# 4. Geometric Objects and Transformations

# Outline

- Geometry
- Representation
- Homogeneous Coordinates
- Transformations
- WebGL Transformations
- Applying Transformations
- Building Models
- The Rotating Square
- Sample Programs

# 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

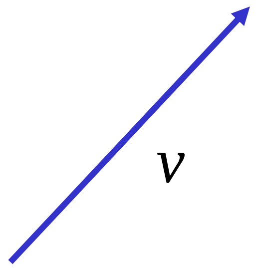
- When we learned simple geometry, most of us started with a Cartesian approach
  - Points were at locations in space **p**=(x,y,z)
  - We derived results by algebraic manipulations involving these coordinates
- This approach was nonphysical
  - Physically, points exist regardless of the location of 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

# 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

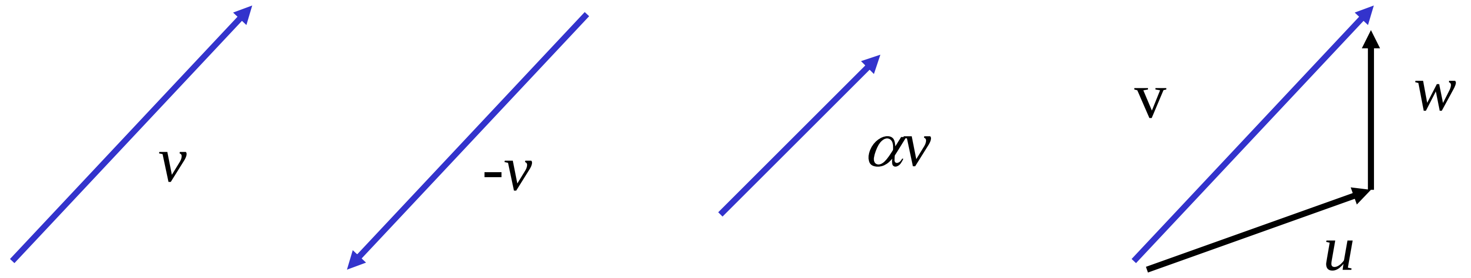
# Vectors

- 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

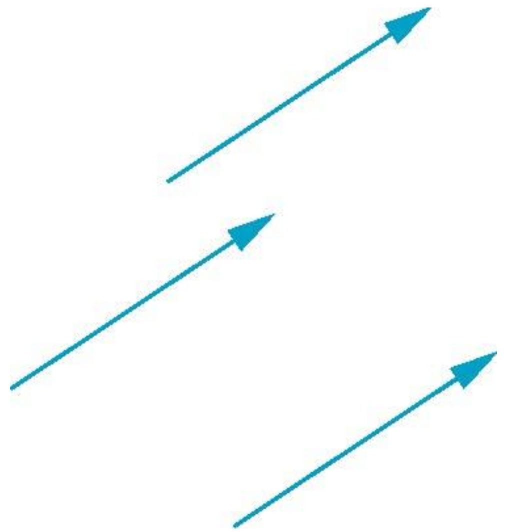
- Mathematical system for manipulating vectors
- Operations
  - Scalar-vector multiplication $u=\alpha v$
  - Vector-vector addition: $w=u+v$
- Expressions such as

  $v=u+2w-3r$

Make sense in a vector space

# Vectors Lack Position

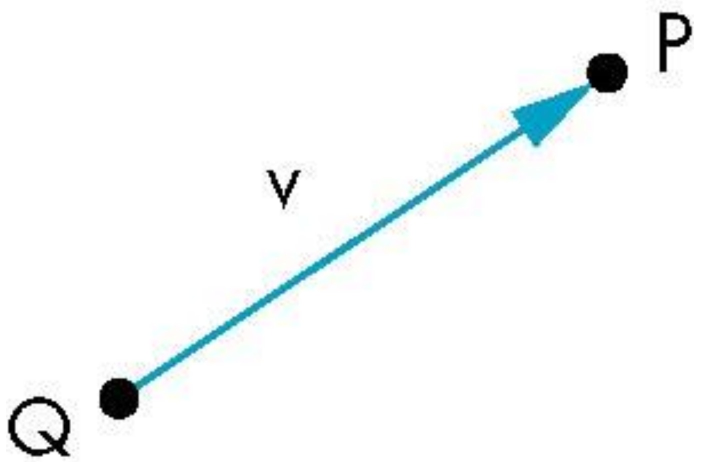
- These vectors are identical
  - Same length and magnitude



- 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




$$v = P - Q$$

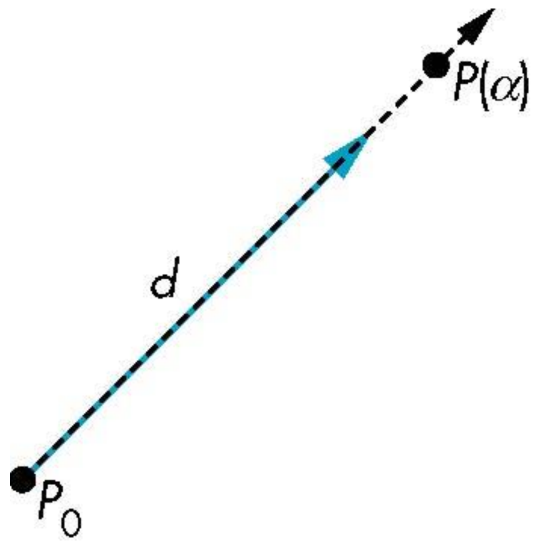
$$P = v + Q$$

# Affine Spaces

- Point + a vector space
- Operations
  - Vector-vector addition
  - Scalar-vector multiplication
  - Point-vector addition
  - Scalar-scalar operations
- For any point define
  - $1 \bullet P = P$
  - $0 \bullet P = \mathbf{0}$ (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

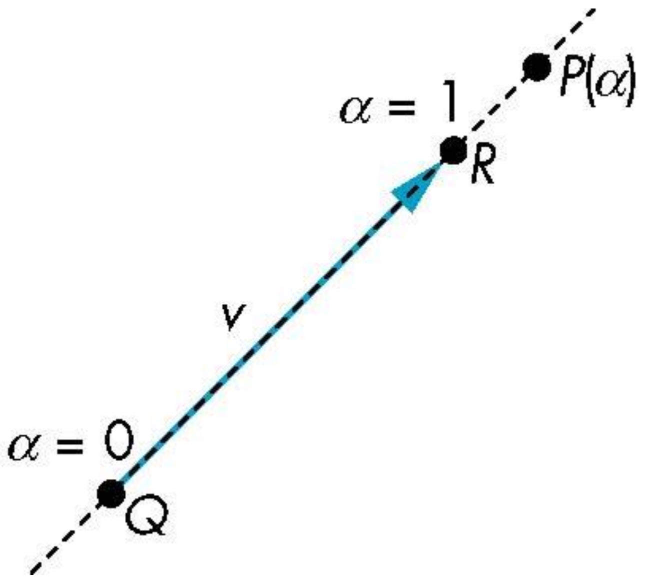
- If $\alpha >= 0$, then $P(\alpha)$ is the *ray* leaving $P_0$ in the direction $\mathbf{d}$

If we use two points to define $v$, then

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

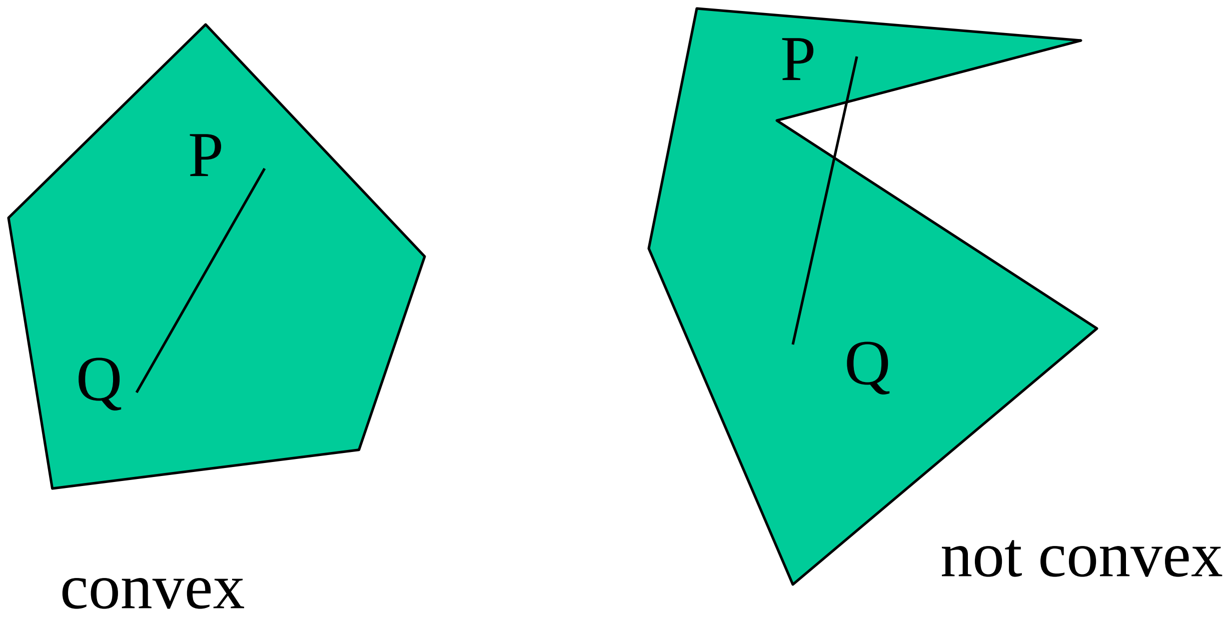
$= \alpha R + (1-\alpha)Q$

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



# Convexity

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



# 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 iff

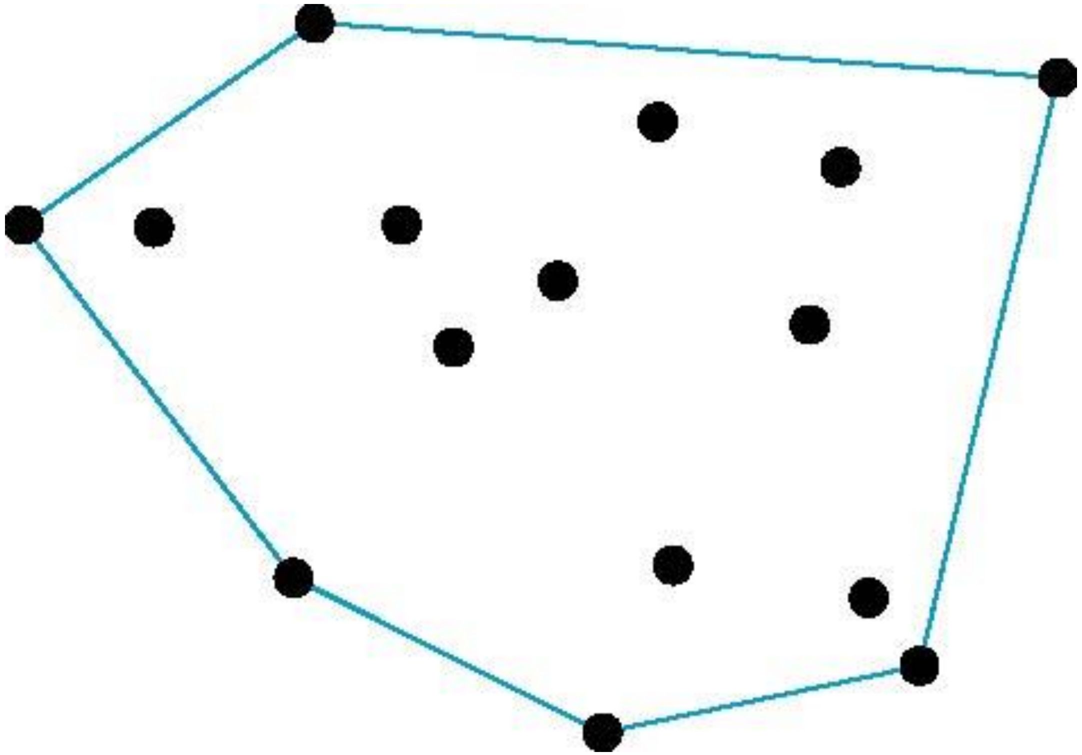
$\alpha_1+\alpha_2+\ldots..\alpha_n=1$

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 $P_1,P_2,\ldots..P_n$

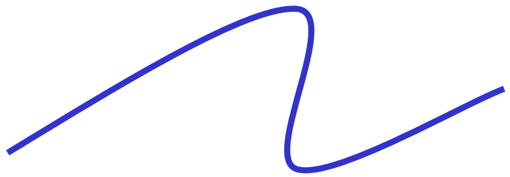
# Convex Hull

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

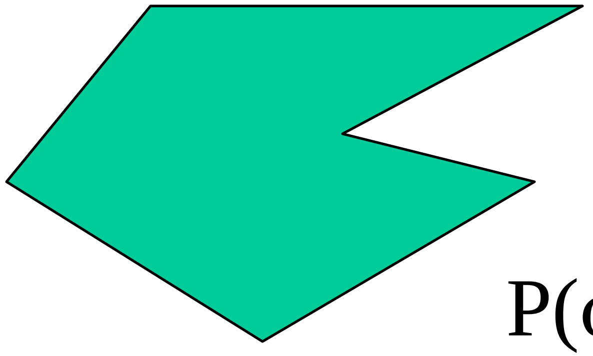


# Curves and Surfaces

- 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



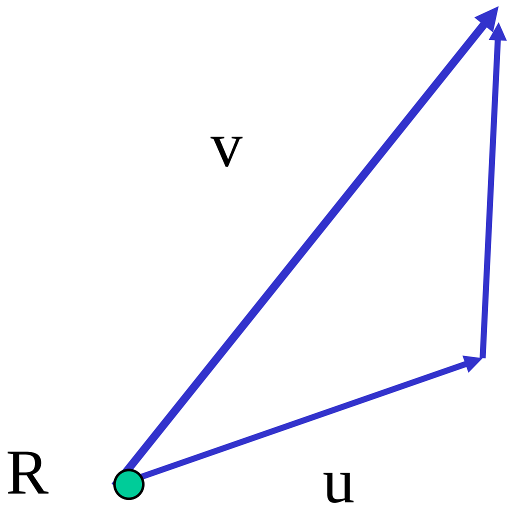
$P(\alpha)$



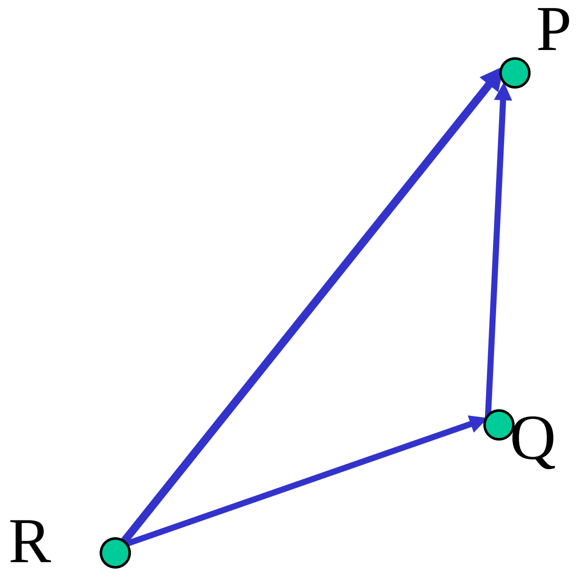
$P(\alpha, \beta)$

# Planes

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

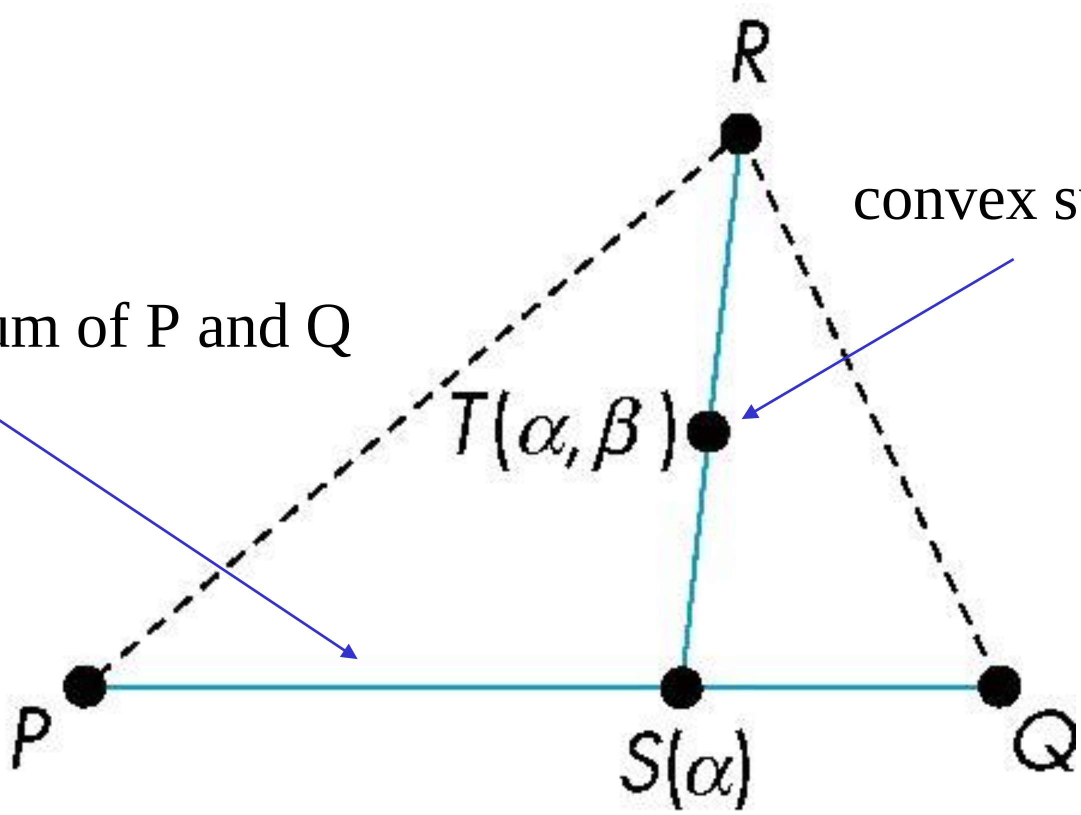



$$P(\alpha,\beta)=R+\alpha u+\beta v$$




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

# Triangles




convex sum of S(α) and R

convex sum of P and Q

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

$P(\alpha_{1,}\ \alpha_{2,}\ \alpha_3)=\alpha_1 P+\alpha_2 Q+\alpha_3 R$

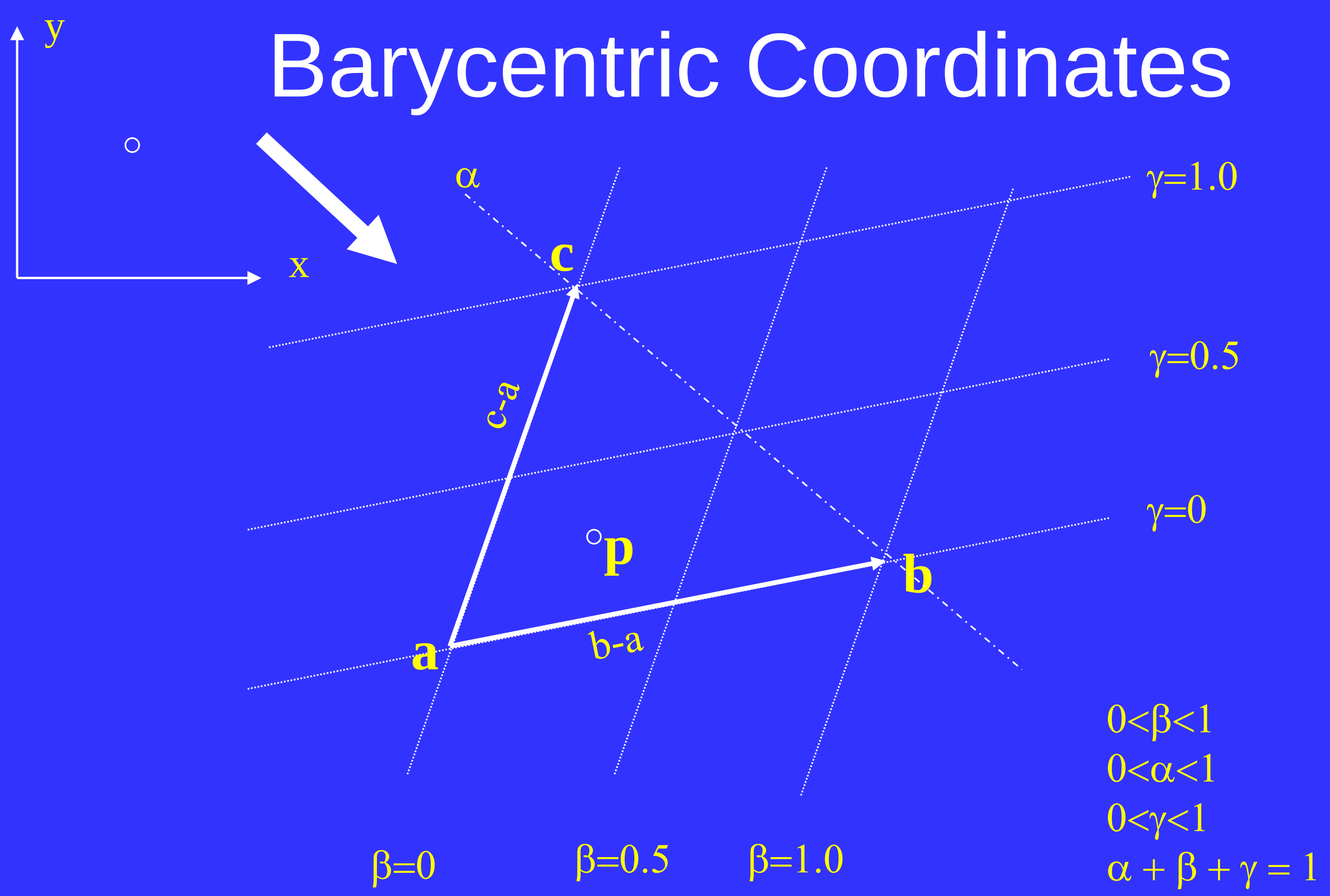
where

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

$\alpha_i >= 0$

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

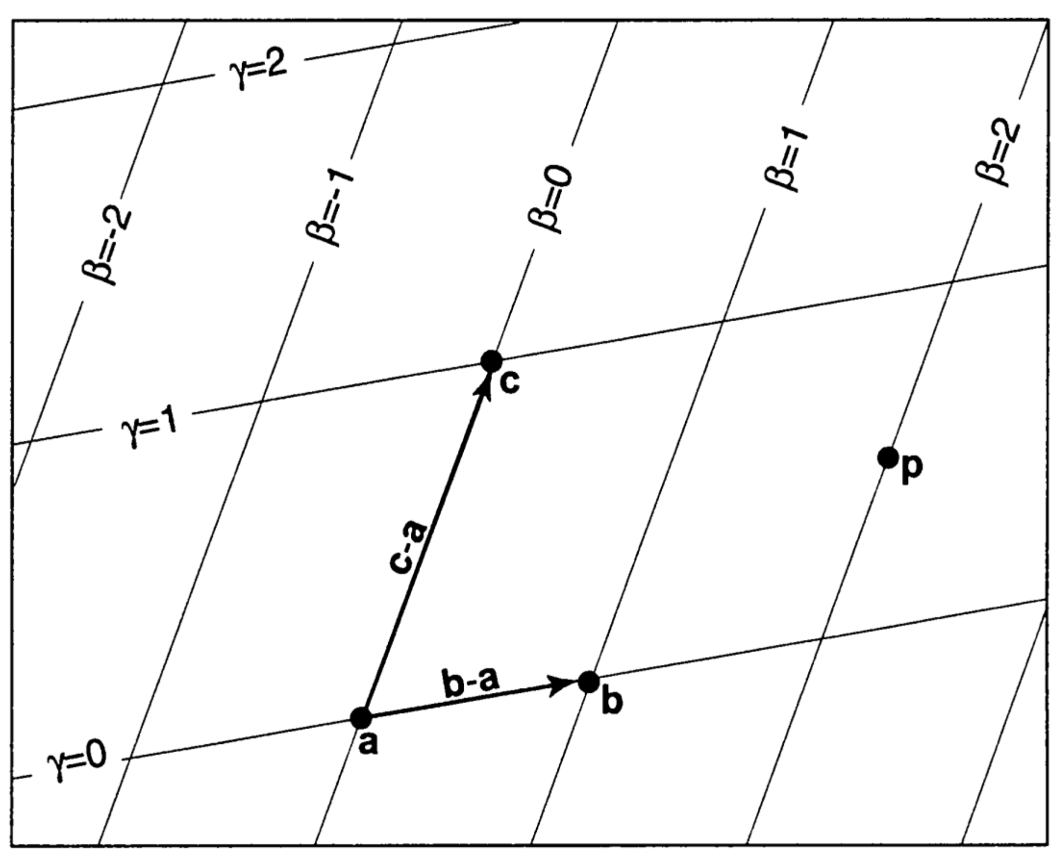
# Barycentric Coordinates




$$p = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$$

**Non-orthogonal coordinate system defined on the edges of the triangle**

# Barycentric Coordinates




For example, the point p = (2.0, 0.5), i.e., p = a + 2.0 (b- a) + 0.5 (c- a).

# Barycentric Coordinates

- Rearrange the terms

$$p = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$$

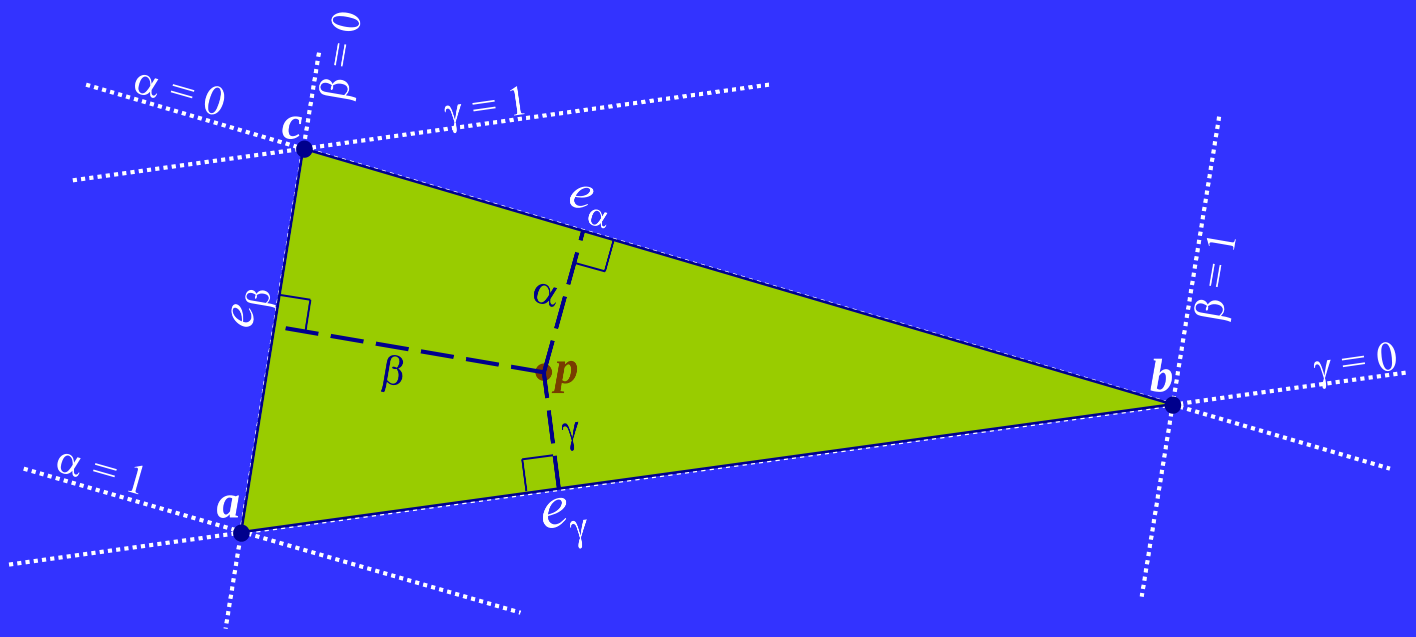
$$p = (1 - \beta - \gamma)\vec{a} + \beta\vec{b} + \gamma\vec{c}$$

**Let** $1 - \beta - \gamma \quad = \alpha$

$$p = \alpha\vec{a} + \beta\vec{b} + \gamma\vec{c}$$

$0<\beta<1$
$0<\alpha<1$
$0<\gamma<1$
$\alpha + \beta + \gamma = 1$

# Barycentric Coordinates




- Can determine points inside the triangle by computing $\alpha, \beta, \gamma$
- If all three values are > 0, inside the triangle
- For all points (inside and out): $\alpha + \beta + \gamma = 1$
- Can directly interpolate values across the triangle:

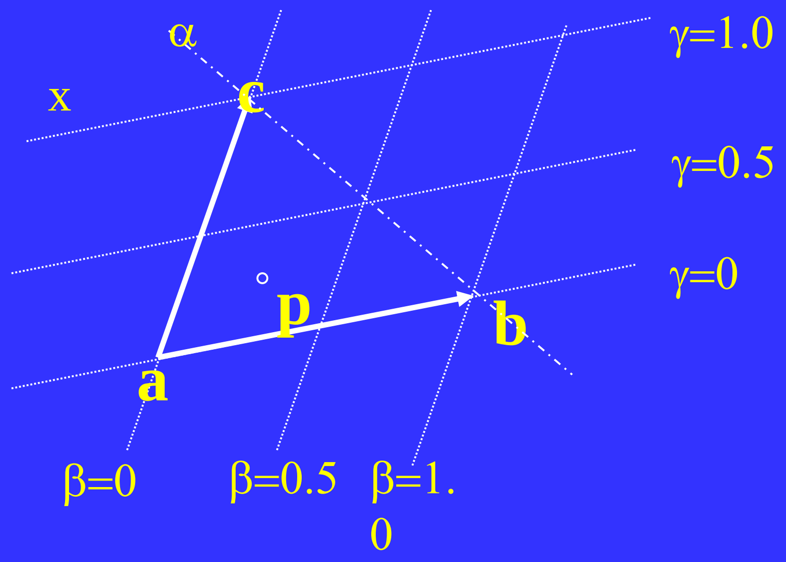
$$c_p = \alpha c_a + \beta c_b + \gamma c_c$$

# Barycentric Coordinates

- If for any point x,y we can compute the barycentric coordinates
  - We can determine if they are in the triangle if what?
  - We can also use them to interpolate colors or any values over the triangle.
  - if one coord = 0 and other two are >0 and < 1
    - on an edge
  - if two coords = 0, other is >0 and < 1,
    - at a vertex
- So, how do we compute these coordinates?

# Computing Barycentric Coordinates

- Consider the edges of the triangle as implicit lines
- Implicit lines give us signed, scaled , distances!




$$kf(x,y)=0$$

Like to choose k s.t.

$$kf(x,y)=\beta$$

At b, we know β = 1 therefore…

$$kf(x_b,y_b)=1$$

$$k=\frac{1}{f(x_{b,}y_b)}$$

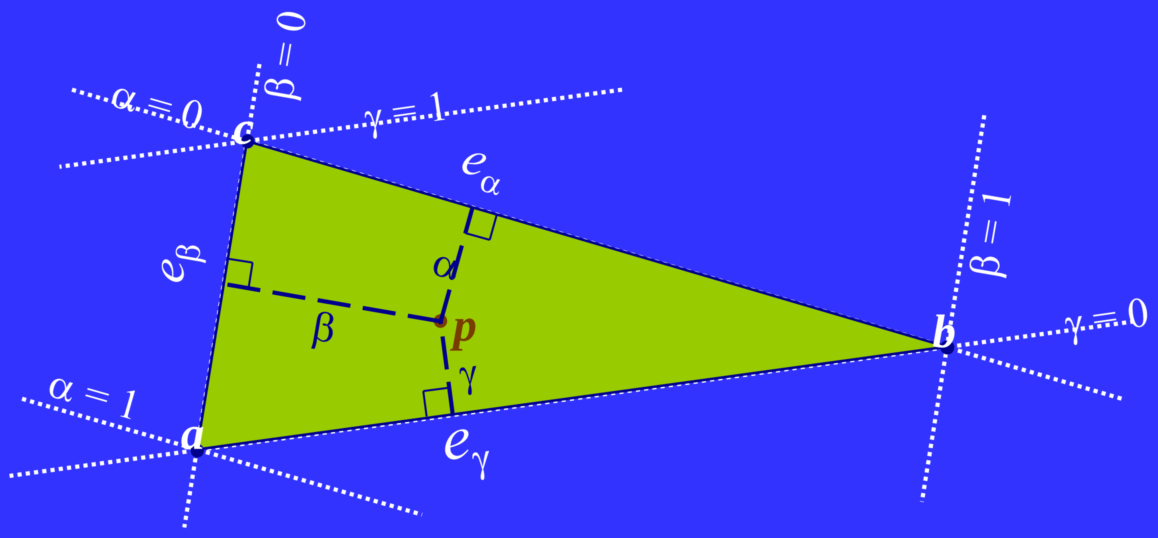
$$\beta=\frac{f_{ac}(x,y)}{f_{ac}(x_{b,}y_b)}$$

# Computing Barycentric Coordinates

- Where the implicit line equation is:

$$f_{ac}(x,y) = (y_a - y_c)x + (x_c - x_a)y + x_a y_c - x_c y_a$$

- Repeat this idea for each coordinate




$$\beta = e_\beta(p) = \frac{f_{ac}(p)}{f_{ac}(b)}$$

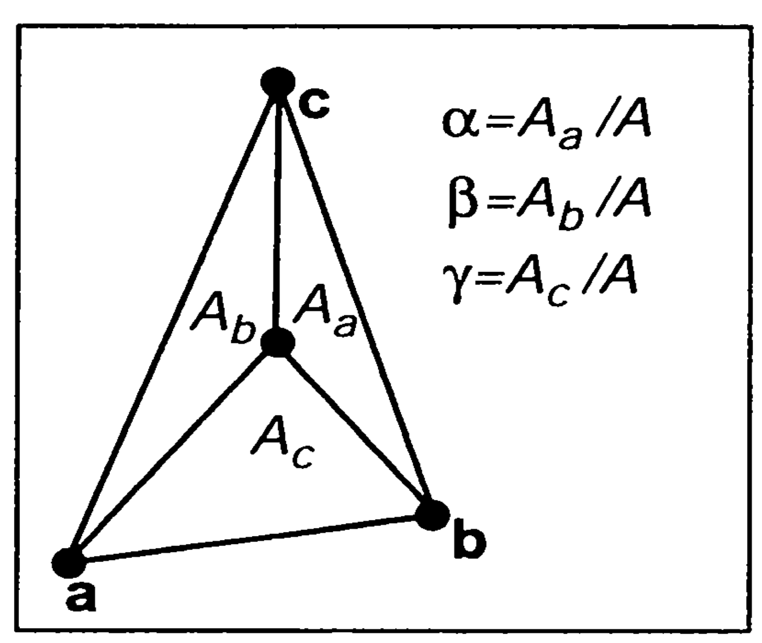
$$\alpha = e_\alpha(p) = \frac{f_{bc}(p)}{f_{bc}(a)}$$

$$\gamma = e_\gamma(p) = \frac{f_{ab}(p)}{f_{ab}(c)}$$

- Note: You actually only need to compute 2 of the 3

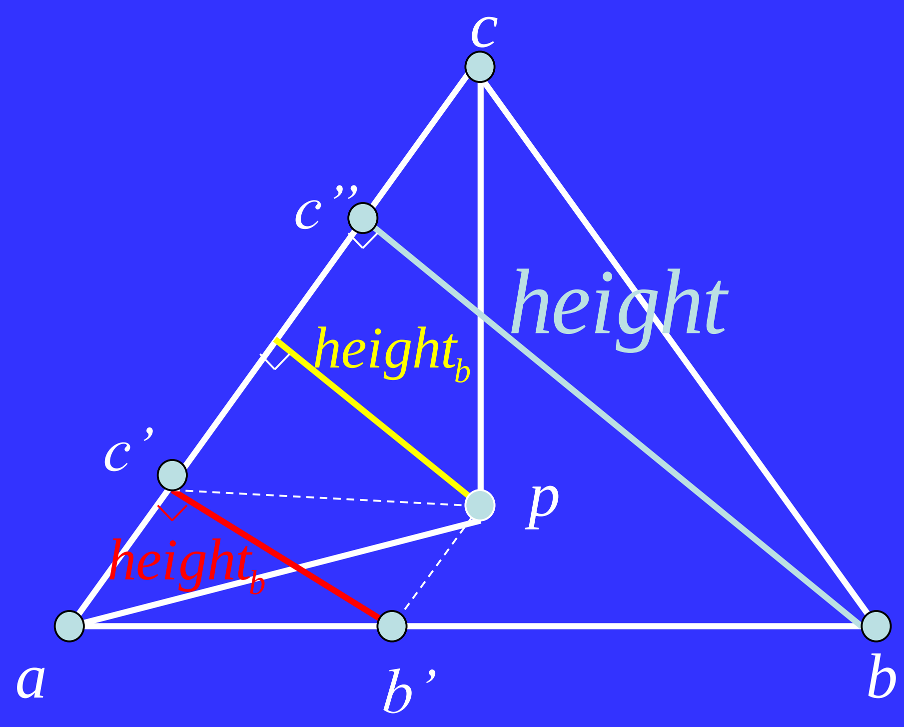
# Computing Barycentric Coordinates

The barycentric coordinates are proportional to the areas of the three subtriangles shown.




$$A = A_a + A_b + A_c$$

Show that $\frac{A_b}{A} = \frac{height_b}{height} = \beta$



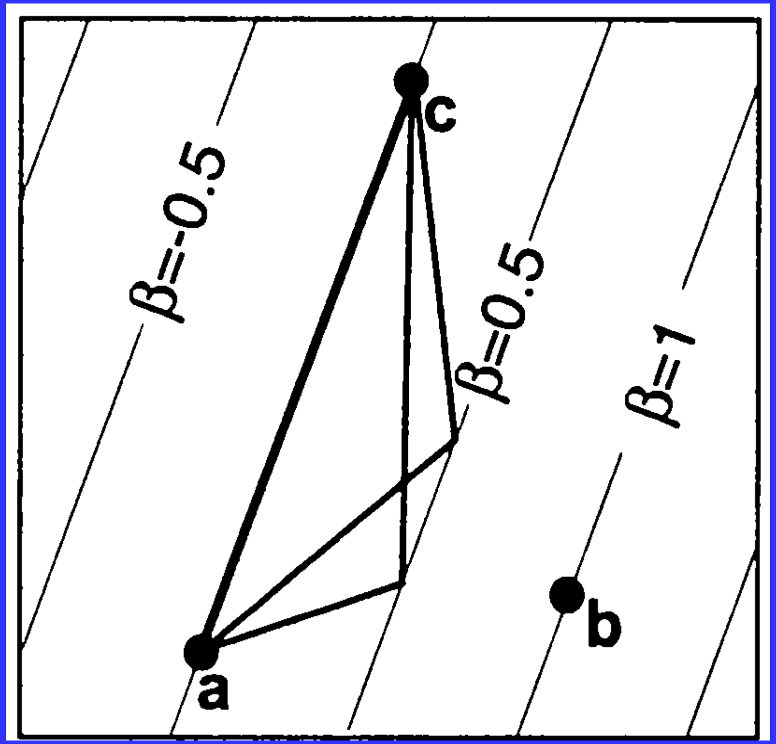

$a\Delta acp = A_b$

$a\Delta abc = A$

$\Delta ab'c' \cong \Delta abc''$

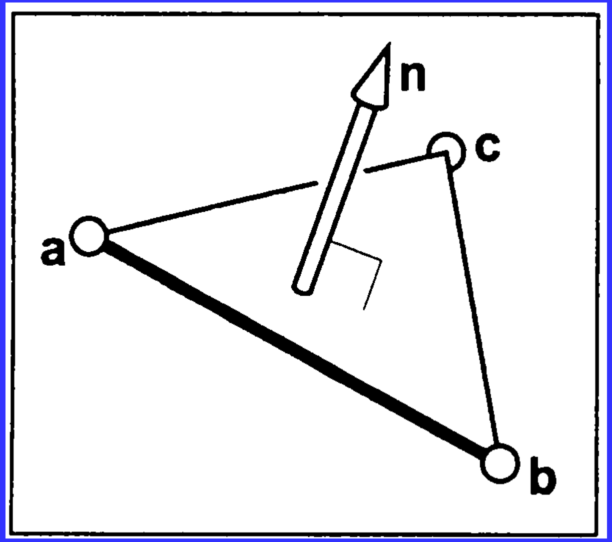
$$\therefore \frac{A_b}{A} = \frac{height_b}{height} = \frac{\ell(a,b')}{\ell(a,b)} = \beta$$

# Computing Barycentric Coordinates




The area of the two triangles shown is base times height and are thus the same, as is any triangle with a vertex on the β = 0.5 line. The height and thus the area is proportional to β.

# Computing Barycentric Coordinates (3D Triangles)



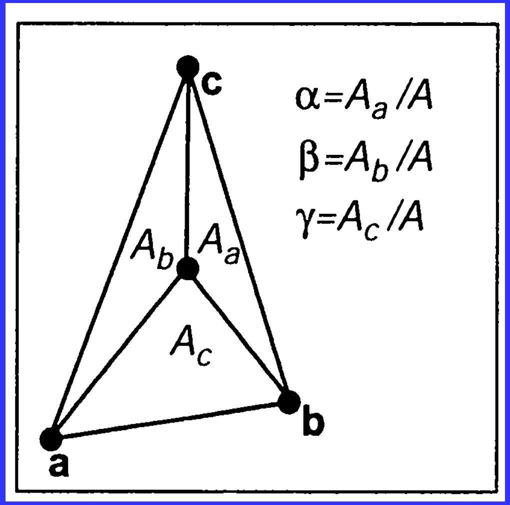

$$p = \alpha\vec{a} + \beta\vec{b} + \gamma\vec{c}$$

n = (b - a) x (c - a)

$$\text{area} = \frac{1}{2}\|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|$$

$$\alpha = \frac{\mathbf{n}\cdot\mathbf{n}_a}{\|\mathbf{n}\|^2} \quad \beta = \frac{\mathbf{n}\cdot\mathbf{n}_b}{\|\mathbf{n}\|^2} \quad \gamma = \frac{\mathbf{n}\cdot\mathbf{n}_c}{\|\mathbf{n}\|^2}$$

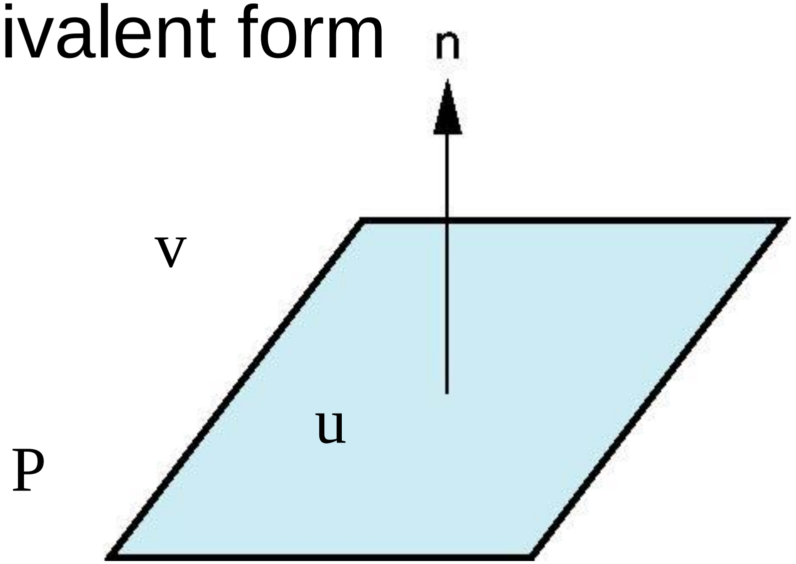
$$\mathbf{n}_a = (\mathbf{c}-\mathbf{b})\times(\mathbf{p}-\mathbf{b})$$
$$\mathbf{n}_b = (\mathbf{a}-\mathbf{c})\times(\mathbf{p}-\mathbf{c}),$$
$$\mathbf{n}_c = (\mathbf{b}-\mathbf{a})\times(\mathbf{p}-\mathbf{a})$$



# 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

# 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

  $$\alpha_1 v_1 + \alpha_2 v_2 + .. \alpha_n v_n = 0 \text{ iff } \alpha_1 = \alpha_2 = \ldots = 0$$

- 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

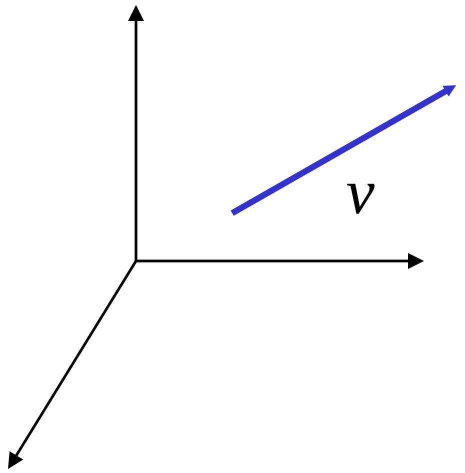
$$\mathbf{a} = [\alpha_1 \;\; \alpha_2 \;\; \ldots \;\; \alpha_n]^T = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ . \\ \alpha_n \end{bmatrix}$$

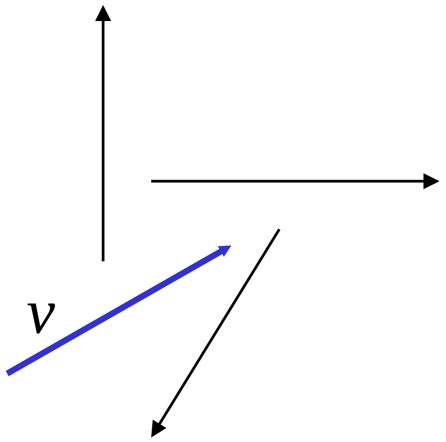
# Example

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

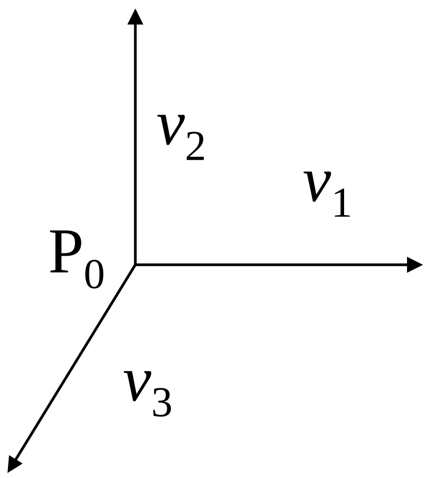



- Both are because vectors have no fixed location

# 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

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

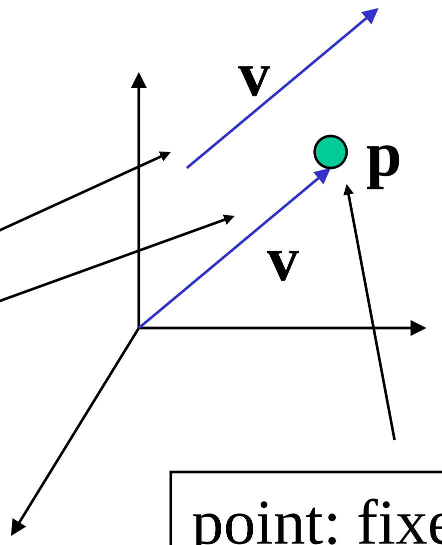
$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

They appear to have the similar representations

**p**$=[\beta_1\ \beta_2\ \beta_3]$ **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 $0 \bullet \mathrm{P} = \mathbf{0}$ and $1 \bullet \mathrm{P} = \mathrm{P}$ then we can write

$$\mathrm{v} = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1\ \alpha_2\ \alpha_3\ 0\,]\,[v_1\ v_2\ v_3\ \mathrm{P}_0]^{\mathrm{T}}$$

$$\mathrm{P} = \mathrm{P}_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [\beta_1\ \beta_2\ \beta_3\ 1\,]\,[v_1\ v_2\ v_3\ \mathrm{P}_0]^{\mathrm{T}}$$

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

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

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

# Homogeneous Coordinates

The homogeneous coordinates form for a three dimensional point [x y z] is given as

$\mathbf{p} = [x'\ y'\ z'\ w]^T = [wx\ wy\ wz\ w]^T$

We return to a three dimensional point (for $w \neq 0$) by

$x \leftarrow x'/w$

$y \leftarrow y'/w$

$z \leftarrow z'/w$

If $w=0$, the representation is that of a vector

Note that homogeneous coordinates replaces points in three dimensions by lines through the origin in four dimensions

For w=1, the representation of a point is [x y z 1]

# 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

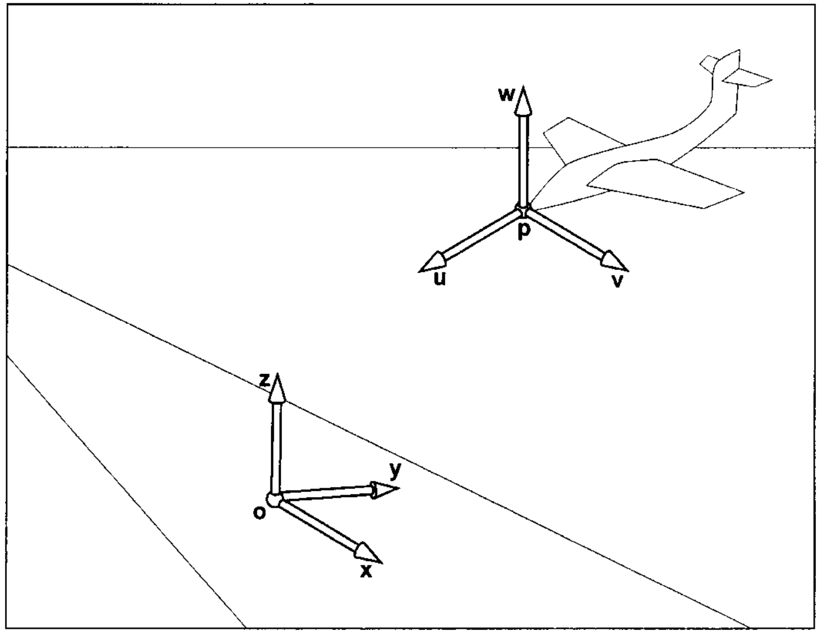
$$\mathbf{a}=[\alpha_1\ \alpha_2\ \ \alpha_3\ ]$$
$$\mathbf{b}=[\beta_1\ \ \beta_2\ \ \beta_3]$$

where

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

# Change of Coordinate Systems

A Flight Simulator



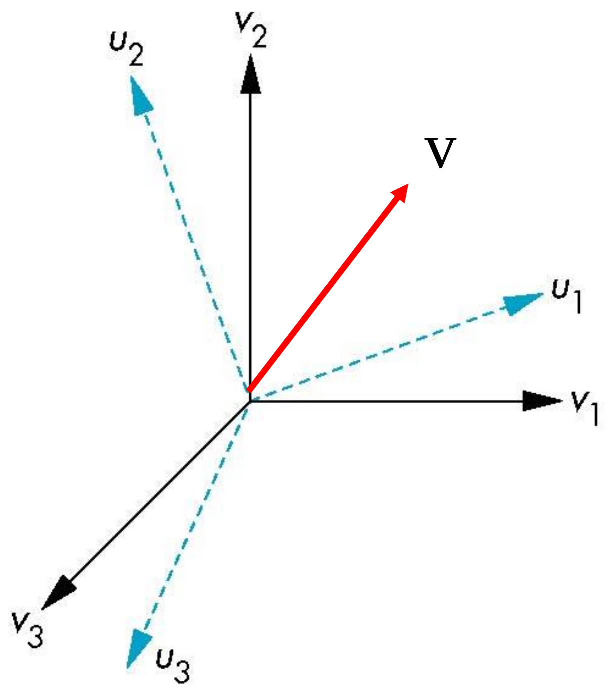

World coordinate system: xyz

Local coordinate system: uvw

# 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 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

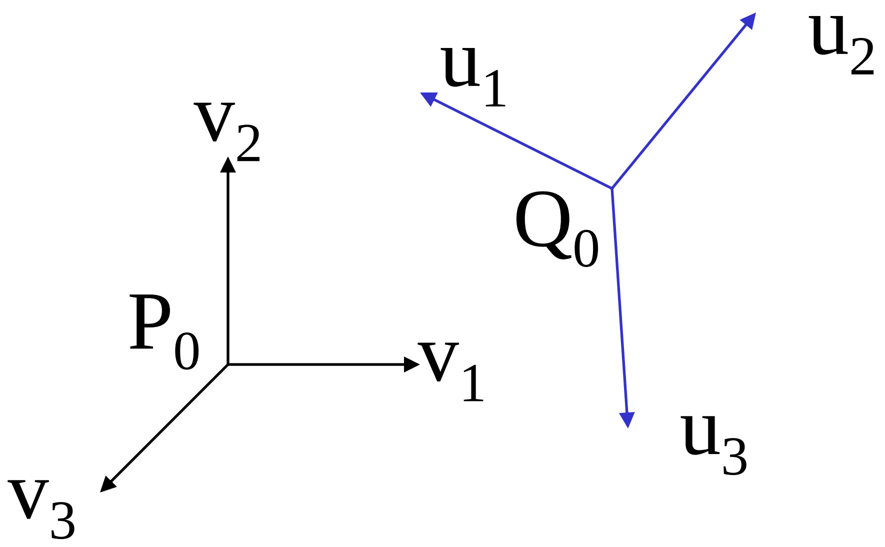
# Change of Frames

- 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

$\mathbf{a}=[\alpha_1\ \alpha_2\ \ \alpha_3\ \alpha_4\ ]$ 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

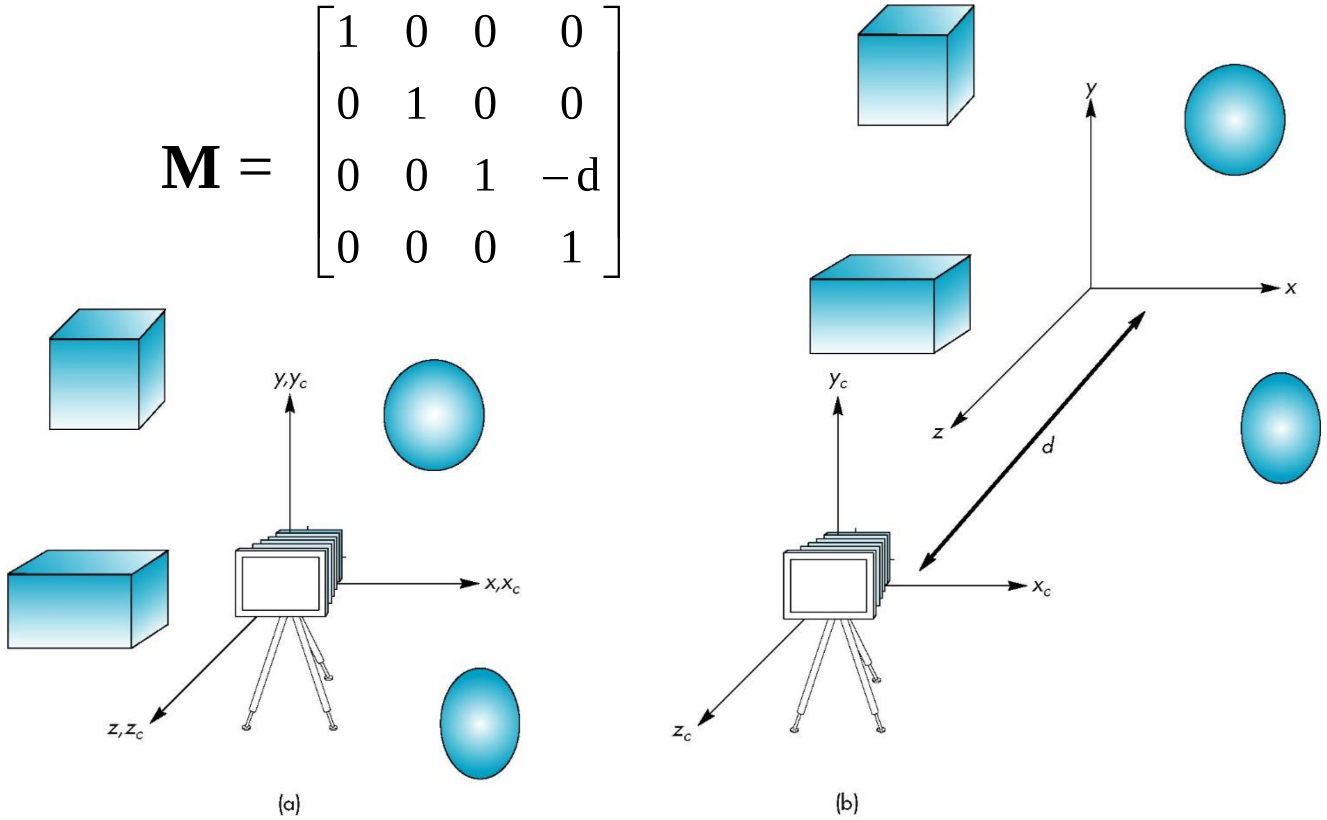
# The World and Camera Frames

- 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

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



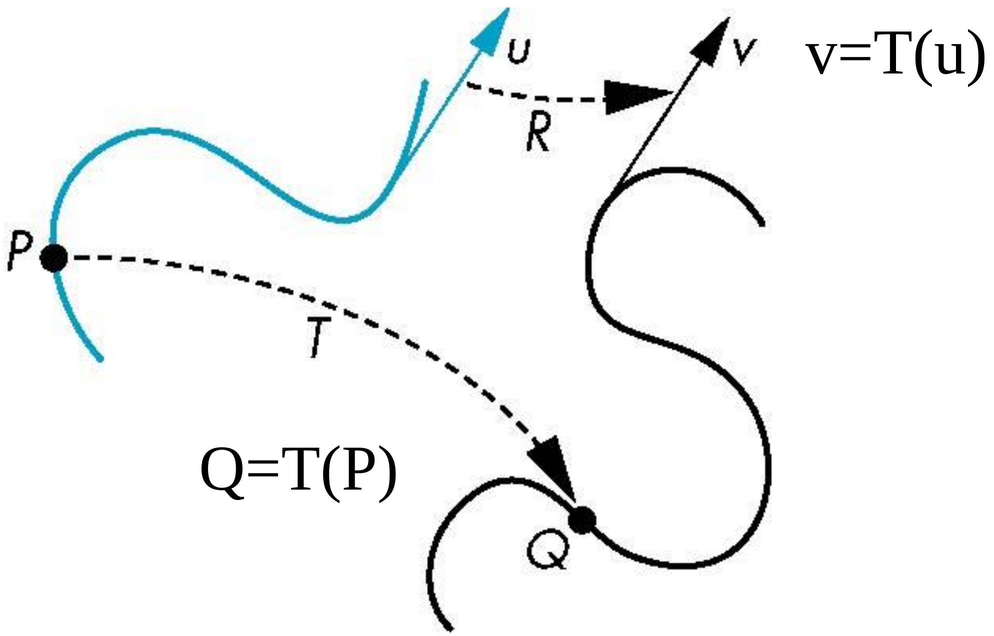
# Transformations

# Objectives

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

# General Transformations

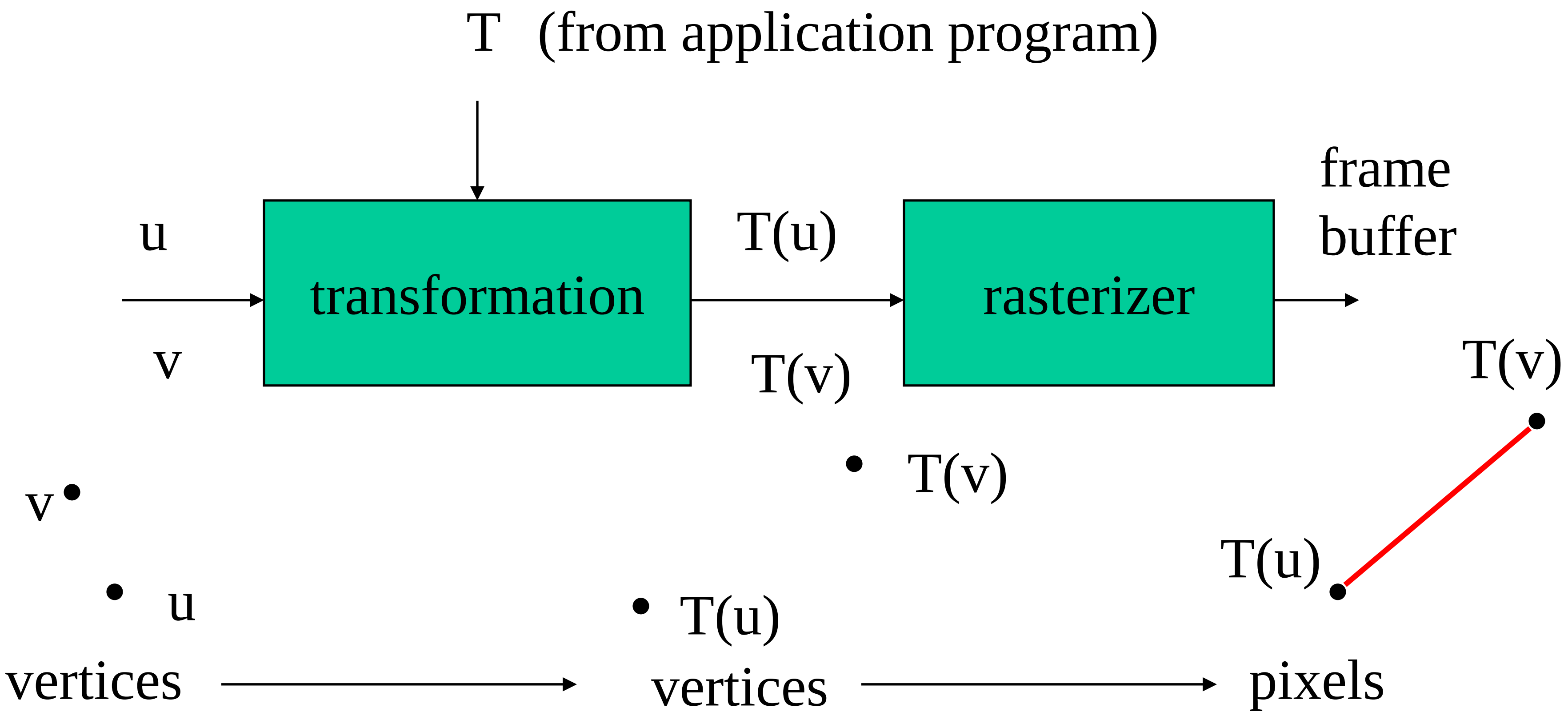
A transformation maps points to other points and/or vectors to other vectors



# 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

$\alpha$, $\beta$, $\gamma$: scalars

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

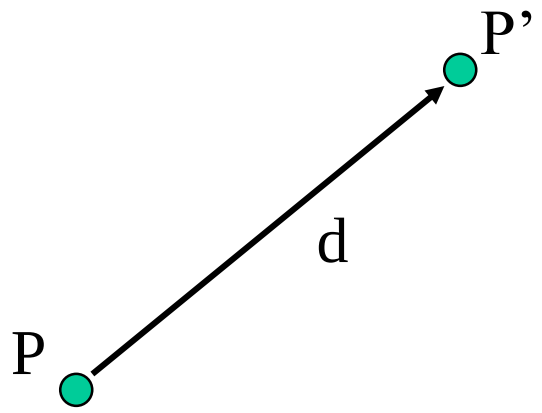
-array of 4 scalars in homogeneous coordinates

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

-array of 4 scalars in homogeneous coordinates

# Translation

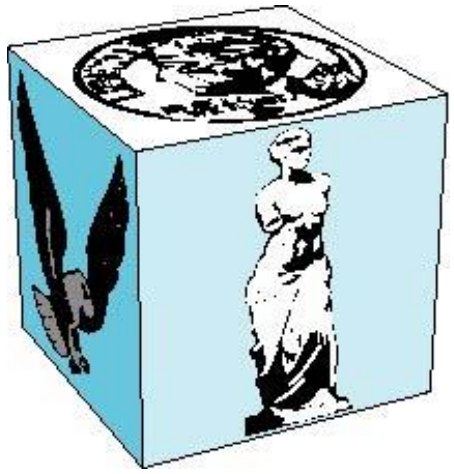
- Move (translate, displace) a point to a new location



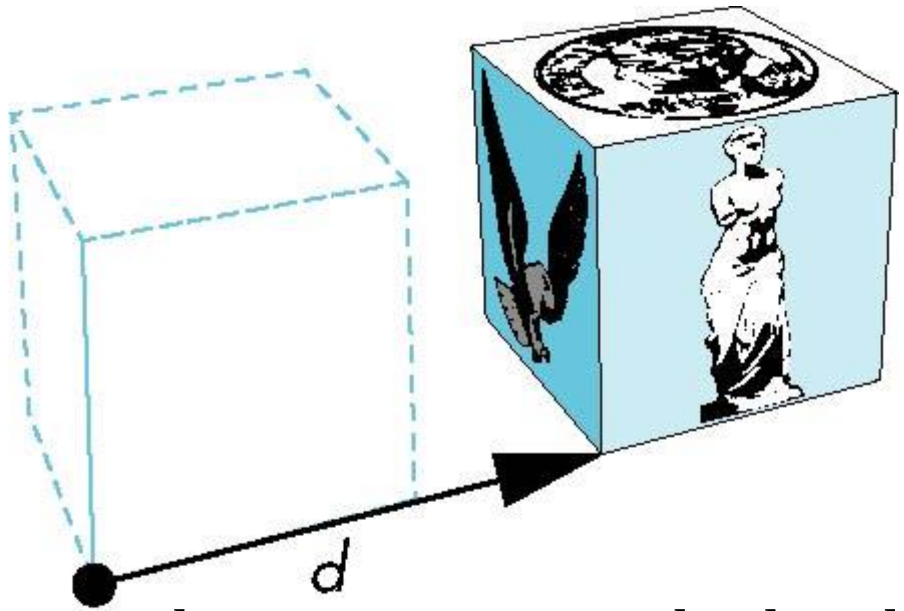

- Displacement determined by a vector d
  - Three degrees of freedom
  - P'=P+d

# 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

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

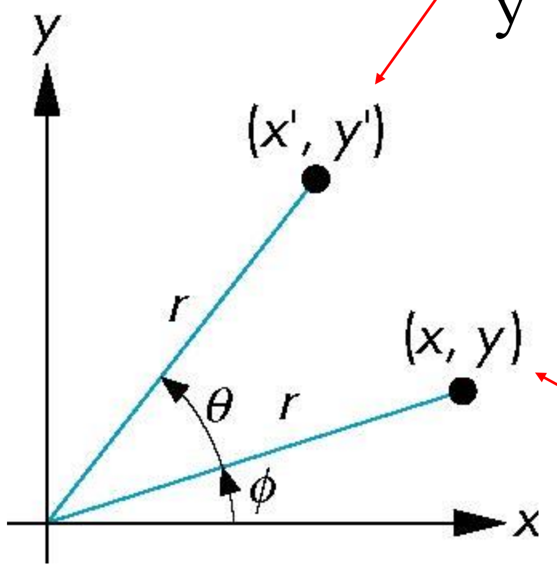
This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated together

# Rotation (2D)

Consider rotation about the origin by θ degrees

- radius stays the same, angle increases by $\theta$

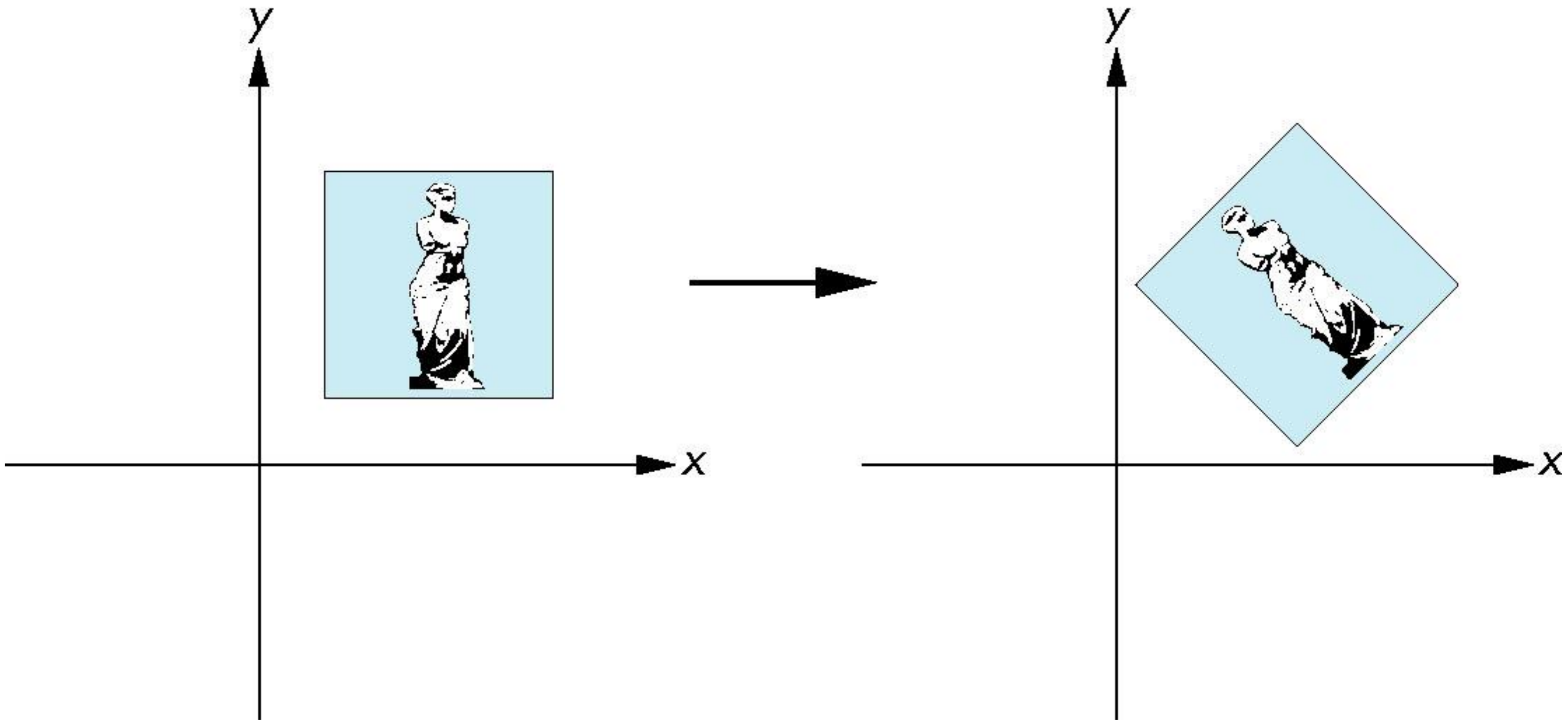
$x' = r \cos(\phi + \theta)$
$y' = r \sin(\phi + \theta)$



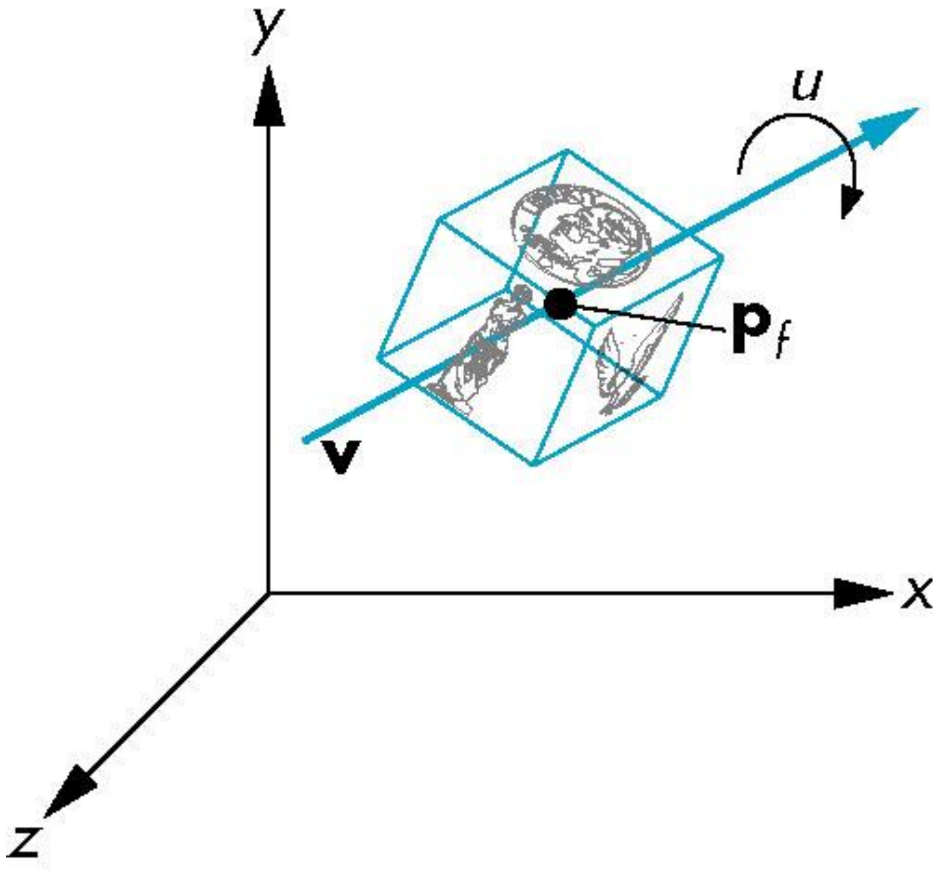

$$\boxed{\begin{aligned} x' &= x \cos\theta - y \sin\theta \\ y' &= x \sin\theta + y \cos\theta \end{aligned}}$$

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

# Rotation About a Fixed Point



# Three-dimensional Rotation



# 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

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

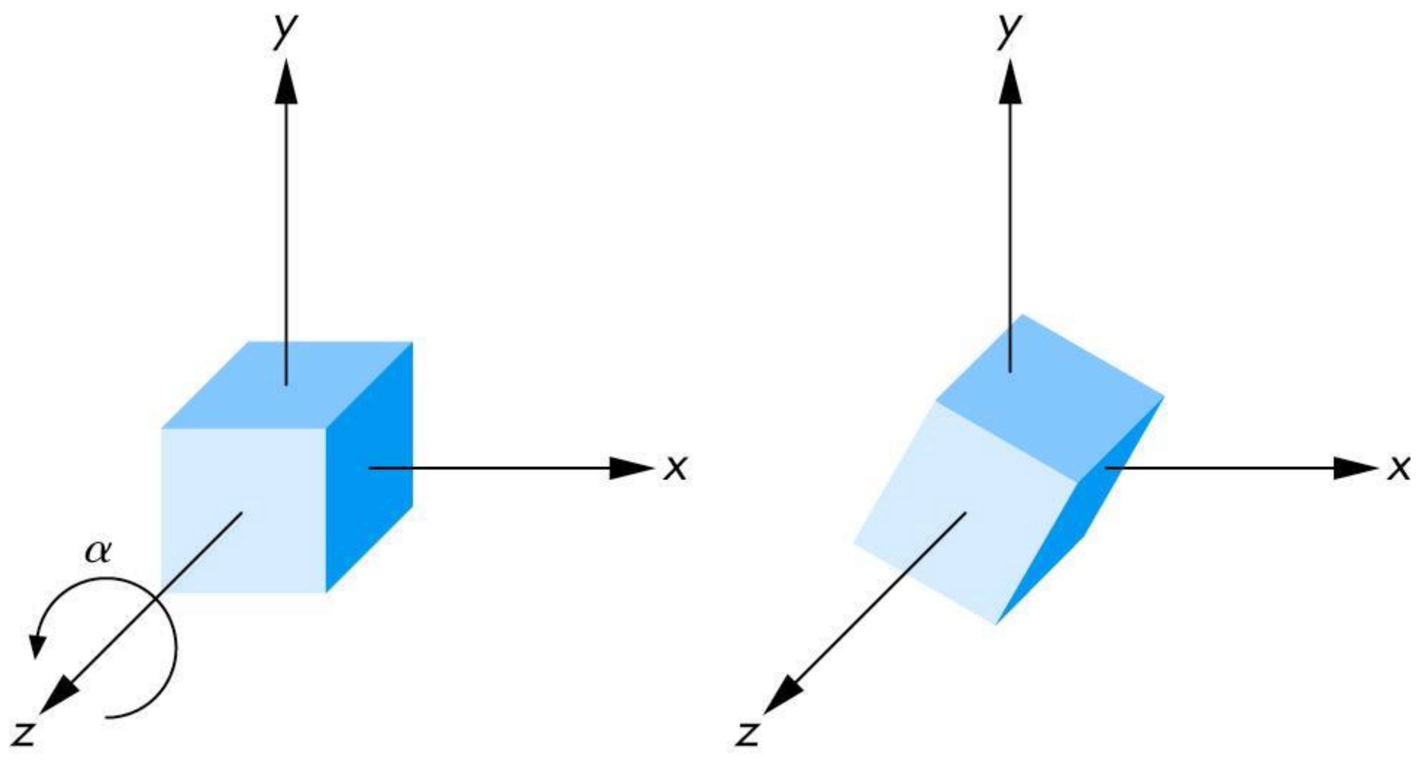
- or in homogeneous coordinates

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

# 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 of a cube about the z-axis




(a) Before rotation

(b) After rotation

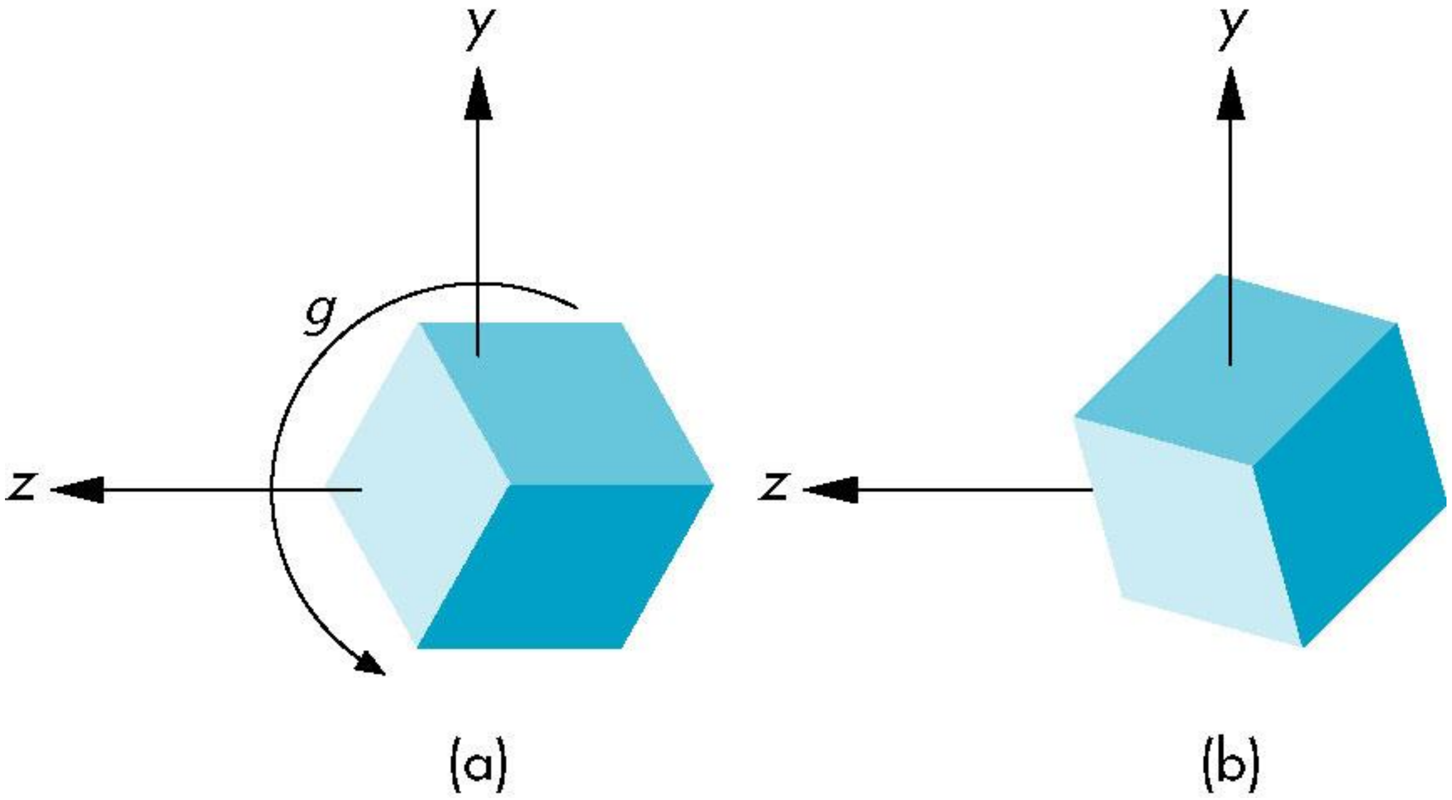
# Rotation about x and y axes

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

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

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

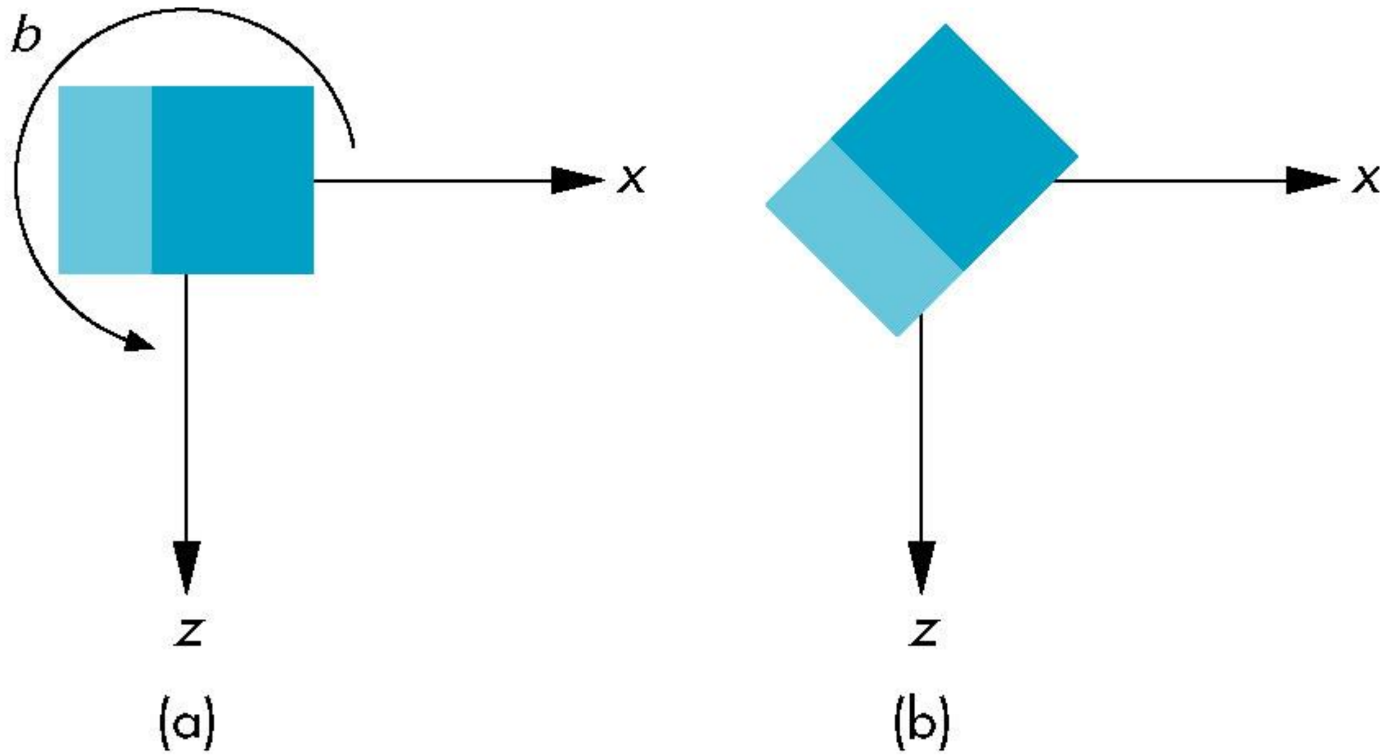
# Rotation of a cube about the x-axis




Before rotation

After rotation

# Rotation of a cube about the y-axis




Before rotation After rotation

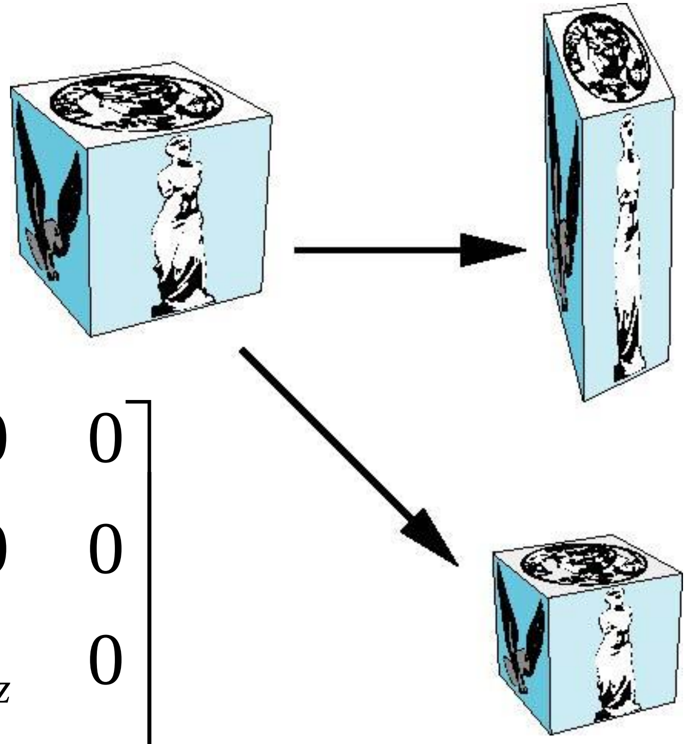
# Scaling

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

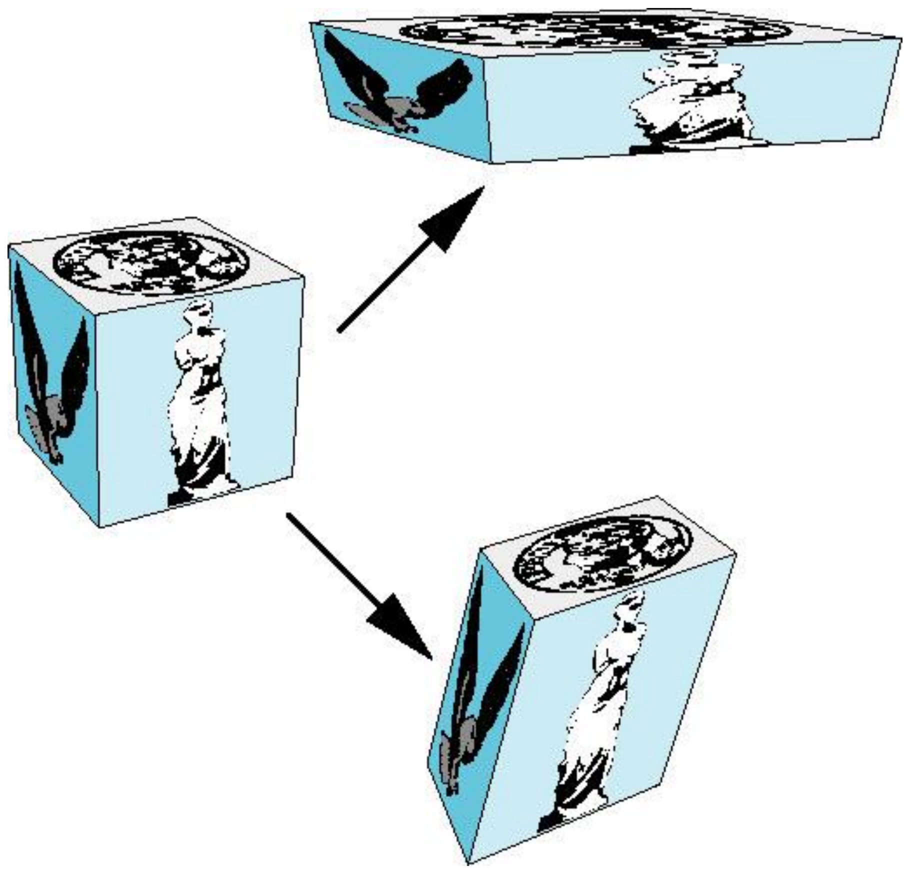
$x' = s_x x$
$y' = s_y y$
$z' = s_z z$

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

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

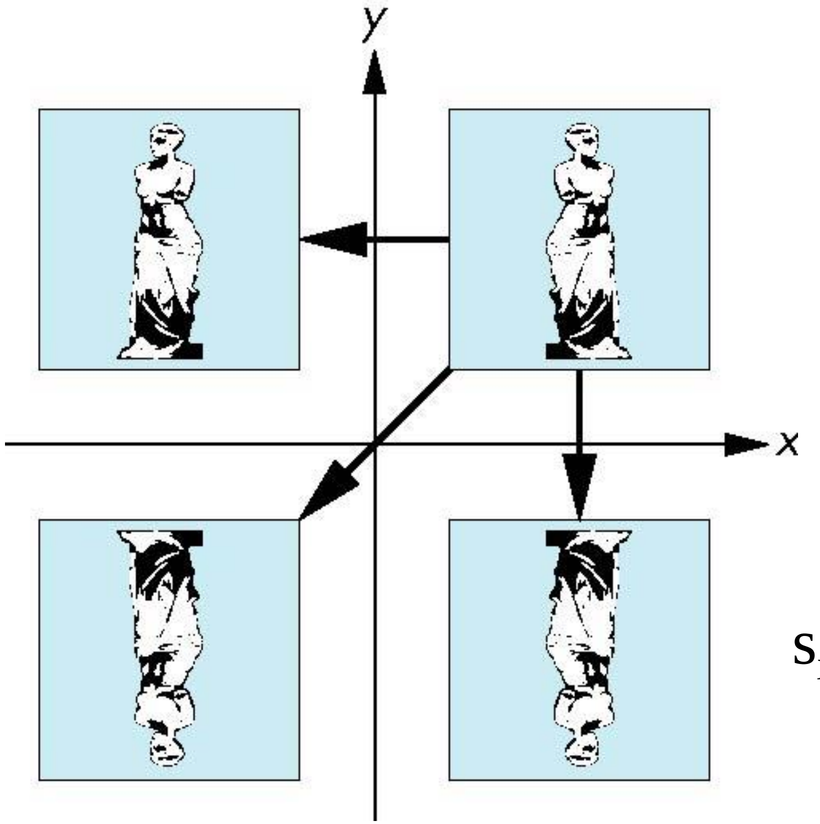


# Non-rigid-body Transformation



# 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

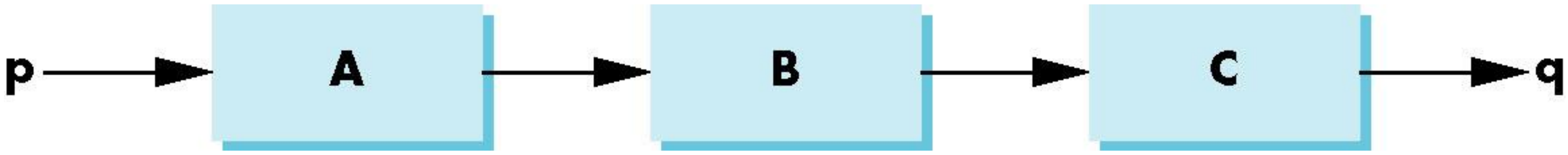
- Note that matrix on the right is the first applied
- Mathematically, the following are equivalent

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

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

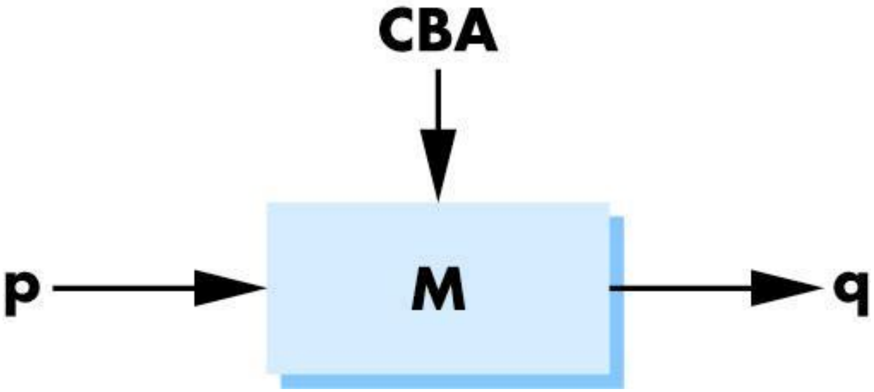
$$\mathbf{p}'^{T} = \mathbf{p}^{T}\mathbf{C}^{T}\mathbf{B}^{T}\mathbf{A}^{T}$$

# Application of transformation one at a time




$q = C\,B\,A\,p$

# Pipeline transformation




$M = CBA$

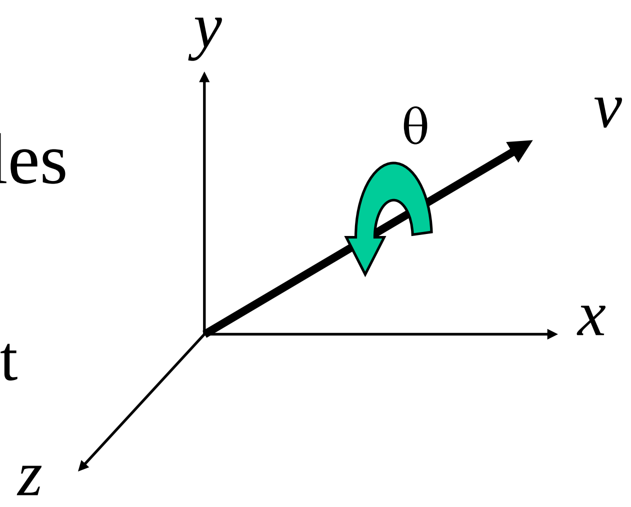
$q = M\,p$

# 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

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

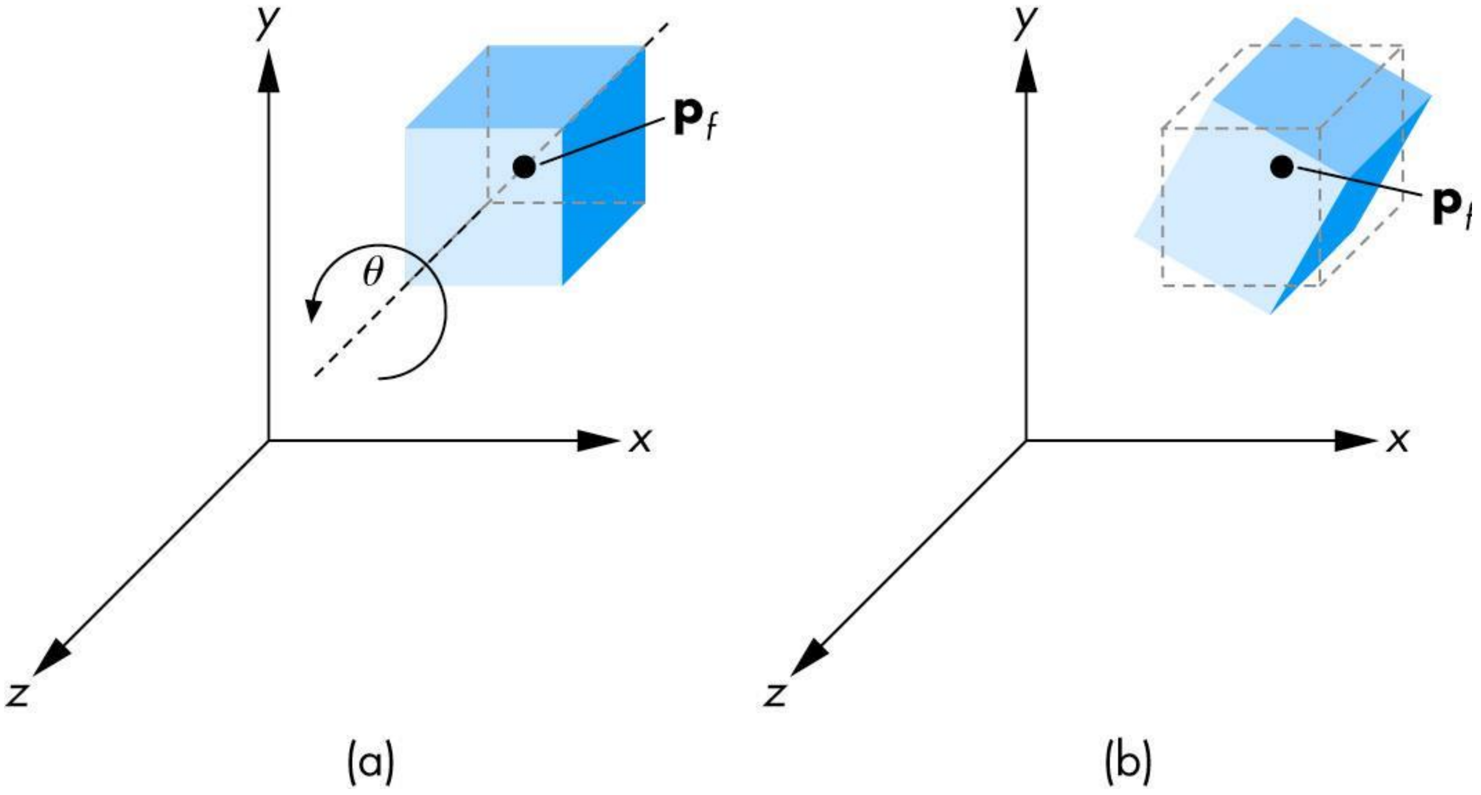
$\theta_x\ \theta_y\ \theta_z$ are called the Euler angles



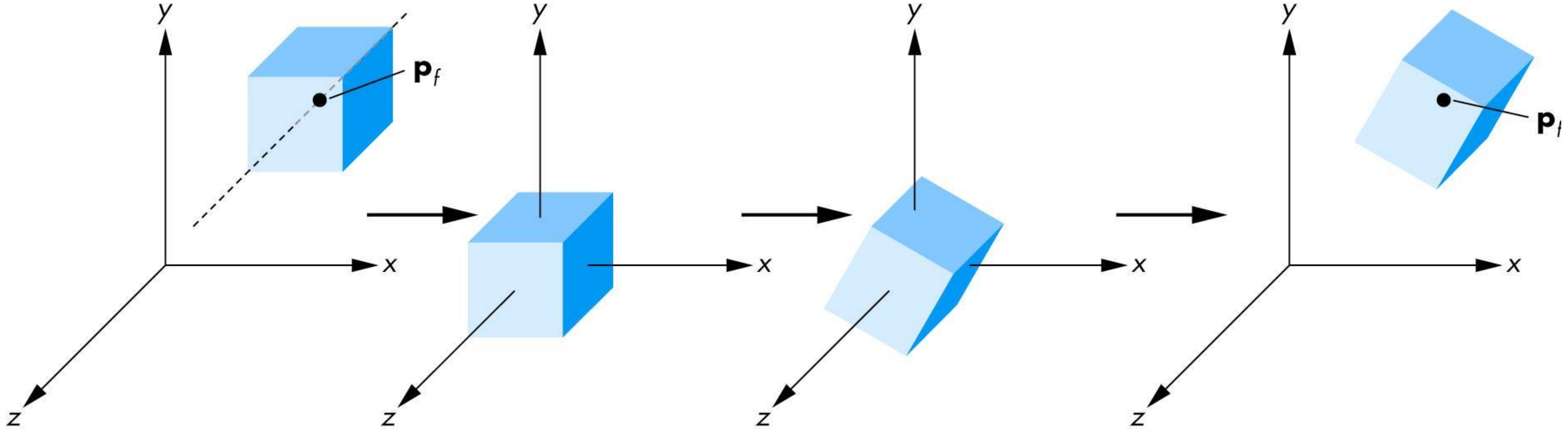

Note that rotations do not commute

We can use rotations in another order but with different angles

# Rotation of a cube about its center



# Rotation of a cube about its center




Move fixed point to origin

Rotate

Move fixed point back

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

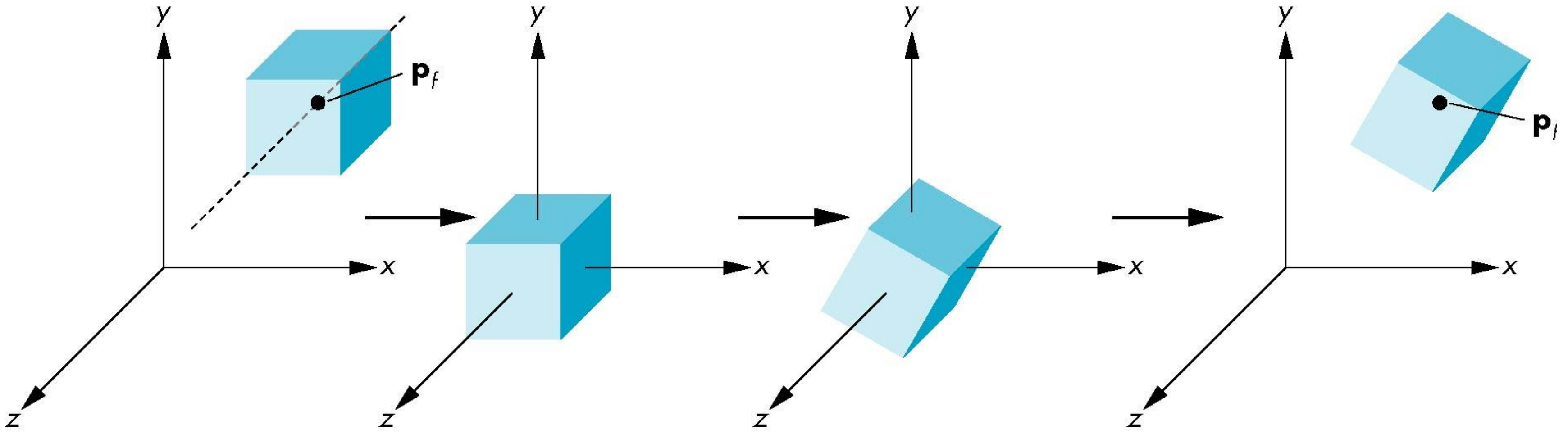
# Rotation About a Fixed Point other than the Origin

Move fixed point to origin

Rotate

Move fixed point back

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



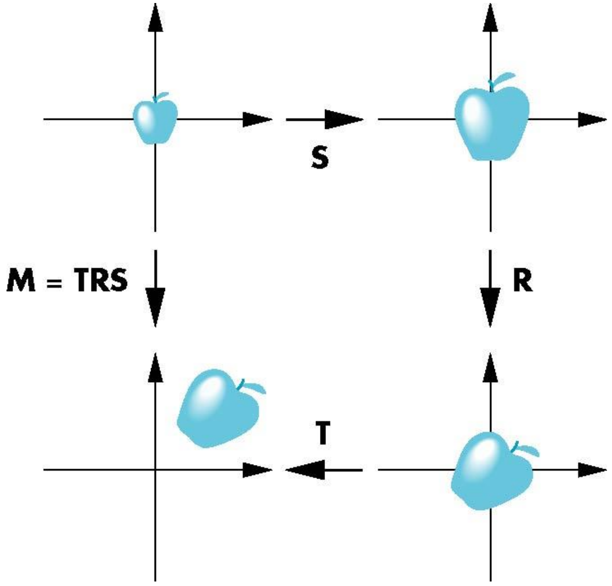
# Instancing

- 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

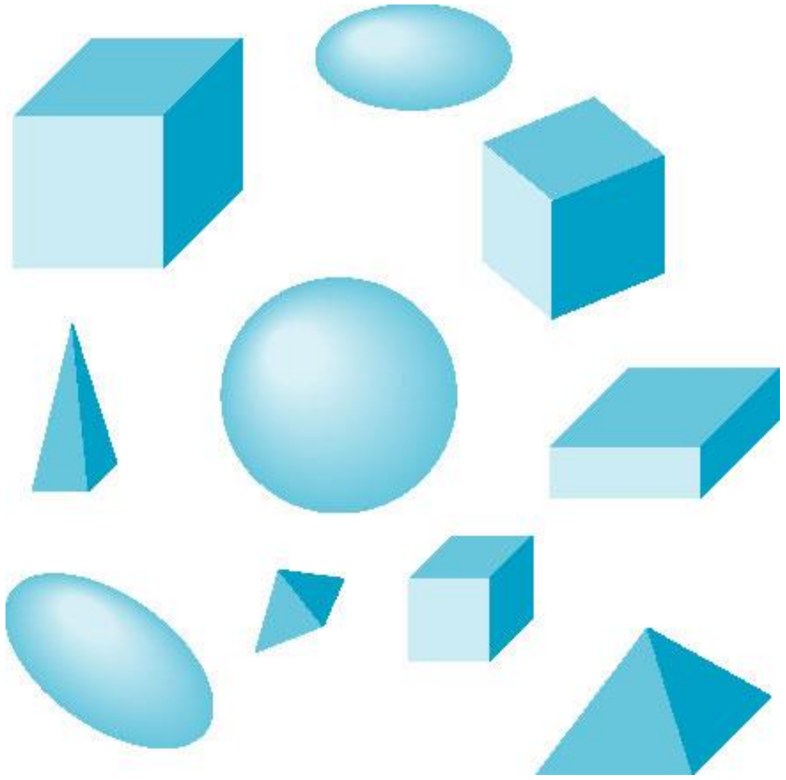
    Scale

    Orient

    Locate

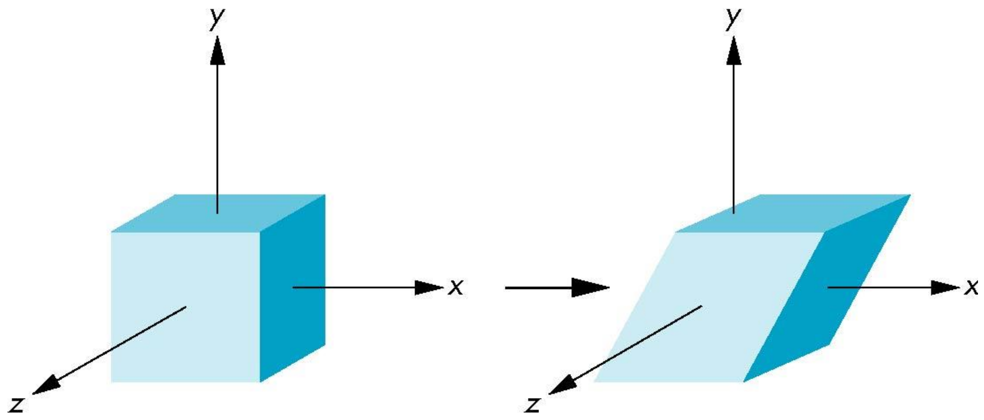


# Scene of simple objects



# Shear

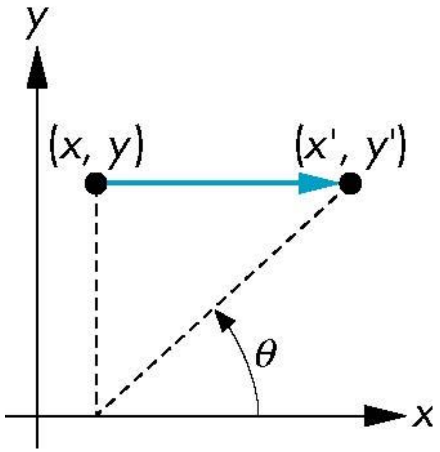
- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions



# 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

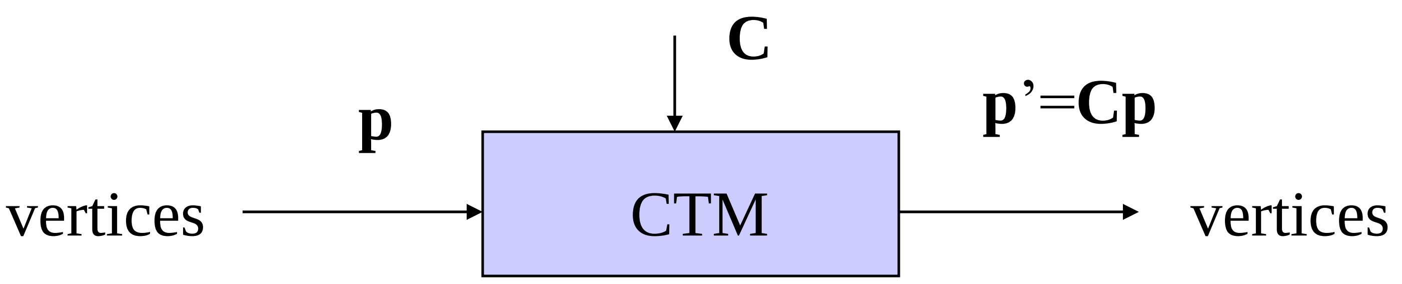
- In Pre 3.1 OpenGL matrices were part of the state
- Multiple types
  - Model-View (`GL_MODELVIEW`)
  - Projection (`GL_PROJECTION`)
  - Texture (`GL_TEXTURE`)
  - Color(`GL_COLOR`)
- Single set of functions for manipulation
- Select which to manipulated by
  - `glMatrixMode(GL_MODELVIEW);`
  - `glMatrixMode(GL_PROJECTION);`

# 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** with the understanding that it may be applied in the shaders

# Current Transformation Matrix (CTM)

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



# CTM operations

- The CTM can be altered either by loading a new CTM or by postmutiplication

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: $\mathbf{C} \leftarrow \mathbf{CM}$
Postmultiply by a translation matrix: $\mathbf{C} \leftarrow \mathbf{CT}$
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 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

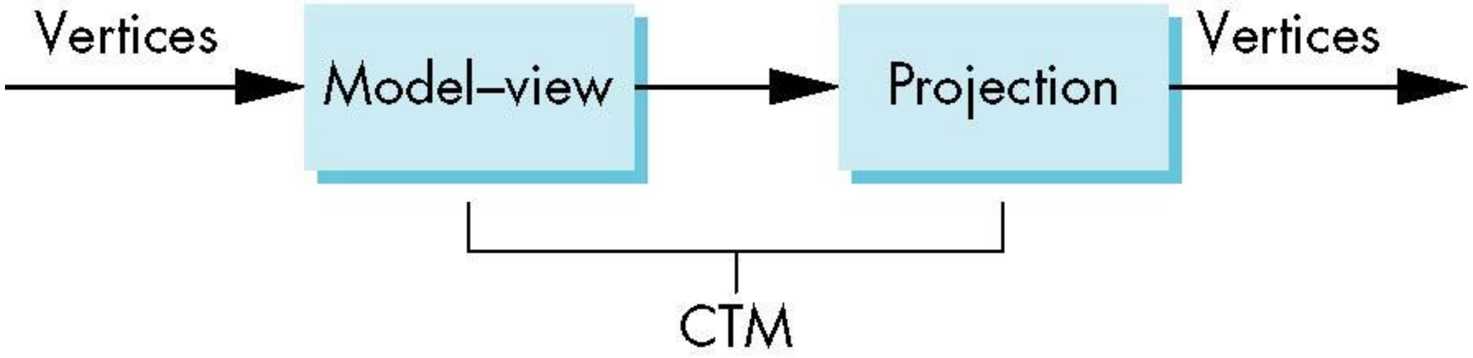
$\mathbf{C} \leftarrow \mathbf{I}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}^{-1}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{R}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$

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

- 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

$q = P*MV*p$ where MV: model-view matrix; P: projection matrix

# 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:

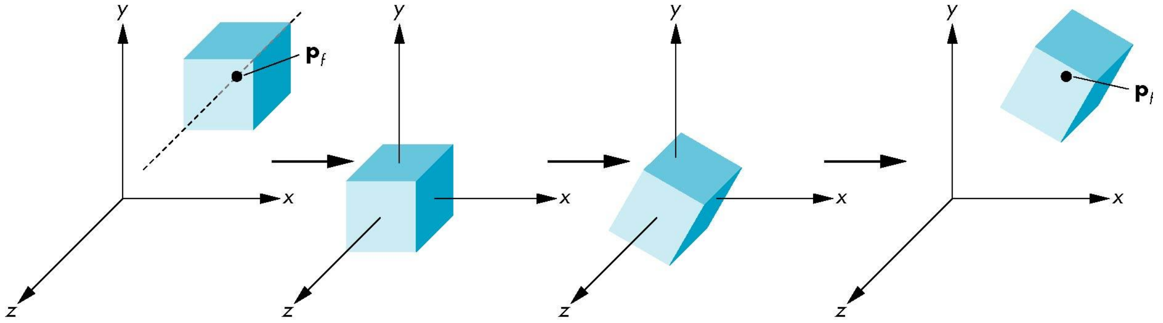
```
var s = scale( sx, sy, sz);
var t = translate(dx, dy, dz);
m = mult(s, t);
```

# 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));
```

Move fixed point to origin
Rotate
Move fixed point back
$\mathbf{M} = \mathbf{T}(p_f)\,\mathbf{R}(\theta)\,\mathbf{T}(-p_f)$




- 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.unifromMatrix4f 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
- 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 );
```

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

# 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;
}
```

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

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

# 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 $\mathbf{M}_0, \mathbf{M}_1, \ldots\ldots, \mathbf{M}_n$ 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 than either

# 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 $\rightarrow$ 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

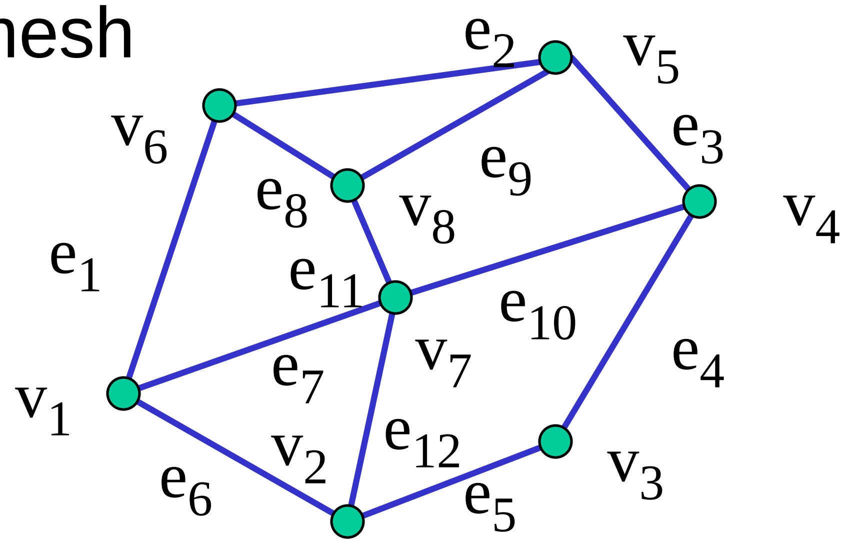
# Building Models

# Objectives

- Introduce simple data structures for building polygonal models
  - Vertex lists
  - Edge lists

# Representing a Mesh

- Consider a mesh




- There are 8 nodes and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges
- Each vertex has a location $v_i = (x_i\ y_i\ z_i)$

# Simple Representation

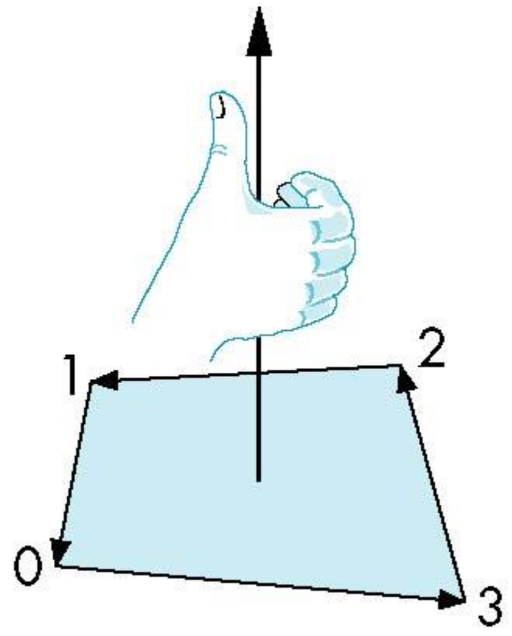
- Define each polygon by the geometric locations of its vertices
- Leads to WebGL code such as

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

- Inefficient and unstructured
  - Consider moving a vertex to a new location
  - Must search for all occurrences

# Inward and Outward Facing Polygons

- The order $\{v_1, v_6, v_7\}$ and $\{v_6, v_7, v_1\}$ are equivalent in that the same polygon will be rendered by OpenGL but the order $\{v_1, v_7, v_6\}$ is different
- The first two describe *outwardly facing* polygons
- Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal
- OpenGL can treat inward and outward facing polygons differently

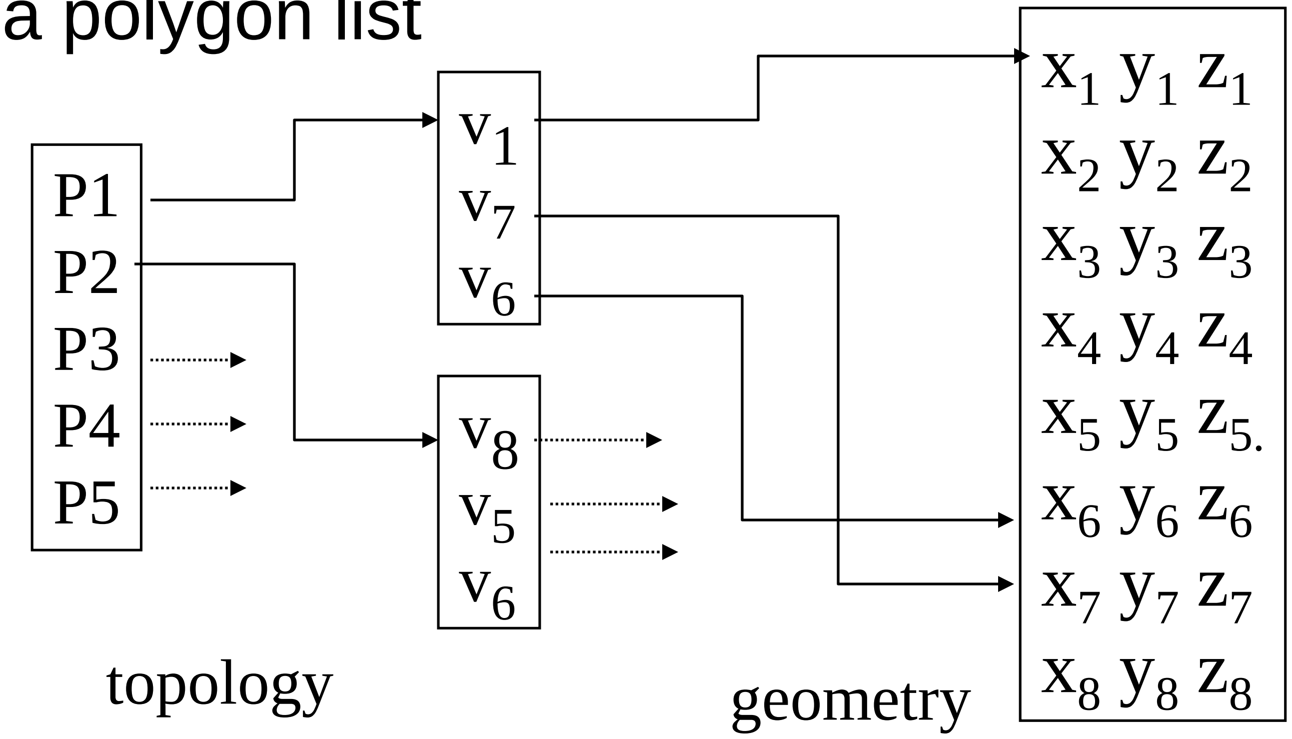


# Geometry vs Topology

- Generally it is a good idea to look for data structures that separate the geometry from the topology
  - Geometry: locations of the vertices
  - Topology: organization of the vertices and edges
  - Example: a polygon is an ordered list of vertices with an edge connecting successive pairs of vertices and the last to the first
  - Topology holds even if geometry changes

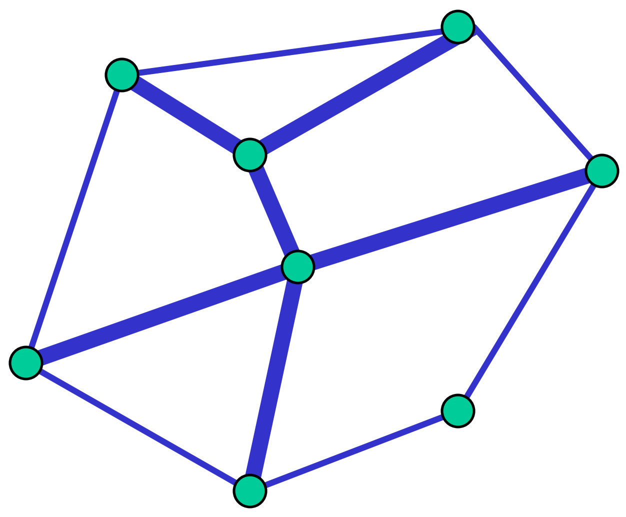
# Vertex Lists

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



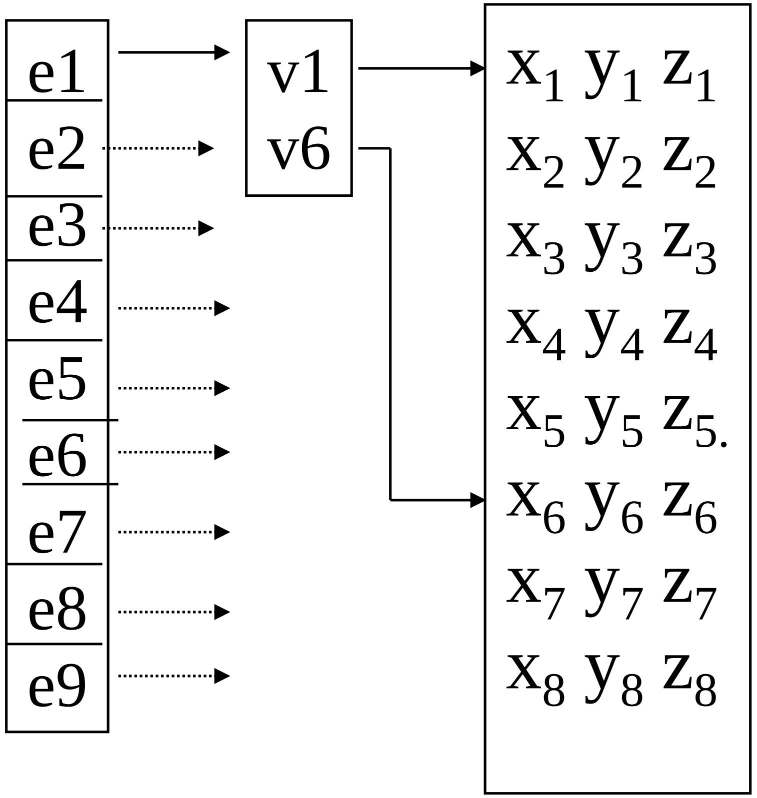
# Shared Edges

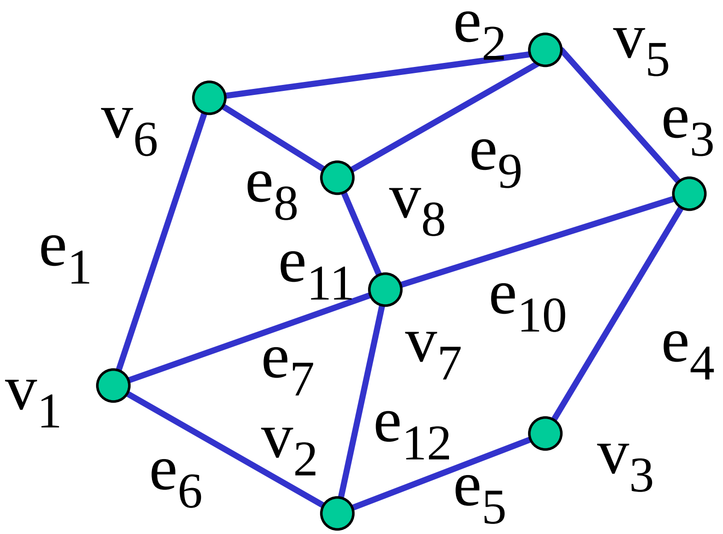
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- Can store mesh by *edge list*

# Edge List

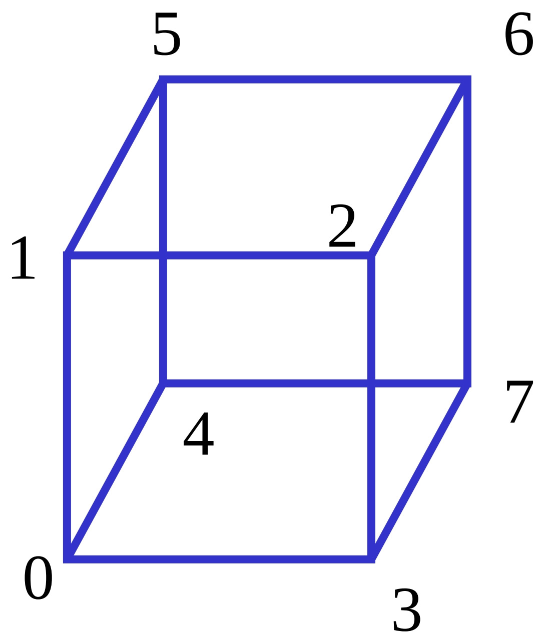



Note polygons are not represented

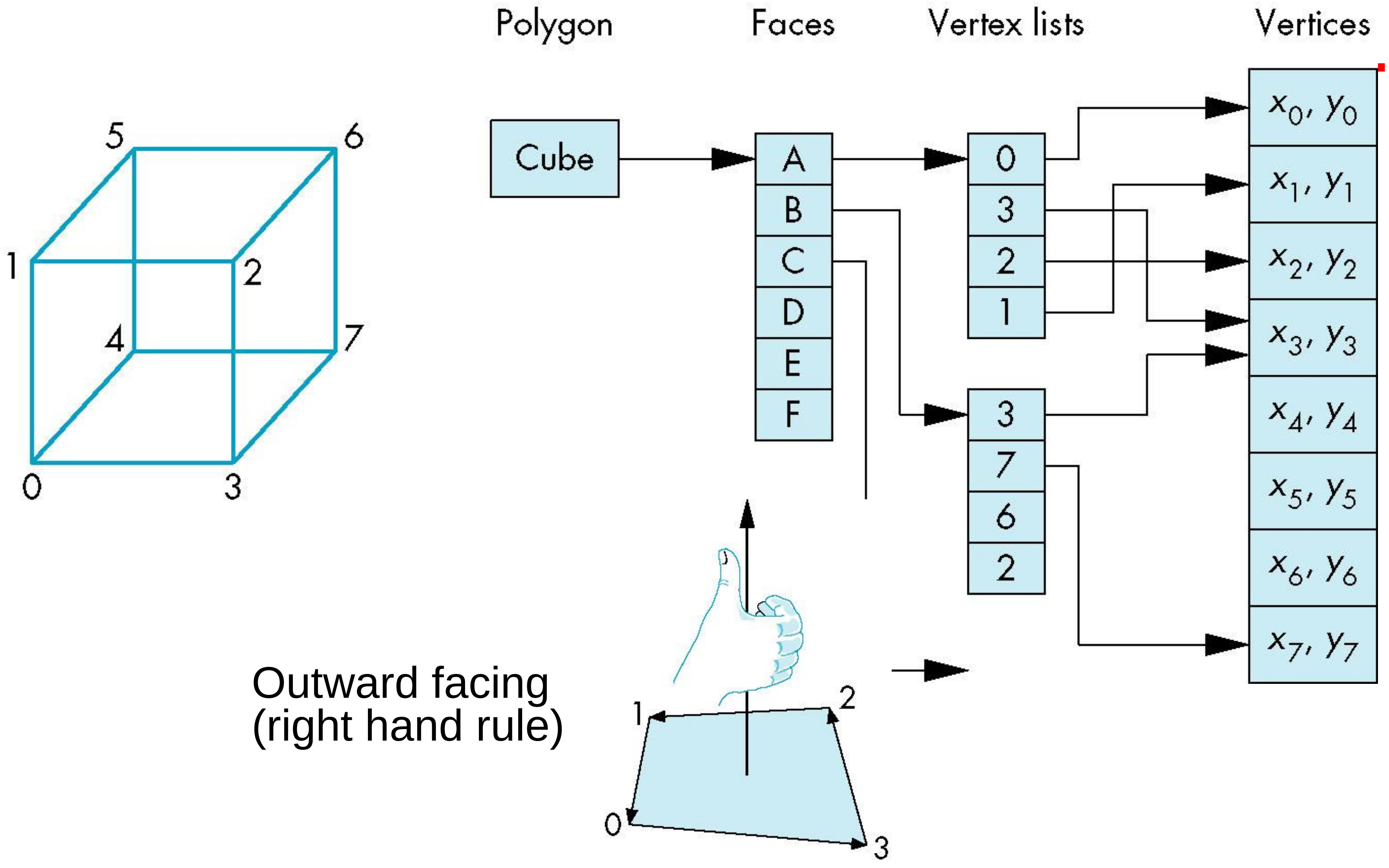
# Draw cube from faces

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

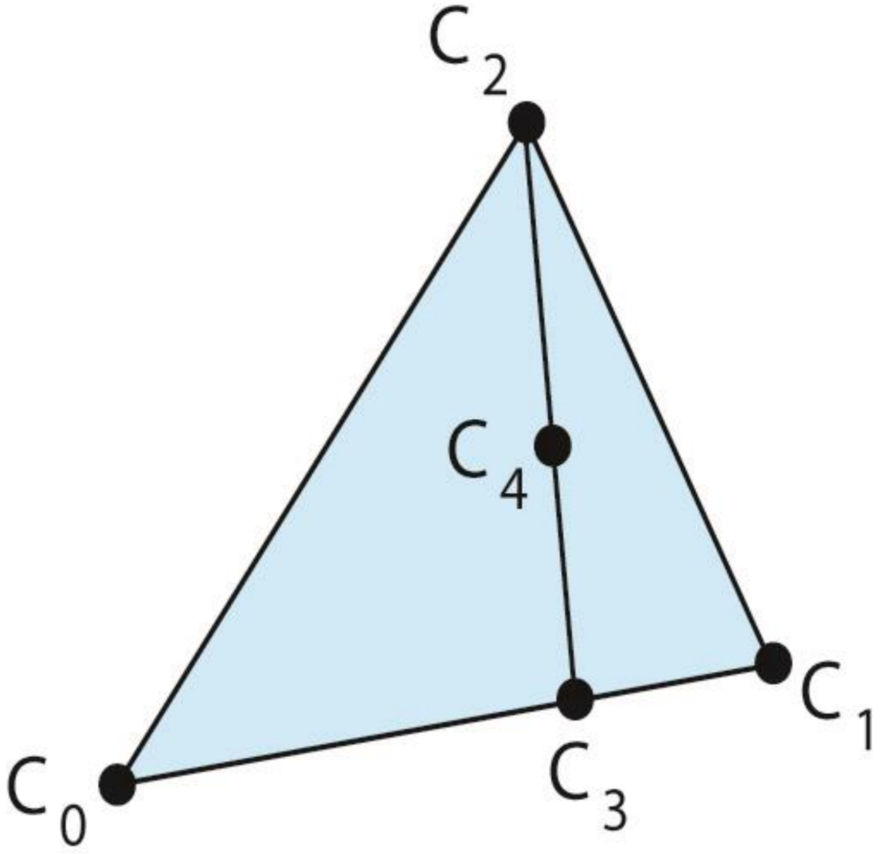



Note that vertices are ordered so that
we obtain correct outward facing normals

# Data Structures for Cube Representation



# Color Interpolation Using Barycentric Coordinates




$$C_{01}(\alpha) = (1-\alpha)C_0 + \alpha C_1 \Leftarrow C_3$$

$$C_{32}(\beta) = (1-\beta)C_3 + \beta C_2 \Leftarrow C_4$$

# The Rotating Square

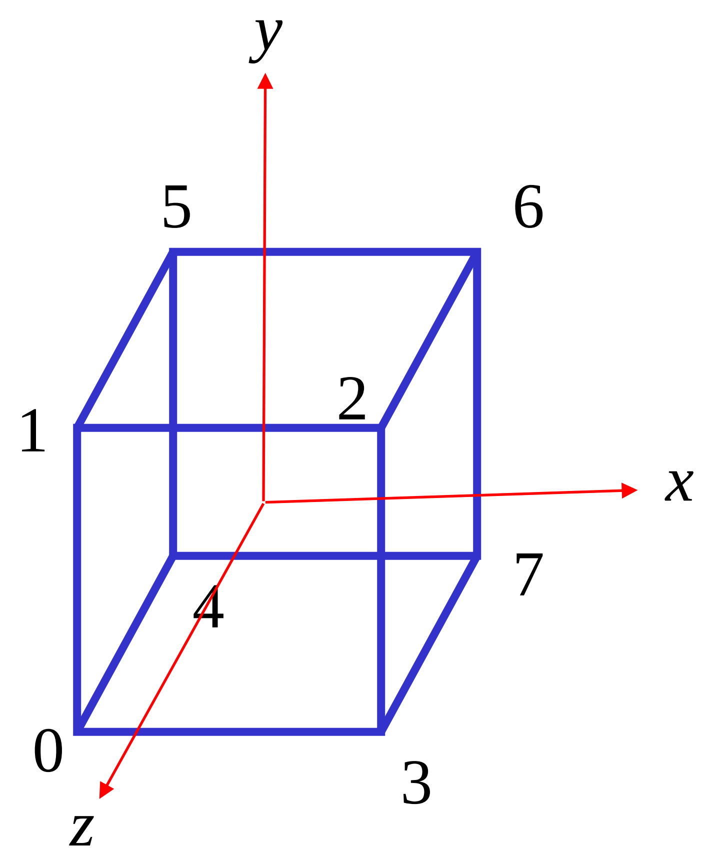
# Objectives

- Put everything together to display rotating cube
- Two methods of display
  - by arrays
  - by elements

# Modeling a Cube

Define global array for vertices

```
var vertices = [
        vec3( -0.5, -0.5,  0.5 ),
        vec3( -0.5,  0.5,  0.5 ),
        vec3(  0.5,  0.5,  0.5 ),
        vec3(  0.5, -0.5,  0.5 ),
        vec3( -0.5, -0.5, -0.5 ),
        vec3( -0.5,  0.5, -0.5 ),
        vec3(  0.5,  0.5, -0.5 ),
        vec3(  0.5, -0.5, -0.5 )
    ];
```



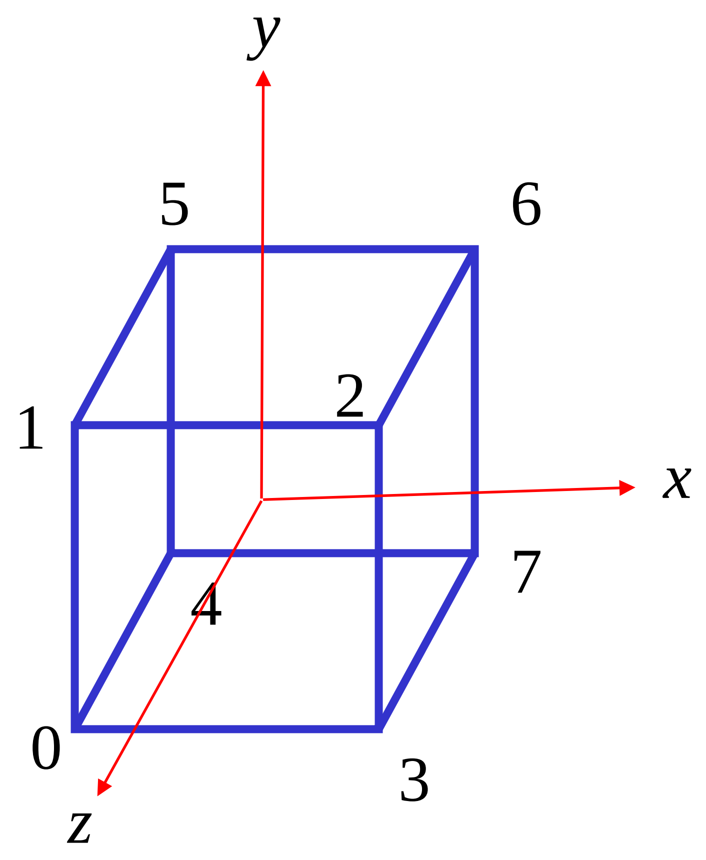
# Colors

Define global array for colors

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

# Draw cube from faces

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




Note that vertices are ordered so that we obtain correct outward facing normals
Each quad generates two triangles

# Initialization

```
var canvas, gl;
var numVertices  = 36;
var points = [];
var colors = [];

window.onload = function init(){
    canvas = document.getElementById( "gl-canvas" );
    gl = WebGLUtils.setupWebGL( canvas );

    colorCube();

   gl.viewport( 0, 0, canvas.width, canvas.height );
   gl.clearColor( 1.0, 1.0, 1.0, 1.0 );
   gl.enable(gl.DEPTH_TEST);

// rest of initialization and html file
// same as previous examples
```

# The quad Function

Put position and color data for two triangles from a list of indices into the array **`vertices`**

```
var quad(a, b, c, d)
{
   var indices = [ a, b, c, a, c, d ];
   for ( var i = 0; i < indices.length; ++i ) {

      points.push( vertices[indices[i]]);
      colors.push( vertexColors[indices[i]] );

// for solid colored faces use
//colors.push(vertexColors[a]);

 }
}
```

# Render Function

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

# Mapping indices to faces

```
var indices = [
1,0,3,
3,2,1,
2,3,7,
7,6,2,
3,0,4,
4,7,3,
6,5,1,
1,2,6,
4,5,6,
6,7,4,
5,4,0,
0,1,5
];
```

# Rendering by Elements

- Send indices to GPU

```
var iBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, iBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
       new Uint8Array(indices), gl.STATIC_DRAW);
```

- Render by elements

```
gl.drawElements( gl.TRIANGLES, numVertices,
    gl.UNSIGNED_BYTE, 0 );
```

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

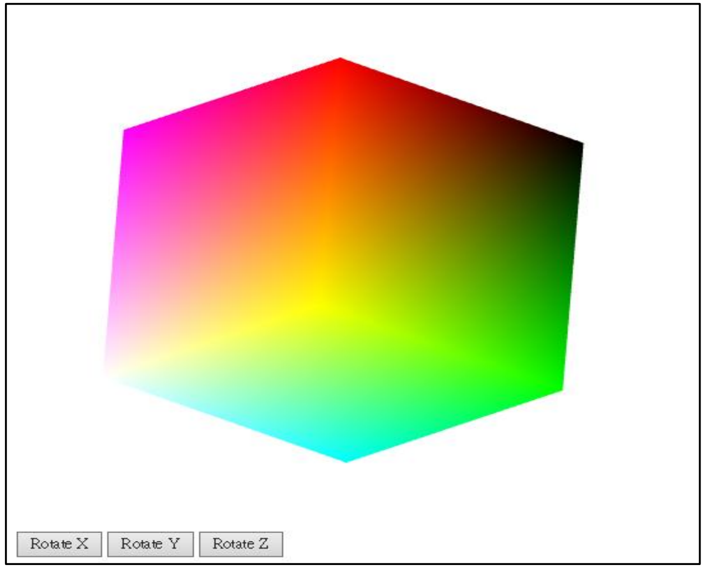
# Adding Buttons for Rotation

```
var xAxis = 0;
var yAxis = 1;
var zAxis = 2;
var axis = 0;
var theta = [ 0, 0, 0 ];
var thetaLoc;

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

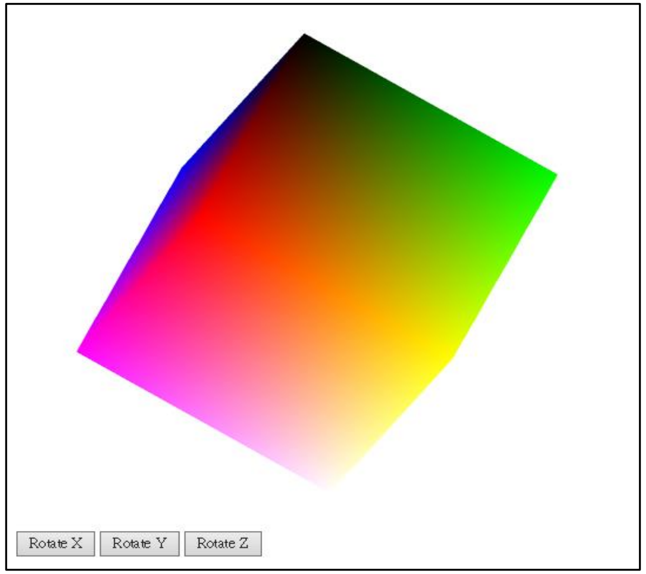
# Render Function

```
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 );
}
```

# Sample Programs

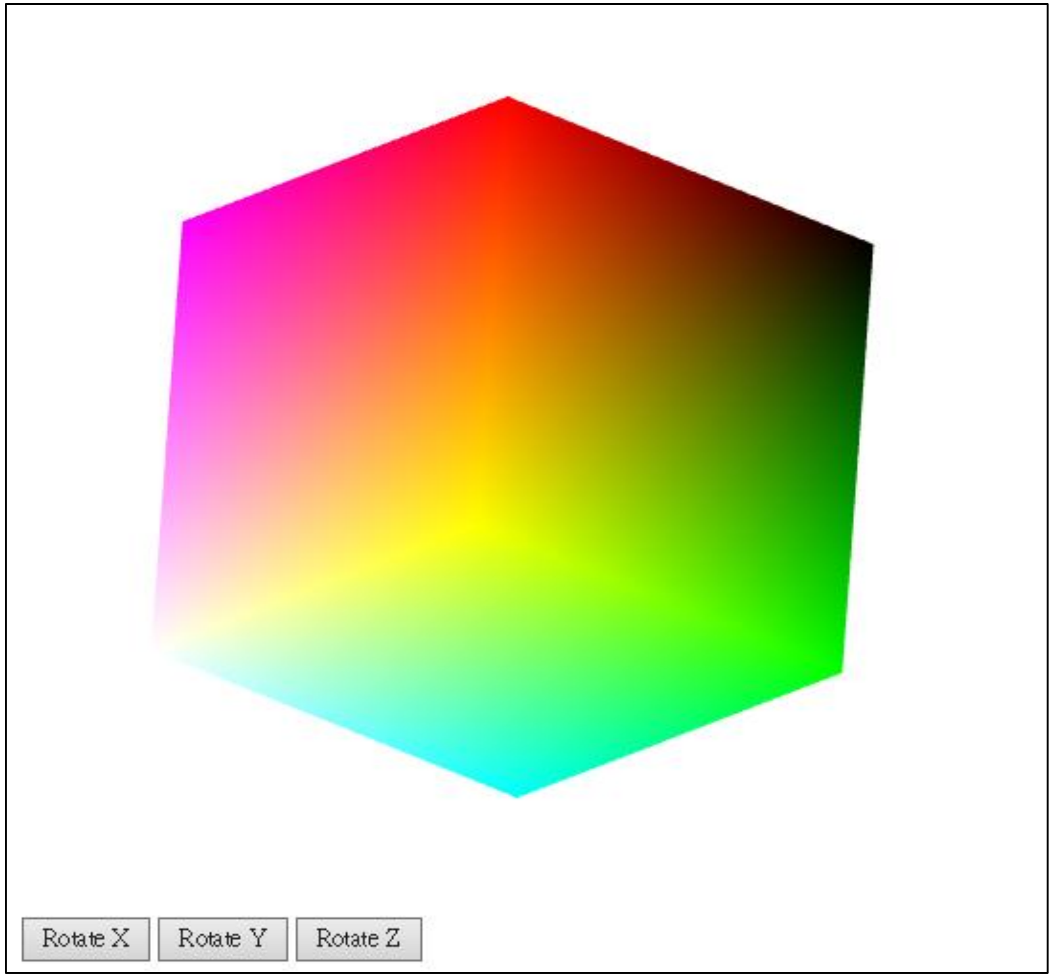



cube




cubev

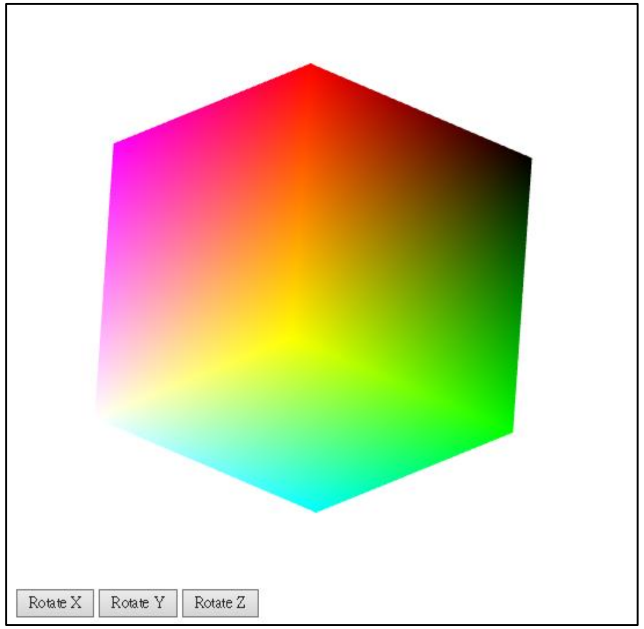
# Sample Programs: cube.html, cube.js




Displaying a rotating cube with vertex colors interpolated across faces

# cube.html (1/4)

```
<html>

<script id="vertex-shader" type="x-shader/x-vertex">

attribute  vec4 vPosition;
attribute  vec4 vColor;
varying vec4 fColor;

uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for each of
    // the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
```



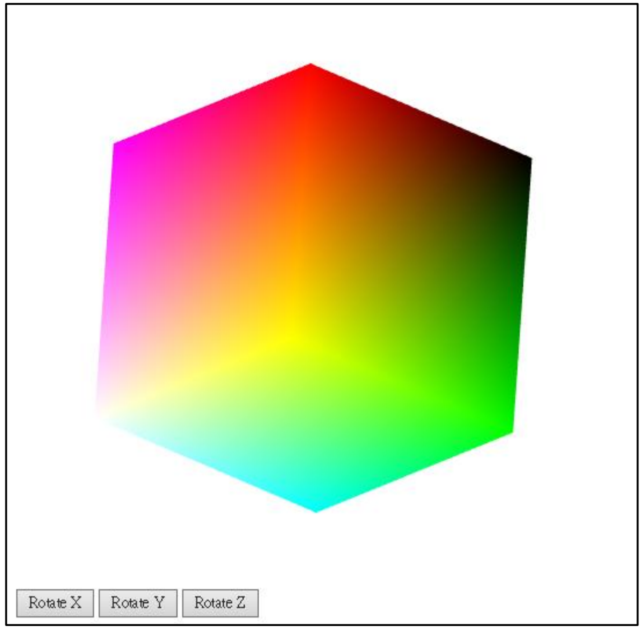
# cube.html (2/4)

```
// 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 );
fColor = vColor;
gl_Position = rz * ry * rx * vPosition;
}
</script>
```

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

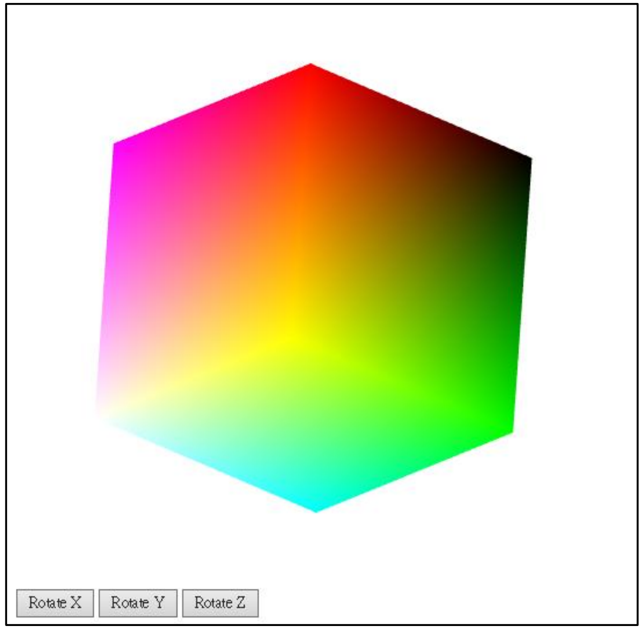
$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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



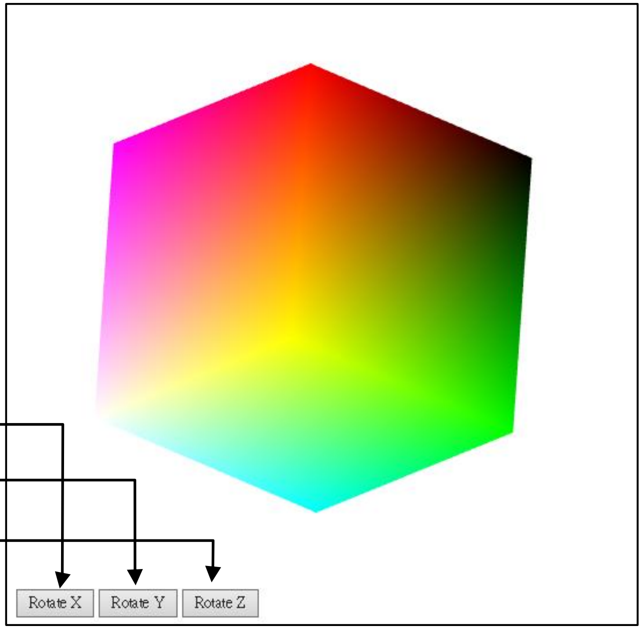
# cube.html (3/4)

```
<script id="fragment-shader" type="x-shader/x-fragment">

precision mediump float;

varying vec4 fColor;

void
main()
{
    gl_FragColor = fColor;
}
</script>

<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
<script type="text/javascript" src="cube.js"></script>
```



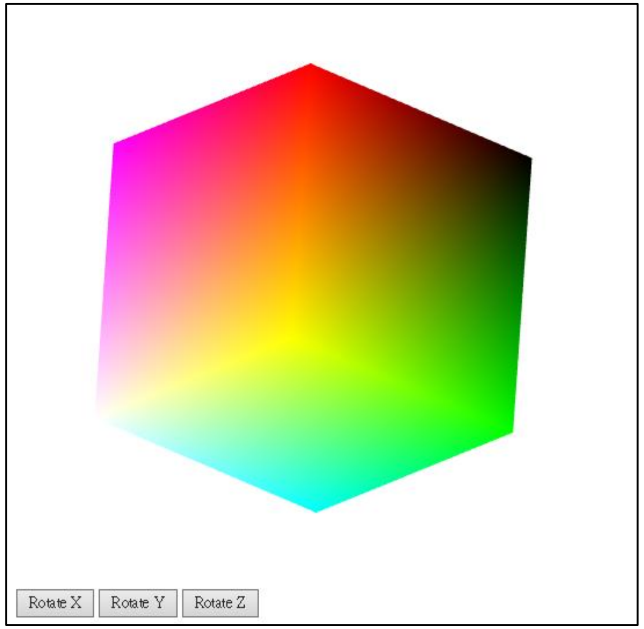
# cube.html (4/4)

```
<body>
<canvas id="gl-canvas" width="512"" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>

<br/>

<button id= "xButton">Rotate X</button>
<button id= "yButton">Rotate Y</button>
<button id= "zButton">Rotate Z</button>

</body>
</html>
```



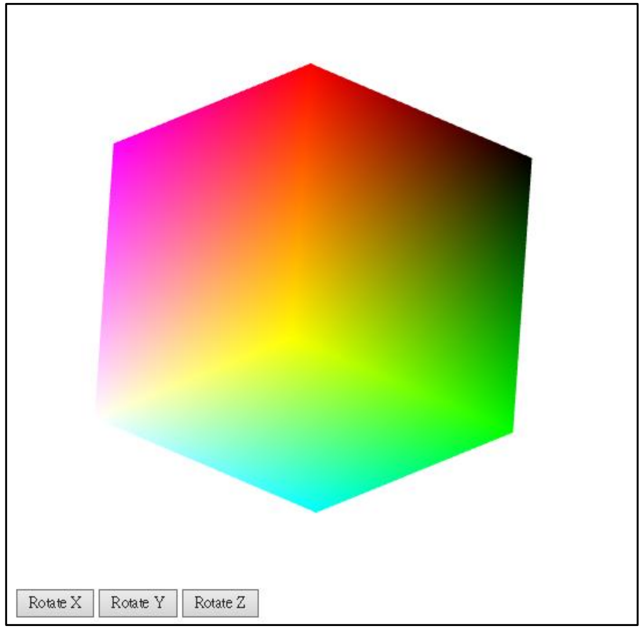
# cube.js (1/10)

```
var canvas;
var gl;

var NumVertices  = 36;

var points = [];
var colors = [];

var xAxis = 0;
var yAxis = 1;
var zAxis = 2;

var axis = 0;
var theta = [ 0, 0, 0 ];

var thetaLoc;
```



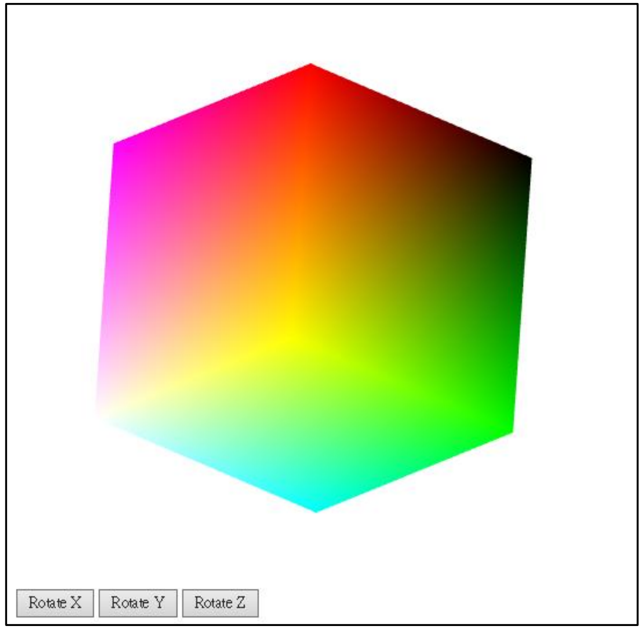
# cube.js (2/10)

```
window.onload = function init()
{
    canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }

    colorCube();

    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

    gl.enable(gl.DEPTH_TEST);
```



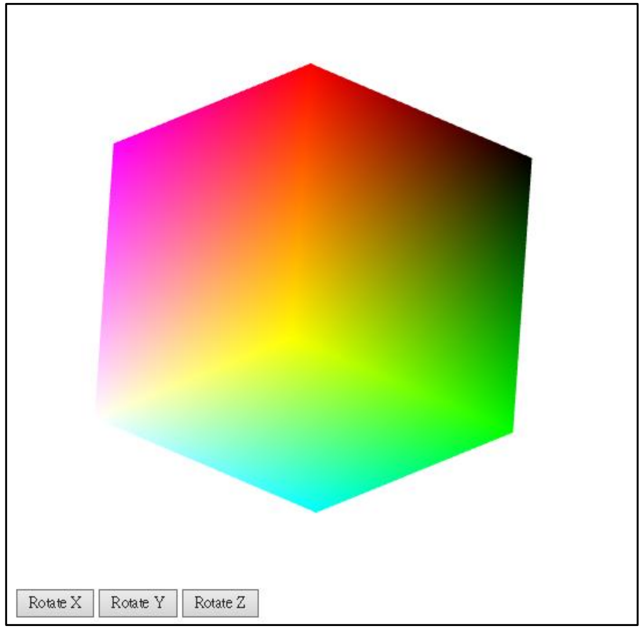
# cube.js (3/10)

```
//
//  Load shaders and initialize attribute buffers
//
var program = initShaders( gl, "vertex-shader", "fragment-shader" );
gl.useProgram( program );

var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW );

var vColor = gl.getAttribLocation( program, "vColor" );
gl.vertexAttribPointer( vColor, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vColor );
```



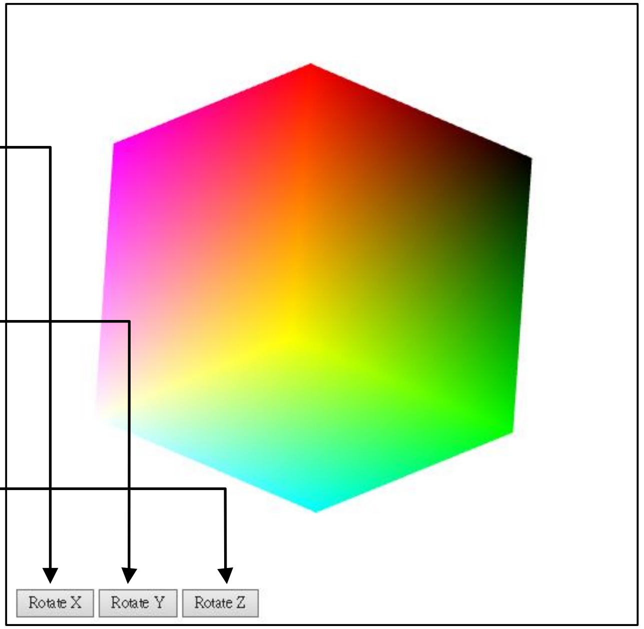
# cube.js (4/10)

```
var vBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );

var vPosition = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( vPosition, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );

thetaLoc = gl.getUniformLocation(program, "theta");
```



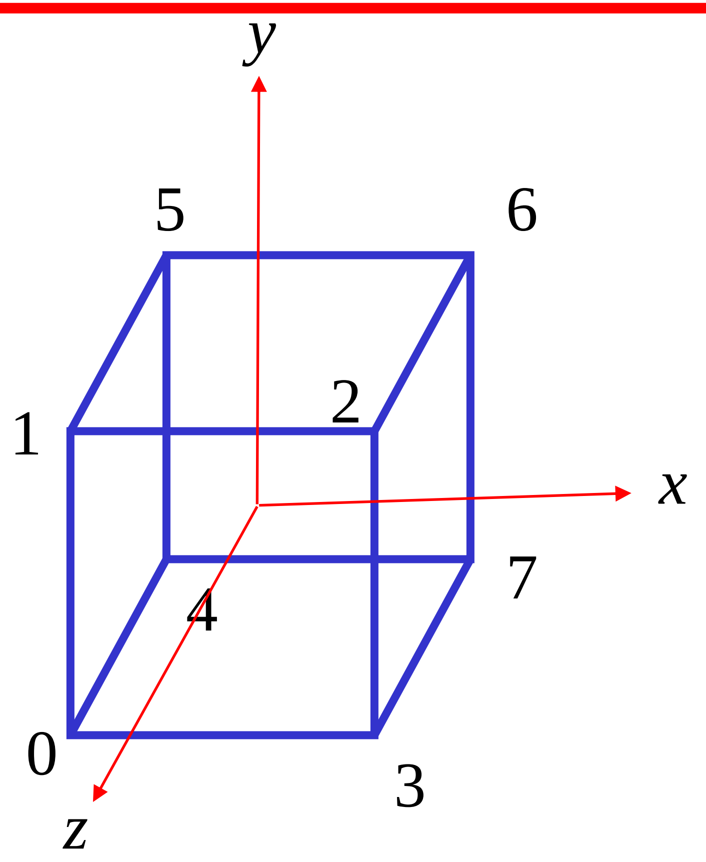
# cube.js (5/10)

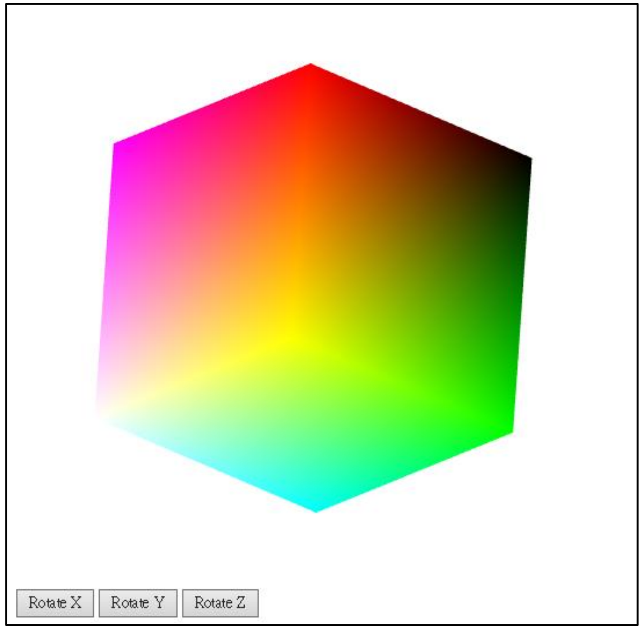
```
//event listeners for buttons

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

render();
}  // end of window.onload
```



# cube.js (6/10)

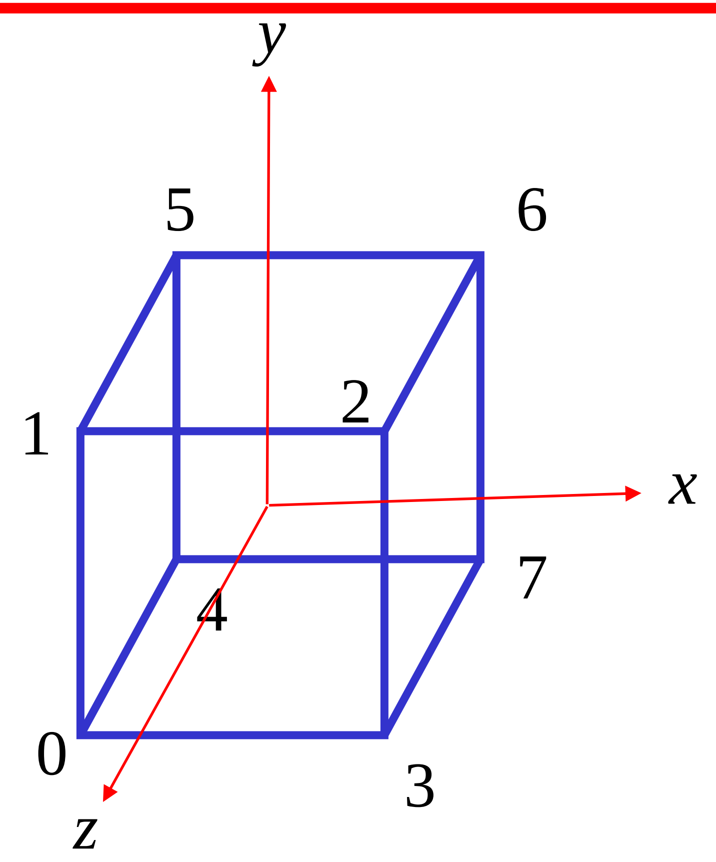
```
function colorCube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

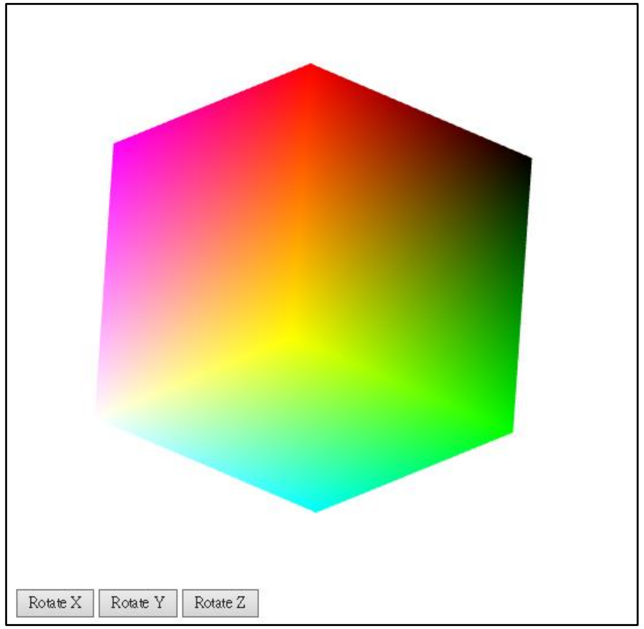


# cube.js (7/10)

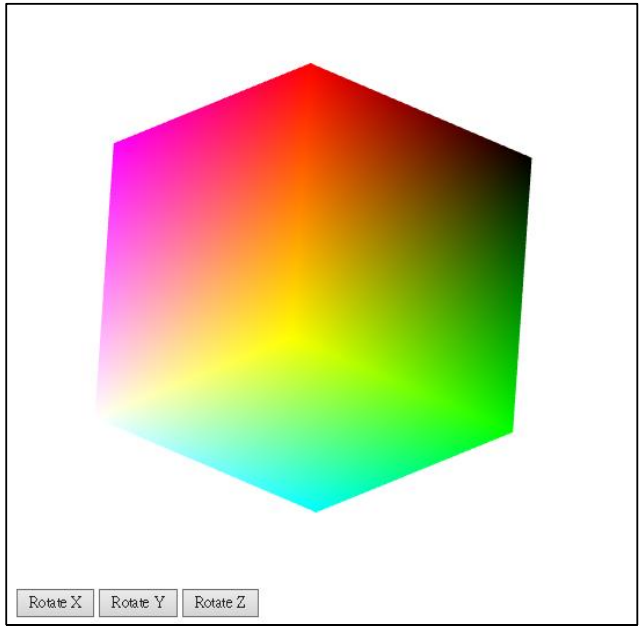
```
function quad(a, b, c, d)
{
    var vertices = [
        vec3( -0.5, -0.5,  0.5 ),
        vec3( -0.5,  0.5,  0.5 ),
        vec3(  0.5,  0.5,  0.5 ),
        vec3(  0.5, -0.5,  0.5 ),
        vec3( -0.5, -0.5, -0.5 ),
        vec3( -0.5,  0.5, -0.5 ),
        vec3(  0.5,  0.5, -0.5 ),
        vec3(  0.5, -0.5, -0.5 )
    ];
```



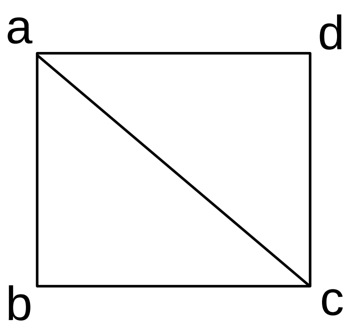
# cube.js (8/10)

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

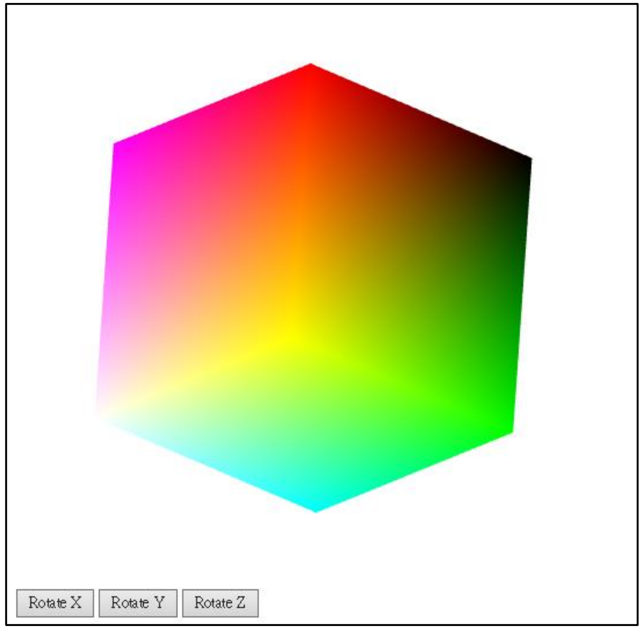


# cube.js (9/10)

```
// We need to parition the quad into two triangles in order for
// WebGL to be able to render it.  In this case, we create two
// triangles from the quad indices

//vertex color assigned by the index of the vertex

var indices = [ a, b, c, a, c, d ];

for ( var i = 0; i < indices.length; ++i ) {
    points.push( vertices[indices[i]] );
    colors.push( vertexColors[indices[i]] );

    // for solid colored faces use
    //colors.push(vertexColors[a]);

}
}  // end of quad(a, b,c,d)
```

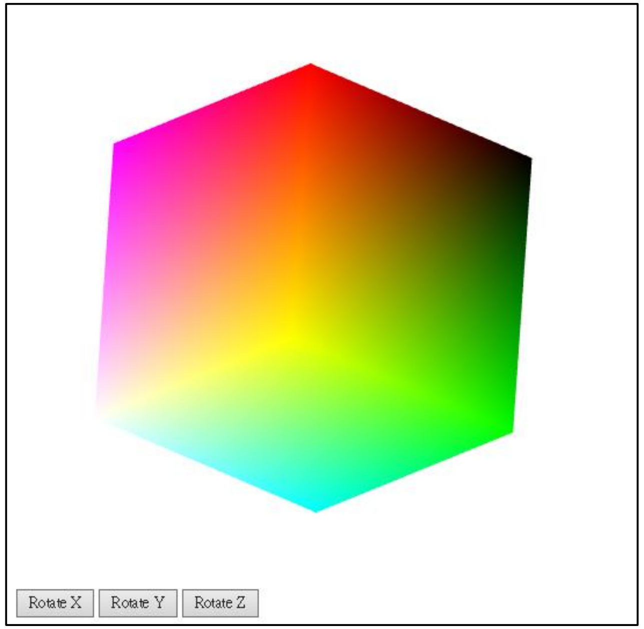



gl.TRIANGLES

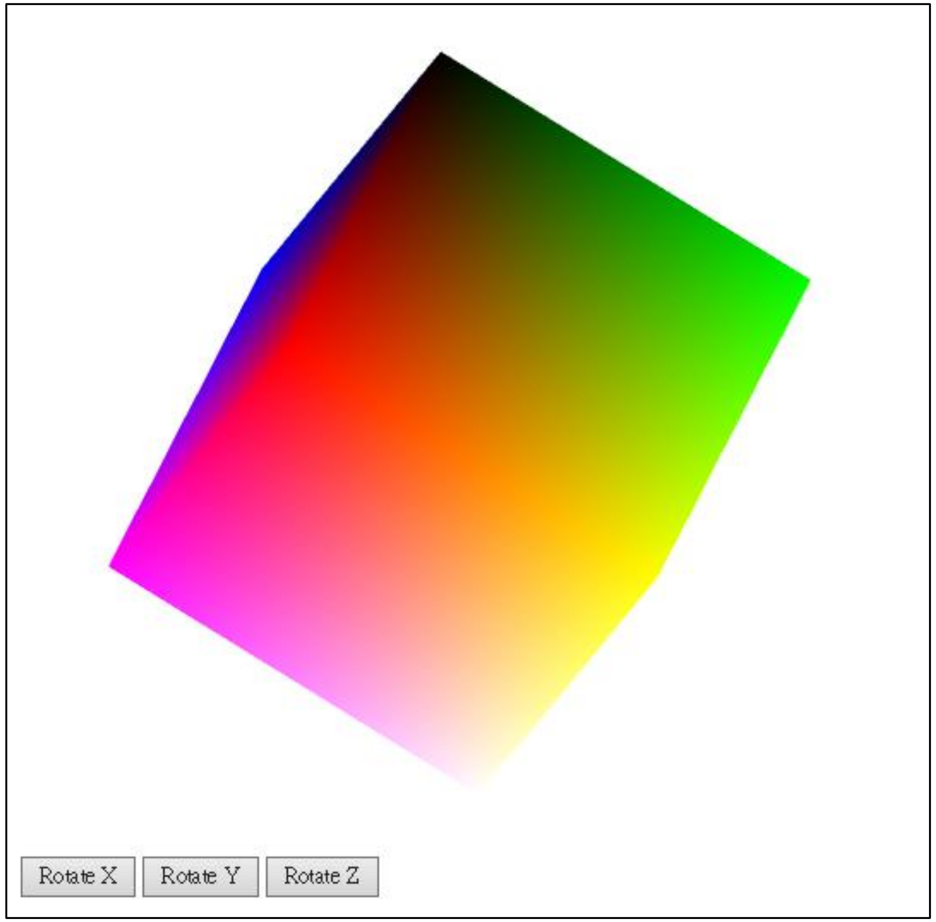


# cube.js (10/10)

```
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 );
}
```



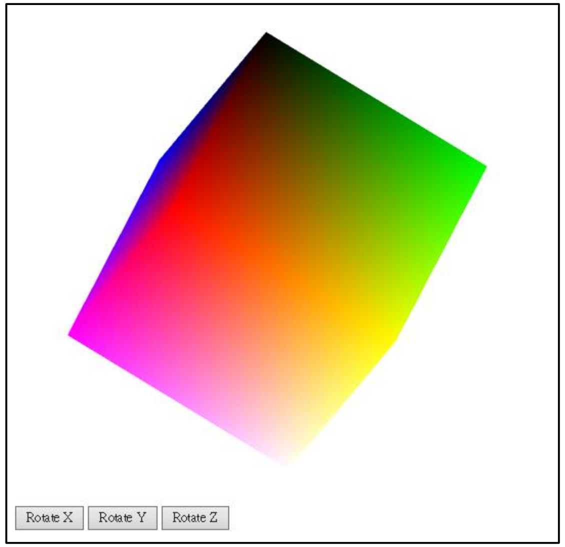
# Sample Programs: cubev.html, cubev.js




Same as cube but with element arrays

# cubev.html (1/4)

```
<html>

<script id="vertex-shader" type="x-shader/x-vertex">

attribute  vec4 vPosition;
attribute  vec4 vColor;
varying vec4 fColor;

uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for each of
    //   the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
```



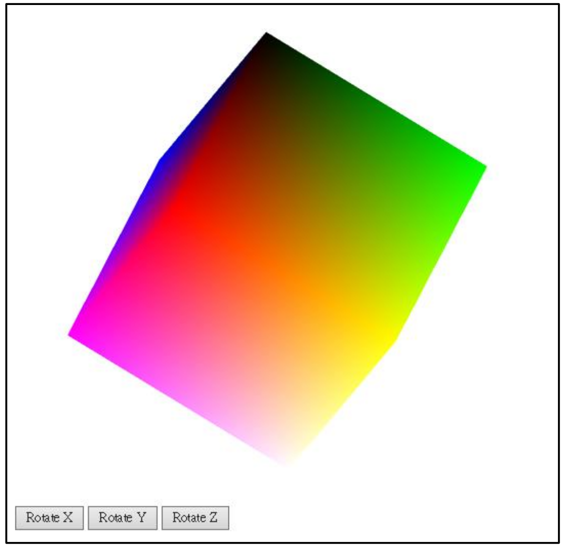
# cubev.html (2/4)

```
// 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 );
fColor = vColor;
gl_Position = rz * ry * rx * vPosition;
}
</script>
```

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

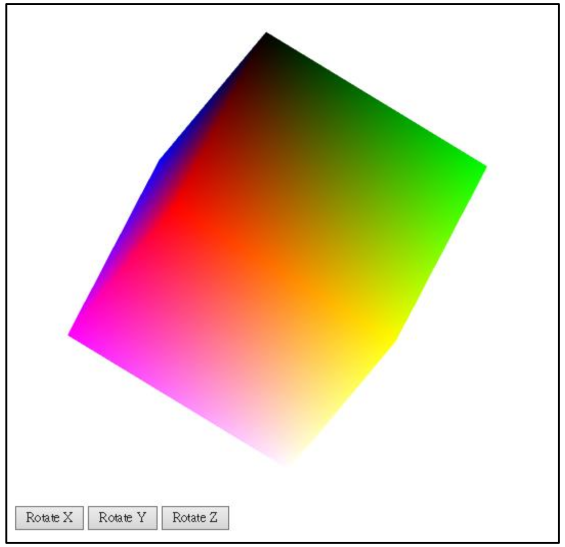
$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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



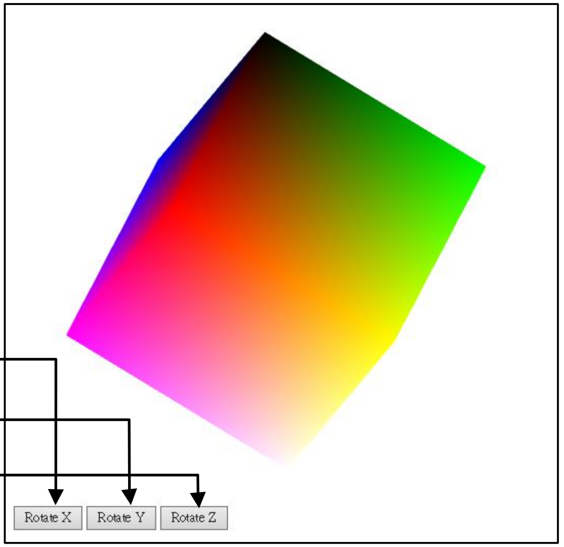
# cubev.html (3/4)

```
<script id="fragment-shader" type="x-shader/x-fragment">

precision mediump float;

varying vec4 fColor;

void main()
{
    gl_FragColor = fColor;
}
</script>

<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
<script type="text/javascript" src="cubev.js"></script>
```



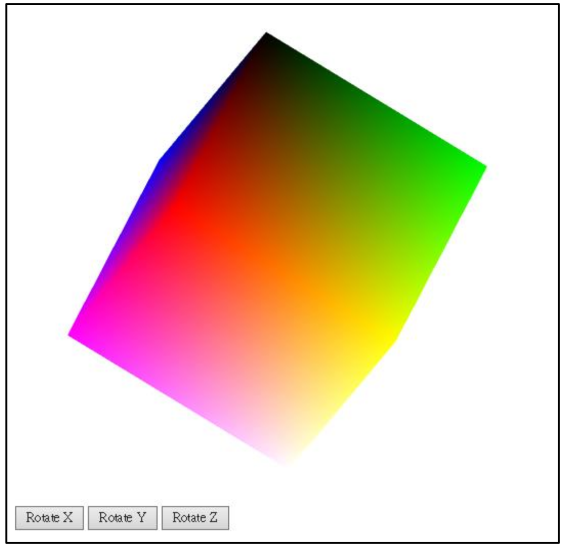
# cubev.html (4/4)

```
<body>
<canvas id="gl-canvas" width="512"" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>

<br/>
<button id= "xButton">Rotate X</button>
<button id= "yButton">Rotate Y</button>
<button id= "zButton">Rotate Z</button>

</body>
</html>
```



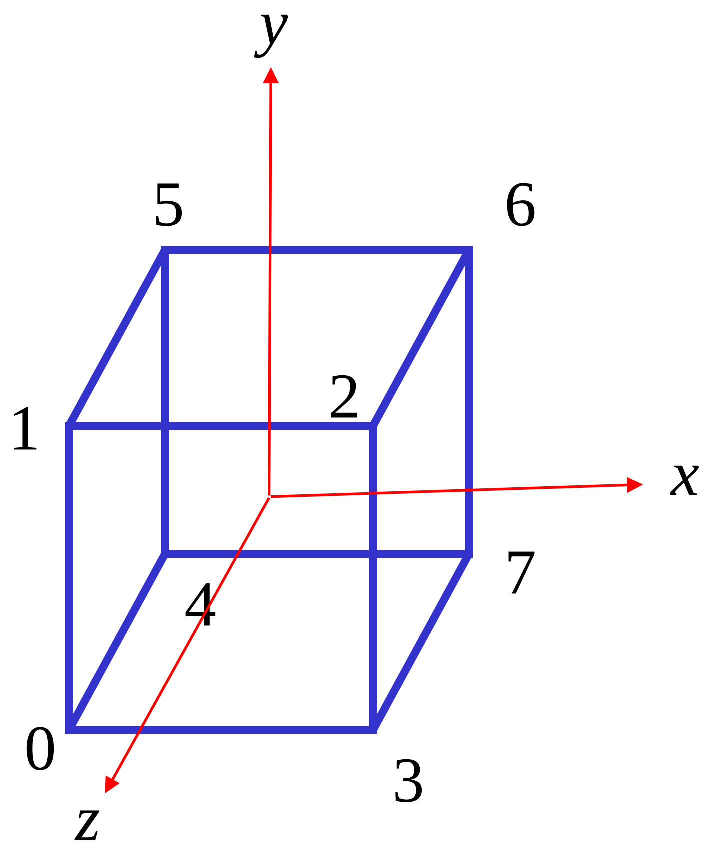
# cubev.js (1/10)

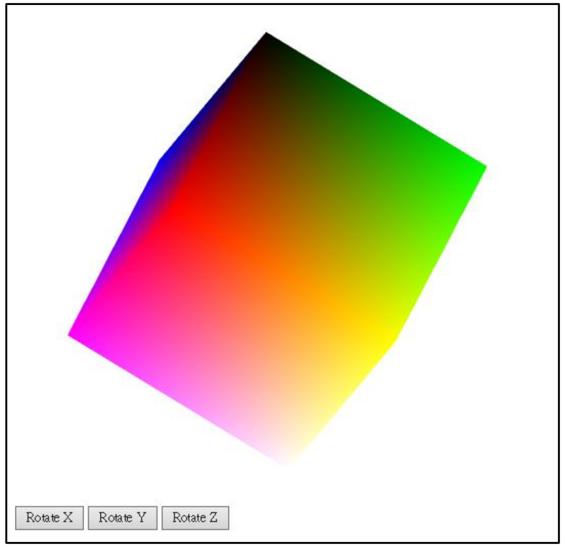
```
var canvas;
var gl;

var numVertices  = 36;

var axis = 0;
var xAxis = 0;
var yAxis =1;
var zAxis = 2;

var theta = [ 0, 0, 0 ];

var thetaLoc;
```



# cubev.js (2/10)

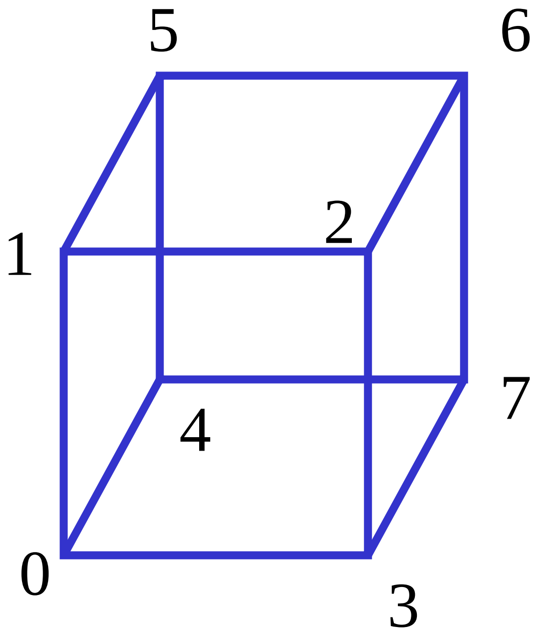
```
var vertices = [
        vec3( -0.5, -0.5,  0.5 ),
        vec3( -0.5,  0.5,  0.5 ),
        vec3(  0.5,  0.5,  0.5 ),
        vec3(  0.5, -0.5,  0.5 ),
        vec3( -0.5, -0.5, -0.5 ),
        vec3( -0.5,  0.5, -0.5 ),
        vec3(  0.5,  0.5, -0.5 ),
        vec3(  0.5, -0.5, -0.5 )
    ];
```

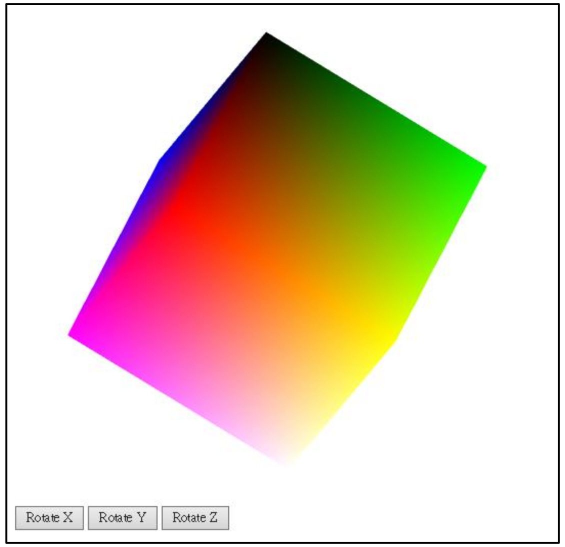


# cubev.js (3/10)

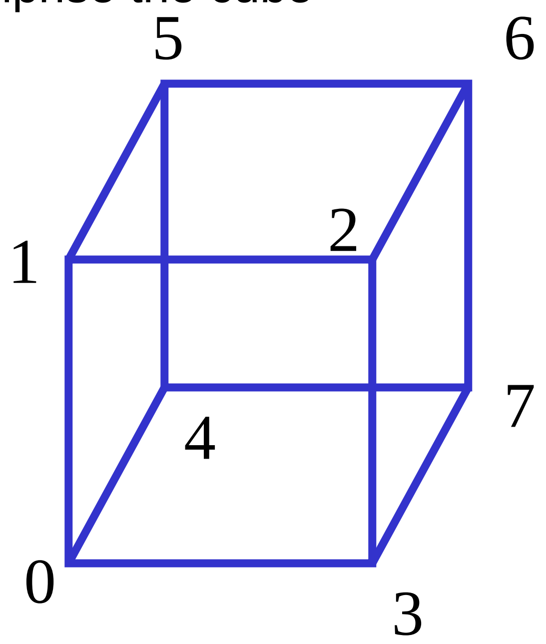
```
var vertexColors = [
      vec4( 0.0, 0.0, 0.0, 1.0 ),  // black
      vec4( 1.0, 0.0, 0.0, 1.0 ),  // red
      vec4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
      vec4( 0.0, 1.0, 0.0, 1.0 ),  // green
      vec4( 0.0, 0.0, 1.0, 1.0 ),  // blue
      vec4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
      vec4( 1.0, 1.0, 1.0, 1.0 ),  // white
      vec4( 0.0, 1.0, 1.0, 1.0 )   // cyan
  ];
```

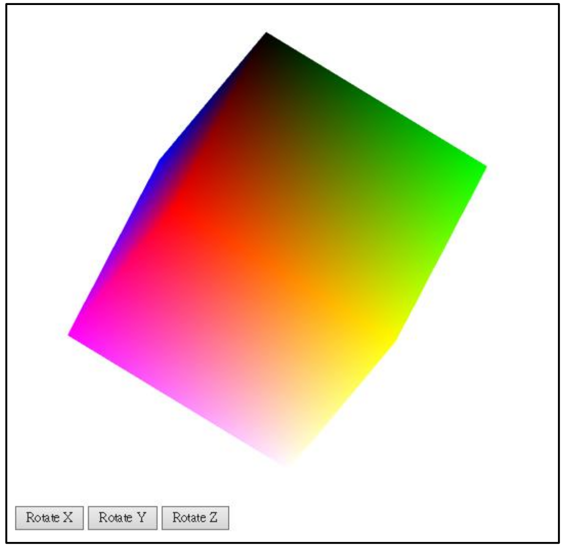


# cubev.js (4/10)

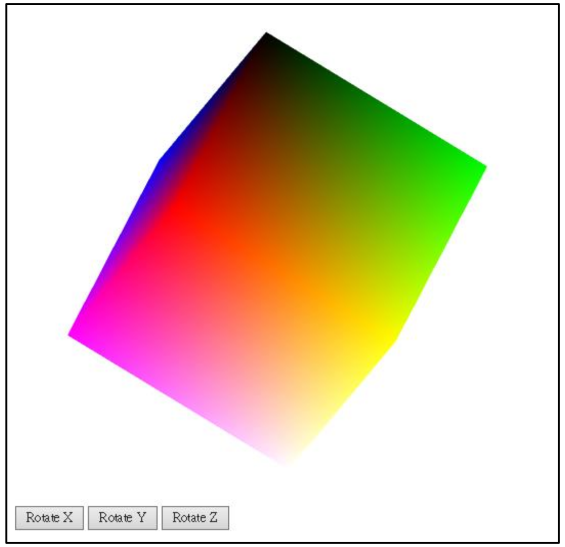
```
// indices of the 12 triangles that comprise the cube

var indices = [
    1, 0, 3,
    3, 2, 1,
    2, 3, 7,
    7, 6, 2,
    3, 0, 4,
    4, 7, 3,
    6, 5, 1,
    1, 2, 6,
    4, 5, 6,
    6, 7, 4,
    5, 4, 0,
    0, 1, 5
];
```



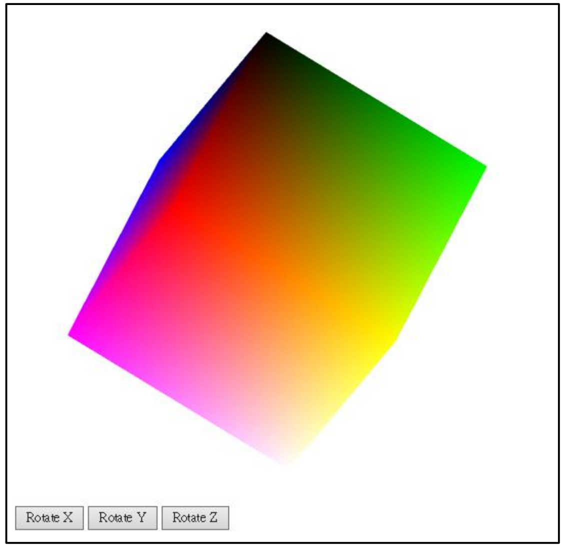



# cubev.js (5/10)

```
window.onload = function init()
{
    canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }

    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

    gl.enable(gl.DEPTH_TEST);;
```



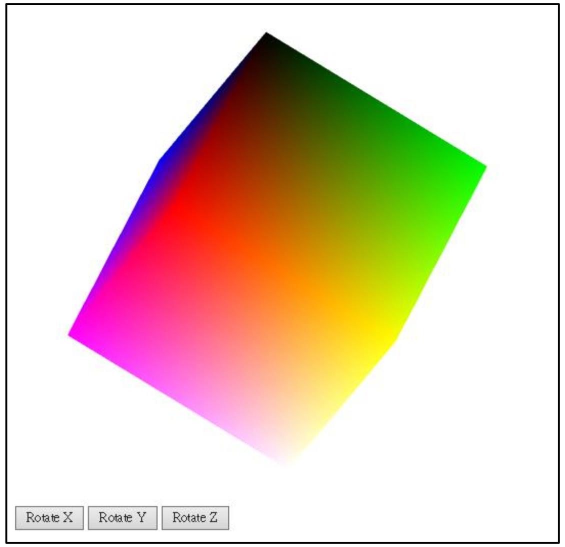
# cubev.js (6/10)




```
//
//  Load shaders and initialize attribute buffers
//
var program = initShaders( gl, "vertex-shader", "fragment-shader" );
gl.useProgram( program );

// array element buffer

var iBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, iBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint8Array(indices), gl.STATIC_DRAW);
```

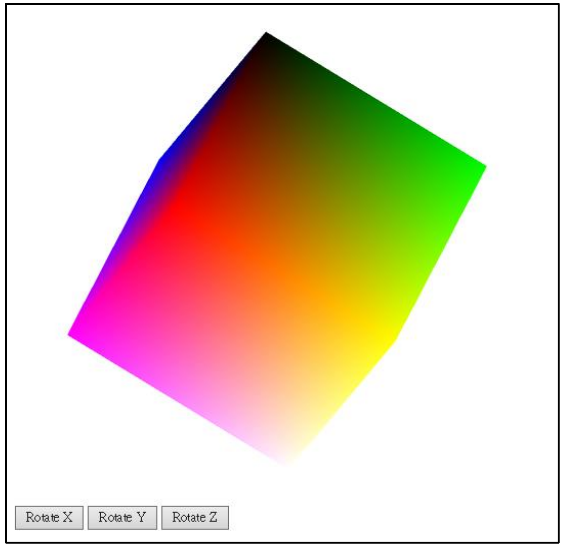
# cubev.js (7/10)

```
// color array atrribute buffer

   var cBuffer = gl.createBuffer();
   gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
   gl.bufferData( gl.ARRAY_BUFFER, flatten(vertexColors), gl.STATIC_DRAW );

   var vColor = gl.getAttribLocation( program, "vColor" );
   gl.vertexAttribPointer( vColor, 4, gl.FLOAT, false, 0, 0 );
   gl.enableVertexAttribArray( vColor );
```



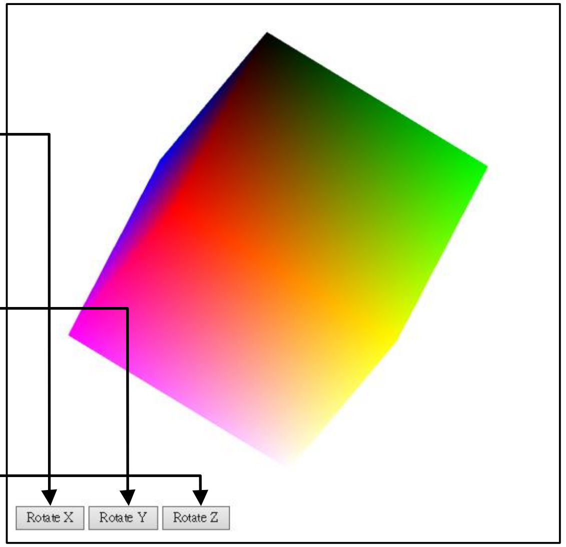
# cubev.js (8/10)

```
// vertex array attribute buffer

var vBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW );

var vPosition = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( vPosition, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );

thetaLoc = gl.getUniformLocation(program, "theta");
```



# cubev.js (9/10)

```
//event listeners for buttons

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


render();
}   // end of window.onload
```



# cubev.js (10/10)

```
function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    theta[axis] += 2.0;
    gl.uniform3fv(thetaLoc, theta);


    gl.drawElements( gl.TRIANGLES, numVertices, gl.UNSIGNED_BYTE, 0 );

    requestAnimFrame( render );
}
```