Week 05:
2D and 3D Transformations

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

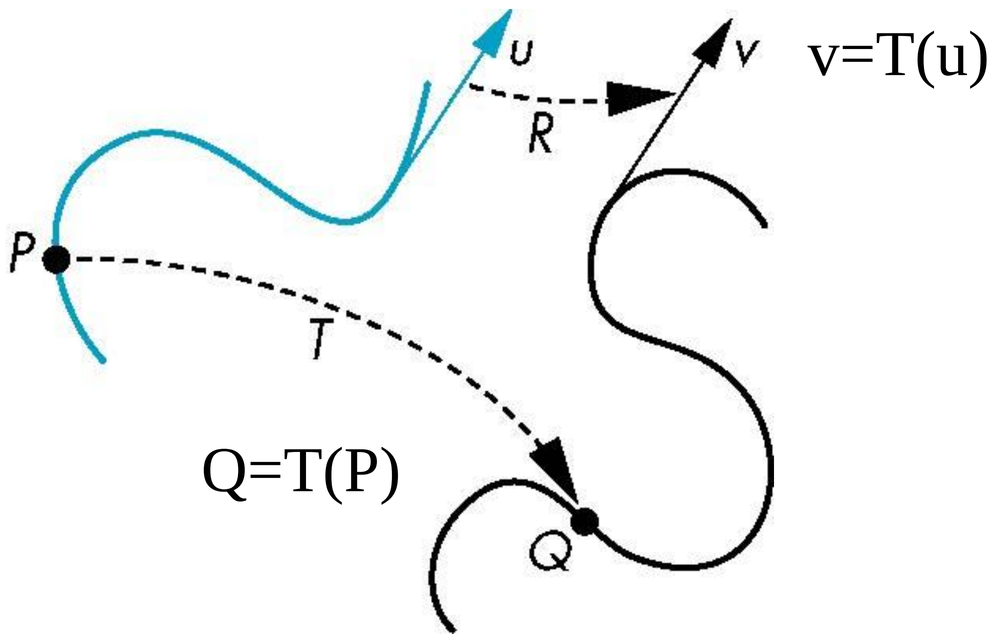
Some materials from the companion slides of Angel and Shreiner, "Interactive Computer Graphics, A Top-Down Approach with WebGL."

Objectives

- Introduce standard transformations
  - Rotation
  - Translation
  - Scaling
- Derive homogenous coordinate transformation matrices
- Learn to build arbitrary transformation matrices from simple transformation building blocks
- Learn how to carry out these same transformations in WebGL

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, shearing
- Important in graphics is that we need only transform endpoints of line segments and let implementation draw line segment between the transformed endpoints

Notation (for these slides)

- We will be working with both coordinate-free representations of transformations and representations within a particular frame

P, Q, R: points in an affine space

u, v, w: vectors in an affine space

α, β, γ: scalars

$\mathbf{p}, \mathbf{q}, \mathbf{r}$: representations of points

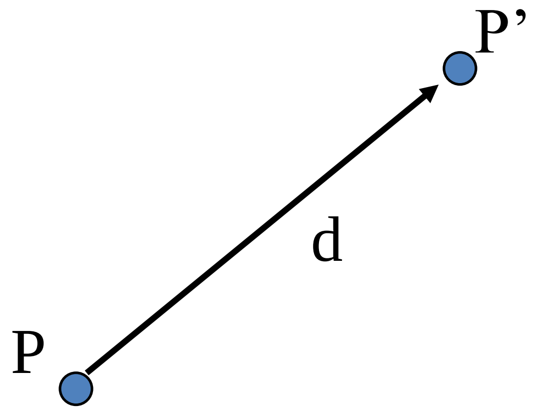
-array of 4 scalars in homogeneous coordinates

$\mathbf{u}, \mathbf{v}, \mathbf{w}$: representations of points

-array of 4 scalars in homogeneous coordinates

Translation

- Move (translate, displace) a point to a new location
- Displacement determined by a vector d
  - Three degrees of freedom (in 3D)
  - P'=P+d



Translation using Representations

- Using the homogeneous coordinate representation in some frame

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

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

$\mathbf{d} = [d_x\ d_y\ d_z\ 0]^T$

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

$x'=x+d_x$

$y'=y+d_y$

$z'=z+d_z$

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

Translation Matrix

- We can also express translation using a 4 x 4 matrix $\mathbf{T}$ in homogeneous coordinates

  $\mathbf{p}' = \mathbf{T}\mathbf{p}$ where:

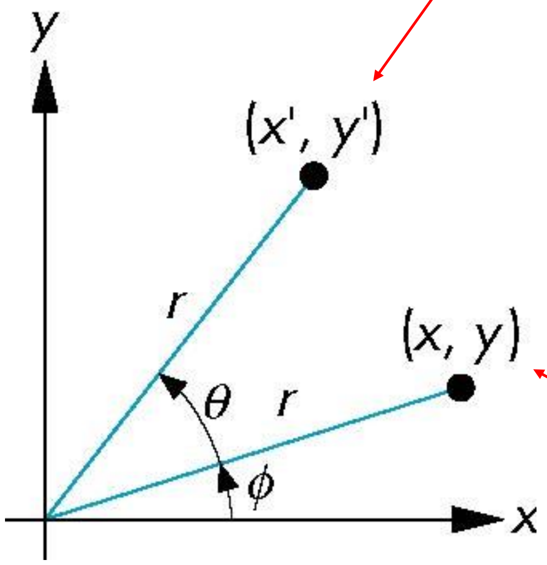
$$\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 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

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



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

(see: [Trigonometric Addition Formulas])

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

Rotation about the z axis (3D)

- 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

$x' = x \cos q - y \sin q$

$y' = x \sin q + y \cos q$

$z' = z$

  - or in homogeneous coordinates:

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

  - or as a matrix:

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

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)

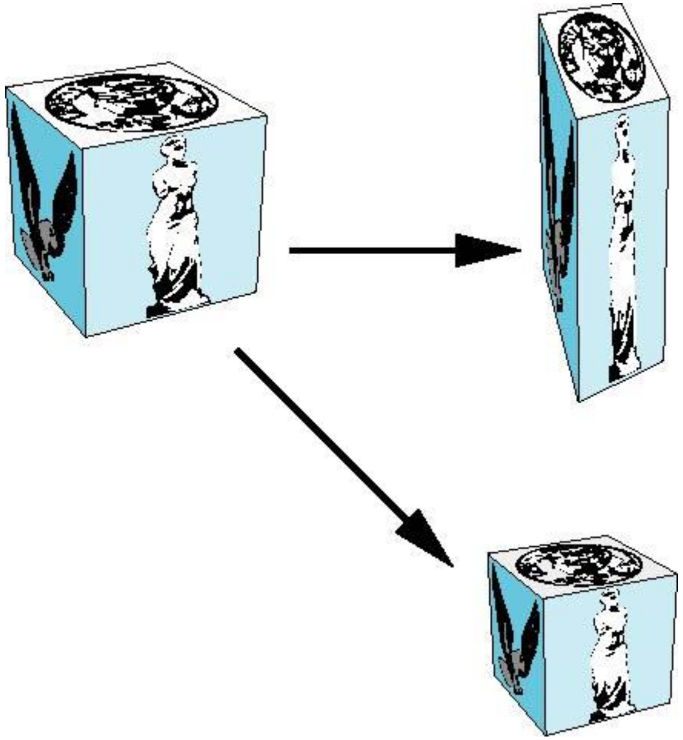
$x' = s_x x$

$y' = s_y y$

$z' = s_z z$

p'=Sp

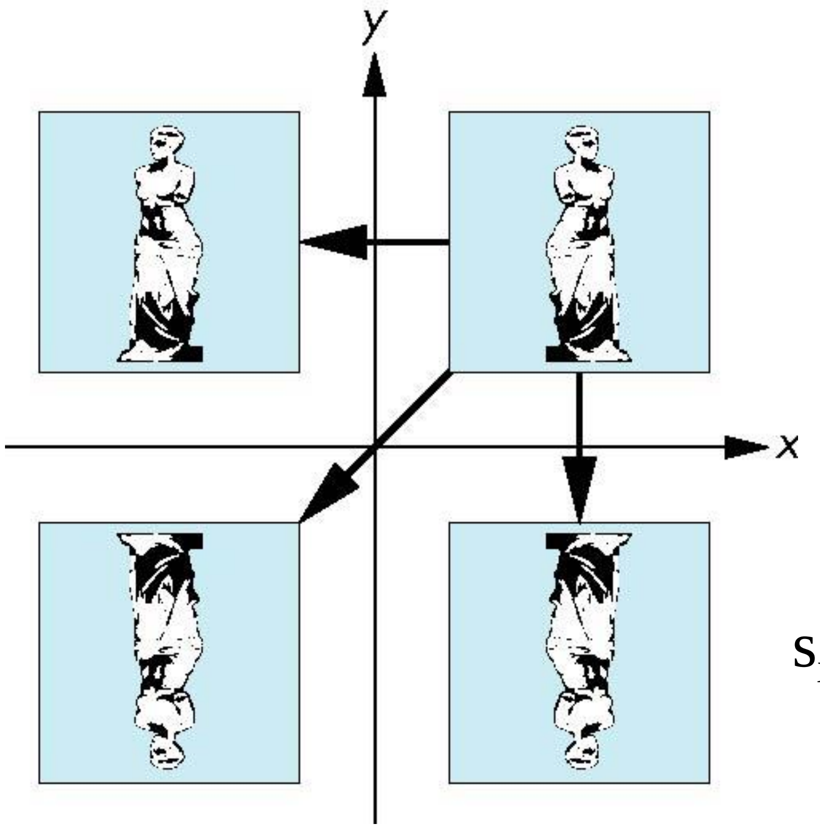
$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- To maintain aspect ratio, $s_x = s_y = s_z$

Reflection

- Corresponds to negative scale factors



$s_x = -1 \; s_y = 1$

original

$s_x = -1 \; s_y = -1$

$s_x = 1 \; s_y = -1$

CS-537: Interactive Computer Graphics

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(-q) = cos(q) and sin(-q)=-sin(q):

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

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

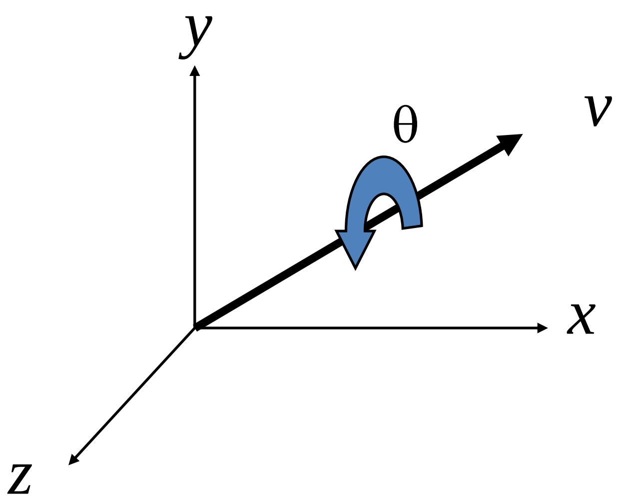
$$\mathbf{p}' = \mathbf{ABCp} = \mathbf{A}(\mathbf{B}(\mathbf{Cp}))$$

- Note many references use column matrices to represent points. In terms of column matrices

$$\mathbf{p}'^{T} = \mathbf{p}^{T}\mathbf{C}^{T}\mathbf{B}^{T}\mathbf{A}^{T}$$

General Rotation About the Origin

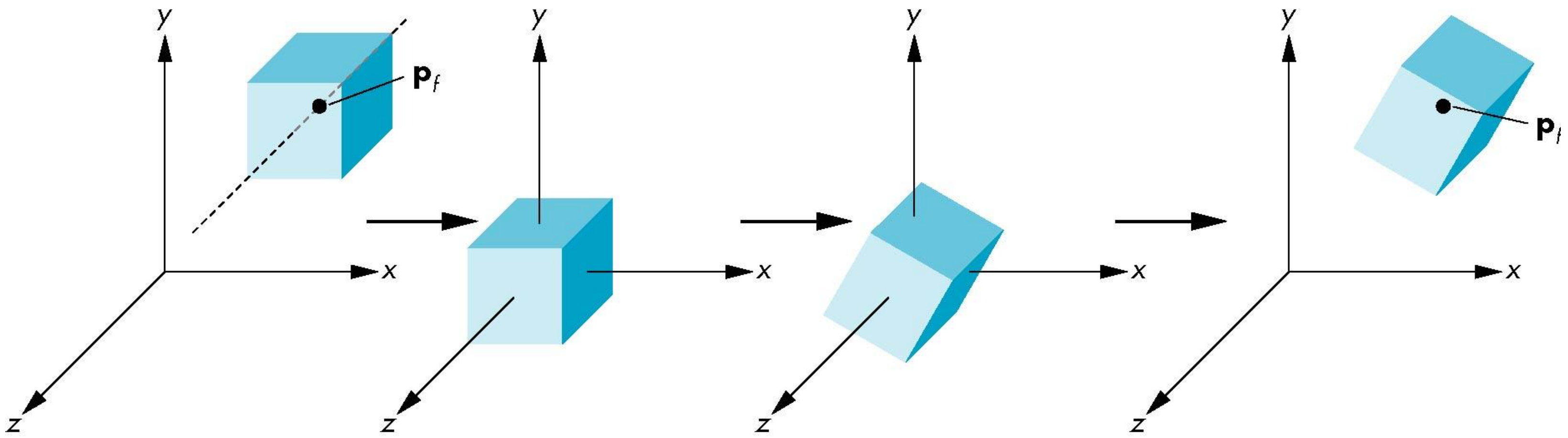
- A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the *x*, *y*, and *z* axes
- $\mathbf{R}(q) = \mathbf{R}_z(q_z)\,\mathbf{R}_y(q_y)\,\mathbf{R}_x(q_x)$
- $\theta_x\ \theta_y\ \theta_z$ are called the Euler angles
- Note that rotations do not commute (order matters). We could use rotations in another order but with different angles.



Rotation about a Fixed Point other than the Origin

- Move fixed point to origin
- Rotate
- Move fixed point back

$\mathbf{M} = \mathbf{T}(p_f)\ \mathbf{R}(\theta)\ \mathbf{T}(-p_f)$



Instancing

- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size
  - (example: Bunny OBJ file in Assignment 2)
- We apply an *instance transformation* to its vertices to scale, orient and locate each
- Allows you to use the same model for many objects

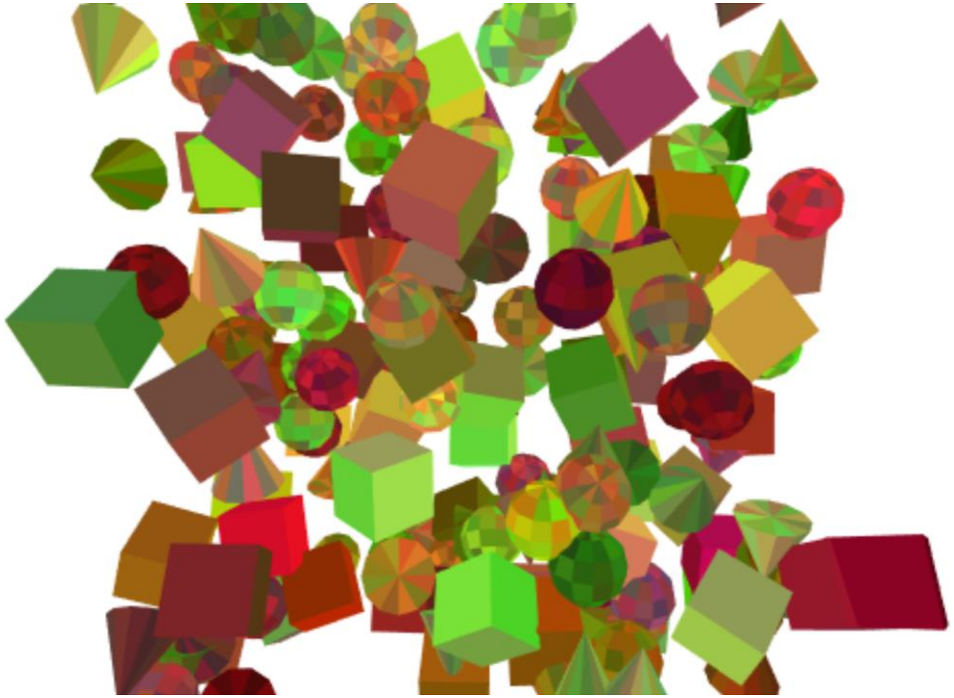
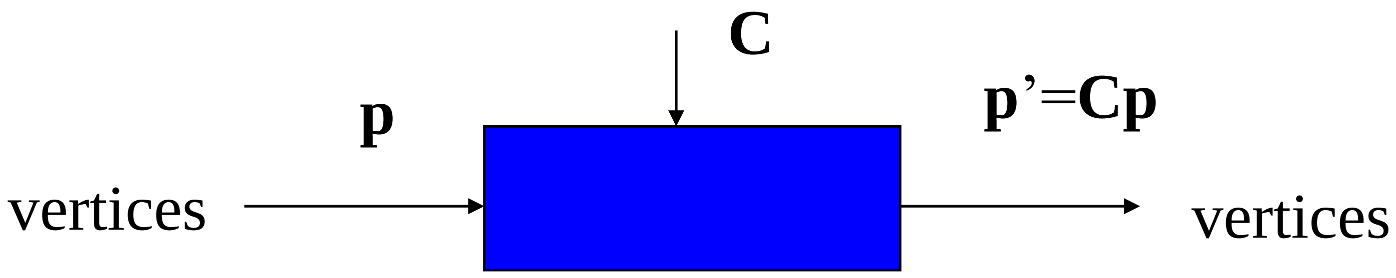


Image: WebGlFundamentals.org

Transformations in WebGL

- We will use the notion of a **current transformation matrix (CTM)** with the understanding that it may be applied in shaders
- Conceptually this is a 4 x 4 homogeneous coordinate matrix 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



CTM Operations

- The CTM can be altered either by loading a new CTM or by post-mutiplication
- Load an identity matrix: $\mathbf{C} \leftarrow \mathbf{I}$
- Load an arbitrary matrix: $\mathbf{C} \leftarrow \mathbf{M}$
- Load a translation matrix: $\mathbf{C} \leftarrow \mathbf{T}$
- Load a rotation matrix: $\mathbf{C} \leftarrow \mathbf{R}$
- Load a scaling matrix: $\mathbf{C} \leftarrow \mathbf{S}$
- Post-multiply by an arbitrary matrix: $\mathbf{C} \leftarrow \mathbf{CM}$
- Post-multiply by a translation matrix: $\mathbf{C} \leftarrow \mathbf{CT}$
- Post-multiply by a rotation matrix: $\mathbf{C} \leftarrow \mathbf{C\,R}$
- Post-multiply by a scaling matrix: $\mathbf{C} \leftarrow \mathbf{C\,S}$

CTM Example: Rotation about a Fixed Point (Attempt 1)

- Start with identity matrix: $\mathbf{C} \leftarrow \mathbf{I}$
- Move fixed point to origin: $\mathbf{C} \leftarrow \mathbf{CT}$
- Rotate: $\mathbf{C} \leftarrow \mathbf{CR}$
- Move fixed point back: $\mathbf{C} \leftarrow \mathbf{CT}^{-1}$
- Result: $\mathbf{C} = \mathbf{TR}\,\mathbf{T}^{-1}$ which is **backwards**.

- This **WRONG** result is a consequence of doing post-multiplications.
- Let's try again.

CTM Example: Rotation about a Fixed Point (Attempt 2)

- We want $\mathbf{C} = \mathbf{T}^{-1}\,\mathbf{R}\,\mathbf{T}$ so we must do the operations in the following order:

  $\mathbf{C} \leftarrow \mathbf{I}$

  $\mathbf{C} \leftarrow \mathbf{CT}^{-1}$

  $\mathbf{C} \leftarrow \mathbf{CR}$

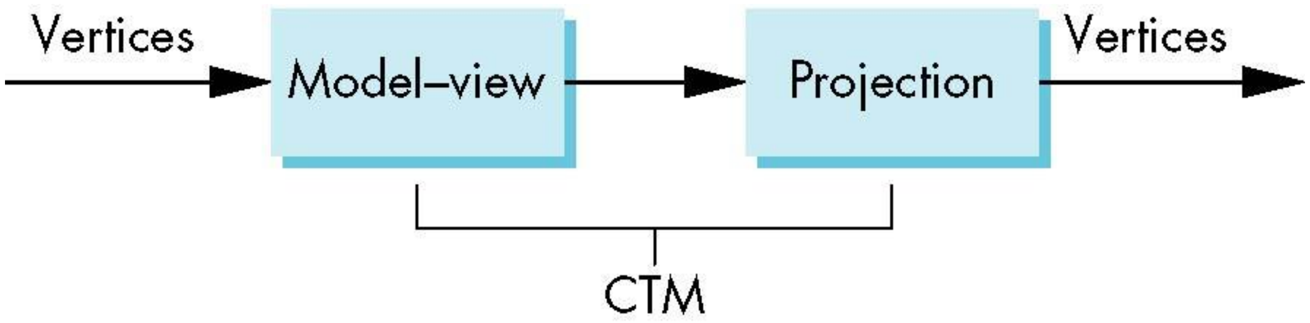
  $\mathbf{C} \leftarrow \mathbf{CT}$
- Each operation corresponds to one function call in the program.
- Note that the last operation specified is the first executed in the program!
  - (See slide 16: order of transformations)

CTM in WebGL

- OpenGL (before WebGL) had a model-view and a projection matrix in the pipeline which were concatenated together to form the CTM
- We will emulate this process

Matrix to convert from model to camera frame

Projection Matrix (next week's slides)



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 virtual 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

Rotation, Translation, Scaling in WebGL

- The file MV.js in the “Common” folder contains helpful matrix construction functions.
- Create an identity matrix:

```
var m = mat4(); // identity constructor defined in MV.js
```

- Multiply on right by rotation matrix of **theta** in degrees where (**vx, vy, vz**) define axis of rotation

```
var r = rotate(theta, vx, vy, vz) // rotate defined in MV.js
m = mult(m, r); // mult defined in MV.js
```

- Do same with translation and scaling:

```
var s = scale( sx, sy, sz); // scale defined in MV.js
var t = translate(dx, dy, dz); // translate defined in MV.js
m = mult(s, t);
```

Rotation, Translation, Scaling in WebGL: Example

- Rotation about z axis by 30 degrees with a fixed point of (1.0, 2.0, 3.0)

```
var m = mult(translate(1.0, 2.0, 3.0), rotate(30.0, 0.0, 0.0, 1.0));
m = mult(m, translate(-1.0, -2.0, -3.0));
```

- $\mathbf{m} = \mathbf{T}^{-1}\,\mathbf{R}\,\mathbf{T}$ in this example, the correct order (see slide 23)
- The first step to be applied to vertices is translating to the origin (subtraction of the fixed point's coordinates)
- Remember that last matrix specified in the program is the first applied

Arbitrary Matrices in WebGL

- Can load and multiply vertices by matrices defined in the application program in shaders
- Usually you want to pass them to shaders as uniforms, for example:
  - In shader:

    uniform mat4 modelViewMatrix;
  - In application:

    gl.uniformMatrix4f
- Matrices in application 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
- However, WebGL and our shaders expect column major data
- gl.uniformMatrix4f has a parameter for automatic transpose to convert row major to column major matrix layout
- The “flatten” function in MV.js automatically converts to column major order which is required by WebGL functions, so make sure not to apply this transpose.

Applying Transformations

- Example: begin with a cube rotating
- Use mouse or button event listener to change the direction of rotation
- Start with a program that draws a cube in a standard way
  - Centered at the origin
  - Sides aligned with axes
- Where should we apply a transformation?
  1. In application – to vertices?
  2. In vertex shader: send transformation matrix?
  3. In vertex shader: send angles of rotation?
- Better choice between 2 and 3 is unclear [amount of data transferred to GPU vs. computed in shader]. Should we perform the trigonometric operations once in the CPU or for every vertex in the shader? GPUs do have trig. functions “hardwired” in silicon

Rotation Vertex Shader (I)

```
attribute  vec4 vPosition;
attribute  vec4 vColor;
varying vec4 fColor;
uniform vec3 theta;

void main() {
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
    // Remember: these matrices are column-major
    mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                    0.0,  c.x,  s.x, 0.0,
                    0.0, -s.x,  c.x, 0.0,
                    0.0,  0.0,  0.0, 1.0 );
```

vPosition and vColor send to shader via application as attributes

varying = set fColor in vertex shader and send to fragment shader

theta sent as uniform from application note this is a vector, representing in this example rotation around x, y, z axes

Rotation Vertex Shader (II)

```
mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );
// Remember: these matrices are column-major
 mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                 s.z,  c.z, 0.0, 0.0,
                 0.0,  0.0, 1.0, 0.0,
                 0.0,  0.0, 0.0, 1.0 );

 fColor = vColor;
 gl_Position = rz * ry * rx * vPosition;
}
```

set fColor in vertex shader from vertex attribute and send to fragment shader

set gl_Position using concatenated rotation matrix