4. Geometry, Coordinate Systems and Transformations

Lecture Overview

- Recap of Lecture III

- Intro to Linear Algebra
- Geometry
- Representation
- Transformations
- OpenGL Transformations

- Reading:
  - ANG Ch. 4, except 4.11 and 4.12
  - Appendices B and C (if necessary)

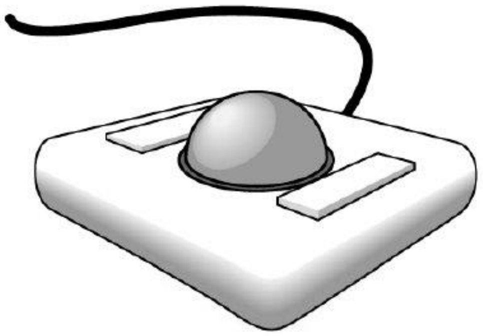
Recap of Lecture III

Input and Interaction

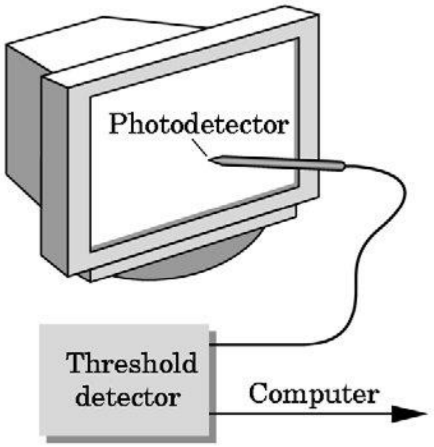
- Introduce the basic input devices
  - Physical Devices
  - Logical Devices
  - Input Modes
- Event-driven input
- Introduce double buffering for smooth animations
- Programming event input with GLUT

Physical Devices

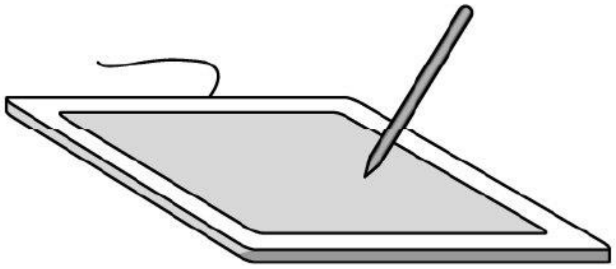
mouse



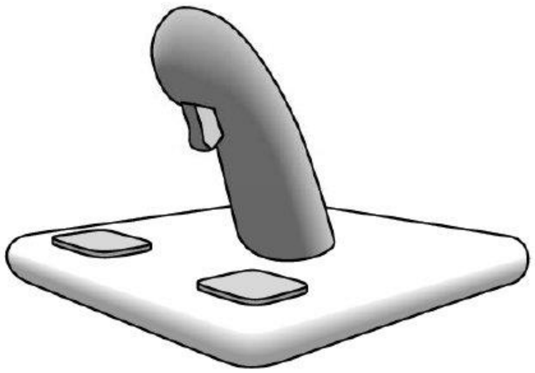
trackball



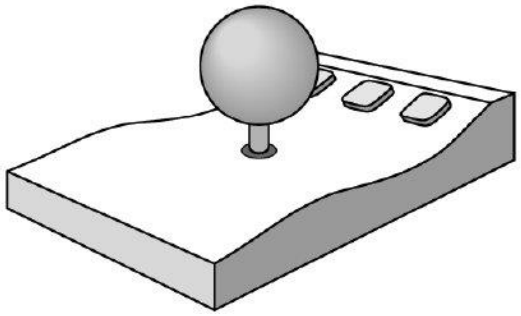
light pen



data tablet



joy stick



space ball

Incremental (Relative) Devices

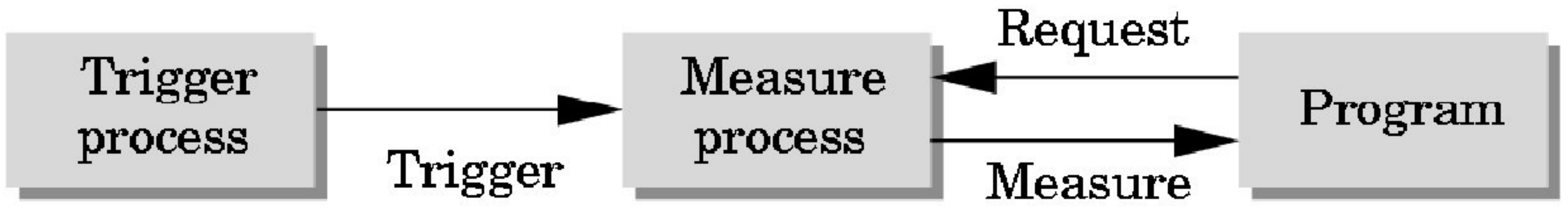
- Devices such as the data tablet return an absolute position directly to the operating system
- Devices such as the mouse, trackball, and joy stick return incremental inputs (or velocities) to the operating system
  - Must integrate these inputs to obtain an absolute position
    - Rotation of cylinders in mouse
    - Roll of trackball
    - Difficult to obtain absolute position
    - Can get variable sensitivity (joysticks)

Graphical Logical Devices

- Graphical input is more varied than input to standard programs which is usually numbers, characters, or bits
- Two older APIs (GKS, PHIGS) defined six types of logical input
  - Locator: return a position
  - Pick: return ID of an object
  - Keyboard or String: return strings of characters
  - Stroke: return array of positions
  - Valuator: return floating point number (widgets: slidebars)
  - Choice: return one of n items (widgets: menus, buttons)

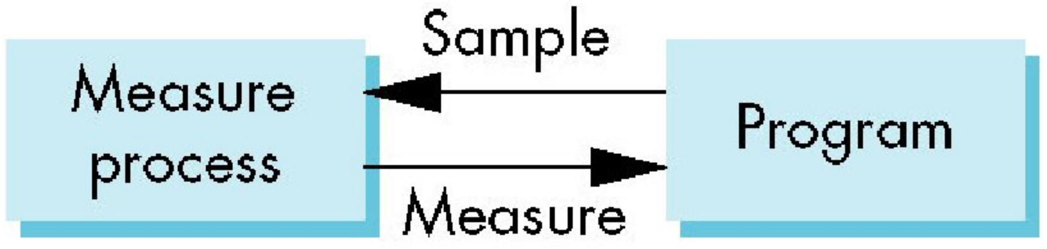
Request Mode

- Input provided to program only when user triggers the device
- Typical of keyboard input
  - Can erase (backspace), edit, correct until enter (return) key (the trigger) is depressed



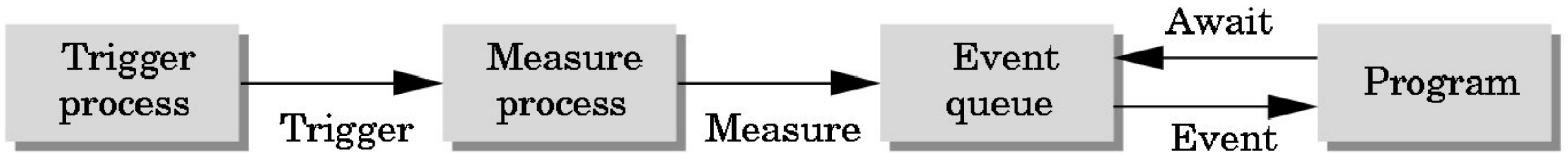
Sample Mode

- Input is immediate, no trigger necessary
- Use must have positioned pointing device or entered data using keyboard before the function is called



Event Mode

- Most systems have more than one input device, each of which can be triggered at an arbitrary time by a user
- Each trigger generates an event whose measure is put in an event queue which can be examined by the user program



GLUT callbacks

GLUT recognizes a subset of the events recognized by any particular window system (Windows, X, Macintosh)

– **glutDisplayFunc**
– **glutMouseFunc**
– **glutReshapeFunc**
– **glutKeyboardFunc**
– **glutIdleFunc**
– **glutMotionFunc, glutPassiveMotionFunc**

GLUT Event Loop

- Recall that the last line in **main.c** for a program using GLUT must be

  glutMainLoop();

which puts the program in an infinite event loop

- In each pass through the event loop, GLUT
  - looks at the events in the queue
  - for each event in the queue, GLUT executes the appropriate callback function if one is defined
  - if no callback is defined for the event, the event is ignored

Double Buffering

- Instead of one color buffer, we use two
  - Front Buffer: one that is displayed but not written to
  - Back Buffer: one that is written to but not displayed
- Program then requests a double buffer in main.c
  - **glutInitDisplayMode(GL_RGB | GL_DOUBLE)**
  - At the end of the display callback buffers are swapped

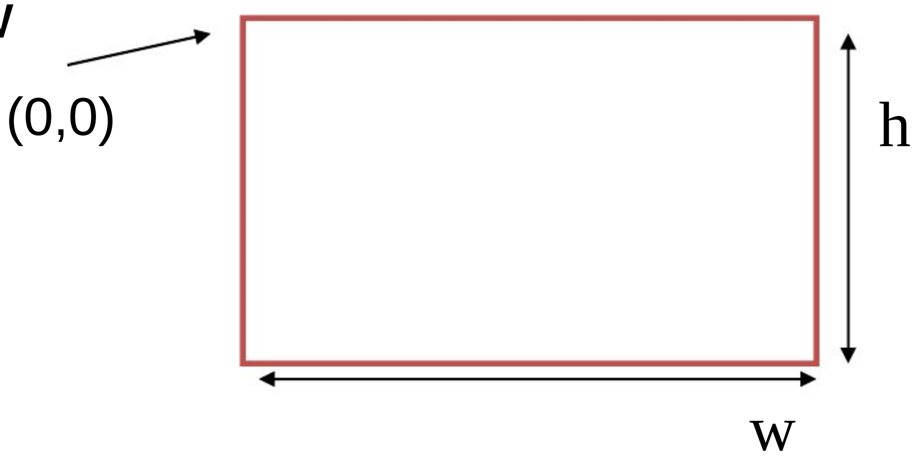
```
void mydisplay()
{
        glClear(GL_COLOR_BUFFER_BIT|….)
.
/* draw graphics here */
.
        glutSwapBuffers()
}
```

Working with Callbacks

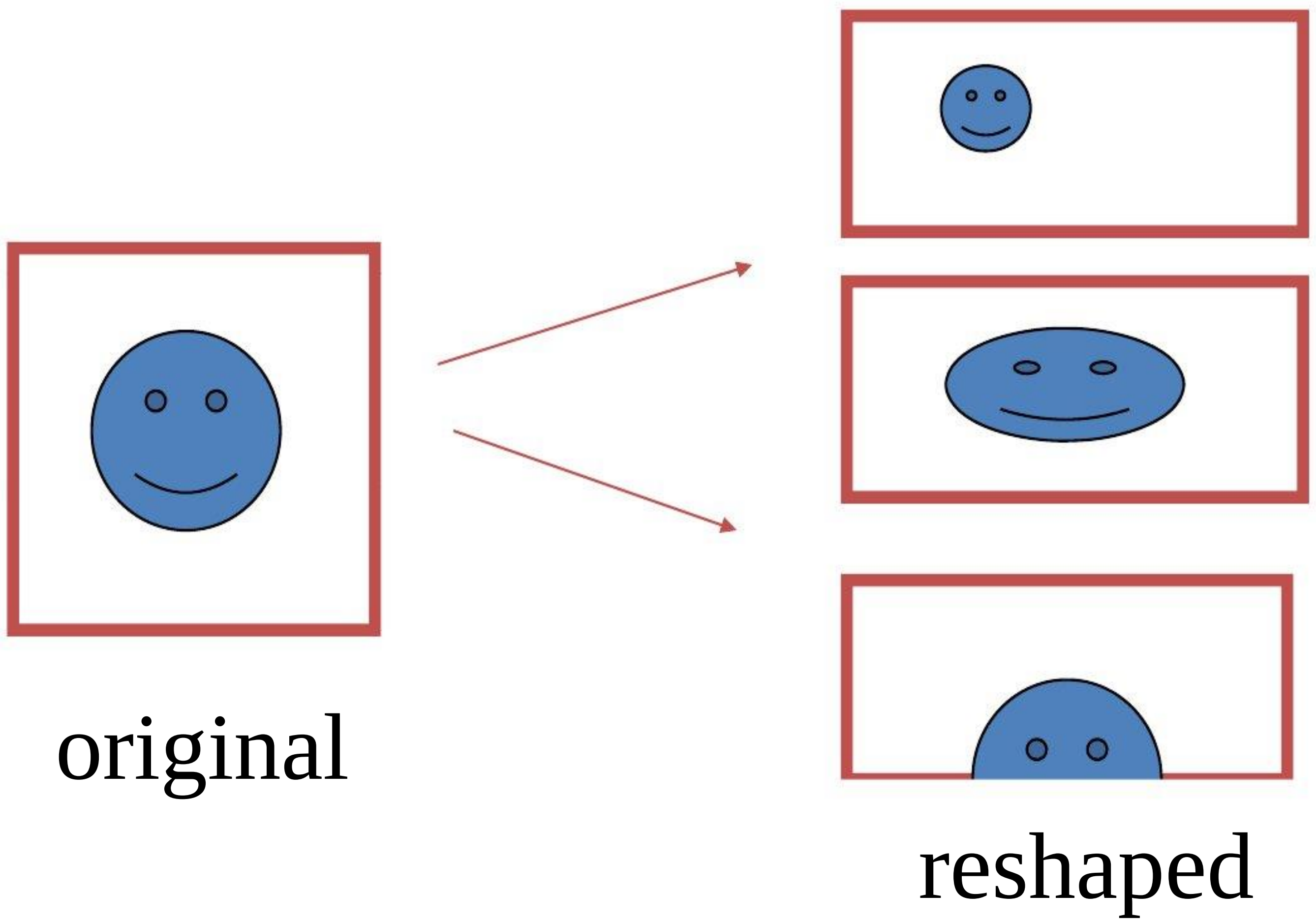
- Learn to build interactive programs using GLUT callbacks
  - Mouse
  - Keyboard
  - Reshape
- Introduce menus in GLUT

Positioning

- The position in the screen window is usually measured in pixels with the origin at the top-left corner
  - Consequence of refresh done from top to bottom
- OpenGL uses a world coordinate system with origin at the bottom left
  - Must invert y coordinate returned by callback by height of window
  - y = h - y;



Reshape possiblities



Better Interactive Programs

- Learn to build more sophisticated interactive programs using
  - Picking
    - Select objects from the display
    - Three methods
  - Rubberbanding
    - Interactive drawing of lines and rectangles
  - Display Lists
    - Retained mode graphics

Picking

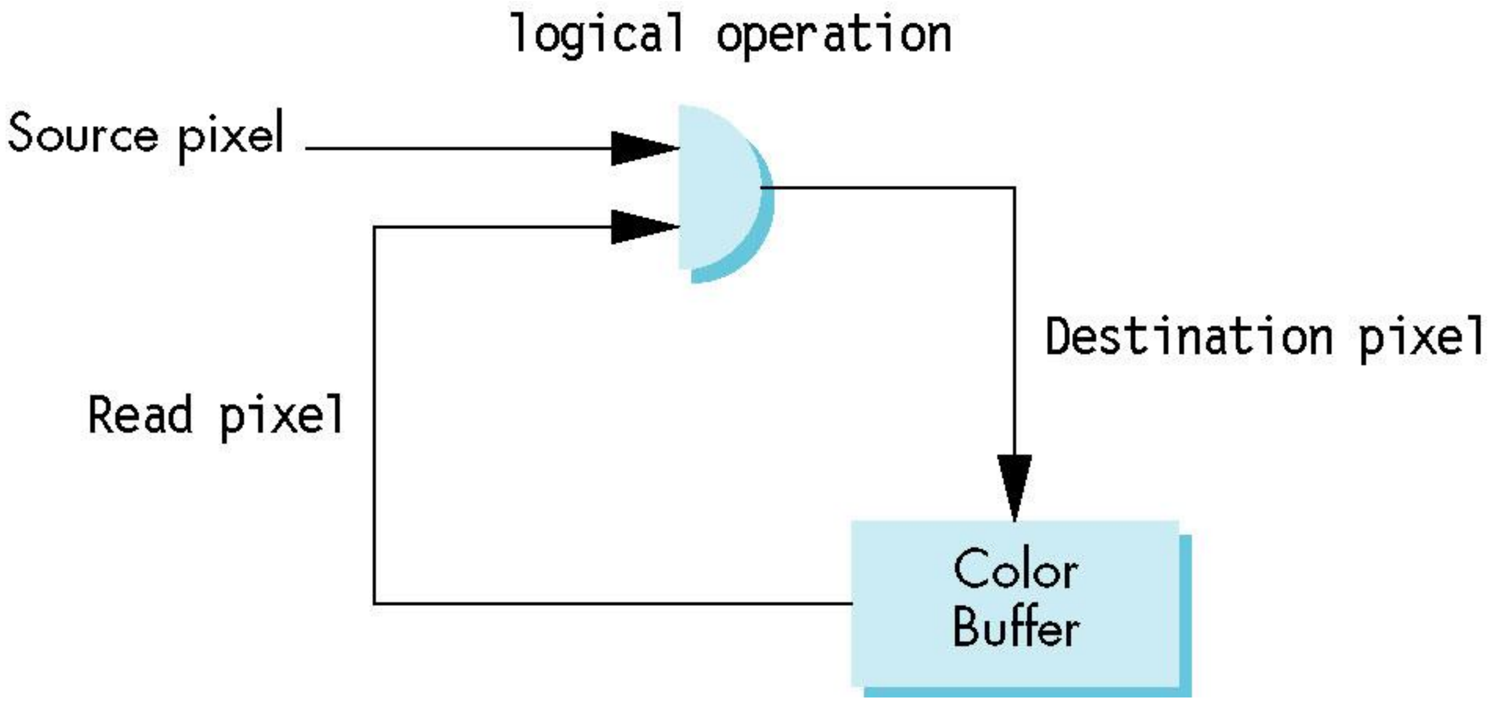
- Identify a user-defined object on the display
- In principle, it should be simple because the mouse gives the position and we should be able to determine to which object(s) a position corresponds
- Practical difficulties
    - Pipeline architecture is feed forward, hard to go from screen back to world
    - Complicated by screen being 2D, world is 3D
    - How close do we have to come to the object to say we selected it?

Three Approaches

- Hit list
  - Most general approach but most difficult to implement
- Use back or some other buffer to store object ids as the objects are rendered
- Rectangular maps
  - Easy to implement for many applications
  - See paint program in text

Writing Modes

application bitwise logical operation

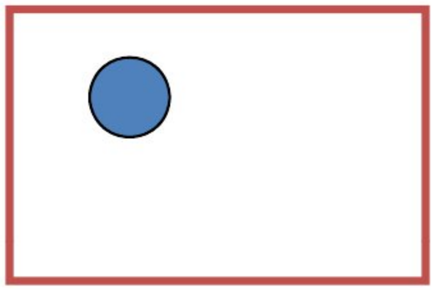


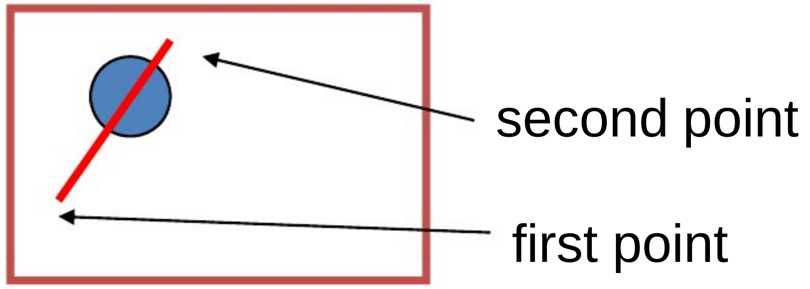
frame buffer

XOR write

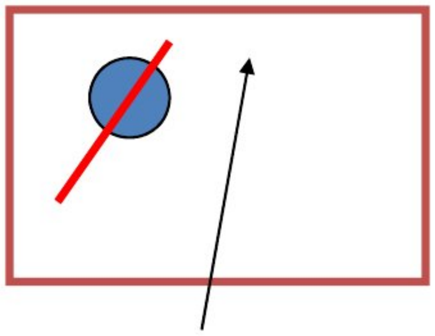
- Usual (default) mode: source replaces destination (d' = s)
  - Cannot write temporary lines this way because we cannot recover what was "under" the line in a fast simple way
- Exclusive OR mode (XOR))((d' = d $\oplus$ s)
  - x $\oplus$ y $\oplus$ x =y
  - Hence, if we use XOR mode to write a line, we can draw it a second time and line is erased!

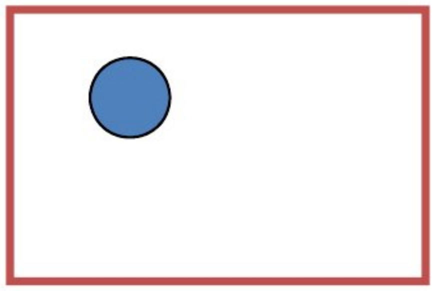
Rubberband Lines

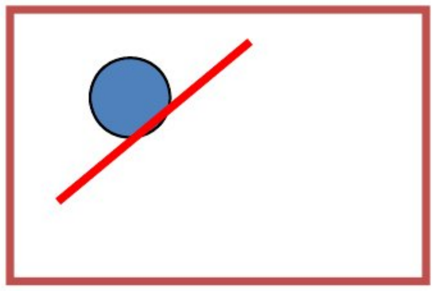

initial display



draw line with mouse in XOR mode


mouse moved to new position


original line redrawn with XOR


new line drawn with XOR

Immediate and Retained Modes

- Recall that in a standard OpenGL program, once an object is rendered there is no memory of it and to redisplay it, we must re-execute the code for it
  - Known as immediate mode graphics
  - Can be especially slow if the objects are complex and must be sent over a network
- Alternative is define objects and keep them in some form that can be redisplayed easily
  - Retained mode graphics
  - Accomplished in OpenGL via display lists

Display Lists

- Conceptually similar to a graphics file
  - Must define (name, create)
  - Add contents
  - Close
- In client-server environment, display list is placed on server
  - Can be redisplayed without sending primitives over network each time

Display List Functions

- Creating a display list

```
GLuint id;

void init()
{
    id = glGenLists( 1 );
    glNewList( id, GL_COMPILE );
    /* other OpenGL routines */
    glEndList();
}
```

returns id of consecutive free lists, equal to the argument (1, here)

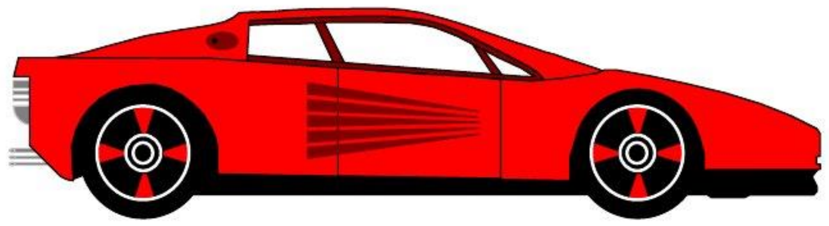
- Call a created list

```
void display()
{
    glCallList( id );
}
```

Hierarchy and Display Lists

- Consider model of a car
  - Create display list for chassis
  - Create display list for wheel

```
glNewList( CAR, GL_COMPILE );
  glCallList( CHASSIS );
  glTranslatef( … );
  glCallList( WHEEL );
  glTranslatef( … );
  glCallList( WHEEL );
        …
glEndList();
```



Calling Display Lists

- Current state determines transformations
- User can change model view or projection matrices between executions of display list
  - E.g. redraw box with increasingly larger clipping rectangle

```
glMatrixMode(GL_PROJECTION);
for(i=0;i<5;i++)
{
            glLoadIdentity();
            glOrtho2D(-2.0*i, 2.0*i,-2.0*i, -2.0*i);
            glCallList(BOX);
}
```

Display Lists and State

- Most OpenGL functions can be put in display lists
- State changes made inside a display list persist after the display list is executed
- Can avoid unexpected results by using **glPushAttrib** and **glPushMatrix** upon entering a display list and **glPopAttrib** and **glPopMatrix** before exiting

Intro to Linear Algebra

Slides by Olga Sorkine

See also ANG Appendices B and C

Vector space

- Informal definition:
  - $V \neq \emptyset$ (a non-empty set of vectors)
  - $\mathbf{v}, \mathbf{w} \in V \Rightarrow \mathbf{v} + \mathbf{w} \in V$ (closed under addition)
  - $\mathbf{v} \in V$, α is scalar $\Rightarrow \alpha\, \mathbf{v} \in V$ (closed under multiplication by scalar)

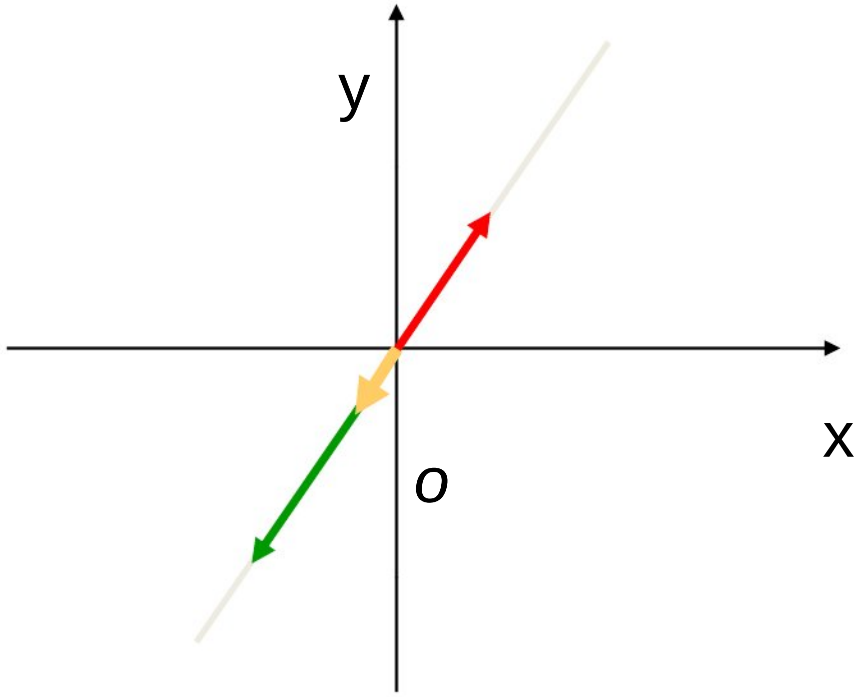
- Formal definition includes axioms about associativity and distributivity of the + and · operators.

- $0 \in V$ always!

O. Sorkine, 2006

Subspace - example

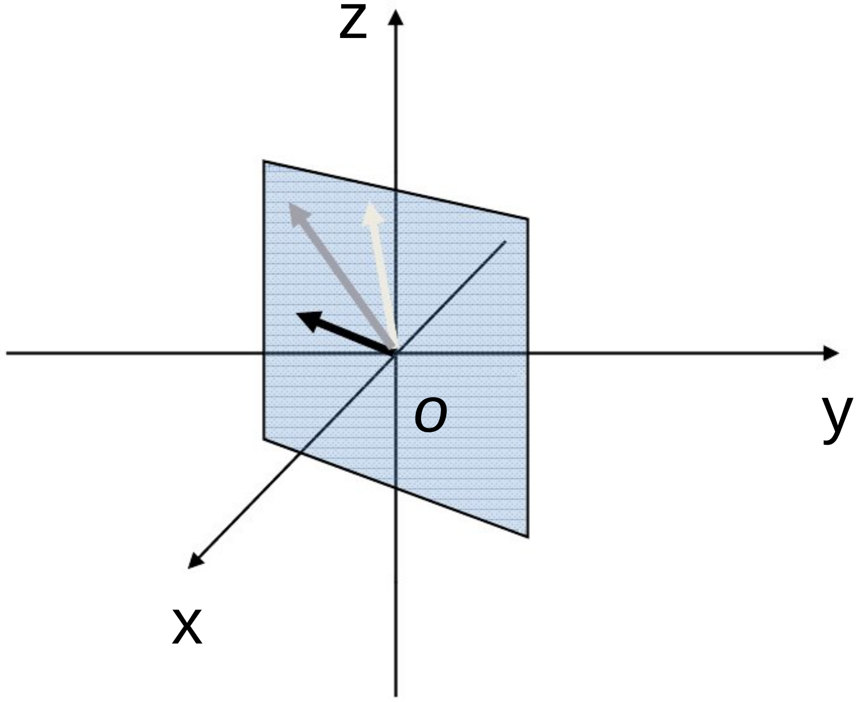
- Let l be a 2D line though the origin
- $L = \{\mathrm{p} - O \mid \mathbf{p} \in l\}$ is a linear subspace of R^2



O. Sorkine, 2006

Subspace - example

- Let π be a plane through the origin in 3D
- $V = \{\mathrm{p} - O \mid \mathbf{p} \in \pi\}$ is a linear subspace of R^3



O. Sorkine, 2006

Linear independence

- The vectors $\{v_1, v_2, \ldots, v_k\}$ are a linearly independent set if:

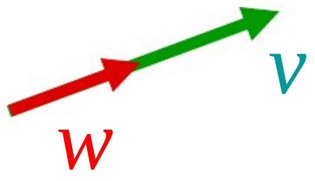
$$\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_k v_k = 0 \Leftrightarrow \alpha_i = 0 \forall i$$

- It means that none of the vectors can be obtained as a linear combination of the others.

O. Sorkine, 2006

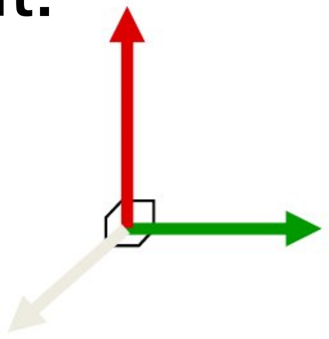
Linear independence - example

- Parallel vectors are always dependent:



$$v = 2.4w \Rightarrow v + (-2.4w) = 0$$

- Orthogonal vectors are always independent.



O. Sorkine, 2006

Basis of V

- $\{v_1, v_2, \ldots, v_n\}$ are linearly independent
- $\{v_1, v_2, \ldots, v_n\}$ span the whole vector space V:

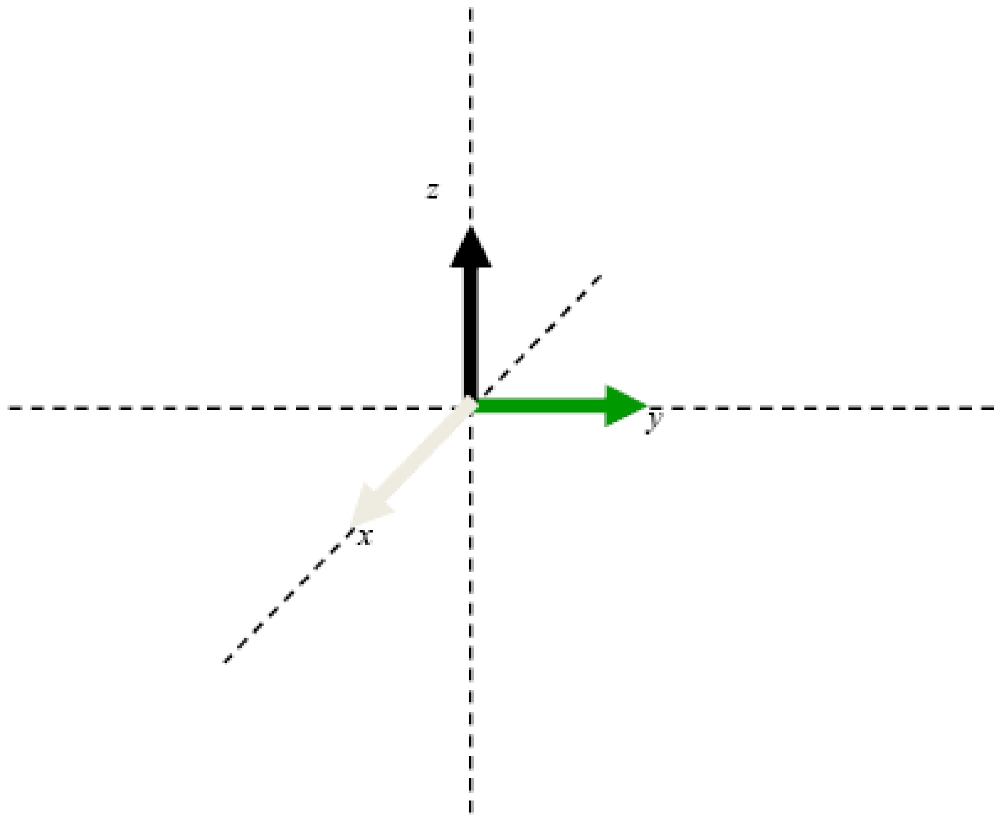
$$V = \{\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n \mid \alpha_i \text{ is scalar}\}$$

- Any vector in V is a unique linear combination of the basis.
- The number of basis vectors is called the dimension of V.

O. Sorkine, 2006

Basis - example

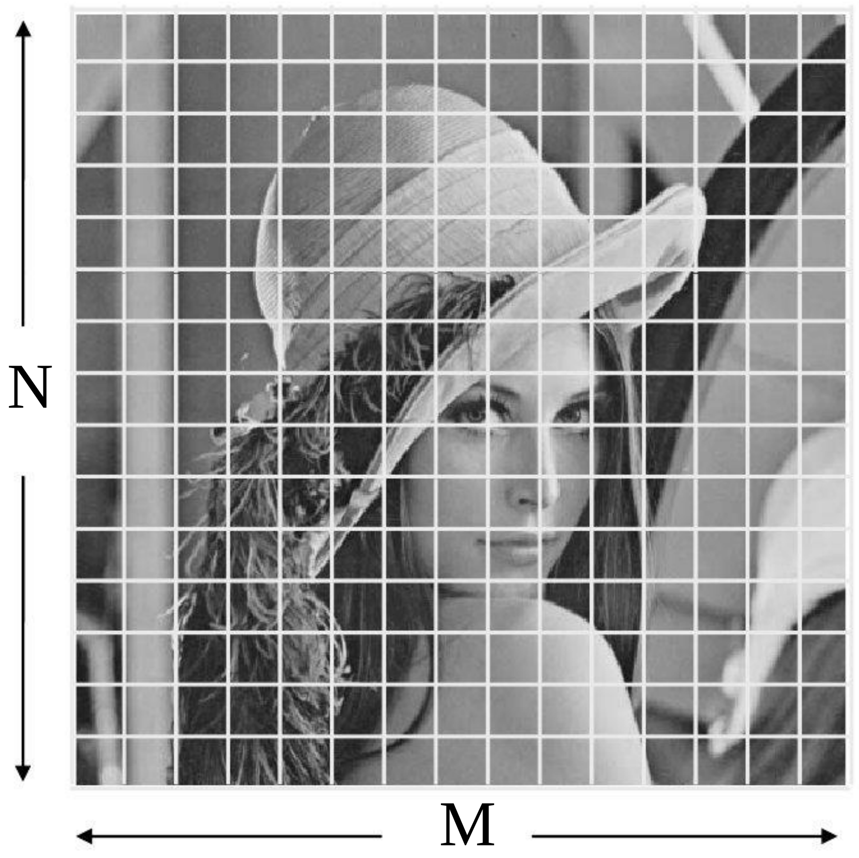
- The standard basis of R^3 - three unit orthogonal vectors $\hat{x}, \hat{y}, \hat{z}$: (sometimes called i, j, k or $\hat{e}_1, \hat{e}_2, \hat{e}_3$)



O. Sorkine, 2006

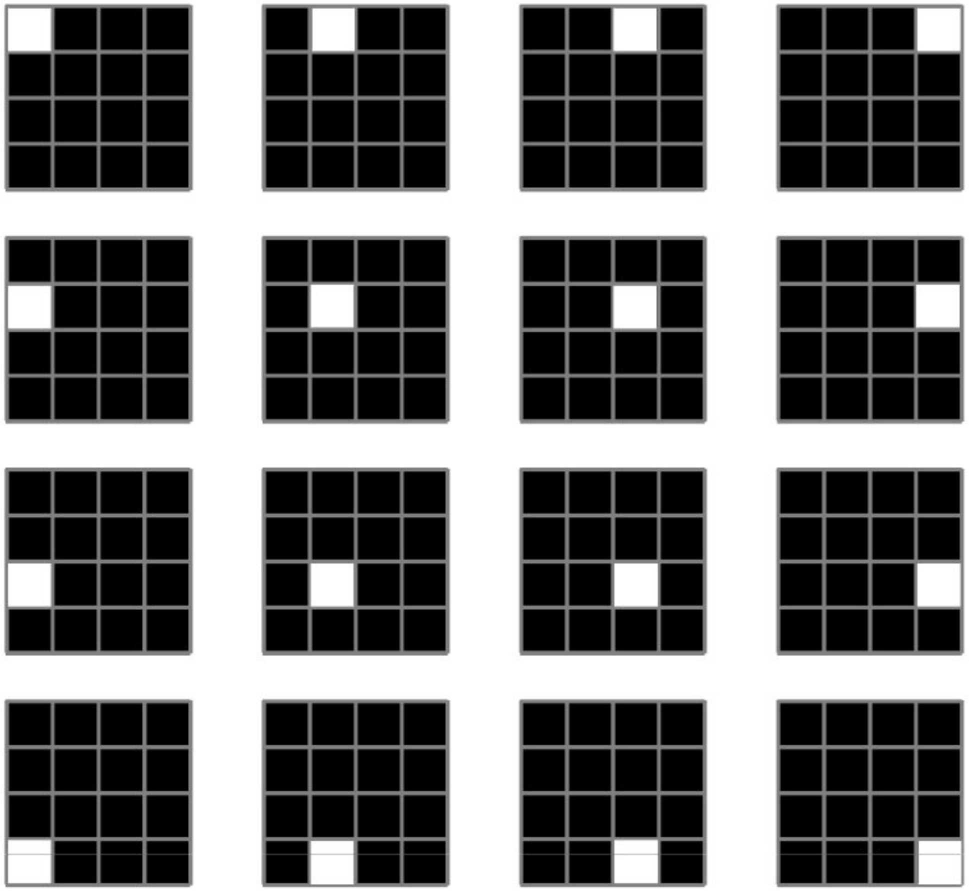
Basis – another example

- Grayscale N x M images:
  - Each pixel has value between 0 (black) and 1 (white)
  - The image can be interpreted as a vector $\in R^{NM}$



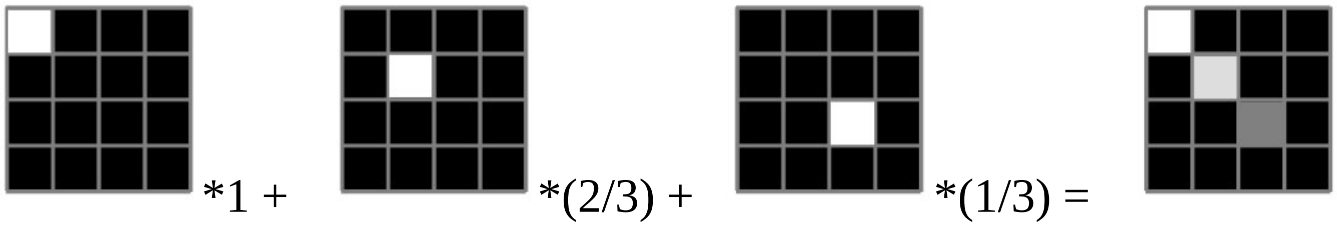
O. Sorkine, 2006

The “standard” basis (4x4)



O. Sorkine, 2006

Linear combinations of the basis



O. Sorkine, 2006

Matrix representation

- Let $\{v_1, v_2, \ldots, v_n\}$ be a basis of V
- Every $v \in V$ has a unique representation

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

- Denote v by the column-vector:

$$v = \begin{pmatrix} \alpha_1 \\ . \\ . \\ . \\ \alpha_n \end{pmatrix}$$

- The basis vectors are therefore denoted:

$$\begin{pmatrix} 1 \\ 0 \\ . \\ . \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ . \\ . \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} 0 \\ 0 \\ . \\ . \\ 1 \end{pmatrix}$$

O. Sorkine, 2006

Linear operators

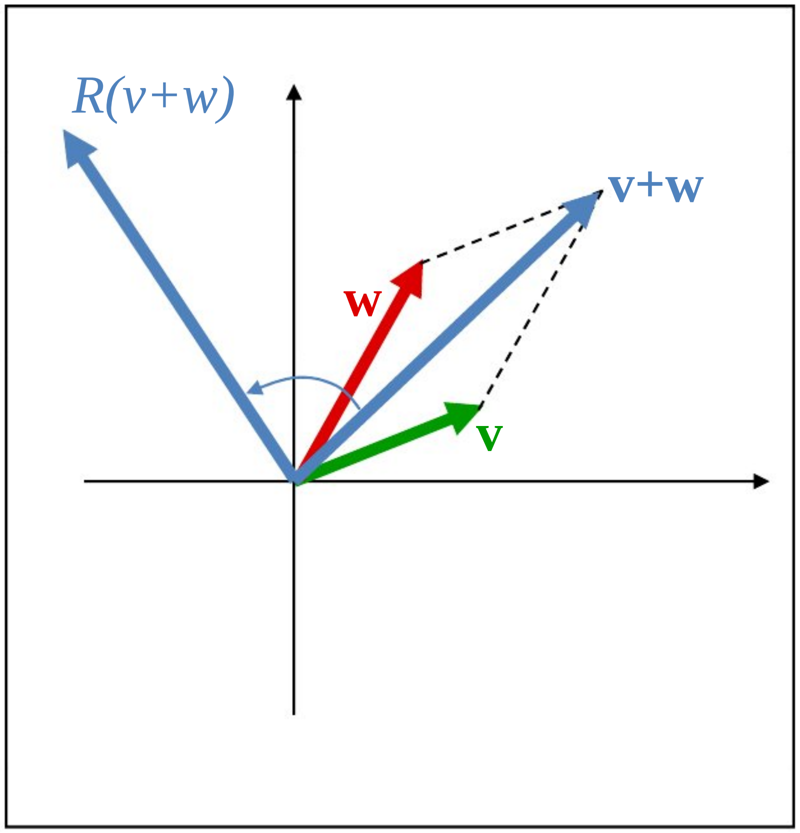
- $A : V \rightarrow W$ is called linear operator if:
  - $A(v + \mathbf{w}) = A(v) + A(w)$
  - $A(\alpha\ \mathbf{v}) = \alpha\ A(v)$

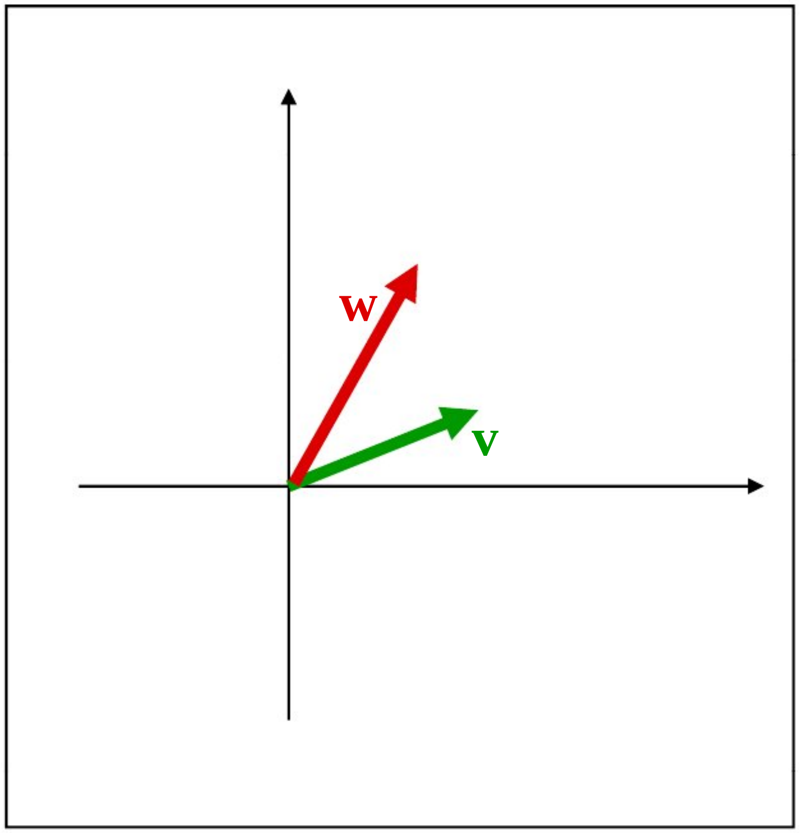
- In particular, $A(0) = 0$
- Linear operators we know:
  - Scaling
  - Rotation, reflection
  - Translation is not linear - moves the origin

O. Sorkine, 2006

Linear operators - illustration

- Rotation is a linear operator:

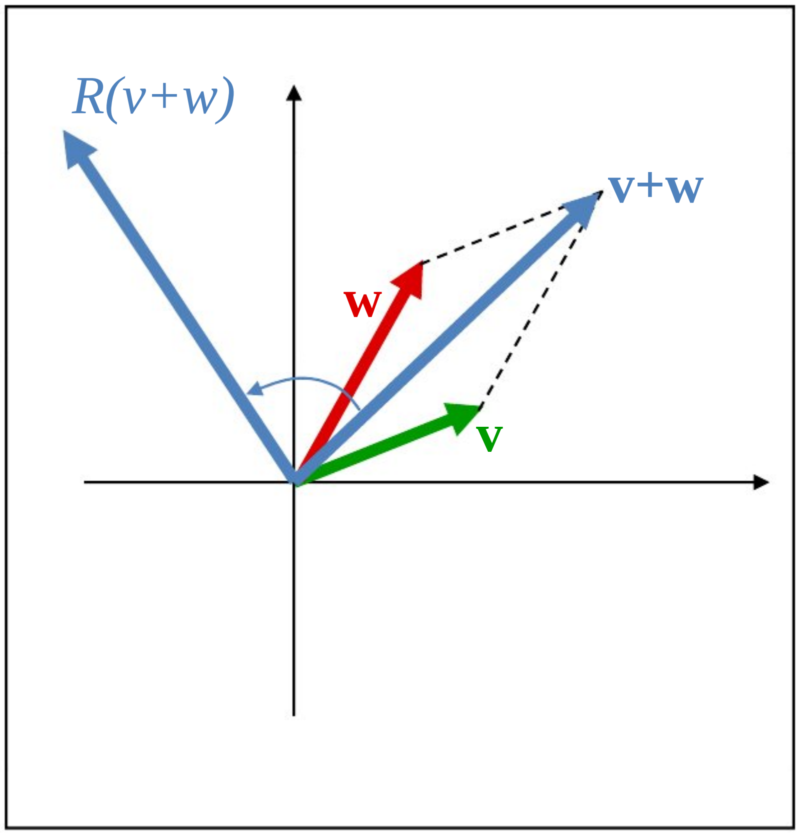


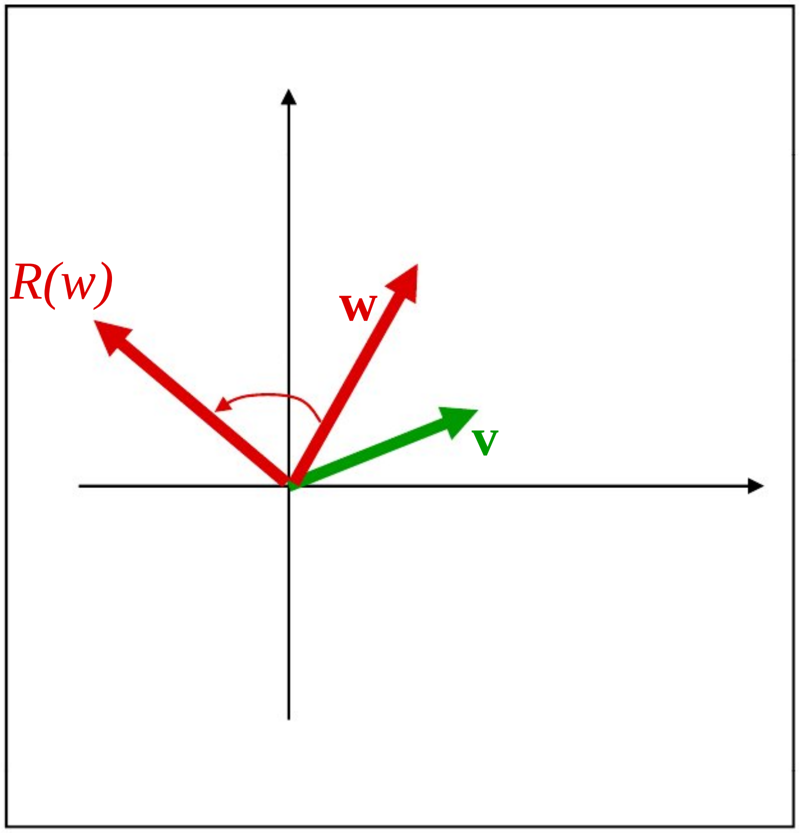


O. Sorkine, 2006

Linear operators - illustration

- Rotation is a linear operator:

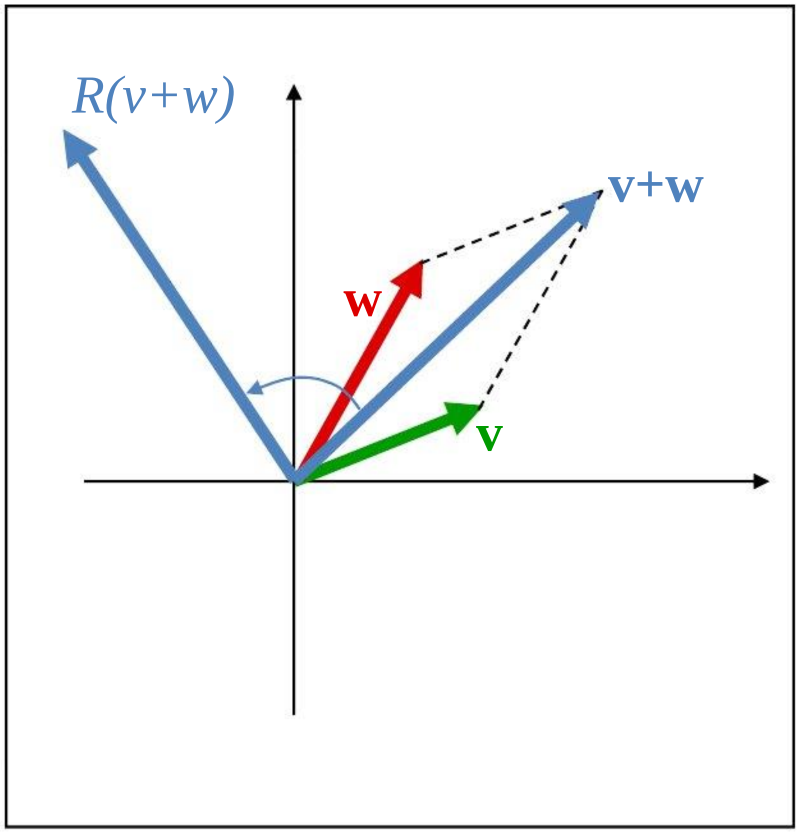


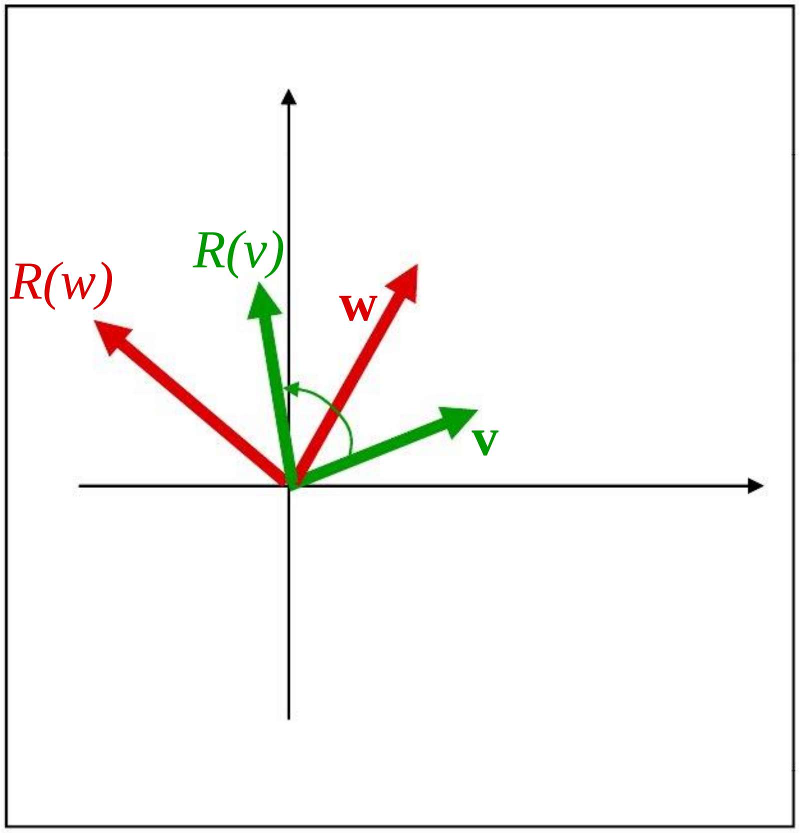


O. Sorkine, 2006

Linear operators - illustration

- Rotation is a linear operator:

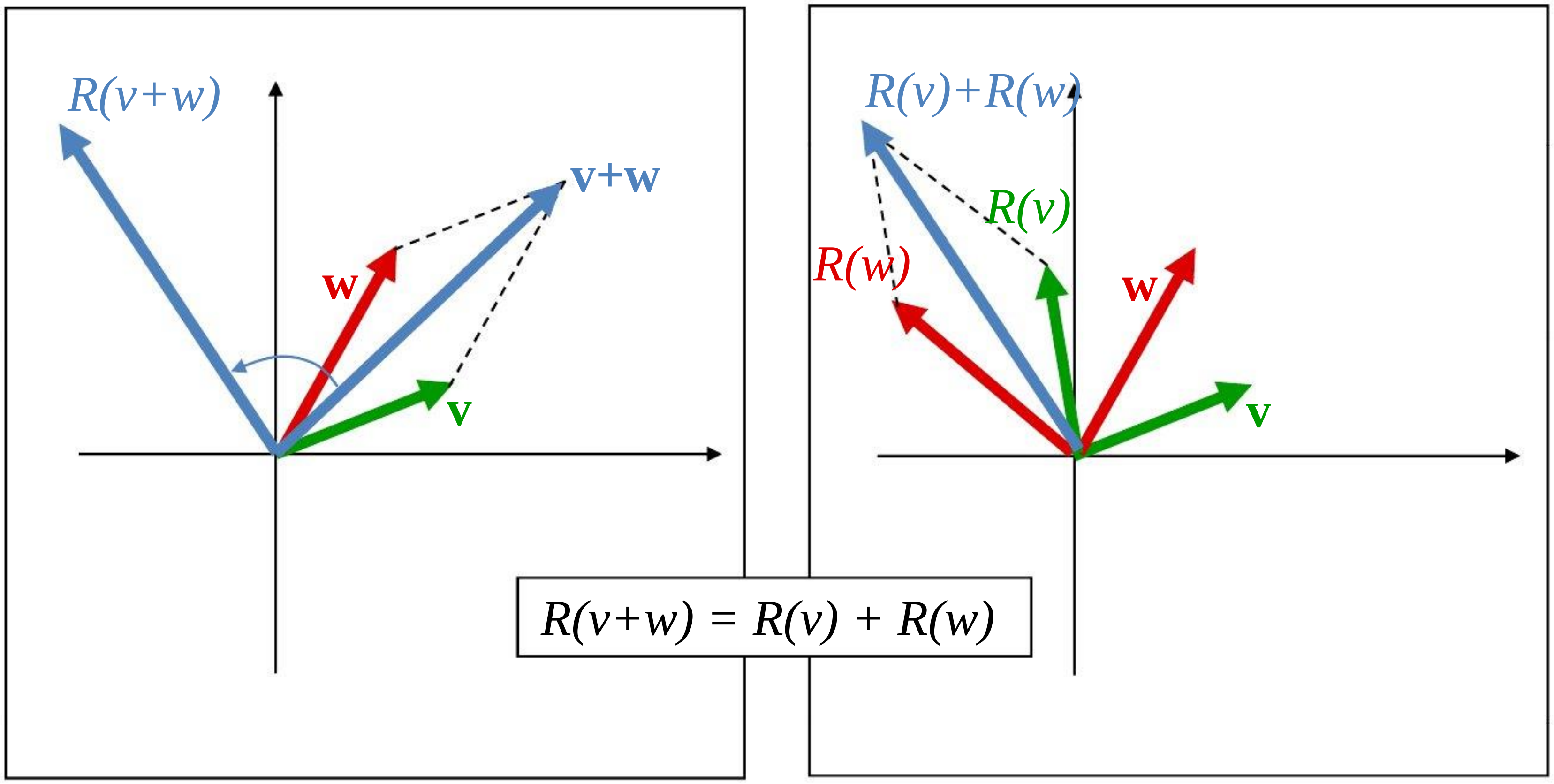




O. Sorkine, 2006

Linear operators - illustration

- Rotation is a linear operator:



$R(v+w) = R(v) + R(w)$

O. Sorkine, 2006

Matrix representation of linear operators

- Look at $A(v_1), A(v_2), \ldots, A(v_n)$ where $\{v_1, v_2, \ldots, v_n\}$ is a basis.
- For all other vectors: $v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$

  $A(v) = \alpha_1 A(v_1) + \alpha_2 A(v_2) + \ldots + \alpha_n A(v_n)$
- So, knowing what A does to the basis is enough
- The matrix representing A is:

$$M_A = \begin{pmatrix} | & | & & | \\ A(\mathbf{v}_1) & A(\mathbf{v}_2) & \cdots & A(\mathbf{v}_n) \\ | & | & & | \end{pmatrix}$$

O. Sorkine, 2006

Matrix representation of linear operators

$$\begin{pmatrix} | & | & & | \\ A(\mathbf{v}_1) & A(\mathbf{v}_2) & \cdots & A(\mathbf{v}_n) \\ | & | & & | \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} | \\ A(\mathbf{v}_1) \\ | \end{pmatrix}$$

$$\begin{pmatrix} | & | & & | \\ A(\mathbf{v}_1) & A(\mathbf{v}_2) & \cdots & A(\mathbf{v}_n) \\ | & | & & | \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} | \\ A(\mathbf{v}_2) \\ | \end{pmatrix}$$

O. Sorkine, 2006

Matrix operations

- Addition, subtraction, scalar multiplication – simple…
- Multiplication of matrix by column vector:

$$\underbrace{\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}}_{\mathbf{b}} = \begin{pmatrix} \sum_i a_{1i} b_i \\ \vdots \\ \sum_i a_{mi} b_i \end{pmatrix} = \begin{pmatrix} < \mathrm{row}_1, \mathbf{b} > \\ \vdots \\ < \mathrm{row}_m, \mathbf{b} > \end{pmatrix}$$

O. Sorkine, 2006

Matrix by vector multiplication

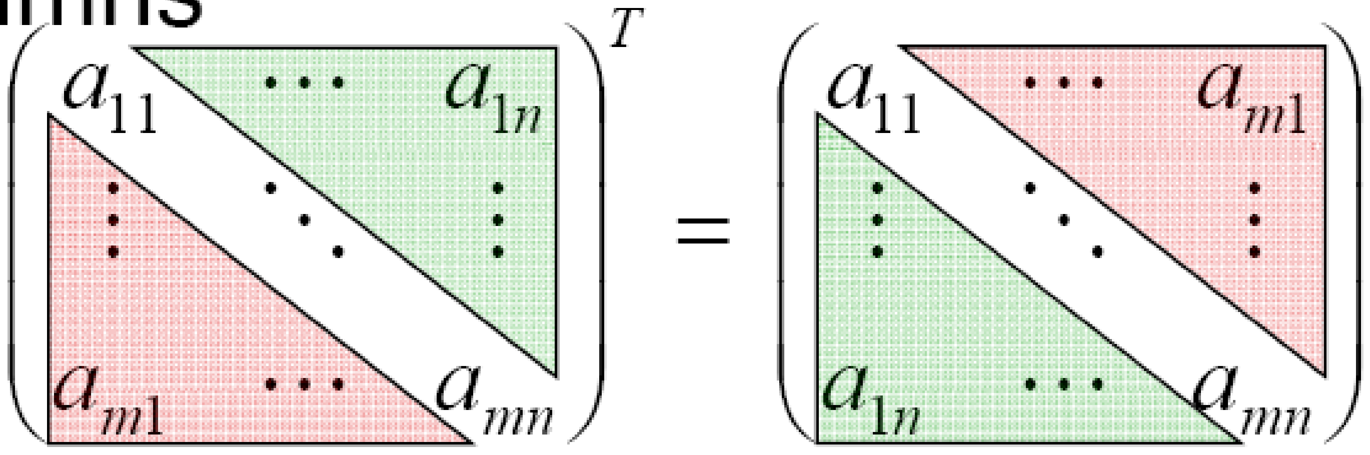
- Sometimes a better way to look at it:
  - *Ab* is a linear combination of *A's* columns!

$$\begin{pmatrix} | & | & & | \\ \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \\ | & | & & | \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = b_1 \begin{pmatrix} | \\ \mathbf{a}_1 \\ | \end{pmatrix} + b_2 \begin{pmatrix} | \\ \mathbf{a}_2 \\ | \end{pmatrix} + \ldots + b_n \begin{pmatrix} | \\ \mathbf{a}_n \\ | \end{pmatrix}$$

O. Sorkine, 2006

Matrix operations

- Transposition: make the rows to be the columns


$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}^T = \begin{pmatrix} a_{11} & \cdots & a_{m1} \\ \vdots & \ddots & \vdots \\ a_{1n} & \cdots & a_{mn} \end{pmatrix}$

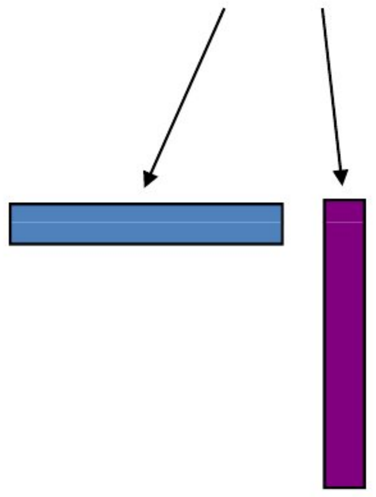
- $(AB)^T = B^T A^T$

O. Sorkine, 2006

Matrix operations

- Inner product can be in matrix form:

$$< v, w > = v^T \, w = w^T \, v$$



O. Sorkine, 2006

Matrix properties

- Matrix A (n x n) is non-singular if $\exists B, AB = BA = I$
- $B = A^{-1}$ is called the inverse of A
- A is non-singular $\Leftrightarrow$ $det\, A \neq 0$

- If A is non-singular then the equation $Ax=b$ has one unique solution for each **b.**
- A is non-singular $\Leftrightarrow$ the rows of A are linearly independent (and so are the columns).

O. Sorkine, 2006

Orthogonal matrices

- Matrix A (n x n) is orthogonal if $A^{-1} = A^T$
- Follows: $AA^T = A^TA = I$
- The rows of A are orthonormal vectors!

Proof:

$$I = A^TA = \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \\ \mathbf{v}_n \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 & \mathbf{v}_2 & & \mathbf{v}_n \end{pmatrix} = \begin{pmatrix} \mathbf{v}_i{}^T\mathbf{v}_j \end{pmatrix} = \begin{pmatrix} \delta_{ij} \end{pmatrix}$$

$$\Rightarrow \langle \mathbf{v}_i, \mathbf{v}_i \rangle = 1 \quad \Rightarrow \quad \|\mathbf{v}_i\| = 1; \quad \langle \mathbf{v}_i, \mathbf{v}_j \rangle = 0$$

Orthogonal operators

- A is orthogonal matrix $\Rightarrow$ A represents a linear operator that preserves inner product (i.e., preserves lengths and angles):

$$< A\mathbf{v}, A\mathbf{w} >= (A\mathbf{v})^T (A\mathbf{w}) = \mathbf{v}^T A^T A\mathbf{w} =$$
$$= \mathbf{v}^T I\,\mathbf{w} = \mathbf{v}^T\mathbf{w} =< \mathbf{v}, \mathbf{w} > .$$

- Therefore, $||Av|| = ||v||$ and $\angle(Av,Aw) = \angle(v,w)$

O. Sorkine, 2006

Orthogonal operators - example

- Rotation by α around the *z-axis* in R^3 :

$$R = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- In fact, any orthogonal 3 x 3 matrix represents a rotation around some axis and/or a reflection
  - $\det A = +1$ rotation only
  - $\det A = -1$ with reflection

O. Sorkine, 2006

Eigenvectors and eigenvalues

- Let A be a square n x n matrix
- $\mathbf{v}$ is eigenvector of A if:
  - $Av = \lambda\mathbf{v}$ (λ is a scalar)
  - $\mathbf{v} \neq 0$
- The scalar λ is called eigenvalue

- $Av = \lambda\mathbf{v} \Rightarrow A(\alpha\mathbf{v}) = \lambda(\alpha\,\mathbf{v}) \Rightarrow \alpha\,\mathbf{v}$ is also eigenvector
- $Av = \lambda\mathbf{v}, Aw = \lambda\mathbf{w} \Rightarrow A(v+w) = \lambda(\mathrm{v}+\mathrm{w})$
- Therefore, eigenvectors of the same λ form a linear subspace.

O. Sorkine, 2006

Finding eigenvalues

- For which λ is there a non-zero solution to $Ax = \lambda\mathbf{x}$?
- $Ax = \lambda\mathbf{x} \Leftrightarrow Ax - \lambda\mathbf{x} = 0 \Leftrightarrow Ax - \lambda Ix = 0 \Leftrightarrow (A-\lambda I)\,\mathbf{x} = 0$
- So, non trivial solution exists $\Leftrightarrow$ $\det(A - \lambda I) = 0$

- $\Delta_A(\lambda) = \det(A - \lambda I)$ is a polynomial of degree n.
- It is called the characteristic polynomial of A.
- The roots of Δ_A are the eigenvalues of A.
- Therefore, there are always at least complex eigenvalues. If n is odd, there is at least one real eigenvalue.

O. Sorkine, 2006

Example of computing Δ_A

$$A = \begin{pmatrix} 1 & 0 & 2 \\ 3 & 0 & -3 \\ -1 & 1 & 4 \end{pmatrix}$$

$$\Delta_A(\lambda) = \det\begin{pmatrix} 1-\lambda & 0 & 2 \\ 3 & -\lambda & -3 \\ -1 & 1 & 4-\lambda \end{pmatrix} =$$

$$= (1-\lambda)(-\lambda(4-\lambda)+3) + 2(3-\lambda) = (1-\lambda)^2(3-\lambda) + 2(3-\lambda) =$$

$$= (3-\lambda)(\lambda^2 - 2\lambda + 3)$$

Cannot be factorized over R
Over C: $(1+i\sqrt{2})(1-i\sqrt{2})$

O. Sorkine, 2006

Computing eigenvectors

- Solve the equation $(A - \lambda I)x = 0$
- We’ll get a subspace of solutions

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

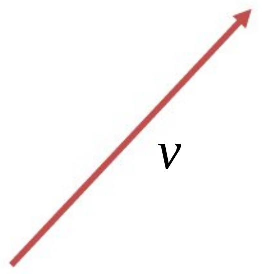
- When we learned simple geometry, most of us started with a Cartesian approach
  - Points were at locations in space p=(x,y,z)
  - We derived results by algebraic manipulations involving these coordinates
- This approach was nonphysical
  - Physically, points exist regardless of the location of an arbitrary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

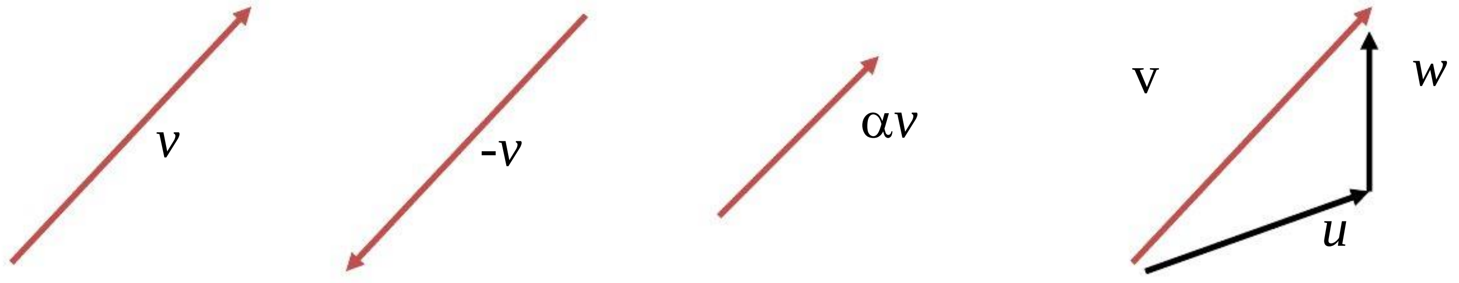
Vectors

- 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

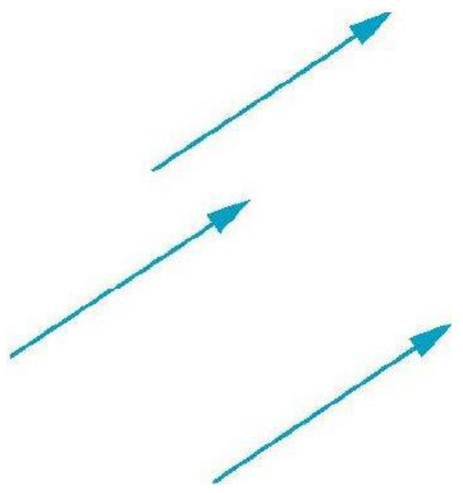
- Mathematical system for manipulating vectors
- Operations
  - Scalar-vector multiplication $u= \quad v$
  - Vector-vector addition: $w=u+v$
- Expressions such as

  $v=u+2w-3r$

Make sense in a vector space

Vectors Lack Position

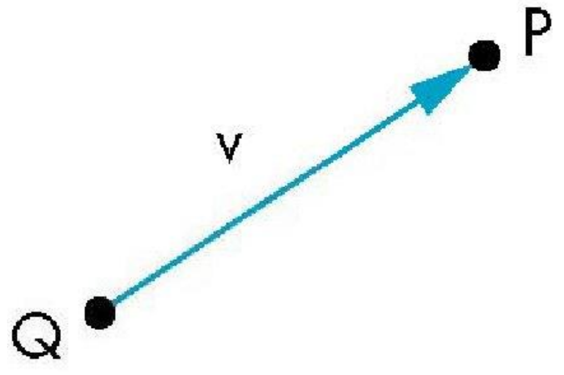
- These vectors are identical
  - Same length and magnitude



- 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



$v = P - Q$

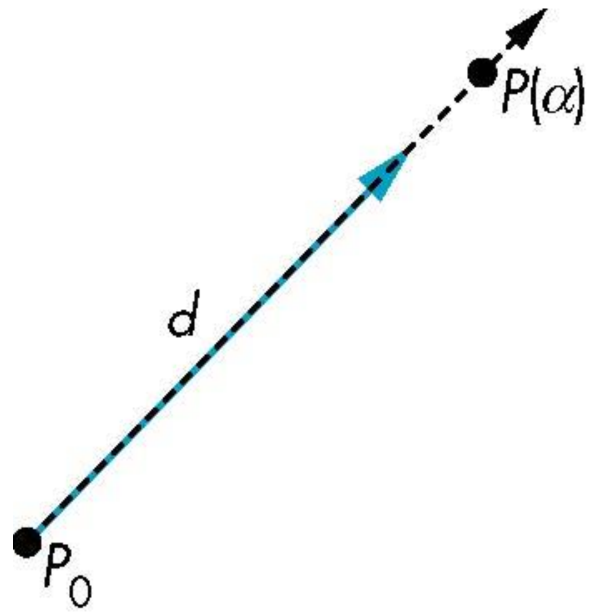
$P = v + Q$

Affine Spaces

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

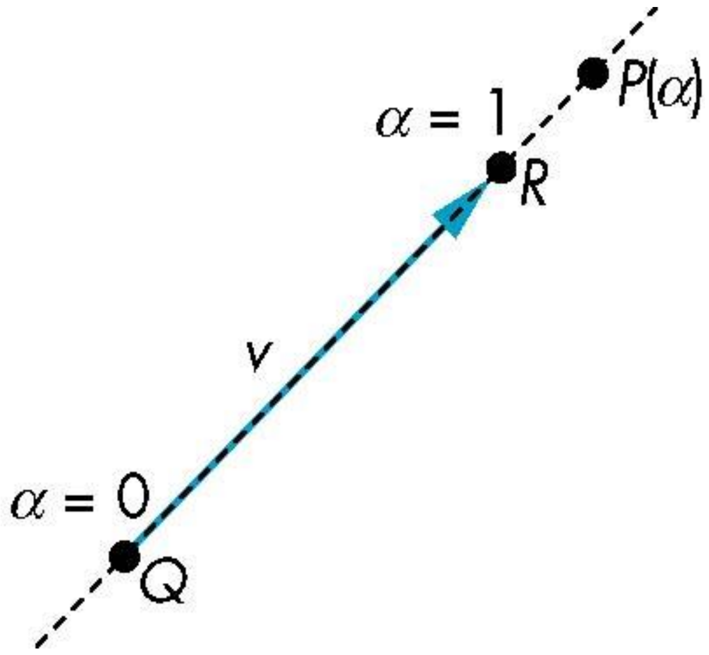
- If $\alpha >= 0$, then $P(\alpha)$ is the *ray* leaving P_0 in the direction **d**

If we use two points to define v, then

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

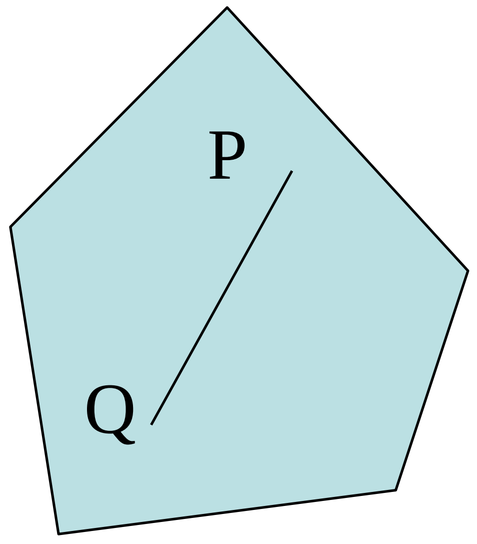
$=\alpha R + (1-\alpha)Q$

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

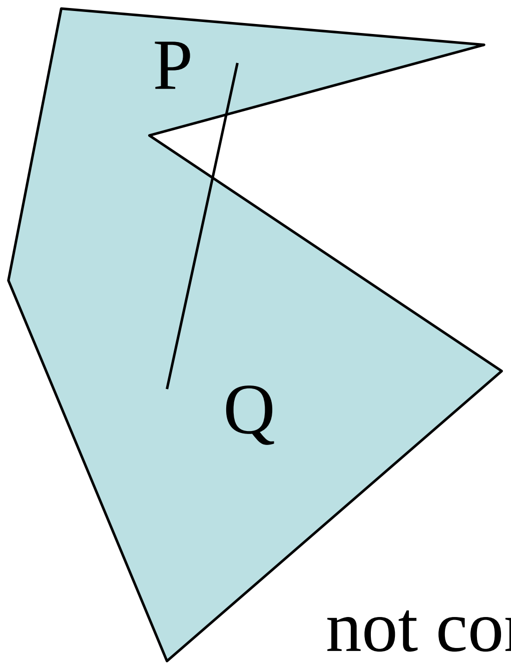


Convexity

- An object is *convex* iff 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"

$P=\alpha_1P_1+\alpha_2P_2+\ldots..+\alpha_nP_n$

Can show by induction that this sum makes sense iff

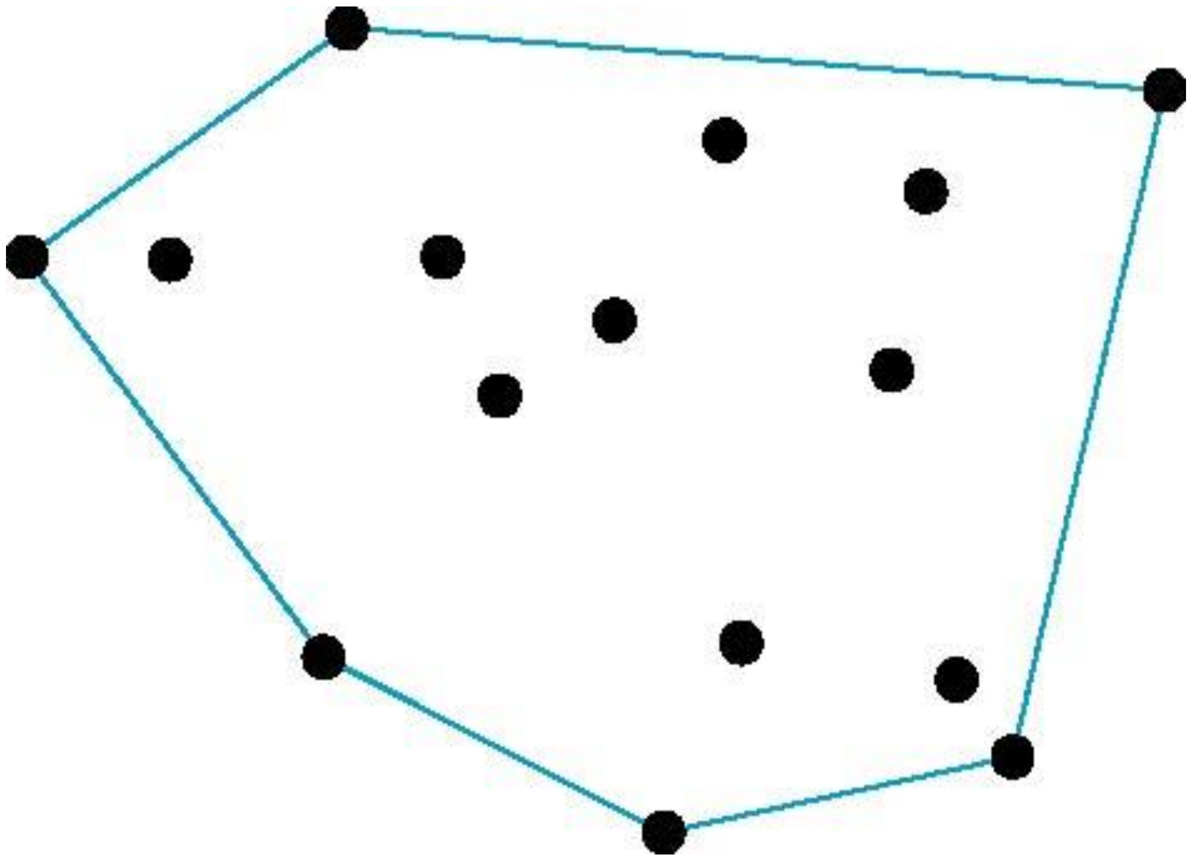
$\alpha_1+\alpha_2+\ldots..\alpha_n=1$

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 $P_1,P_2,\ldots..P_n$

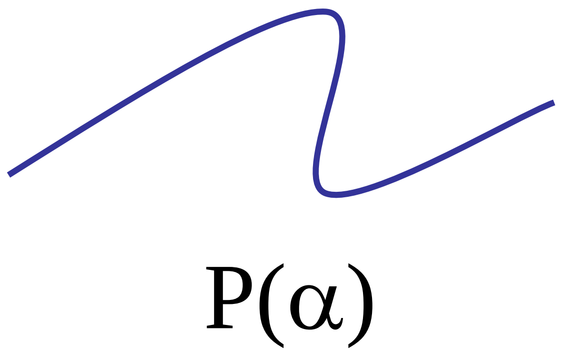
Convex Hull

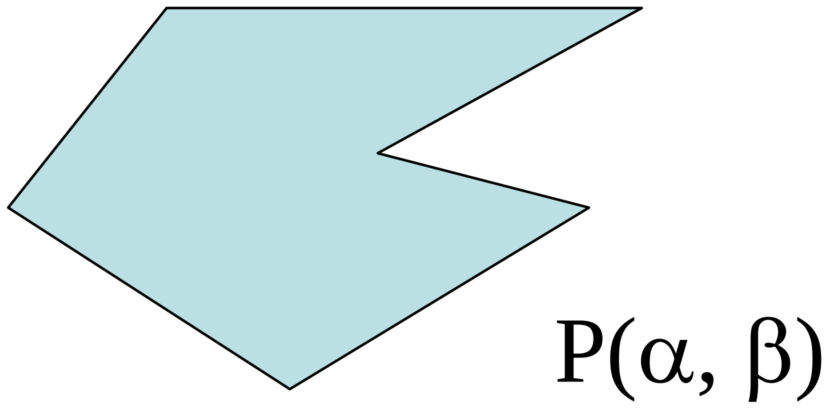
- Smallest convex object containing $P_1, P_2, \ldots\ldots P_n$
- Formed by "shrink wrapping" points



Curves and Surfaces

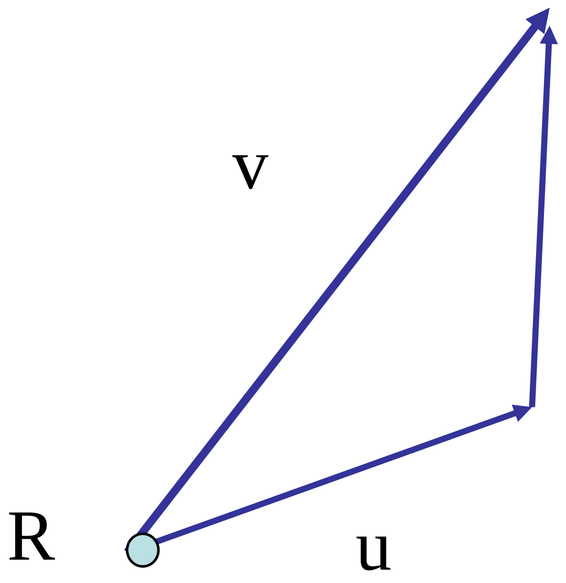
- Curves are one parameter entities of the form $P(\alpha)$ where the function is nonlinear
- Surfaces are formed from two-parameter functions $P(\alpha, \beta)$
  - –Linear functions give planes and polygons

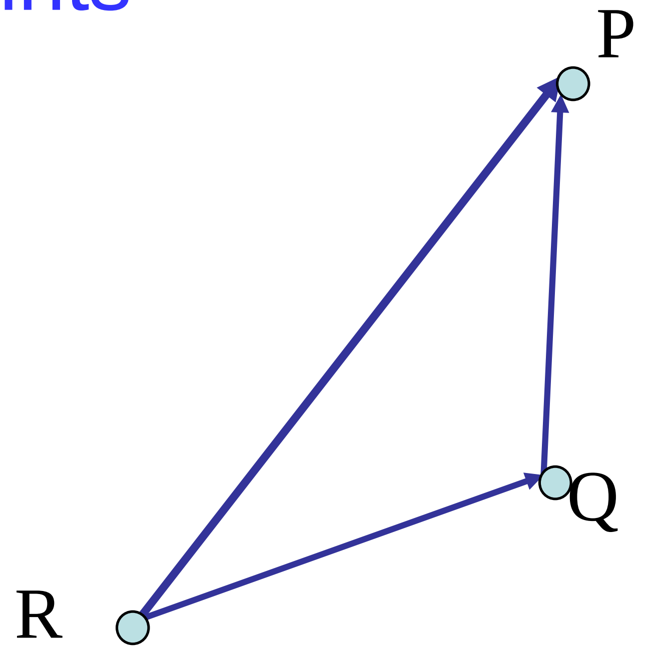




Planes

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

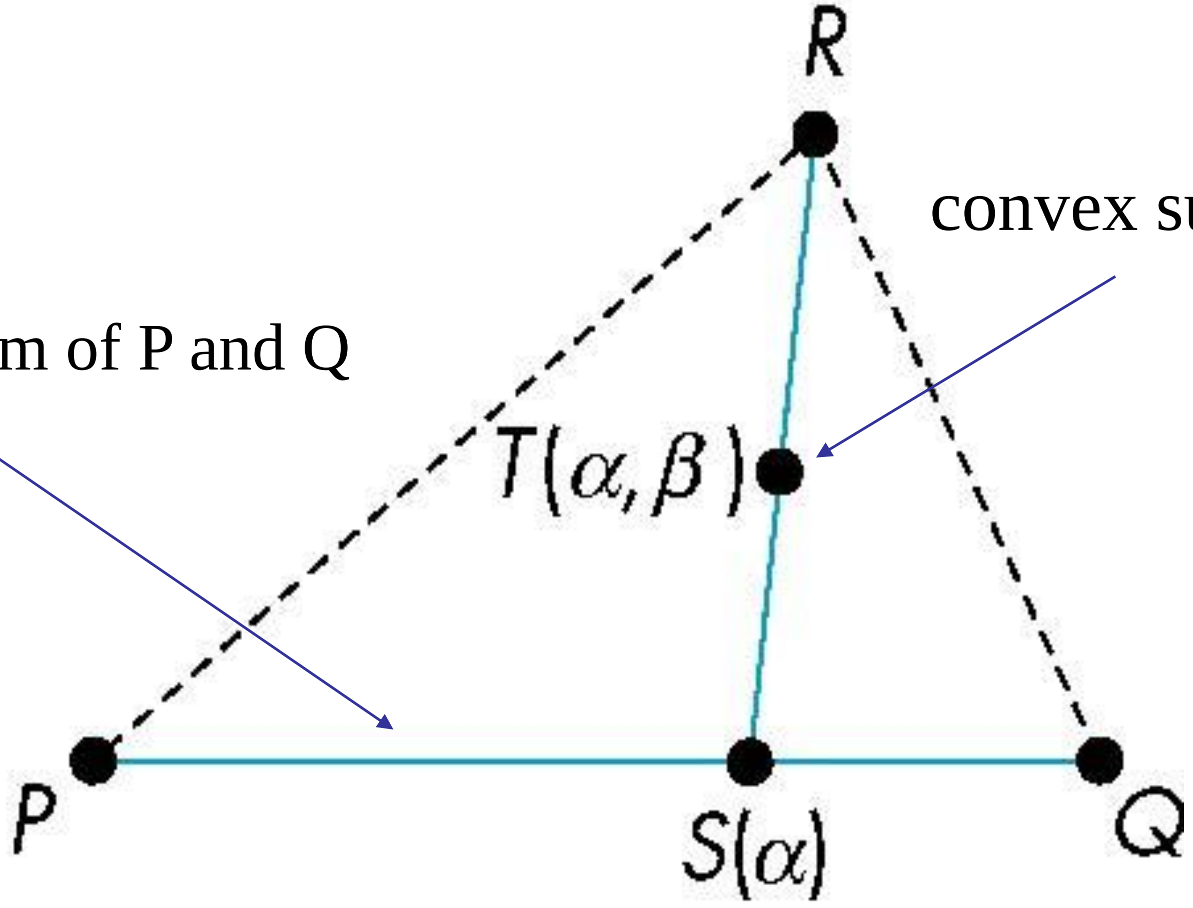




$P(\alpha,\beta)=R+\alpha u+\beta v$

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

Triangles



convex sum of $S(\alpha)$ and R

convex sum of P and Q

for $0 <= \alpha, \beta <= 1$, we get all points in triangle

Barycentric Coordinates

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

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

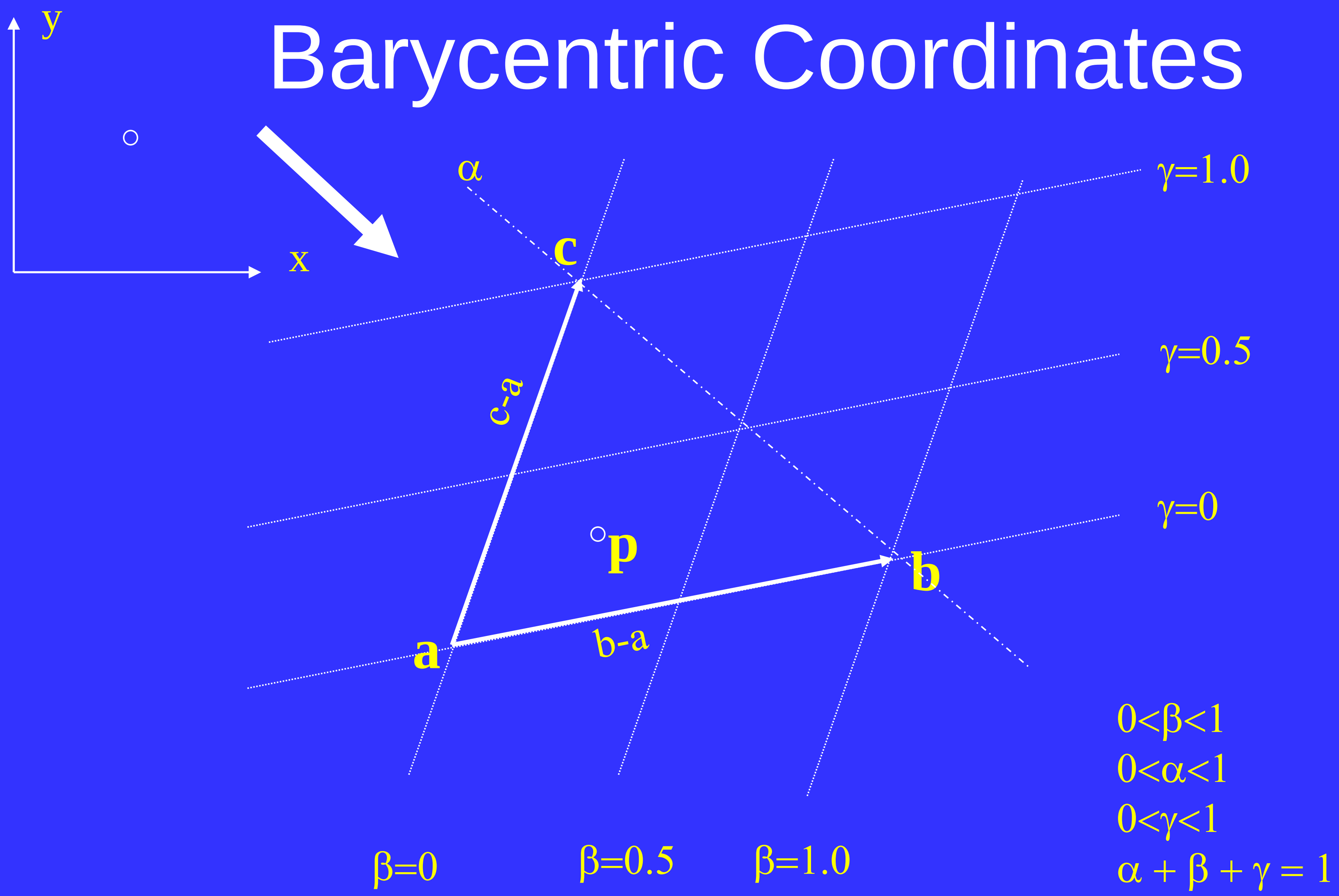
where

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

$\alpha_i >= 0$

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

Barycentric Coordinates



$0<\beta<1$

$0<\alpha<1$

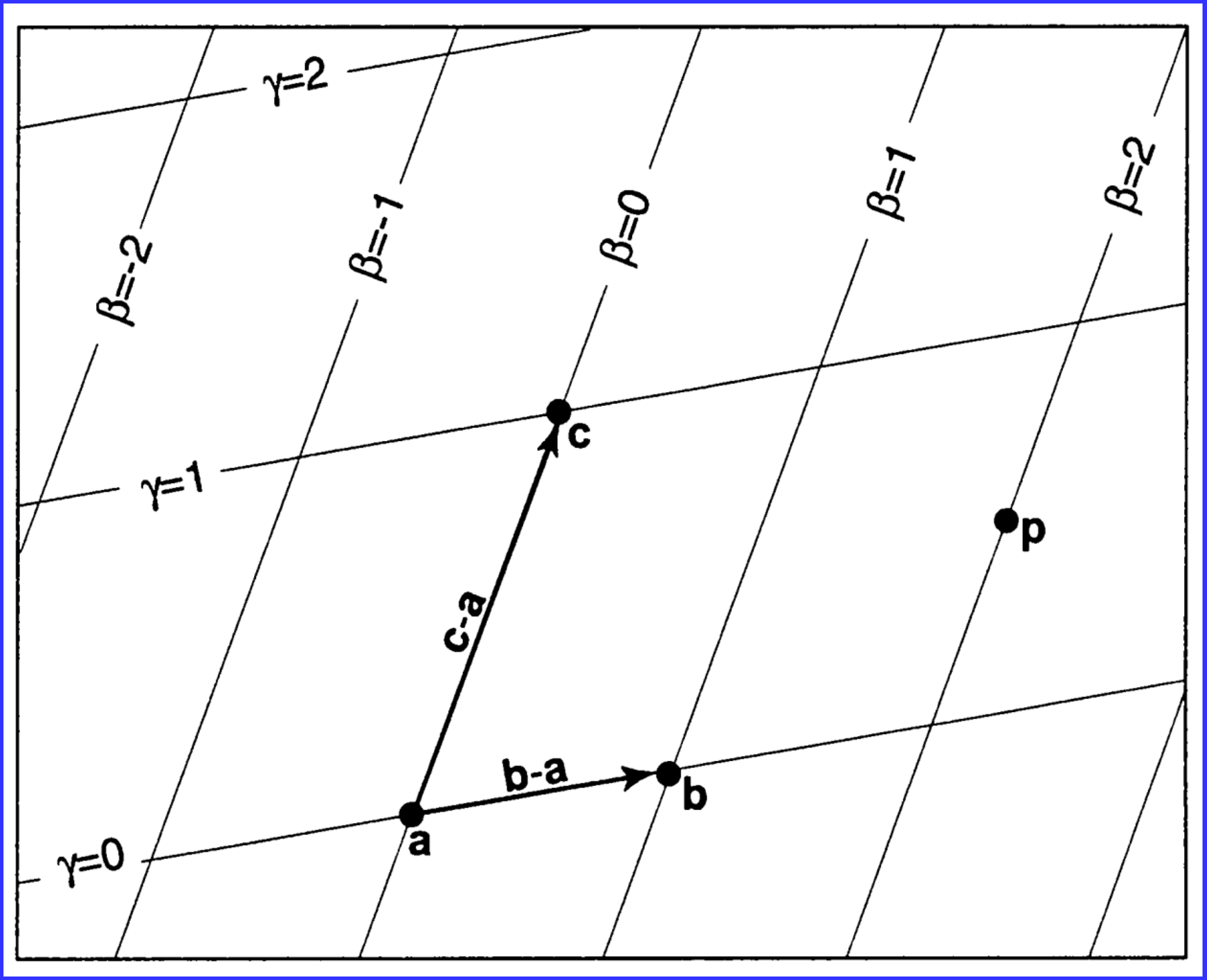
$0<\gamma<1$

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

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

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

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

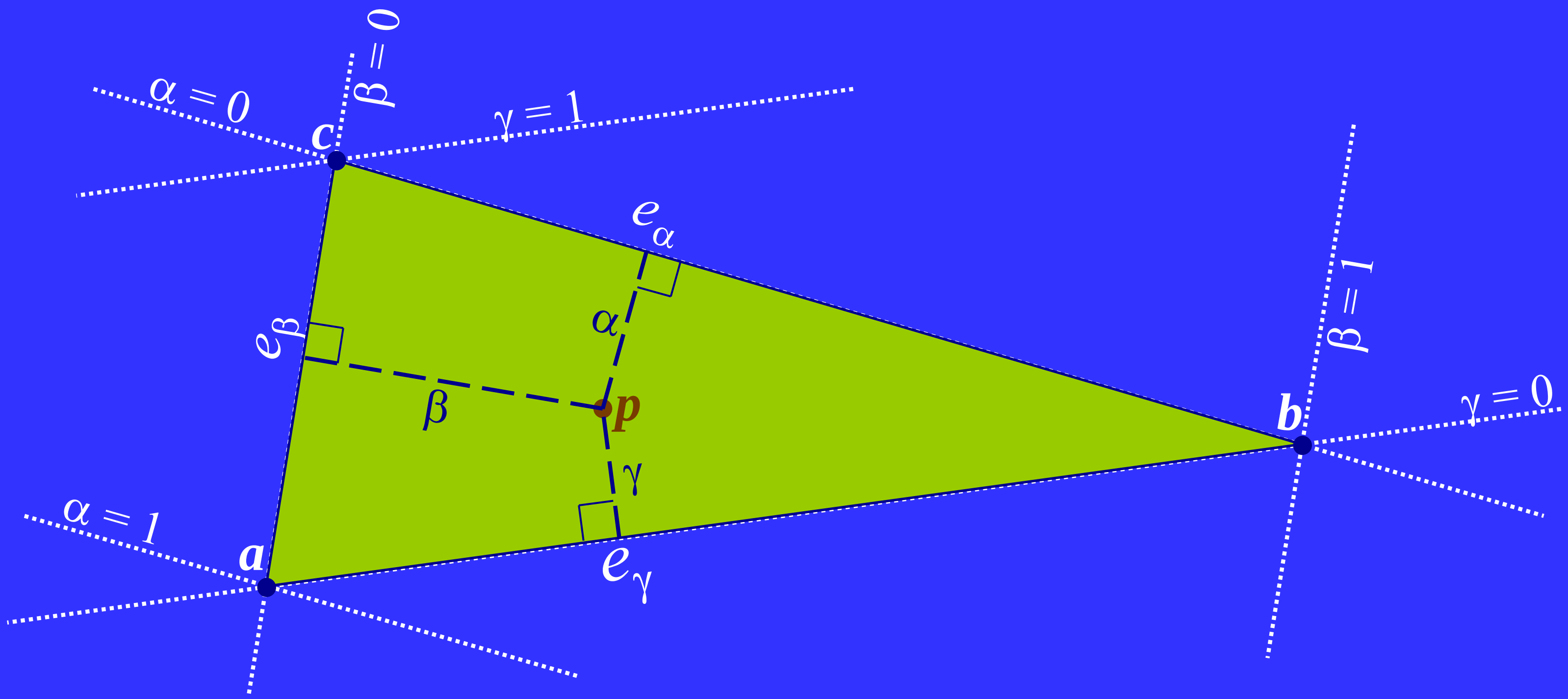
$$p = (1 - \beta - \gamma)\vec{a} + \beta\vec{b} + \gamma\vec{c}$$

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

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

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

Barycentric Coordinates



- Can determine points inside the triangle by computing: α, β, γ
- 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:

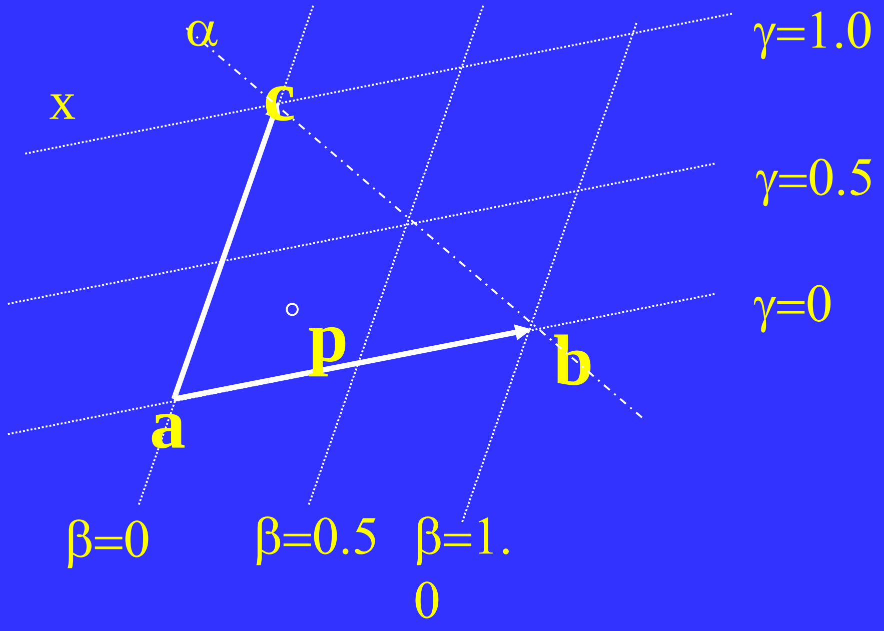
$$c_p = \alpha c_a + \beta c_b + \gamma c_c$$

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…

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

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

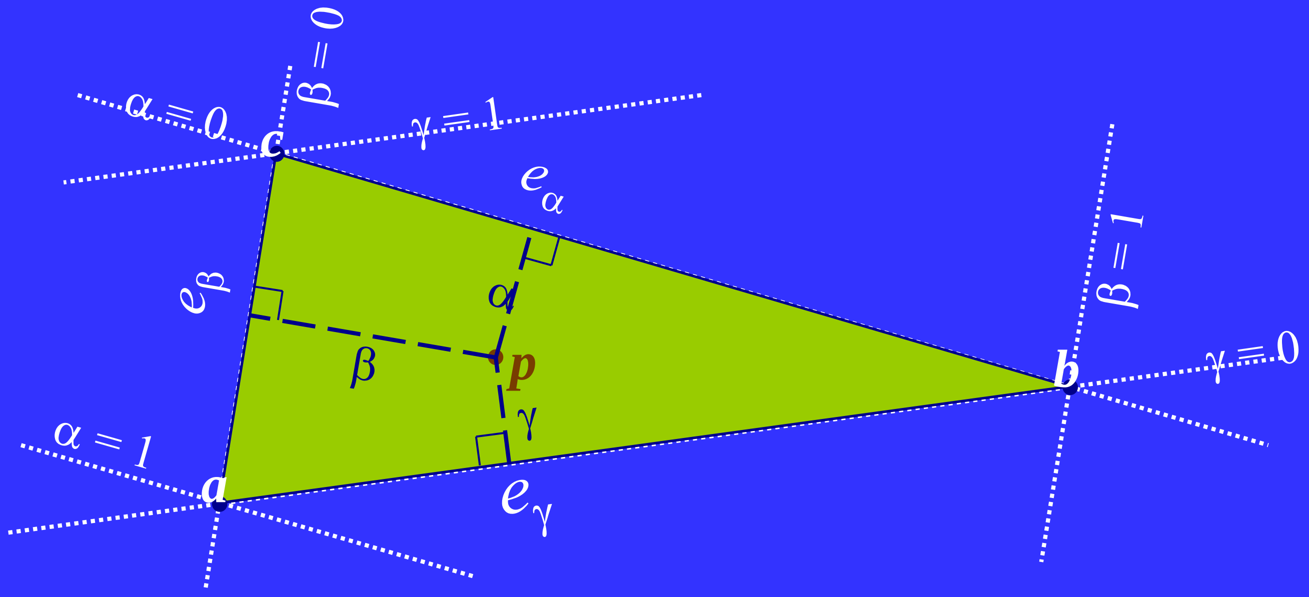
$$\beta=\frac{f_{ac}(x,y)}{f_{ac}(x_{b,}y_b)}$$

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



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

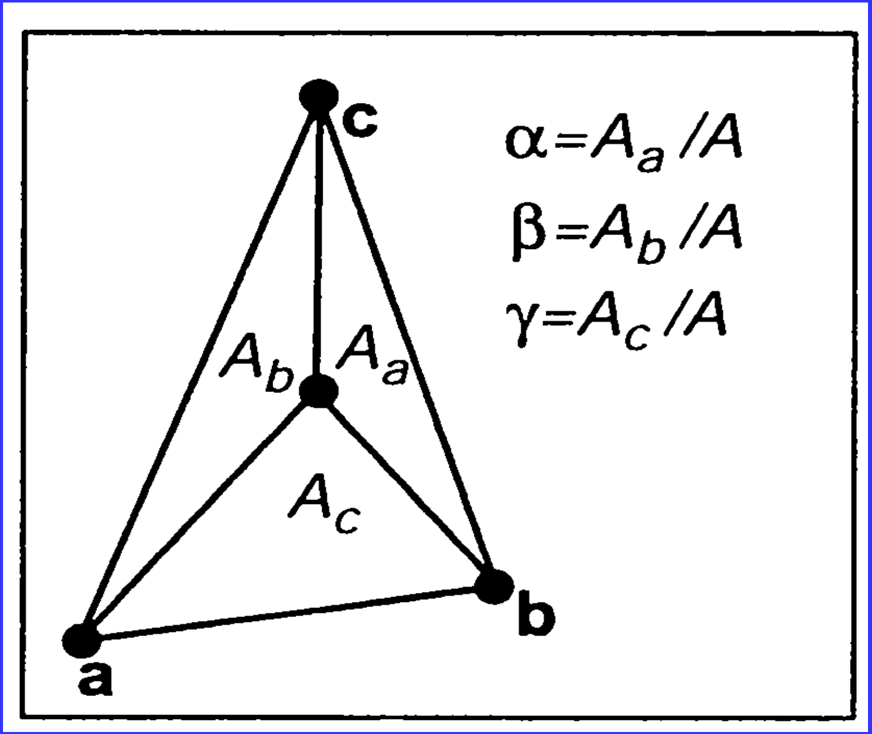
$$\alpha = e_\alpha(p) = \frac{f_{bc}(p)}{f_{bc}(a)}$$

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

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

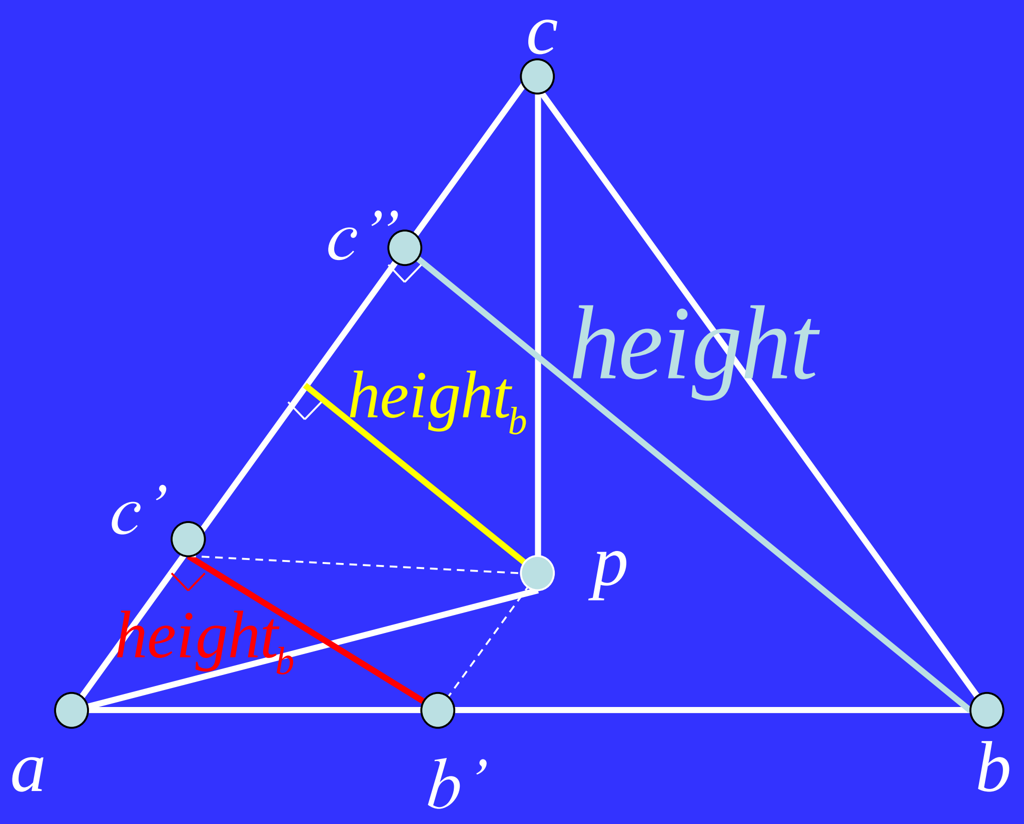
Computing Barycentric Coordinates

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



$$A = A_a + A_b + A_c$$

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



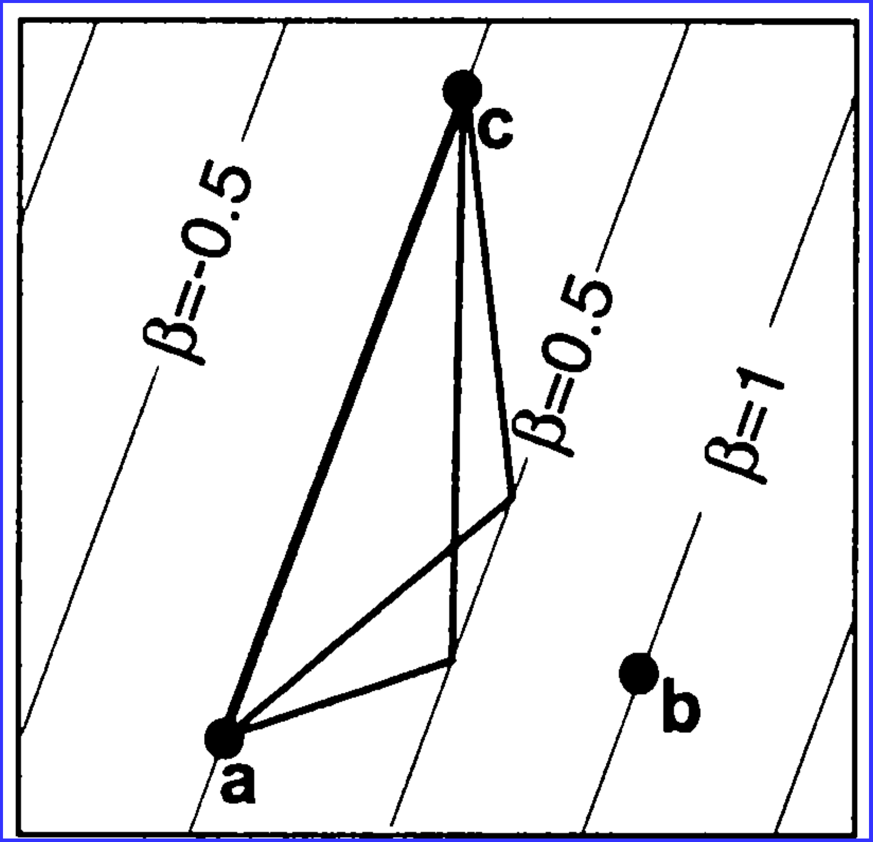
$$a\Delta acp = A_b$$

$$a\Delta abc = A$$

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

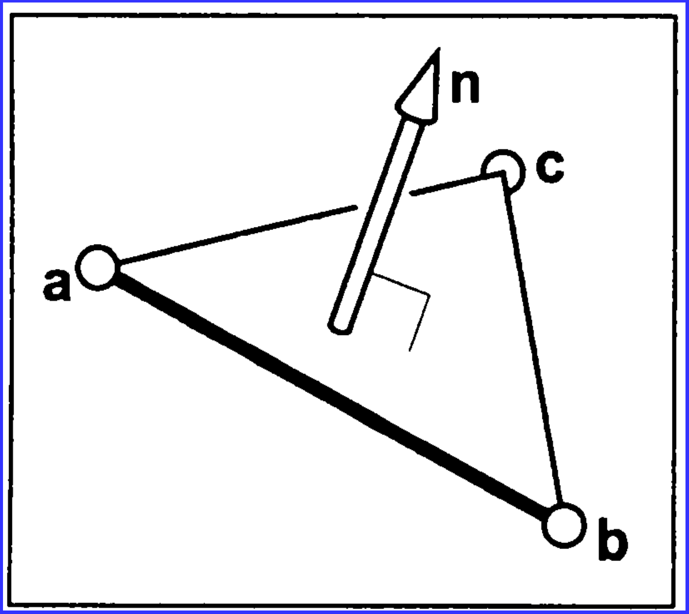
$$\therefore \frac{A_b}{A} = \frac{height_b}{height} = \frac{\ell(a,b')}{\ell(a,b)} = \beta$$

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 $\beta = 0.5$ line. The height and thus the area is proportional to β.

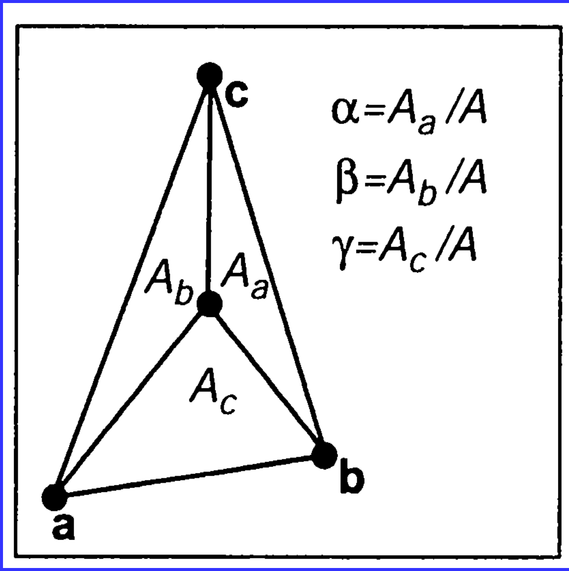
Computing Barycentric Coordinates (3D Triangles)



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

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

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



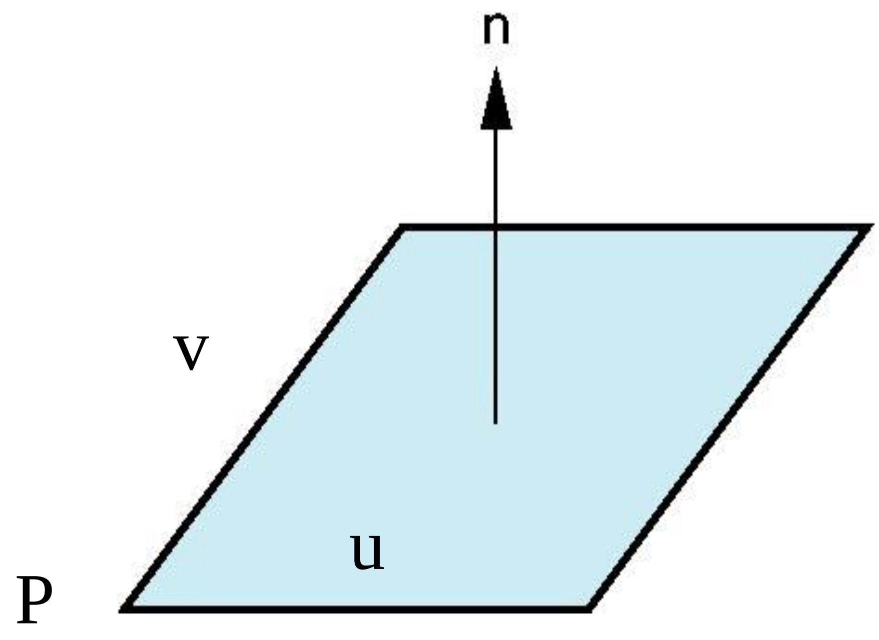
$$\alpha = \frac{\mathbf{n} \cdot \mathbf{n}_a}{\|\mathbf{n}\|^2} \qquad \beta = \frac{\mathbf{n} \cdot \mathbf{n}_b}{\|\mathbf{n}\|^2} \qquad \gamma = \frac{\mathbf{n} \cdot \mathbf{n}_c}{\|\mathbf{n}\|^2}$$

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

Normals

- Every plane has a vector n normal (perpendicular, orthogonal) to it
- From point-two vector form $P(\alpha,\beta)=R+\alpha u+\beta v$, we know we can use the cross product to find $n = u \times v$ and the equivalent form

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



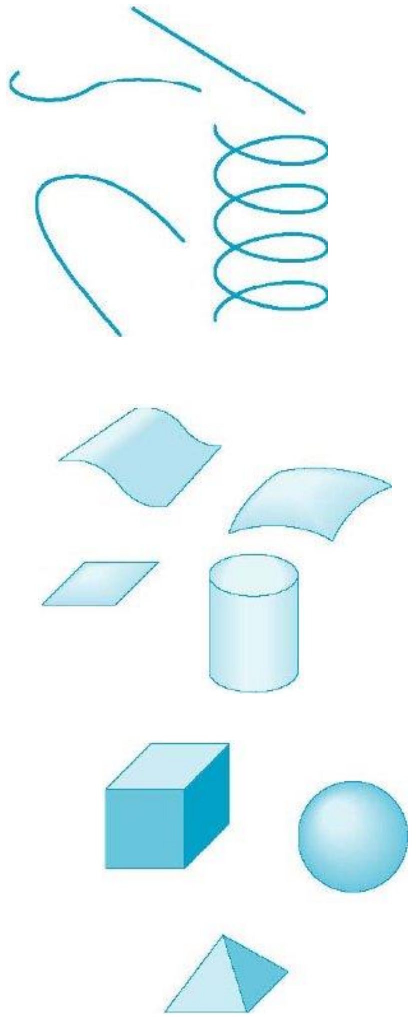
Representation

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
- Introduce homogeneous coordinates

Three-Dimensional Primitives

- Objects:
  - are described by their surfaces and are thought of as being hollow
  - can be specified by their vertices in three dimensions
  - can be composed or approximated by flat, simple, convex polygons



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, as 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

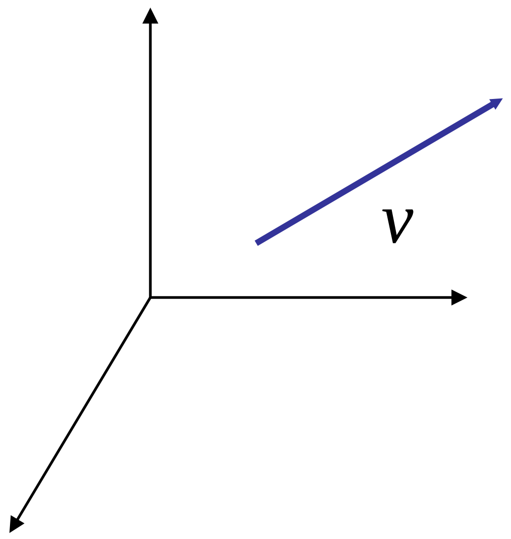
$$\mathbf{a}=[\alpha_1 \ \ \alpha_2 \ \ \ldots.\ \alpha_n]^T = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ . \\ \alpha_n \end{bmatrix}$$

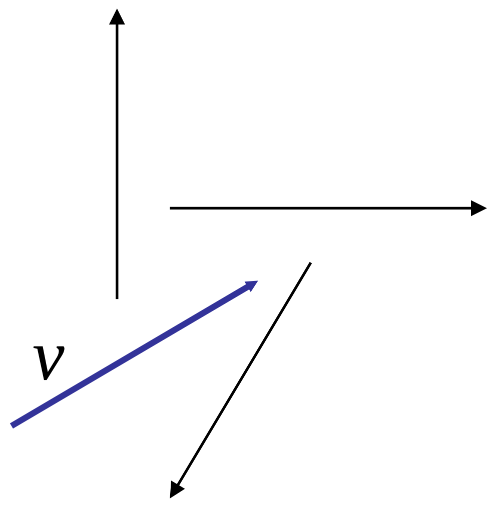
Example

- $v=2v_1+3v_2-4v_3$
- $\mathbf{a}=[2\ 3\ -4]^T$
- Note that this representation is with respect to a particular basis
- For example, in OpenGL we 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?

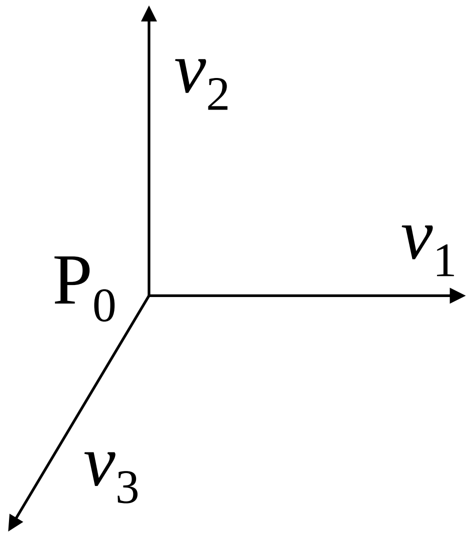




- Both are because vectors have no fixed location

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

  $v=\alpha_1 v_1+ \alpha_2 v_2 +....+\alpha_n v_n$
- Every point can be written as

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

Confusing Points and Vectors

Consider the point and the vector

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

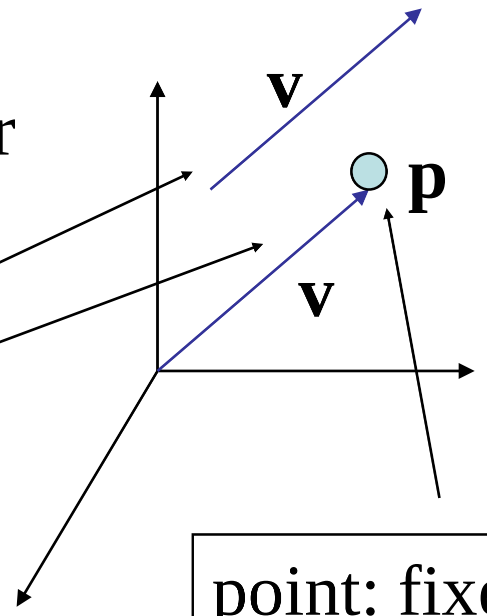
$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$

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



Vector can be placed anywhere

point: fixed

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

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

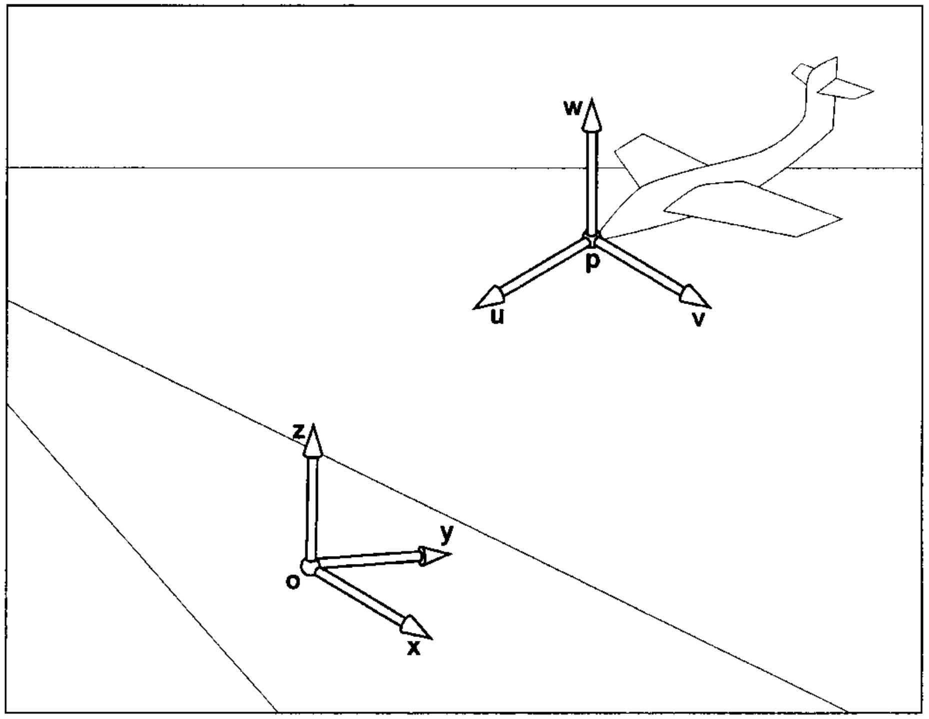
$$\mathbf{a}=[\alpha_1\ \alpha_2\ \alpha_3]$$
$$\mathbf{b}=[\beta_1\ \beta_2\ \beta_3]$$

where

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

Change of Coordinate Systems

A Flight Simulator



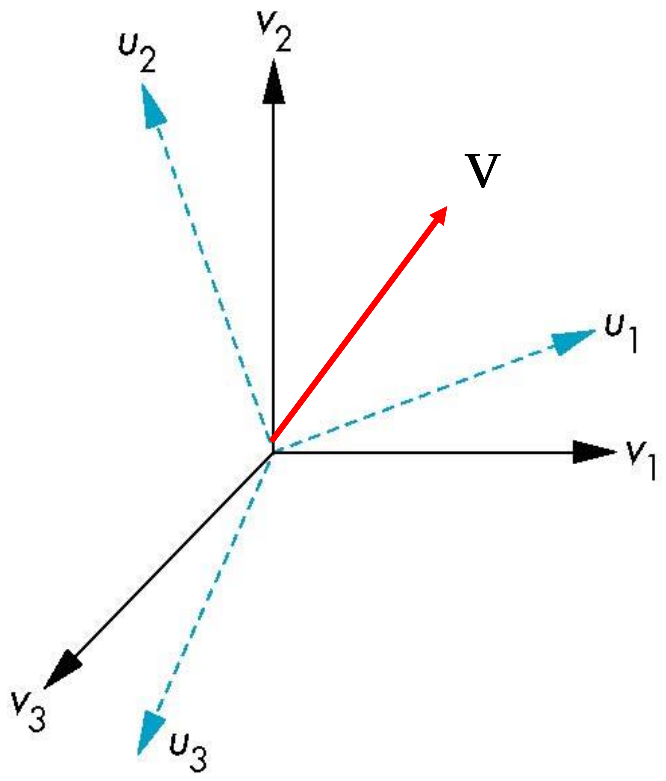
World coordinate system: xyz

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 Basis Example

$$\mathbf{v} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \mathbf{v}_3 \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \end{bmatrix}$$

old *new*

$$\begin{aligned} \mathbf{u}_1 &= \mathbf{v}_1 \\ \mathbf{u}_2 &= \mathbf{v}_1 + \mathbf{v}_2 \\ \mathbf{u}_3 &= \mathbf{v}_1 + \mathbf{v}_2 + \mathbf{v}_3 \end{aligned} \Rightarrow M = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow \begin{aligned} \mathbf{u} &= \mathbf{M}^* \mathbf{v} \\ \mathbf{v} &= \left(\mathbf{M}^*\right)^{-1} \mathbf{u} \end{aligned}$$

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \Rightarrow \mathbf{b} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 \\ 1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ 3 \end{bmatrix}$$

That is,

$$\mathbf{w} = \mathbf{v}_1 + 2\mathbf{v}_2 + 3\mathbf{v}_3 = -\mathbf{u}_1 - \mathbf{u}_2 + 3\mathbf{u}_3$$

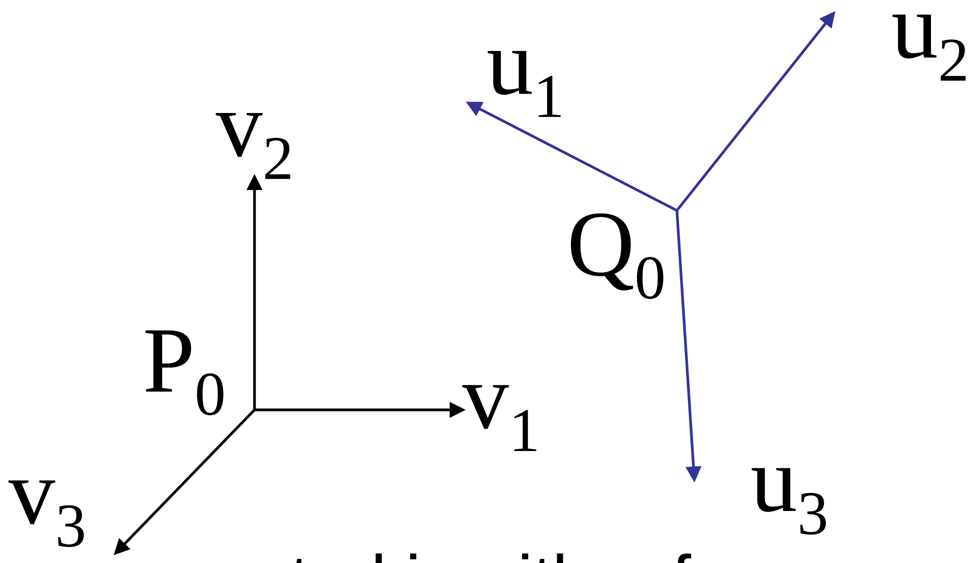
Change of Frames

- 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

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

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 (**M**=**I**)

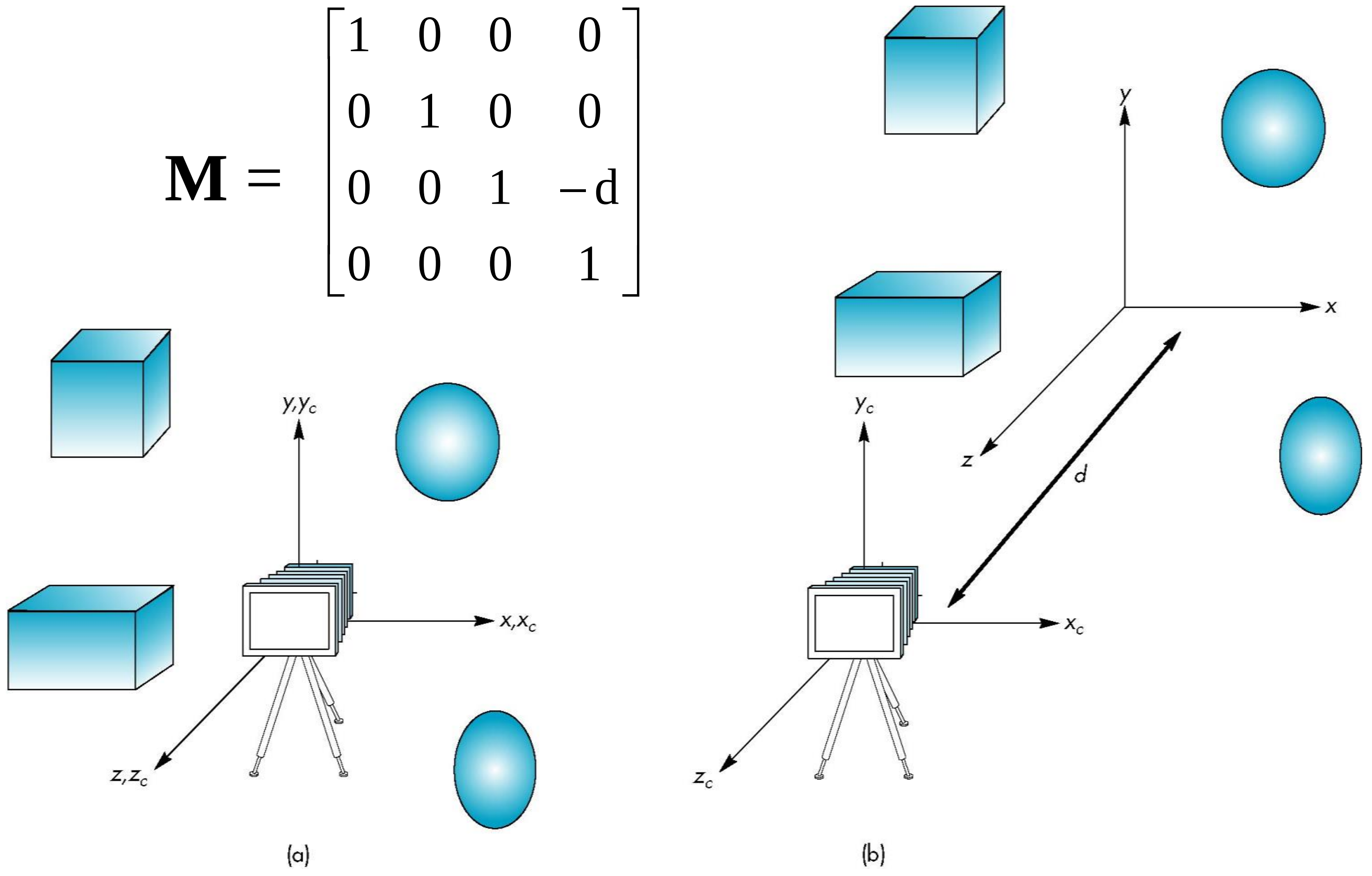
Frames in OpenGL

- Object or model coordinates
- World coordinates
- Eye (or camera) coordinates
- Clip coordinates
- Normalized device coordinates
- Window (or screen) coordinates

Moving the Camera

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

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



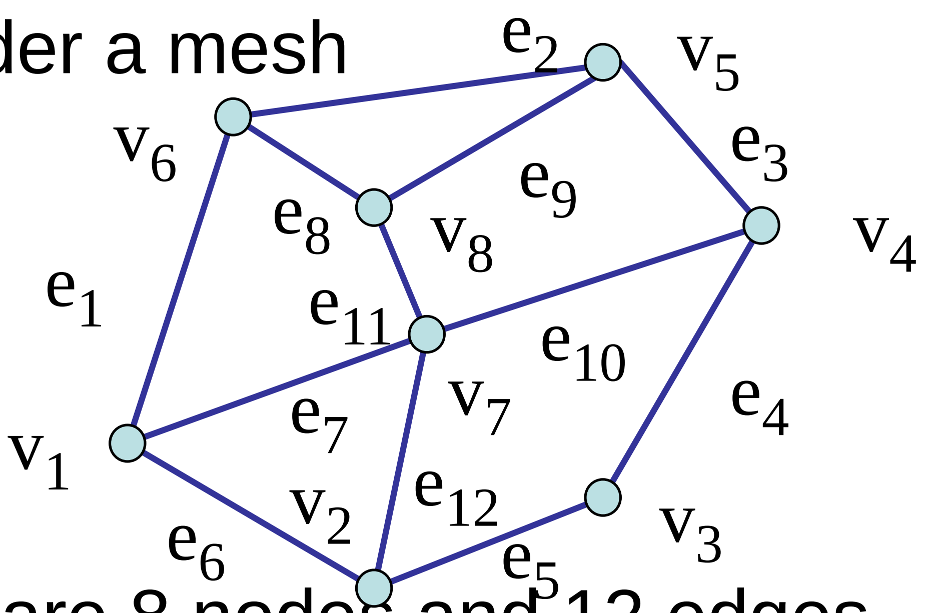
Building Models

Objectives

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

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

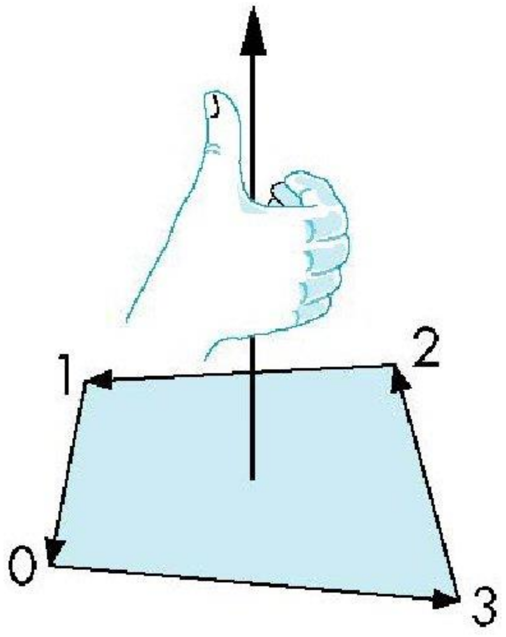
- Define each polygon by the geometric locations of its vertices
- Leads to OpenGL code such as

```
glBegin(GL_POLYGON);
    glVertex3f(x1, x1, x1);
    glVertex3f(x6, x6, x6);
    glVertex3f(x7, x7, x7);
glEnd();
```

- 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 points 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

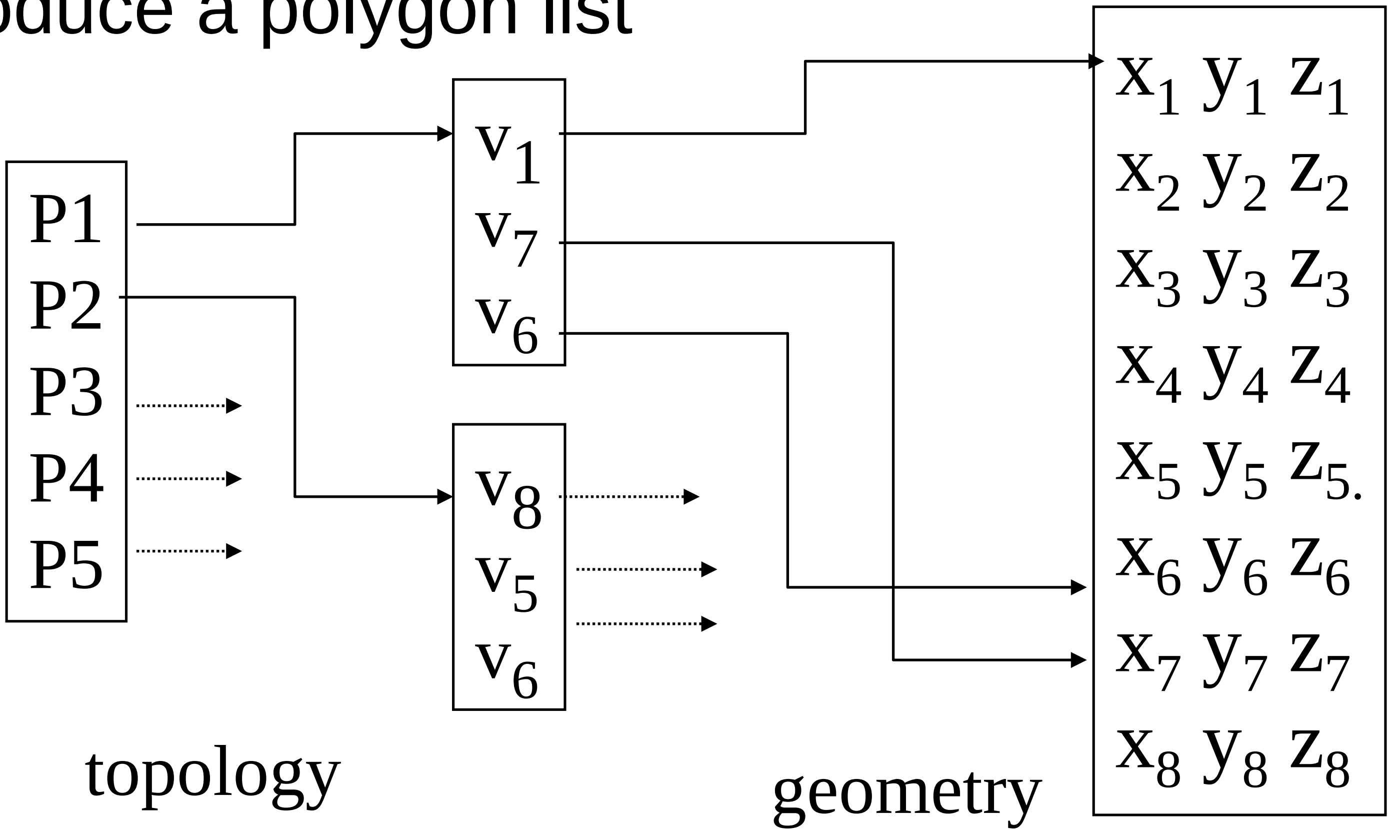


Geometry vs Topology

- 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

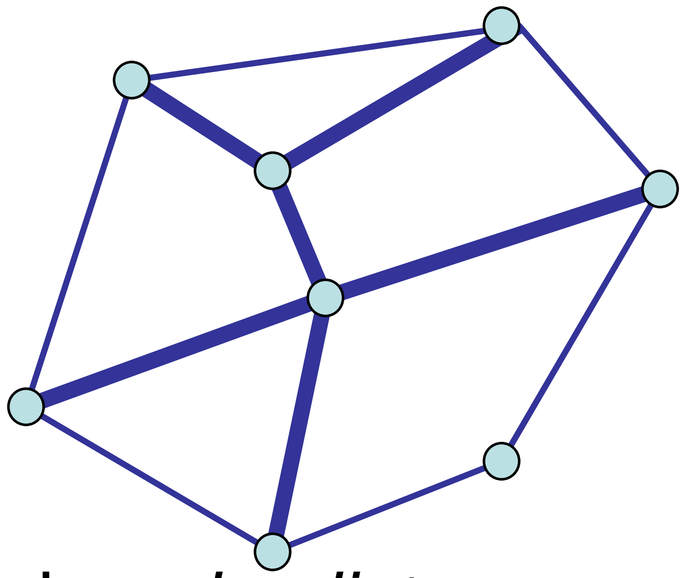
Vertex Lists

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



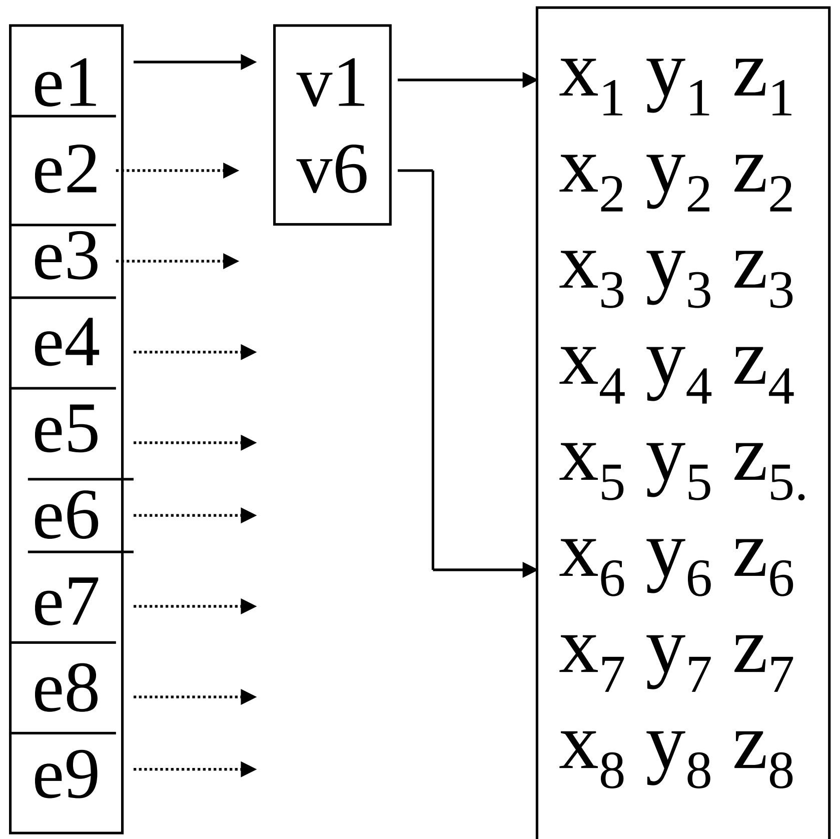
Shared Edges

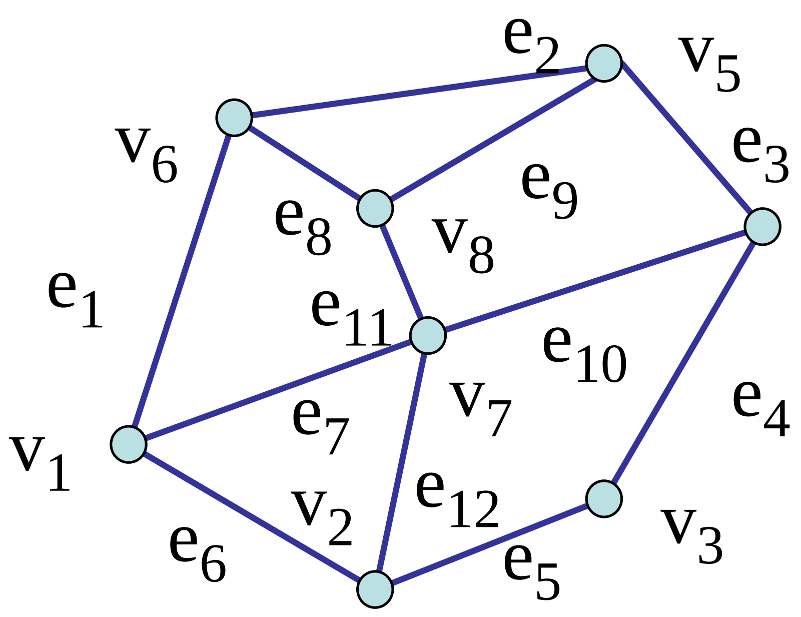
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- Can store mesh by *edge list*

Edge List



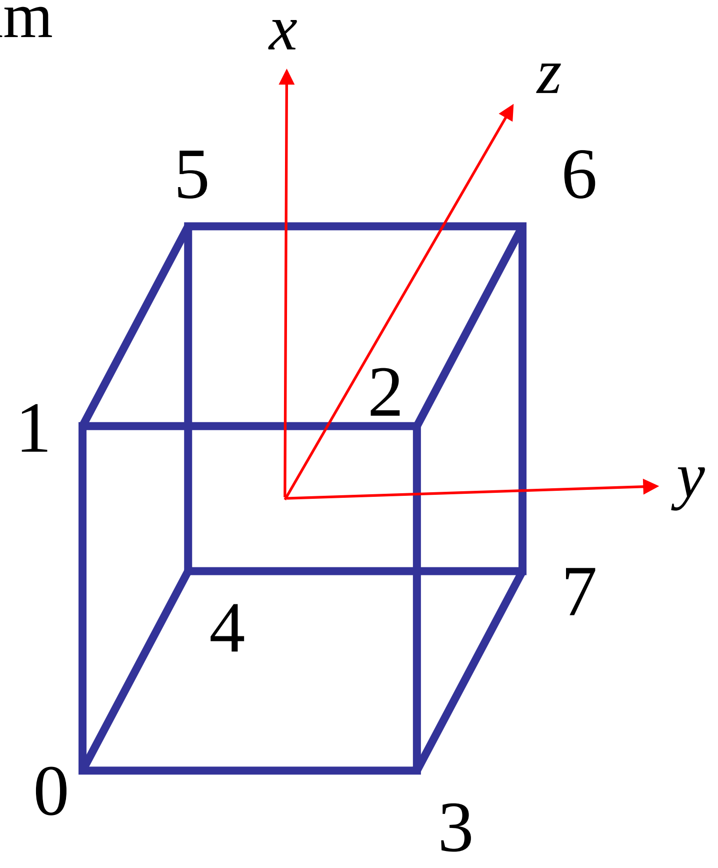


Note polygons are not represented

Modeling a Cube

Model a color cube for rotating cube program

Define global arrays for vertices and colors



```
GLfloat vertices[][3] =
{{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
```

```
GLfloat colors[][3] =
{{0.0,0.0,0.0},{1.0,0.0,0.0},
{1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
{1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
```

```
GLfloat normals[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
            {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
            {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
```

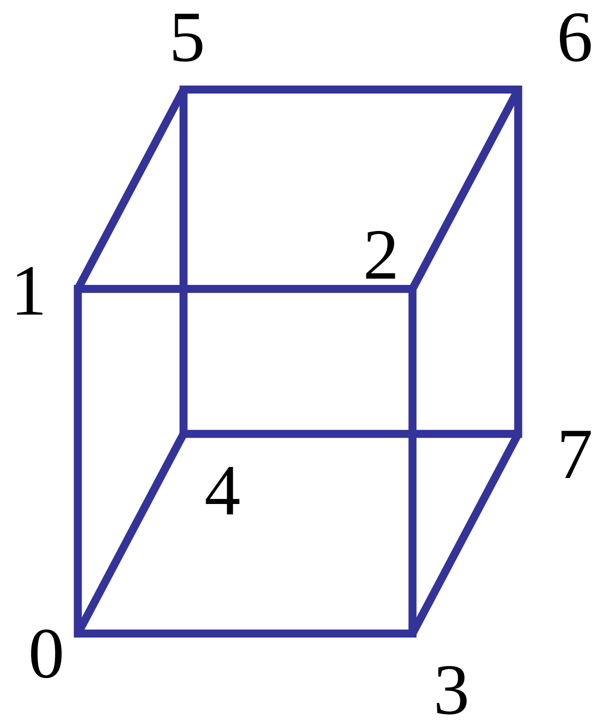
Drawing a polygon from a list of indices

Draw a quadrilateral from a list of indices into the array **`vertices`** and use color corresponding to first index

```
void polygon(int a, int b, int c
, int d)
{
   glBegin(GL_POLYGON);
      glColor3fv(colors[a]);
      glVertex3fv(vertices[a]);
      glVertex3fv(vertices[b]);
      glVertex3fv(vertices[c]);
      glVertex3fv(vertices[d]);
    glEnd();
 }
```

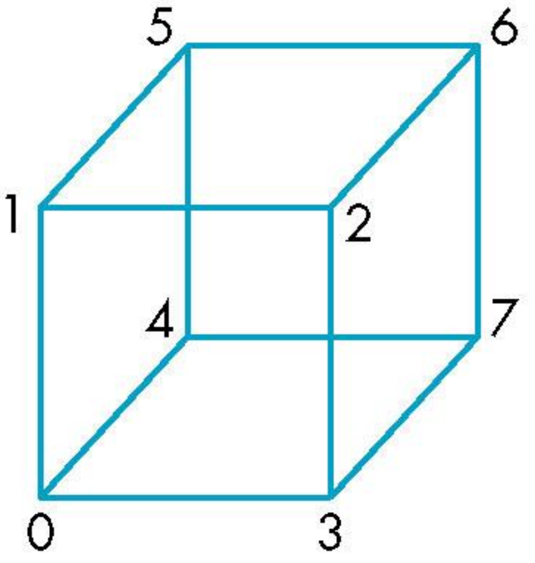
Draw cube from faces

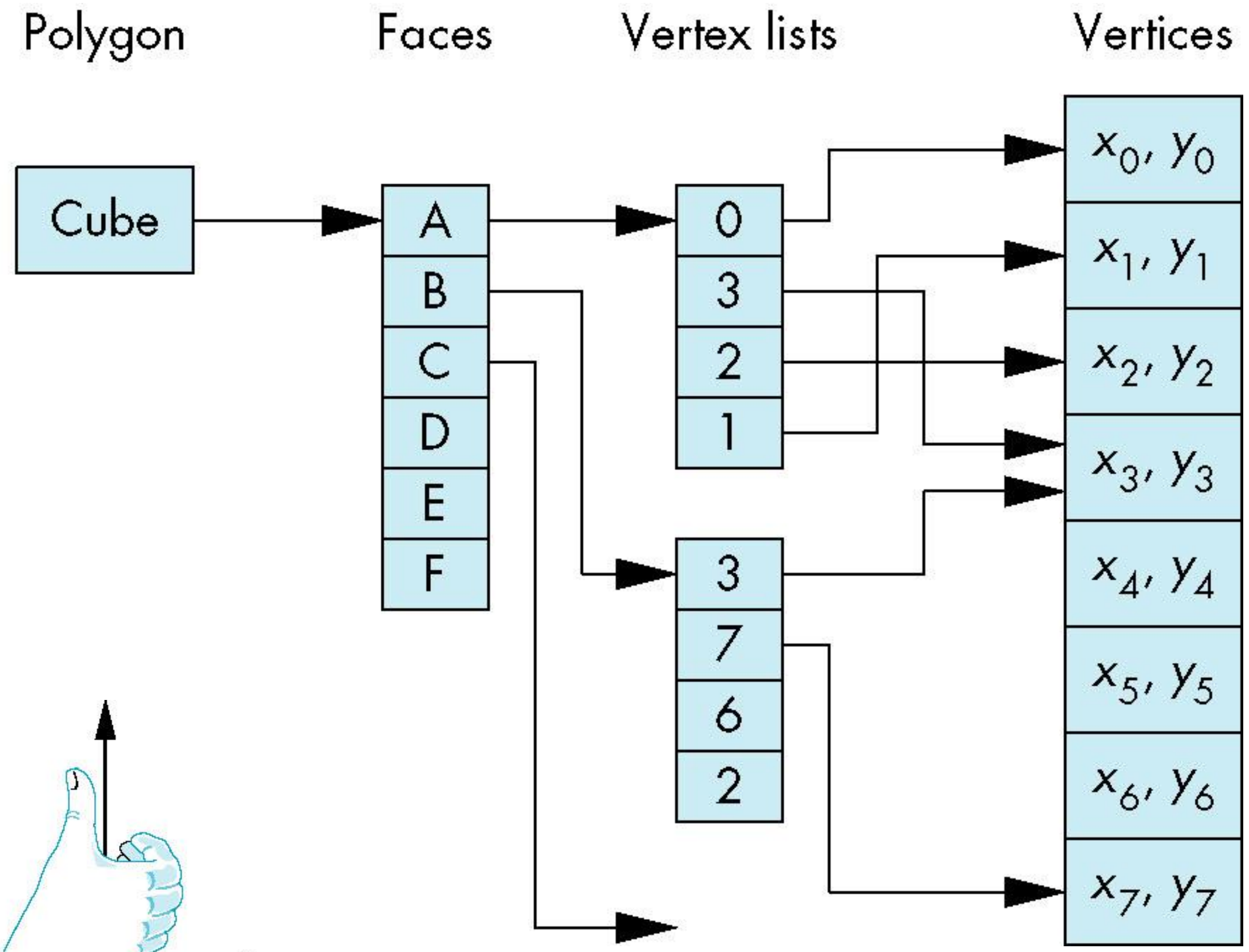
```
void colorcube( )
{
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}
```



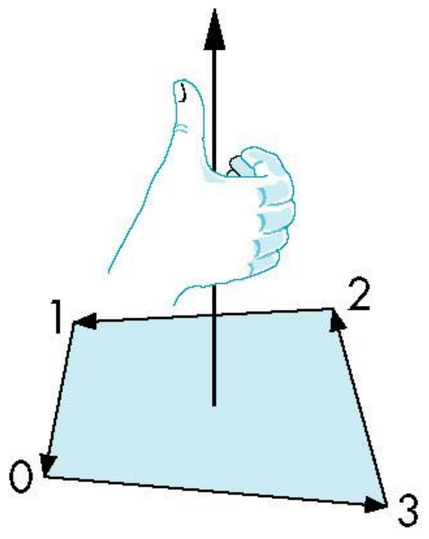
Note that vertices are ordered so that we obtain correct outward facing normals

Data Structures for Cube Representation





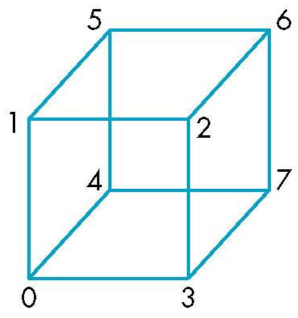
Outward facing
(right hand rule)

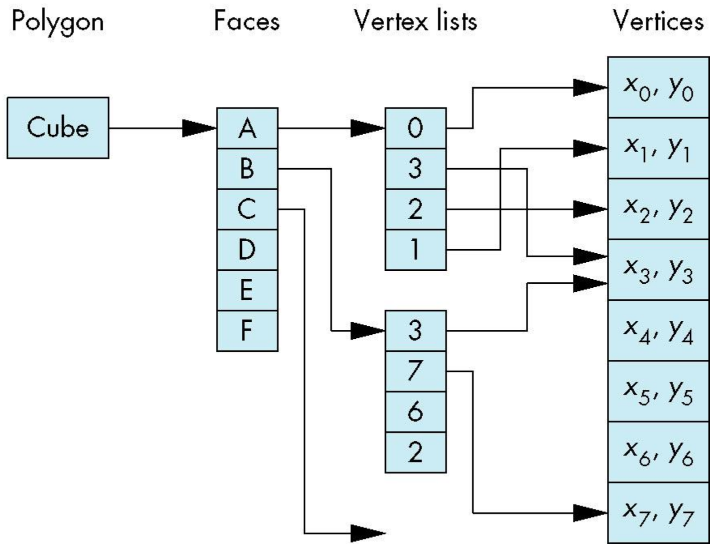


The Color Cube

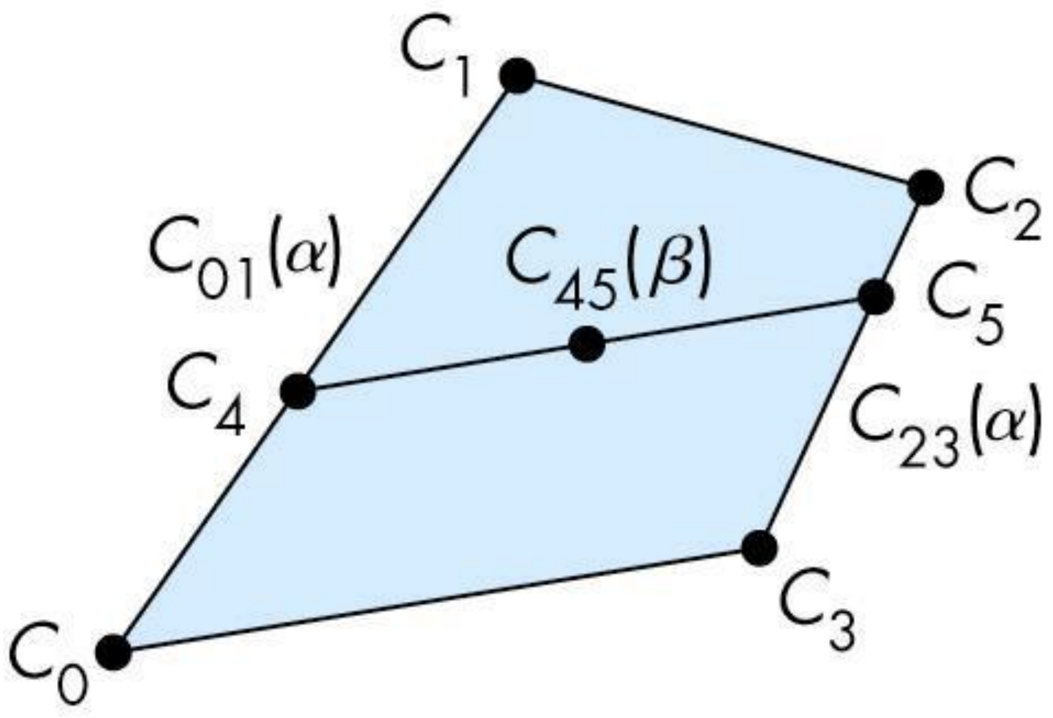
```
void polygon(int a, int b, int c , int d)
{   glBegin(GL_POLYGON);
        glColor3fv(colors[a]);
        glNormal3fv(normals[a]);
        glVertex3fv(vertices[a]);
        glColor3fv(colors[b]);
        glNormal3fv(normals[b]);
        glVertex3fv(vertices[b]);
        glColor3fv(colors[c]);
        glNormal3fv(normals[c]);
        glVertex3fv(vertices[c]);
        glColor3fv(colors[d]);
        glNormal3fv(normals[d]);
        glVertex3fv(vertices[d]);
  glEnd();
}
```

```
void colorcube(void)
{       polygon(0,3,2,1);
        polygon(2,3,7,6);
        polygon(0,4,7,3);
        polygon(1,2,6,5);
        polygon(4,5,6,7);
        polygon(0,1,5,4);
}
```





Bilinear Interpolation

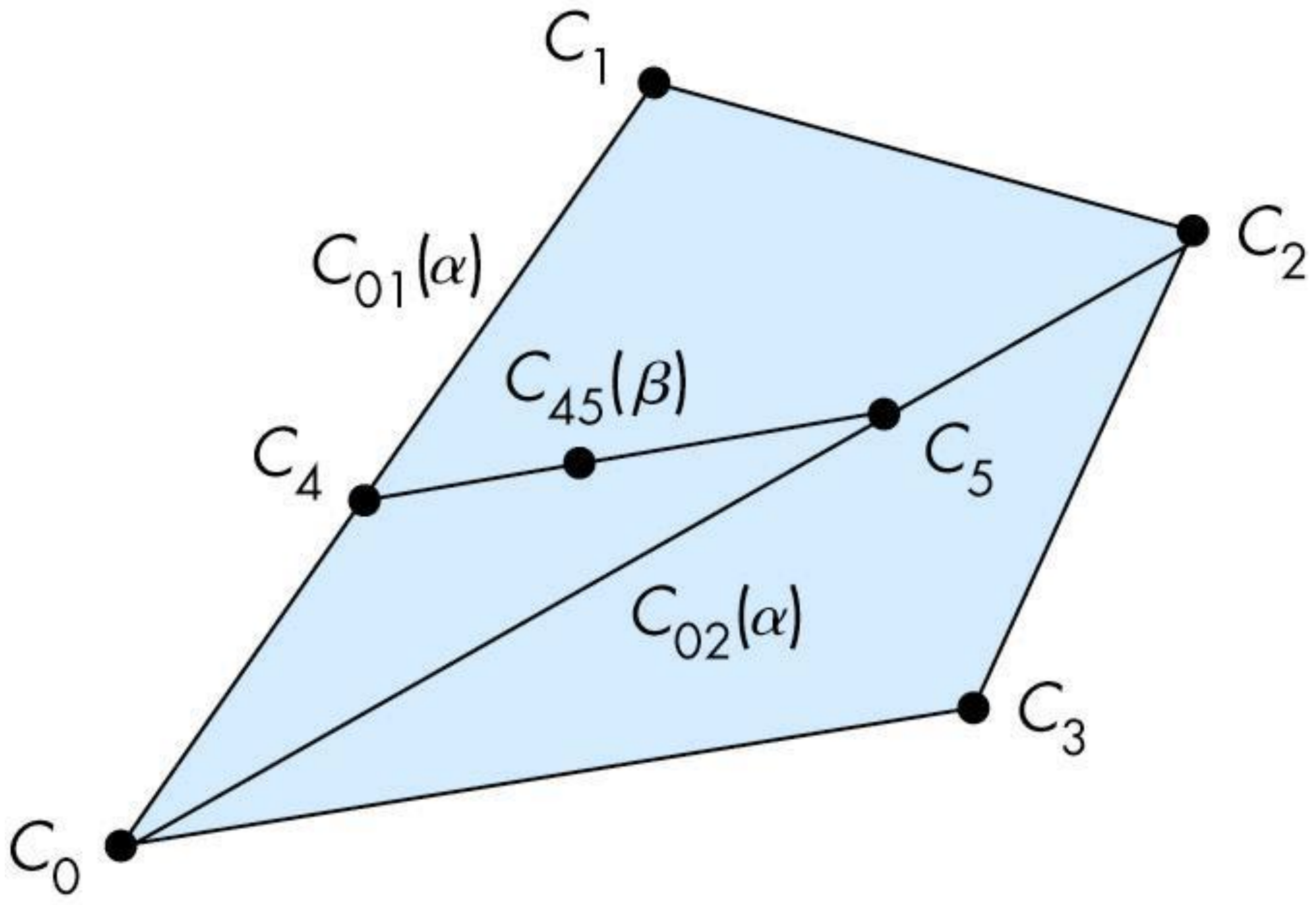


$$C_{01}(\alpha) = (1-\alpha)C_0 + \alpha C_1$$

$$C_{23}(\alpha) = (1-\alpha)C_2 + \alpha C_3$$

$$C_{45}(\beta) = (1-\beta)C_4 + \beta C_5$$

Bilinear Interpolation of a Triangle



Efficiency

- The weakness of our approach is that we are building the model in the application and must do many function calls to draw the cube
- Drawing a cube by its faces in the most straight forward way requires
  – 6 **glBegin,** 6 **glEnd**
  – 6 **glColor**
  – 24 **glVertex**
  – More if we use texture and lighting

Vertex Arrays

- OpenGL provides a facility called vertex arrays that allows us to store array data in the implementation
- Six types of arrays supported
  - Vertices
  - Colors
  - Color indices
  - Normals
  - Texture coordinates
  - Edge flags
- We will need only colors and vertices

Initialization

- Using the same color and vertex data, first we enable

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

- Identify location of arrays

```
glVertexPointer(3, GL_FLOAT, 0, vertices);
```

3d arrays   stored as floats   data contiguous   data array

```
glColorPointer(3, GL_FLOAT, 0, colors);
```

Mapping indices to faces

- Form an array of face indices

```
GLubyte cubeIndices[24] = {0,3,2,1,2,3,7,6
    0,4,7,3,1,2,6,5,4,5,6,7,0,1,5,4};
```

- Each successive four indices describe a face of the cube
- Draw through **`glDrawElements`** which replaces all **`glVertex`** and **`glColor`** calls in the display callback

Drawing the cube

- Method 1:

what to draw

number of indices

```
for(i=0; i<6; i++) glDrawElements(GL_POLYGON, 4,
      GL_UNSIGNED_BYTE, &cubeIndices[4*i]);
```

format of index data

start of index data

- Method 2:

```
glDrawElements(GL_QUADS, 24,
      GL_UNSIGNED_BYTE, cubeIndices);
```

Draws cube with 1 function call!!

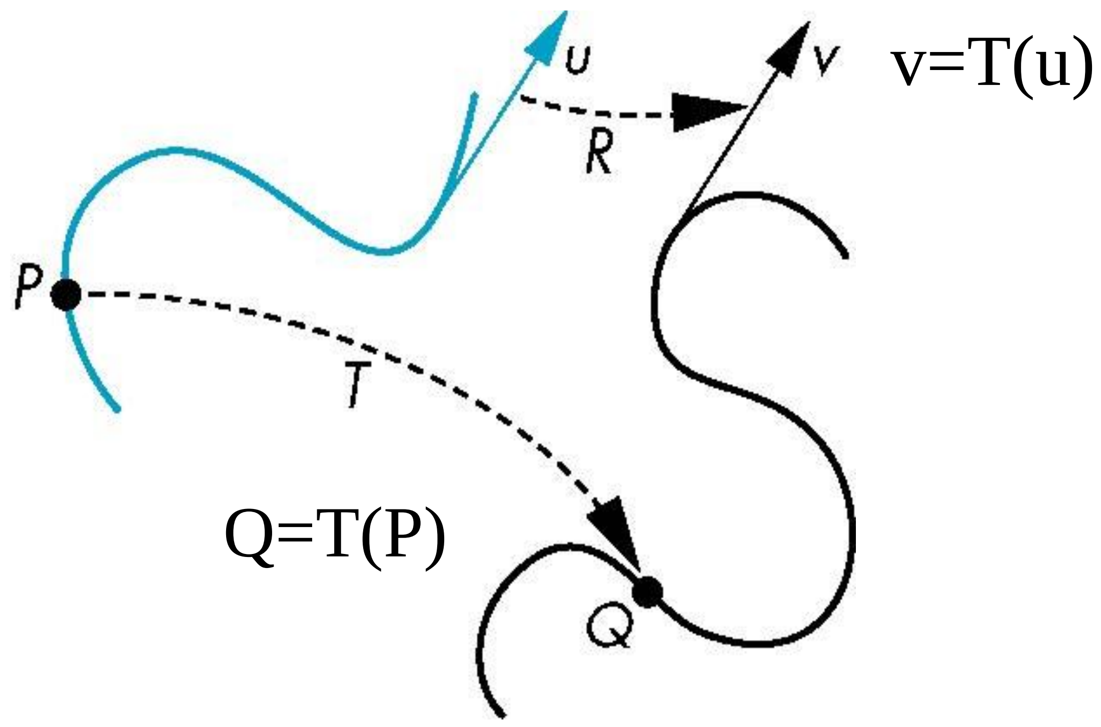
Transformations

Objectives

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

General Transformations

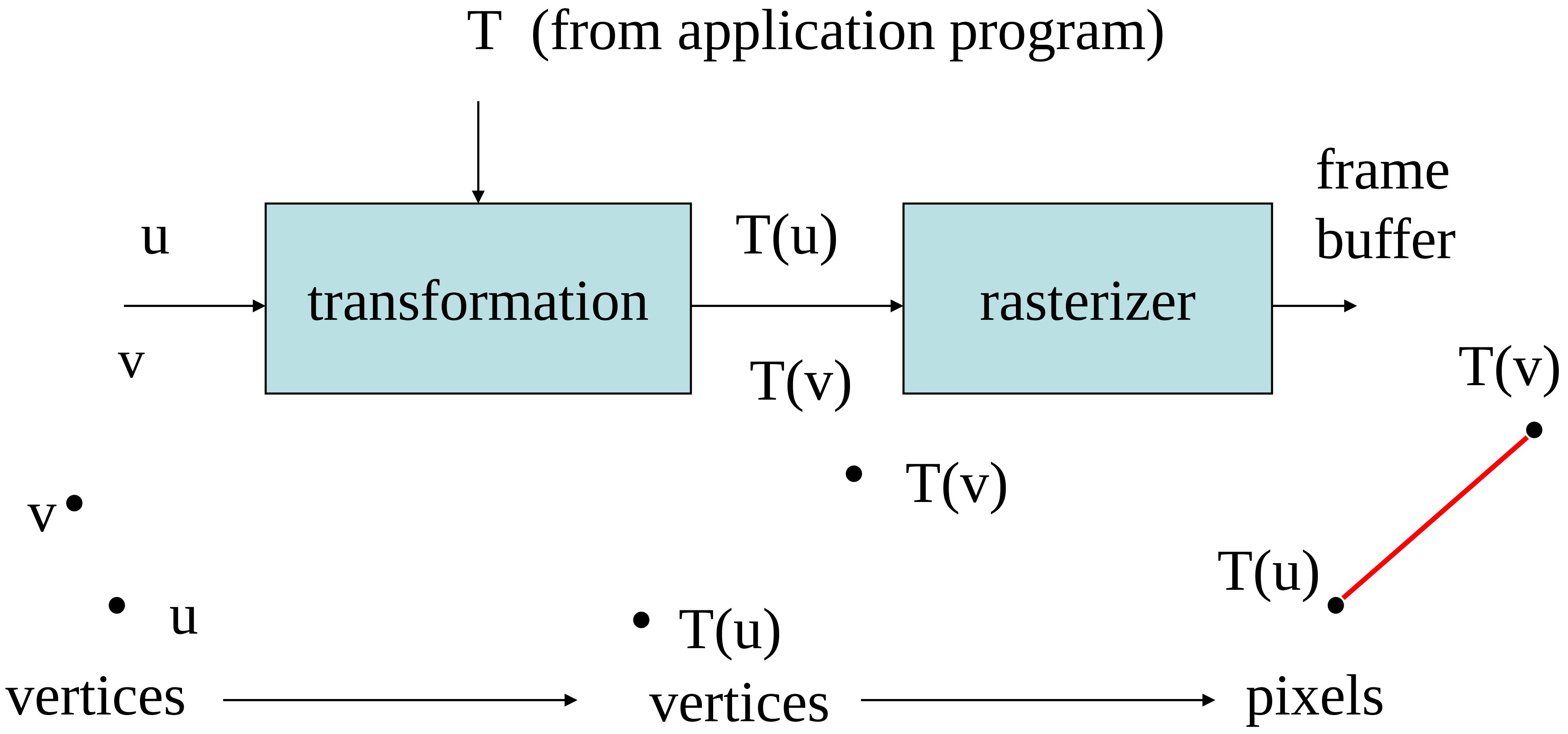
A transformation maps points to other points and/or vectors to other vectors



Affine Transformations

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

Pipeline Implementation



Notation

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

P,Q, R: points in an affine space

u, v, w: vectors in an affine space

α, β, γ: scalars

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

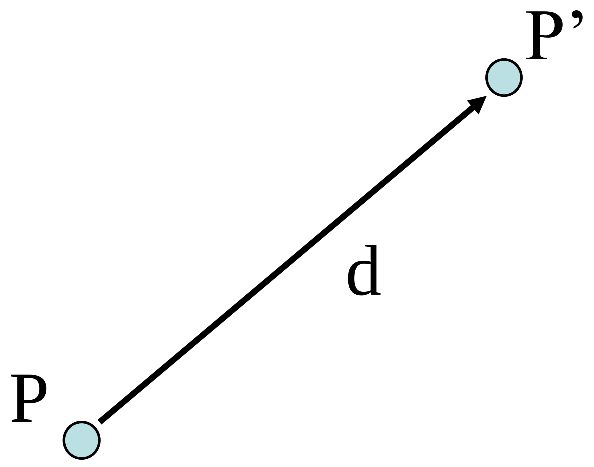
-array of 4 scalars in homogeneous coordinates

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

-array of 4 scalars in homogeneous coordinates

Translation

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

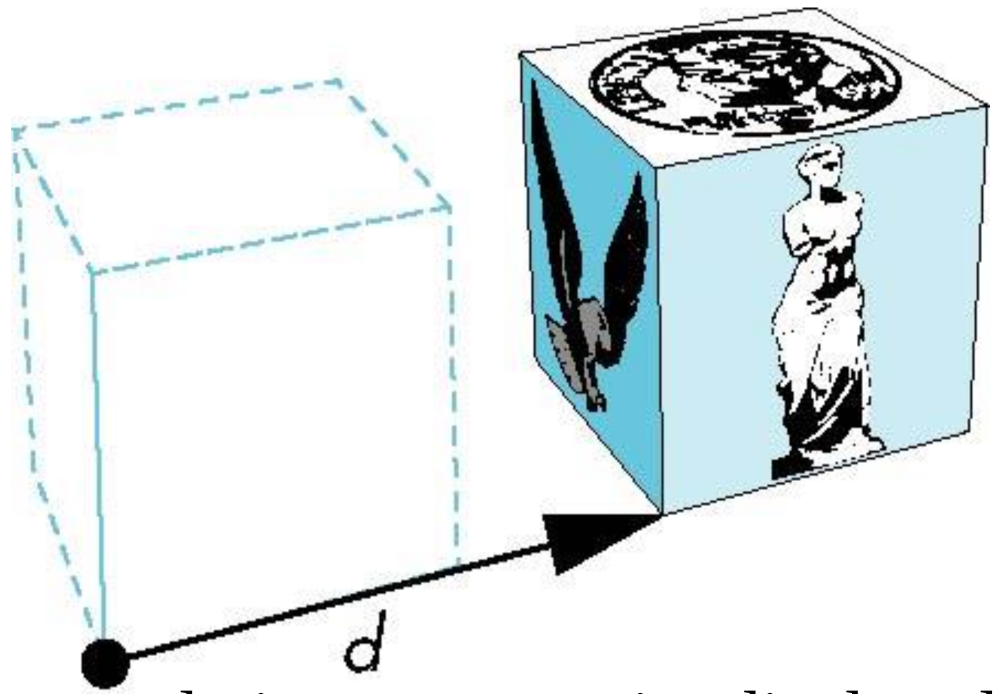


- Displacement determined by a vector d
  - Three degrees of freedom
  - P'=P+d

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

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

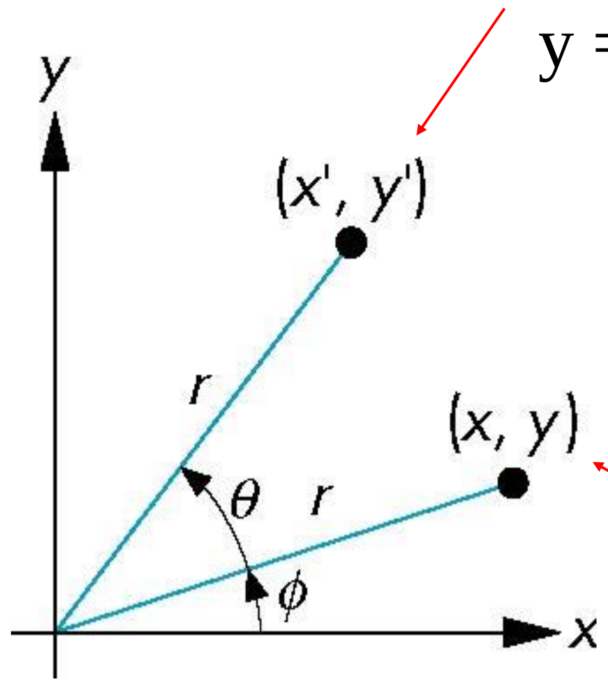
This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated together

Rotation (2D)

Consider rotation about the origin by θ degrees

– radius stays the same, angle increases by θ

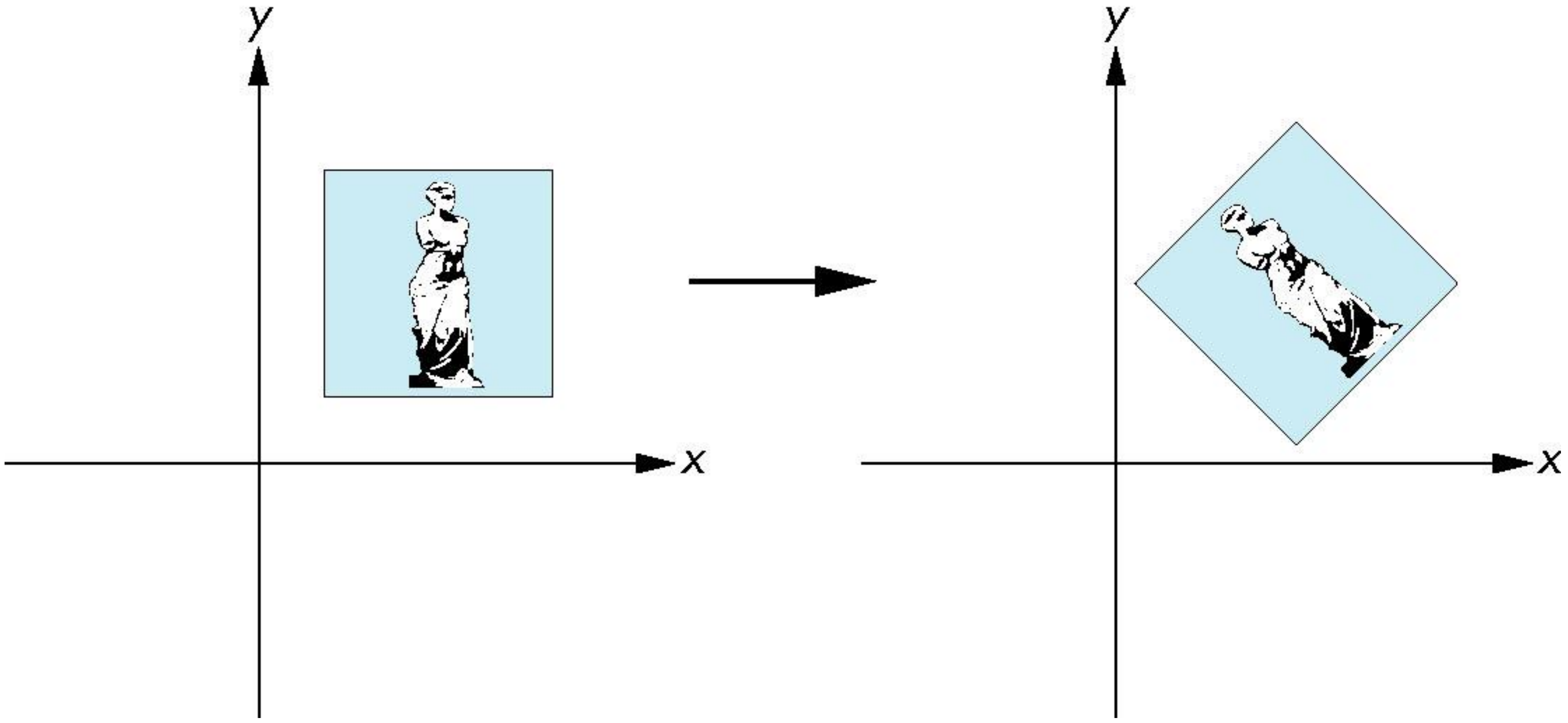
$x = r \cos(\phi + \theta)$
$y = r \sin(\phi + \theta)$



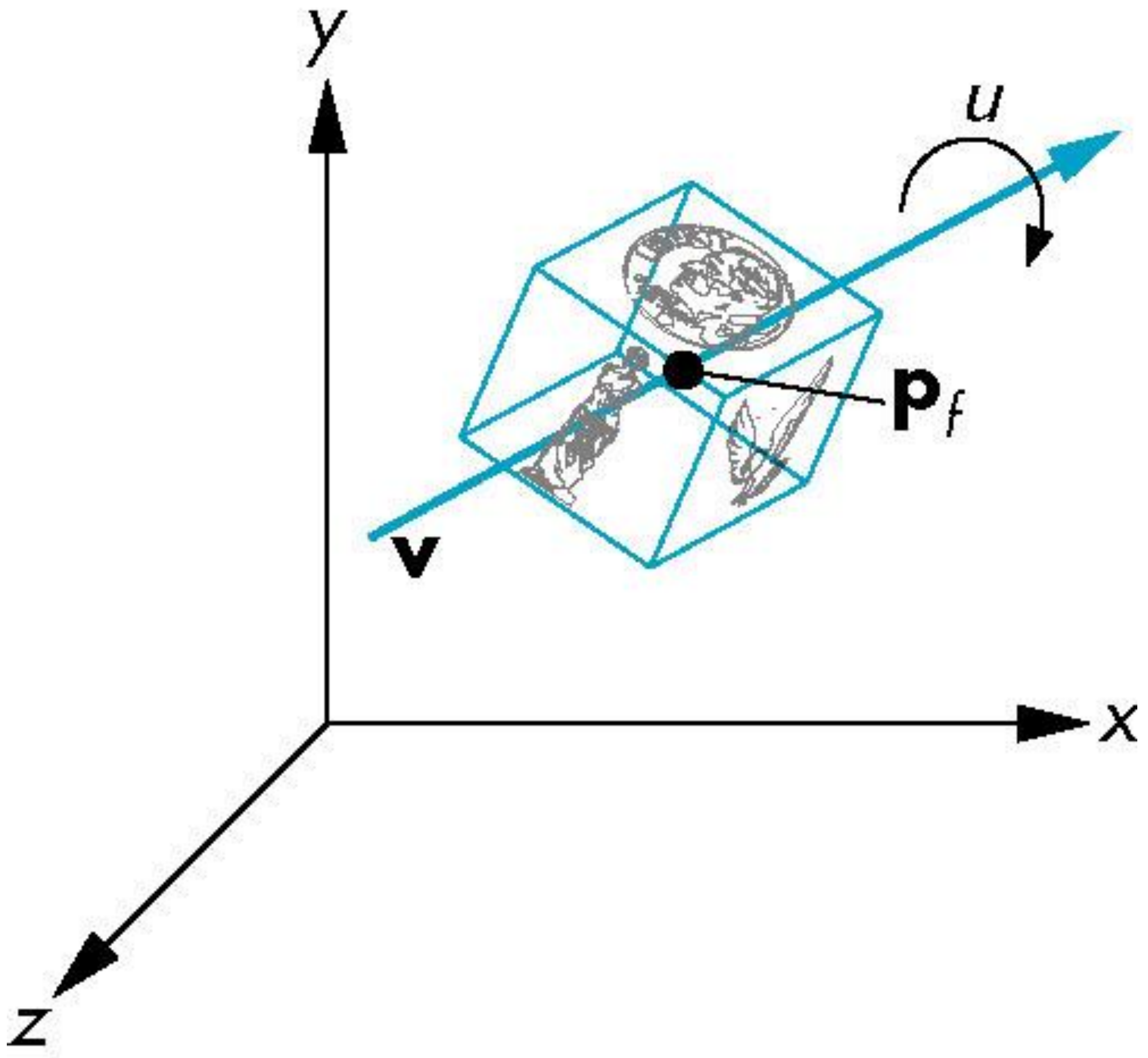
$$\boxed{\begin{aligned} x' &= x \cos\theta - y \sin\theta \\ y' &= x \sin\theta + y \cos\theta \end{aligned}}$$

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

Rotation about a fixed point



Three-dimensional rotation



Rotation about the z axis

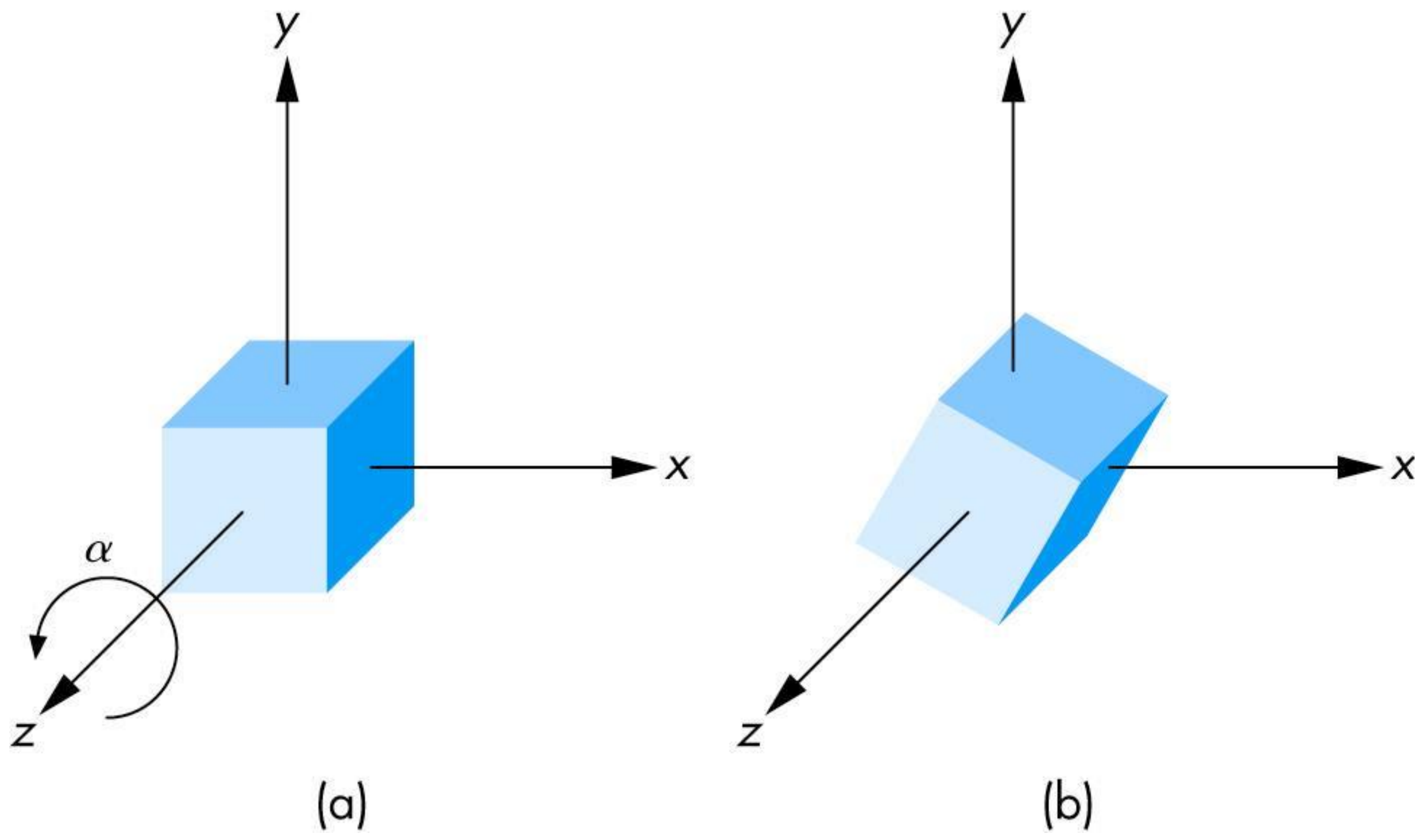
- Rotation about z axis in three dimensions leaves all points with the same z
  - Equivalent to rotation in two dimensions in planes of constant z

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

  - or in homogeneous coordinates

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

Rotation of a cube about the z-axis



Before rotation

After rotation

Rotation Matrix

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

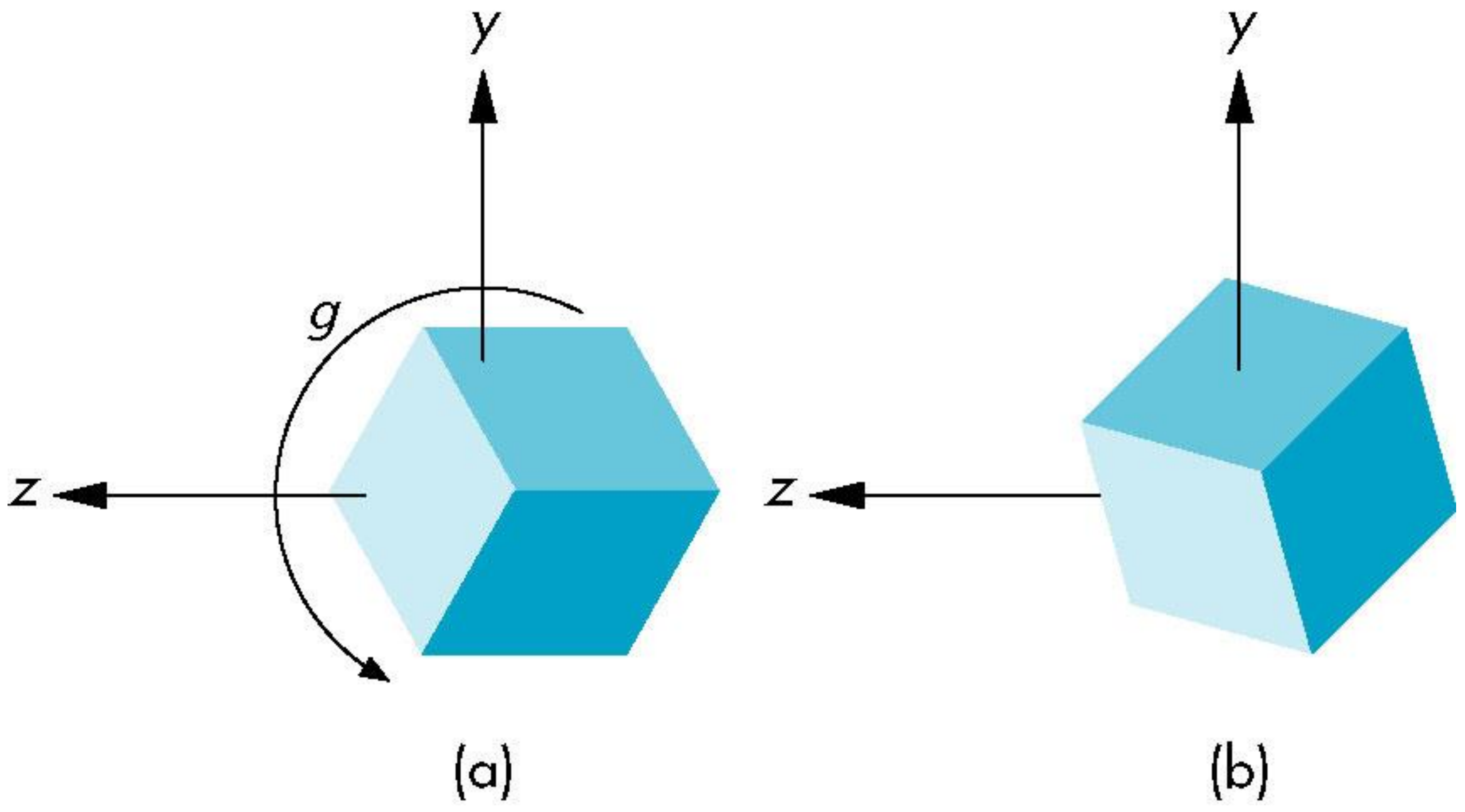
Rotation about x and y axes

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

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

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

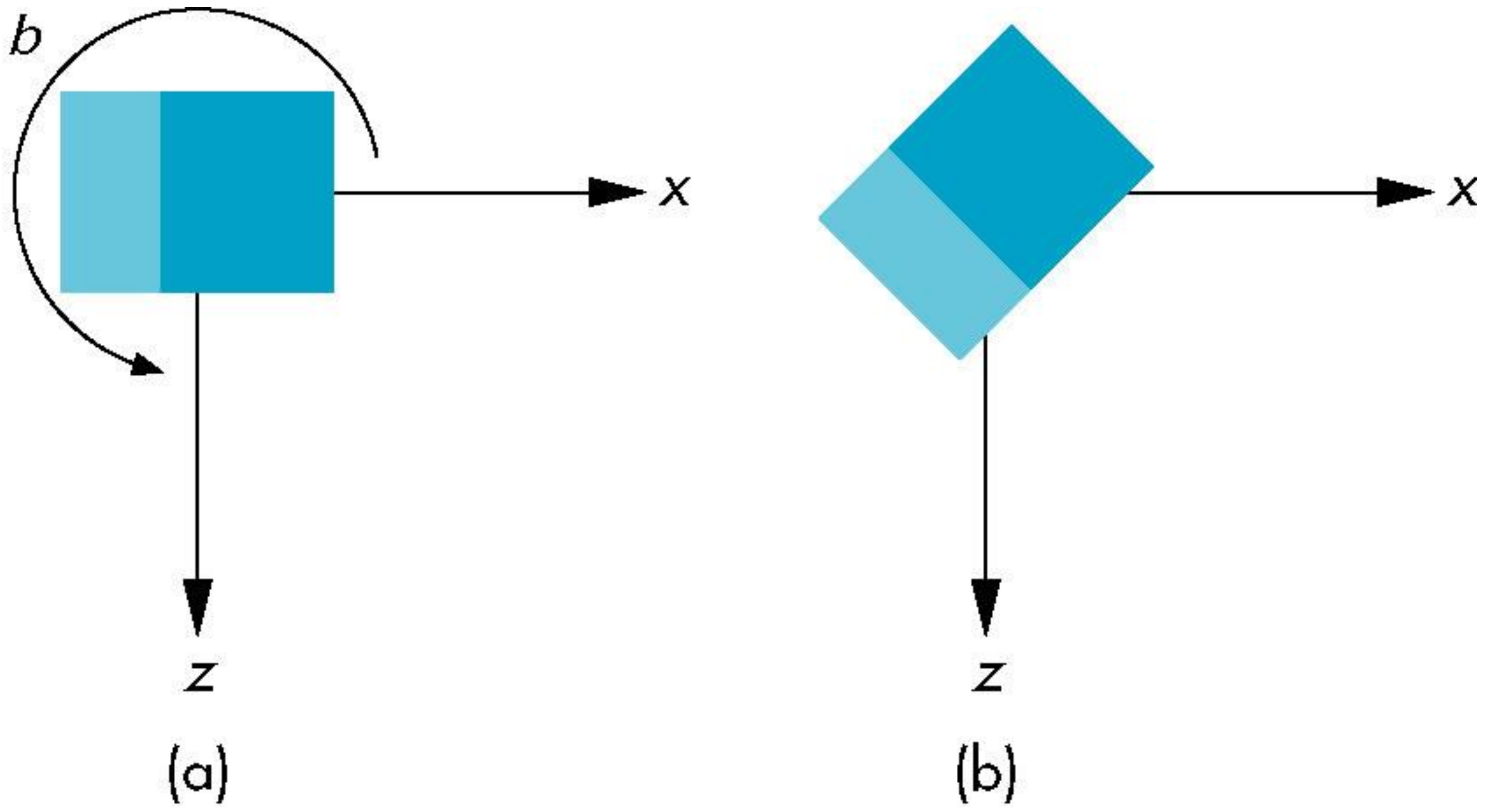
Rotation of a cube about the x-axis



Before rotation

After rotation

Rotation of a cube about the y-axis



Before rotation After rotation

Scaling

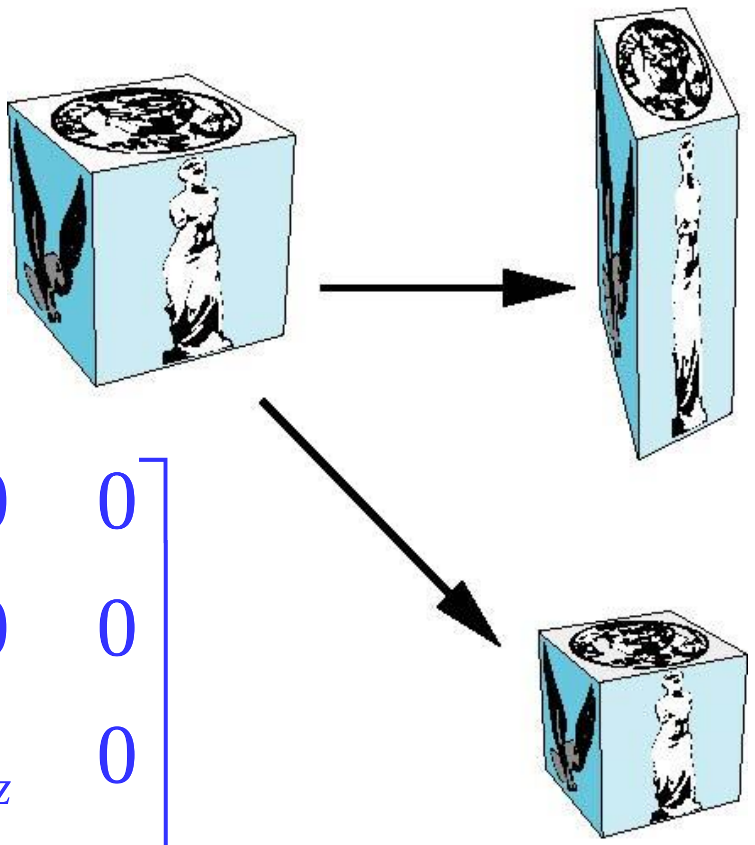
Expand or contract along each axis (fixed point of origin)

$x' = s_x x$

$y' = s_y y$

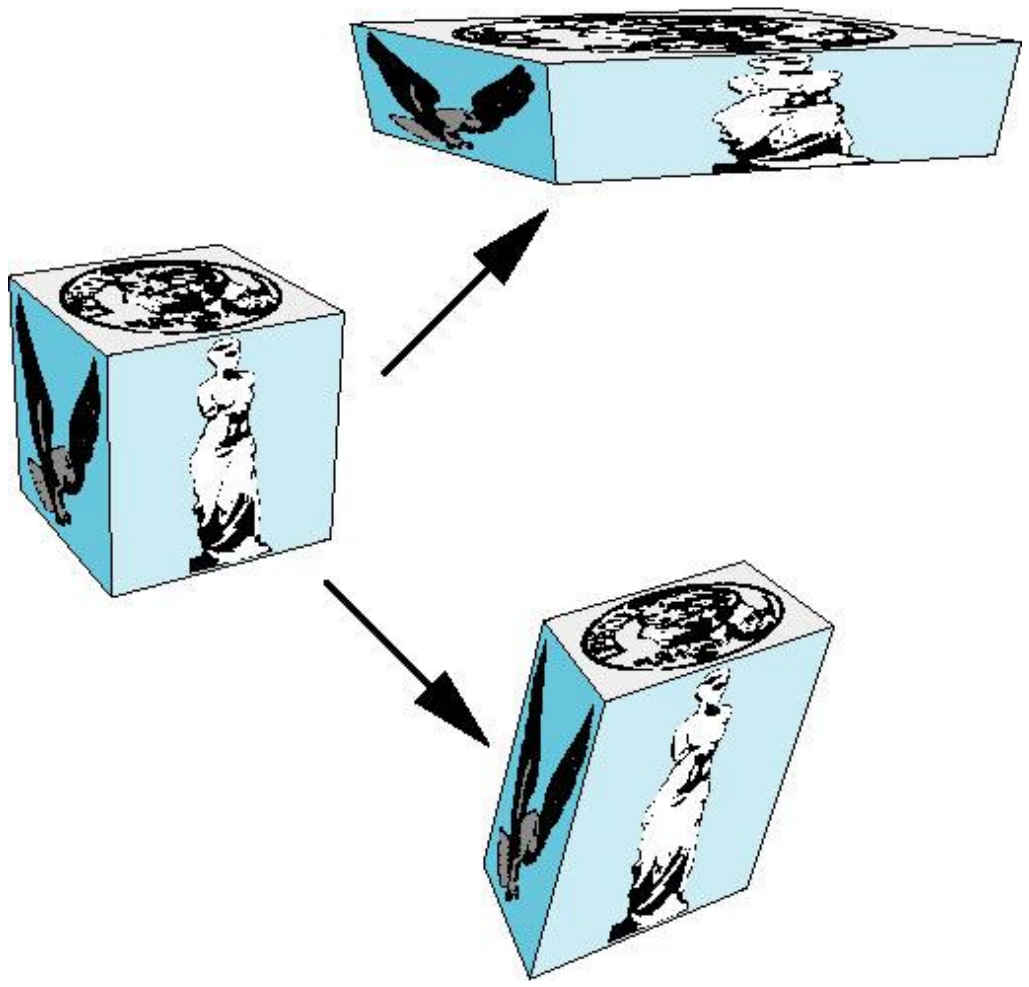
$z' = s_z z$

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



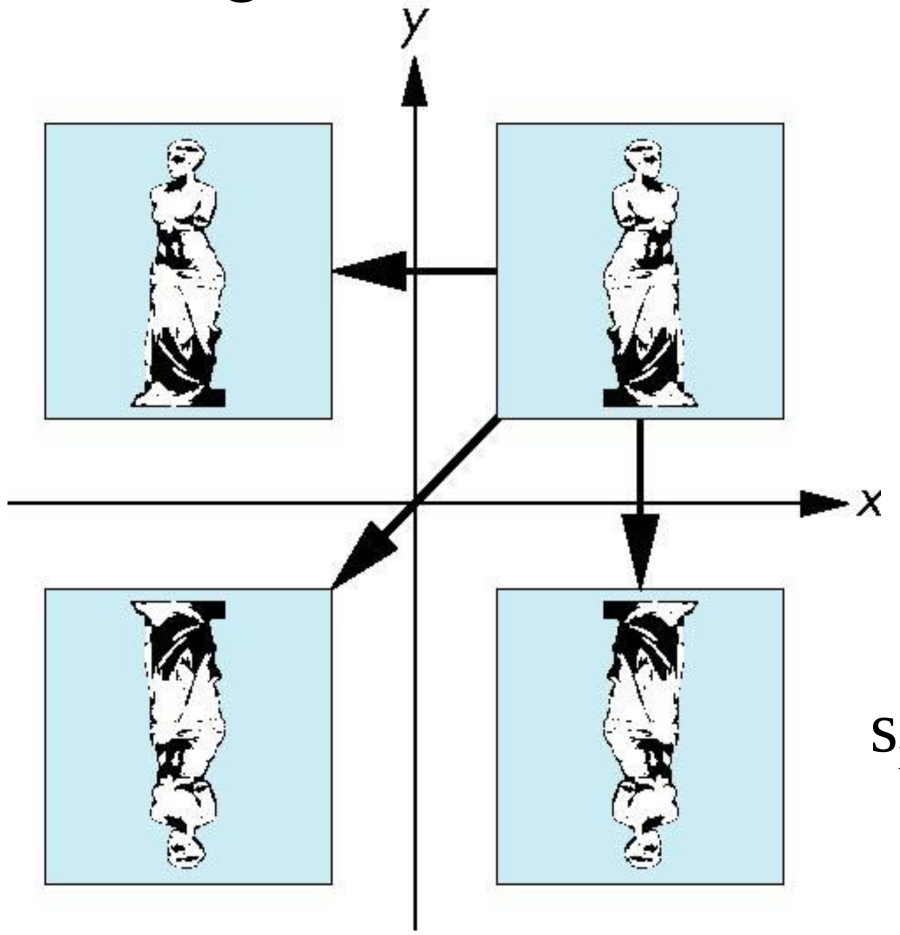
$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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 **M=ABCD** is not significant compared to the cost of computing **Mp** for many vertices **p**
- The difficult part is how to form a desired transformation from the specifications in the application

Order of Transformations

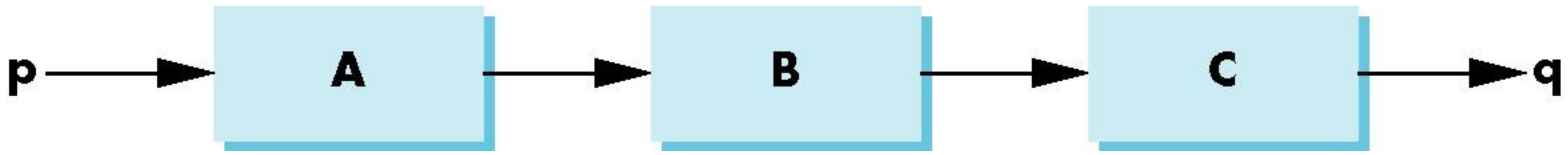
- Note that matrix on the right is the first applied
- Mathematically, the following are equivalent

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

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

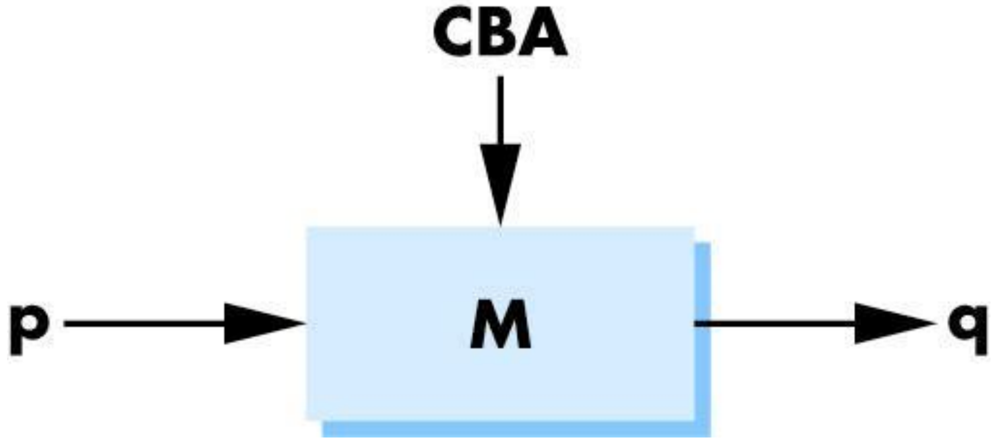
$$\mathbf{p}'^{T} = \mathbf{p}^{T}\mathbf{C}^{T}\mathbf{B}^{T}\mathbf{A}^{T}$$

Application of transformation one at a time



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

Pipeline transformation



$$M = CBA$$

$$q = M\,p$$

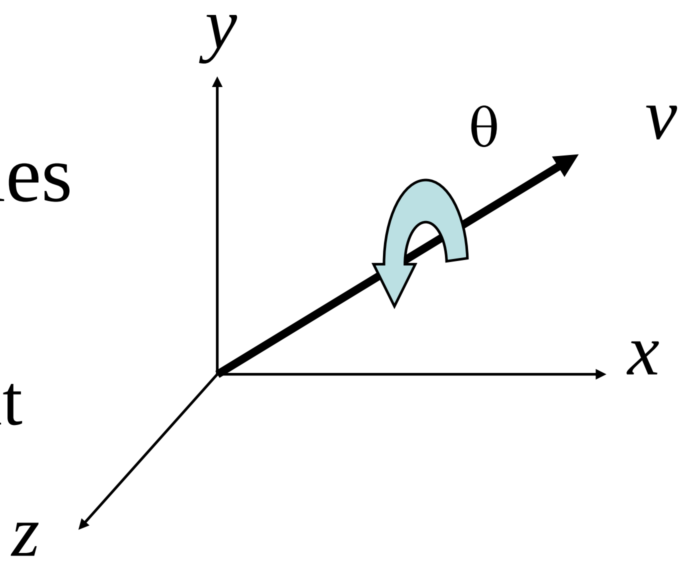
General Rotation About the Origin

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

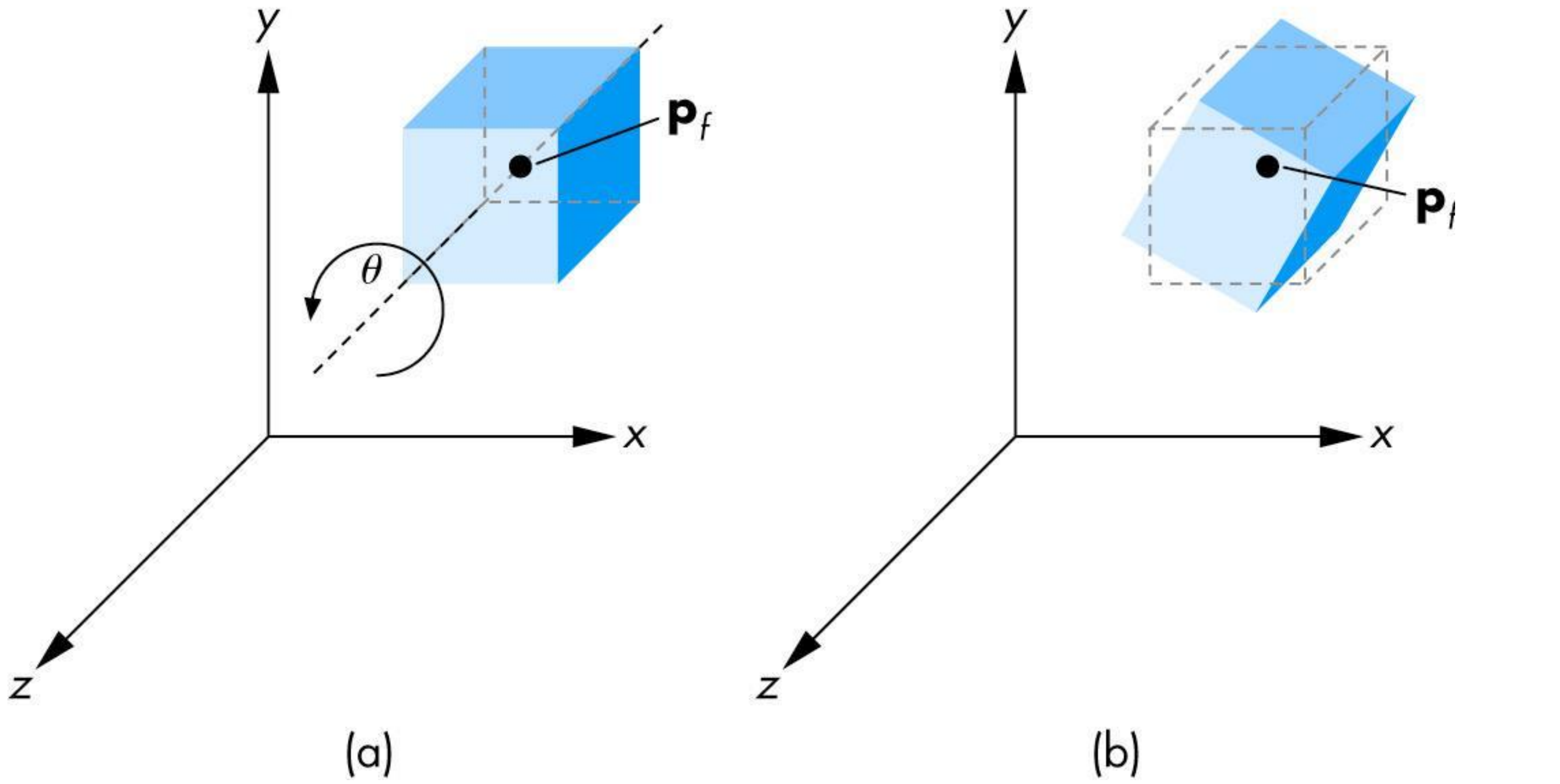
$$\mathbf{R}(\theta) = \mathbf{R}_z(\theta_z)\,\mathbf{R}_y(\theta_y)\,\mathbf{R}_x(\theta_x)$$

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

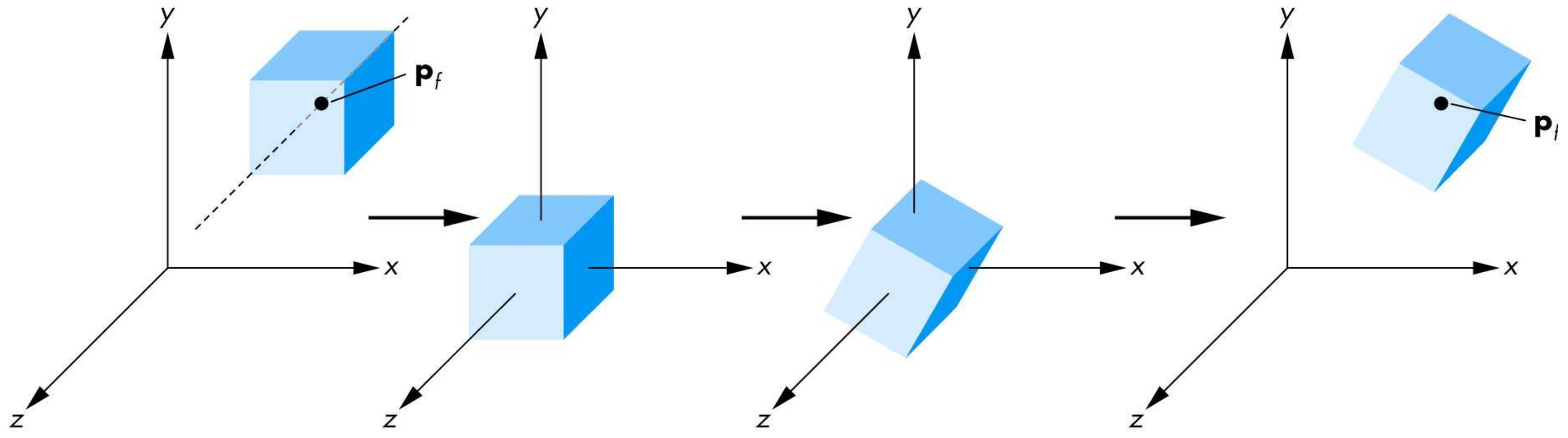
Note that rotations do not commute
We can use rotations in another order but with different angles



Rotation of a cube about its center



Rotation of a cube about its center



Move fixed point to origin

Rotate

Move fixed point back

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

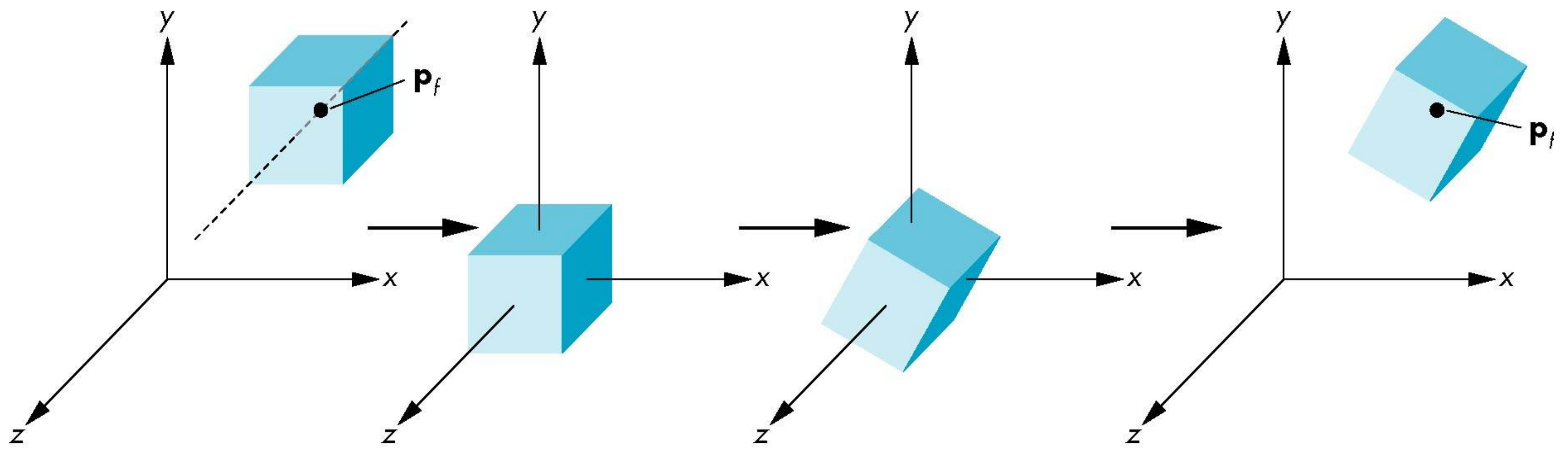
Rotation About a Fixed Point other than the Origin

Move fixed point to origin

Rotate

Move fixed point back

$\mathbf{M} = \mathbf{T}(\mathrm{p}_f)\ \mathbf{R}(\theta)\ \mathbf{T}(-\mathrm{p}_f)$



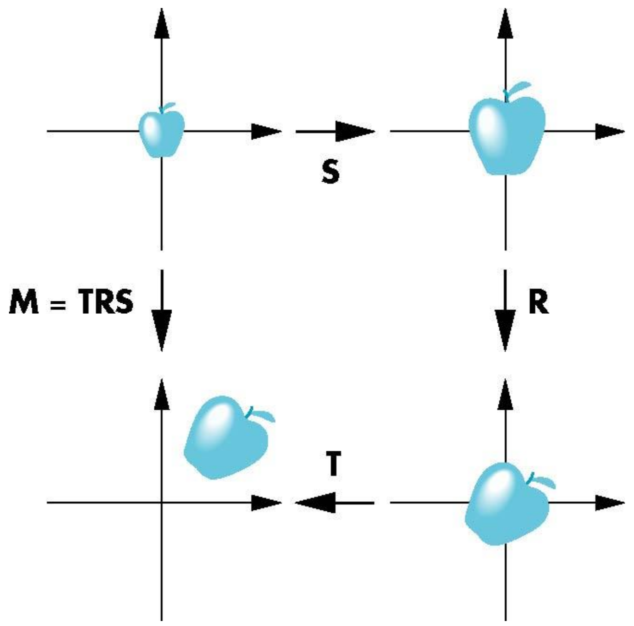
Instancing

- 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

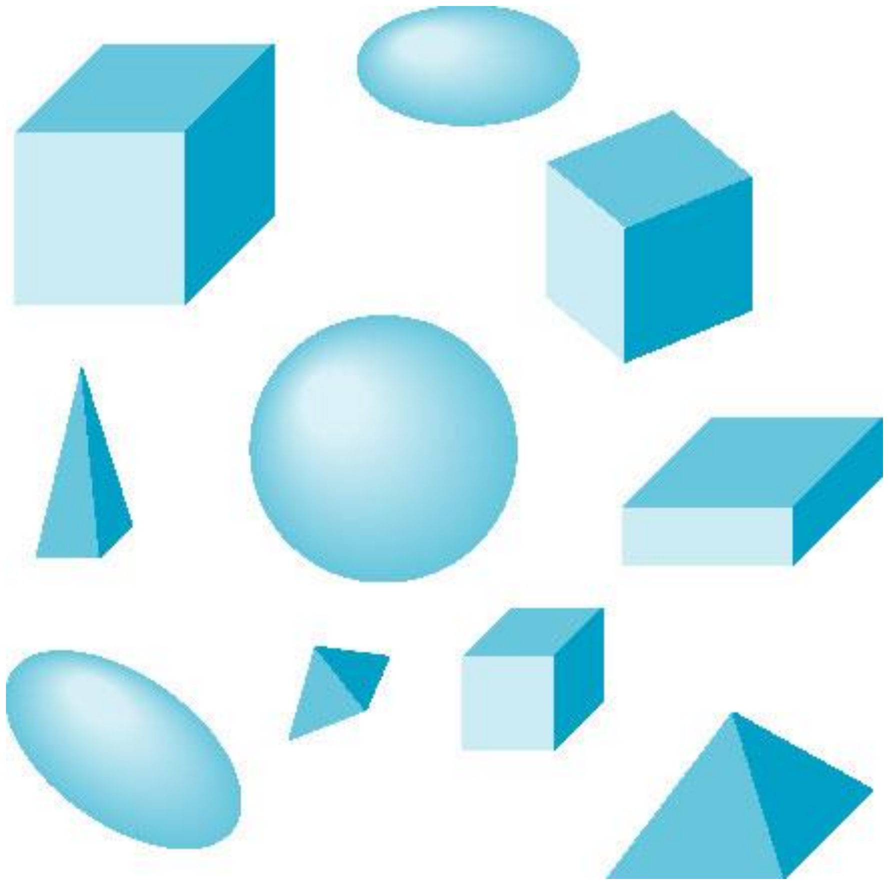
  Scale

  Orient

  Locate

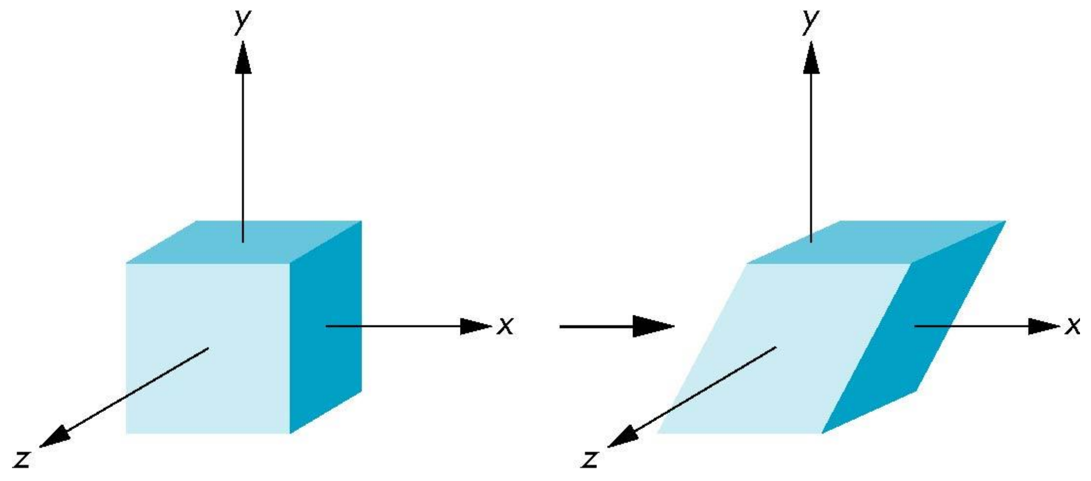


Scene of simple objects



Shear

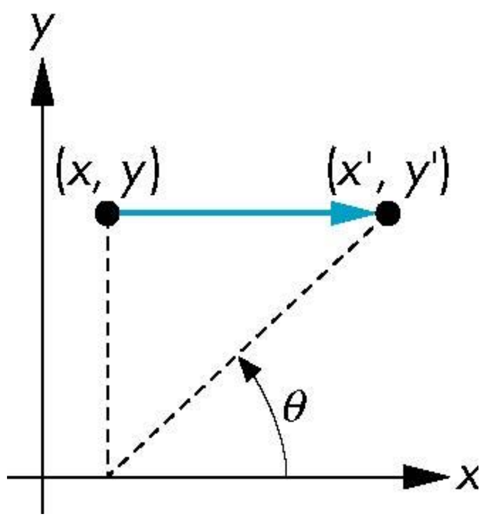
- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions



Shear Matrix

Consider simple shear along *x* axis

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



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

OpenGL Transformations

Objectives

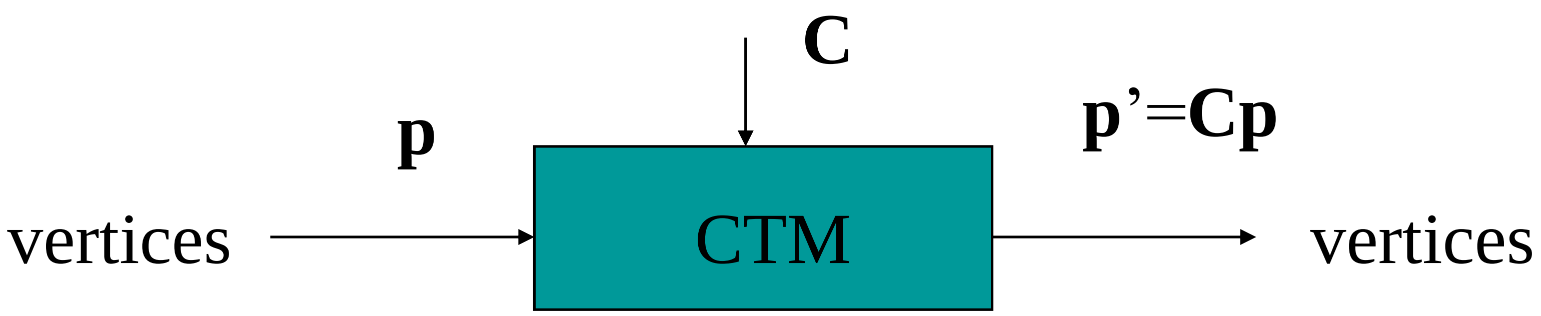
- Learn how to carry out transformations in OpenGL
  - –Rotation
  - –Translation
  - –Scaling
- Introduce OpenGL matrix modes
  - –Model-view
  - –Projection

OpenGL Matrices

- In OpenGL matrices are part of the state
- Multiple types
  - Model-View (**`GL_MODELVIEW`**)
  - Projection (**`GL_PROJECTION`**)
  - Texture (**`GL_TEXTURE`**) (ignore for now)
  - Color(**`GL_COLOR`**) (ignore for now)
- Single set of functions for manipulation
- Select which to manipulated by
  - **`glMatrixMode(GL_MODELVIEW);`**
  - **`glMatrixMode(GL_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 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: $\mathbf{C} \leftarrow \mathbf{C\,R}$
  Postmultiply by a scaling matrix: $\mathbf{C} \leftarrow \mathbf{C\,S}$

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

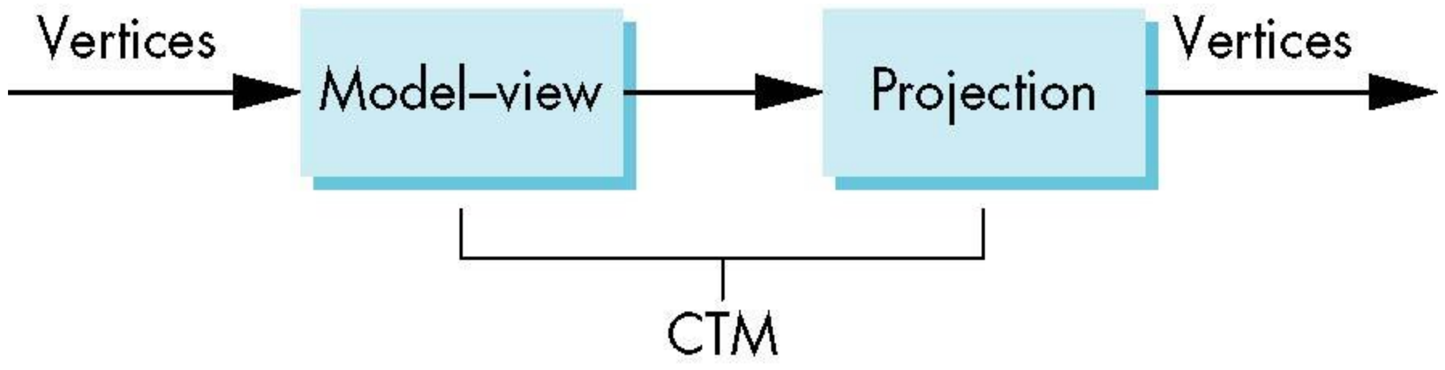
$\mathbf{C} \leftarrow \mathbf{I}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}^{-1}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{R}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$

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 OpenGL

- OpenGL has a model-view and a projection matrix in the pipeline which are concatenated together to form the CTM
- Can manipulate each by first setting the correct matrix mode



Rotation, Translation, Scaling

Load an identity matrix:

```
glLoadIdentity()
```

Multiply on right:

```
glRotatef(theta, vx, vy, vz)
```

`theta` in degrees, (**`vx, vy, vz`**) define axis of rotation

```
glTranslatef(dx, dy, dz)

glScalef( sx, sy, sz)
```

Each has a float (f) and double (d) format (**`glScaled`**)

Example

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

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(1.0, 2.0, 3.0);
glRotatef(30.0, 0.0, 0.0, 1.0);
glTranslatef(-1.0, -2.0, -3.0);
```

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

Arbitrary Matrices

- Can load and multiply by matrices defined in the application program

```
glLoadMatrixf(m)
glMultMatrixf(m)
```

- The matrix **`m`** is a one dimension array of 16 elements which are the components of the desired 4 x 4 matrix stored by columns
- In **`glMultMatrixf`**, **`m`** multiplies the existing matrix on the right

Matrix Stacks

- In many situations we want to save transformation matrices for use later
  - –Traversing hierarchical data structures (Chapter 10)
  - –Avoiding state changes when executing display lists
- OpenGL maintains stacks for each type of matrix
  - –Access present type (as set by **`glMatrixMode)`** by

```
glPushMatrix()
glPopMatrix()
```

Reading Back Matrices

- Can also access matrices (and other parts of the state) by *query* functions

```
glGetIntegerv
glGetFloatv
glGetBooleanv
glGetDoublev
glIsEnabled
```

- For matrices, we use as

```
double m[16];
glGetFloatv(GL_MODELVIEW, m);
```

Using Transformations

- Example: use idle function to rotate a cube and mouse function to change direction of rotation
- Start with a program that draws a cube (`colorcube.c`) in a standard way
  - Centered at origin
  - Sides aligned with axes
  - Will discuss modeling in next lecture

main.c

```
void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
       GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("colorcube");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutIdleFunc(spinCube);
    glutMouseFunc(mouse);
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
}
```

Idle and Mouse callbacks

```
void spinCube()
{
 theta[axis] += 2.0;
 if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
 glutPostRedisplay();
}

 void mouse(int btn, int state, int x, int y)
 {
    if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
            axis = 0;
    if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
            axis = 1;
    if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
            axis = 2;
 }
```

Display callback

```
void display()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity();
   glRotatef(theta[0], 1.0, 0.0, 0.0);
   glRotatef(theta[1], 0.0, 1.0, 0.0);
   glRotatef(theta[2], 0.0, 0.0, 1.0);
   colorcube();
   glutSwapBuffers();
}
```

Note that because of fixed from of callbacks, variables such as `theta` and `axis` must be defined as globals

Camera information is in standard reshape callback

Using the Model-view Matrix

- In OpenGL the model-view matrix is used to
  - –Position the camera
    - Can be done by rotations and translations but is often easier to use **`gluLookAt`**
  - –Build models of objects
- The projection matrix is used to define the view volume and to select a camera lens

Model-view and Projection Matrices

- Although both are manipulated by the same functions, we have to be careful because incremental changes are always made by postmultiplication
  - For example, rotating model-view and projection matrices by the same matrix are not equivalent operations. Postmultiplication of the model-view matrix is equivalent to premultiplication of the projection matrix

Smooth Rotation

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

Incremental Rotation

- Consider the two approaches
  - For a sequence of rotation matrices $\mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_n$, find the Euler angles for each and use $\mathbf{R}_i = \mathbf{R}_{iz} \mathbf{R}_{iy} \mathbf{R}_{ix}$
    - Not very efficient
  - Use the final positions to determine the axis and angle of rotation, then increment only the angle
- Quaternions can be more efficient 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

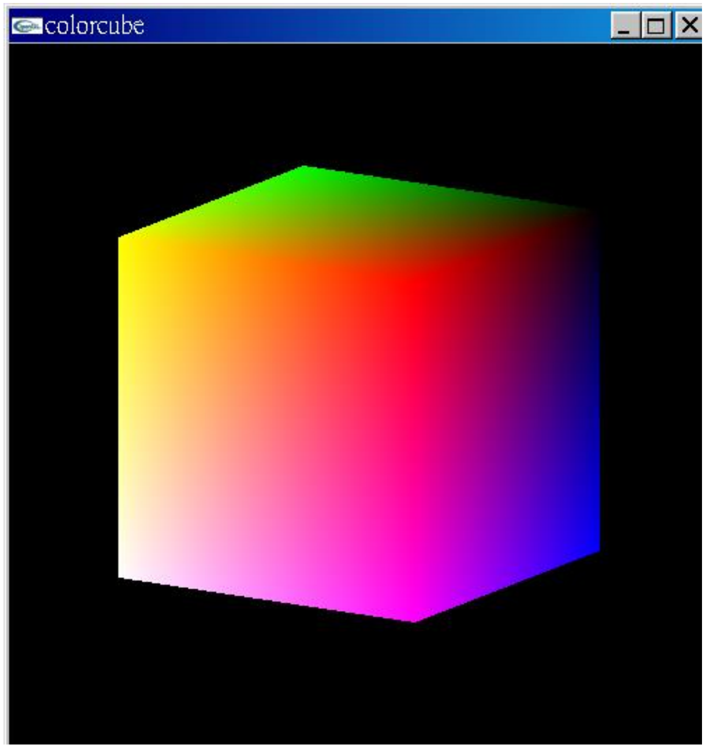
Interfaces

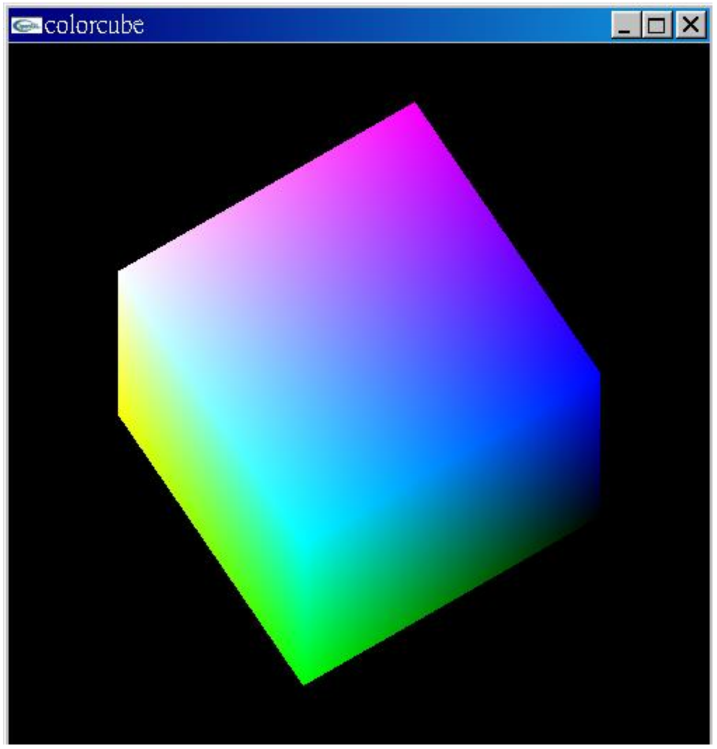
- One of the major problems in interactive computer graphics is how to use two-dimensional devices such as a mouse to interface with three dimensional obejcts
- Example: how to form an instance matrix?
- Some alternatives
  - Virtual trackball
  - 3D input devices such as the spaceball
  - Use areas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

Sample Programs

- Rotating cubes
  - A.8 cube.c
- Rotating cubes using vertex arrays
  - A.9 cubev.c

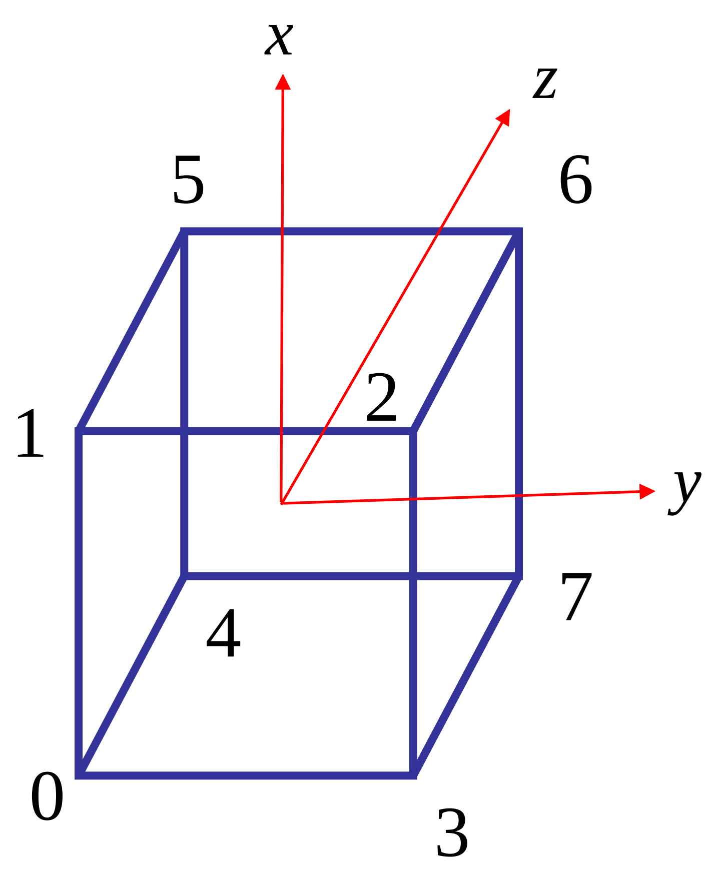
A.8 cube.c (1/5)





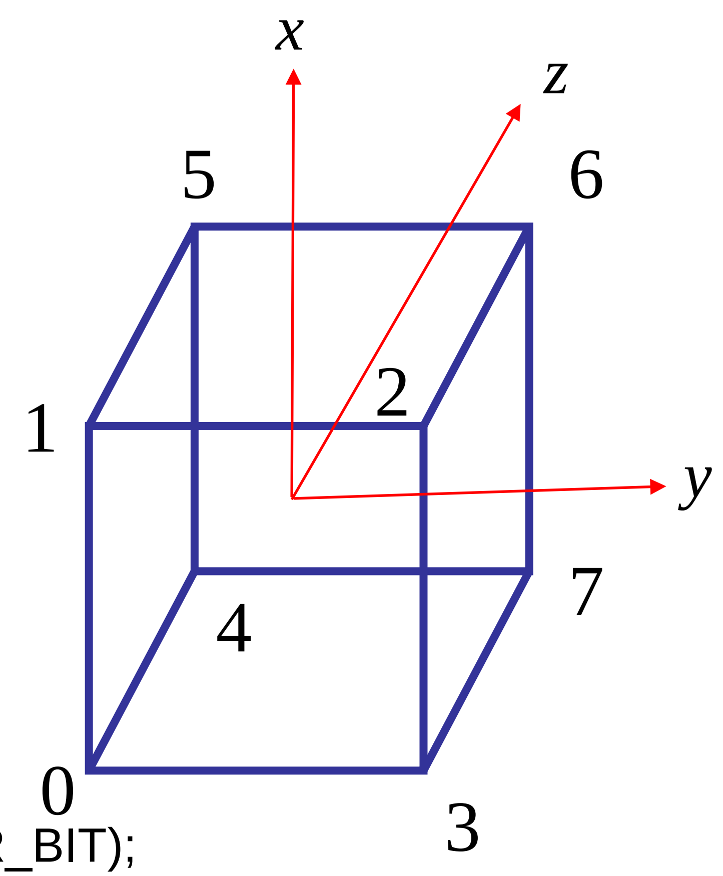
A.8 cube.c (2/5)

```
/* Rotating cube with color interpolation */
#include <GL/glut.h>
GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
          {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
          {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
GLfloat normals[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
          {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
          {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
GLfloat colors[][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
          {1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
          {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
void polygon(int a, int b, int c , int d)
{     /* draw a polygon via list of vertices */
          glBegin(GL_POLYGON);
                    glColor3fv(colors[a]);
                    glNormal3fv(normals[a]);
                    glVertex3fv(vertices[a]);
                    glColor3fv(colors[b]);
                    glNormal3fv(normals[b]);
                    glVertex3fv(vertices[b]);
                    glColor3fv(colors[c]);
                    glNormal3fv(normals[c]);
                    glVertex3fv(vertices[c]);
                    glColor3fv(colors[d]);
                    glNormal3fv(normals[d]);
                    glVertex3fv(vertices[d]);
          glEnd();
}
```



A.8 cube.c (3/5)

```
void colorcube(void)
{
/* map vertices to faces */
            polygon(0,3,2,1);
            polygon(2,3,7,6);
            polygon(0,4,7,3);
            polygon(1,2,6,5);
            polygon(4,5,6,7);
            polygon(0,1,5,4);
}
static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 2;
void display(void)
{
/* display callback, clear frame buffer and z buffer,
   rotate cube and draw, swap buffers */

 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
            glLoadIdentity();
            glRotatef(theta[0], 1.0, 0.0, 0.0);
            glRotatef(theta[1], 0.0, 1.0, 0.0);
            glRotatef(theta[2], 0.0, 0.0, 1.0);
 colorcube();
 glFlush();
 glutSwapBuffers();
}
```



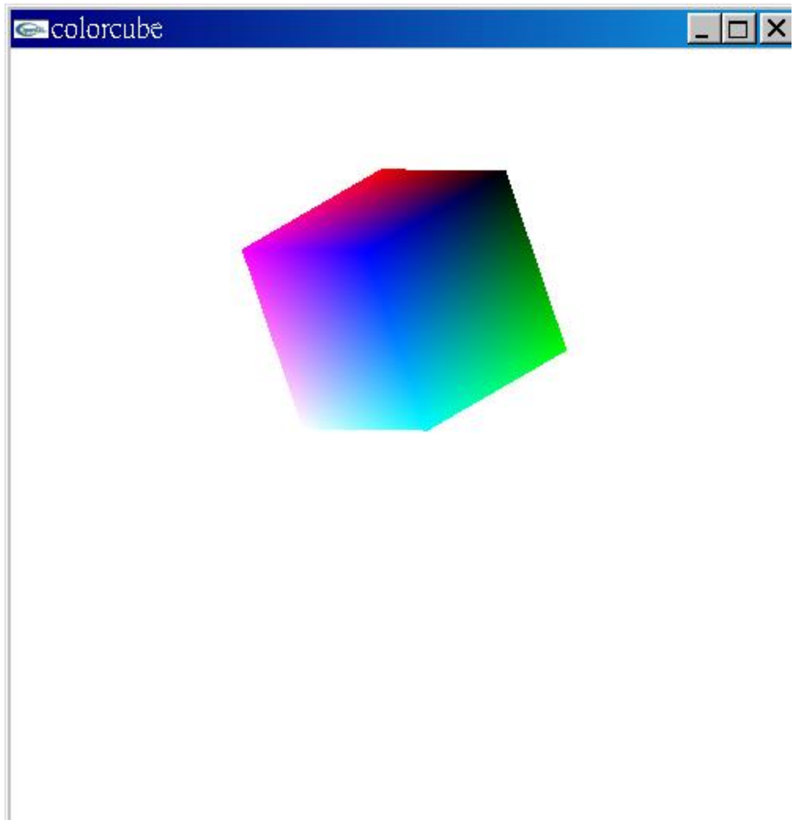
A.8 cube.c (4/5)

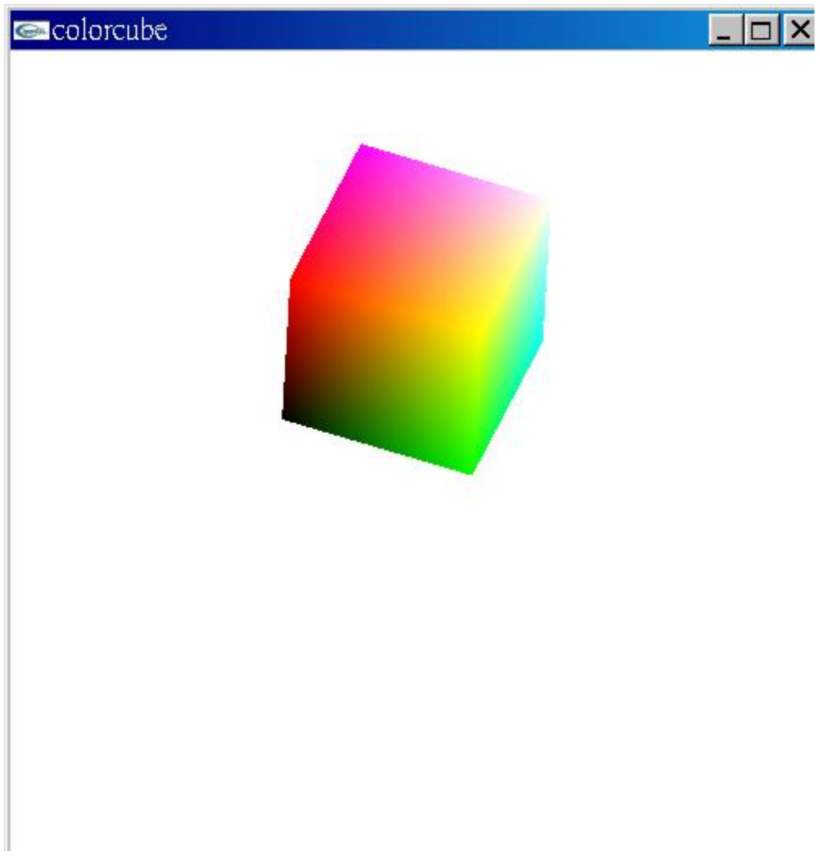
```
void spinCube()
{

/* Idle callback, spin cube 2 degrees about selected axis */

        theta[axis] += 2.0;
        if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
        /* display(); */
        glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{

/* mouse callback, selects an axis about which to rotate */

        if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
        if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
        if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
}
```

A.8 cube.c (5/5)

```
void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
             2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    else
        glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
            2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}
Void main(int argc, char **argv)
{
    glutInit(&argc, argv);
/* need both double buffering and z buffer */
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("colorcube");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutIdleFunc(spinCube);
    glutMouseFunc(mouse);
    glEnable(GL_DEPTH_TEST); /* Enable hidden--surface--removal */
    glutMainLoop();
}
```

A.9 cubev.c (1/6)





A.9 cubev.c (2/6)

```
#include <GL/glut.h>
  GLfloat vertices[] = {-1.0,-1.0,-1.0,1.0,-1.0,-1.0,
              1.0,1.0,-1.0, -1.0,1.0,-1.0, -1.0,-1.0,1.0,
              1.0,-1.0,1.0, 1.0,1.0,1.0, -1.0,1.0,1.0};

   GLfloat colors[] = {0.0,0.0,0.0,1.0,0.0,0.0,
              1.0,1.0,0.0, 0.0,1.0,0.0, 0.0,0.0,1.0,
              1.0,0.0,1.0, 1.0,1.0,1.0, 0.0,1.0,1.0};

   GLubyte cubeIndices[]={0,3,2,1,2,3,7,6,0,4,7,3,1,2,6,5,4,5,6,7,0,1,5,4};



static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 2;
```

A.9 cubev.c (3/6)

```
void display(void)
{


/* display callback, clear frame buffer and z buffer,
   rotate cube and draw, swap buffers */

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glLoadIdentity();
    gluLookAt(1.0,1.0,1.0,0.0,0.0,0.0,0.0,1.0,0.0);
    glTranslatef(0.0, 3.0, 0.0);
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    glColorPointer(3,GL_FLOAT, 0, colors);
    glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndices);
    glutSwapBuffers();
}
```

A.9 cubev.c (4/6)

```
void spinCube()
{

/* Idle callback, spin cube 2 degrees about selected axis */

        theta[axis] += 2.0;
        if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
        glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{

/* mouse callback, selects an axis about which to rotate */

        if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
        if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
        if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
}
```

A.9 cubev.c (5/6)

```
void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-4.0, 4.0, -3.0 * (GLfloat) h / (GLfloat) w,
            5.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    else
        glOrtho(-4.0 * (GLfloat) w / (GLfloat) h,
            4.0 * (GLfloat) w / (GLfloat) h, -3.0, 5.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}
```

A.9 cubev.c (6/6)

```
void main(int argc, char **argv)
{


/* need both double buffering and z buffer */

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("colorcube");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutIdleFunc(spinCube);
    glutMouseFunc(mouse);
    glEnable(GL_DEPTH_TEST); /* Enable hidden--surface--removal */
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glColorPointer(3,GL_FLOAT, 0, colors);
    glClearColor(1.0,1.0,1.0,1.0);
    glColor3f(1.0,1.0,1.0);
    glutMainLoop();
}
```