Introduction to Computer Graphics with WebGL

Week5

Instructor: Hooman Salamat

Geometry

Objectives

- Introduce the elements of geometry
  - Scalars
  - Vectors
  - Points
- Develop mathematical operations among them in a coordinate-free manner
- Define basic primitives
  - Line segments
  - Polygons

Basic Elements

- Geometry is the study of the relationships among objects in an n-dimensional space
  - In computer graphics, we are interested in objects that exist in three dimensions
- Want a minimum set of primitives from which we can build more sophisticated objects
- We will need three basic elements
  - Scalars
  - Vectors
  - Points

Coordinate-Free Geometry

- When we learned simple geometry, most of us started with a Cartesian approach
  - Points were at locations in space **p**=(x,y,z)
  - We derived results by algebraic manipulations involving these coordinates
- This approach was nonphysical
  - Physically, points exist regardless of the location of an arbitrary coordinate system
  - Most geometric results are independent of the coordinate system
  - Example Euclidean geometry: two triangles are identical if two corresponding sides and the angle between them are identical

Homogeneous Coordinates

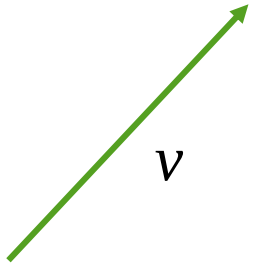
- You can specify a point or a vector in 3D with 3 coordinates. However, this can be a bit confusing since both points and vectors are specified in the same way.
- With homogeneous coordinates, a forth coordinate “w” is added.
- For vectors, w = 0
- You can easily convert from a homogenous point (p_x,p_y,p_z,p_w) to a point represented by three coordinates, by dividing all coordinates(p_x,p_y,p_z) by p_w
- You simply add a 1 at the forth position to get the corresponding point $(p_x,p_y,p_z,1)$ in homogenous coordinates.
- If you represent a point with 4 homogenous coordinates, it’s possible to represent transformations (such as translations, rotations, scaling, and shearing) with a 4x4 matrix.

Scalars

- Need three basic elements in geometry
  - Scalars, Vectors, Points
- Scalars can be defined as members of sets which can be combined by two operations (addition and multiplication) obeying some fundamental axioms (associativity, commutivity, inverses)
- Examples include the real and complex number systems under the ordinary rules with which we are familiar
- Scalars alone have no geometric properties

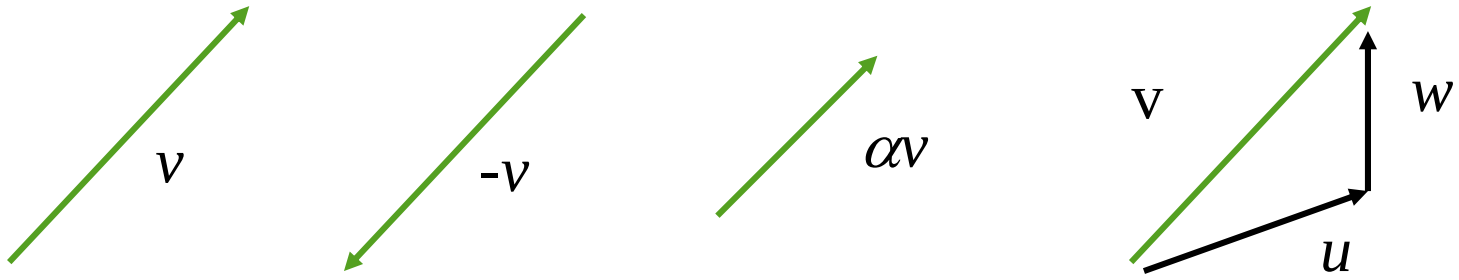
Vectors

- Physical definition: a vector is a quantity with two attributes
  - Direction
  - Magnitude
- Examples include
  - Force
  - Velocity
  - Directed line segments
    - Most important example for graphics
    - Can map to other types



Vector Operations

- Every vector has an inverse
  - Same magnitude but points in opposite direction
- Every vector can be multiplied by a scalar
- There is a zero vector
  - Zero magnitude, undefined orientation
- The sum of any two vectors is a vector
  - Use head-to-tail axiom



Linear Vector Spaces

- Mathematical system for manipulating vectors
- Operations
  - Scalar-vector multiplication $u=\alpha v=(\alpha\, v_x\,,\ \alpha v_y,\ \alpha v_z)$
  - Vector-vector addition: $w=u+v=(u_x+v_x, u_y+v_y, u_z+v_z)$
- Expressions such as

  $v=u+2w-3r$

Make sense in a vector space

Dot Product

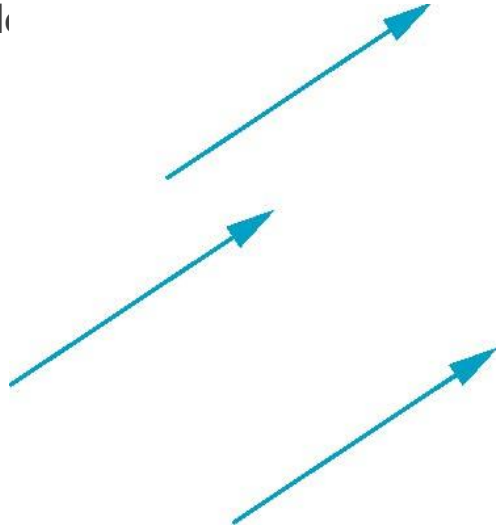
- For two vectors u and v, the dot product can be defined in the following way:
- u.v = |u||v| cos(a)
- This definition is based on the length of the two vectors and the smallest angle between them.
  - Cos(90) = 0 ➔ u.v = 0
  - 0<= a < 90 ➔ u.v > 0
  - a > 90 ➔ u.v < 0
- Vector-vector algebraic multiplication: $w=u.v= u_x v_x + u_y v_y + u_z v_z$

Cross product

- The cross product of u and v is denoted by
- $w = u \times v = (u_y v_z - u_z v_y , u_z v_x - u_x v_z , u_x v_y - u_y v_x)$
- The result of a cross product is a new vector w with the following properties
- $|w| = |u||v| \sin(a)$ (where a is the smallest angle between u and v)
- w is orthogonal against both u and v
- w is right handed with respect to both u and v
- $u \times v = 0$ if and only if u and v are parallel ($\sin(a) = 0$)
- The cross product is not commutative. $u \times v = -v \times u$
- An important usage of the cross product in 3D graphics is to calculate normal for a surface such as triangle.

Vectors Lack Position

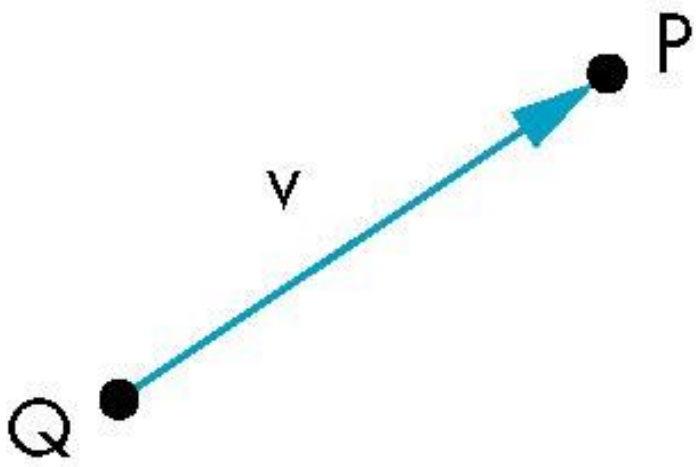
- These vectors are identical
  - Same length and magnitud



- Vectors spaces insufficient for geometry
  - Need points

Points

- Location in space
- Operations allowed between points and vectors
  - Point-point subtraction yields a vector
  - Equivalent to point-vector addition



v=P-Q

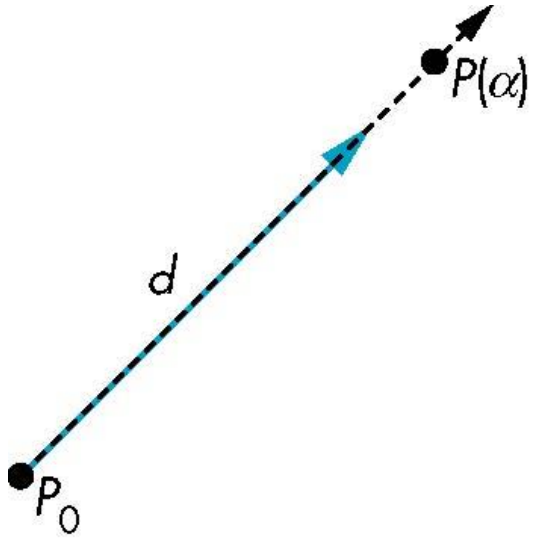
P=v+Q

Affine Spaces

- Point + a vector space
- Operations
  - Vector-vector addition
  - Scalar-vector multiplication
  - Point-vector addition
  - Scalar-scalar operations
  - Affine transformation maps parallel lines to parallel lines and finite points to finite points.
- For any point define
  - $1 \cdot P = P$
  - $0 \cdot P = \mathbf{0}$ (zero vector)

Lines

- Consider all points of the form
  - $P(\alpha)=P_0 + \alpha\ \mathbf{d}$
  - Set of all points that pass through P_0 in the direction of the vector $\mathbf{d}$



Parametric Form

- This form is known as the parametric form of the line
  - More robust and general than other forms
  - Extends to curves and surfaces
- Two-dimensional forms
  - Explicit: $y = mx + h$
  - Implicit: $ax + by + c = 0$
  - Parametric:

$$x(\alpha) = \alpha x_0 + (1-\alpha)x_1$$

$$y(\alpha) = \alpha y_0 + (1-\alpha)y_1$$

Rays and Line Segments

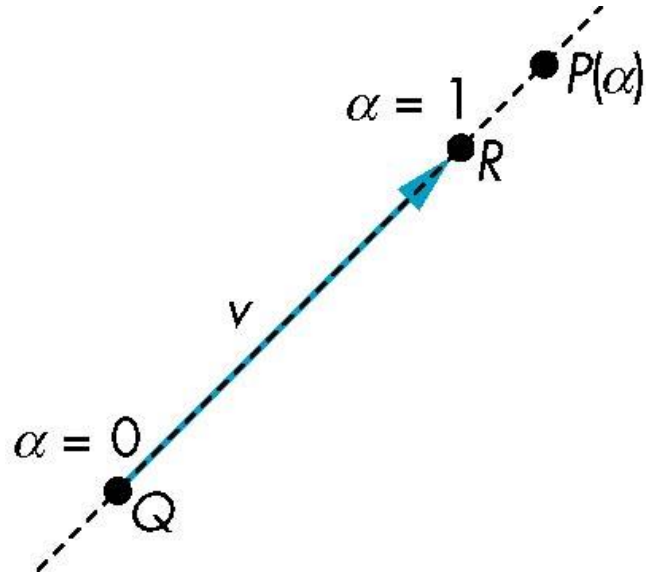
- If $\alpha >= 0$, then $P(\alpha)$ is the *ray* leaving P_0 in the direction $\mathbf{d}$

If we use two points to define v, then

$P(\alpha) = Q + \alpha (R-Q) = Q + \alpha v$

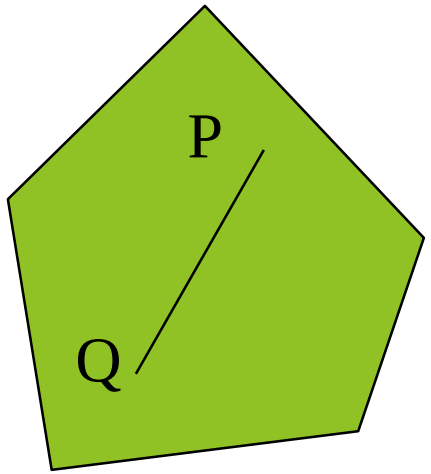
$= \alpha R + (1-\alpha) Q$

For $0 <= \alpha <= 1$ we get all the points on the *line segment* joining R and Q

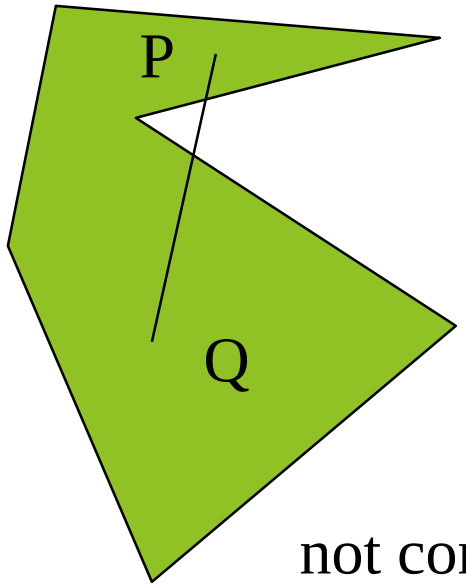


Convexity

- An object is *convex* if for any two points in the object all points on the line segment between these points are also in the object



convex



not convex

Affine Sums

- Consider the "sum"

$P=\alpha_1 P_1+\alpha_2 P_2+\ldots..+\alpha_n P_n$

Can show by induction that this sum makes sense if

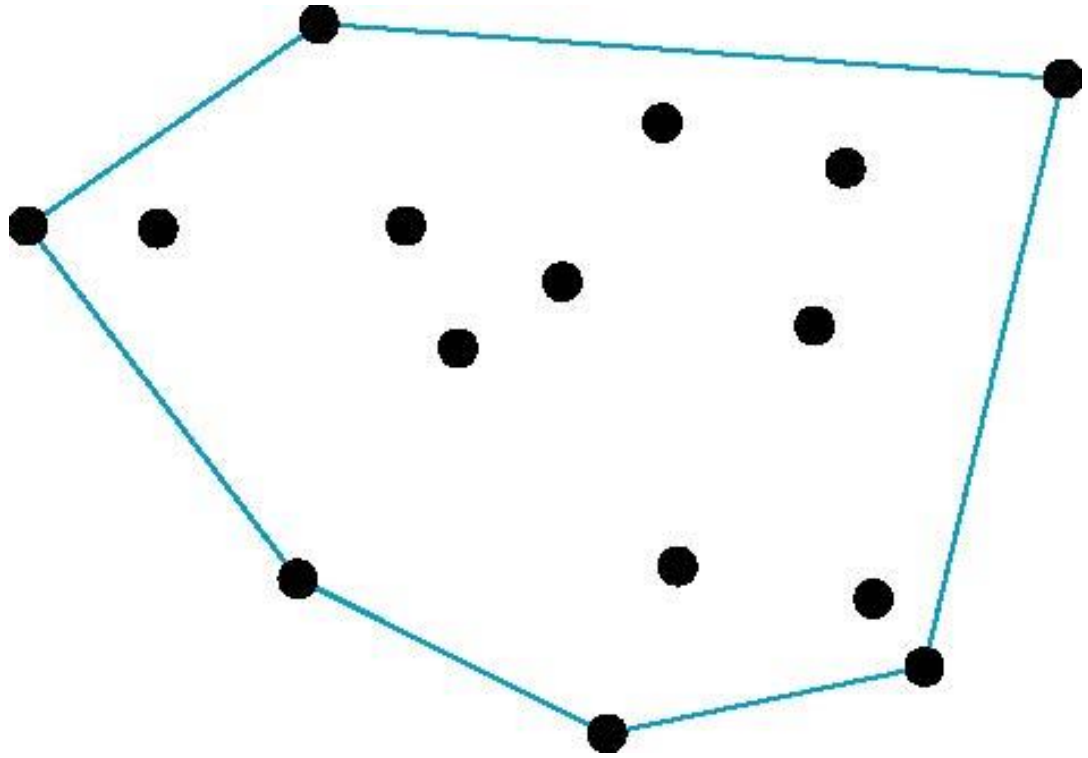
$\alpha_1+\alpha_2+\ldots..\alpha_n=1$

in which case we have the *affine sum* of the points $P_1,P_2,\ldots..P_n$

- If, in addition, $\alpha_i>=0$, we have the *convex hull* of $P_1,P_2,\ldots..P_n$

Convex Hull

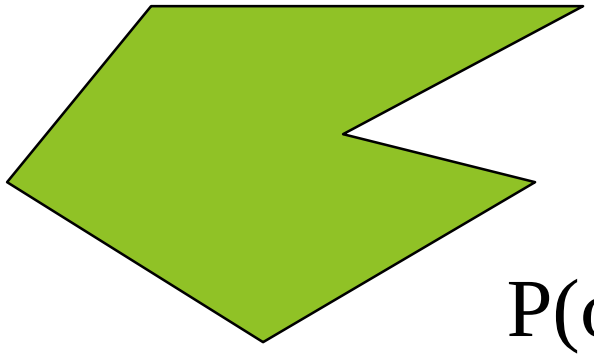
- Smallest convex object containing $P_1, P_2, \ldots . P_n$
- Formed by “shrink wrapping” points



Curves and Surfaces

- Curves are one parameter entities of the form $P(\alpha)$ where the function is nonlinear
- Surfaces are formed from two-parameter functions $P(\alpha, \beta)$
  - Linear functions give planes and polygons

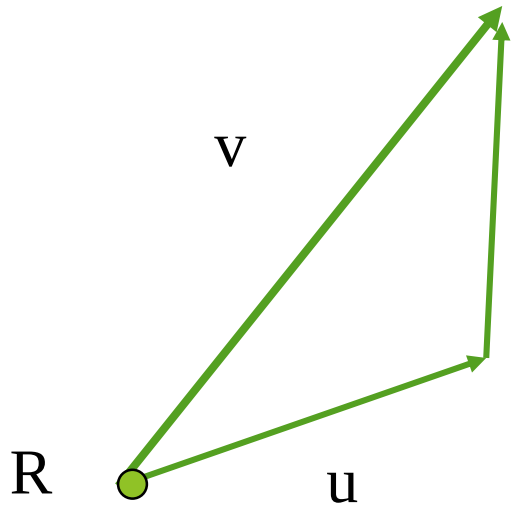
$P(\alpha)$



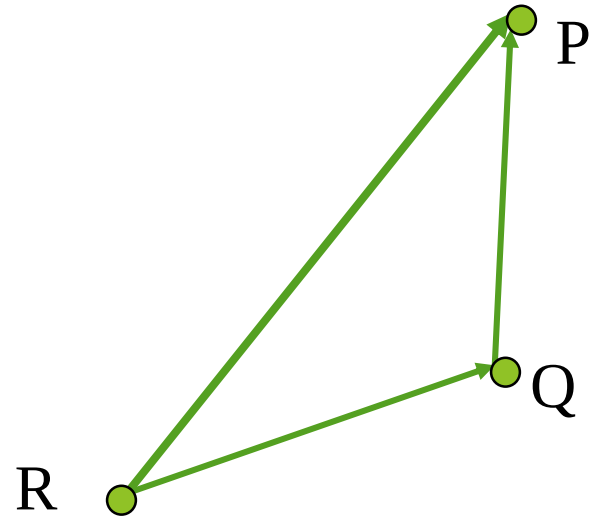
$P(\alpha, \beta)$

Planes

- A plane can be defined by a point and two vectors or by three points

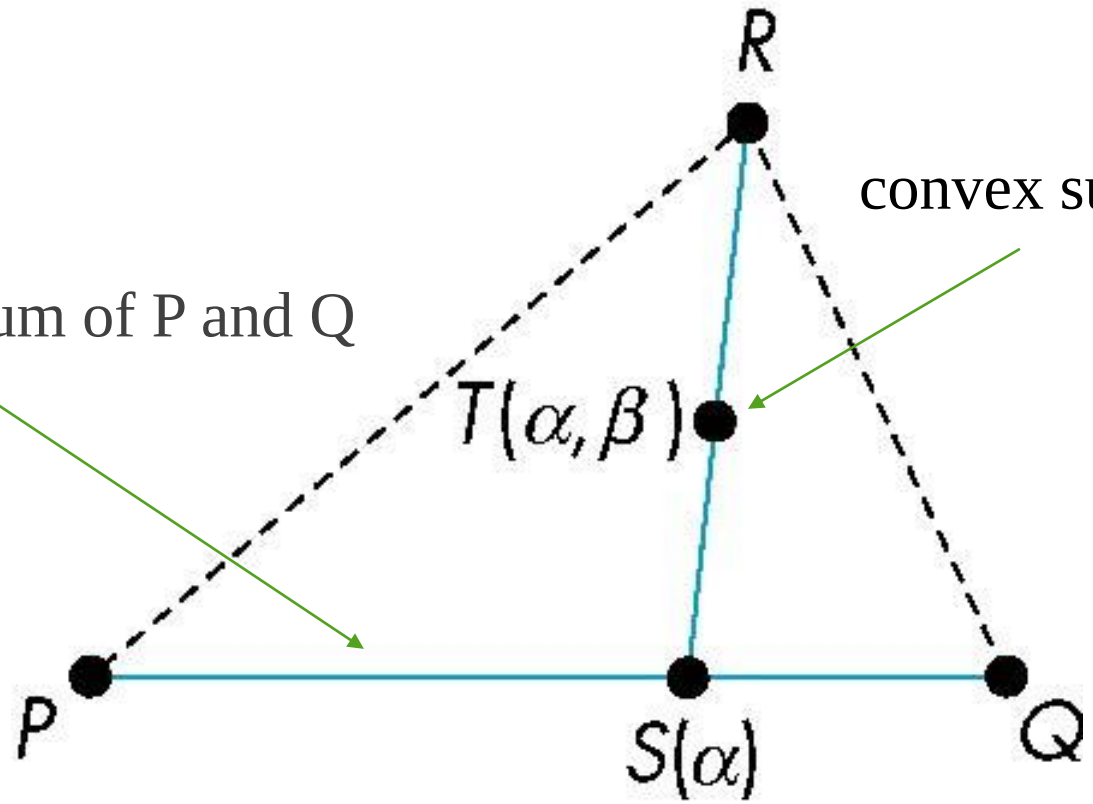


$P(\alpha,\beta)=R+\alpha u+\beta v$



$P(\alpha,\beta)=R+\alpha(Q-R)+\beta(P-Q)$

Triangles



for 0<=α,β<=1, we get all points in triangle

Barycentric Coordinates

Triangle is convex so any point inside can be represented as an affine sum

$P(\alpha_1, \alpha_2, \alpha_3)=\alpha_1 P+\alpha_2 Q+\alpha_3 R$

where

$\alpha_1 +\alpha_2 +\alpha_3 = 1$

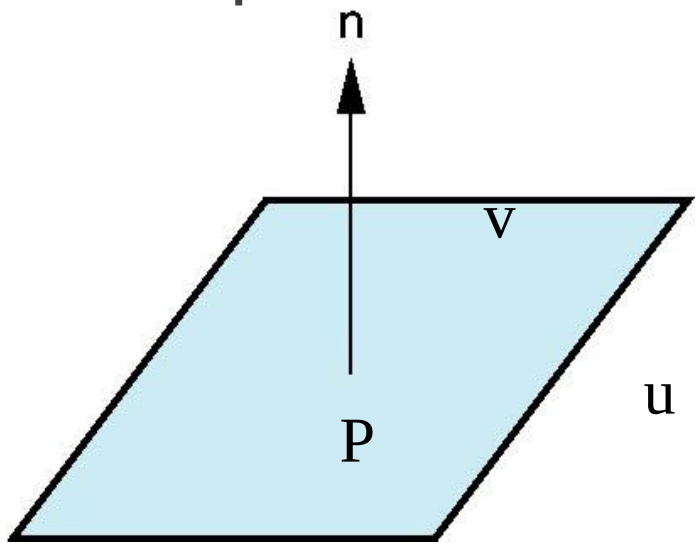
$\alpha_i >= 0$

The representation is called the **barycentric coordinate** representation of P

Normals

- In three dimensional spaces, every plane has a vector n perpendicular or orthogonal to it called the **normal vector**
- From the two-point vector form $P(\alpha,\beta)=P+\alpha u+\beta v$, we know we can use the cross product to find $n = u \times v$ and the equivalent form

$$(P(\alpha, \beta)-P) \cdot n=0$$



Representation

- 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, …, 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

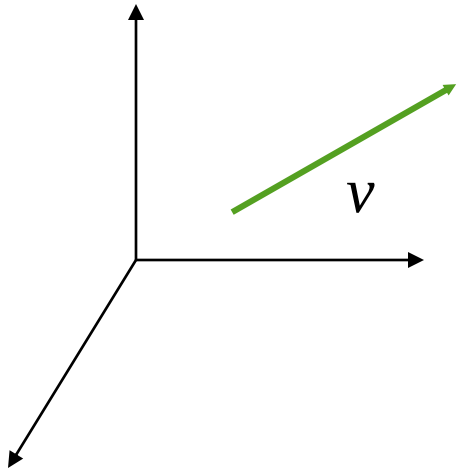
$$\mathbf{a} = [\alpha_1 \;\; \alpha_2 \;\; \ldots \alpha_n]^T = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ . \\ \alpha_n \end{bmatrix}$$

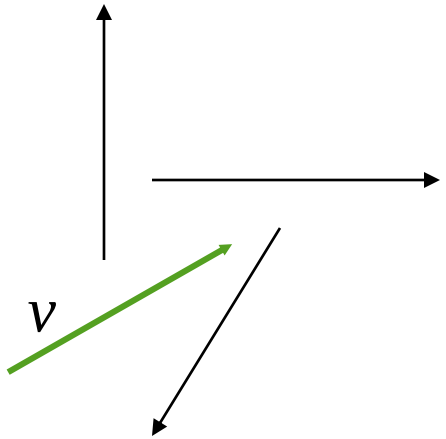
Example

- $v=2v_1+3v_2-4v_3$
- $\mathbf{a}=[2\ 3\ -4]^T$
- 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?

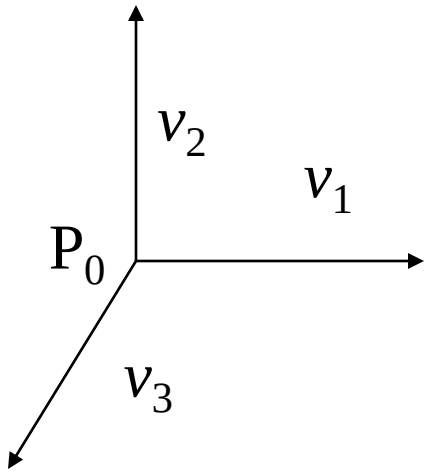




- Both are because vectors have no fixed location

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

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

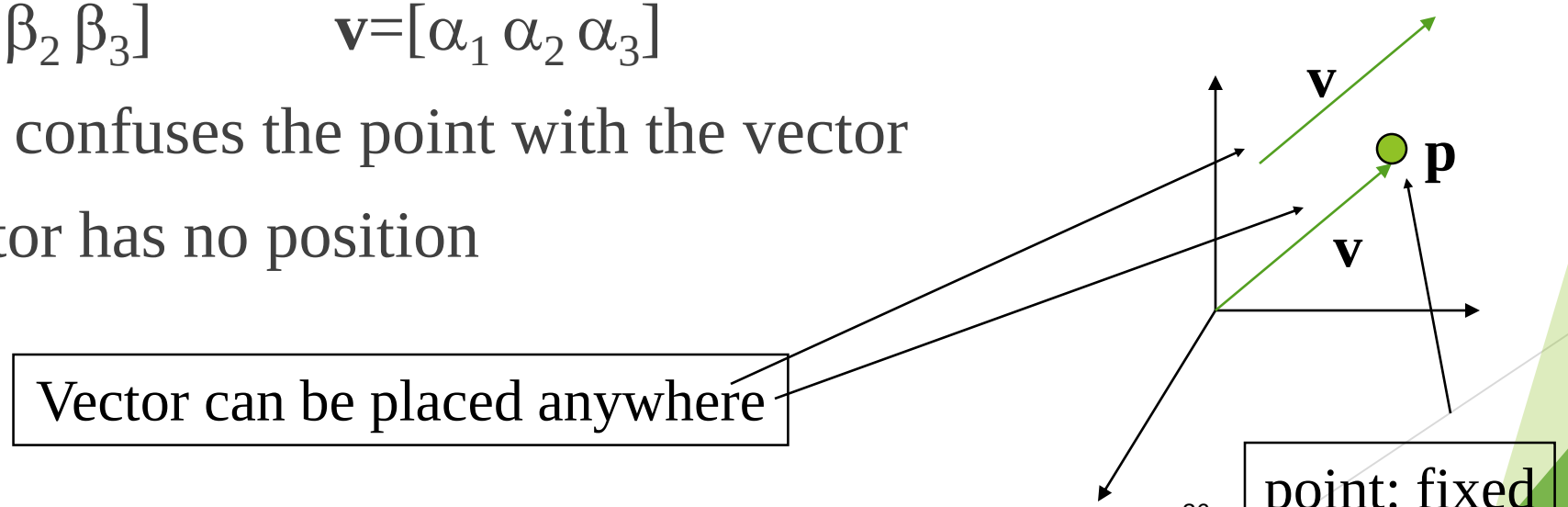
$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

They appear to have the similar representations

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

which confuses the point with the vector

A vector has no position



A Single Representation

If we define $0 \bullet P = \mathbf{0}$ and $1 \bullet P = P$ 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

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

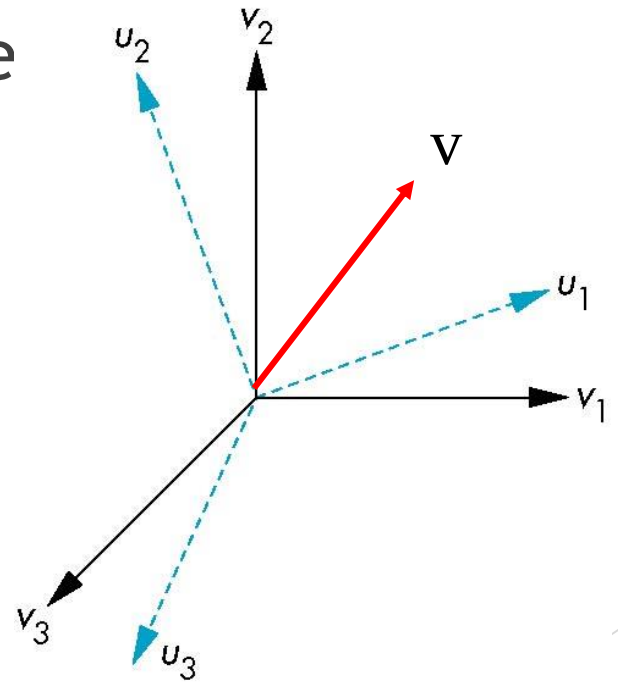
where

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

Representing second basis in terms of first

Each of the basis vectors, u1,u2, u3, are vectors that can be represented in terms of the

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



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

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

see text for numerical examples

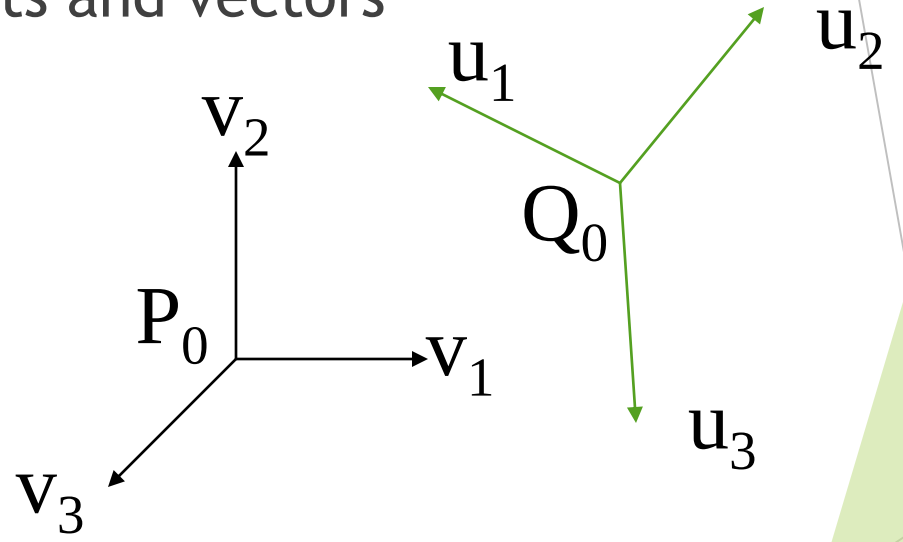
Change of Frames

- 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

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

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

$\mathbf{a}=[\alpha_1\ \alpha_2\ \alpha_3\ \alpha_4]$ in the first frame
$\mathbf{b}=[\beta_1\ \beta_2\ \beta_3\ \beta_4]$ 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 $\mathbf{M}$ 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

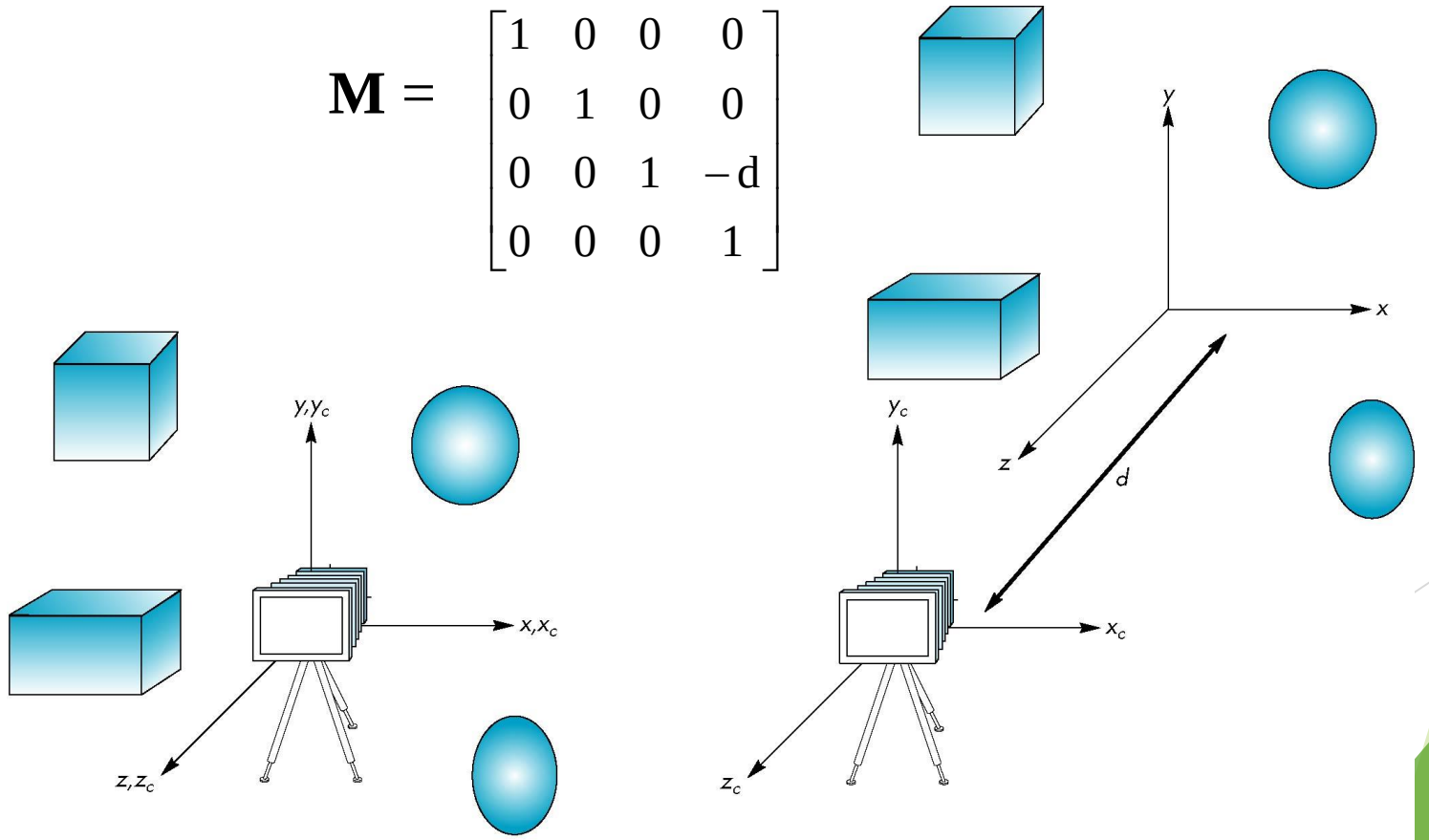
The World and Camera Frames

- 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 ($\mathbf{M}=\mathbf{I}$)

Moving the Camera

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

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



Working With Matrices and Vectors in JavaScript

- JavaScript doesn't have any built-in support to handle matrices or vectors.
- The JavaScript data type that is closest to a matrix or a vector is the built-in "Array" object.
- You can either write your own functionality (like MV.js) to handle matrices and vector in JavaScript
- Or you can use one of the open source libraries that are available. The following are three libraries that you can find on the web and that are quite commonly used in WebGL application (included in the code under Common folder)

1. Sylvester
2. WebGL-mjs
3. glMatrix

Sylvester

- Sylvester was written by James Coglan and is general vector and matrix mathematics library for JavaScript.
- It's not written specifically for WebGL.

```
var u = Vector.create([1,2,3]);
var v = Vector.create([4,5,6]);
var s = u.add(v);
alert(s.inspect());        // s = [5, 7, 9]
var d = u.dot(v); // d = 1*4+2*5+3*6 = 32
alert(d);        // will alert 32
var c = u.cross(v);
alert(c.inspect()); // Will alert [-3, 6, -3]
var M = Matrix.create([[ 2, -1],   // first row
                       [-2,  1],   // second row
                       [-1,  2]]); // third row
var N = Matrix.create([[4, -3],    // first row
                       [3,  5]]);  // second row
var MN = M.multiply(N);
alert(MN.inspect()); // will alert [ 5, -11]
                     //            [-5,  11]
                     //            [ 2,  13]
var I = Matrix.I(4); // create identity matrix
alert(I.inspect());  // will alert [1, 0, 0, 0]
                     //            [0, 1, 0, 0]
                     //            [0, 0, 1, 0]
                     //            [0, 0, 0, 1]
```

WebGL-mjs

- WebGL-mjs was written by Vladimir Vukicevic specifically for WebGL.

```
var u = V3.$(1,2,3);
 var v = V3.$(4,5,6);
 var s = V3.add(u,v);        // s = [5,7,9]
 alert(convertToString(s));  // will alert [5,7,9]
 var d = V3.dot(u,v); // d = 1*4+2*5+3*6=32
 alert(d);           // will alert 32
 var c = V3.cross(u,v);
 alert(convertToString(c)); //will alert [-3,6,-3]
 var I = M4x4.$(1,0,0,0,   // first column
                0,1,0,0,   // second column
                0,0,1,0,   // third column
                0,0,0,1); // fourth column
 var M = M4x4.$(1,0,0,0,   // first column
                0,1,0,0,   // second column
                0,0,1,0,   // third column
                2,3,4,1); // fourth column
 var IM = M4x4.mul(I,M);
 alert(convertToString(IM)); // will alert 1,0,0,0
                          //          0,1,0,0,
                          //          0,0,1,0,
                          //          2,3,4,1
 var T = M4x4.makeTranslate3(2,3,4);
 alert(convertToString(T));  // will alert 1,0,0,0
                          //          0,1,0,0,
                          //          0,0,1,0,
                          //          2,3,4,1
```

glMatrix

- The JavaScript library glmatrix was written by Brandon Jones.

```
var u = vec3.create([1,2,3]);
  var v = vec3.create([4,5,6]);
  var r = vec3.create();
  var t = vec3.create([1,2,3]);
var s = vec3.add(u,v,r);   // s = r = [5,7,9] as expected, u not changed
  alert(vec3.str(s));        // will alert [5,7,9]
  alert(vec3.str(r));        // will alert [5,7,9]
  alert(vec3.str(u));        // will alert [1,2,3]
var s2 = vec3.add(u,v)   // s2 = [5,7,9] as expected,
                        // but now also u = [5,7,9]
  alert(vec3.str(s2));     // will alert [5,7,9]
  alert(vec3.str(u));      // will alert [5,7,9]
  var d = vec3.dot(t,v);   // d = 1*4+2*5+3*6 = 32
  alert(d);                // will alert 32
  var c = vec3.cross(t,v,r);  // c = r = [-3,6,-3]
  alert(vec3.str(r));         // will alert [-3,6,-3]
  var I = mat4.create([1,0,0,0,    // first column
                       0,1,0,0,    // second column
                       0,0,1,0,    // third column
                       0,0,0,1]);  // fourth column
  var M = mat4.create([1,0,0,0,    // first column
                       0,1,0,0,    // second column
                       0,0,1,0,    // third column
                       2,3,4,1]);  // fourth column
 vvar IM = mat4.create();
  mat4.multiply(I, M, IM);
```

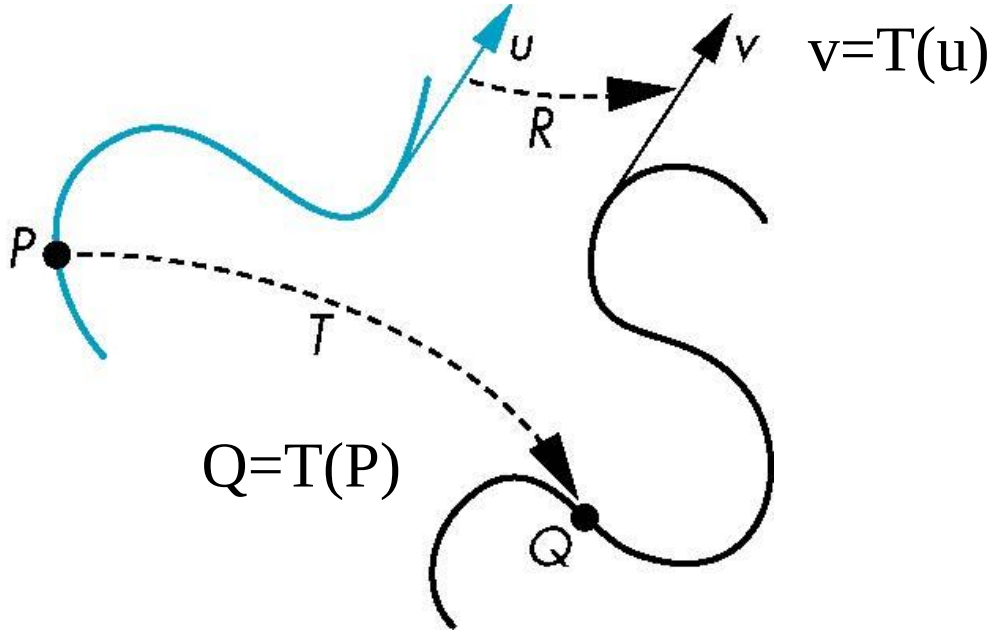
Transformations

Objectives

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

General Transformations

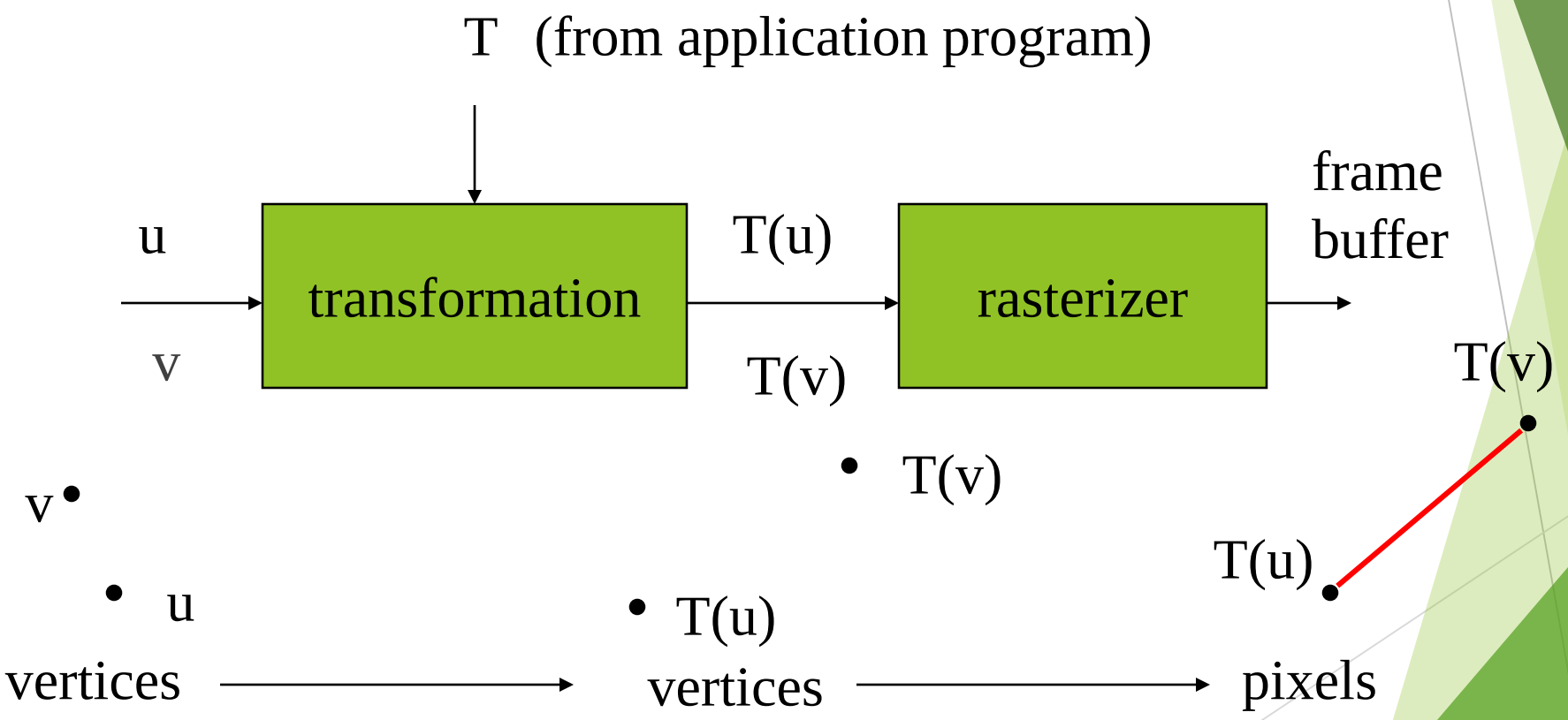
A transformation maps points to other points and/or vectors to other vectors



Affine Transformations

- Affine transform is a transform that performs a linear transformation and then a translation. If you use homogenous coordinates for the points or vectors, then you can achieve the transformation by multiplying 4x4 matrix transformation matrix and the column vector.
- MV=$\begin{bmatrix} m00 & m01 & m02 & m03 \\ m10 & m11 & m12 & m13 \\ m20 & m21 & m22 & m23 \\ m30 & m31 & m32 & m33 \end{bmatrix} \begin{bmatrix} v0 \\ v1 \\ v2 \\ v3 \end{bmatrix} = \begin{bmatrix} m00v0 + m01v1 + m02v2 + m03v3 \\ m10v0 + m11v1 + m12v2 + m13v3 \\ m20v1 + m21v1 + m22v2 + m23v3 \\ m30v1 + m31v1 + m32v2 + m33v3 \end{bmatrix}$
- Importance in graphics is that we need only transform endpoints of line segments and let implementation draw line segment between the transformed endpoints
- The four types of affine transforms are: rotation, translation, Scaling, Shearing

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

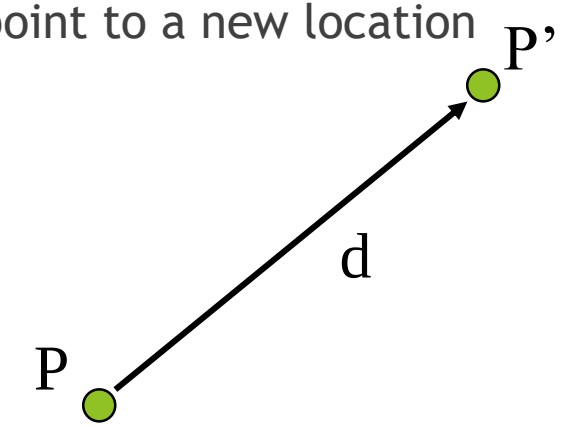
-array of 4 scalars in homogeneous coordinates

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

-array of 4 scalars in homogeneous coordinates

Translation

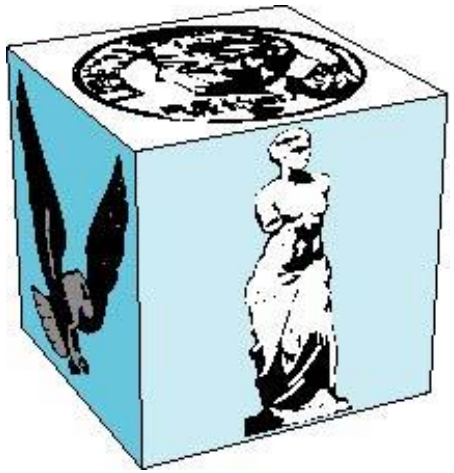
- Move (translate, displace) a point to a new location



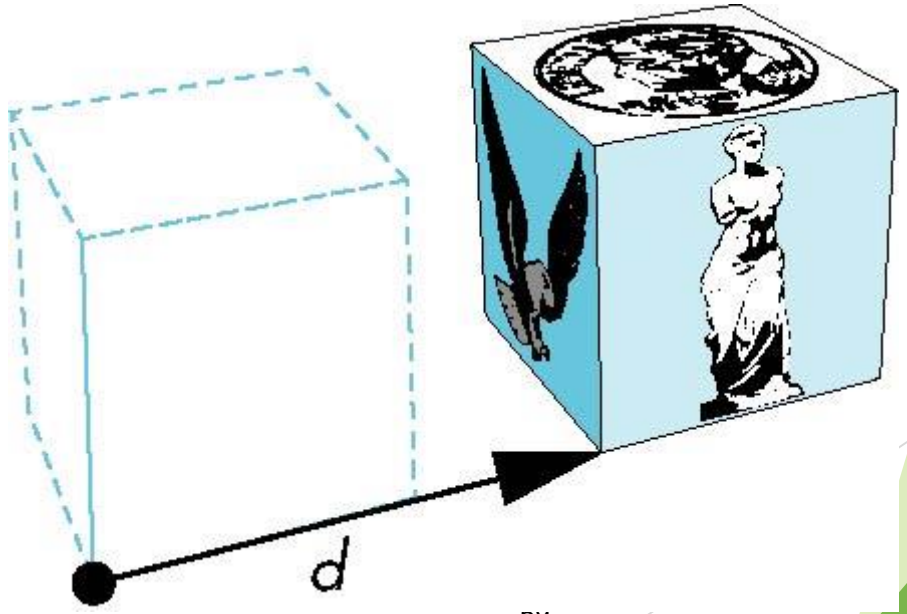
- Displacement determined by a vector d
  - Three degrees of freedom
  - P'=P+d

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

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

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

$\mathbf{d}=[dx\ dy\ dz\ 0]^T$

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 p is a single point on homogeneous notation

P=$\begin{bmatrix} Px \\ Py \\ Pz \\ 1 \end{bmatrix}$ and new point is translated by the vector (Tx,Ty,Tz)

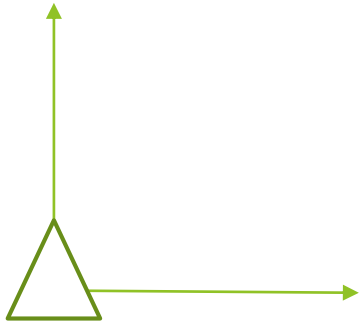
Tp=$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Px \\ Py \\ Pz \\ 1 \end{bmatrix} = \begin{bmatrix} Px + Tx \\ Py + Ty \\ Pz + Tz \\ 1 \end{bmatrix}$

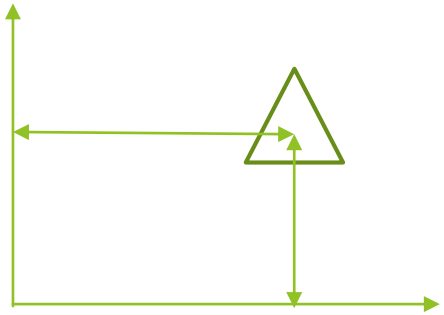
- Since a vector doesn't have a position, but just a direction and size, it should not be affected by translation matrix.

- Tv=$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Vx \\ Vy \\ Vz \\ 0 \end{bmatrix} = \begin{bmatrix} Vx \\ Vy \\ Vz \\ 0 \end{bmatrix}$

Translation Example

- To the left, triangle is shown before the translation is applied.





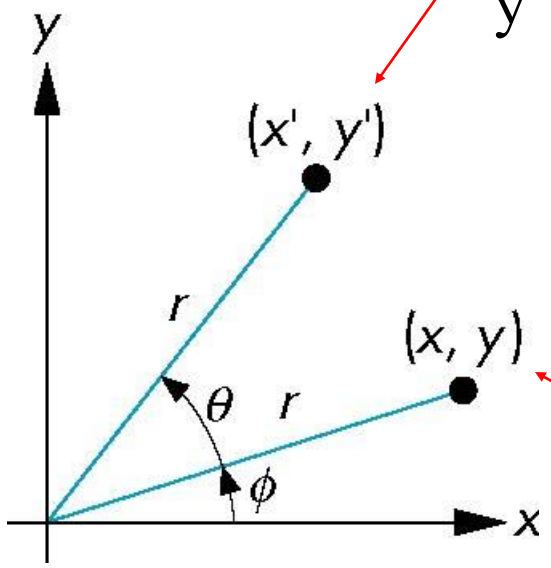
- $T(4,5,0) = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$
- To the right, the triangle is translated by 4 in the x-direction and 5 in y-direction. The z-axis is not shown but has a direction perpendicular out from the paper.

Rotation (2D)

Consider rotation about the origin by θ degrees

- radius stays the same, angle increases by θ

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



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

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

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

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

A positive rotation is given by the right-hand grip so that you take your right hand and grip around the axis you want to rotate around so the thumb is pointing in the positive direction of the axis. Then your other fingers are pointing in the direction that corresponds to the positive rotation.

Rotation Matrix

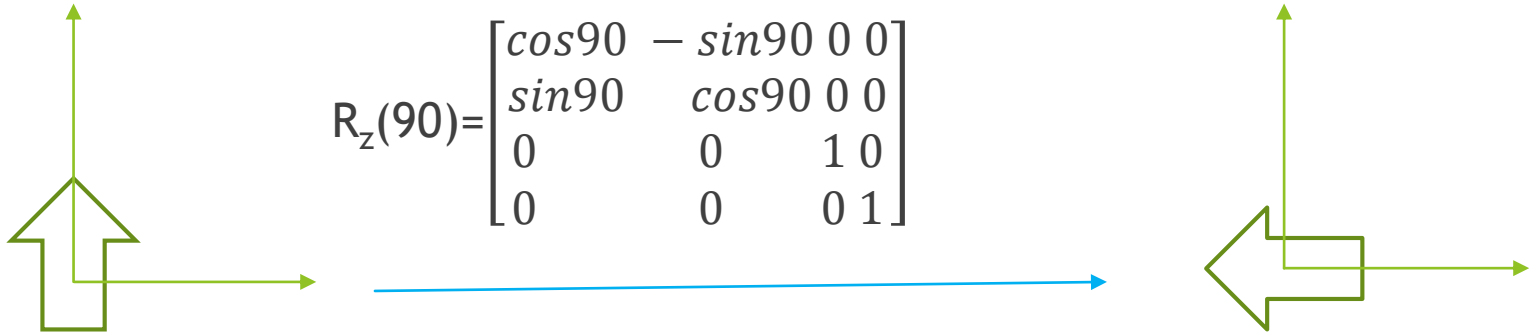
- A Rotation Matrix rotates a point or a vector by a given angle around a given axis that passes through the origin. Rotation matrices that are commonly used are R_x, R_y, and R_z , which are used for rotations around the x-axis, y-axis, and z-axis, respectively.

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

Example

- Example: An object that is rotated by 90 degrees around the z-axis by applying the rotation matrix R_z.(CounterClockWise)

- $$R_z(90)=\begin{bmatrix} cos90 & -sin90 & 0 & 0 \\ sin90 & cos90 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



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

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

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

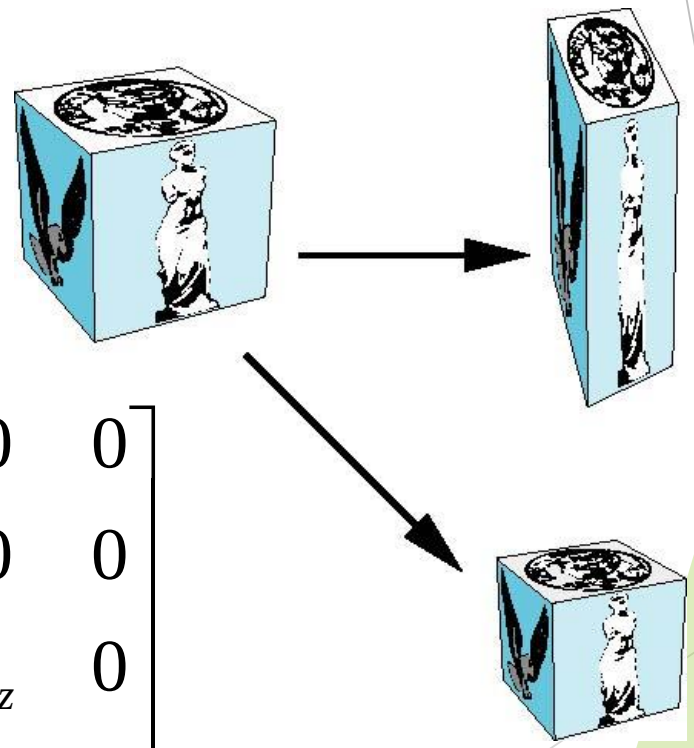
Scaling

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

Scaling is used to enlarge or diminish an object.

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

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



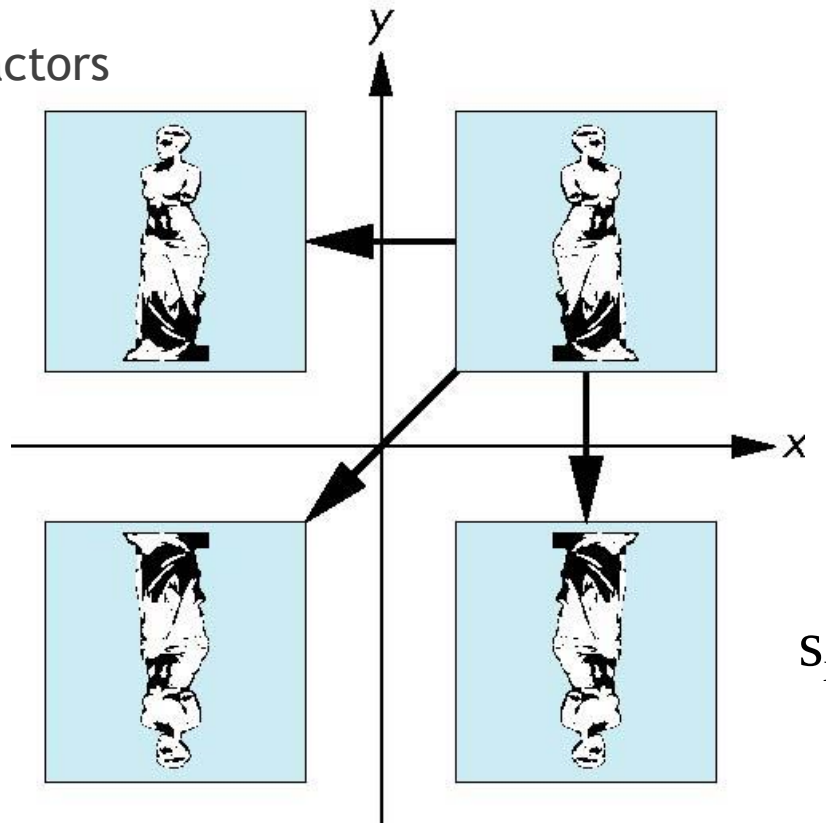
$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling Example

- The scaling matrix scales objects with a factor S_x along the x-direction, S_y along the y direction, and S_z along the z-direction.
- If the $S_x=S_y=S_z$, then the transform is called uniform scaling. A uniform scaling changes the size of objects, but doesn't change the shape of the objects.
- To the left, you see a square before the scaling is applied. To the right, you see the square after scaling matrix S(2,1,1) is applied. The square has now been scaled into a rectangle.
- $S(2,1,1)=\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Reflection

corresponds to negative scale factors



$s_x = -1\ s_y = 1$

original

$s_x = -1\ s_y = -1$

$s_x = 1\ s_y = -1$

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

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

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

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

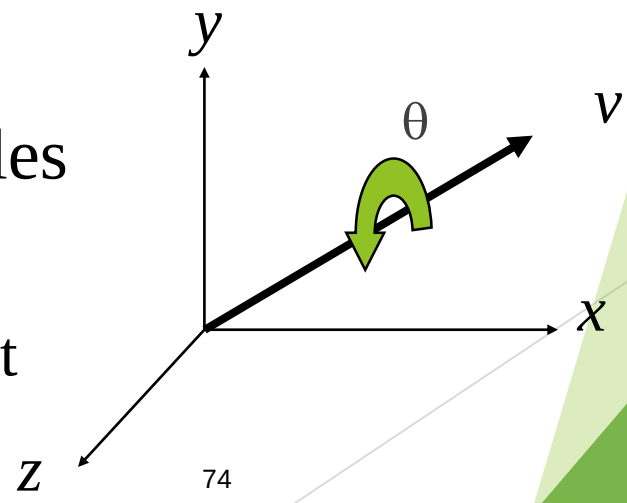
General Rotation About the Origin

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



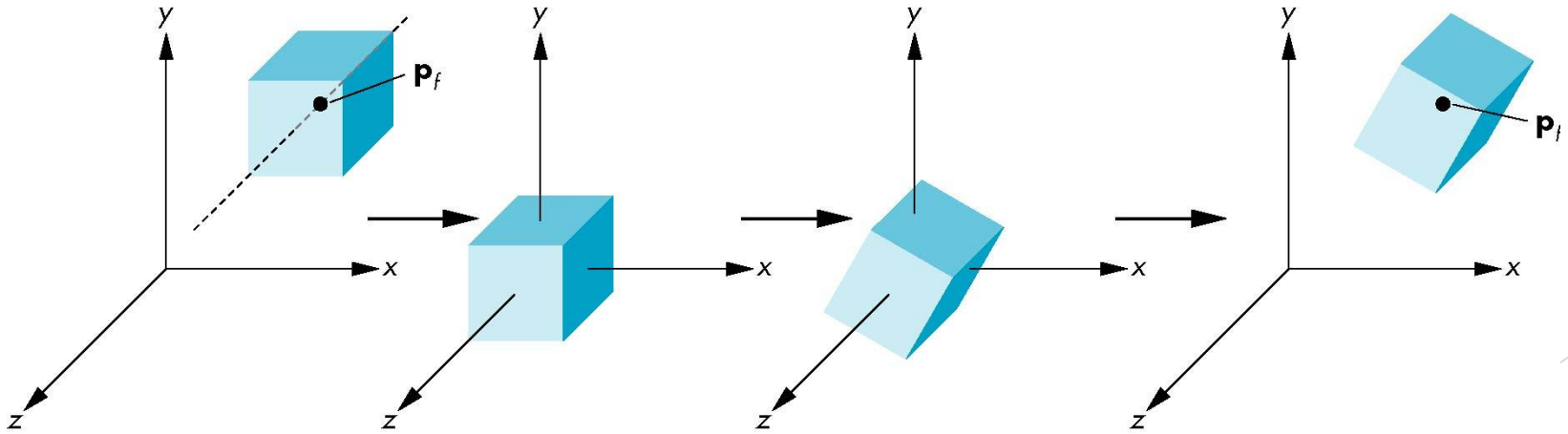
Rotation About a Fixed Point other than the Origin

Move fixed point to origin

Rotate

Move fixed point back

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



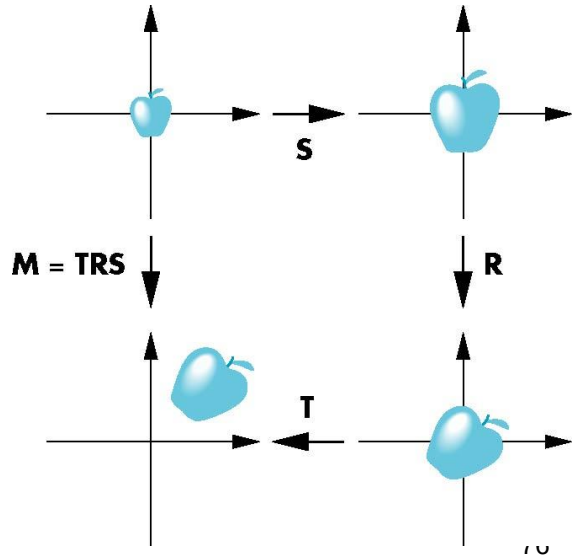
Instancing

- 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

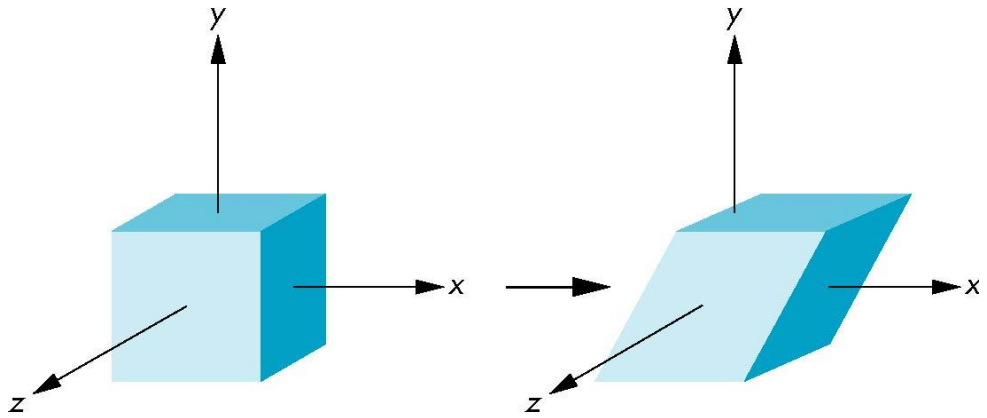
Orient

Locate



Shear

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



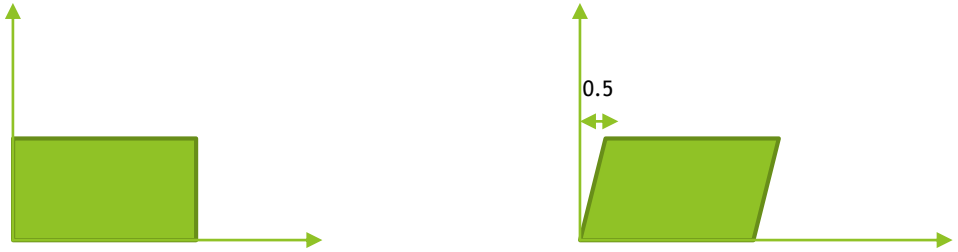
Shearing matrix

- This shearing matrix changes the x-coordinate when y-coordinate is changed. If you multiply the shearing matrix $H_{xy}(s)$ with a point p, it shows how the shearing matrix affects the point.

- $$H_{xy}(s)=\begin{bmatrix}1 & s & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}Px\\ Py\\ Pz\\ 1\end{bmatrix}=\begin{bmatrix}Px+sPy\\ Py\\ Pz\\ 1\end{bmatrix}$$

Shearing matrix

- You obtain a shearing matrix if you start from the identity matrix and then change one of the zeroes in the top-left 3x3 corner to a value that is not zero. This means are there are 6 possible basic shearing matrices $H_{zy}(s)$, $H_{xy}(s)$, $H_{xz}(s)$, $H_{yz}(s)$, $H_{yx}(s)$, $H_{zx}(s)$. In this case, the first subscript indicates which coordinate is changed by the matrix and the second subscript indicates which coordinate performs the shearing.
- $H_{xy}(0.5)=\begin{bmatrix} 1 & 0.5 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

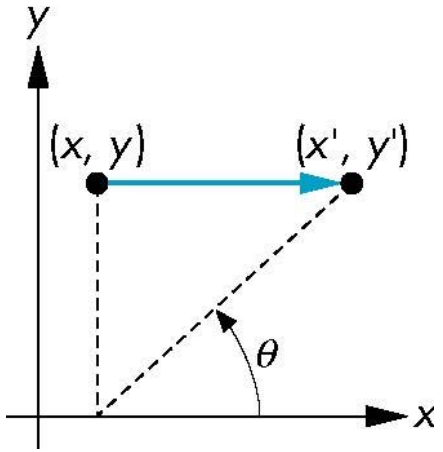


Shear Matrix

Consider simple shear along *x* axis

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

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



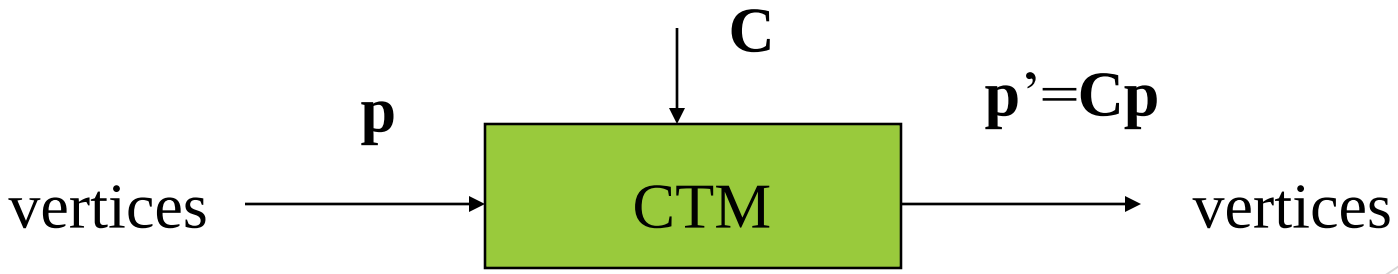
WebGL Transformations

Objectives

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

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: **C** ← **I**
Load an arbitrary matrix: **C** ← **M**

Load a translation matrix: **C** ← **T**
Load a rotation matrix: **C** ← **R**
Load a scaling matrix: **C** ← **S**

Postmultiply by an arbitrary matrix: **C** ← **CM**
Postmultiply by a translation matrix: **C** ← **CT**
Postmultiply by a rotation matrix: **C** ← **C R**
Postmultiply by a scaling matrix: **C** ← **C 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 postmultiplications.
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

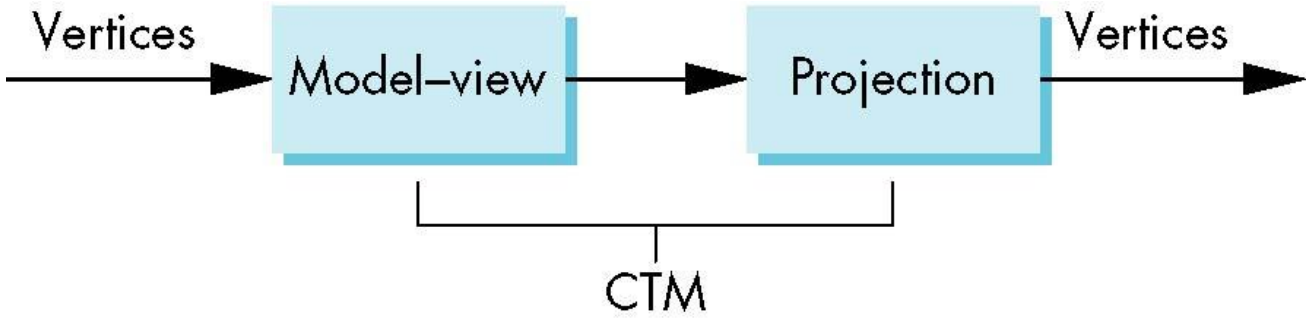
$\mathbf{C} \leftarrow \mathbf{I}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}^{-1}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{R}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$

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

- 3D graphics is about to compare the transforms with using a camera and taking a photo of some objects:
  1. Model Transform: Position your objects in the scene
  2. View Transform: Position and pointing your camera
  3. Projection Transform: Selecting a form for your camera
- 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

$$q = P*MV*p$$

Model Transformation

- The initial coordinates that the vertices for an object have when they are sent in through the WebGL API and stored in the WebGLBuffer, are called object coordinates.
- The model transformation is used to position and orient the model in a world coordinate system.
- When all models are transformed to the world coordinate system, they all exist in the same coordinate system.
- The model transformation is generally composed of a combination of the following transformations:
  - Translate()
  - Rotate()
  - Scale()

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
Do 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

View Transformation

- In a 3D scene, WebGL should only render objects that the camera (Viewer) sees.
- The view transformation is used to position and orient an imaginary camera.
- The camera is always located in the originlooking down the negative z-direction.
- The view transformation is usually composed of a combination of the following transformations:
  - Translate()
  - Rotate()
- After the view transformation has been applied, they are in eye coordinates.

Modelview Transformation

- The model transformation and the view transformation are actually the same thing when it comes to how the final scene will appear.
- Imagine that you have both an object and a camera located in the origin. In order to see that object:
  - Move the camera backwards
  - Move the object forward into the scene
- Example: you want to have 10 units between the camera and object
  - Translate(modelViewMatrix, [0,0,-10])
- The term modelview transformation means that the model and view transformations combined into a single matrix.
- Object Coordinates → Model Transformation → World Coordinate → View Transformation → Eye Coordinates

Projection Transformation

- The projection transformation is applied after your modelview transformation.
- Use it to determine how your 3D scene is projected on the screen.
- It takes your view volume and transform it into a unit cube.
- Two types of projection transform:
  - Orthographic Projection: Parallel Projection. Parallel lines remain parallel after the projection. All objects keep their relative size.

  Mat4.ortho(left,right,bottom,top,near,far, projectionMatrix) creates a projection matrix that has a rectangular view volume.

  - Perspective Projection: Objects that are far away from the camera look smaller than objects that are close to camera. This type of projection often gives you a more realistic scene than orthographic projection since it's close to how your eyes work.

  Mat4.perspective(fovy,aspect,near,far,projectionMatrix)

  Mat4.frustum(left,right,bottom,top,near,far,projectionMatrix)

Perspective Division

- When the vertex shader writes the coordinates to the variable gl_position, it's done in clip coordinates that are still on homogeneous notation (x_c, y_c, z_c, w_c)
- During the primitive assembly, the vertices go through perspective division, which divides all coordinates with w_c to yield normalized device coordinates (NDC) (x_d, y_d, z_d)
- This step is not implemented as a matrix multiplication.
- It's normally not a step you can explicitly affect yourself.

Viewport Transformation

- This step is not implemented as a matrix multiplication.
- The viewport transformation is performed for you as part of the primitive assembly.
- You can affect how it's done by calling the following methods:
- gl.viewport(x,y,w,h)
  - It specifies the viewport by giving the lower-left coordinate of the viewport with (x,y) and width and height of the viewport with w and h.
- gl.depthRange(n,f)
  - It specifies the desired depth range. n is near plane and f is the far plane. Both n and f are clamped to lie with the range [0.0,1.0]
- Object Coordinates → ModelView Matrix → Eye Coordinates → Projetction Matrix → Clip Coorinates (written to gl_position) → Perpective Division → Normalized Device Coordinates → Viewport Transformation → Window Coordinates

Creating Transformation Matrices with glMatrix

- When you setup your buffers, you need to setup the transformation matrices and make sure that they are uploaded to the vertex shader before you can call gl.drawArrays or gl.drawElements

```
modelViewMatrix = mat4.create();

projectionMatrix = mat4.create();

 function draw() {

//setup the projection matrix with a vertical field of view of 60 degrees, the aspect ratio, a near plane 0.1 units in front of the view, and a far plane 100.0 units from the viewer

mat4.perspective(60, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, projectionMatrix);

mat4.identity(modelViewMatrix); //load the identity matrix into modelview

//set up the view transform: the viewer is located at (8,5,10), view direction is towards at origin, the up direction is the positive y-axis

mat4.lookAt([8,5,10],[0,0,0],[0,1,0],modelViewMatrix);

//adds a translation of 3.0 units in the positive y-direction

mat4.translate(modelViewMatrix, [0.0,3.0,0.0],modelViewMatrix);
```

Uploading the transformation matrices to the vertex shader in the GPU

- When you have created yout transformation matrices, you must upload them to the GPU.
- You do this by loading the 16 float values into a uniform with the method gl.uniformMatrix4fv(uniformMVMatrix,false,modelViewMatrix))
- 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 bit it must be set to false.
- flatten function converts to column major order which is required by WebGL functions

Performing Matrix Multiplications in the vertex shader

- Code snippet:

```
<script id="vertex-shader" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;

  varying vec4 vColor;

  void main() {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vColor = aVertexColor;
  }
</script>
```

ModelView 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();
```

- Example: Assume that you want to draw a table by using a cube that you scale to five cuboids (one for table top and one cuboid for each of the four legs). You could draw this table by performing a translation from the table's origin to where you want to have a table top positioned, then scale the cube to a cuboid that looks like a table leg, and draw it. Then you would translate to the position for the first table leg, scale the cube to a cuboid that looks like a table leg, and draw it.

Applying Transformations

Getting Practical with Transformation

1. Setup your WebGLBuffer objects that contain object coordinates for the object in your scene.
2. Before you call any drawing method, you create a modelview matrix and a projection matrix
3. Transformation matrices are loaded to vertex shader in GPU by setting gl.uniformMatrix4fv()
4. You draw your scene by calling gl.drawArrays or gl.drawElements()
5. Vertex Shader is executed for each vertex in the scene, which gets the input from the WebGLBuffer (object coordinates), it uses uniforms where the transformation matrices were uploaded and perform transformation by matrix multiplication.

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 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 trig functions hardwired in silicon

Rotation Event Listeners

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

Rotation Shader (cont)

```
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 );

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

Smooth Rotation

- From a practical standpoint, we are often want to use transformations to move and reorient an object smoothly
  - Problem: find a sequence of model-view matrices $\mathbf{M_0}, \mathbf{M_1}, \ldots\ldots, \mathbf{M_n}$ 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

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 than either

Quaternions

- Extension of imaginary numbers (i*i=-1) from two to three dimensions
- Requires one real and three imaginary components $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$

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

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