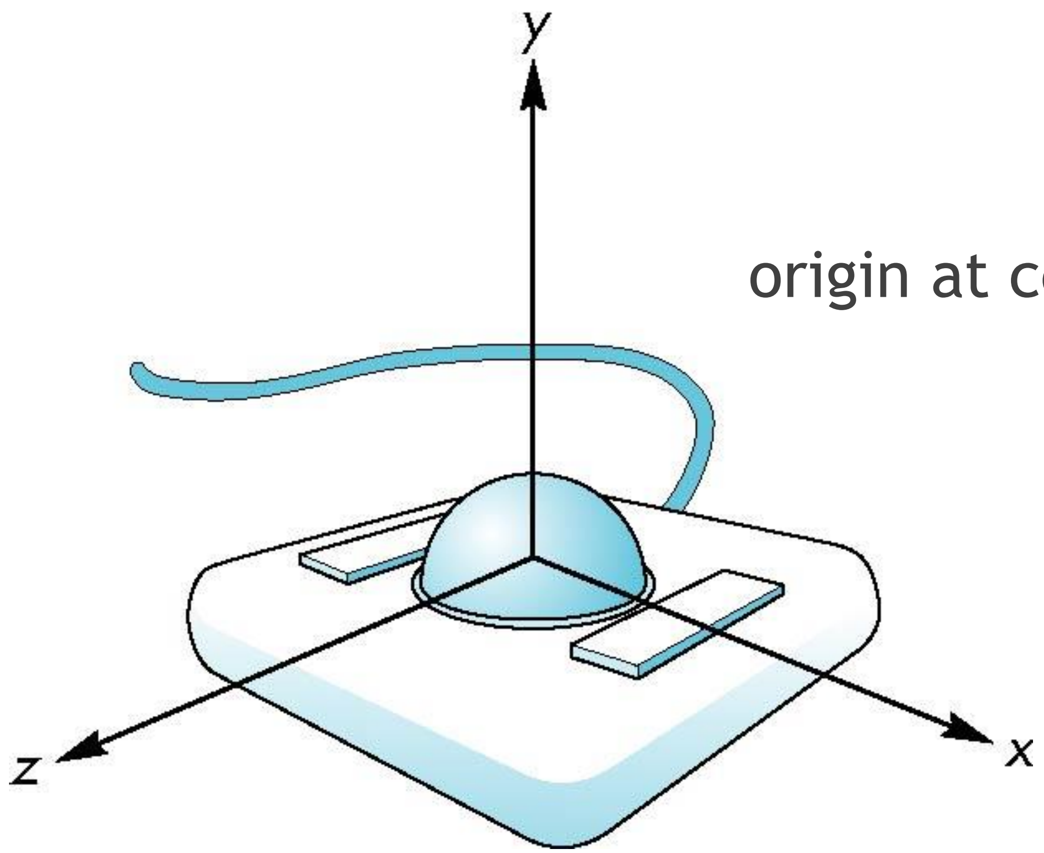


- If there is little friction between the ball and the rollers, we can give the ball a push and it will keep rolling yielding continuous changes
- Two possible modes of operation
  - Continuous pushing or tracking hand motion
  - Spinning

A Trackball from a Mouse

- Problem: we want to get the two behavior modes from a mouse
- We would also like the mouse to emulate a frictionless (ideal) trackball
- Solve in two steps
  - Map trackball position to mouse position
  - Use event listeners to handle the proper modes

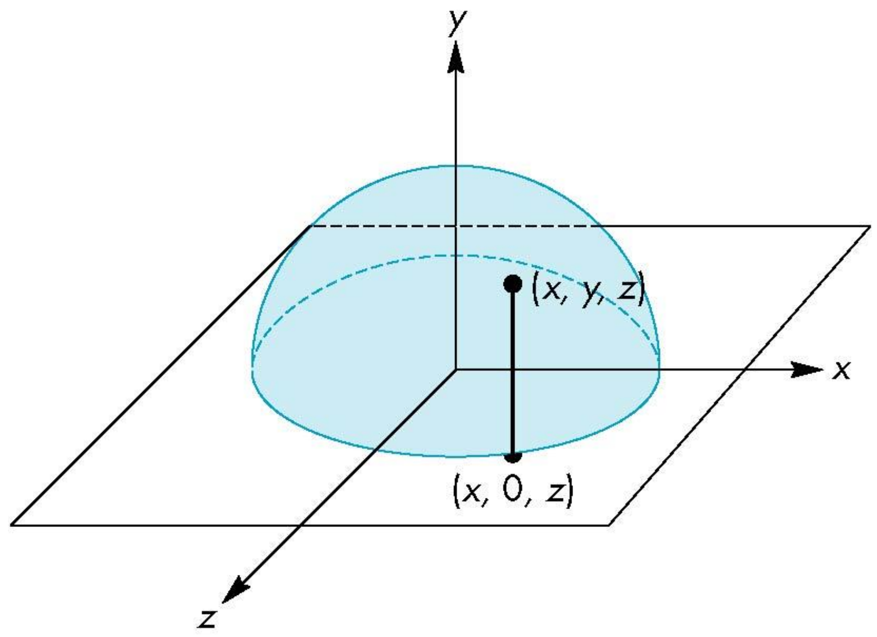
Trackball Frame



origin at center of ball

Projection of Trackball Position

- We can relate position on trackball to position on a normalized mouse pad by projecting orthogonally onto pad



Reversing Projection

- Because both the pad and the upper hemisphere of the ball are two-dimensional surfaces, we can reverse the projection
- A point (x,z) on the mouse pad corresponds to the point (x,y,z) on the upper hemisphere where

$$y = \sqrt{r^2 - x^2 - z^2} \qquad \text{if } r \geq |x| \geq 0,\ r \geq |z| \geq 0$$

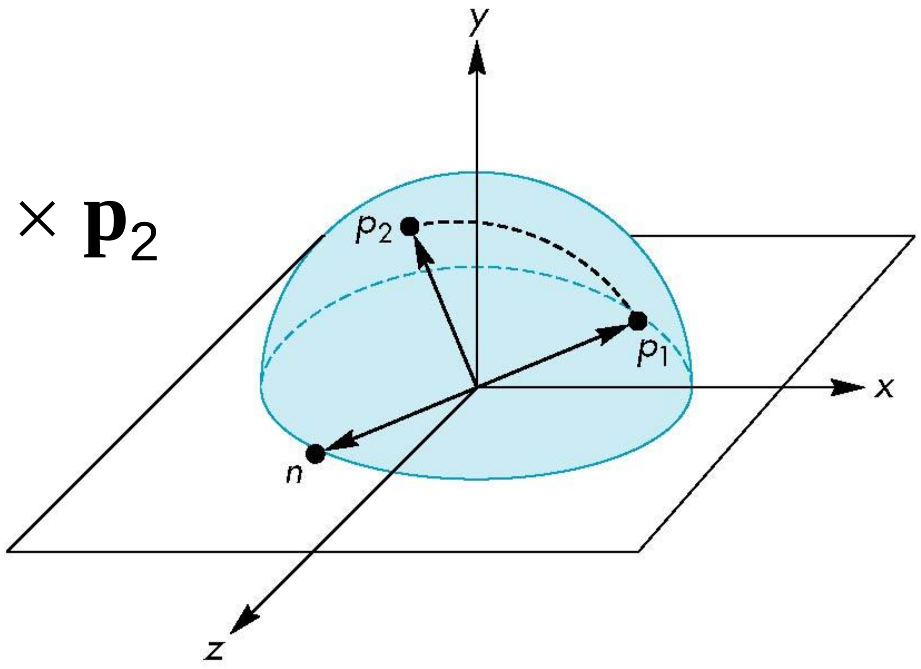
Computing Rotations

- Suppose that we have two points that were obtained from the mouse.
- We can project them up to the hemisphere to points $\mathbf{p}_1$ and $\mathbf{p}_2$
- These points determine a great circle on the sphere
- We can rotate from $\mathbf{p}_1$ to $\mathbf{p}_2$ by finding the proper axis of rotation and the angle between the points

Using the cross product

- The axis of rotation is given by the normal to the plane determined by the origin, $\mathbf{p}_1$, and $\mathbf{p}_2$

$$\mathbf{n} = \mathbf{p}_1 \times \mathbf{p}_2$$



Obtaining the angle

- The angle between $\mathbf{p}_1$ and $\mathbf{p}_2$ is given by

$$|\sin\theta| = \frac{|\mathbf{n}|}{|\mathbf{p}_1||\mathbf{p}_2|}$$

- If we move the mouse slowly or sample its position frequently, then θ will be small and we can use the approximation

$$\sin\theta \approx \theta$$

Implementing with WebGL

- Define actions in terms of three booleans
- `trackingMouse`: if true update trackball position
- `redrawContinue`: if true, idle function posts a redisplay
- `trackballMove`: if true, update rotation matrix

Using Quaternions

- Quaternion arithmetic works well for representing rotations around the origin
- Can use directly avoiding rotation matrices in the virtual trackball
- Code was made available long ago (pre shader) by SGI
- Quaternion shaders are simple
- Suppose you are in the northern hemisphere looking at north star. We can describe its position uniquely by the angle between our line of sight and the ground (the elevation) and some fixed longitude (the azimuth)
- Now suppose you are in north pole, the elevation is 90 degrees and the azimuth is irrelevant! The name gimbal lock comes from the gibal device that is used in gyroscopes. The problem has arisen in spacecraft and robots.
- $R = Rz(\Psi)\ Ry(\Phi)(\ Rx(\theta)$ The problem is that we lose one degree of freedom for Ry and there are infinite ways to combine θ and Ψ to get the same rotation.
- That’s why we use quaternions.

Quaternions

- Extension of imaginary numbers from two to three dimensions
- Recalling Euler’s identity: $e^{i\theta} = \cos(\theta) + i \sin(\theta)$
- $c = a + ib = r e^{i\theta}$ where $a^2 + b^2 = r^2$
- If we rotate c about the origin by Φ
- $C' = r e^{i(\theta+\Phi)} = r e^{i\theta} e^{i\Phi}$ where $e^{i\Phi}$ is a rotation operator in the complex plane
- In three dimensional, we need to specify both a direction (a vector) and the amount of rottion about it (a scalar)
- Requires one real and three imaginary components **i**, **j**, **k**

- $i^2 = j^2 = k^2 = ijk = -1$

$$q = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}$$

- Quaternions can express rotations on sphere smoothly and efficiently. Process:
  - Model-view matrix $\rightarrow$ quaternion
  - Carry out operations with quaternions
  - Quaternion $\rightarrow$ Model-view matrix

Vertex Shader I

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec4 rquat; // rotation quaternion

// quaternion multiplier

vec4 multq(vec4 a, vec4 b)
{
   return(vec4(a.x*b.x - dot(a.yzw, b.yzw),
        a.x*b.yzw+b.x*a.yzw+cross(b.yzw, a.yzw)));
}
```

Vertex Shader II

```
// inverse quaternion
vec4 invq(vec4 a)
{ return(vec4(a.x, -a.yzw)/dot(a,a)); }

  void main() {
  vec3 axis = rquat.yxw;
  float theta = rquat.x;
  vec4 r, p;
  p = vec4(0.0, vPosition.xyz);  // input point quaternion
  p = multq(rquat, multq(p, invq(rquat))); // rotated point quaternion
  gl_Position = vec4( p.yzw, 1.0); // back to homogeneous coordinates
  color = vColor;
}
```

Classical Viewing

Objectives

- Introduce the classical views
- Compare and contrast image formation by computer with how images have been formed by architects, artists, and engineers
- Learn the benefits and drawbacks of each type of view

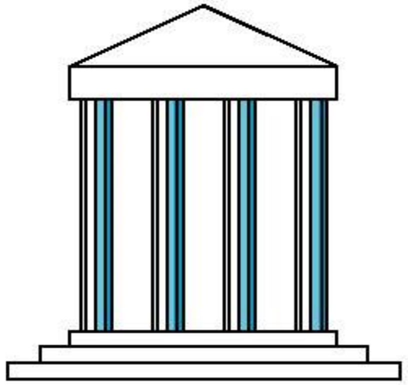
Classical Viewing

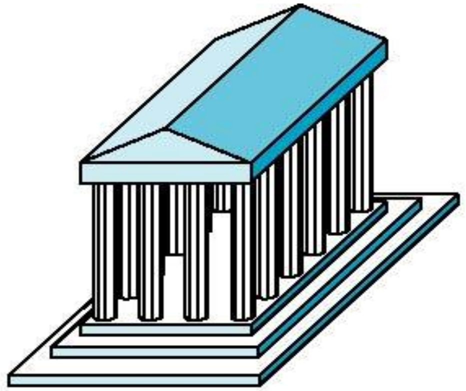
- Viewing requires three basic elements
  - One or more objects
  - A viewer with a projection surface
  - Projectors that go from the object(s) to the projection surface
- Classical views are based on the relationship among these elements
- The viewer picks up the object and orients it how she would like to see it
- In classical viewing, there is the underlying notion of a <u>principal face</u>
- For a rectangular object, such as a building, there are natural notions of front, back, top, bottom, right, and left faces.
- Each object is assumed to be constructed from flat *principal faces*
  - Buildings, polyhedra, manufactured objects

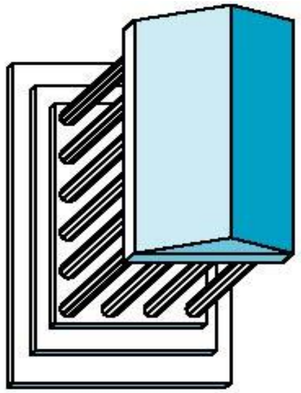
Planar Geometric Projections

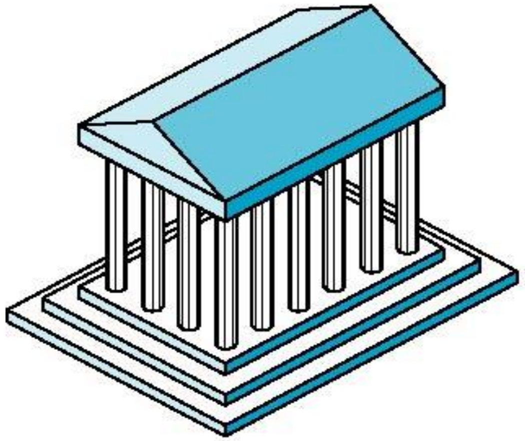
- Most modern APIs support both parallel and perspective viewing. The class of projections produced by these systems is known as “**planar geometric projection**”
- Standard projections project onto a plane
- Planar geometric projection: projection surface is a plane and the projectors are line.
- Projectors are lines that either
  - converge at a center of projection (COP) → perspective projection
  - are parallel → parallel projection
- Both parallel and perspective projections preserve lines
  - but not necessarily preserve angles
- Nonplanar projections are needed for applications such as map construction

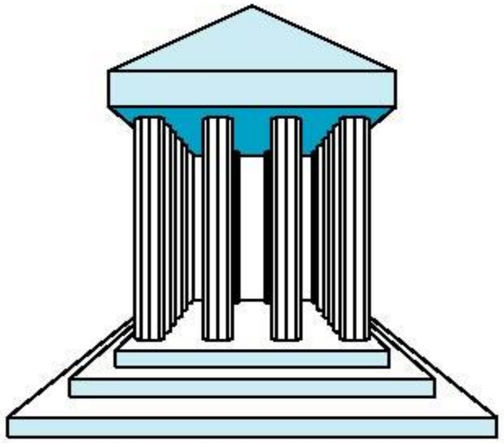
Classical Projections

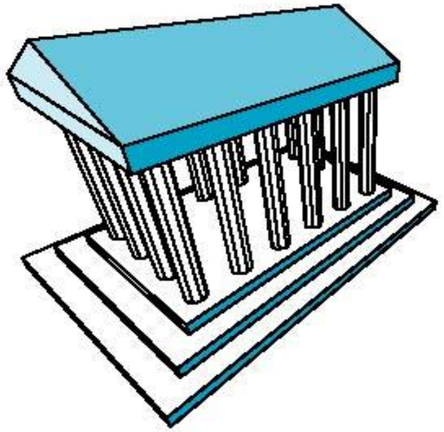

Front elevation


Elevation oblique


Plan oblique


Isometric

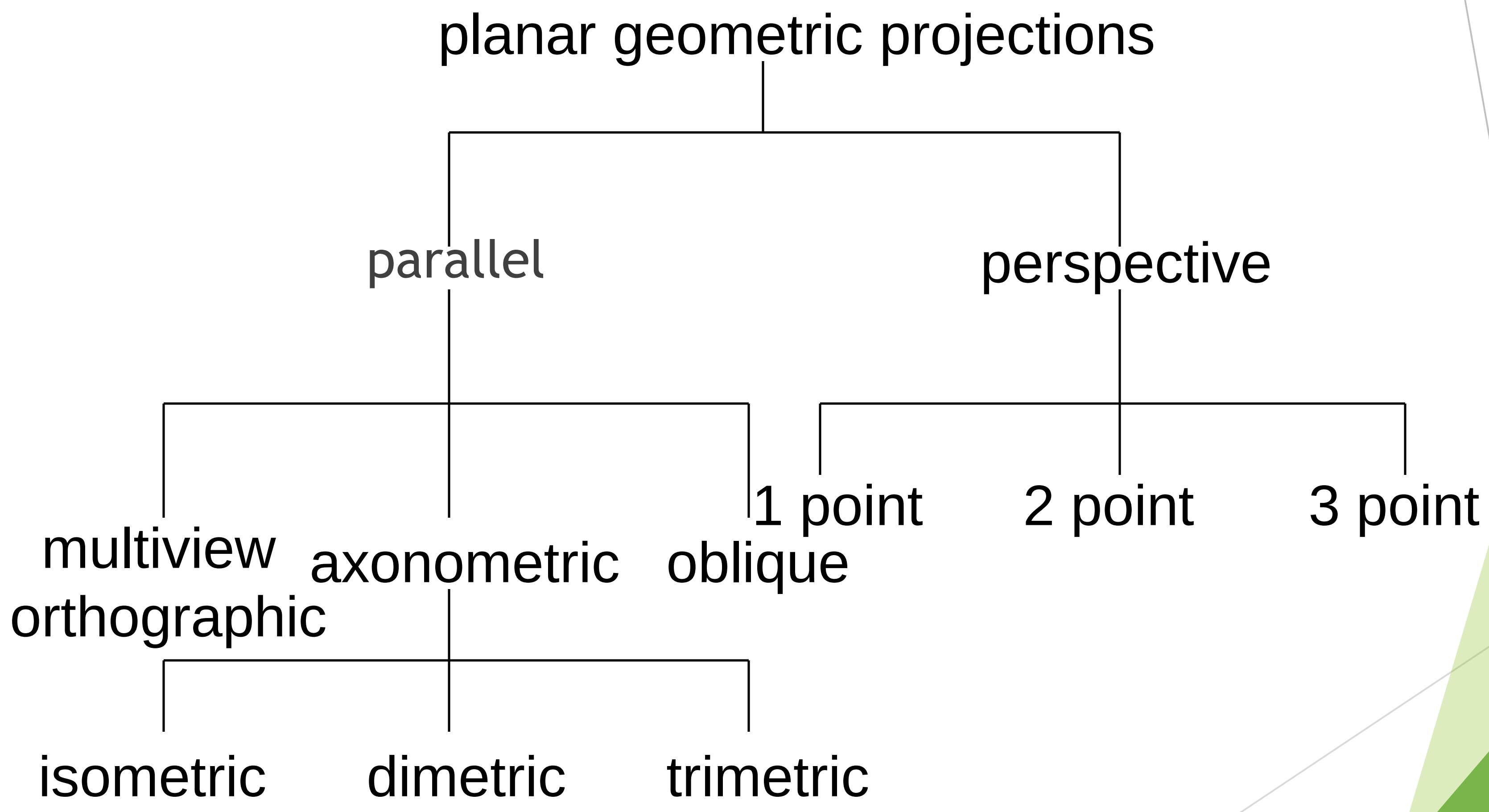

One-point perspective


Three-point perspective

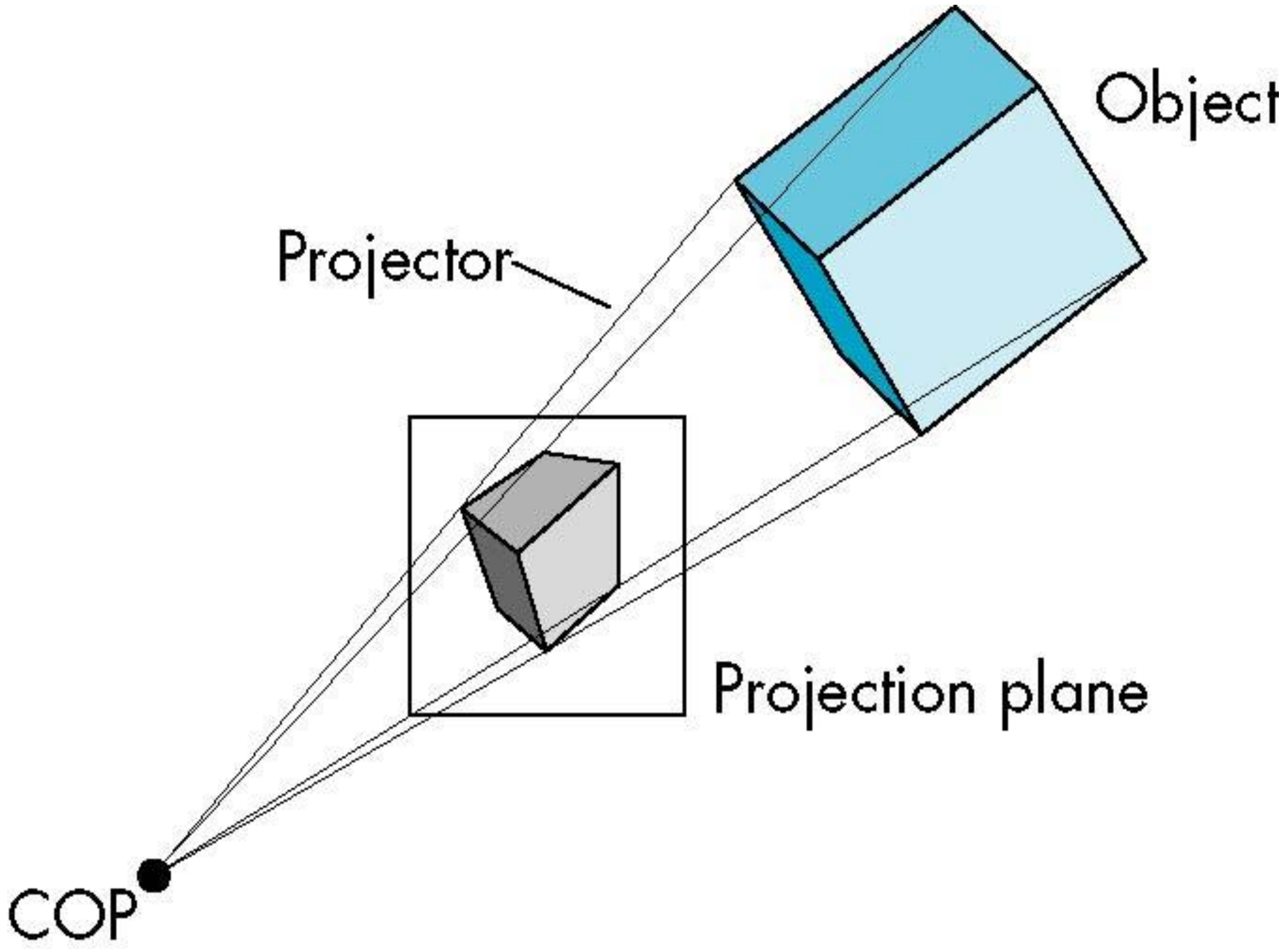
Perspective vs Parallel

- Computer graphics treats all projections the same and implements them with a single pipeline
- Classical viewing developed different techniques for drawing each type of projection
- Fundamental distinction is between parallel and perspective viewing even though mathematically parallel viewing is the limit of perspective viewing
- Views with finite COP are called perspective views;
- Views with COP at infinity are called parallel views. For parallel views, the origin of the camera frame usually lies in the projection plane.

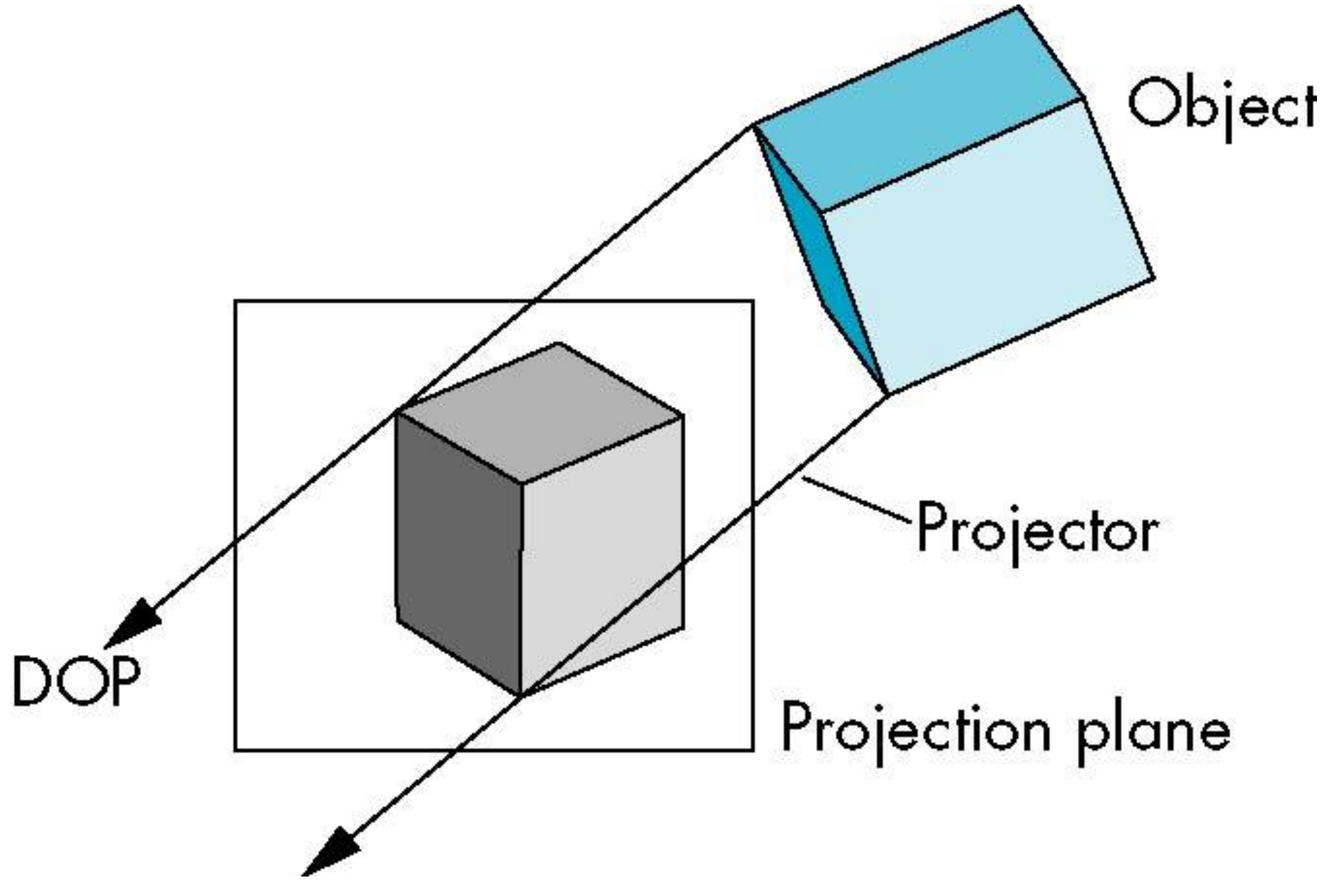
Taxonomy of Planar Geometric Projections



Perspective Projection

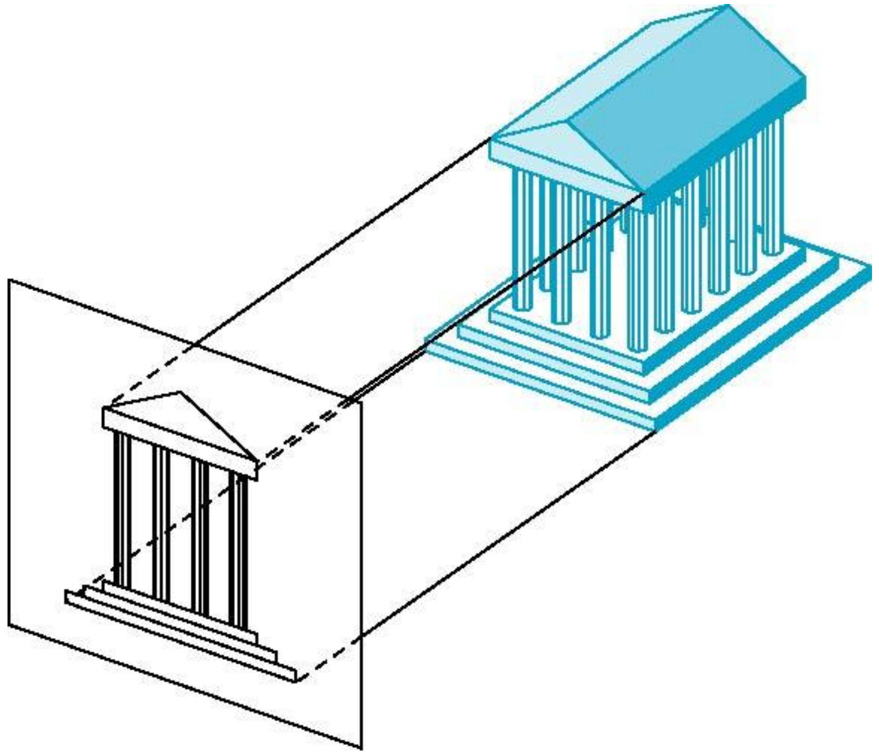


Parallel Projection



Orthographic Projection

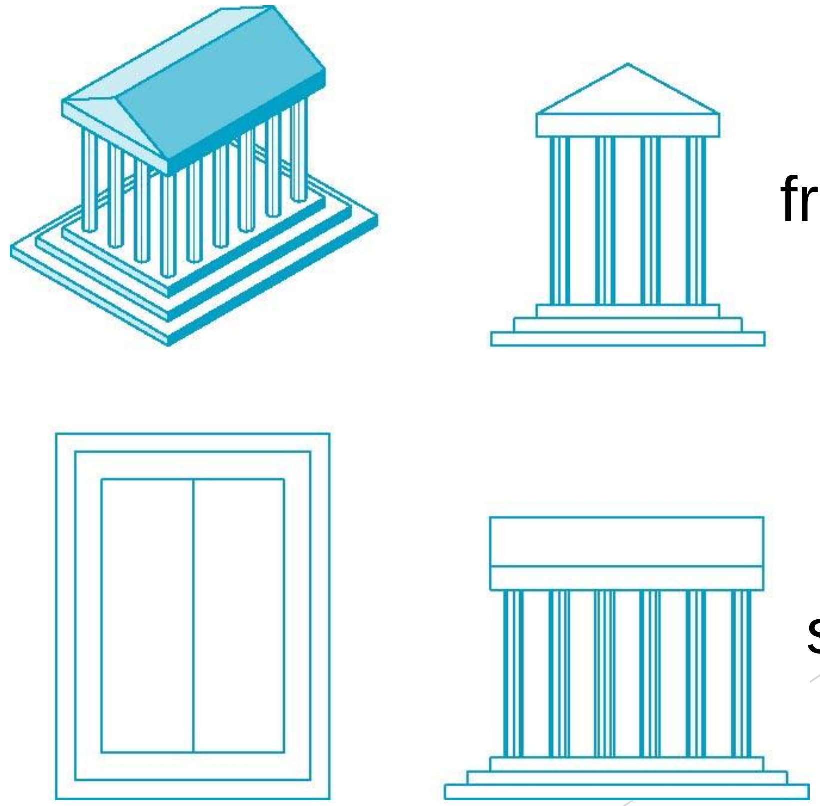
Projectors are orthogonal to projection surface



Multiview Orthographic Projection

- Projection plane parallel to principal face
- Usually form front, top, side views

isometric (not multiview orthographic view)



in CAD and architecture, we often display three multiviews plus isometric

top

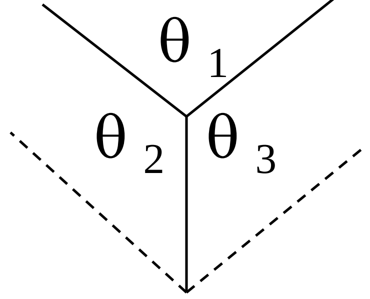
Advantages and Disadvantages

- Preserves both distances and angles
  - Shapes preserved
  - Can be used for measurements
    - Building plans
    - Manuals
- Cannot see what object really looks like because many surfaces hidden from view
  - Often we add the isometric

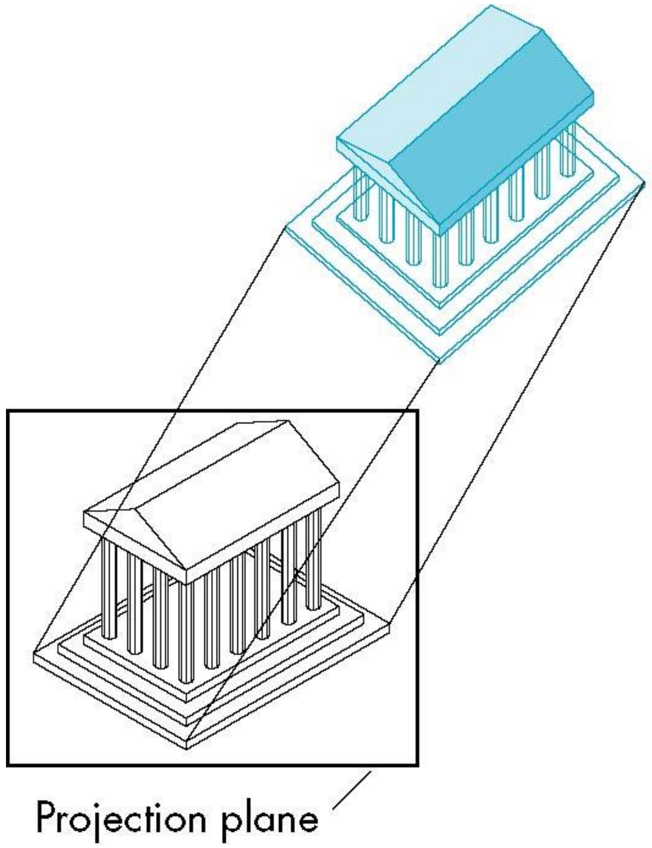
Axonometric Projections

Allow projection plane to move relative to object

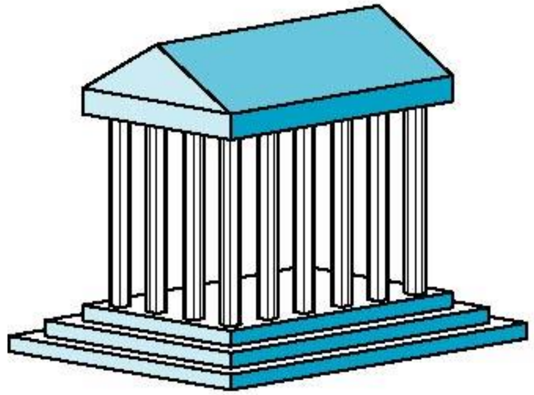
classify by how many angles of a corner of a projected cube are the same

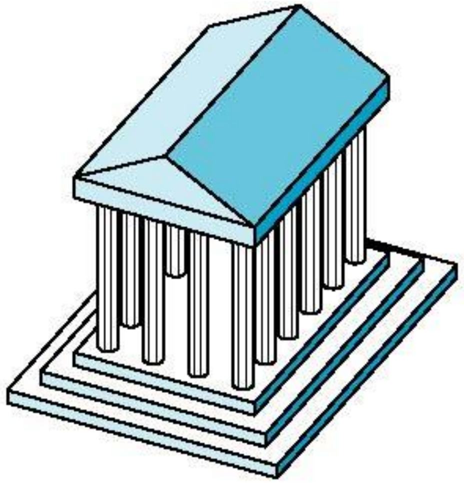


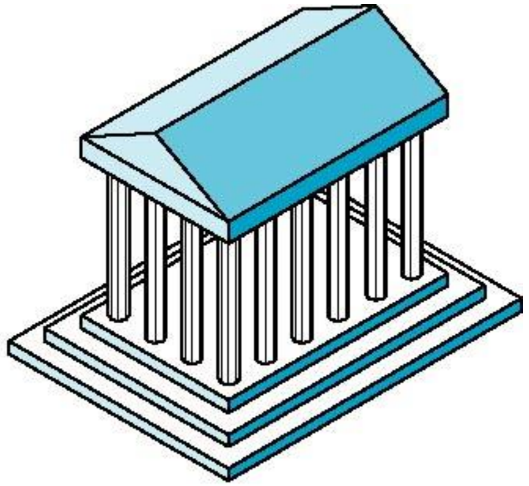
none: trimetric
two: dimetric
three: isometric



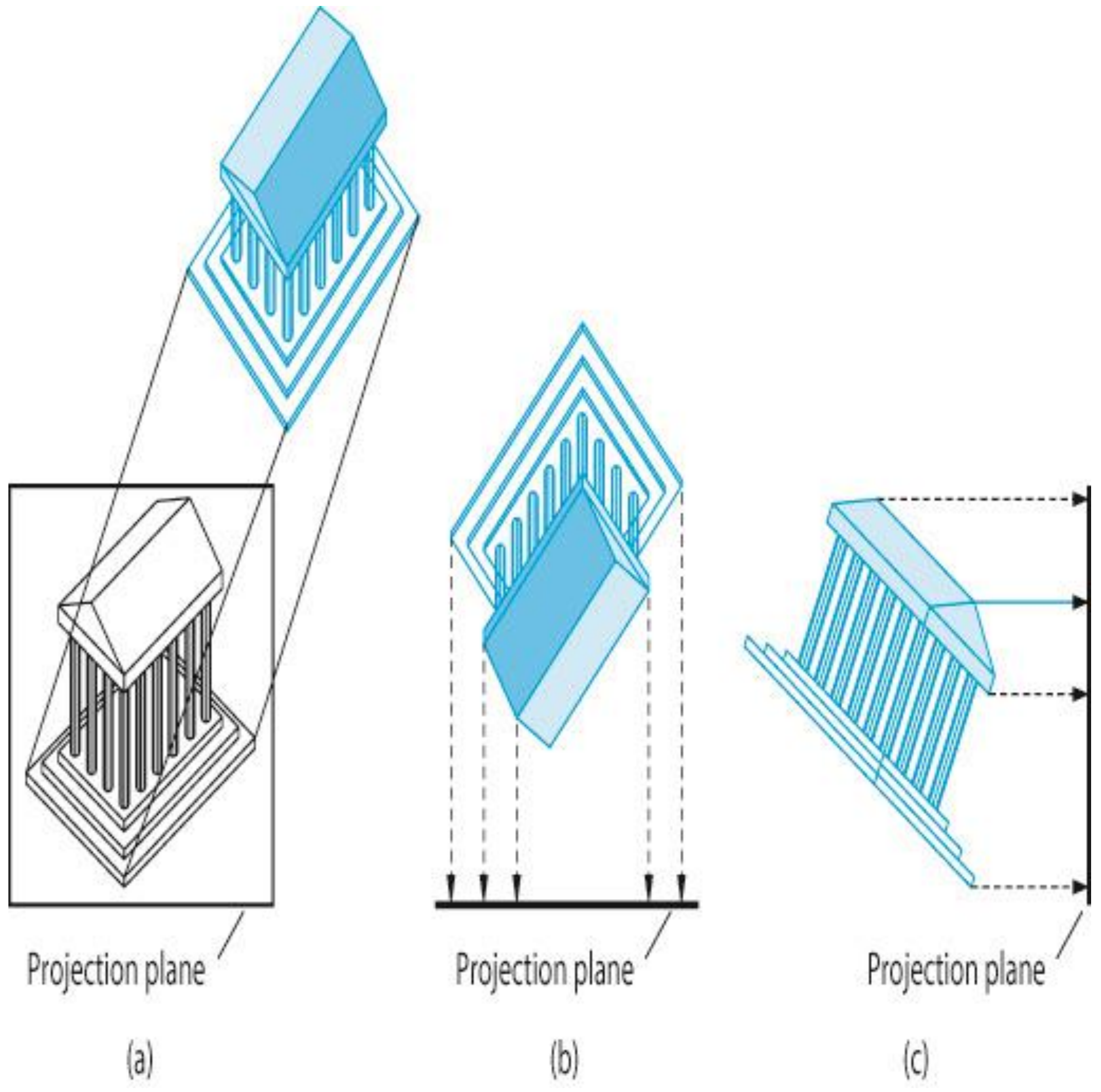
Types of Axonometric Projections


Dimetric


Trimetric


Isometric

Axonometric Projections

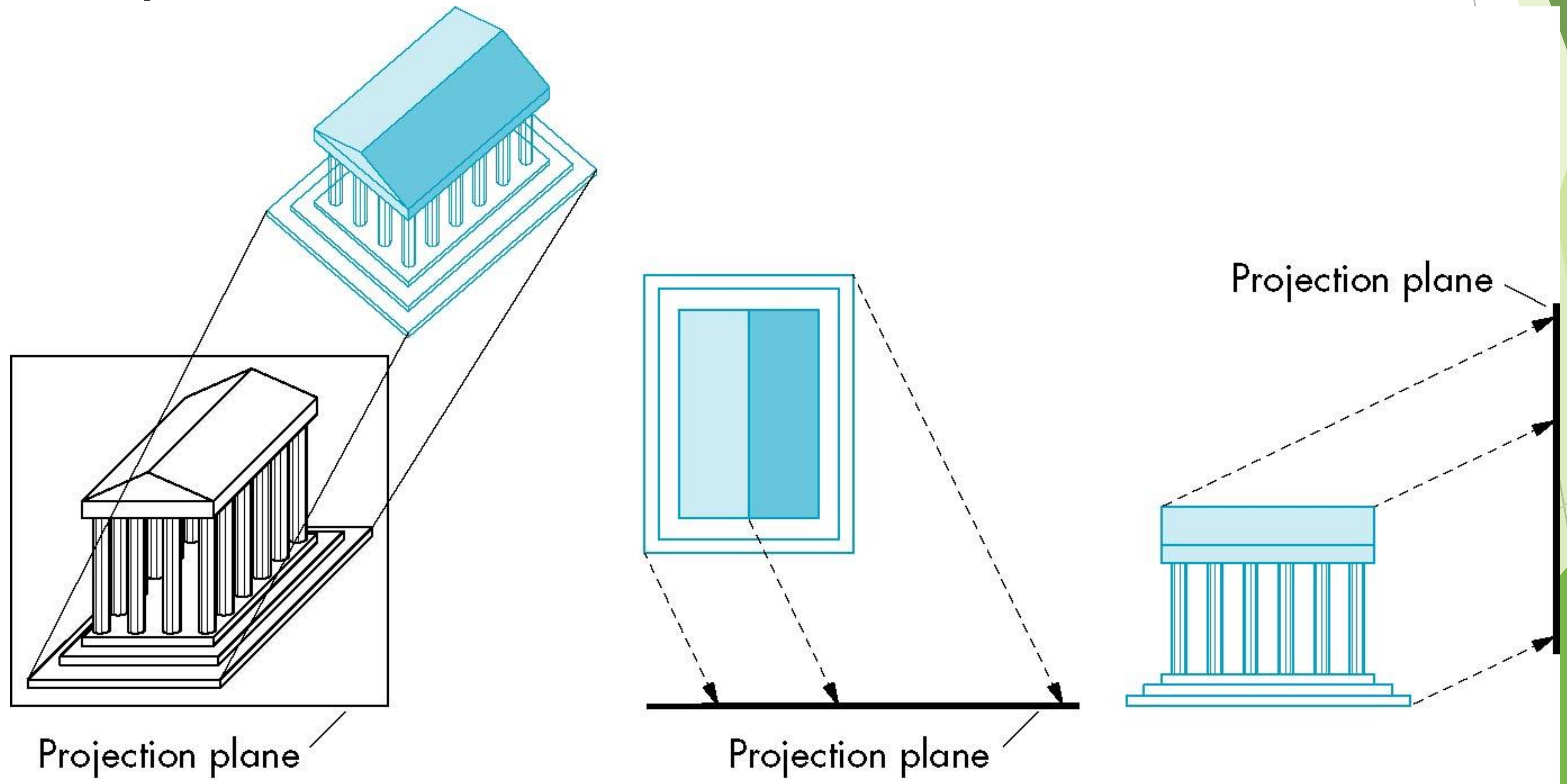


Advantages and Disadvantages

- Lines are scaled (*foreshortened*) but can find scaling factors
- Lines preserved but angles are not
  - Projection of a circle in a plane not parallel to the projection plane is an ellipse
- Can see three principal faces of a box-like object
- Some optical illusions possible
  - Parallel lines appear to diverge
- Does not look real because far objects are scaled the same as near objects
- Used in CAD applications (Architectural and Mechanical design)

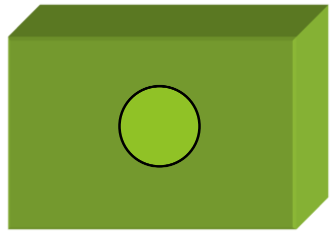
Oblique Projection

Arbitrary relationship between projectors and projection plane



Advantages and Disadvantages

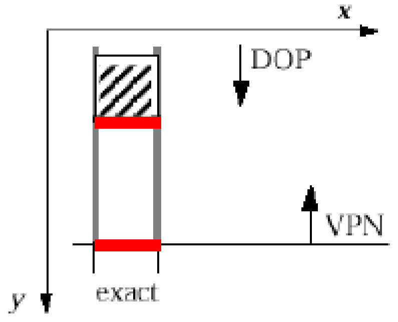
- Can pick the angles to emphasize a particular face
  - Architecture: plan oblique, elevation oblique
- Angles in faces parallel to projection plane are preserved while we can still see "around" side

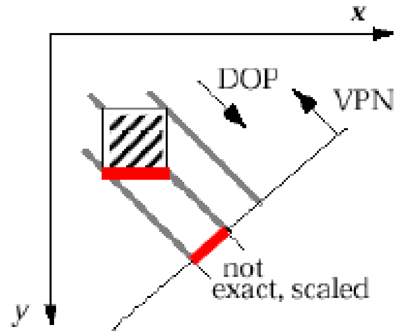


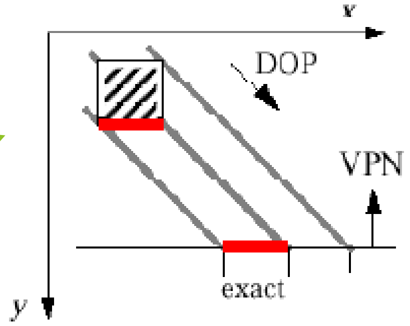
- In physical world, cannot create with simple camera; possible with bellows camera or special lens (architectural)

Parallel Projections

- Assume object face of interest lies in principal plane, i.e., parallel to xy, yz, or zx planes. DOP is the direction of projection. VPN is view plane normal.
- Multiview orthographic: VPN || a principal coordinate axis. DOP || VPN. Shows single face, exact measurements.
- Axonometric: VPN is not parallel to a principal coordinate axis. DOP || VPN. Shows adjacent faces, none exact measurement, uniformly foreshortened (function of angle between face normal and DOP).
- Oblique: VPN || a principal coordinate axis. DOP is not parallel to VPN. Adjacent faces, one exact, others uniformly foreshortened.

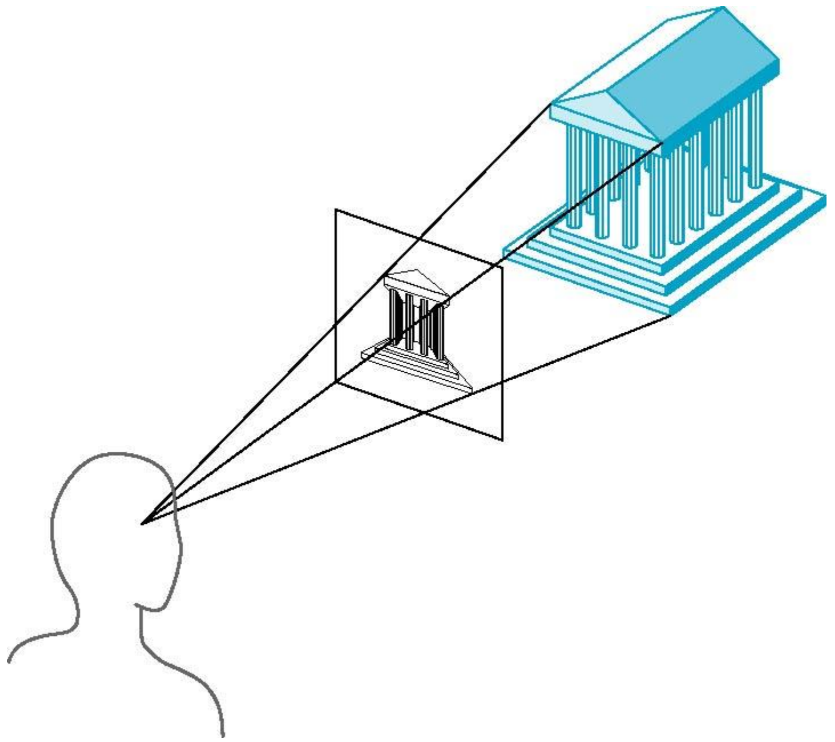






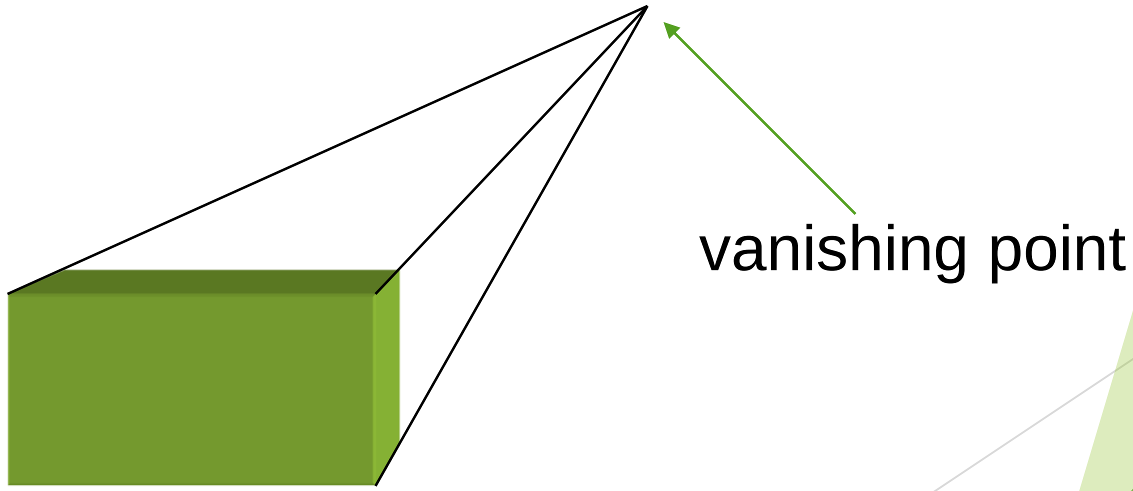
Perspective Projection

Projectors converge at center of projection. All perspective views are characterized by “diminution” of size.



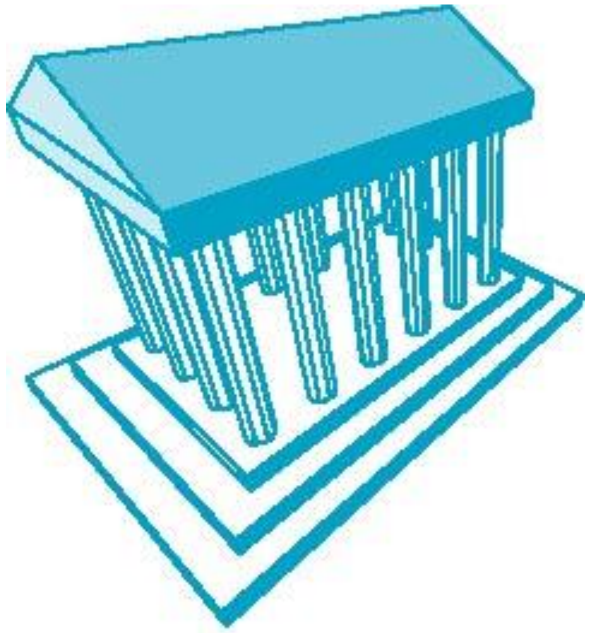
Vanishing Points

- Parallel lines (not parallel to the projection plan) on the object converge at a single point in the projection (the *vanishing point*)
- Drawing simple perspectives by hand uses these vanishing point(s)



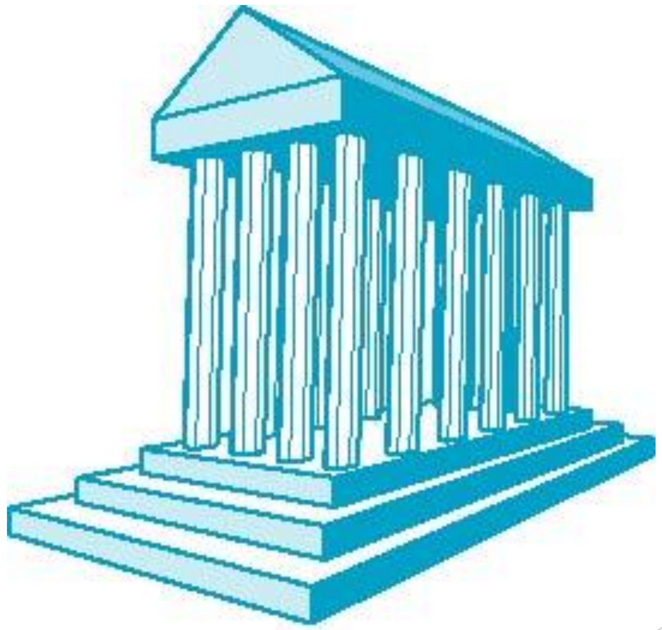
Three-Point Perspective

- No principal face parallel to projection plane
- Three vanishing points for cube



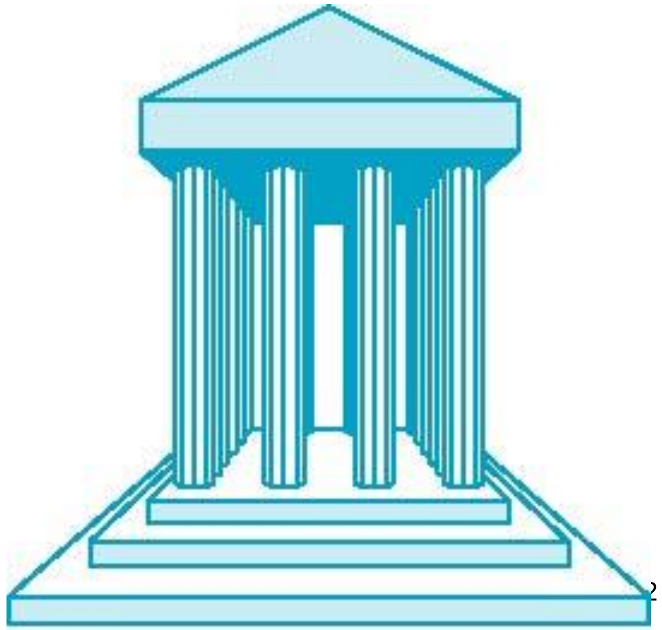
Two-Point Perspective

- One principal direction parallel to projection plane
- Two vanishing points for cube



One-Point Perspective

- One principal face parallel to projection plane
- One vanishing point for cube



Advantages and Disadvantages

- Objects further from viewer are projected smaller than the same sized objects closer to the viewer (*diminution*)
  - Looks realistic
- Equal distances along a line are not projected into equal distances (*nonuniform foreshortening*)
- Angles preserved only in planes parallel to the projection plane
- More difficult to construct by hand than parallel projections (but not more difficult by computer)

Computer Viewing

Positioning the Camera

Objectives

- Introduce the mathematics of projection
- Introduce WebGL viewing functions in MV.js
- Look at alternate viewing APIs

From the Beginning

- In the beginning:
  - fixed function pipeline
  - Model-View and Projection Transformation
  - Predefined frames: model, object, camera, clip, ndc, window
- After deprecation
  - pipeline with programmable shaders
  - no transformations
  - clip, ndc window frames
- MV.js reintroduces original capabilities

Computer Viewing

- There are three aspects of the viewing process, all of which are implemented in the pipeline,
  - Positioning the camera
    - Setting the model-view matrix
  - Selecting a lens
    - Setting the projection matrix
  - Clipping
    - Setting the view volume

3D Viewing: Synthetic Camera

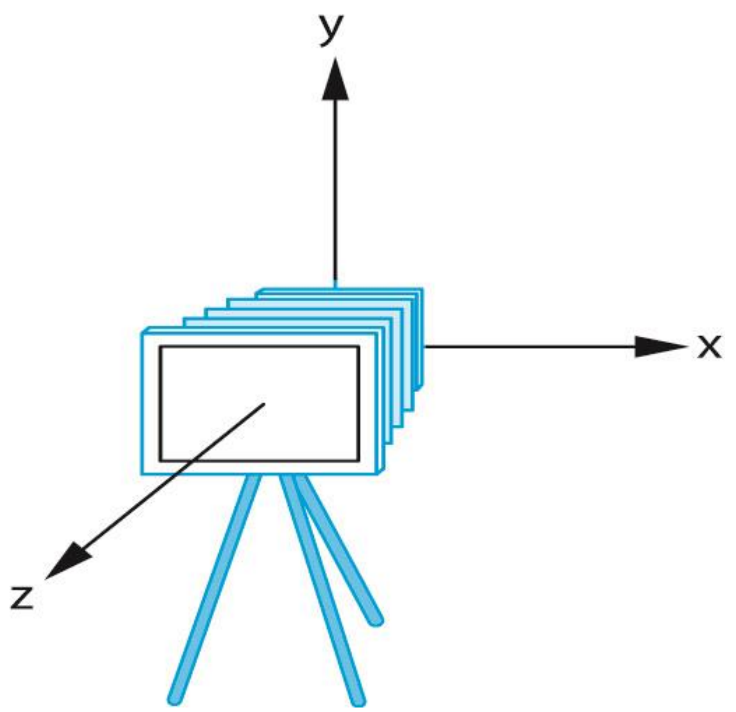
- The synthetic camera is the programmer’s model to specify 3D view projection parameters to the computer.
  - Position of the camera
  - Orientation
  - Field of view(wide angle, normal,…)
  - Depth of field (near distance, far distance)
  - Focal distance
  - Tilt of view/film plane (if not normal to view direction, produces oblique projections)
  - Perspective or parallel (camera near objects or an infinite distance away)

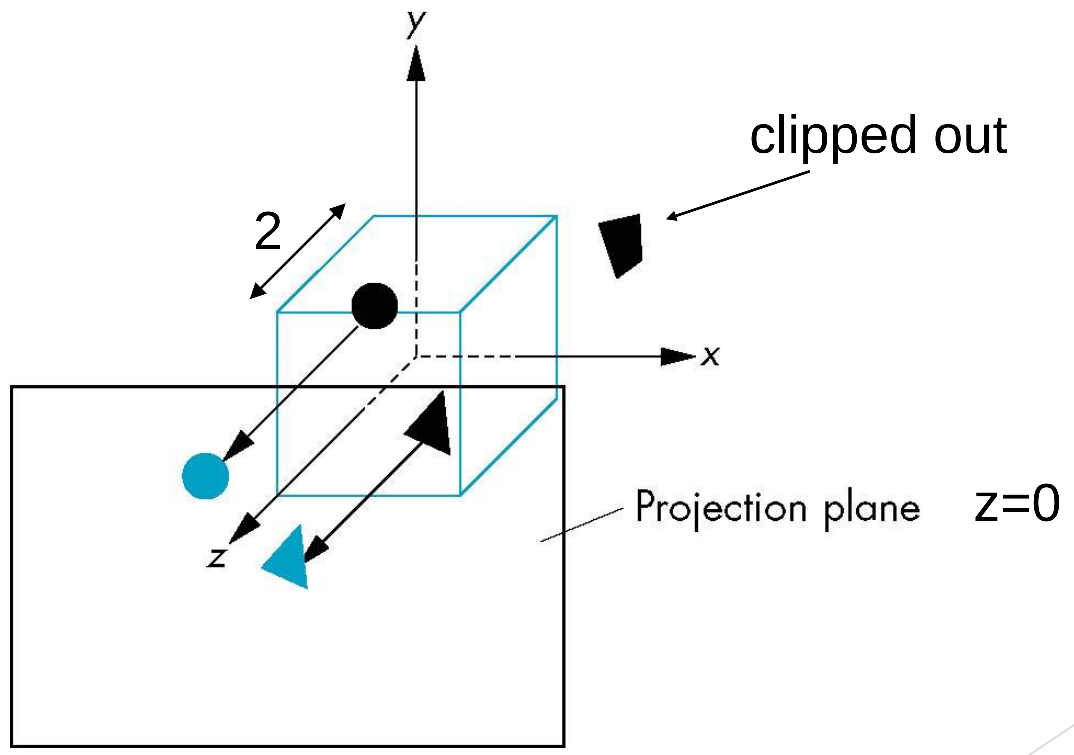
The WebGL Camera

- 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

Default Projection

Default projection is orthogonal





Moving the Camera Frame

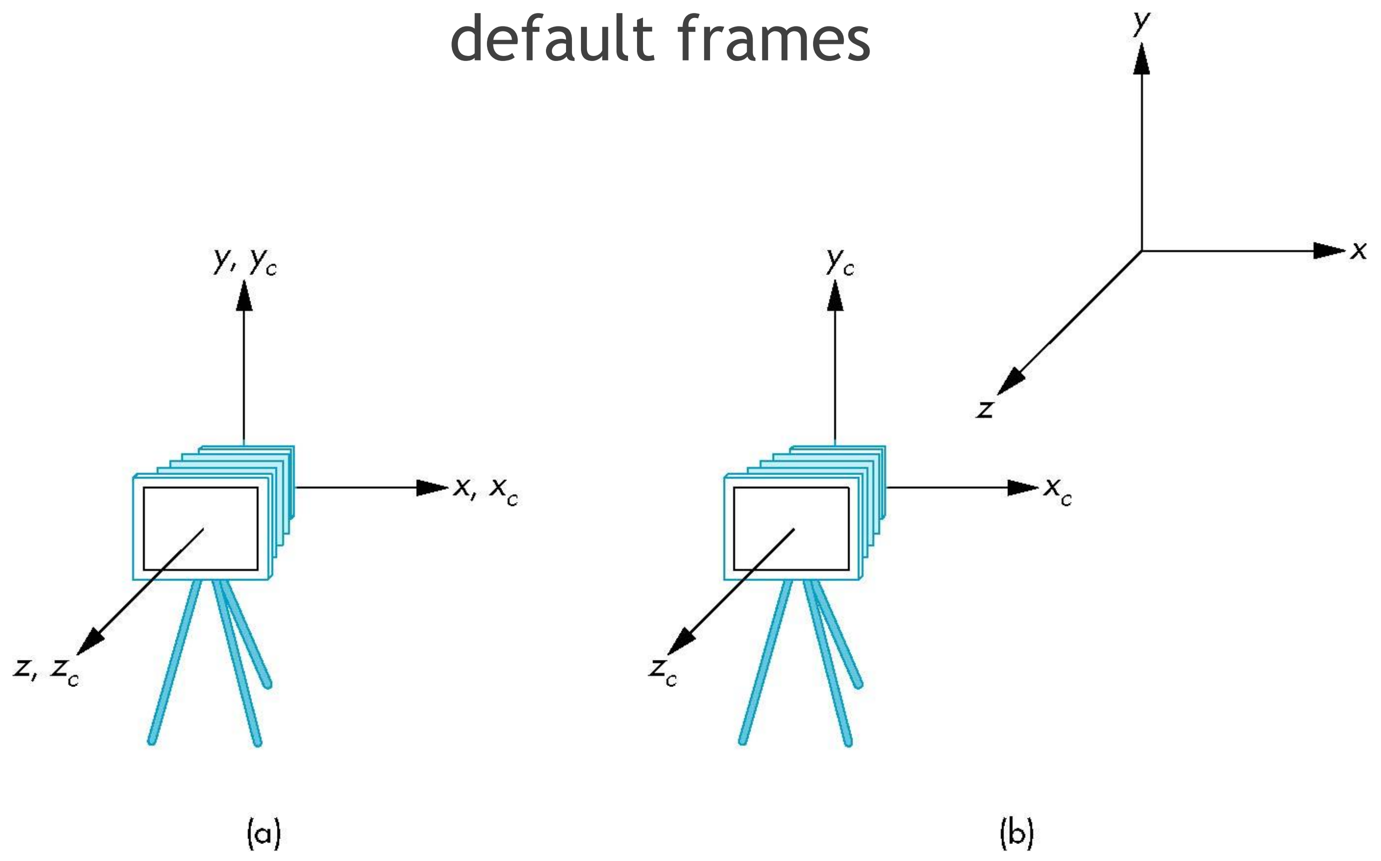
- If we want to visualize objects with both positive and negative z values we can either
  - Move the camera in the positive z direction
    - Translate the camera frame
  - Move the objects in the negative z direction
    - Translate the world frame
- Both of these views are equivalent and are determined 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

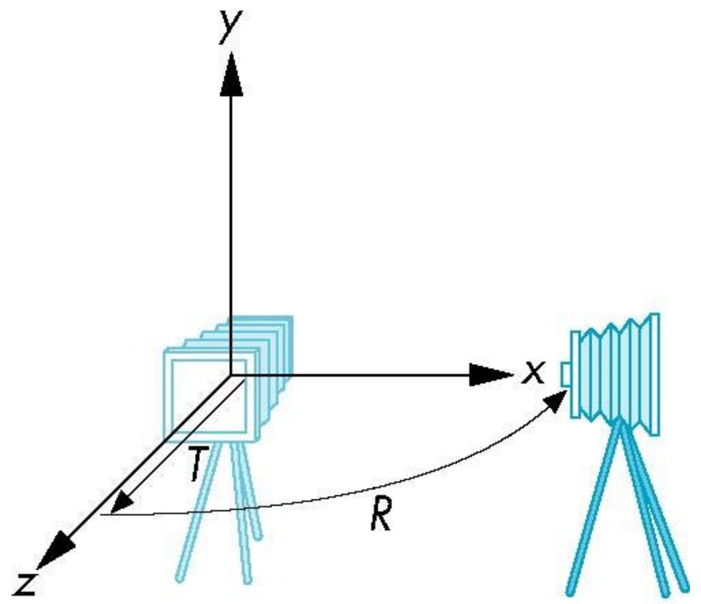
d > 0

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 C = TR



WebGL code

- Remember that last transformation specified is first to be applied

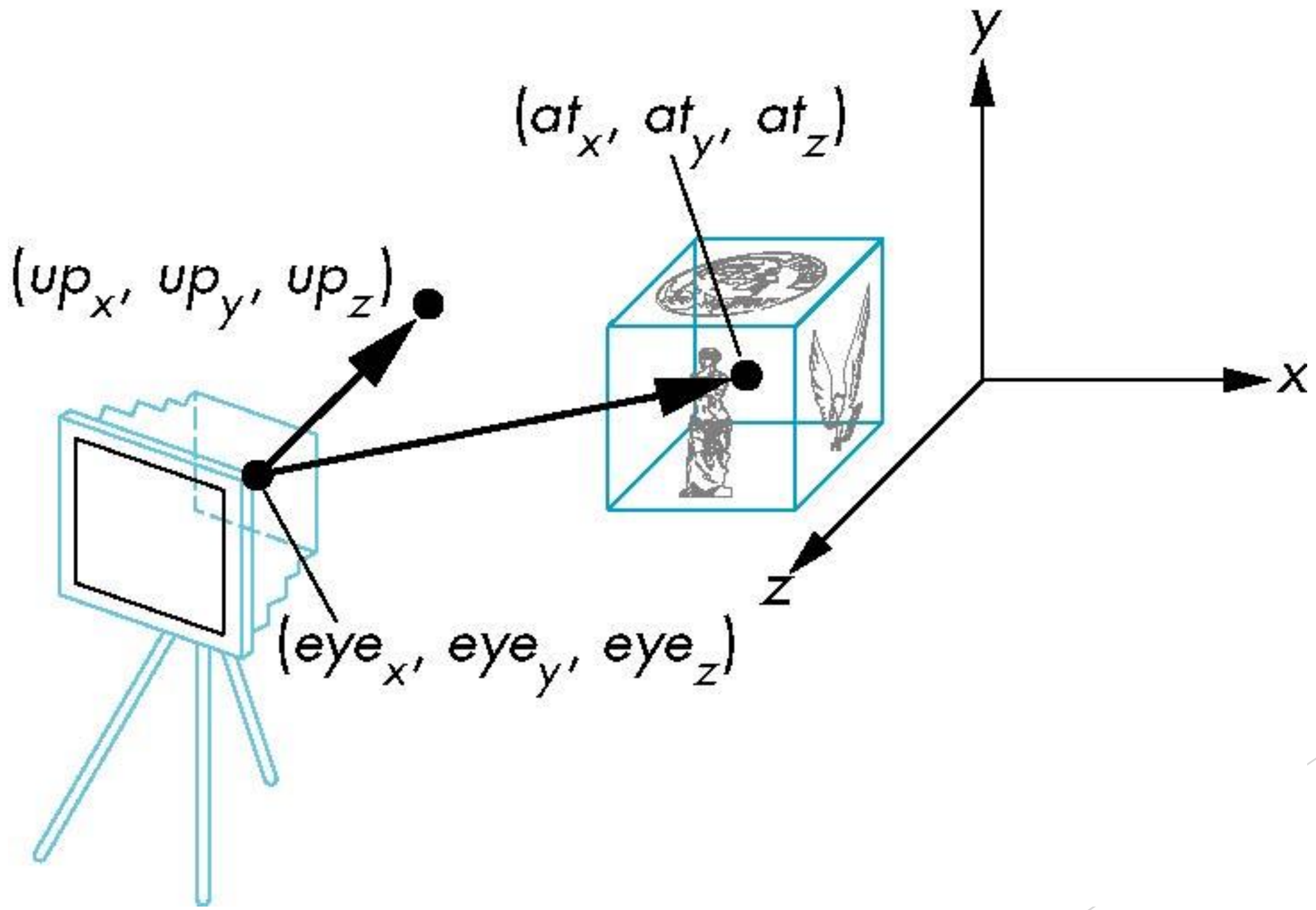
```
// Using MV.js

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););
```

View Volume Specification

- We need to know six things about our synthetic camera model in order to take a picture
  1. Position of the camera (three degrees of freedom: x,y, and z coordinates in 3D)
  2. The look vector: what direction is camera pointing
  3. Up vector: The camera orientation is determined by look vector and angle through which the camera is rotated about that vector, i.e., the direction of up vector
  4. Aspect ratio of the electronic film: ratio of width to height. Square viewing 1:1. Movie theater “letter box” format aspect ratio of 2:1. NTSC television: 4:3. HDTV 16:9
  5. Height Angle: determines how much of the scene will fit into our new view volume. Larger height angles fit more of the scene into view volume (width angle determined by height angle and aspect ratio). The greater the angle, the greater amount of distortion. Choosing a view angle analogous to photographer choosing a specific type of lens (wide angle or telephoto lens)
  6. Front and back clipping plane: limit extent of camera’s view by rendering parts of objects lying between them and throwing away everything outside of them.

lookAt

```
LookAt(eye, at, up)
```

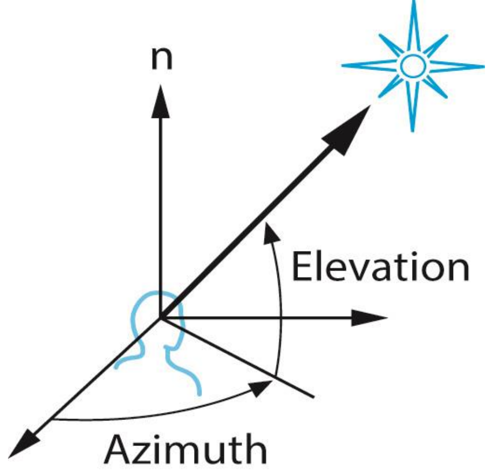


The lookAt Function

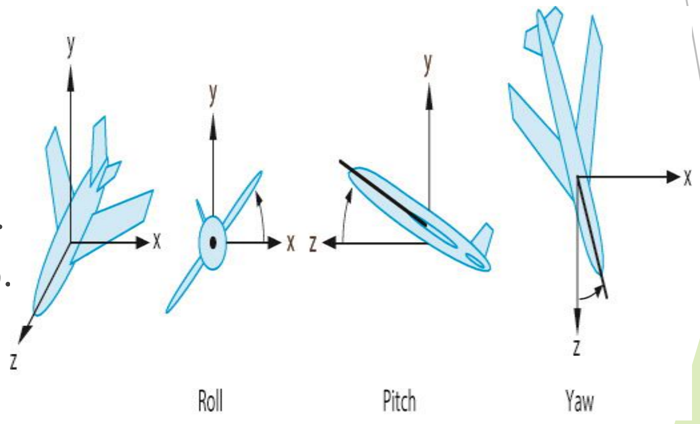
- The GLU library contained the function gluLookAt to form the required modelview matrix through a simple interface
- Note the need for setting an up direction
- Replaced by lookAt() in MV.js
  - Can concatenate with modeling transformations
- Example: isometric view of cube aligned with axes

```
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);
```

Other Viewing APIs



- The LookAt function is only one possible API for positioning the camera
- Others include
  - View reference point, view plane normal, view up.
  - Yaw, pitch, roll
    - Rotation around the front-to-back axis is called **roll (using ailerons)**.
    - Rotation around the side-to-side axis is called **pitch (using elevator)**.
    - Rotation around the vertical axis is called **yaw (using rudder)**.
  - Elevation, azimuth, twist
    - Elevation is the angle above the plane of the viewer at which the star appears.
    - Azimuth is the angle measured from an axis in this plane to the projection onto the plane of the line between the view and star.
    - The camera can still be rotated by a twist angle about the direction it's pointed.
  - Direction angles

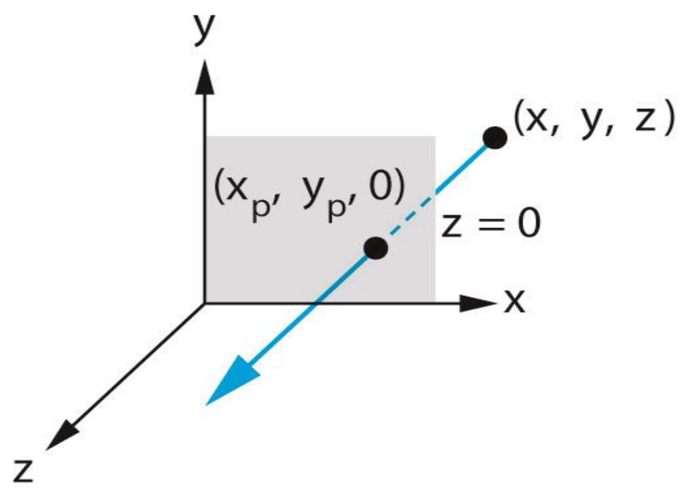


Objectives

- Introduce the mathematics of projection
- Add WebGL projection functions in MV.js

Projections and Normalization

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



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

- 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

Homogeneous Coordinate Representation

default orthographic projection

Orthogonal projections corresponds to a camera with a back plane parallel to lens, which has an infinite focal length.

$x_p = x$

$y_p = y$

$z_p = 0$

$w_p = 1$

$P = (x,y,z,w)$

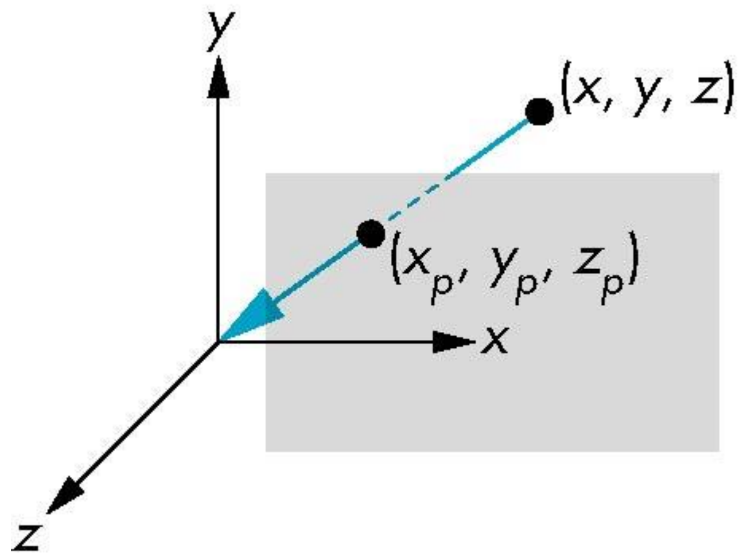
$P_p=(x_p, y_p, z_p, w_p)$

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{p}_p = \mathbf{Mp}$$

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

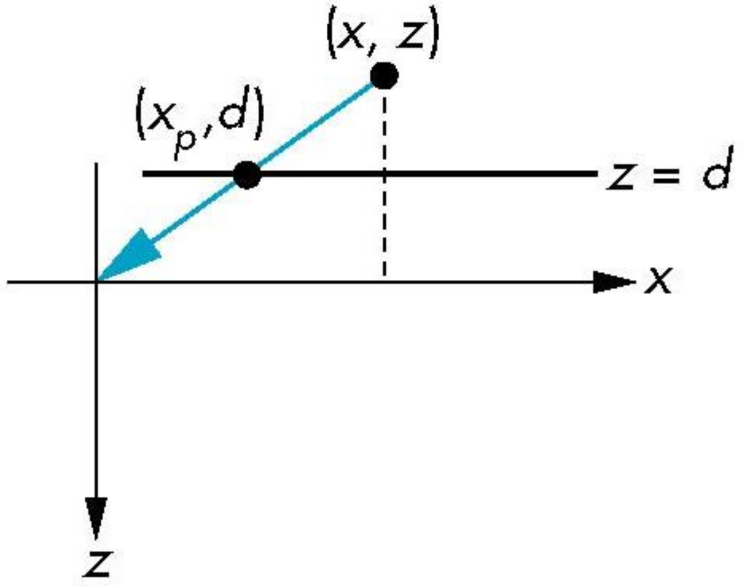
Simple Perspective

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

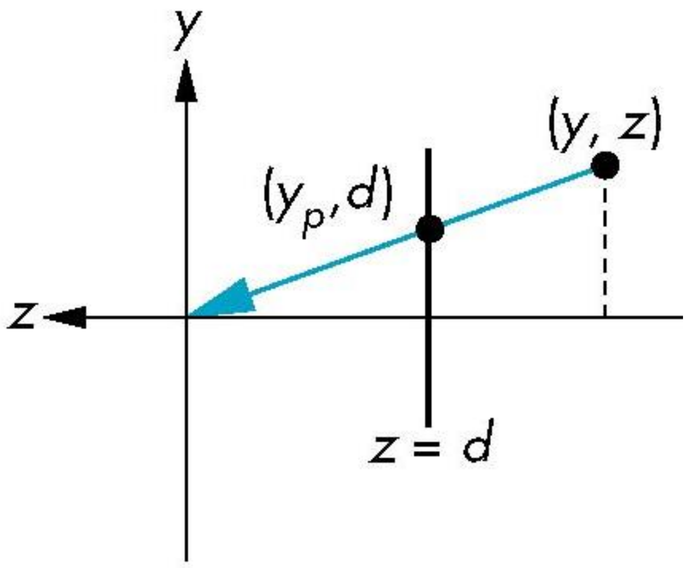


Perspective Equations

Consider top and side views



$\frac{x}{z/d}$



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

Homogeneous 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 $w \neq 1$, so we must divide by w to return from homogeneous coordinates
- This *perspective division* yields

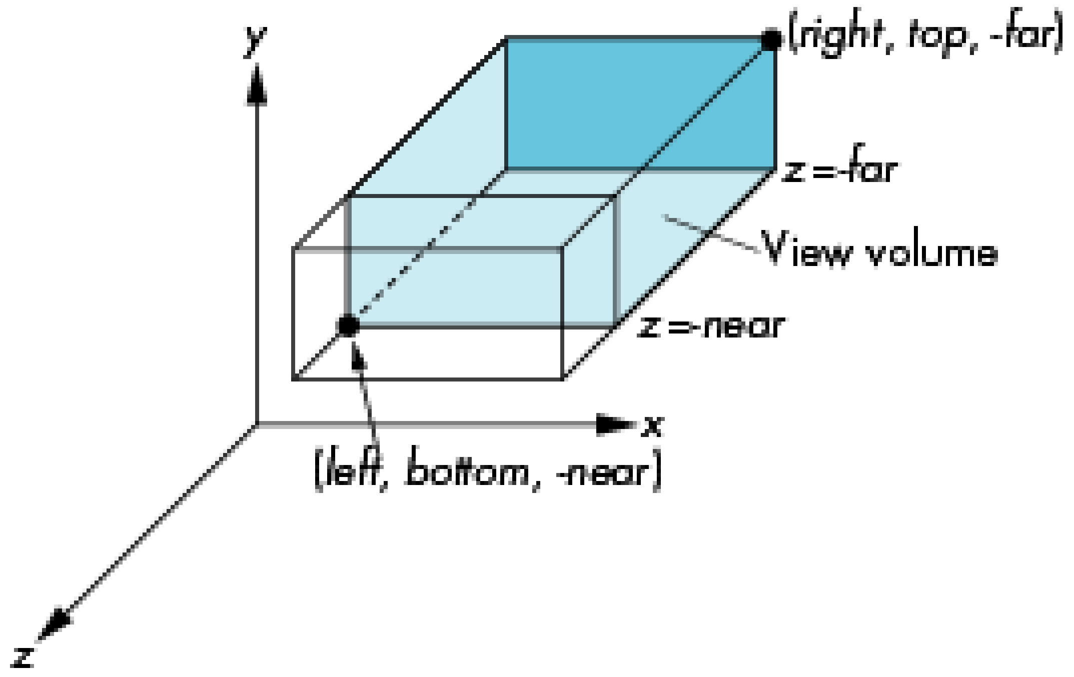
$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

the desired perspective equations

- We will consider the corresponding clipping volume with mat.h functions that are equivalent to deprecated OpenGL functions

WebGL Orthogonal Viewing

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

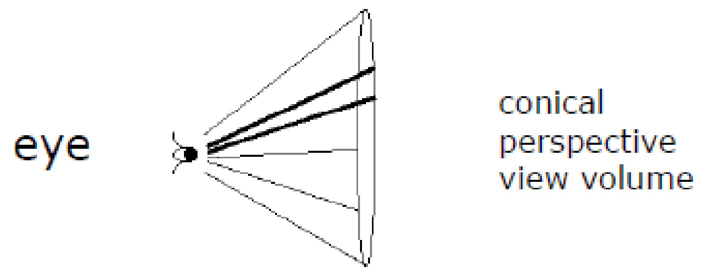


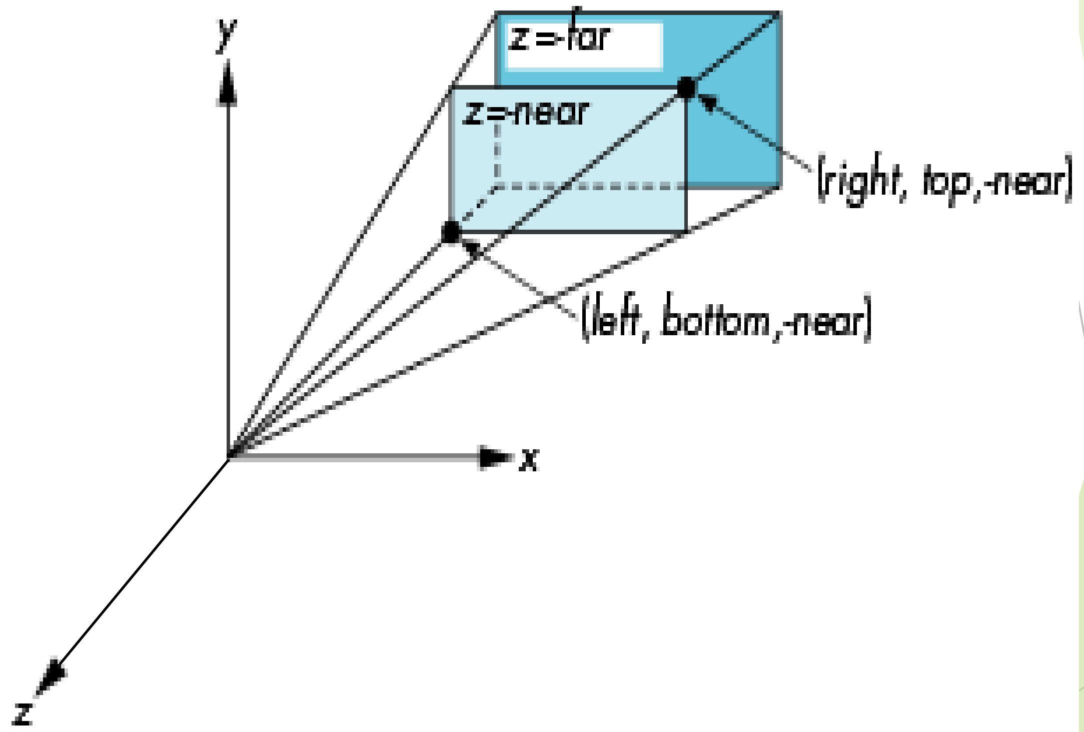
`near` and **`far`** measured <u>from</u> camera

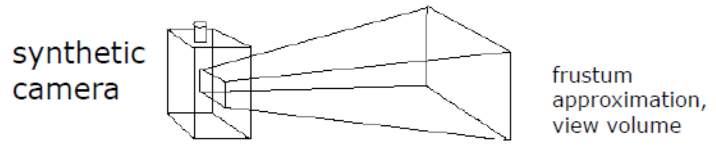
WebGL Perspective

- A view volume contains everything visible from the point of view or direction (What camera sees)
- Conical view volumes: approximates what eye sees. Expensive math (simultaneous quadratics) when clipping objects against cone's surface
- Can approximate with rectangular cone instead (called a frustum)
- Works well with a rectangular viewing window.
- Simultaneous linear equations for easy clipping of objects against sides

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

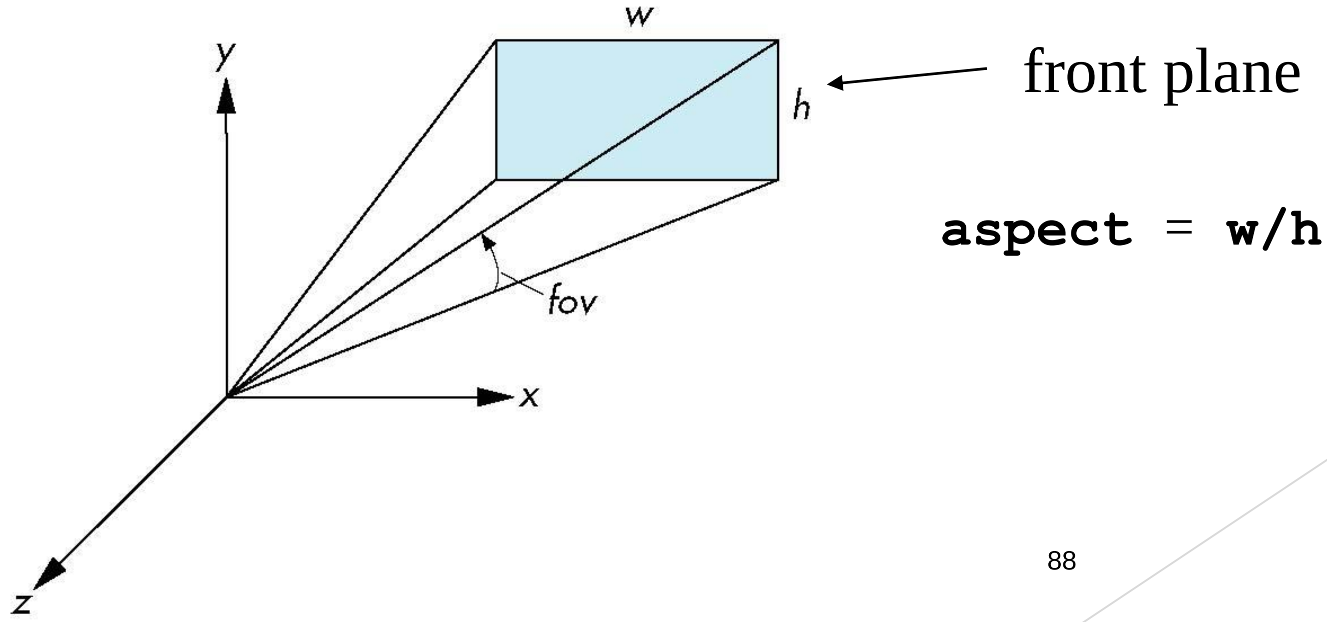






Using Field of View

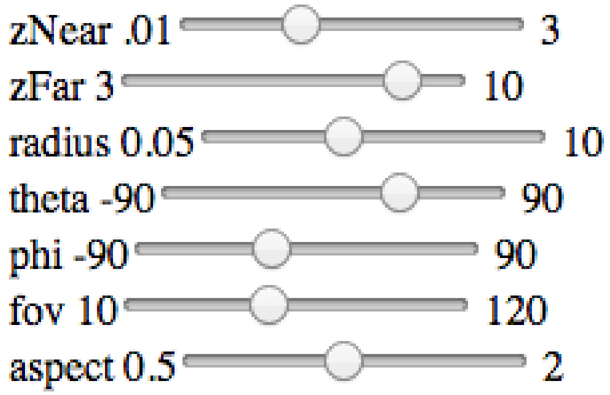
- With **`frustum`** it is often difficult to get the desired view
- **`perspective(fovy, aspect, near, far)`** often provides a better interface

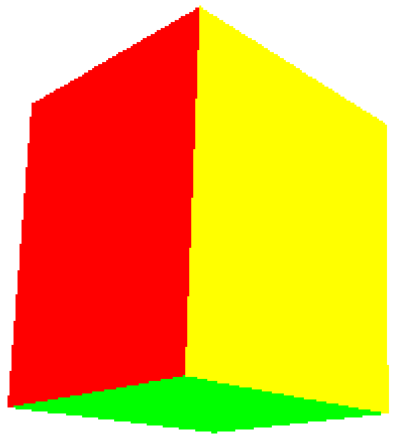


`aspect = w/h`

Computing Matrices

- Compute in JS file, send to vertex shader with gl.uniformMatrix4fv
- Dynamic: update in render() or shader


zNear .01 3
zFar 3 10
radius 0.05 10
theta -90 90
phi -90 90
fov 10 120
aspect 0.5 2



persepctive2.js

```
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);
}
```

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;
}
```