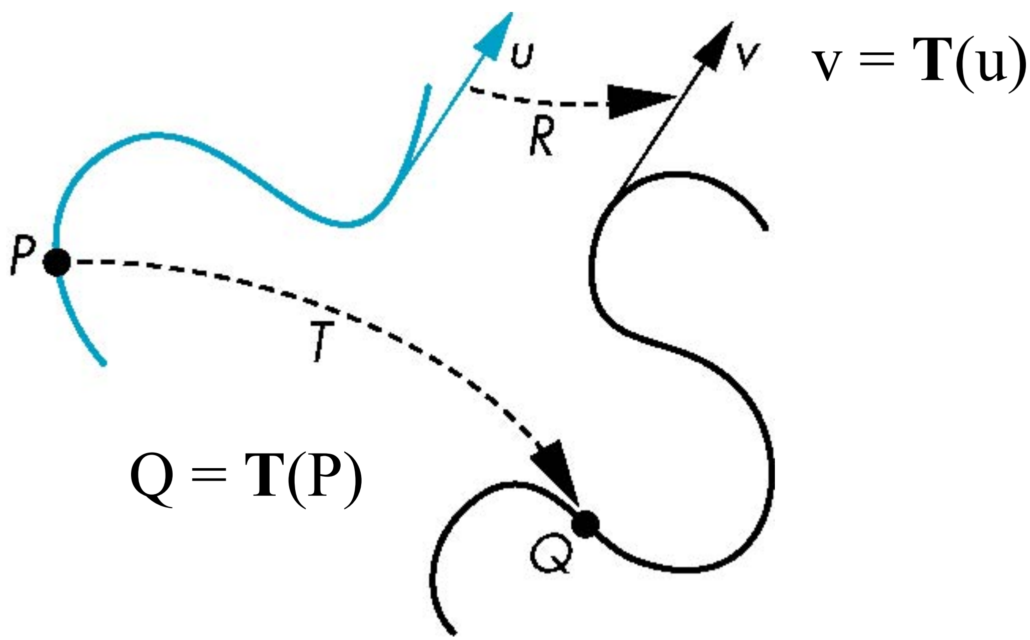
Learning Objectives

- Students completing this lecture will be able to
  - Write the transformation matrix for translation, rotation (along x, y, z axis), and scaling
  - Explain the following terms: affine transformation, object instancing, current transformation matrix
  - Describe how a geometric model can be stored efficiently
  - Write WebGL code to perform desired transformations

Transformations

General Transformations

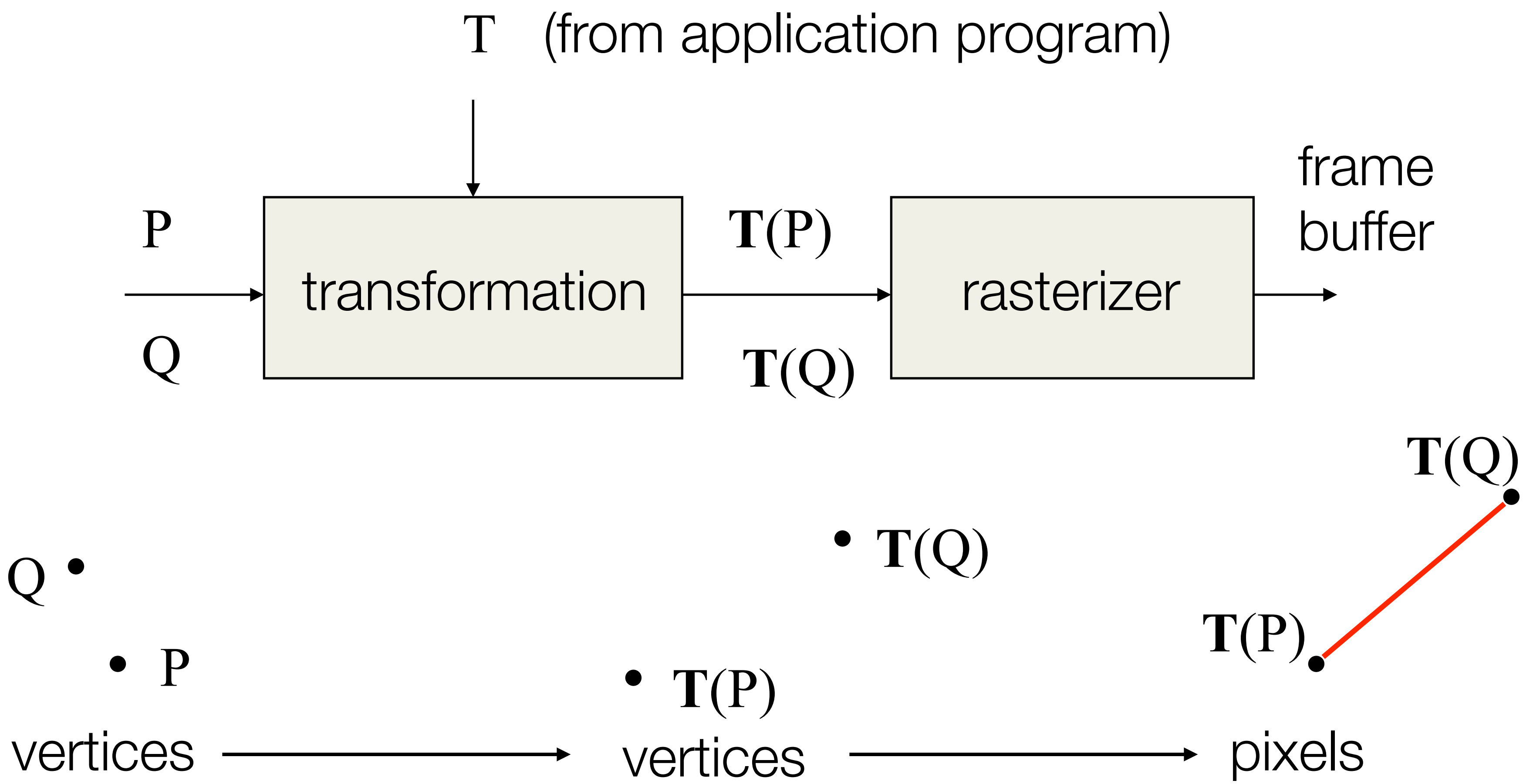
- A transformation maps points to other points and/or vectors to other vectors



Affine Transformations

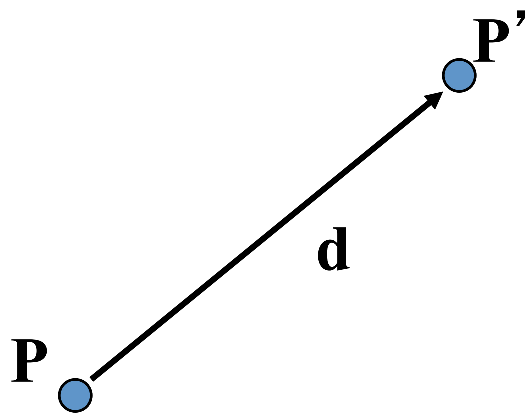
- Line preserving
- Characteristic of many physically important transformations
  - Rigid body transformations: rotation, translation
  - Scaling, shear
- The good thing is that we only need to transform endpoints of line segments and let implementation draw line segment between the transformed endpoints!

Pipeline Implementation



Translation

- Move (translate, displace) a point to a new location



- Displacement determined by a vector **d**
  - Three degrees of freedom
  - $\mathbf{P'} = \mathbf{P} + \mathbf{d}$

Translation Using Representations

- Using the homogeneous coordinate representation in some frame

$$\mathbf{p} = [\, x \; y \; z \; 1]^T$$

$$\mathbf{p}' = [x' \; y' \; z' \;\; 1]^T$$

$$\mathbf{d} = [\mathrm{d}_x \; \mathrm{d}_y \; \mathrm{d}_z \; 0]^T$$

$$[\, x \; y \; z \; w]^T$$

$w = 1$ point; $w = 0$ vector

Hence $\mathbf{p}' = \mathbf{p} + \mathbf{d}$ or

$$x' = x + \mathrm{d}_x$$

$$y' = y + \mathrm{d}_y$$

$$z' = z + \mathrm{d}_z$$

note that this expression is in four dimensions and expresses point = vector + point

Translation Matrix

- We can also express translation using a 4×4 matrix $\mathbf{T}$ in homogeneous coordinates $\mathbf{p}' = \mathbf{T}\mathbf{p}$ where

$$\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

note that the translation matrix is multiplied to the left of point

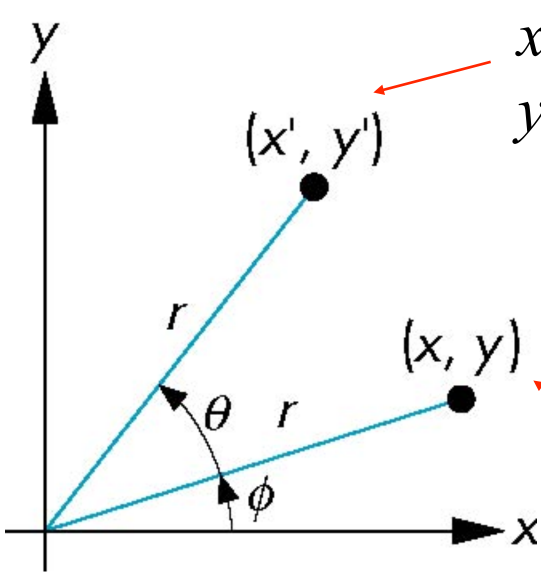
- This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated together

Rotation (2D)

- Consider rotation about the origin by θ degrees
  - Radius stays the same, angle increases by θ

Trigonometric Identities

$$x' = r\cos(\phi + \theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$y' = r\sin(\phi + \theta) = r\sin\phi\cos\theta + r\cos\phi\sin\theta$$



$$x = r\cos\phi$$
$$y = r\sin\phi$$

$$\boxed{\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned}}$$

Rotation about the z axis

- Rotation about z axis in three dimensions leaves all points with the same z
  - Equivalent to rotation in two dimensions in planes of constant z

$$x' = x \cos \theta - y \sin \theta$$
$$y' = x \sin \theta + y \cos \theta$$
$$z' = z$$

  - or in homogeneous coordinates

$$\mathbf{p}' = \mathbf{R}_z(\theta)\mathbf{p}$$

note that the rotation matrix is multiplied to the left of point

Rotation Matrix

$$\mathbf{R} = \mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boxed{\begin{aligned} x' &= x \cos\theta - y \sin\theta \\ y' &= x \sin\theta + y \cos\theta \end{aligned}}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotation about x and y axes

- Same argument as for rotation about z axis
  - For rotation about x axis, x is unchanged
  - For rotation about y axis, y is unchanged

$$\mathbf{R} = \mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

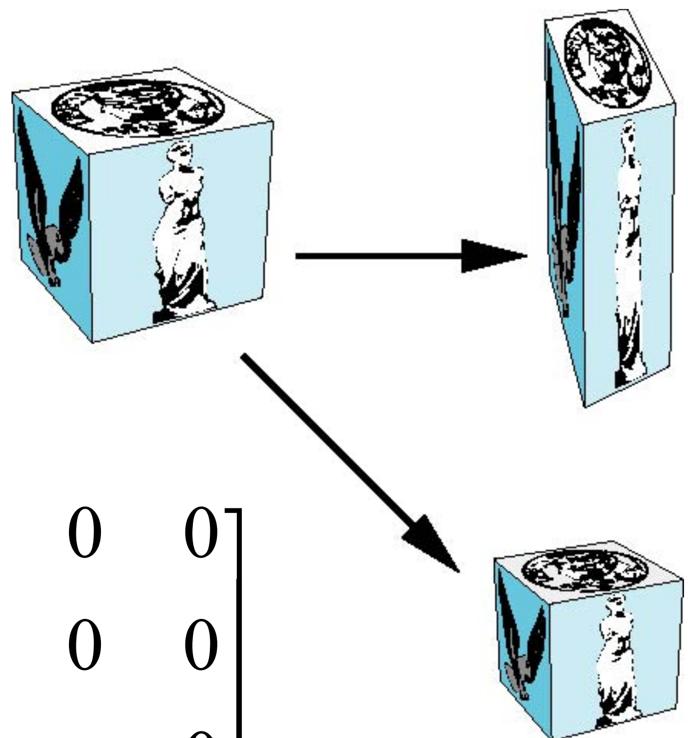
Scaling

Expand or contract along each axis (fixed point of origin)

$$x' = s_x x$$
$$y' = s_y x$$
$$z' = s_z x$$

$$\mathbf{p}' = \mathbf{S}\mathbf{p}$$

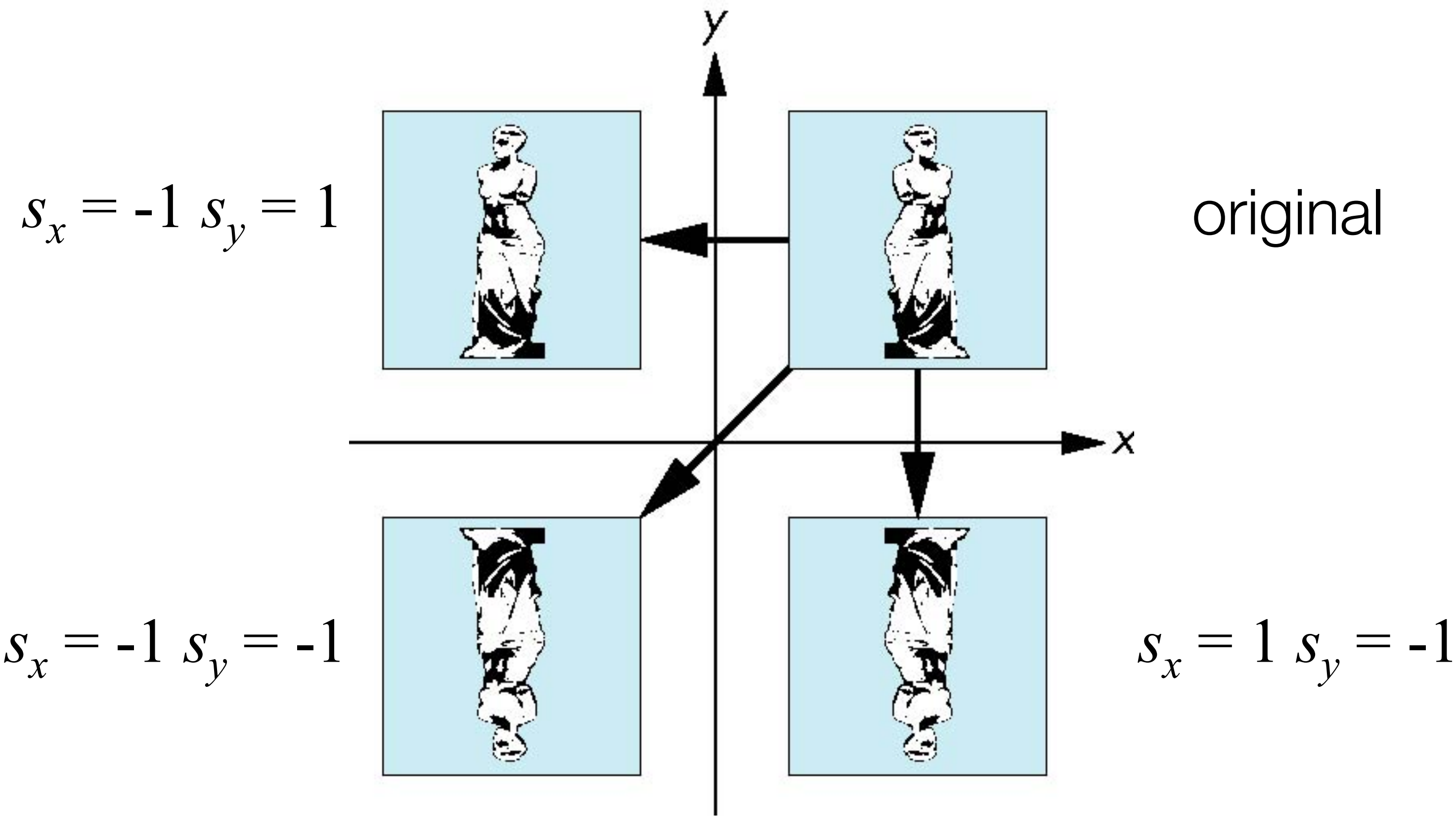
note that the scaling matrix is multiplied to the left of point



$$\mathbf{S} = \mathbf{S}(\mathrm{s_x}, \mathrm{s_y}, \mathrm{s_z}) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Reflection

Corresponds to negative scale factors



Inverses

- Although we could compute inverse matrices by general formulas, we can use simple geometric observations
  - Translation: $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathbf{T}(-d_x, -d_y, -d_z)$
  - Rotation: $\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta)$
    - Holds for any rotation matrix
    - Note that since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$

    $\mathbf{R}^{-1}(\theta) = \mathbf{R}^{T}(\theta)$
  - Scaling: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{R}^{-1}(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{R}^{T}(\theta)$$

Concatenation

- We can form arbitrary affine transformation matrices by multiplying together rotation, translation, and scaling matrices
- Because the same transformation is applied to many vertices, the cost of forming a matrix $\mathbf{M} = \mathbf{ABCD}$ is not significant compared to the cost of computing $\mathbf{Mp}$ for many vertices $\mathbf{p}$
- The difficult part is how to form a desired transformation from the specifications in the application

Order of Transformations

- Note that matrix on the right is the first applied
- Mathematically, the following are equivalent

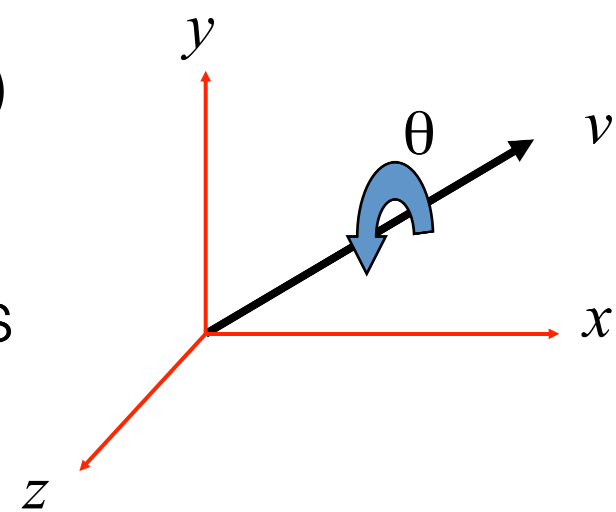
$$\mathbf{p}' = \mathbf{ABCp} = \mathbf{A(B(Cp))}$$

General Rotation about the Origin

- A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the x, y, and z axes

$$\mathbf{R}(\theta) = \mathbf{R}_z(\theta_z)\ \mathbf{R}_y(\theta_y)\ \mathbf{R}_x(\theta_x)$$

$\theta_x\ \theta_y\ \theta_z$ are called the Euler angles

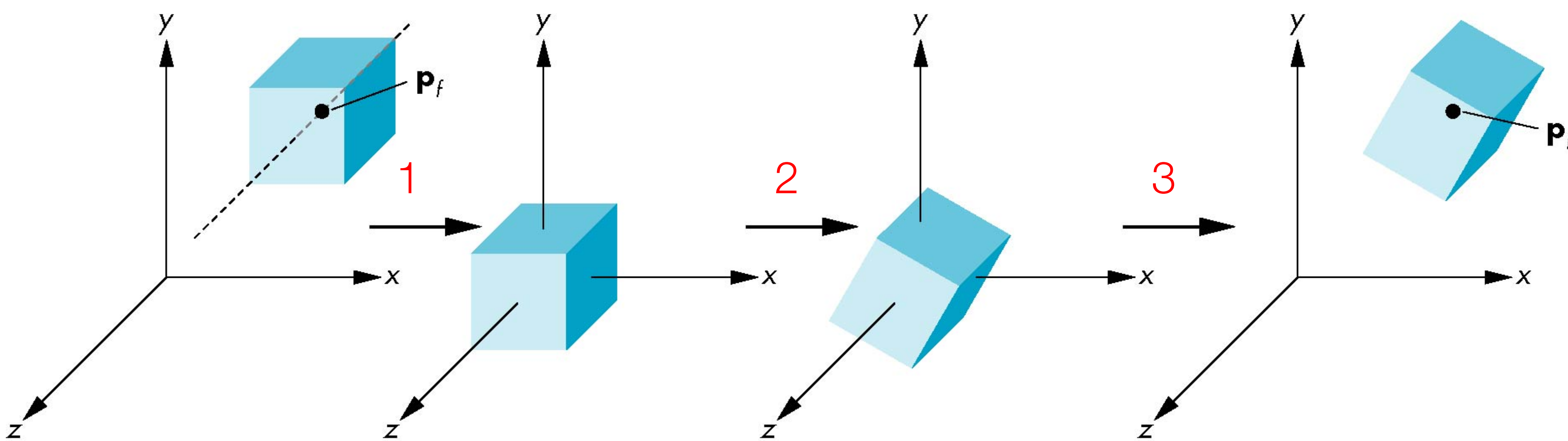


- Note that rotations do not commute
- We can use rotations in another order but with different angles

Rotation about a Fixed Point other than the Origin

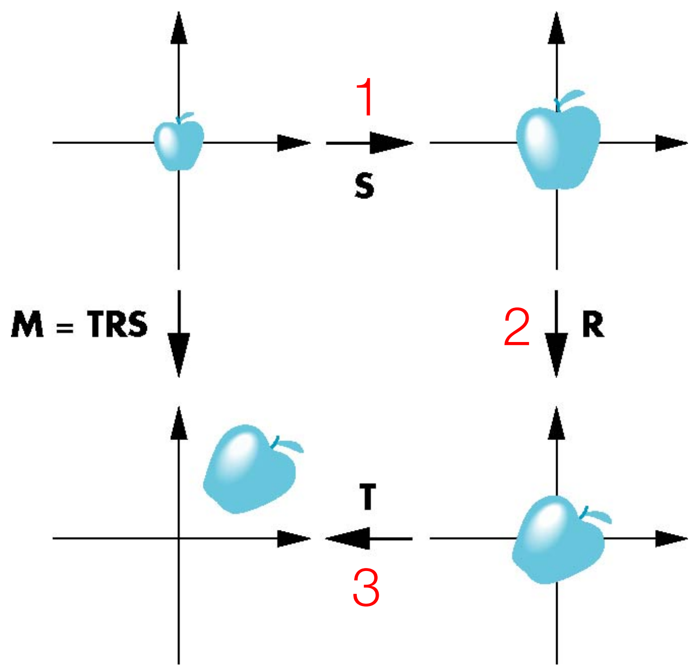
1. Move fixed point to origin
2. Rotate
3. Move fixed point back

$$\mathbf{M} = \overset{3}{\mathbf{T}(p_f)}\ \overset{2}{\mathbf{R}(\theta)}\ \overset{1}{\mathbf{T}(-p_f)}$$



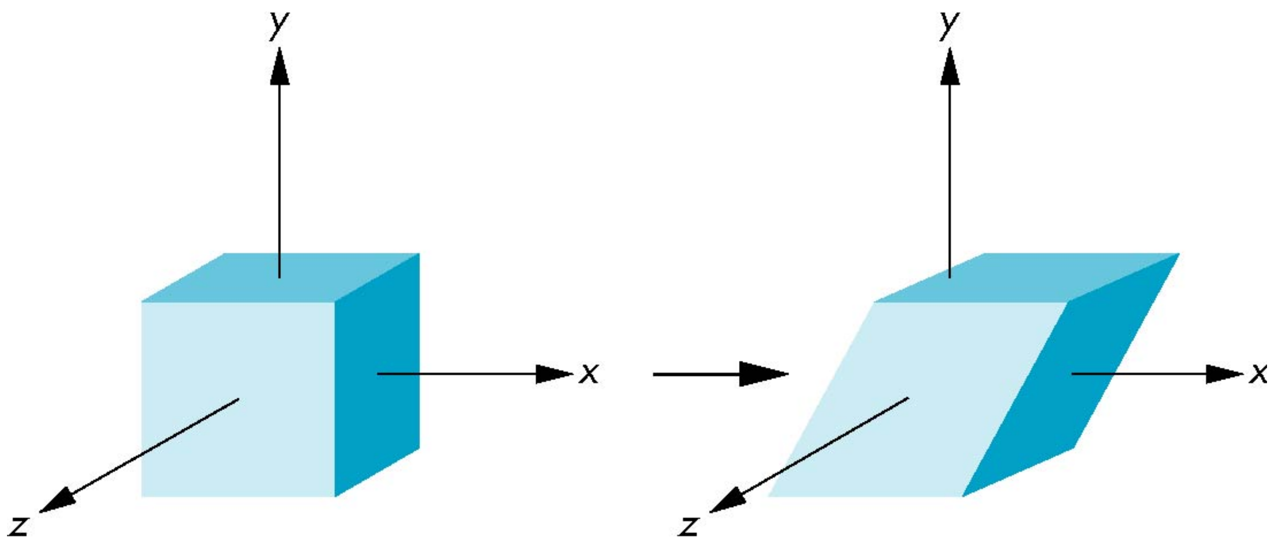
Object Instancing

- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size
- We apply an instance transformation to its vertices to
  - 1. Scale
  - 2. Orient (rotate)
  - 3. Locate (translate)



Shear

- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions

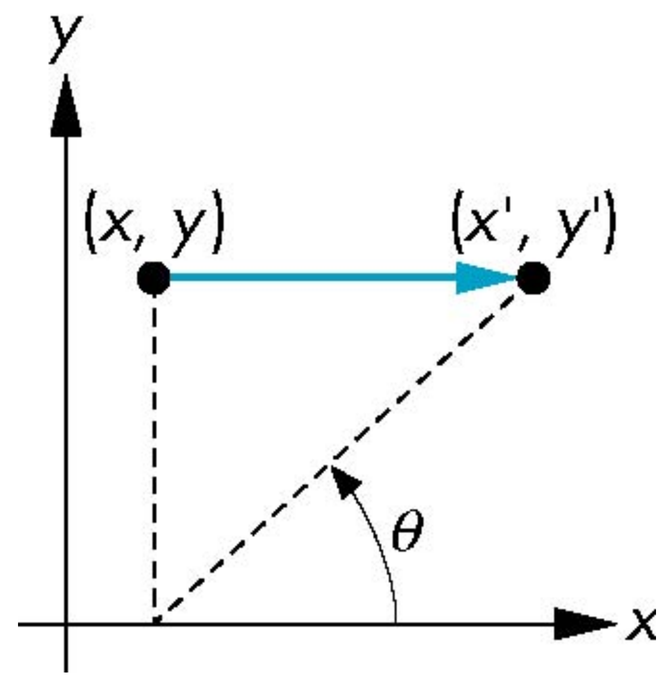


Shear Matrix

- Consider simple shear along x axis

$$x' = x + y \cot \theta$$
$$y' = y$$
$$z' = z$$

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



WebGL Transformations

OpenGL Matrices (Pre 3.1, now deprecated)

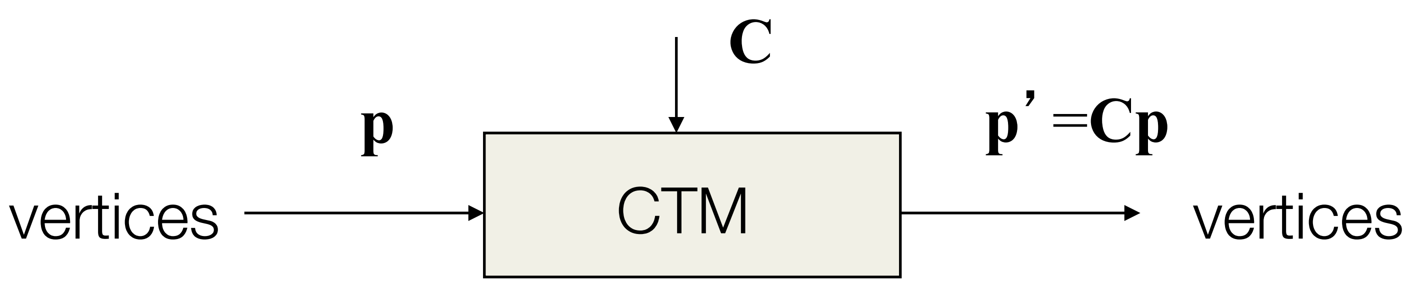
- In OpenGL matrices are part of the state
- Multiple types
  - Model-View `(GL_MODELVIEW)`
  - Projection `(GL_PROJECTION)`
  - Texture `(GL_TEXTURE)`
  - Color `(GL_COLOR)`
- Single set of functions for manipulation
- Select which to manipulated by
  - `glMatrixMode(GL_MODELVIEW);`
  - `glMatrixMode(GL_PROJECTION);`

Why Deprecation

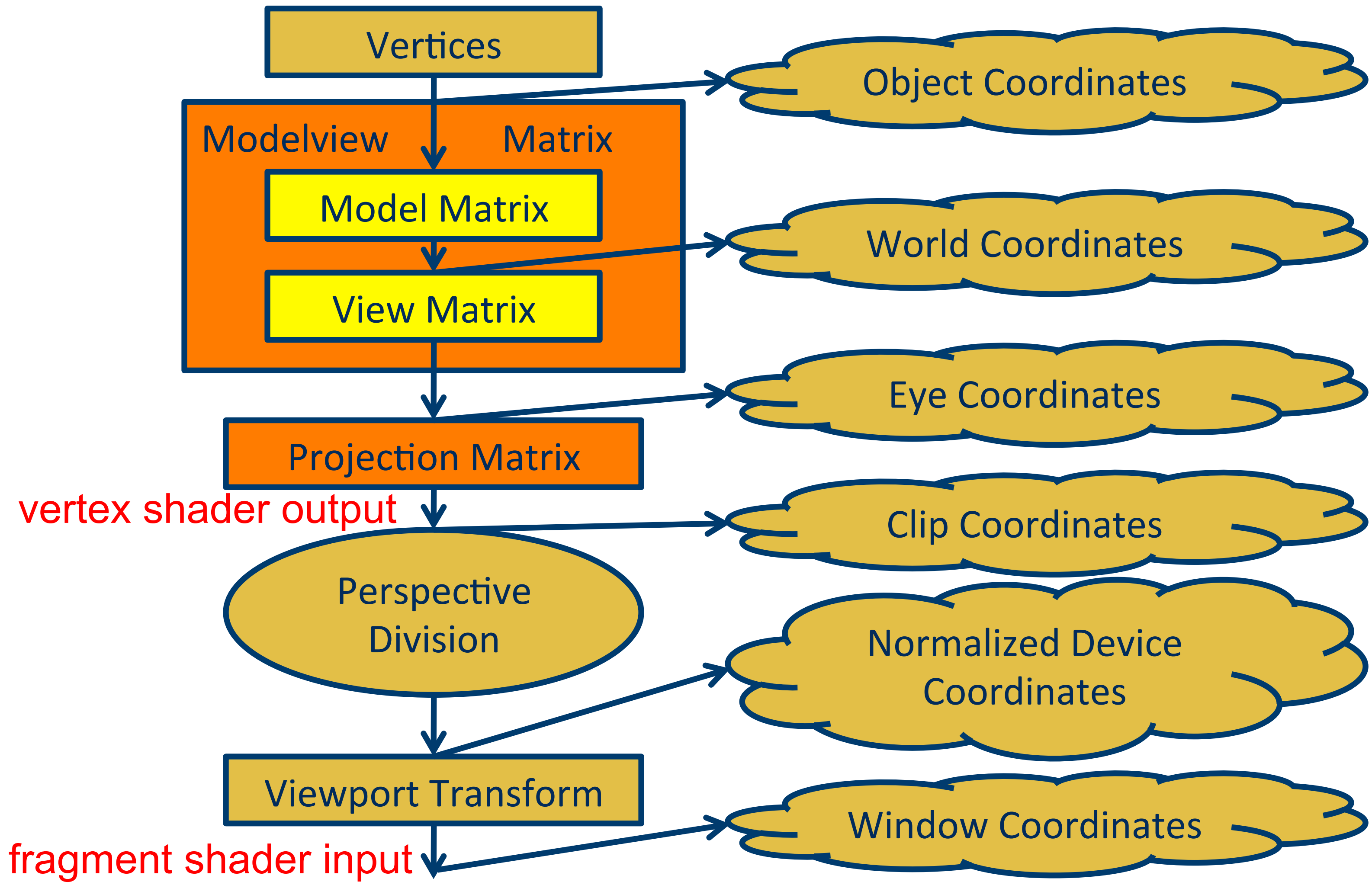
- Functions were based on carrying out the operations on the CPU as part of the fixed function pipeline
- Current model-view and projection matrices were automatically applied to all vertices using CPU
- We will use the notion of a current transformation matrix with the understanding that it may be applied in the shaders

Current Transformation Matrix (CTM)

- Conceptually there is a 4 x 4 homogeneous coordinate matrix, the current transformation matrix (CTM) that is part of the state and is applied to all vertices that pass down the pipeline
- The CTM is defined in the user program and loaded into a transformation unit

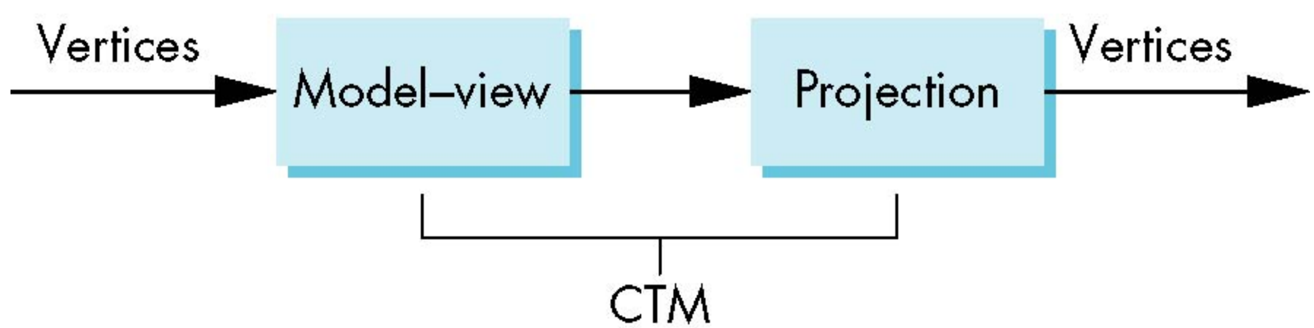


Vertex Transformation Pipeline



CTM in WebGL

- OpenGL has a model-view and a projection matrix in the pipeline which are concatenated together to form the CTM
- We will emulate this process



Using the ModelView Matrix

- In WebGL, the model-view matrix is used to
  - Position the camera
    - Can be done by rotations and translations but is often easier to use the `lookAt` function in `MV.js`
  - Build models of objects
- The projection matrix is used to define the view volume and to select a camera lens
- Although these matrices are no longer part of the OpenGL state, it is usually a good strategy to create them in our own applications

$$\mathbf{q} = \underbrace{\mathbf{P}}_{\text{projection}}\underbrace{\mathbf{MV}}_{\text{modelview}}\mathbf{p}$$

Transformation Functions

Defined in `MV.js` file:

```
translate(x, y, z); // return the translation matrix
rotate(theta, axis); // return the rotation matrix
rotateX(theta); // rotate along the x axis
rotateY(theta); // rotate along the y axis
rotateZ(theta); // rotate along the z axis
scalem(x, y, z); // return the scaling matrix, note
                    that scale(s, u) is for vector
                    scaling

w = mult(u, v); // w = uv, note the order of
                   multiplication

transpose(m); // return the transposed matrix
inverse(m); // return the inversed matrix
```

Rotation, Translation, Scaling

Create an identity matrix:

`var m = mat4(1.0);` or `var m = mat4();`

Multiply on left by rotation matrix of `theta` in degrees where `(vx, vy, vz)` define axis of rotation

```
var r = rotate(theta, vec3(vx, vy, vz));
m = mult(r, m);
```

Also have `rotateX`, `rotateY`, `rotateZ`

Do same with translation and scaling:

```
var s = scalem(sx, sy, sz);
m = mult(s, m);
var t = translate(dx, dy, dz);
m = mult(t, m);
```

Example

- Rotation about `z` axis by 30 degrees with a fixed point of `(1.0, 2.0, 3.0)`
- The correct order: $\texttt{Trans}_{+}$ $\texttt{Rot}_{z}$ $\texttt{Trans}_{-}$

```
var m = mult(translate(1.0, 2.0, 3.0),
             rotateZ(30.0));
m = mult(m, translate(-1.0, -2.0, -3.0));
```

or

```
var m = mult(rotateZ(30.0),
             translate(-1.0, -2.0, -3.0));
m = mult(translate(1.0, 2.0, 3.0), m);
```

Arbitrary Matrices

- Can load and multiply by matrices defined in the application program
- Matrices are stored as one dimensional array of 16 elements by `MV.js` but can be treated as 4 x 4 matrices in row major order
- OpenGL wants column major data
- `gl.unifromMatrix4fv` has a parameter for automatic transpose (it must be set to false)
- `flatten` function converts to column major order which is required by WebGL functions

```
gl.uniformMatrix4fv(ctMatrixLoc, false, flatten(ctMatrix));
```

Matrix Stacks

- In many situations we want to save transformation matrices for use later
  - Traversing hierarchical data structures
- Pre 3.1 OpenGL maintained stacks for each type of matrix
- Easy to create the same functionality in JS
  - Push and pop are part of Array object

```
var stack = [ ];
stack.push(matrix); // push in a 4×4 matrix
…
matrix = stack.pop(); // pop out the top matrix
                         from the stack
```

Applying Transformations

Using Transformations

- Example: Begin with a cube rotating
- Use mouse or button listener to change direction of rotation
- Start with a program that draws a cube in a standard way
  - Centered at origin
  - Sides aligned with axes
  - Will discuss modeling next

Where do we apply transformation?

- Same issue as with rotating square
  - 1. In application to vertices?
  - 2. In vertex shader: send MV matrix?
  - 3. In vertex shader: send angles?
- We show the second way

See example: cube-arrays.html

Interfaces

- One of the major problems in interactive computer graphics is how to use a two-dimensional device such as a mouse to interface with three dimensional objects
- Example: how to form an instance matrix?
- Some alternatives
  - Virtual trackball
  - 3D input devices such as the spaceball
  - Use areas of the screen (distance from center controls angle, position, scale depending on mouse button depressed)

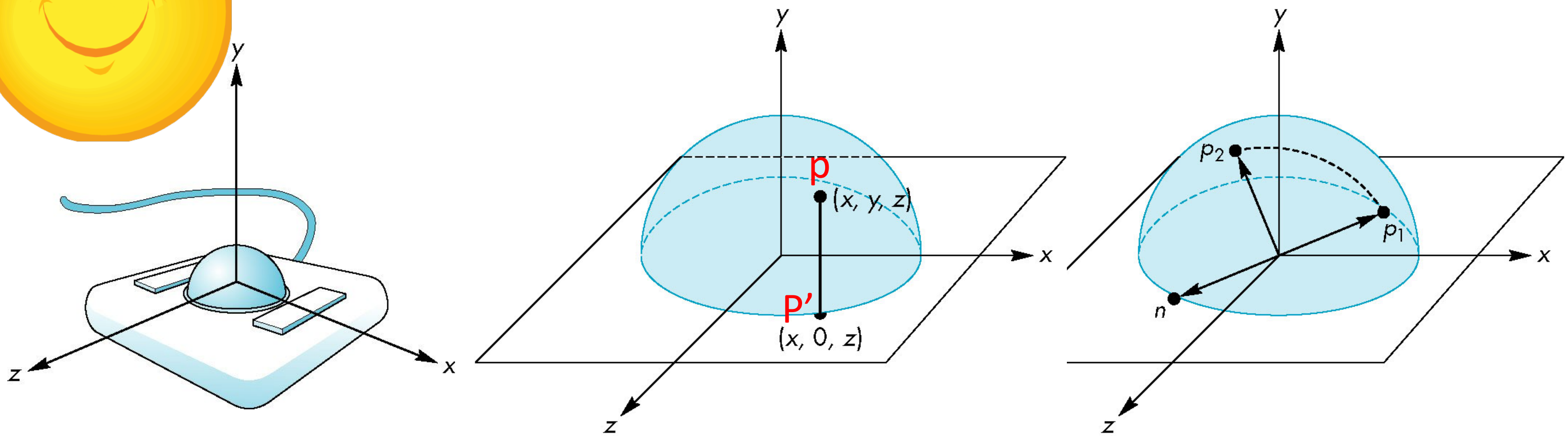
Handle rotation, translation and scaling

- We can use left, middle and right mouse button for rotation, translation and scaling
- Keep track of `x` and `y` coordinates when the mouse button is pressed down and when released
- Use the displacement along the `x` and `y` directions to control the amount of rotation along the `y` and `x` axes, translation along the `x` and `y` direction, and scaling (scale up if `dtx>0` and `dty>0`, scale down if `dtx<0` and `dty<0`)

Smooth Rotation

- From a practical standpoint, we often want to use transformations to move and reorient an object smoothly
  - Problem: find a sequence of model-view matrices $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_n$ so that when they are applied successively to one or more objects we see a smooth transition
- For orientating an object, we can use the fact that every rotation corresponds to part of a great circle on a sphere
  - Find the axis of rotation and angle
  - Virtual trackball

Virtual Trackball

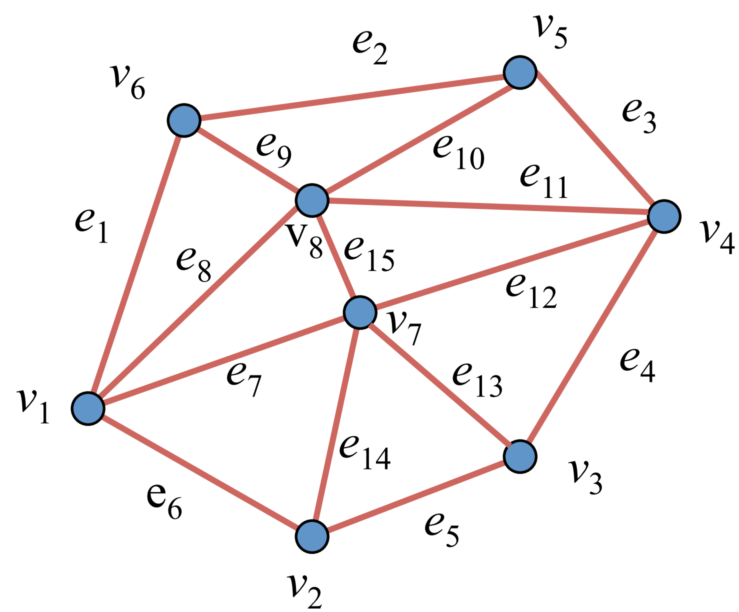


- Assume mouse is moving on the surface of a hemisphere
- Use $x^2 + y^2 + z^2 = 1$ to map the surface of the sphere to the x-z plane (project $\mathbf{p}_1$ to $\mathbf{p}_1'$ on the x-z plane)
- Map the x-z plane to the surface of the trackball (unproject $\mathbf{p}_2'$ back to $\mathbf{p}_2$ on the hemisphere)
- Calculate the normal $\mathbf{n} = \mathbf{p}_1 \times \mathbf{p}_2$ ($\mathbf{p}_1$ and $\mathbf{p}_2$ on the hemisphere)
- $\mathbf{n} = \sin\theta$ (a unit vector)
- $\sin\theta$ is approximately θ for small angles (fast mousing)

Building Models

Representing a Mesh

- Consider a mesh



- There are 8 nodes and 15 edges
  - 8 interior triangles
  - 9 interior (shared) edges
- Each vertex has a location $v_i = (x_i, y_i, z_i)$

Simple Representation

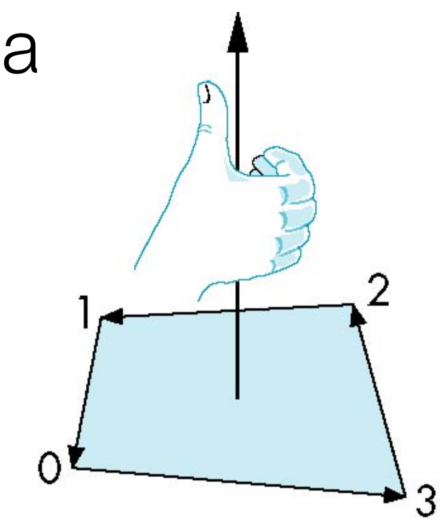
- Define each polygon by the geometric locations of its vertices
- Leads to code such as

```
vertex.push(vec3(x1, y1, z1));
vertex.push(vec3(x2, y2, z2));
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 $\{v_1, v_2, v_7\}$ and $\{v_2, v_7, v_1\}$ are equivalent in that the same polygon will be rendered but the order $\{v_1, v_7, v_2\}$ is different
- The first two describe outwardly facing polygons
- Use the right-hand rule = counter-clockwise encirclement of outward-pointing norma
- WebGL can treat inward and outward facing polygons differently

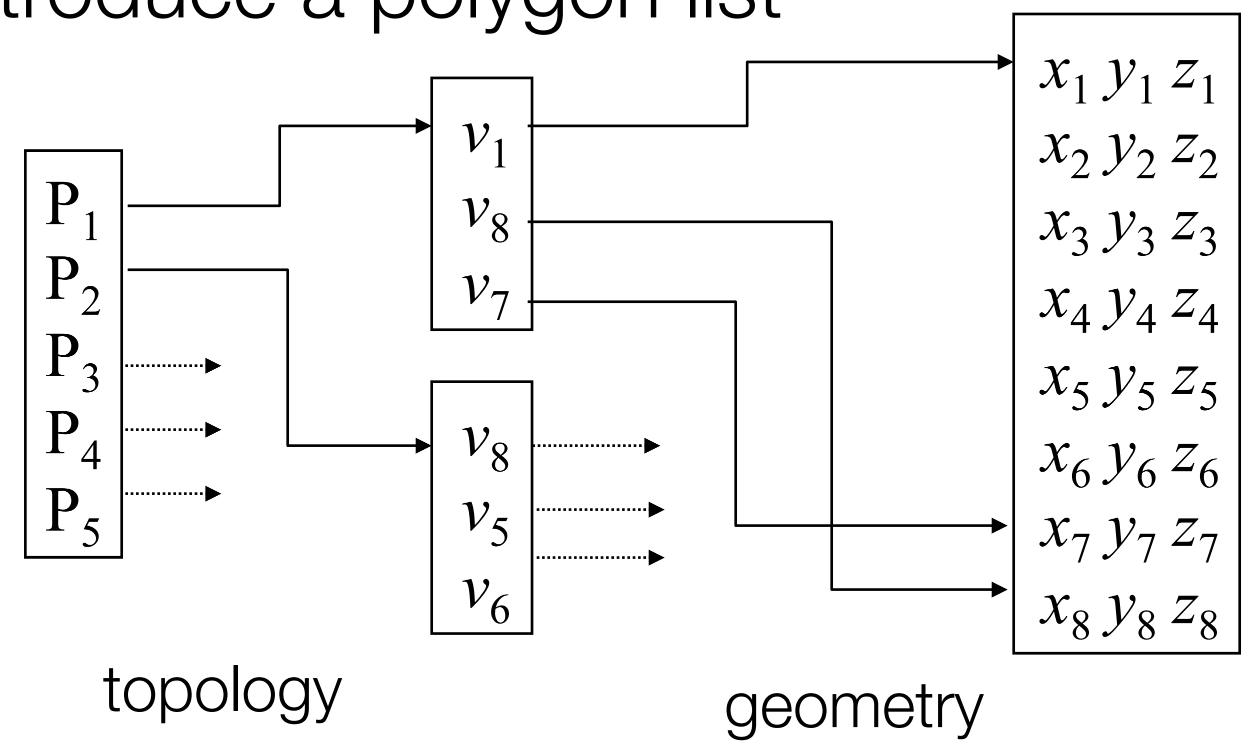


Geometry vs. Topology

- 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

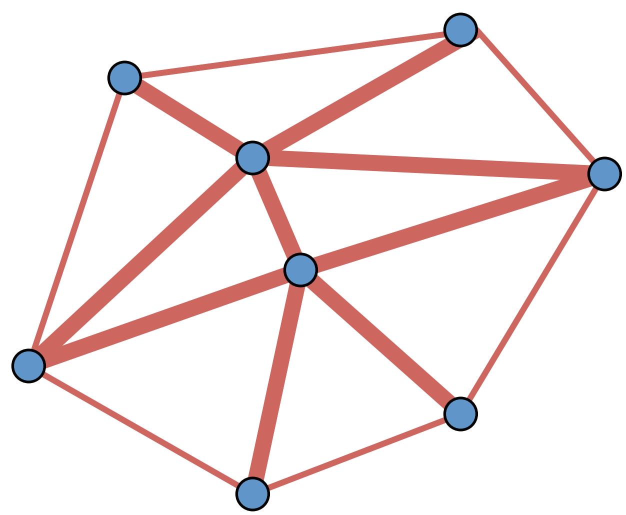
Vertex Lists

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



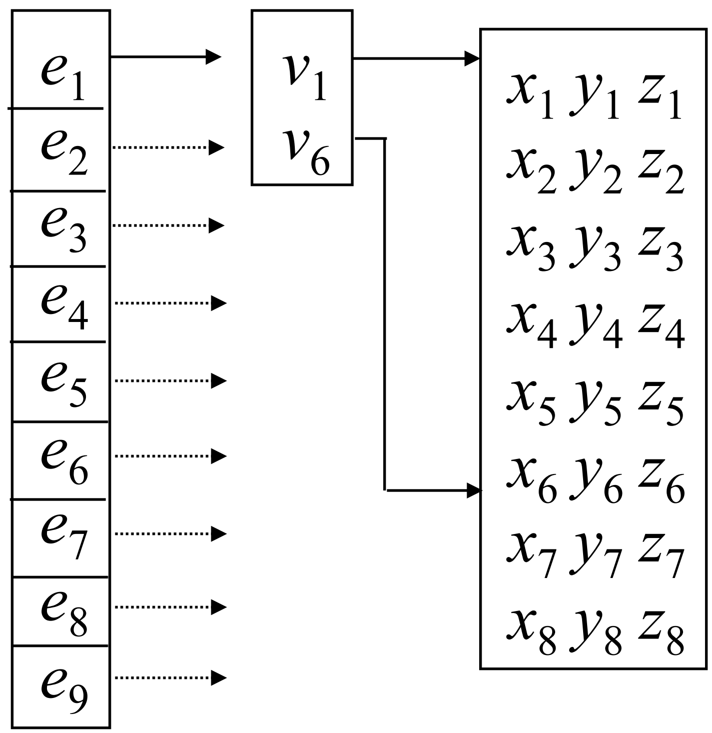
Shared Edges

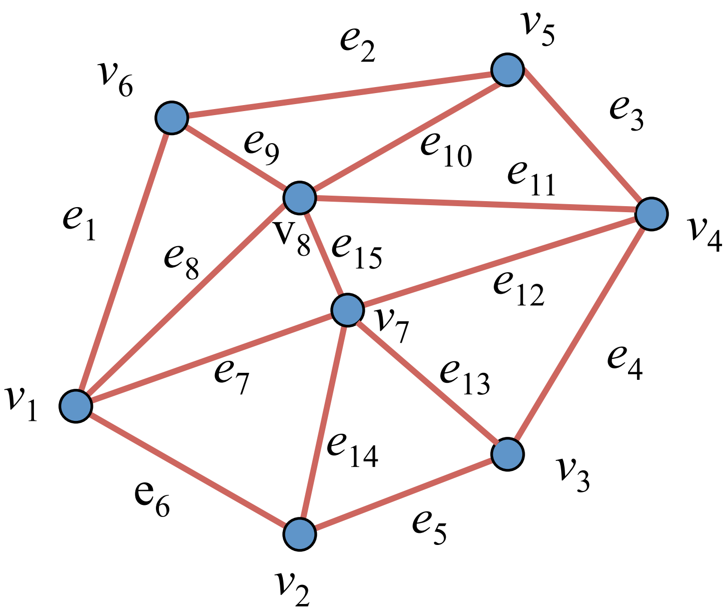
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- Can store mesh by edge list

Edge List





Note polygons are not represented

Modeling a 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 )
    ];
```

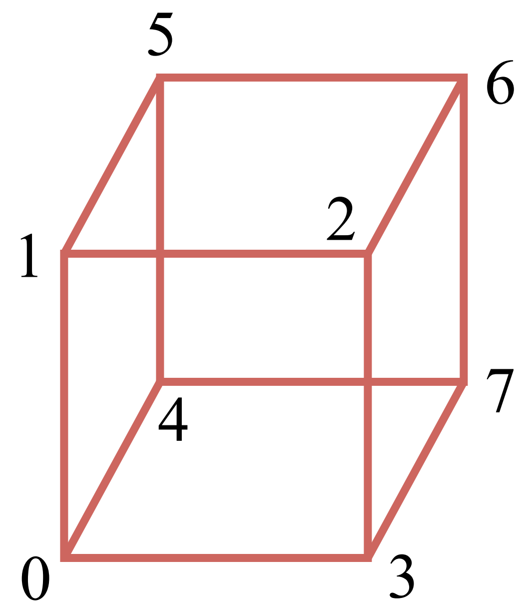
Colors

- 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; // why 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 (flat shading) 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 (why this is more efficient?)

- 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 );
```

See example: cube-elements.html

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