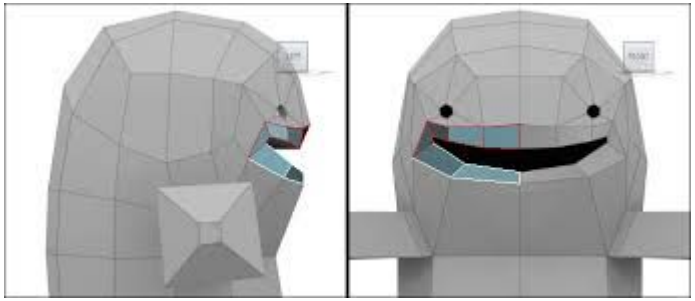
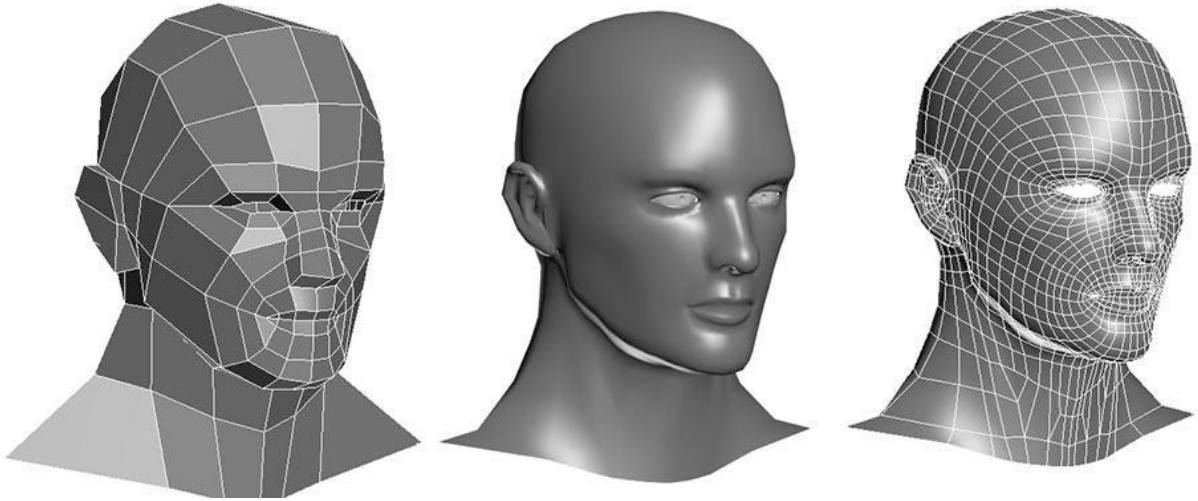
CIS367 - Computer Graphics

Building models / Viewing

Erik Fredericks - frederer@gvsu.edu





What do we need to model things?

(We'll start a bit simpler)

Data structures to manage our models!

- Lists of vertices and edges

The concept of a *mesh:*

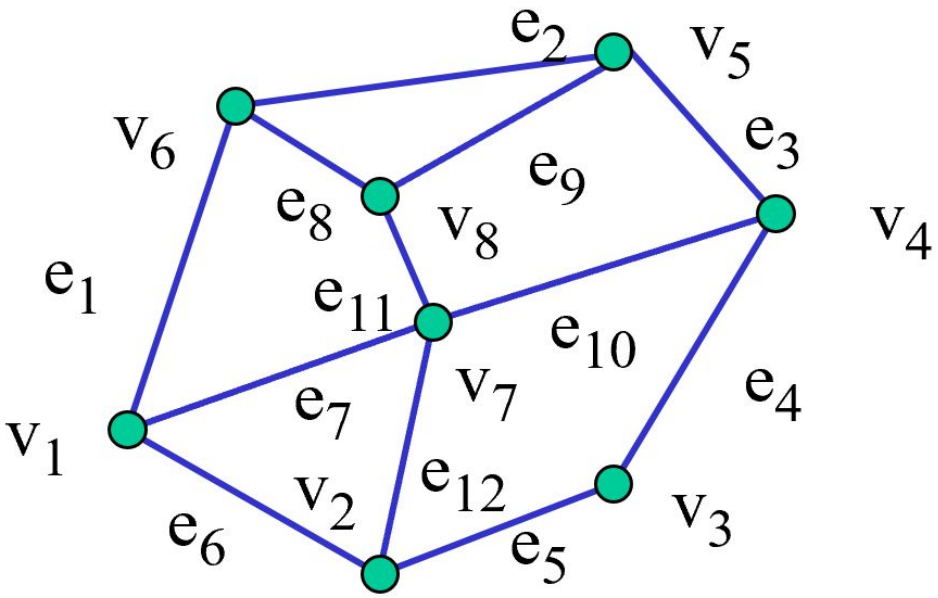
"A polygon mesh is a collection of vertices, edges and faces that defines the shape of a polyhedral object in 3D computer graphics and solid modeling. The faces usually consist of **triangles**, **quadrilaterals**, or **other simple convex polygons**, since this simplifies rendering, but may also be more generally composed of concave polygons, or even polygons with holes" ➔ % WikiPedia

Mesh

8 nodes and 12 edges

- 5 interior polygons
- 6 interior (shared) edges

Each vertex has location:
$v_i = (x_i\ y_i\ z_i)$



(Simple) Representation

Define polygon by geometric locations of vertices

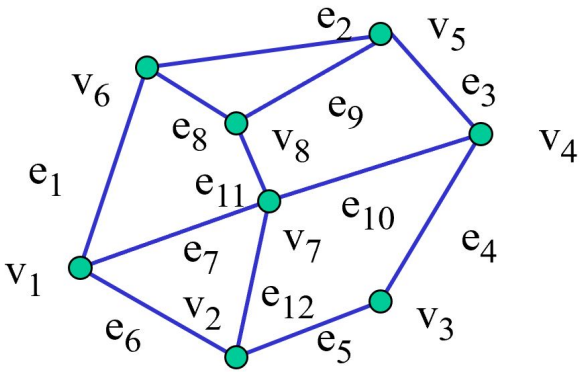
E.g.,

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

Inefficient/unstructured!

- What if we want to move/adjust a vertex?
  - Must search for "all" occurrences of this!!

Inward/Outward facing polygons



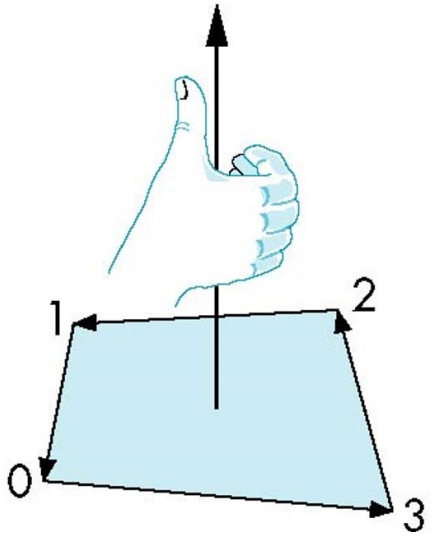
Order $\{v_1, v_6, v_7\}$ and $\{v_6, v_7, v_1\}$ are equivalent in OpenGL

- $\{v_1, v_7, v_6\}$ is different

First two are *outwardly-facing polygons*

Right-hand rule ➔

- Counter-clockwise encirclement of outward pointing normal



- OpenGL/WebGL can treat inward/outward facing polygons differently!

Geometry vs. topology

Data structure should separate geometry and topology

- Geometry ➔ locations of vertices
- Topology ➔ organization of vertices and edges

Example:

Polygon is an ordered list of vertices with an edge connecting successive vertices, plus last to first (LINE_LOOP)

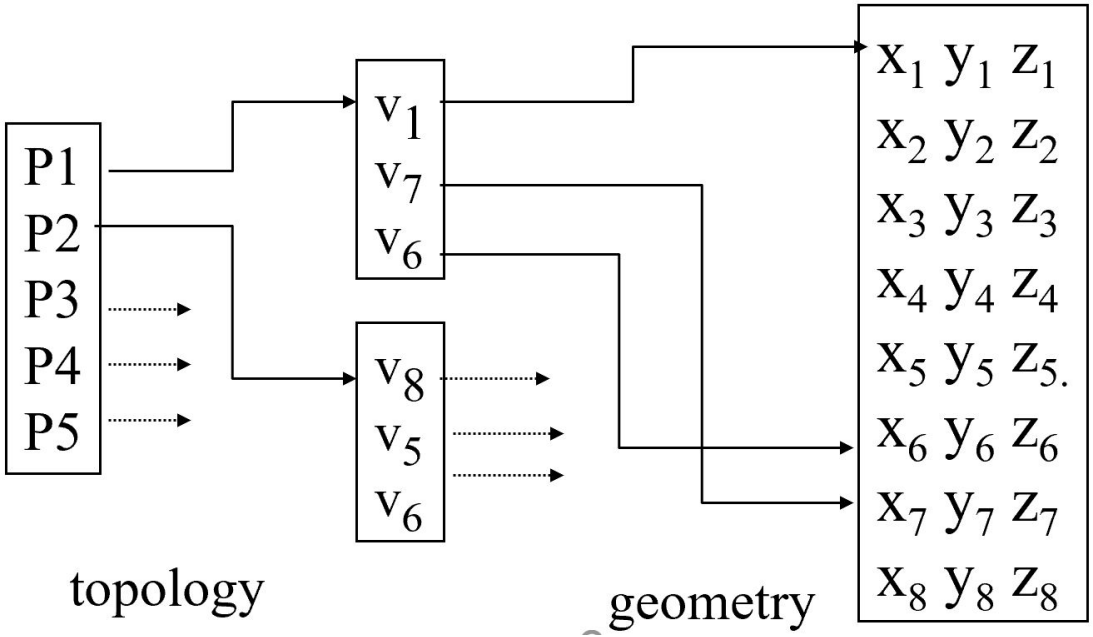
- Topology will hold even if geometry changes!

Vertex lists

Put geometry into array

Use pointers from vertices into array

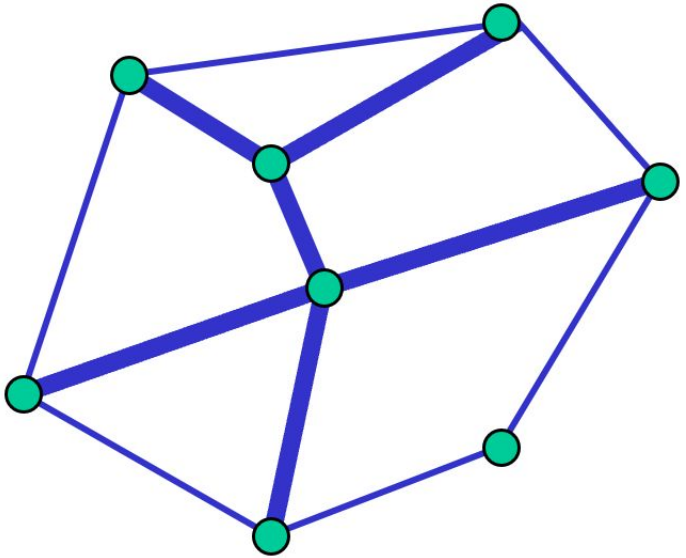
Introduce polygon list



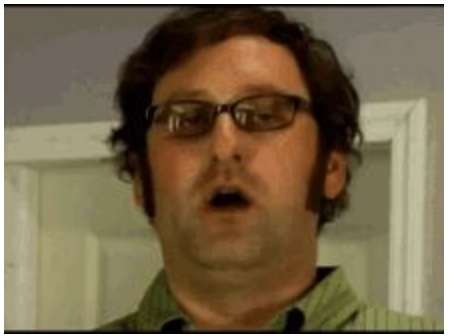
Shared edges

Vertex lists draw filled polygons correctly, but:

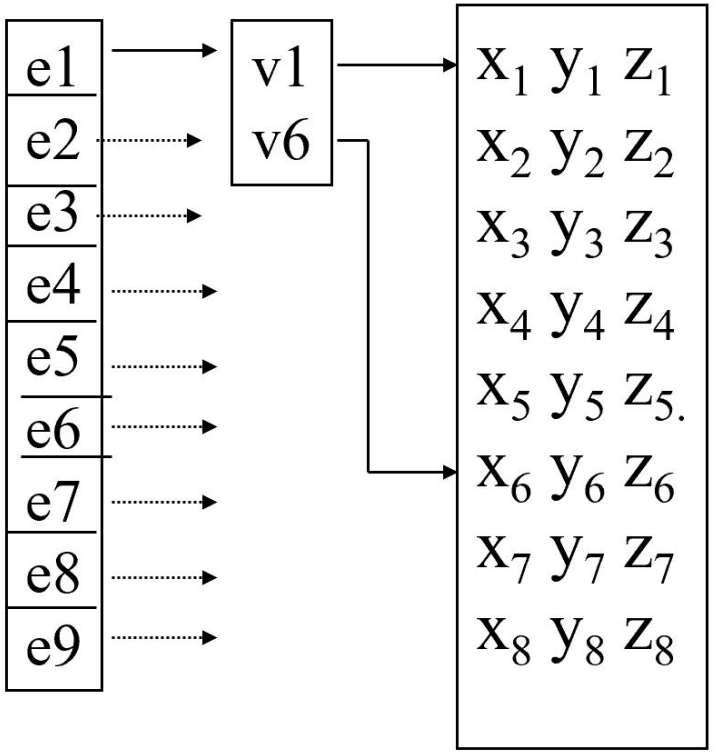
- Drawing by edges will have shared edges drawn more than once
- Store mesh by **edge list**

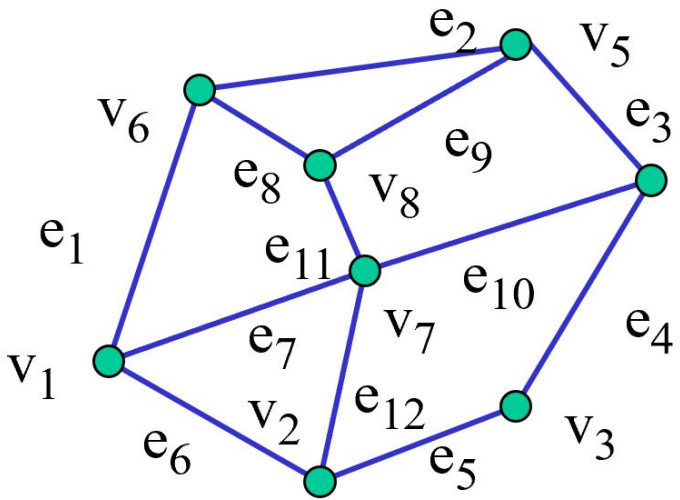


DAY THE SECOND



Edge list

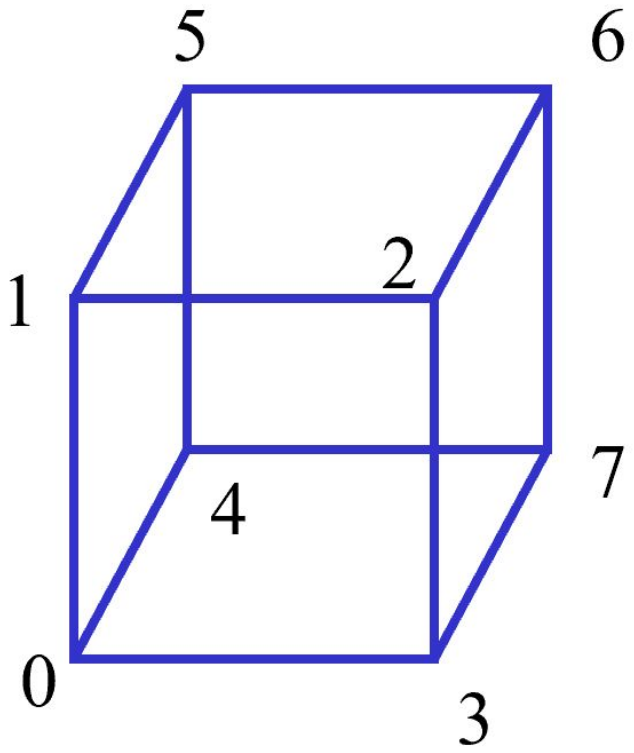




Note polygons are not represented

Draw cube from faces

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



Now let's take this to a rotating cube

We can display either by:

- Arrays
- Elements

Cube modeling

Global array for vertices and colors

```
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 )
];
```

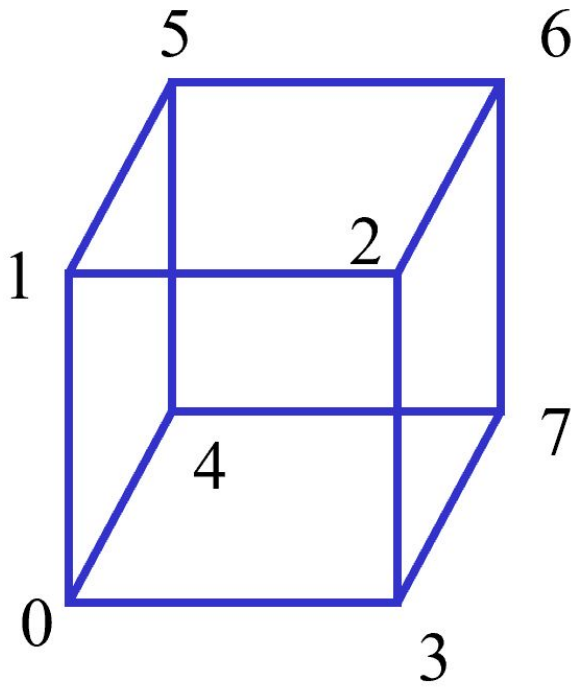
```
var vertexColors = [
    [ 0.0, 0.0, 0.0, 1.0 ],  // black
    [ 1.0, 0.0, 0.0, 1.0 ],  // red
    [ 1.0, 1.0, 0.0, 1.0 ],  // yellow
    [ 0.0, 1.0, 0.0, 1.0 ],  // green
    [ 0.0, 0.0, 1.0, 1.0 ],  // blue
    [ 1.0, 0.0, 1.0, 1.0 ],  // magenta
    [ 0.0, 1.0, 1.0, 1.0 ],  // cyan
    [ 1.0,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

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);
}
```

Vertices ordered so we obtain correct outward-facing normals

Each quad generates 2 triangles!



code.py

```
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
```

quad()

Add position/color data for 2 triangles from list of indices into `vertices` array

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

Index/face mapping

```
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/fans!

Adding **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;
};
```

(updated) render()

```
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 );
}
```

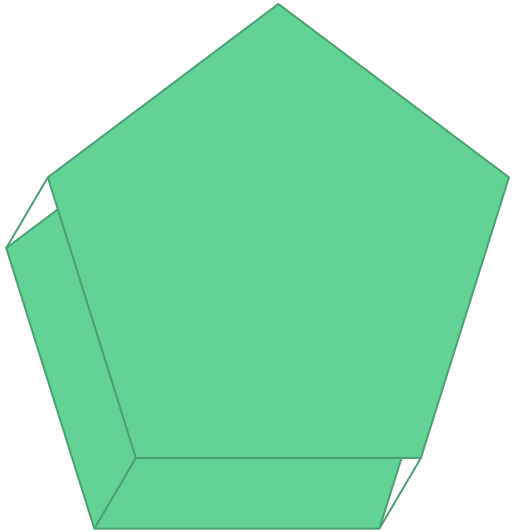
Clearly this uses drawArrays, but you could have used drawElements as well!
Reference ➜ https://www.tutorialspoint.com/webgl/webgl_drawing_a_model.htm

Some in-class things

How would you track a 3-dimensional pentagon?

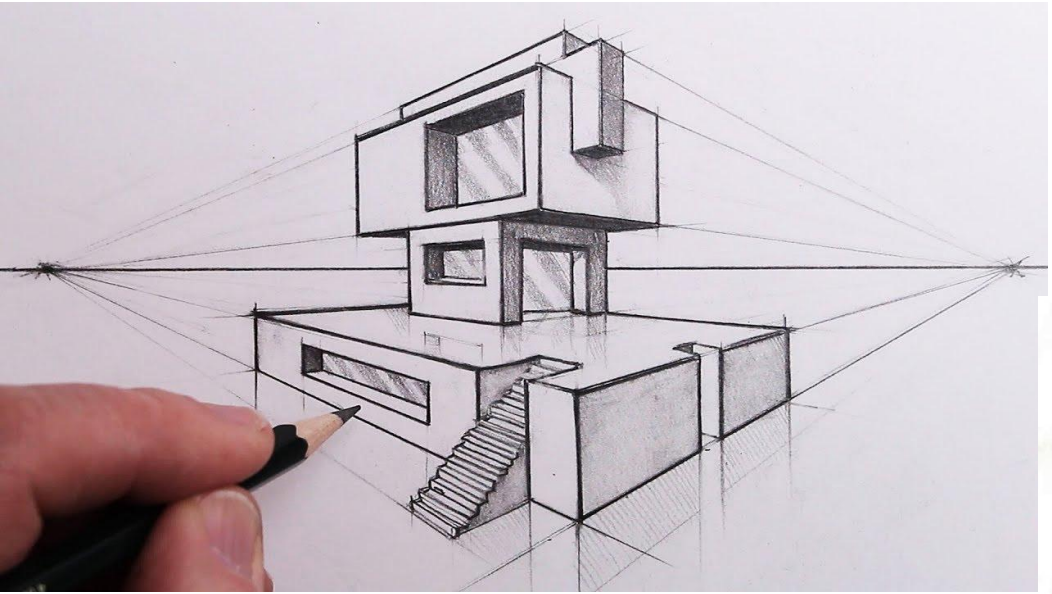
WRITE DOWN:

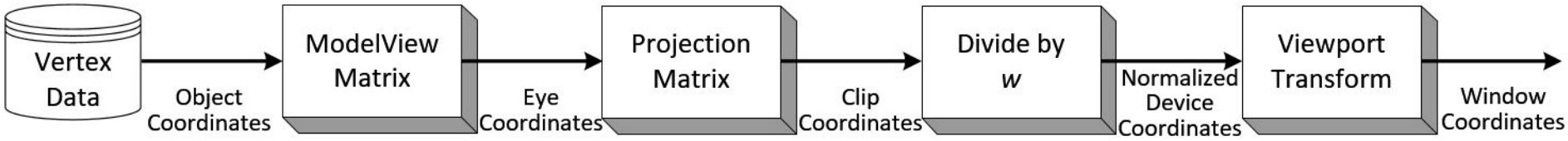
1) **Number the vertices**
2) **Create a list of either:**
    a) **Faces**
    b) **Edges**
3) **Create a list of vertices**
4) **Link them up in the proper order!**



Assume we have a `pentagon` function to draw a triangle fan

Time to get some *perspective*





https://www.songho.ca/opengl/gl_transform.html

Let's talk about views and how they work

This will be a little bit on the high-level side

Classical viewing

Viewing requires **three** basic elements:

- *n* objects
- Viewer with projection surface
- Projects from object(s) to projection surface

Classical views based on relationship between these three!

- Viewer picks object and orients based on preference

Each object assumed to be made of **flat principle faces**

- Buildings, polyhedra, etc.

Planar geometric projections

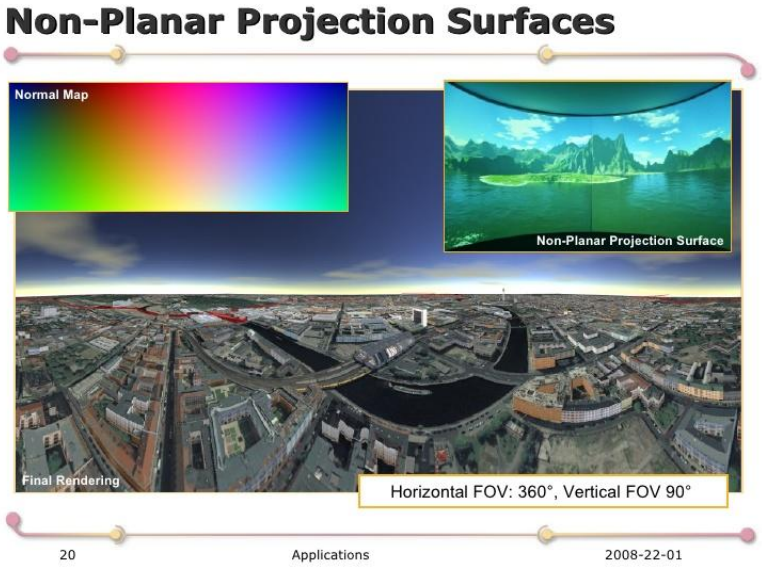
Standard projects … project … onto a plane

Projectors are lines that either:

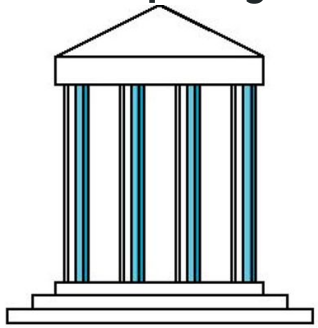
- Converge @ center of projection (CoP)
- Parallel

Projections preserve lines, not necessarily angles

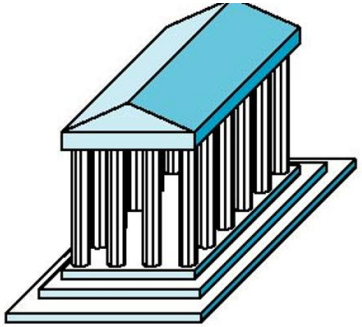
Non-planar can be useful as well:



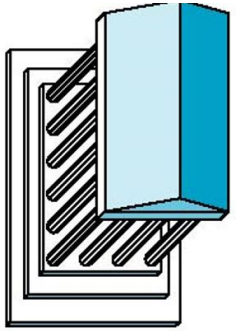
Classical projections



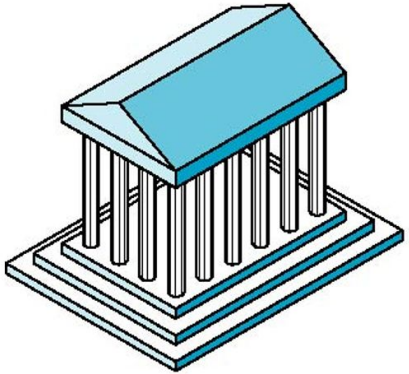
Front elevation



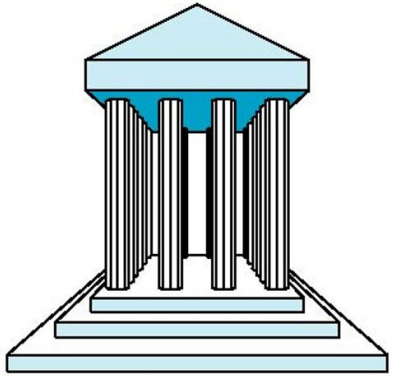
Elevation oblique



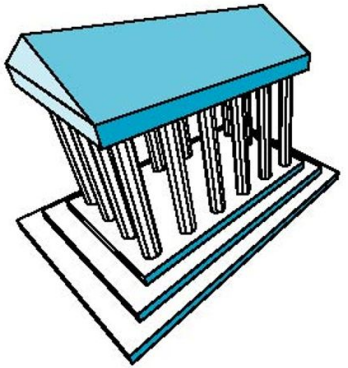
Plan oblique



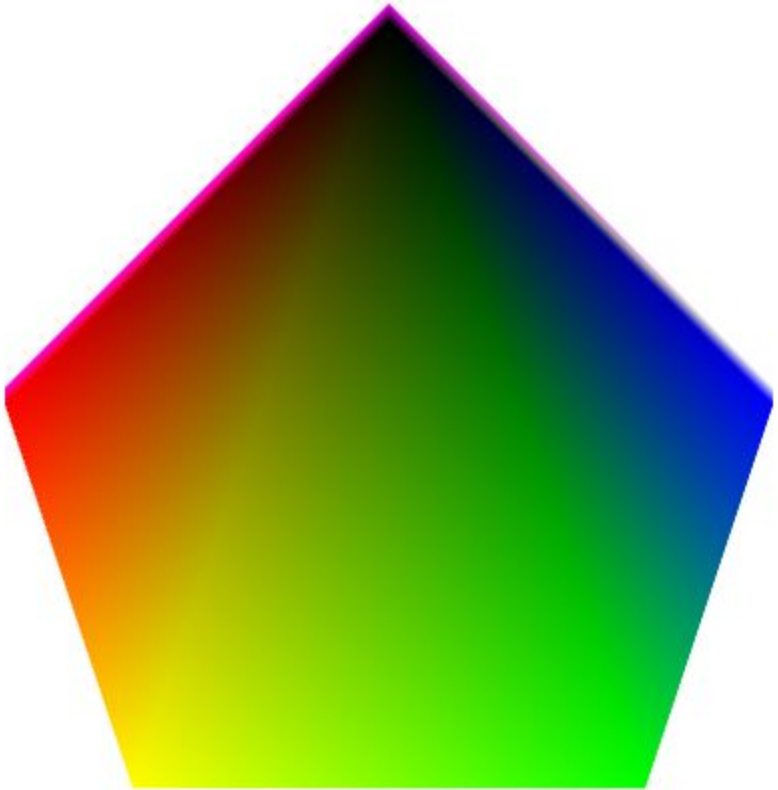
Isometric



One-point perspective



Three-point perspective

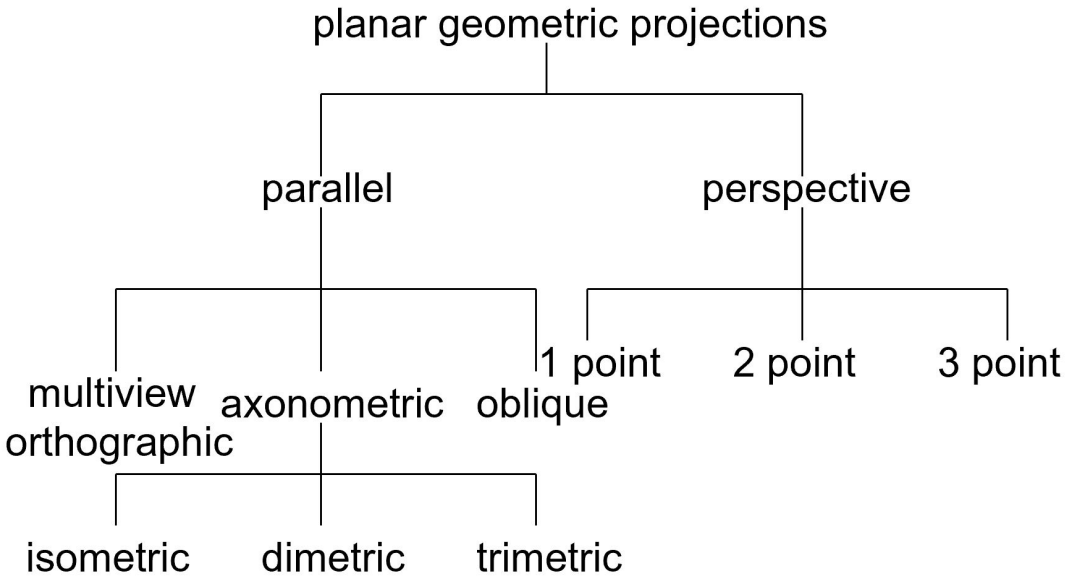


https://efredericks.github.io/CIS367-ComputerGraphics/demos/pentagon-class.html

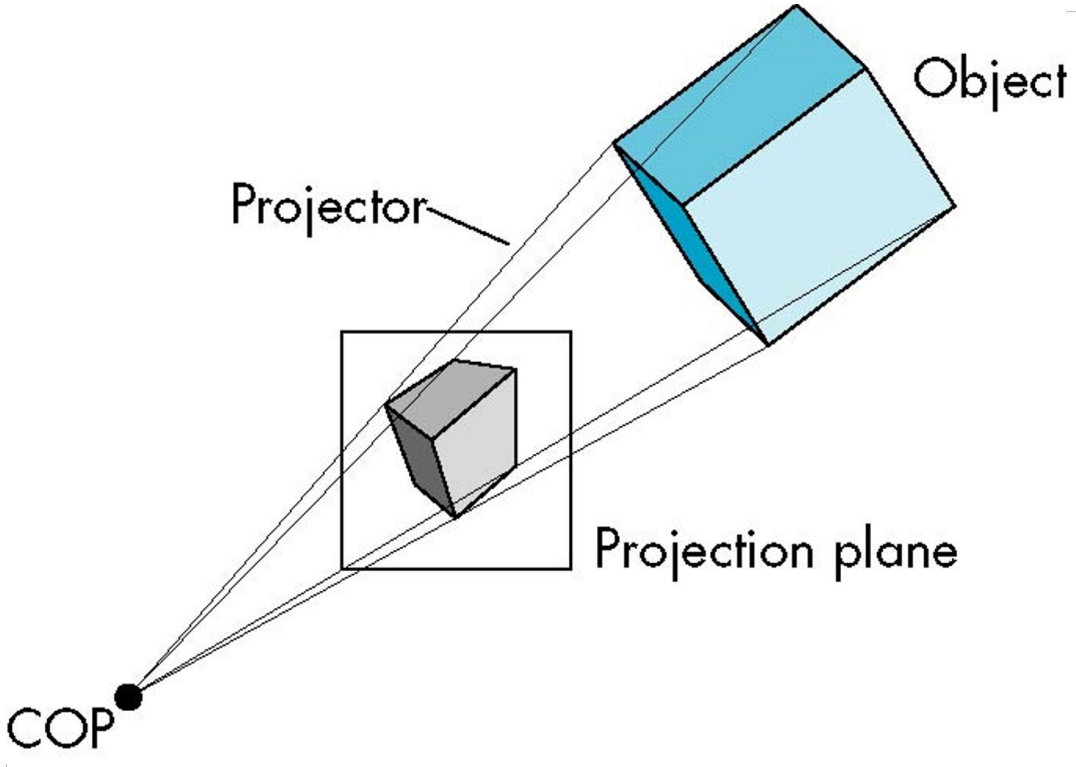
Perspective vs. parallel

All projections are treated the same and implemented in single pipeline (in gfx)

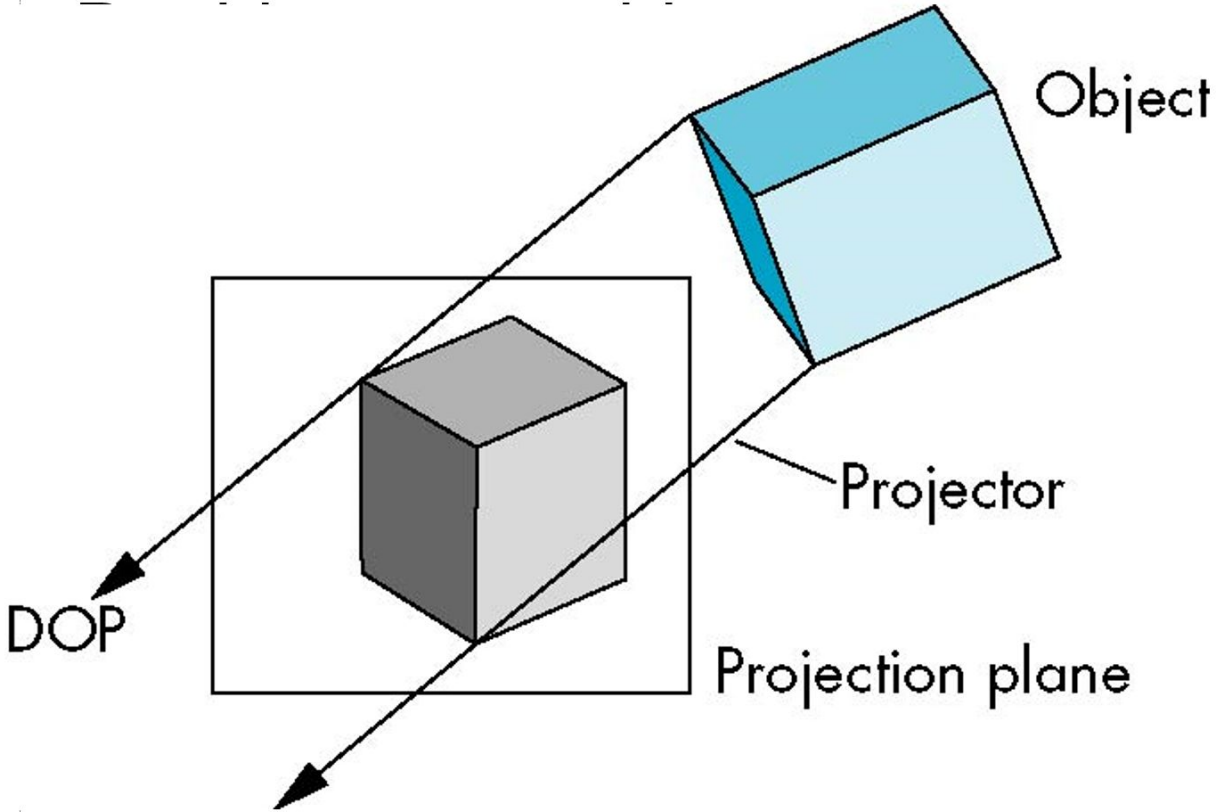
Fundamental distinction is between parallel and perspective viewing



Perspective projection

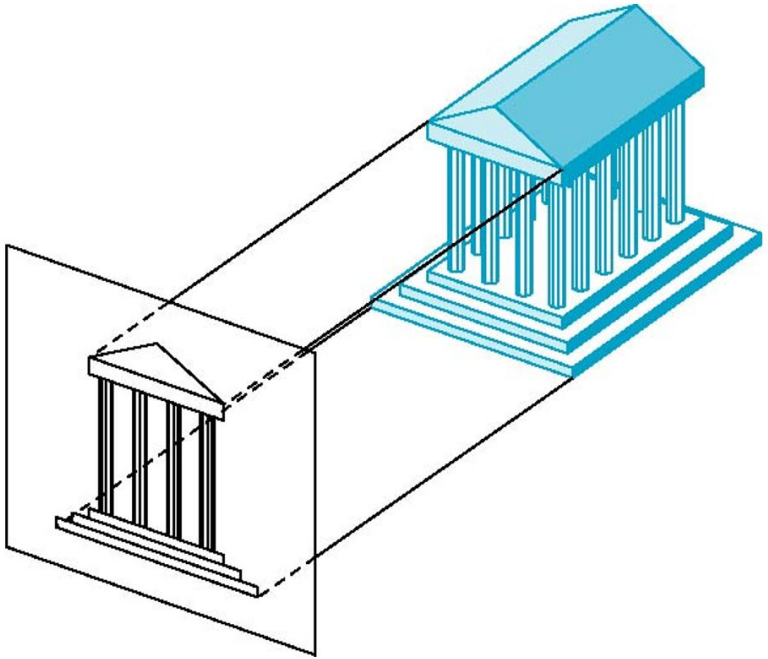


Parallel projection



Orthographic projection

Projectors orthogonal to projection surface



Multiview orthographic projection

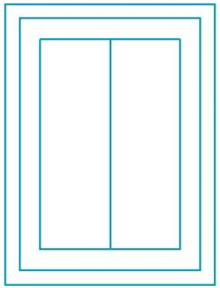
Projection plane parallel to principal face
Generally: front, top, side views

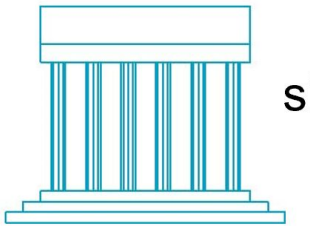
isometric (not multiview orthographic view)

front

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

top





side

Advantages/disadvantages

Preserves distances and angles

- Shapes preserved
- Can be used for measurements
  - Building plans
  - Manuals
  - etc.

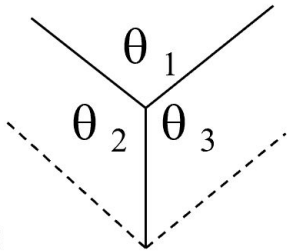
Can't see what object *really* looks like given hidden surfaces

- Isometric view can reveal hidden areas

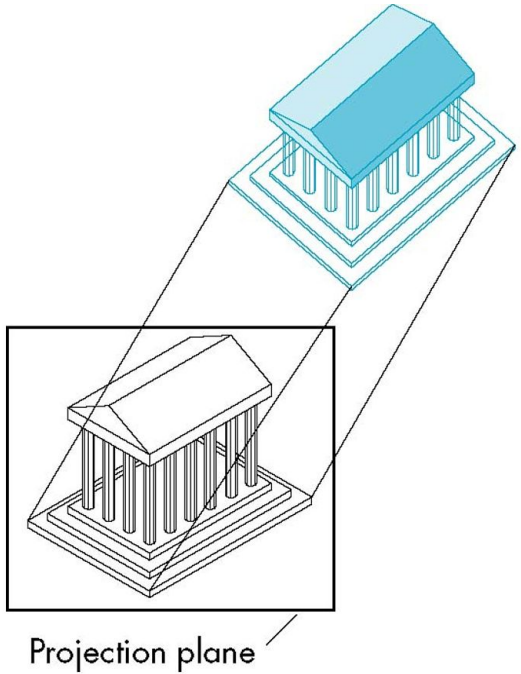
Axonometric projections

Projection plane moves relative to object

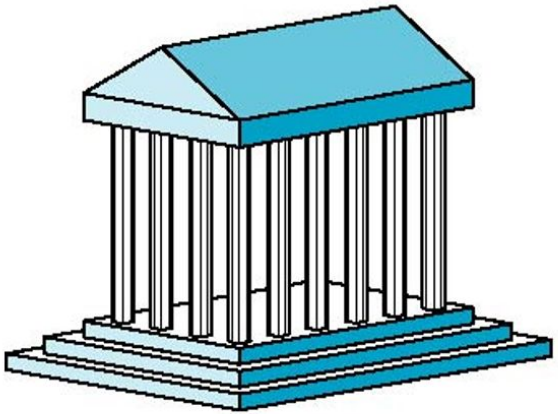
classify by how many angles of a corner of a projected cube are the same



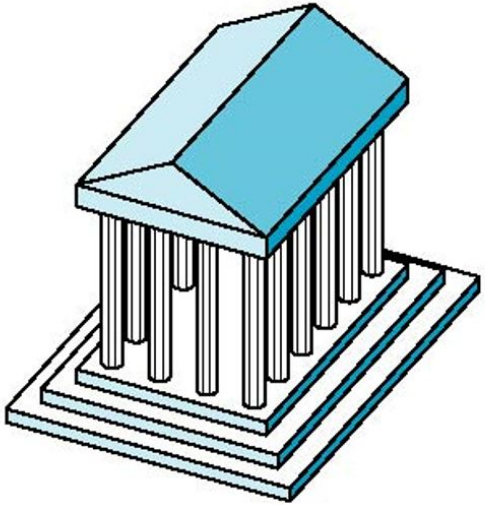
none: trimetric
two: dimetric
three: isometric



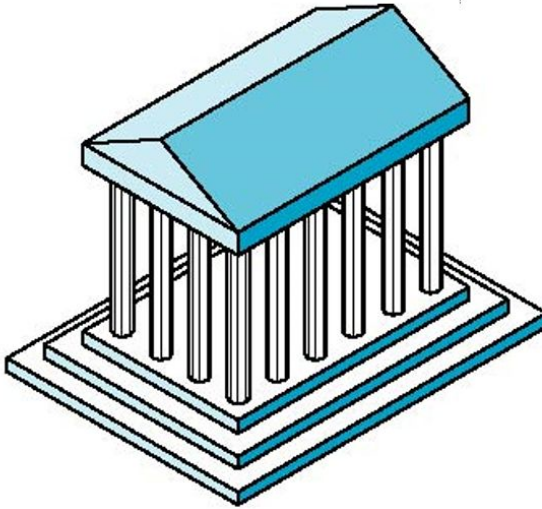
Axonometric types



Dimetric



Trimetric



Isometric

Advantages/disadvantages

Lines scaled (foreshortened), but can find scaling factors

Lines preserved but angles are not

- Projection of circle in a plane not parallel to projection plane is an ellipse

Can see 3 principal faces of box-like object

Optical illusions possible

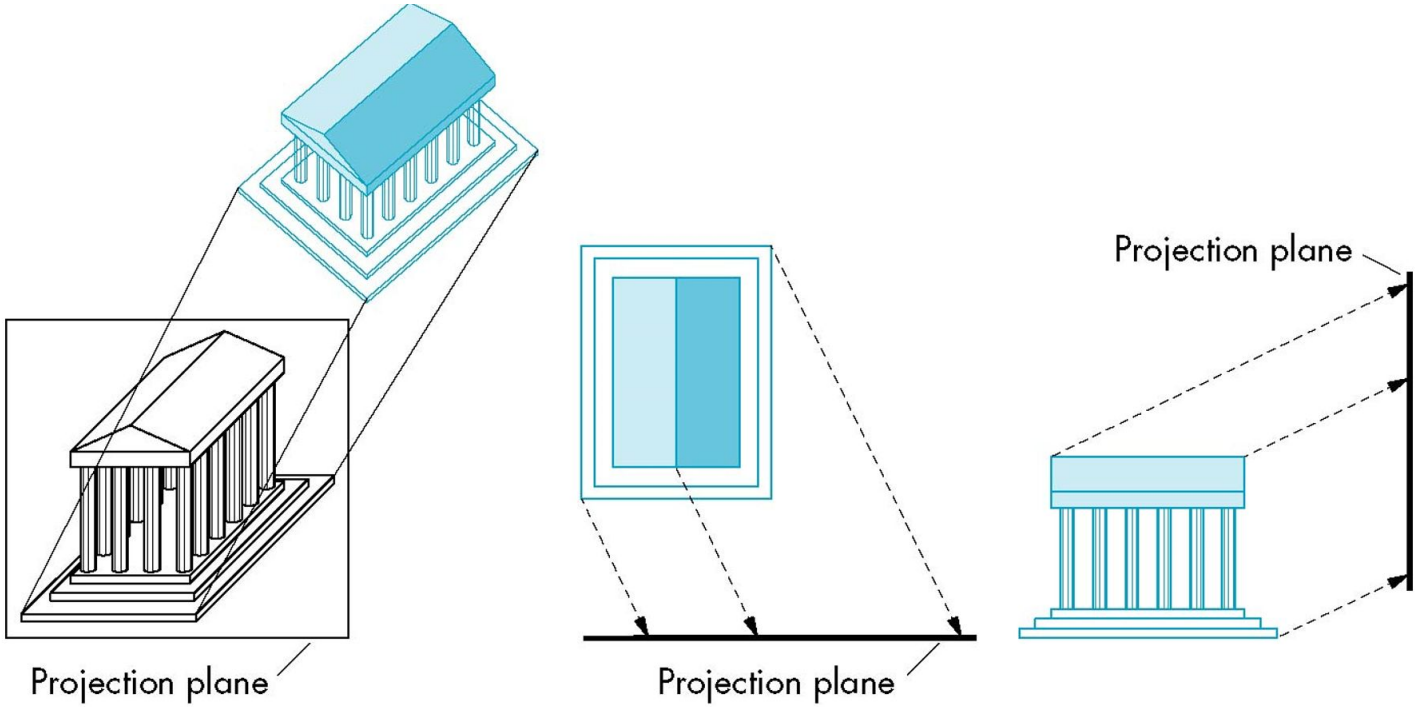
- Parallel lines appear to diverge

Does not look real as far objects scaled same as near objects

But, used in CAD applications

Oblique projection

Arbitrary relationship between projectors and projection plane

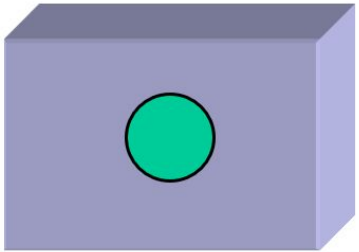


Advantages/disadvantages

Can pick angles to emphasize particular face

- Architecture: plan oblique, elevation oblique

Angles in faces parallel to projection plane preserved while we can see "around" side



Physical world ➔ cannot create with simple camera

- Possible with bellows camera or special lens

Interestingly

From a programmer's perspective, we just pick our view and specify parameters related to that view

- Viewing angle, distance, etc.

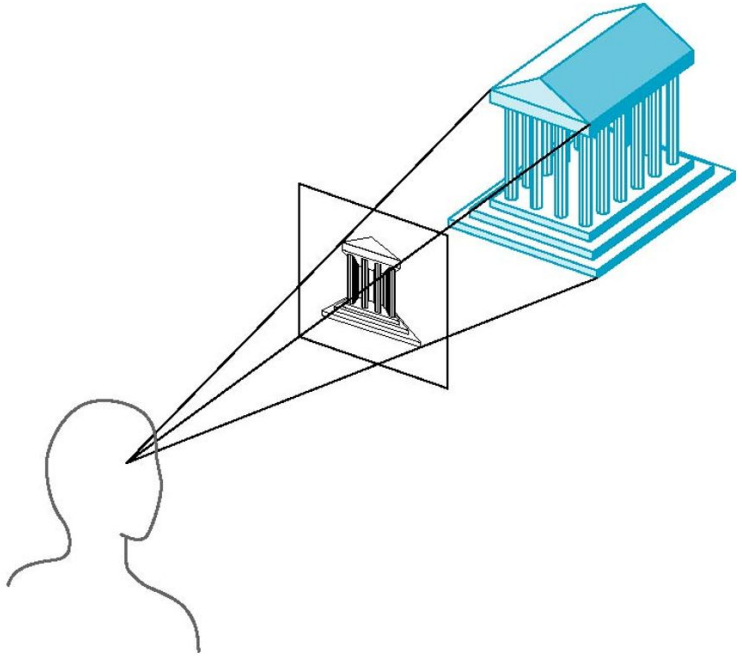
Issue for us is how to approximate any of these 'classical' viewing angles

Perspective projection

Projectors converge at center of projection

Characterized by **diminution** of size

- Objects get smaller as moved farther away from viewer
    - Parallel -- not the case!

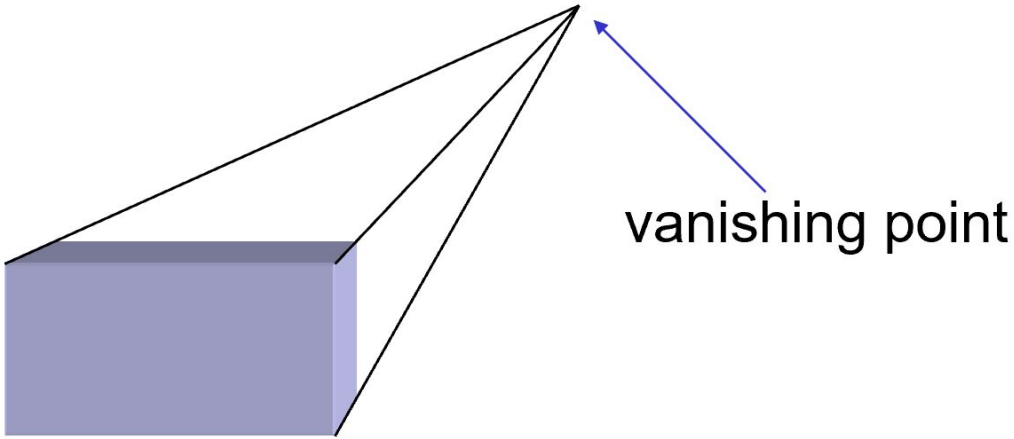


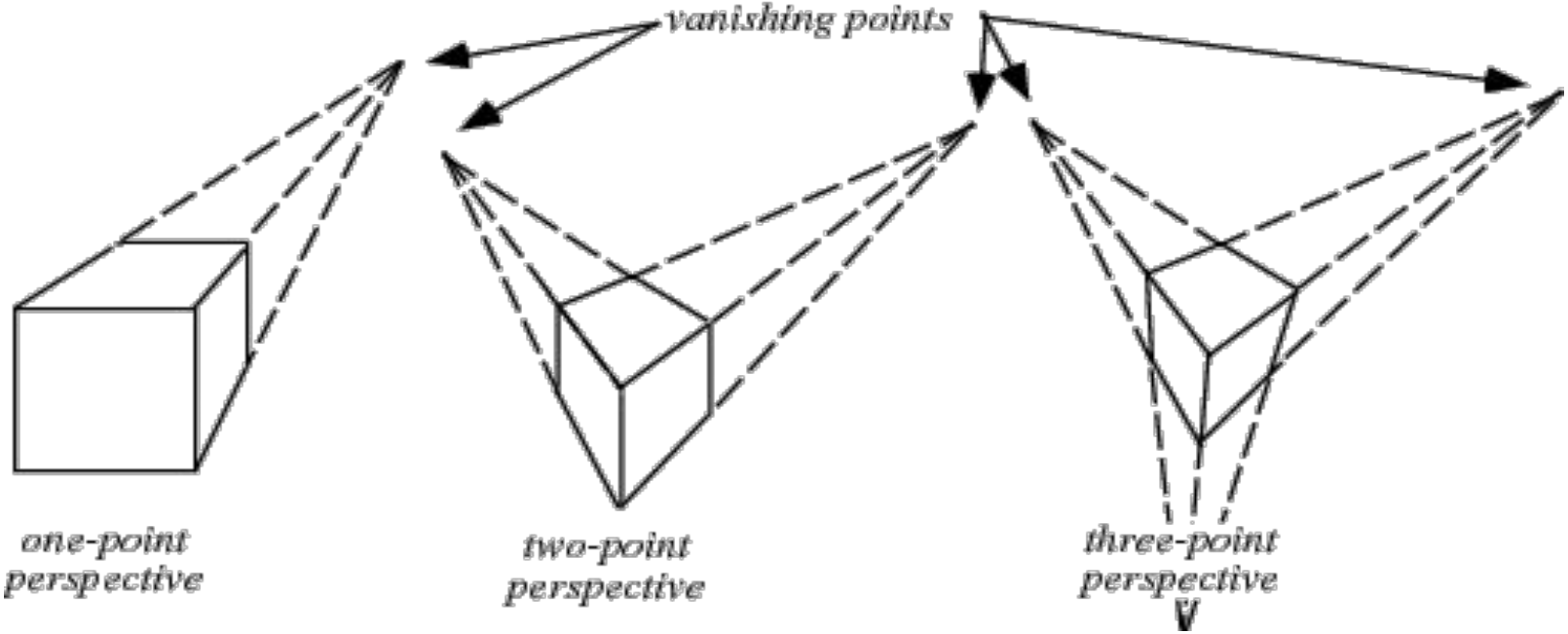
Vanishing points

Parallel lines (not parallel to projection plane) on object converge at single point in projection

- The *vanishing point*

Simple perspectives by hand use vanishing point

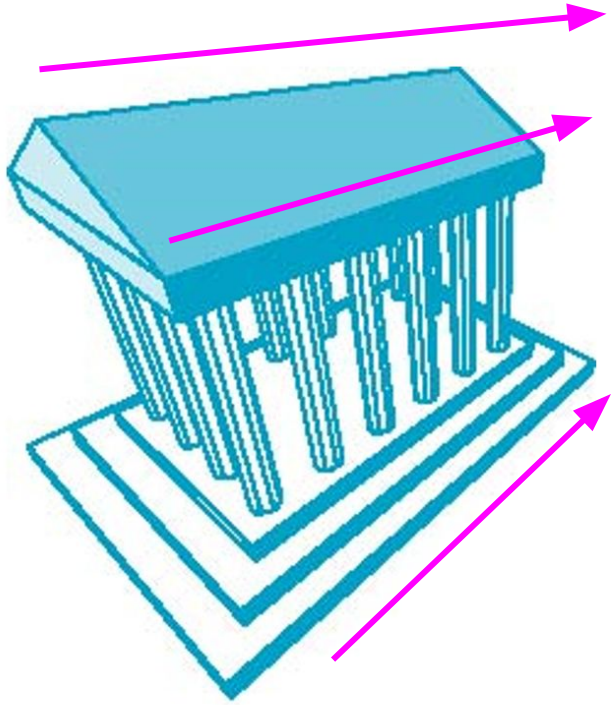



vanishing points
one-point perspective
two-point perspective
three-point perspective

Three-point perspective

No principal direction parallel to projection plane

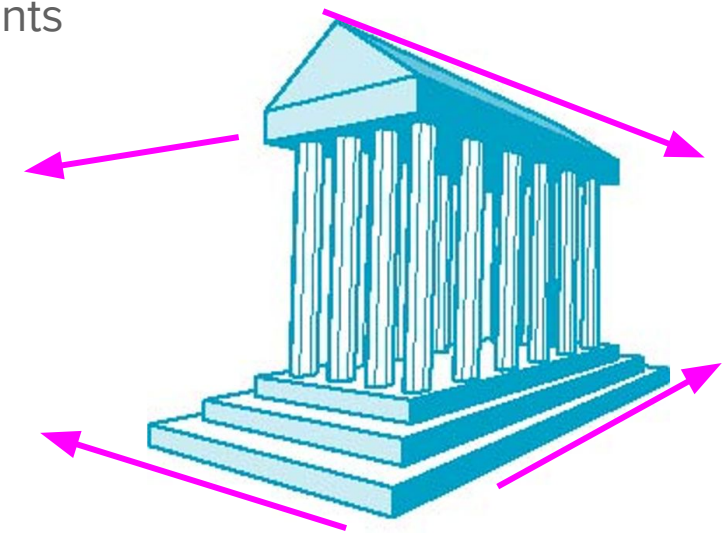
- Three vanishing points
- Three lines converge to vanishing point



Two-point perspective

Two lines converge to vanishing point for cube on principal direction, parallel to projection plane

- Two vanishing points



One-point perspective

One principal face parallel to projection plane

- One vanishing point per cube

Advantages/disadvantages

Objects further from viewer projected smaller than same-sized objects closer to viewer

- Diminution ➔ looks realistic

Equal distances along line not projected into equal distances

- Non-uniform foreshortening

Angles preserved only in planes parallel to projection plane

More difficult to construct by hand than parallel

- NOT more difficult for computer!

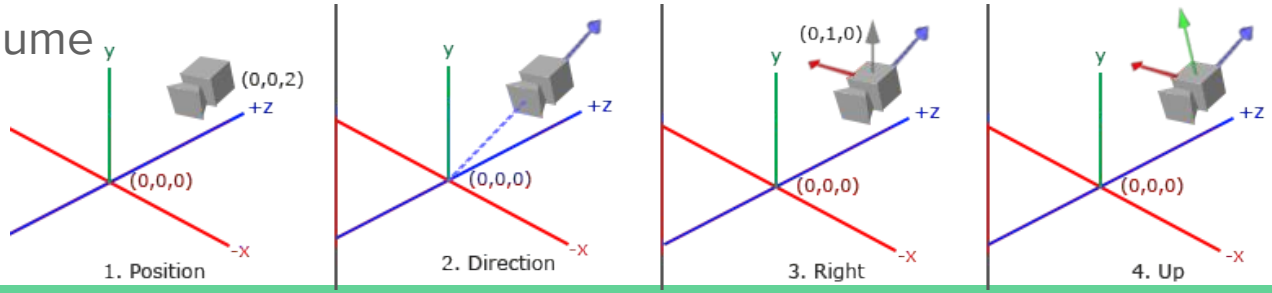
Now, let's position that camera

Three aspects to consider:

1) **Position the camera**
   - Set the model-view matrix

2) **Select a lens**
   - Set the projection matrix

3) **Clip**
   - Set the view volume



WebGL camera

Object and camera frames are the same initially

- This is what we've seen so far!
- Default model-view matrix is an **identity matrix**

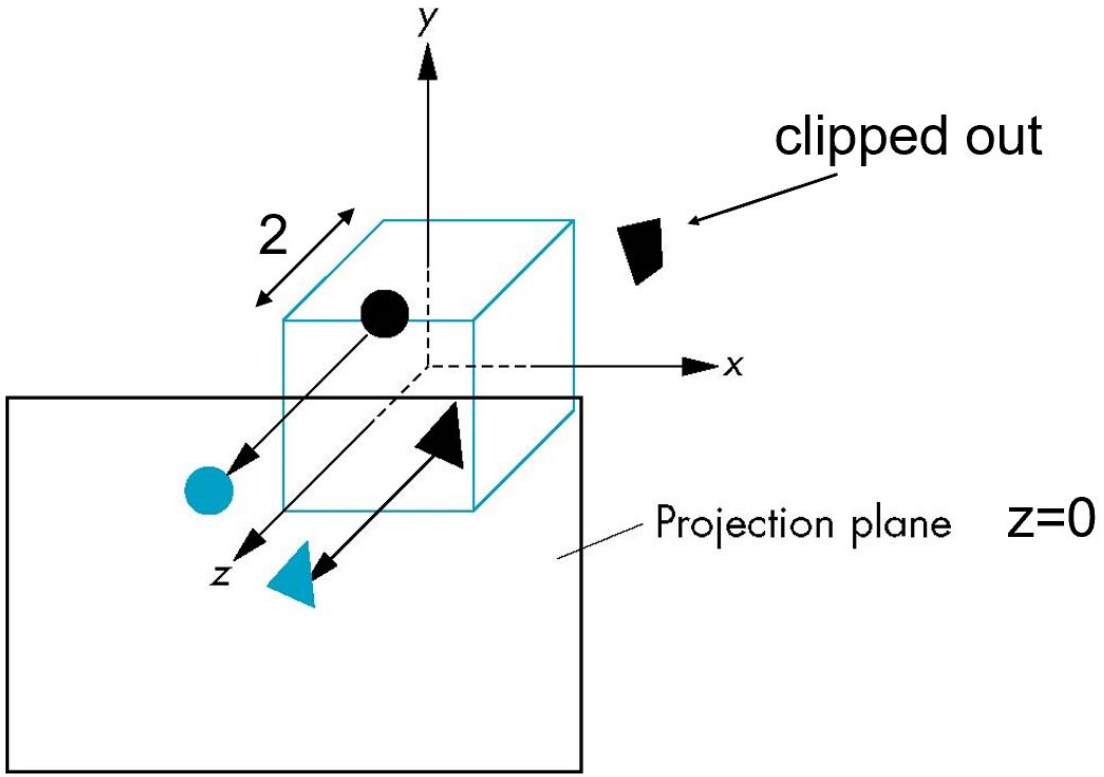
Camera is located at origin, points in negative z direction

WebGL also specifies default **view volume**

- Cube with sides of length 2
- Centered at origin
- Default projection matrix is also an **identity matrix**

Default projection

Orthogonal!



Moving the camera frame

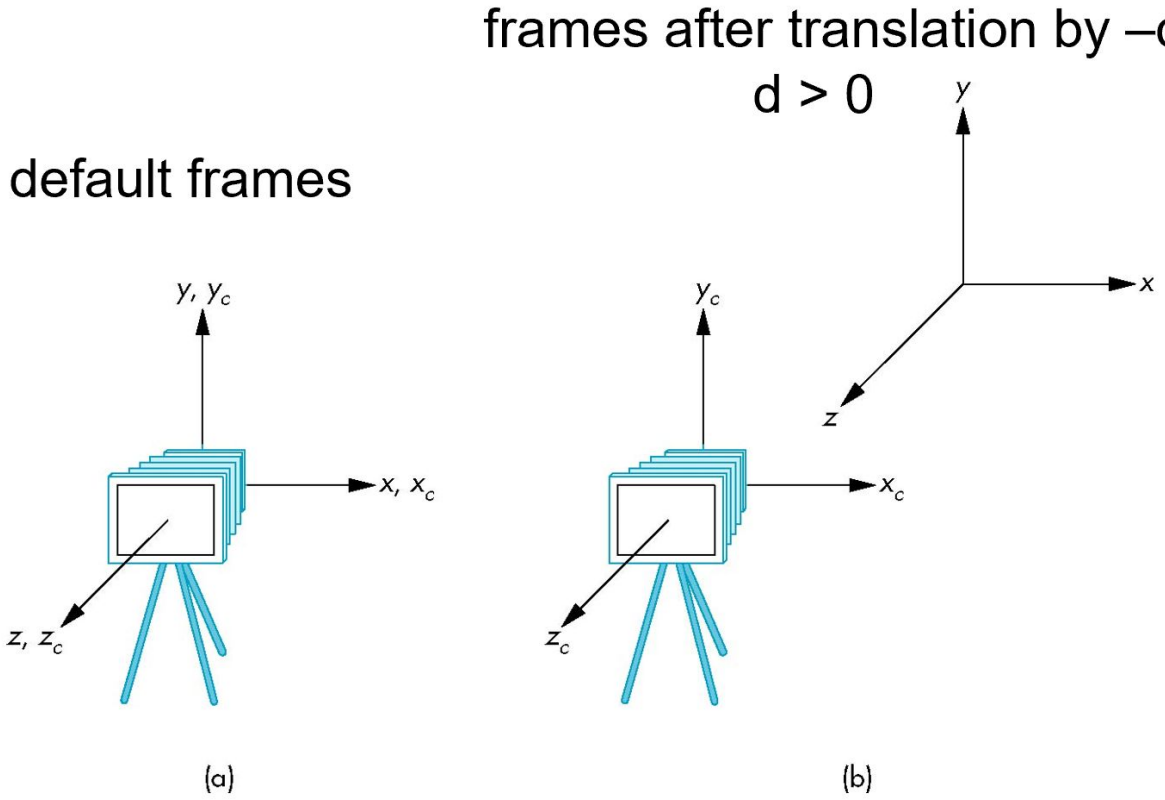
If we want visualizations with **both** positive and negative z values, either:

- Move camera in positive z (translate camera frame)
- Move objects in negative z (translate world frame)

Both views are **equivalent** and determined by *model-view matrix*

Translation: `translate(0.0, 0.0, -d); // d > 0`

Move camera back to origin



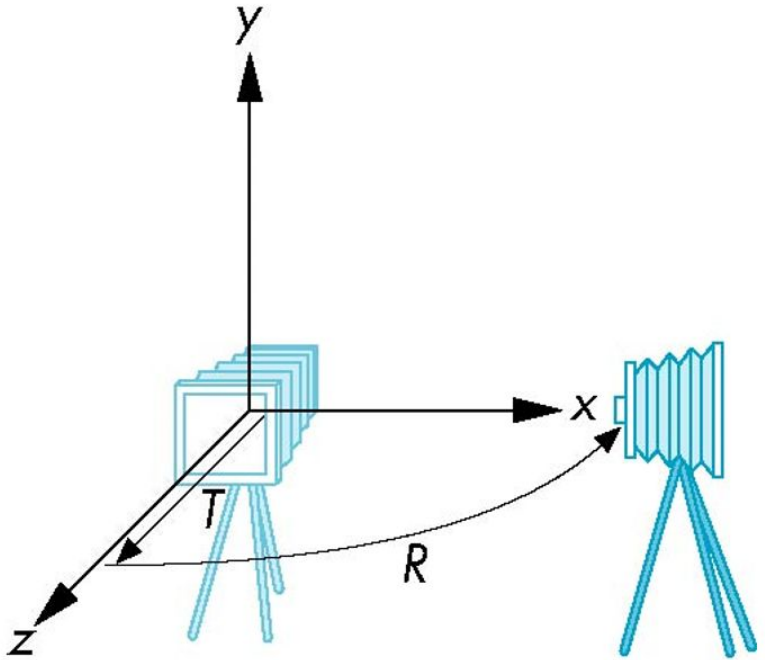
Moving the camera

Move camera to any desired position/orientation via series of rotations and translations

Side view:

- Rotate camera
- Move away from origin

- Model-view matrix: **C** = **TR**



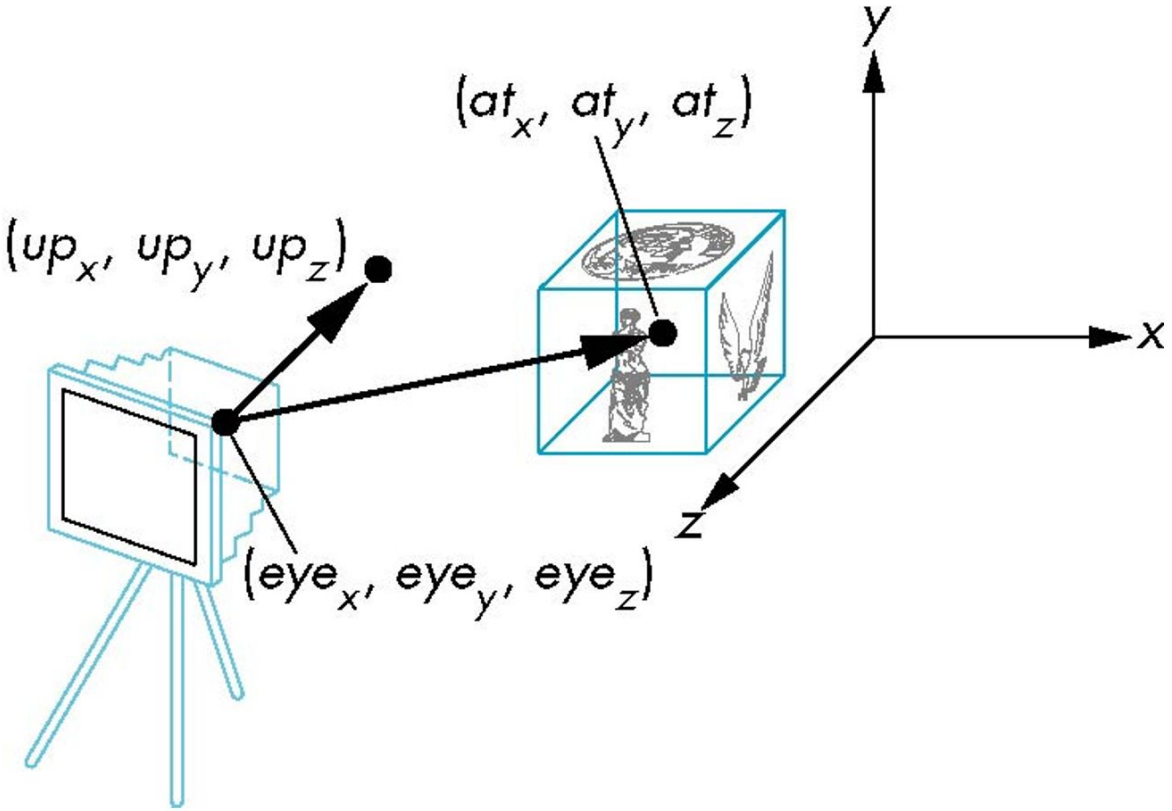
The code

// last xform is first 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));
```

lookAt

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



LookAt

(History) GLU library contained a `gluLookAt` function to form model-view matrix

`LookAt()` defined in MV.js

- Concatenation feasible

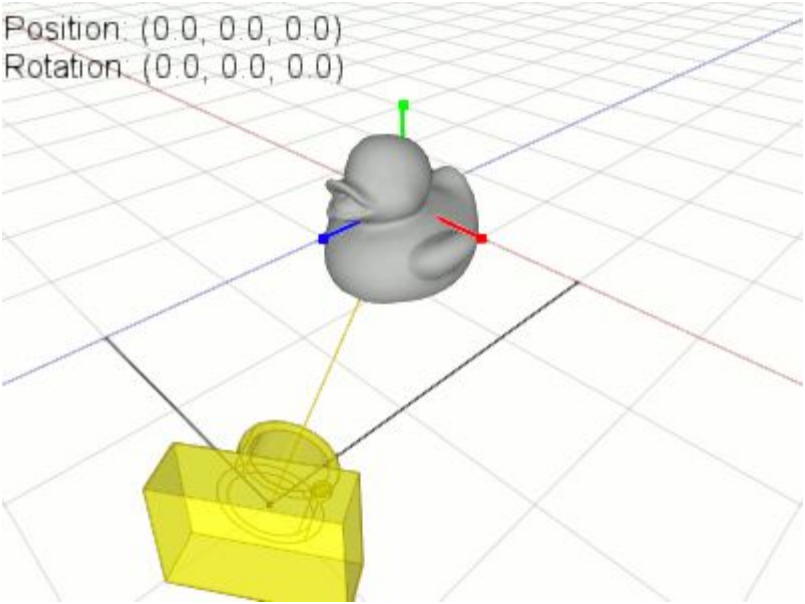
Use example: isometric view of cube aligned with axes

```
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);
```

In-Class Things!

How would you do this?

Write down the *generalized* steps
for some sweet, sweet in-class points



Resume lecture . gif

That was the model-view matrix

Now, the projection matrix!

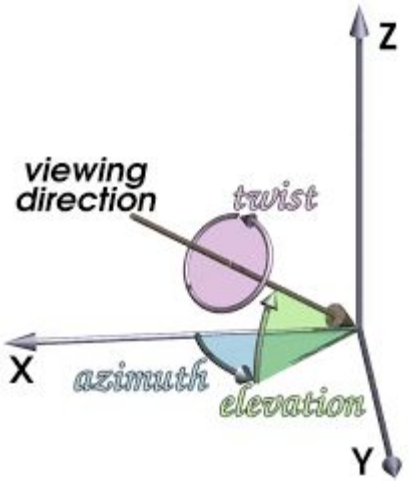
LookAt

Just one method of positioning camera

- Not standard ➜ provided in custom library

Also available:

- View reference point / plane normal / view up
- Yaw, pitch, roll
- Elevation, azimuth, twist
- Direction angles



DEMO

ortho.html/js, if we haven't beaten it to death already

Now for parallel projection

Default projection (eye/camera) is *orthogonal*

Points in default view volume:

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

Graphics systems tend to use *view normalization*

- All other views converted to default view by projection matrix transforms
- Enables same pipeline for all views!

Homogeneous coordinate representation

Default orthographic:

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

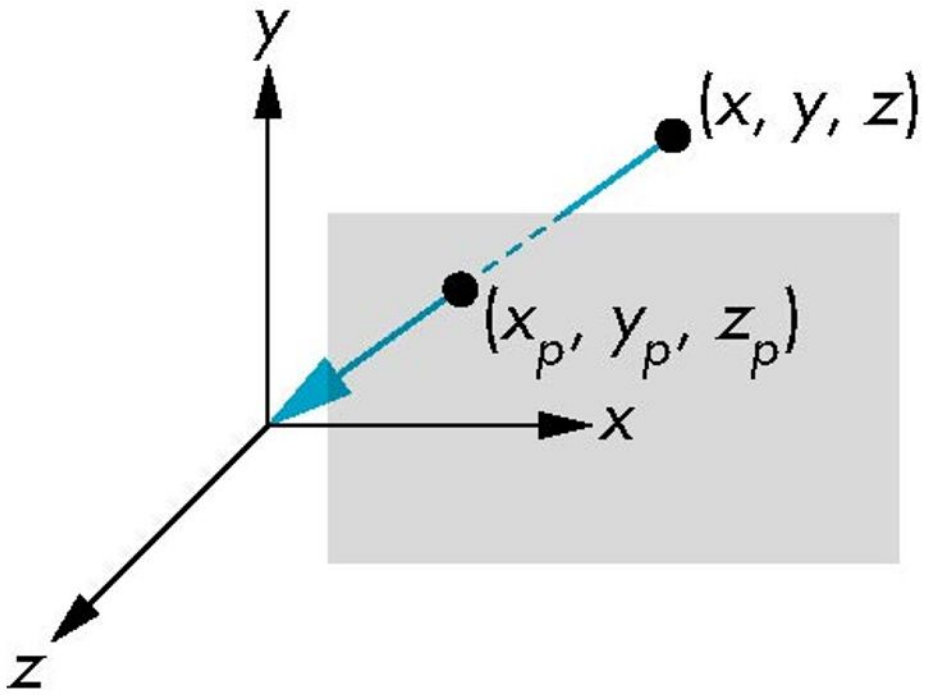
$$\mathbf{p}_p = \mathbf{M}\mathbf{p}$$

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Let **M** = **I** and set z to 0 later

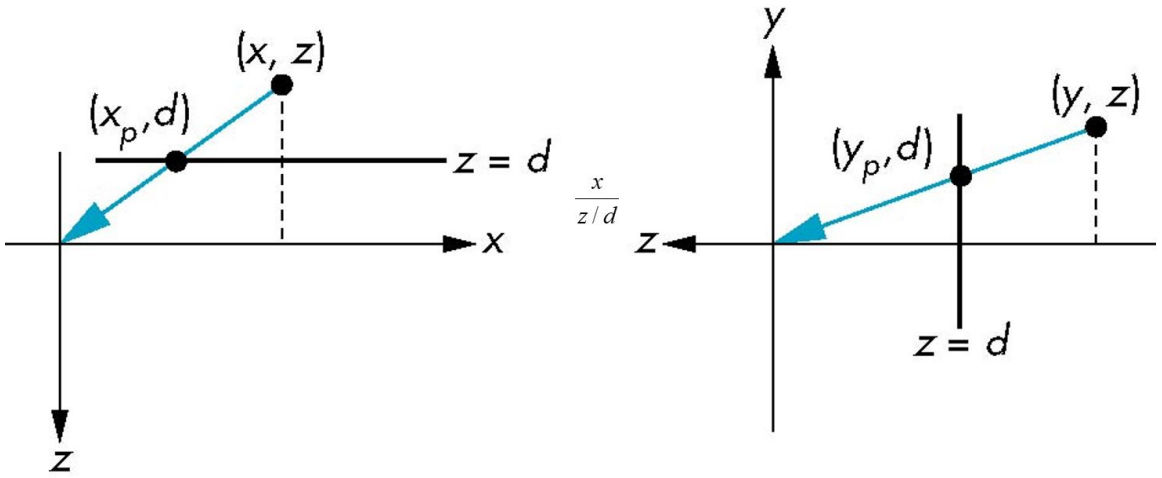
Simple perspective

CoP @ origin
Projection plane: $z = d$, $d < 0$



Perspective equations

Top and side views:



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

Take it to homogeneous coordinate form

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

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

Perspective division

In this case, *w != 1*, so must divide by *w* to return from homogeneous coordinates

Perspective division:

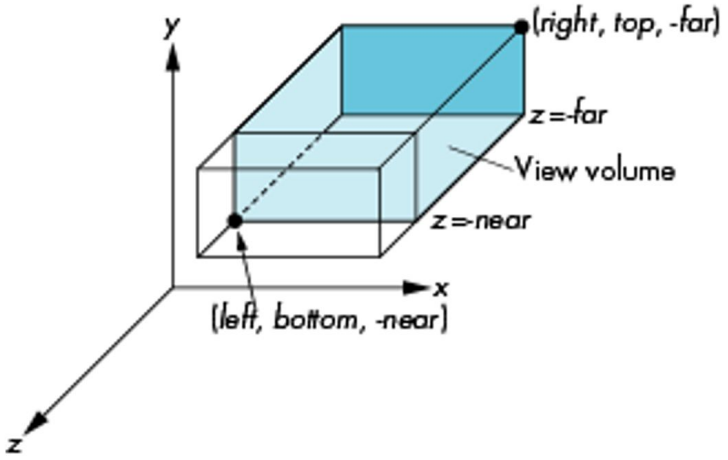
$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

yields desired perspective equations


gl

Orthogonal viewing

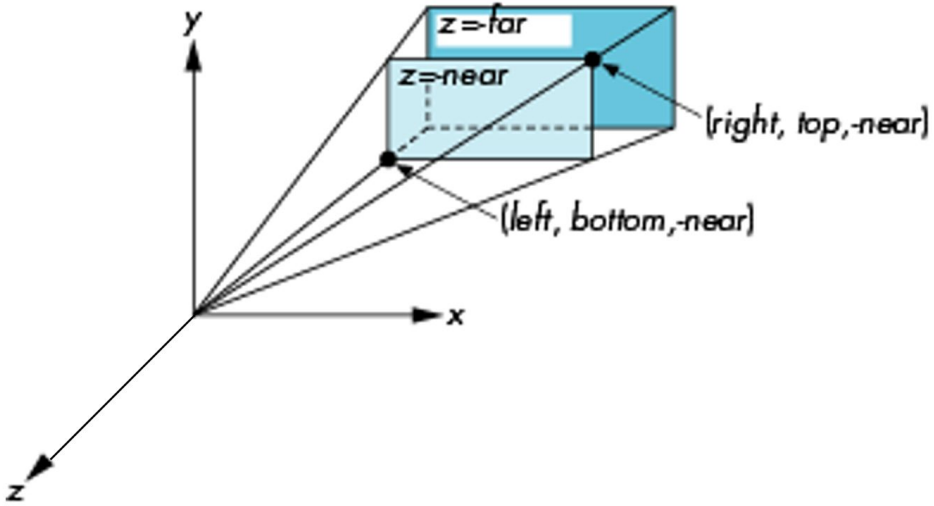
`ortho(left,right,bottom,top,near,far)`



`near` and `far` measured from camera

Perspective

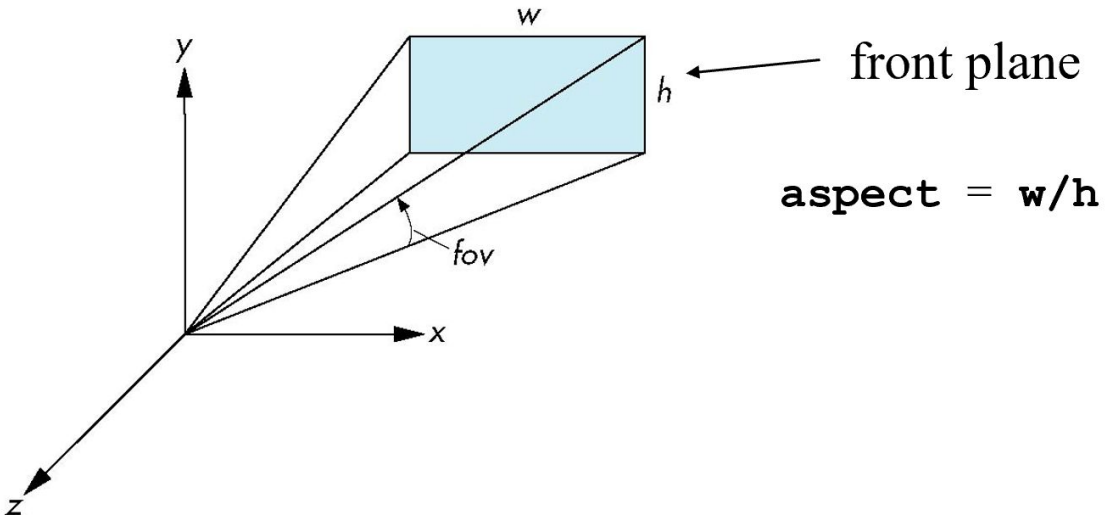
```
frustum(left,right,bottom,top,near,far)
```



Field of view

Often difficult to get desired view with `frustum`

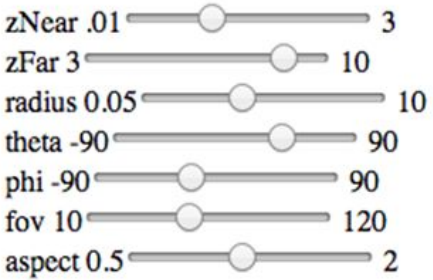
`perspective(fovy, aspect, near, far)` provides a better interface

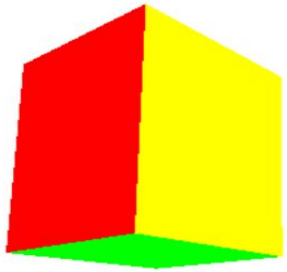


Computing matrices

1) Compute in JavaScript (application)
2) Send to vertex shader with gl.uniformMatrix4fv

If dynamic:
Update in render() or shader





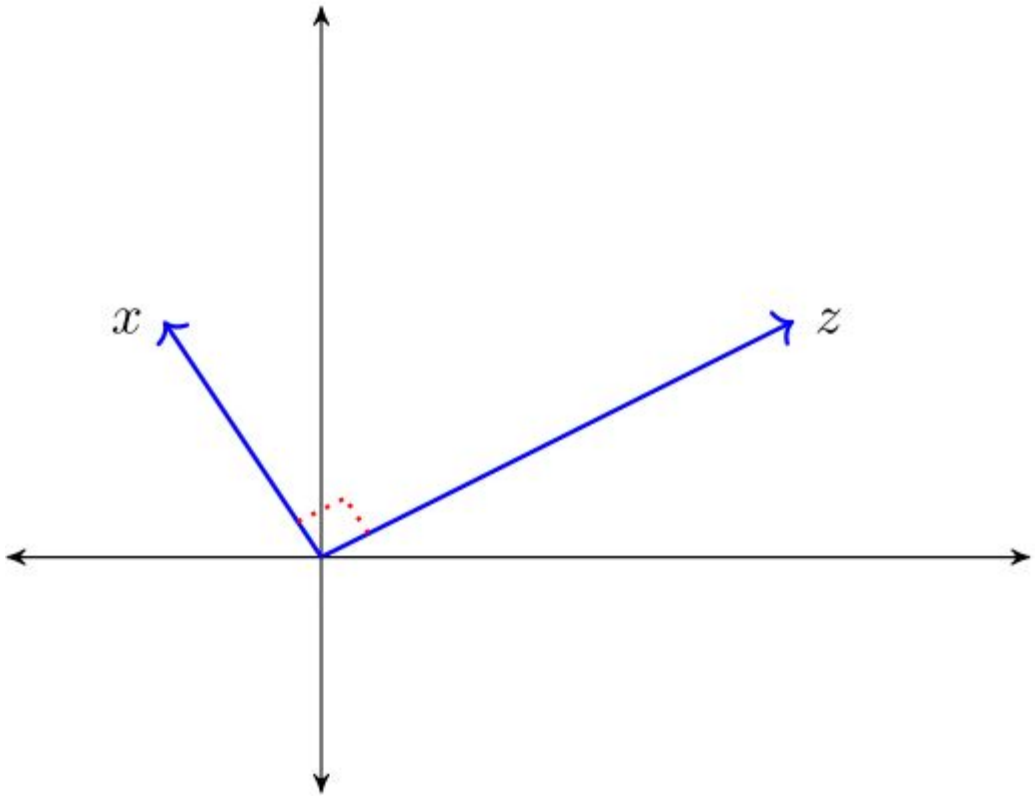
perspective2.js

```
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;
}
```

Orthogonal Projection Matrices



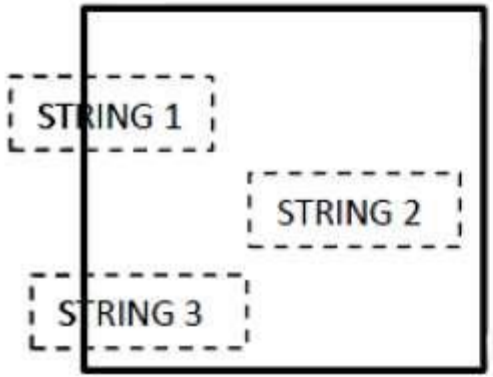
Projection normalization

Convert all projections to orthogonal with default view volume

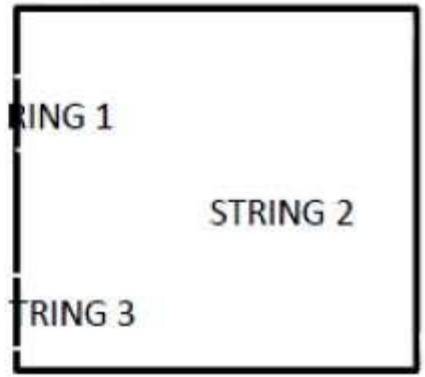
- Rather than derive different projection matrix for **each type**

Use standard transformations in pipeline

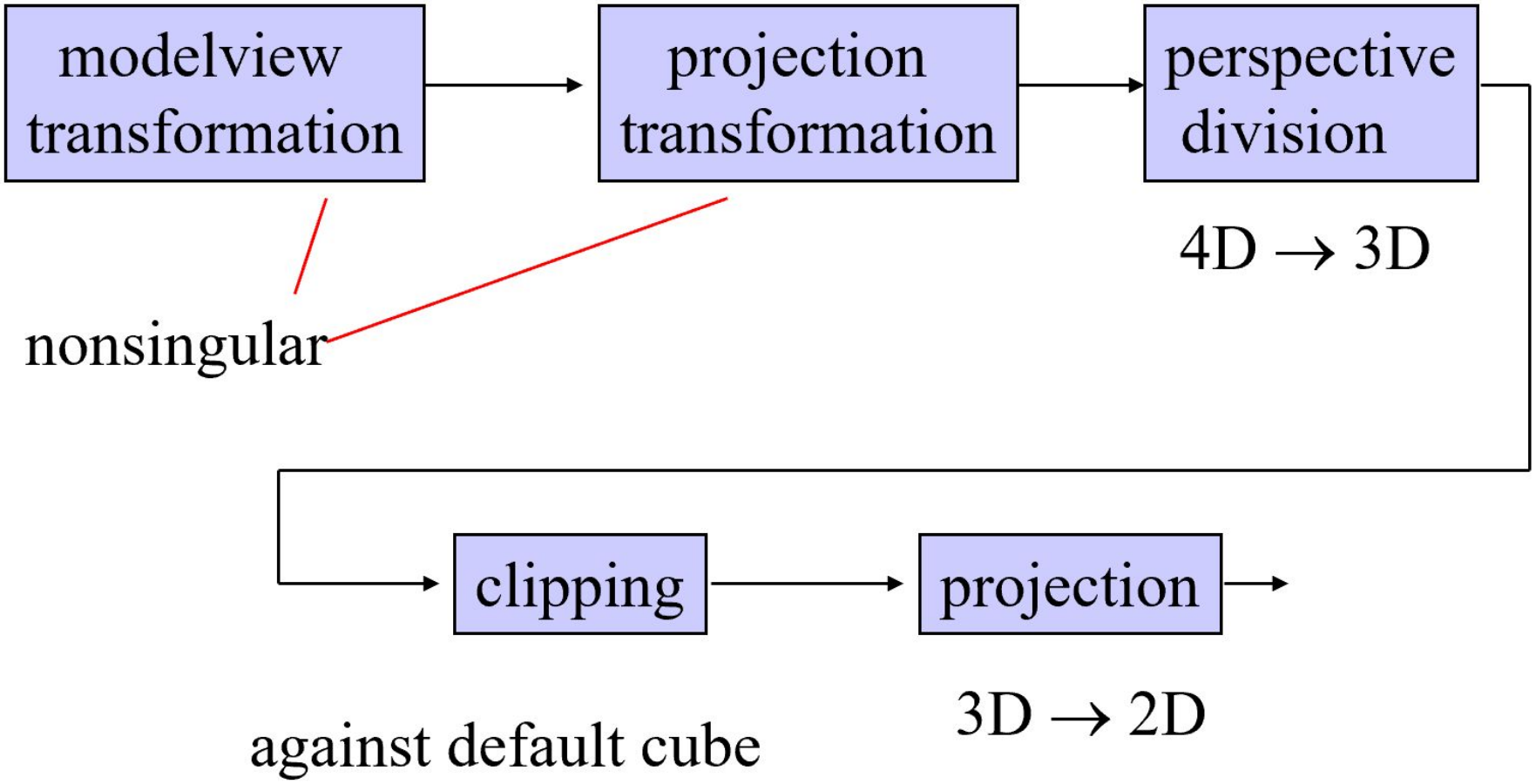
- Makes for efficient clipping



Before Clipping



After Clipping


modelview transformation
projection transformation
perspective division
4D → 3D
nonsingular
clipping
projection
3D → 2D
against default cube

Notes

Stay in 4-dimensional homogeneous coordinates through model-view / projection transforms

- Both transformations are **non-singular**
  - **Non-singular** -- has an inverse
- Default to identity matrices (orthogonal view!)

Normalization lets us clip against a single cube **regardless** of projection type

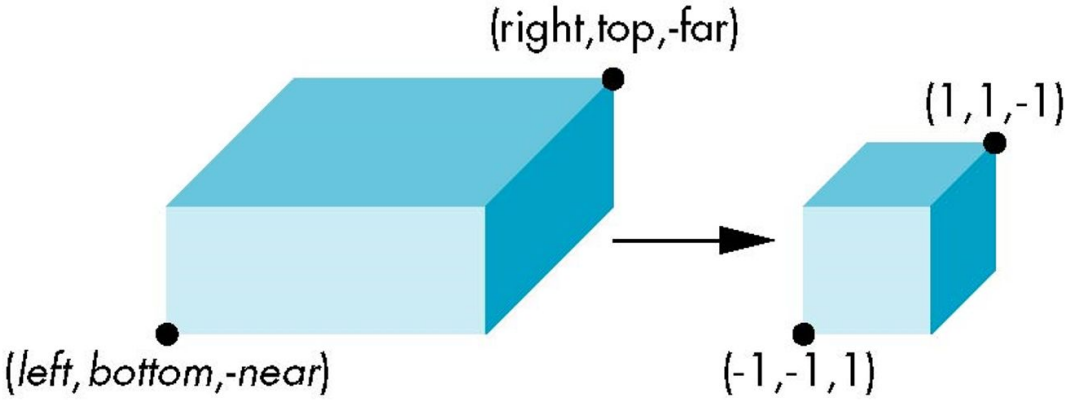
Delay final projection until end

- Important for hidden surface removal
  - Retain depth as long as possible!

Orthogonal normalization

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

normalization ⇒ find transformation to convert specified clipping volume to default



What is happening (orthogonal matrix)

Two steps:

1) Move center to origin

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

1) Scale to have sides of length 2

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

What is happening (orthogonal matrix)

Two steps:

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

Final projection

Set $z = 0$

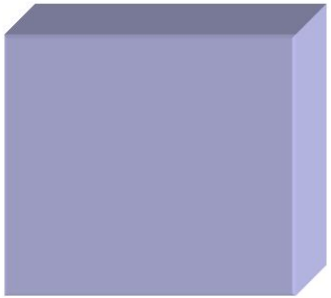
(Equivalent to homogeneous coordinate transformation)

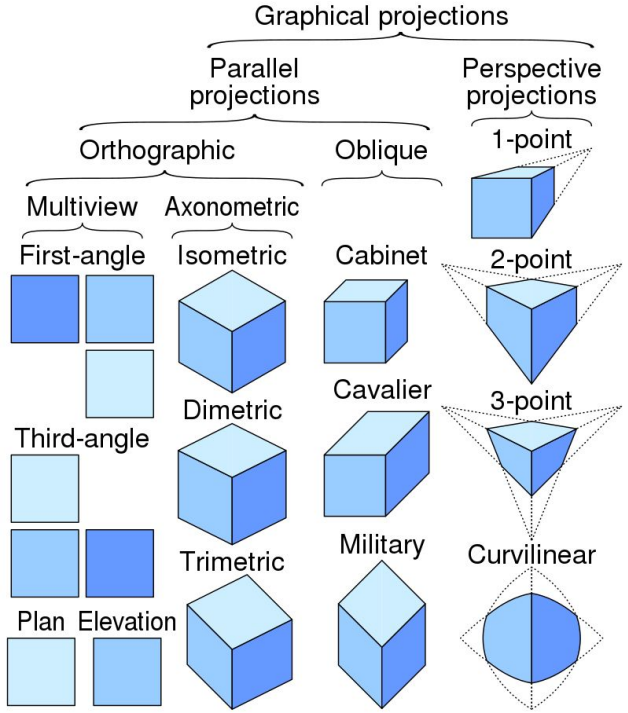
$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

General orthogonal projection in 4D is: $\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{ST}$

Oblique projection

OpenGL/WebGL projection functions cannot produce general parallel projections

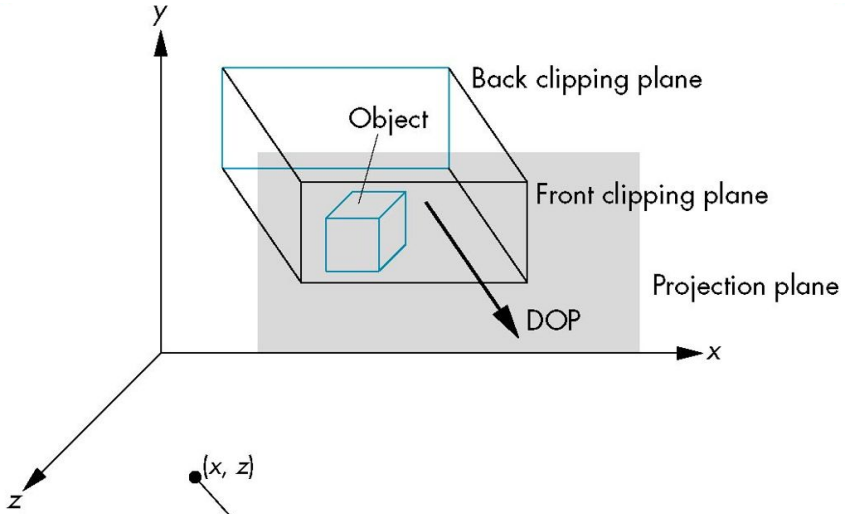


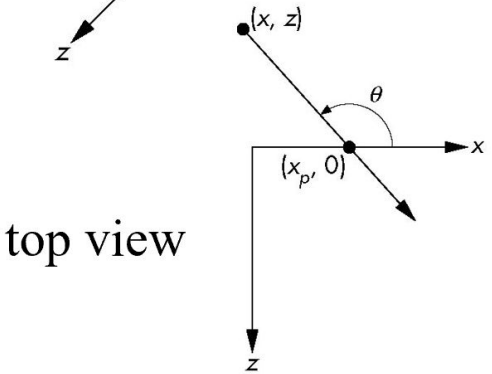


But, it looks like the cube was sheared!

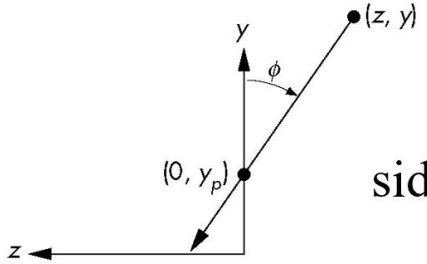
- **Oblique = Shear + Orthogonal**

Shear





top view



side view

Shear matrix

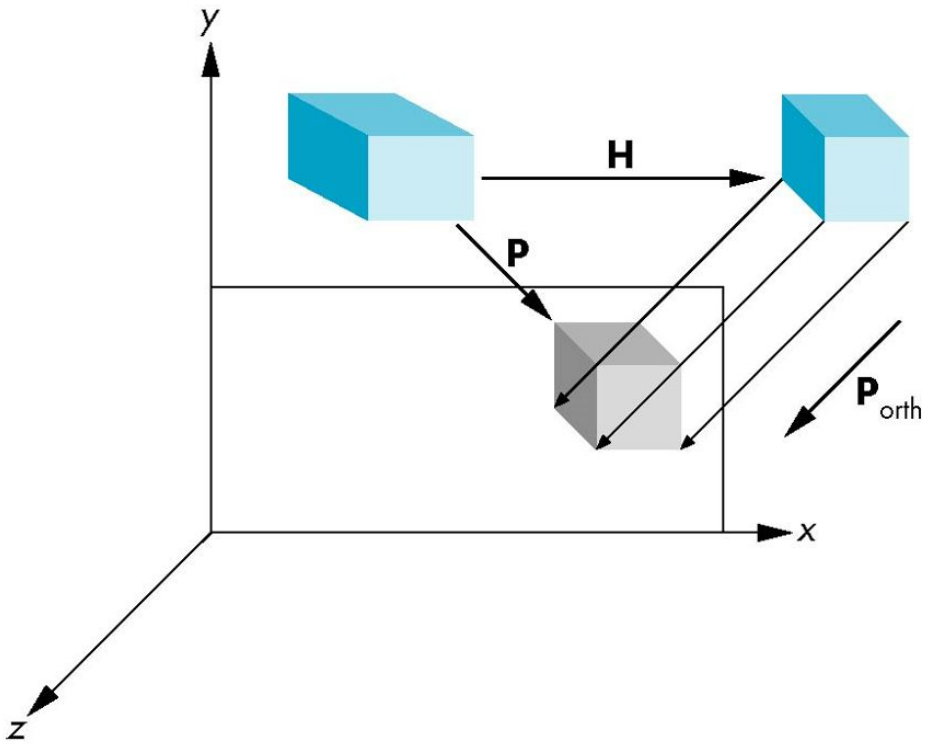
xy shear (*z* unchanged)

$$\mathbf{H}(\theta,\phi) = \begin{bmatrix} 1 & 0 & -\cot\theta & 0 \\ 0 & 1 & -\cot\varphi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Projection matrix: $\mathbf{P} = \mathbf{M}_{orth}\mathbf{H}(\Theta, \phi)$

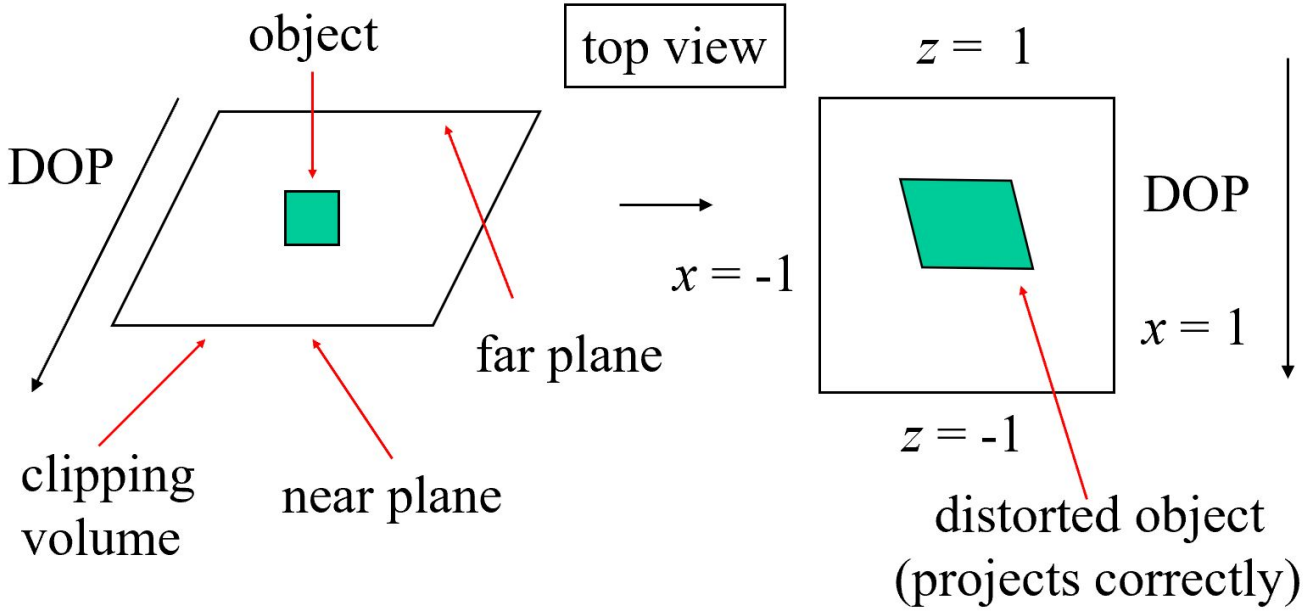
General case: $\mathbf{P} = \mathbf{M}_{orth}\mathbf{STH}(\Theta, \phi)$

Equivalency



Effect on clipping

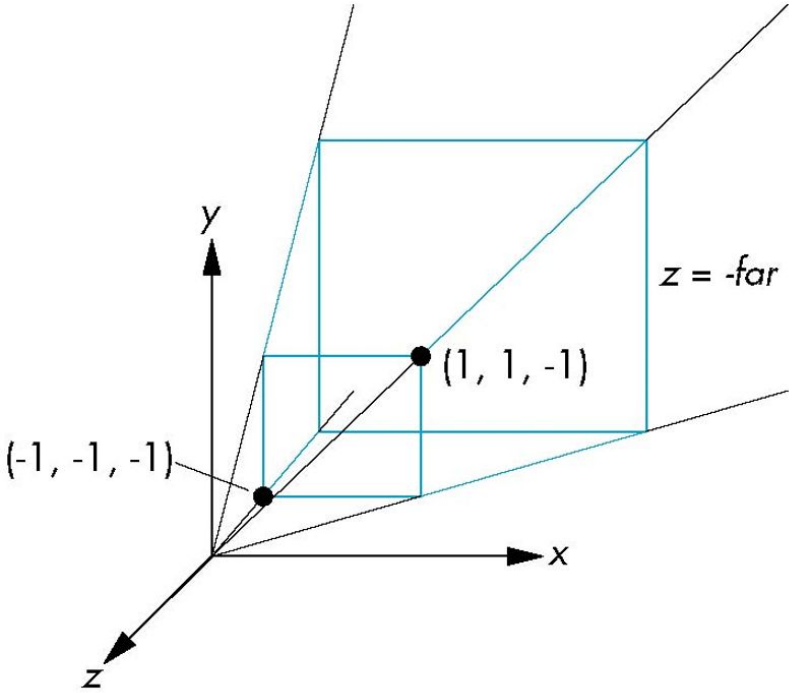
Projection matrix **P** = **STH** transforms original clipping volume to default clipping volume



Let's generalize -- simple perspective

Consider simple perspective with:

- CoP at origin
- Near clipping plane at *z = -1*
- 90deg FoV determined by planes:
  - *x = +/- z*
  - *y = +/- z*



Perspective matrices

Simple projection matrix in homogeneous coordinates:

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

Matrix is independent of **far clipping plane!**

Generalization

After perspective division, point (x, y, z, 1) goes to:

$x'' = x/z$
$y'' = y/z$
$z'' = -(\alpha+\beta/z)$

Projects orthogonally to desired point **regardless** of α or β

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

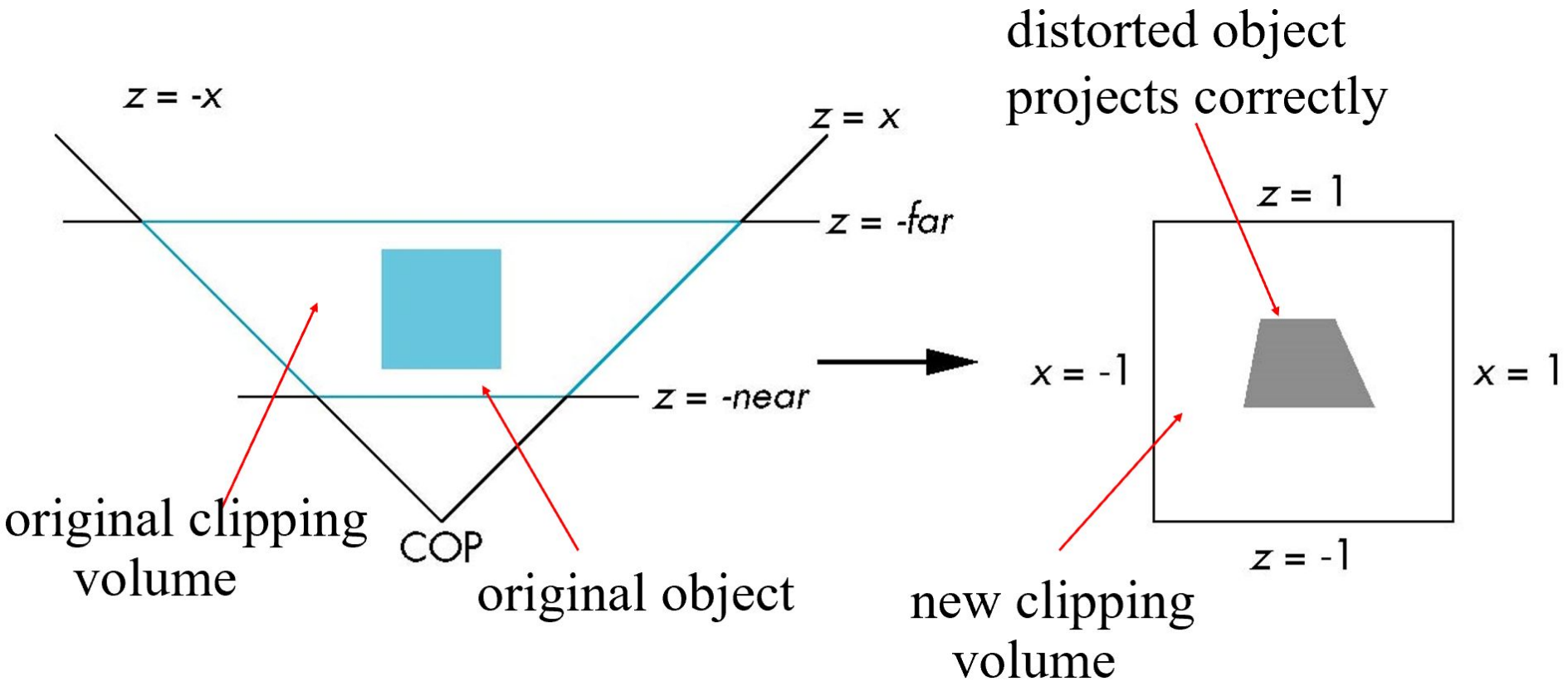
Selecting alpha and beta

Near plane: mapped to z = -1
Far plane: mapped to z = 1
Sides mapped to x=±1, y=±1

$$\alpha = \frac{near + far}{far - near}$$

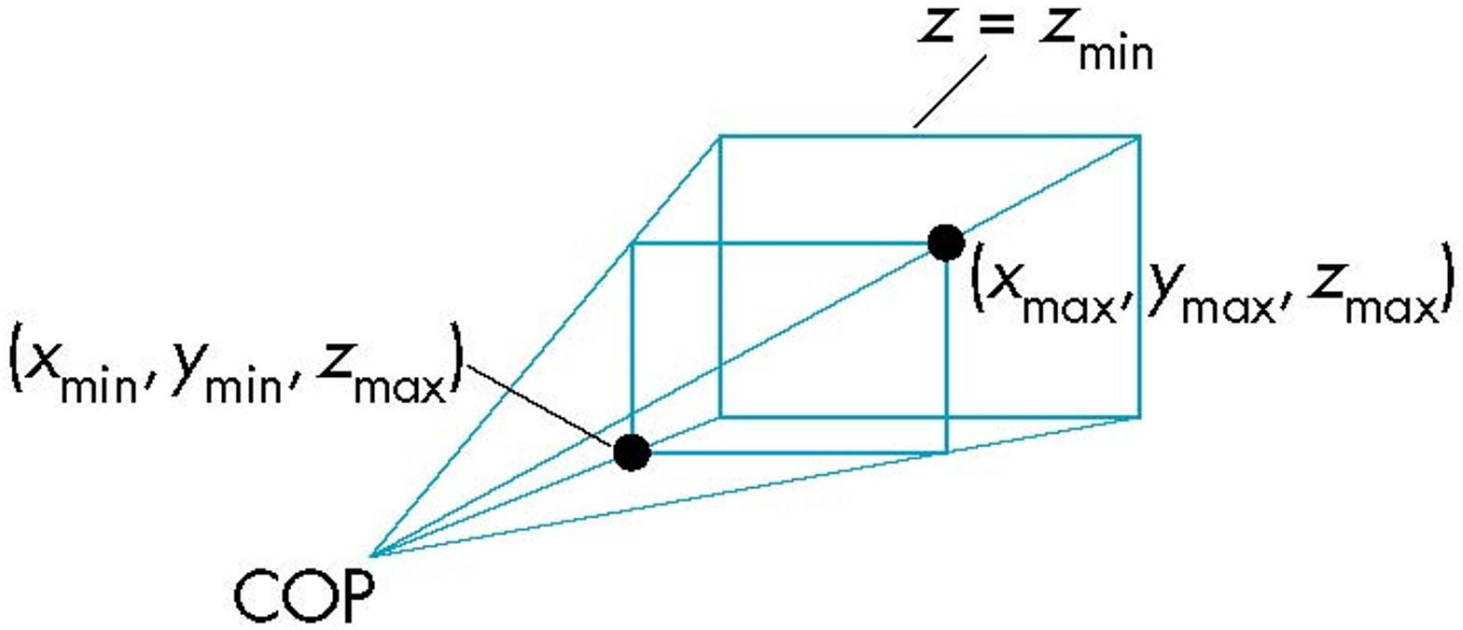
$$\beta = \frac{2near * far}{near - far}$$

New clipping volume is default clipping volume!


distorted object
projects correctly
z = -x
z = x
z = -far
z = -near
COP
original clipping
volume
original object
z = 1
x = -1
x = 1
z = -1
new clipping
volume

Now back to WebGL

`gl.frustum` → unsymmetric viewing frustum (`gl.perspective` doesn't allow this)



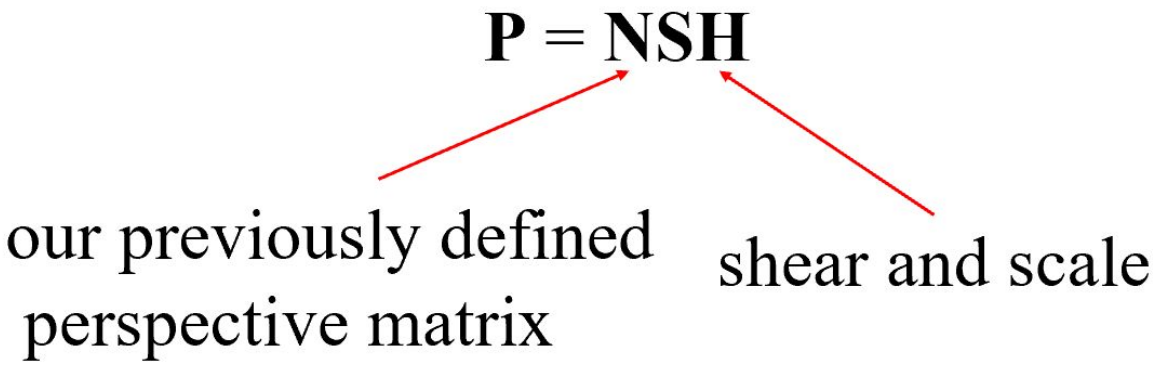
Perspective matrix

frustum normalization requires initial shear

- Form right viewing pyramid

Followed by scaling to get normalized perspective volume

Followed by orthogonal transformation



...why?

Normalization enables single pipeline approach

- Both perspective and orthogonal viewing

Stay in four-dimensional homogeneous coordinates as long as possible

- Retain depth information
- Needed for hidden surface removal and shading

Simplifies clipping!

frustum

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

perspective

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


Ending here, additional practice after:

Add some *perspective* to your rotating cube

Let's make it an **orthographic** view!

Remember:

```
Merge the code from ortho into your rotating cube (any are fine)

Use the ortho function

Update their theta to be something else (e.g., theta2)
  (If you look at ortho.html and update theta, you should see
   similar behavior)
```