Representation

Sorin Babii
sorin.babii@cs.upt.ro

Objectives

- Introduce concepts such as dimension and basis
- Introduce coordinate systems for representing vectors spaces and frames for representing affine spaces
- Discuss change of frames and bases

Linear Independence

- A set of vectors $v_1, v_2, \ldots, v_n$ is *linearly independent* if
  $\alpha_1 v_1 + \alpha_2 v_2 + .. \alpha_n v_n = 0$ iff $\alpha_1 = \alpha_2 = \ldots = 0$
- If a set of vectors is linearly independent, we cannot represent one in terms of the others
- If a set of vectors is linearly dependent, at least one can be written in terms of the others

Dimension

- In a vector space, the maximum number of linearly independent vectors is fixed and is called the *dimension* of the space
- In an n-dimensional space, any set of n linearly independent vectors form a *basis* for the space
- Given a basis $v_1, v_2, \ldots, v_n$, any vector v can be written as

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

where the $\{\alpha_i\}$ are unique

Representation

- Until now we have been able to work with geometric entities without using any frame of reference, such as a coordinate system
- Need a frame of reference to relate points and objects to our physical world.

    For example, where is a point? Can’t answer without a reference system

    World coordinates

    Camera coordinates

Coordinate Systems

- Consider a basis $v_1, v_2, \ldots, v_n$
- A vector is written $v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$
- The list of scalars $\{\alpha_1, \alpha_2, \ldots \alpha_n\}$ is the *representation* of v with respect to the given basis
- We can write the representation as a row or column array of scalars

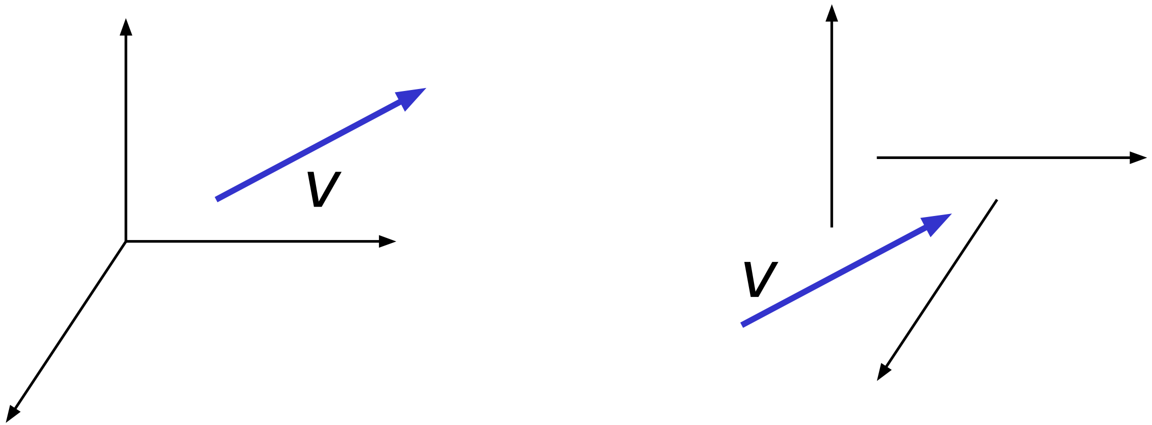
$$\mathbf{a} = [\alpha_1 \;\; \alpha_2 \;\; \ldots \; \alpha_n]^T = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \cdot \\ \alpha_n \end{bmatrix}$$

Example

- $v = 2v_1 + 3v_2 - 4v_3$
- $\mathbf{a} = [2\ 3\ -4]^T$
- Note that this representation is with respect to a particular basis
- For example, in WebGL we will start by representing vectors using the object basis but later the system needs a representation in terms of the camera or eye basis

Coordinate Systems

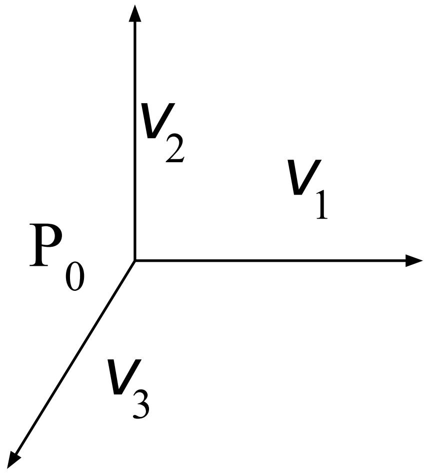
- Which is correct?



- Both are, because vectors have no fixed location

Frames

- A coordinate system is insufficient to represent points
- If we work in an affine space we can add a single point, the *origin*, to the basis vectors to form a *frame*



Representation in a Frame

- Frame determined by (P_0, v_1, v_2, v_3)
- Within this frame, every vector can be written as

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots. + \alpha_n v_n$$

- Every point can be written as

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots. + \beta_n v_n$$

Confusing Points and Vectors

Consider the point and the vector

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n$$

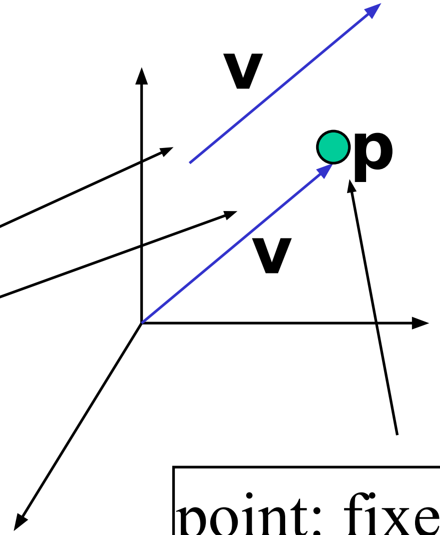
$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

They appear to have the similar representations

$$\mathbf{p} = [\beta_1\ \beta_2\ \beta_3] \qquad \mathbf{v} = [\alpha_1\ \alpha_2\ \alpha_3]$$

which confuses the point with the vector

A vector has no position



Vector can be placed anywhere

point: fixed

Homogeneous Coordinates

Objectives

- Introduce homogeneous coordinates
- Introduce change of representation for both vectors and points

A Single Representation

If we define 0•P = 0 and 1•P =P then we can write

$$v=\alpha_1 v_1+ \alpha_2 v_2 +\alpha_3 v_3 = [\alpha_1\ \alpha_2\ \alpha_3\ 0\ ]\ [v_1\ v_2\ v_3\ \ P_0]^T$$

$$P = P_0 + \beta_1 v_1+ \beta_2 v_2 +\beta_3 v_3= [\beta_1\ \beta_2\ \beta_3\ 1\ ]\ [v_1\ v_2\ v_3\ \ P_0]^T$$

Thus we obtain the four-dimensional *homogeneous coordinate* representation

$$\mathbf{v} = [\alpha_1\ \alpha_2\ \alpha_3\ 0\ ]^T$$

$$\mathbf{p} = [\beta_1\ \beta_2\ \beta_3\ 1\ ]^T$$

Homogeneous Coordinates and Computer Graphics

- Homogeneous coordinates are key to all computer graphics systems
  - All standard transformations (rotation, translation, scaling) can be implemented with matrix multiplications using 4 x 4 matrices
  - Hardware pipeline works with 4 dimensional representations
  - For orthographic viewing, we can maintain w=0 for vectors and w=1 for points
  - For perspective we need a *perspective division*

Change of Coordinate Systems

- Consider two representations of a the same vector with respect to two different bases. The representations are

$$\mathbf{a}=[\alpha_1 \ \alpha_2 \ \alpha_3]$$
$$\mathbf{b}=[\beta_1 \ \beta_2 \ \beta_3]$$

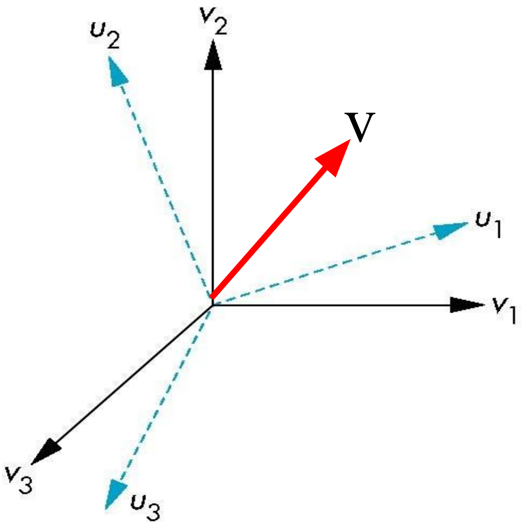
where

$$\mathrm{v}=\alpha_1 v_1+\alpha_2 v_2+\alpha_3 v_3=[\alpha_1 \ \alpha_2 \ \alpha_3]\,[v_1 \ v_2 \ v_3]^T$$
$$=\beta_1 u_1+\beta_2 u_2+\beta_3 u_3=[\beta_1 \ \beta_2 \ \beta_3]\,[u_1 \ u_2 \ u_3]^T$$

Representing second basis in terms of first

Each of the basis vectors, u_1, u_2, u_3, are vectors that can be represented in terms of the first basis

$$u_1 = \gamma_{11}v_1+\gamma_{12}v_2+\gamma_{13}v_3$$
$$u_2 = \gamma_{21}v_1+\gamma_{22}v_2+\gamma_{23}v_3$$
$$u_3 = \gamma_{31}v_1+\gamma_{32}v_2+\gamma_{33}v_3$$



Matrix Form

The coefficients define a 3 x 3 matrix

$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}$$

and the bases can be related by

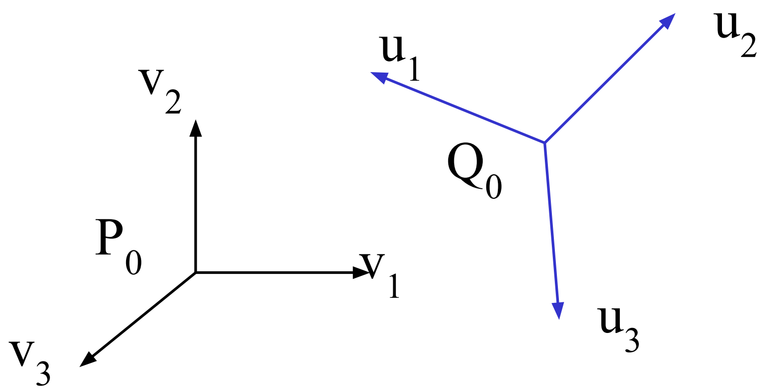
$$\mathbf{a} = \mathbf{M}^T \mathbf{b}$$

see text for numerical examples

Change of Frames

- We can apply a similar process in homogeneous coordinates to the representations of both points and vectors

Consider two frames:
(P_0, v_1, v_2, v_3)
(Q_0, u_1, u_2, u_3)



- Any point or vector can be represented in either frame
- We can represent Q_0, u_1, u_2, u_3 in terms of P_0, v_1, v_2, v_3

Representing One Frame in Terms of the Other

Extending what we did with change of bases

$$
\begin{aligned}
u_1 &= \gamma_{11}v_1+\gamma_{12}v_2+\gamma_{13}v_3\\
u_2 &= \gamma_{21}v_1+\gamma_{22}v_2+\gamma_{23}v_3\\
u_3 &= \gamma_{31}v_1+\gamma_{32}v_2+\gamma_{33}v_3\\
Q_0 &= \gamma_{41}v_1+\gamma_{42}v_2+\gamma_{43}v_3+\gamma_{44}P_0
\end{aligned}
$$

defining a 4 x 4 matrix

$$
\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}
$$

Working with Representations

Within the two frames any point or vector has a representation of the same form

a=[α_1 α_2 α_3 α_4] in the first frame
b=[β_1 β_2 β_3 β_4] in the second frame

where $\alpha_4 = \beta_4 = 1$ for points and $\alpha_4 = \beta_4 = 0$ for vectors and

$$\mathbf{a} = \mathbf{M}^T \mathbf{b}$$

The matrix **M** is 4 x 4 and specifies an affine transformation in homogeneous coordinates

Affine Transformations

- Every linear transformation is equivalent to a change in frames
- Every affine transformation preserves lines
- However, an affine transformation has only 12 *degrees of freedom* because 4 of the elements in the matrix are fixed and are a subset of all possible 4 x 4 linear transformations

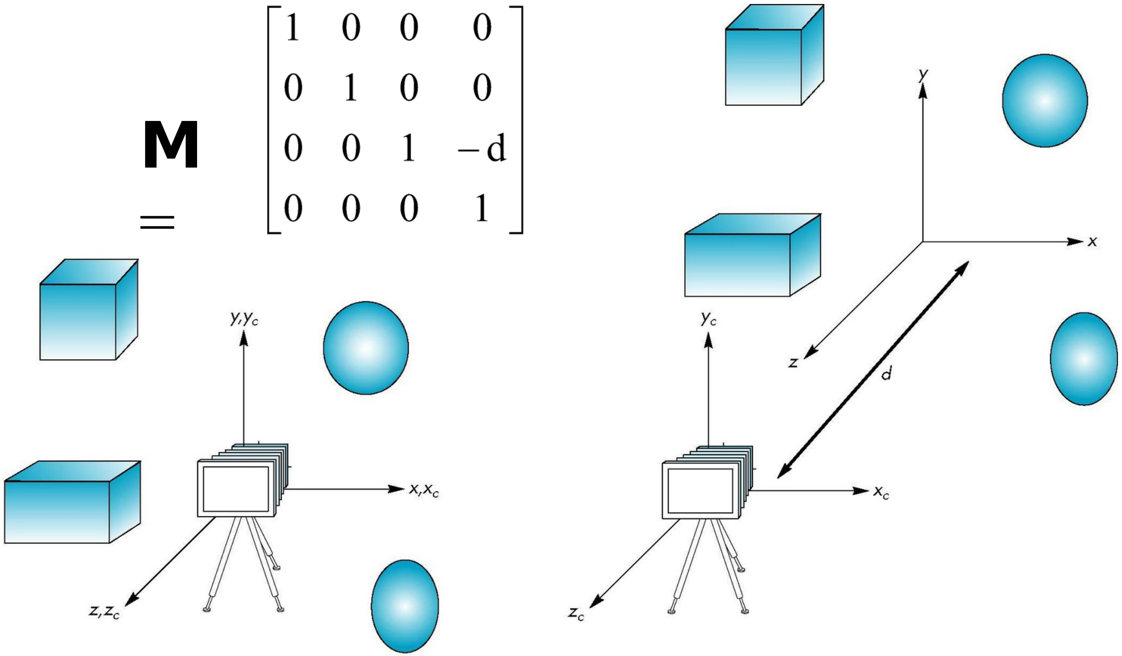
The World and Camera Frames

- When we work with representations, we work with n-tuples or arrays of scalars
- Changes in frame are then defined by 4 x 4 matrices
- In OpenGL, the base frame that we start with is the world frame
- Eventually we represent entities in the camera frame by changing the world representation using the model-view matrix
- Initially these frames are the same (**M**=**I**)

Moving the Camera

If objects are on both sides of z=0, we must move camera frame

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



(a) (b)

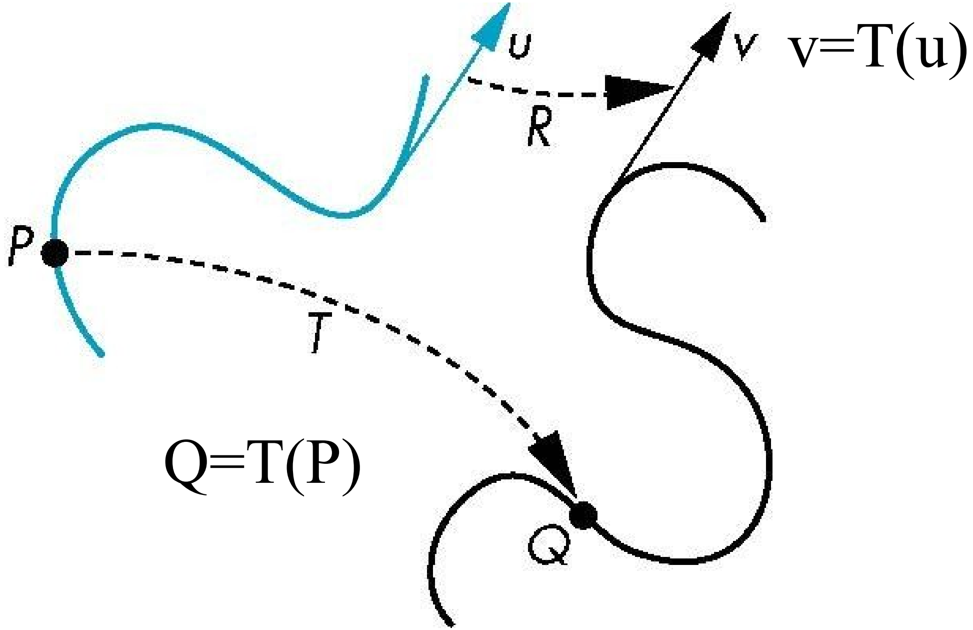
Transformations

Objectives

- Introduce standard transformations
  - Rotation
  - Translation
  - Scaling
  - Shear
- Derive homogeneous coordinate transformation matrices
- Learn to build arbitrary transformation matrices from simple transformations

General Transformations

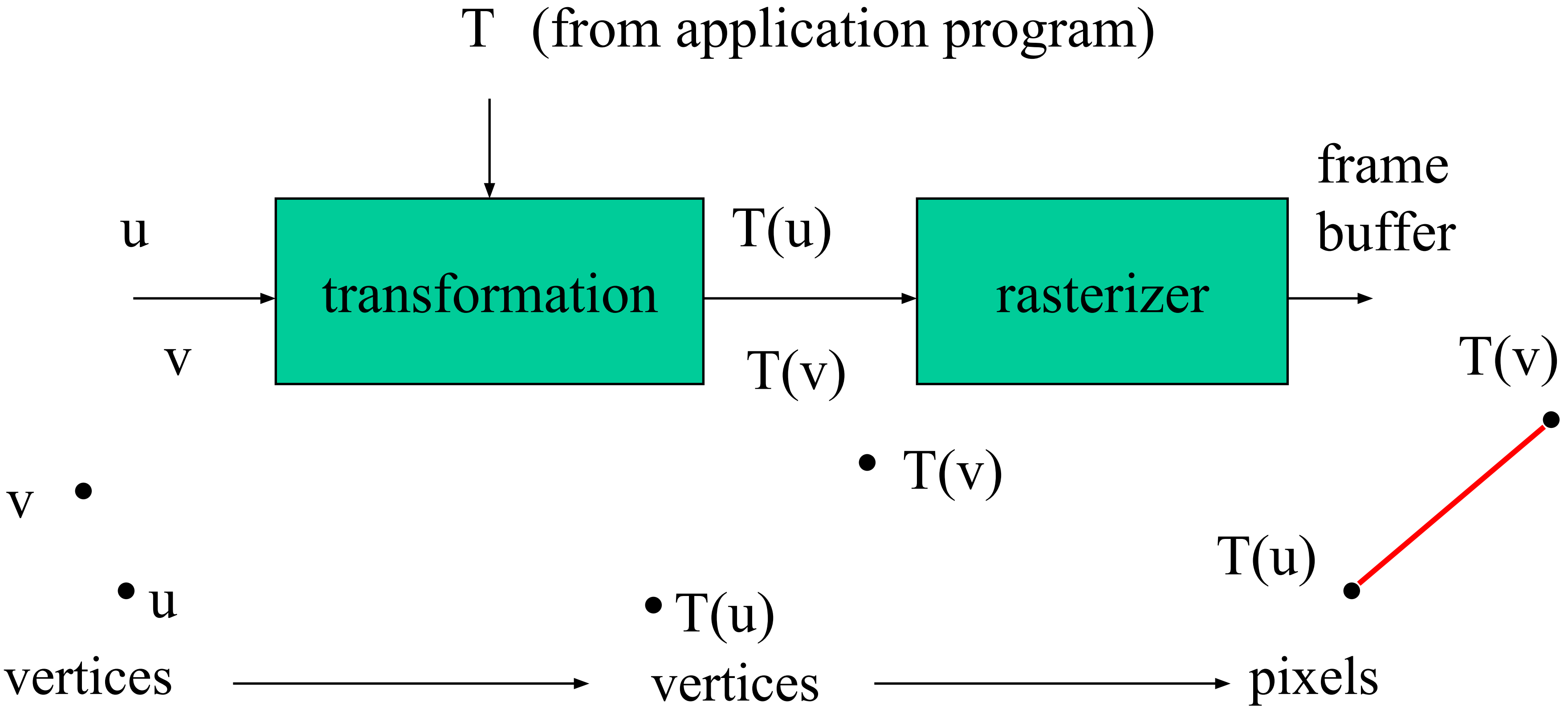
A transformation maps points to other points and/or vectors to other vectors



Affine Transformations

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

Pipeline Implementation



Notation

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

p, **q**, **r**: representations of points

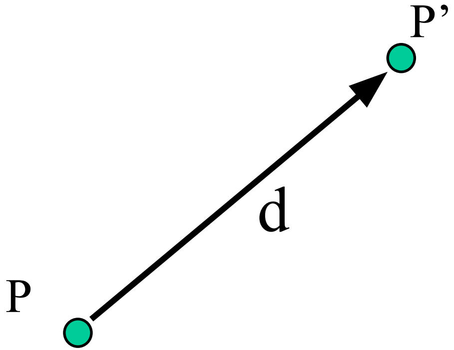
-array of 4 scalars in homogeneous coordinates

u, **v**, **w**: representations of vectors

-array of 4 scalars in homogeneous coordinates

Translation

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



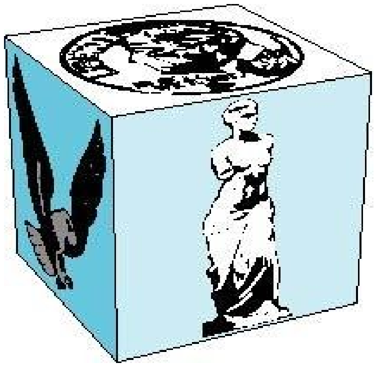
- Displacement determined by a vector d

    Three degrees of freedom

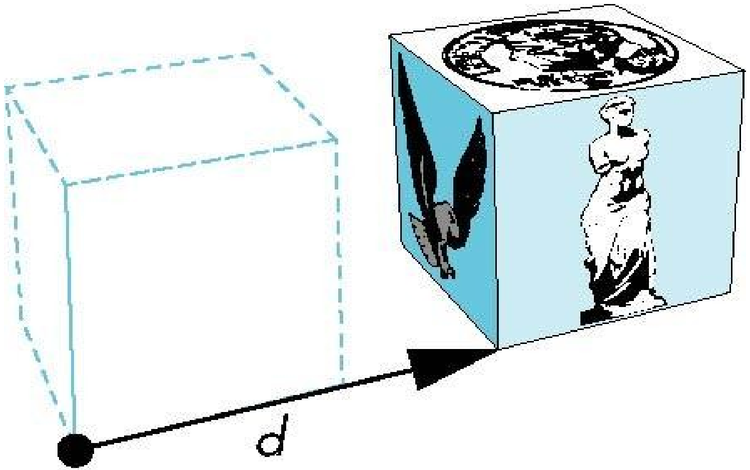
    P'=P+d

How many ways?

Although we can move a point to a new location in infinite ways, when we move many points there is usually only one way



object



translation: every point displaced by same vector

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 **T** in homogeneous coordinates

p'=Tp where

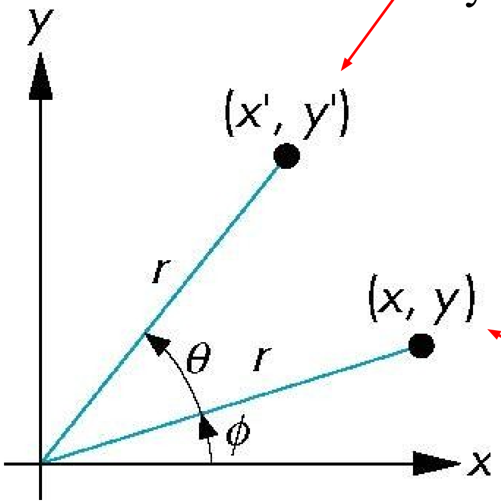
$$\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

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



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

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

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

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

Rotation Matrix

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

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

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

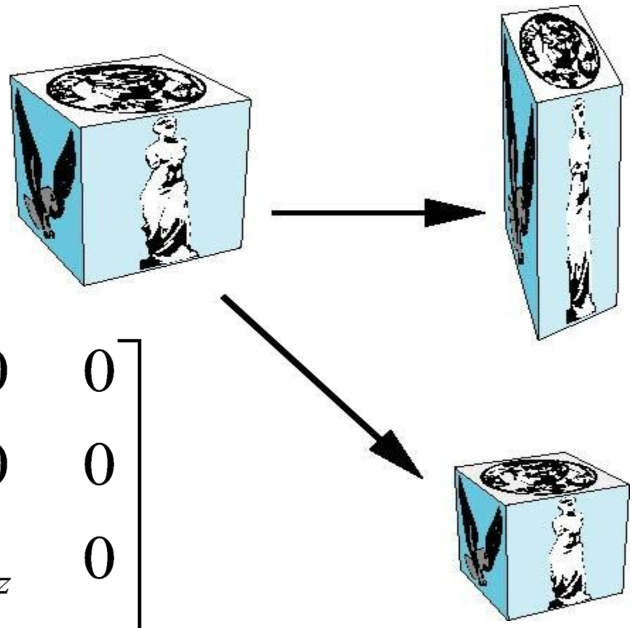
Scaling

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

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

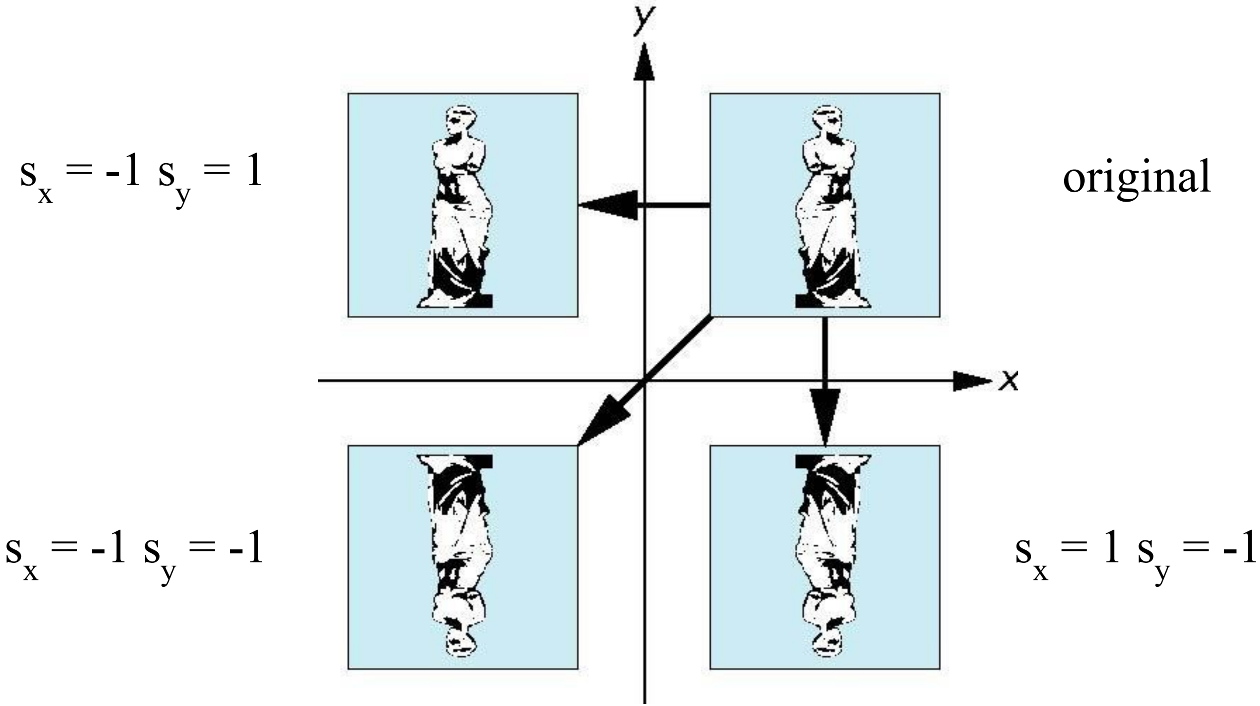
$$\mathbf{p}' = \mathbf{S}\mathbf{p}$$

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



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

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

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

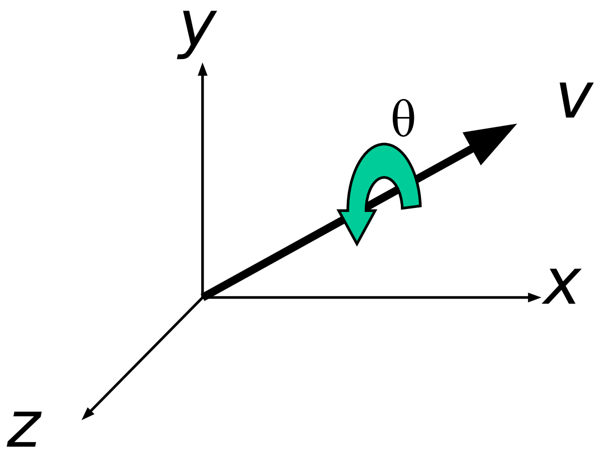
General Rotation About the Origin

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

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

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

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



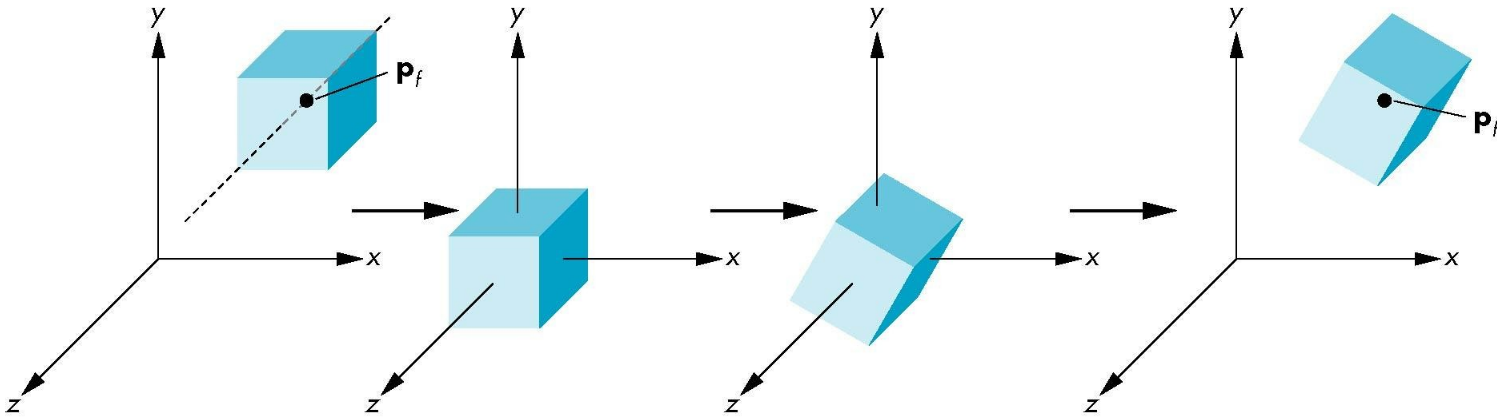
Rotation About a Fixed Point other than the Origin

Move fixed point to origin

Rotate

Move fixed point back

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



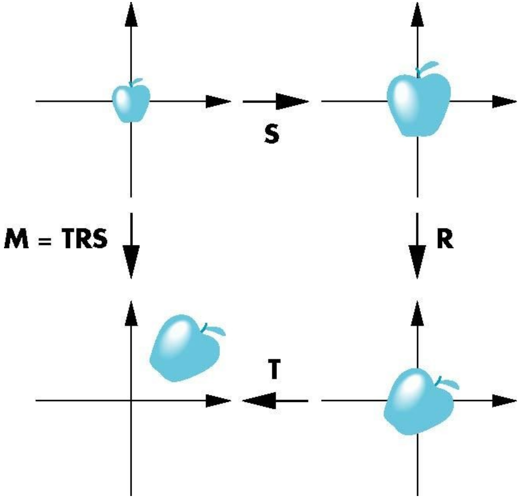
Instancing

- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size
- We apply an *instance transformation* to its vertices to

    Scale

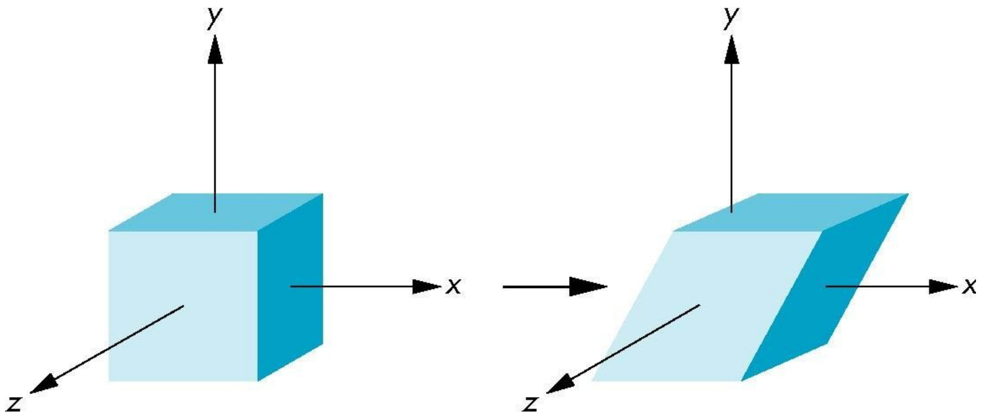
    Orient

    Locate



Shear

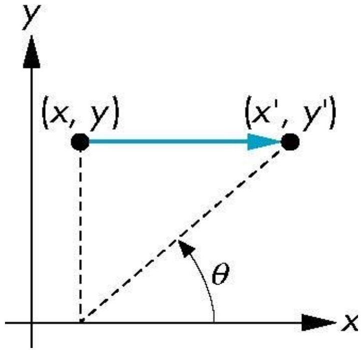
- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions



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

Objectives

- Learn how to carry out transformations in WebGL
  - Rotation
  - Translation
  - Scaling
- Introduce MV.js transformations
  - Model-view
  - Projection

Pre 3.1 OpenGL Matrices

- In Pre 3.1 OpenGL matrices were *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

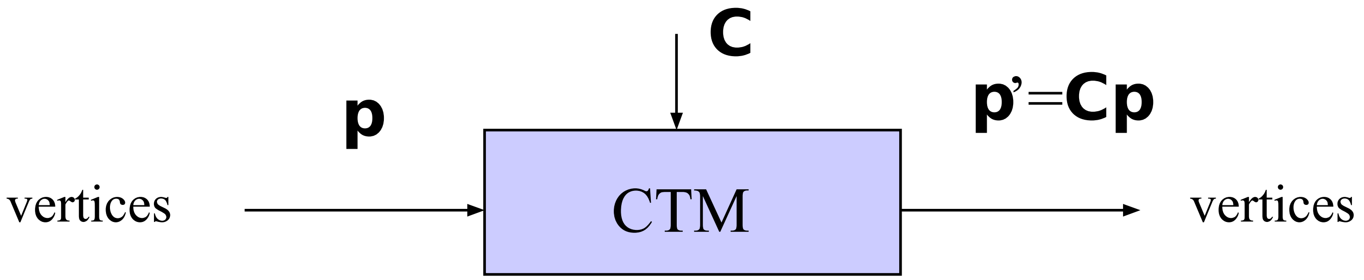
```
glMatrixMode(GL_MODELVIEW);
glMatrixMode(GL_PROJECTION);
```

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** meaning 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



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: $\mathbf{C} \leftarrow \mathbf{S}$

Postmultiply by an arbitrary matrix: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{M}$
Postmultiply by a translation matrix: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{T}$
Postmultiply by a rotation matrix: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{R}$
Postmultiply by a scaling matrix: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{S}$

Rotation about a Fixed Point

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 result is a consequence of doing post-multiplications.

Let's try again.

Reversing the Order

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

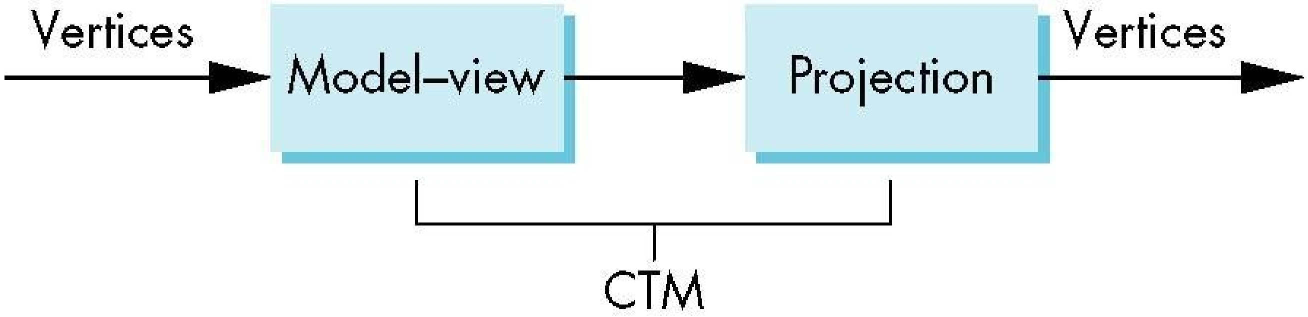
$\mathbf{C} \leftarrow \mathbf{I}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}^{-1}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{R}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$

Each operation corresponds to one function call in the program.

Note that the last operation specified is the first executed in the program

CTM in WebGL

- OpenGL had a model-view and a projection matrix in the pipeline which were 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

$$q = P \cdot MV \cdot p$$

Rotation, Translation, Scaling

Create an identity matrix:

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

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)
m = mult(m, r);
```

Also have rotateX, rotateY, rotateZ

The same with translation and scaling:

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

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));
```

- Remember that last matrix specified in the program is the first applied

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.uniformMatrix4f` has a parameter for automatic transpose by it must be set to false.
- `flatten` function converts to column major order which is required by WebGL functions

Matrix Stacks

- In many situations we want to save transformation matrices for use later
  - Traversing hierarchical data structures (Chapter 9)
- 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(modelViewMatrix);
...
modelViewMatrix = stack.pop();
```

Applying Transformations

Using Transformations

- Example: Begin with a cube rotating - video w04/cube
- 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 in next lecture

Where do we apply transformation?

- Same issue as with rotating square
    - in application to vertices
    - in vertex shader: send MV matrix
    - in vertex shader: send angles
- Choice between second and third - unclear
- Do we do trigonometry once in CPU or for every vertex in shader?
    - GPUs have trigonometry functions hardwired

Rotation Event Listeners

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

Rotation Event Listeners

```
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 );
}
```

Rotation Shader

```
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 );
```

Rotation Shader (cont)

```
// 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 );
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 );
 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 );
```

Rotation Shader (cont)

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

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 (see text)

Incremental Rotation

- Consider the two approaches

  For a sequence of rotation matrices $\mathbf{R_0}, \mathbf{R_1}, \ldots, \mathbf{R_n}$, find the Euler angles for each and use $\mathbf{R_i} = \mathbf{R_{iz}}\,\mathbf{R_{iy}}\,\mathbf{R_{ix}}$

    - Not very efficient

  Use the final positions to determine the axis and angle of rotation, then increment only the angle

- Quaternions can be more efficient

Quaternions

- Extension of imaginary numbers from two to three dimensions
- Requires one real and three imaginary components **i, j, k**

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

- Quaternions can express rotations on sphere smoothly and efficiently. Process:

  Model-view matrix → quaternion
  Carry out operations with quaternions
  Quaternion → Model-view matrix

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