Lecture 5

Representations, Geometry, Homogenous Coordinates

CS 354 Computer Graphics
http://www.cs.utexas.edu/~bajaj/
Department of Computer Science
University of Texas at Austin

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

Geometry Elements

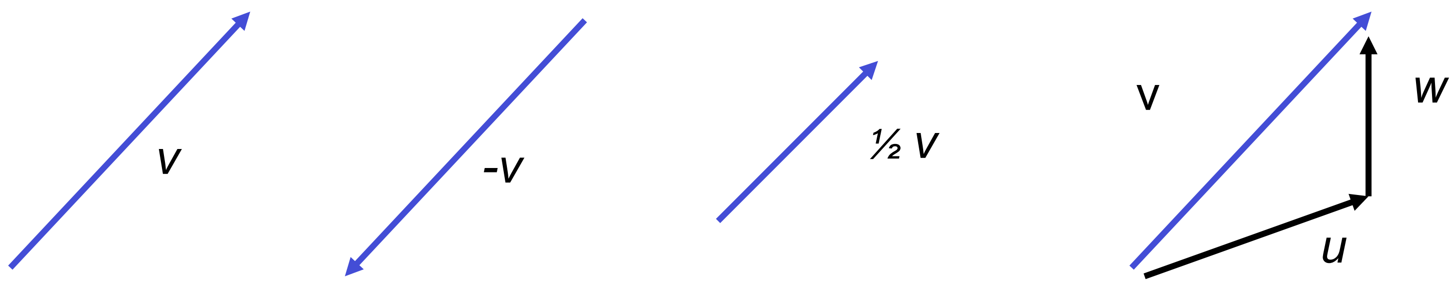
- Need three basic elements in geometry
  - Scalars, Vectors, Points
- Scalars can be defined as members of sets which can be combined by two operations (addition and multiplication) obeying some fundamental axioms (associativity, commutivity, inverses)
- Examples include the real and complex number 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



Figures from *Ed Angel: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

Linear Vector Spaces

- Mathematical system for manipulating vectors
- Operations
  - Scalar-vector multiplication $u=2v$
  - Vector-vector addition: $w=u+v$
- Expressions such as

  $v=u+2w-3r$

Make sense in a vector space

Dimension of Vector Spaces

- 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

$$v = a_1 v_1 + a_2 v_2 + \ldots + a_n v_n$$

where the $\{a_i\}$ are unique

Linear Independence

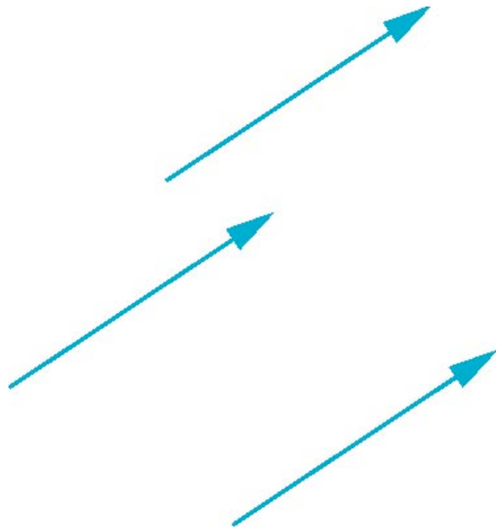
- A set of vectors $v_1, v_2, \ldots, v_n$ is *linearly independent* if

$$a_1 v_1 + a_2 v_2 + .. \, a_n v_n = 0 \text{ iff } a_1 = a_2 = \ldots = 0$$

- 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

Vectors lack Position

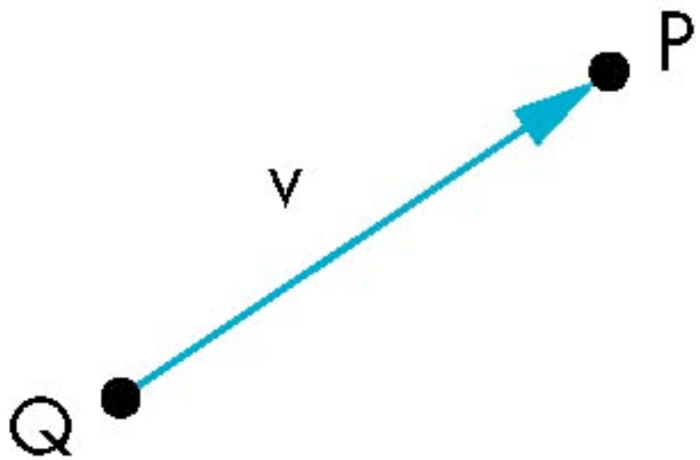
- These vectors are identical
  - Same length and magnitude



- 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

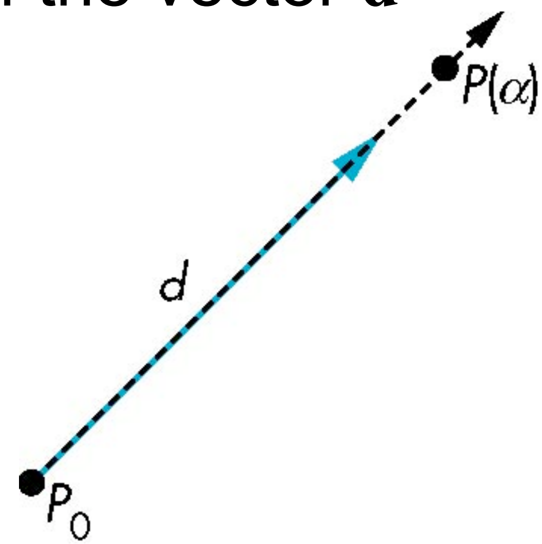
P=v+Q

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



Figures from *Ed Angel: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

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:

$$x(a) = (1-a)x_0 + ax_1$$

$$y(a) = (1-a)y_0 + ay_1 \quad \text{and} \quad 0 <= a <= 1$$

Rays and Line Segments

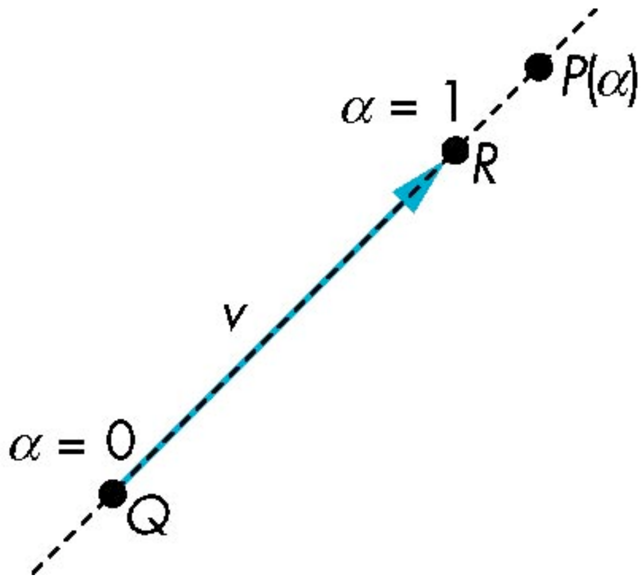
- If a >= 0, then P(a) is the *ray* leaving P_0 in the direction **d**

If we use two points to define v, then

$$P(a) = Q + a(R-Q) = Q + av$$

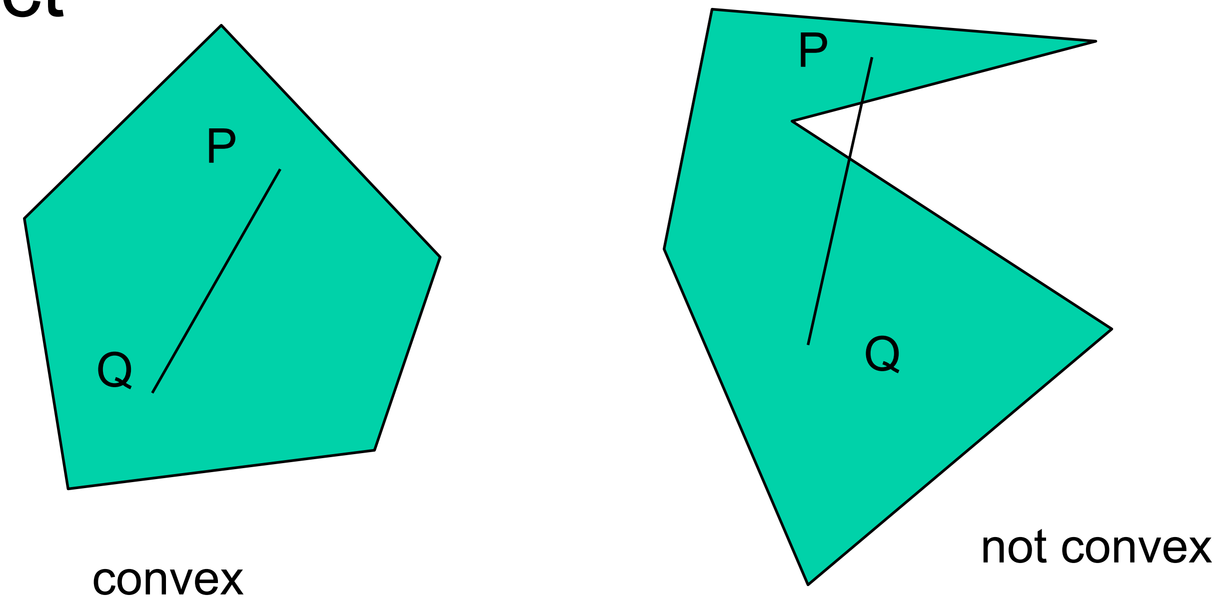
$$= aR + (1-a)Q$$

For $0 <= a <= 1$ 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



Figures from *Ed Angel: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

Affine Sums

- Consider the "sum"

$P=a_1P_1+a_2P_2+.....+a_nP_n$

Can show by induction that this sum makes sense iff

$a_1+a_2+.....a_n=1$

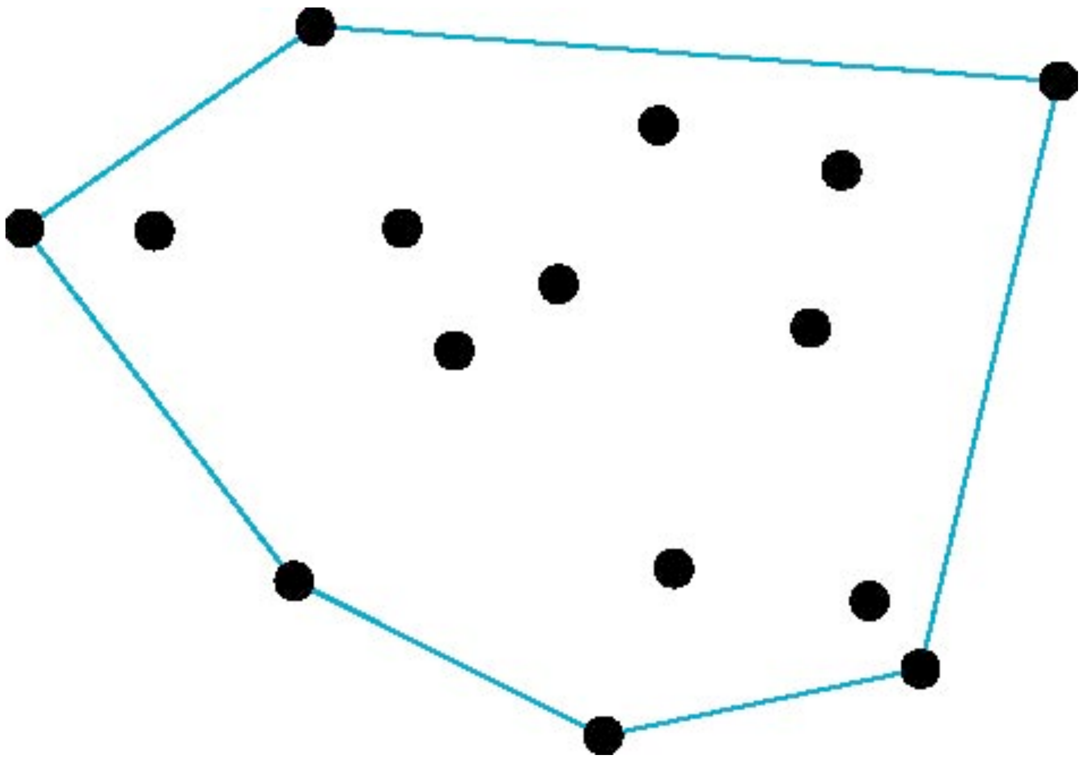
in which case we have the *affine sum* of the points $P_1,P_2,.....P_n$

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

Figures from *Ed Angel: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

Convex Hull

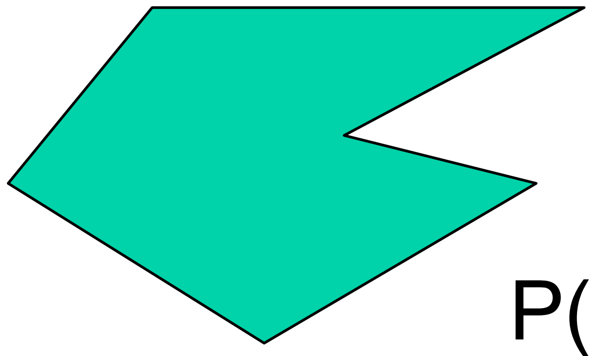
- Smallest convex object containing $P_1,P_2,\ldots..P_n$
- Formed by "shrink wrapping" points



Curves & Surfaces

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

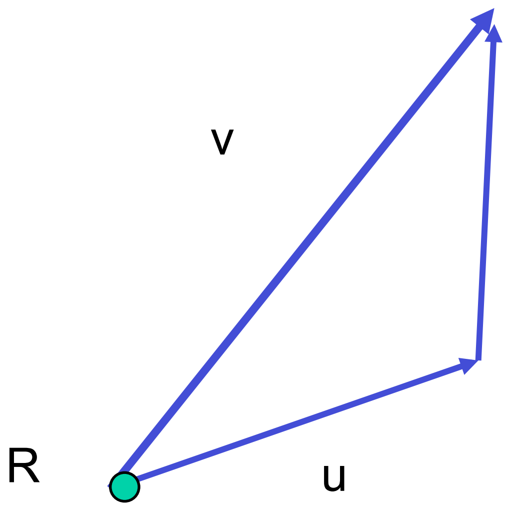
P(a)

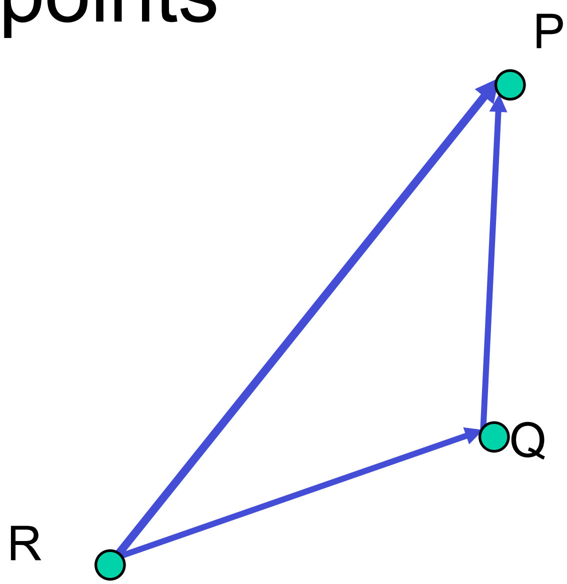


P(a, b)

Planes

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

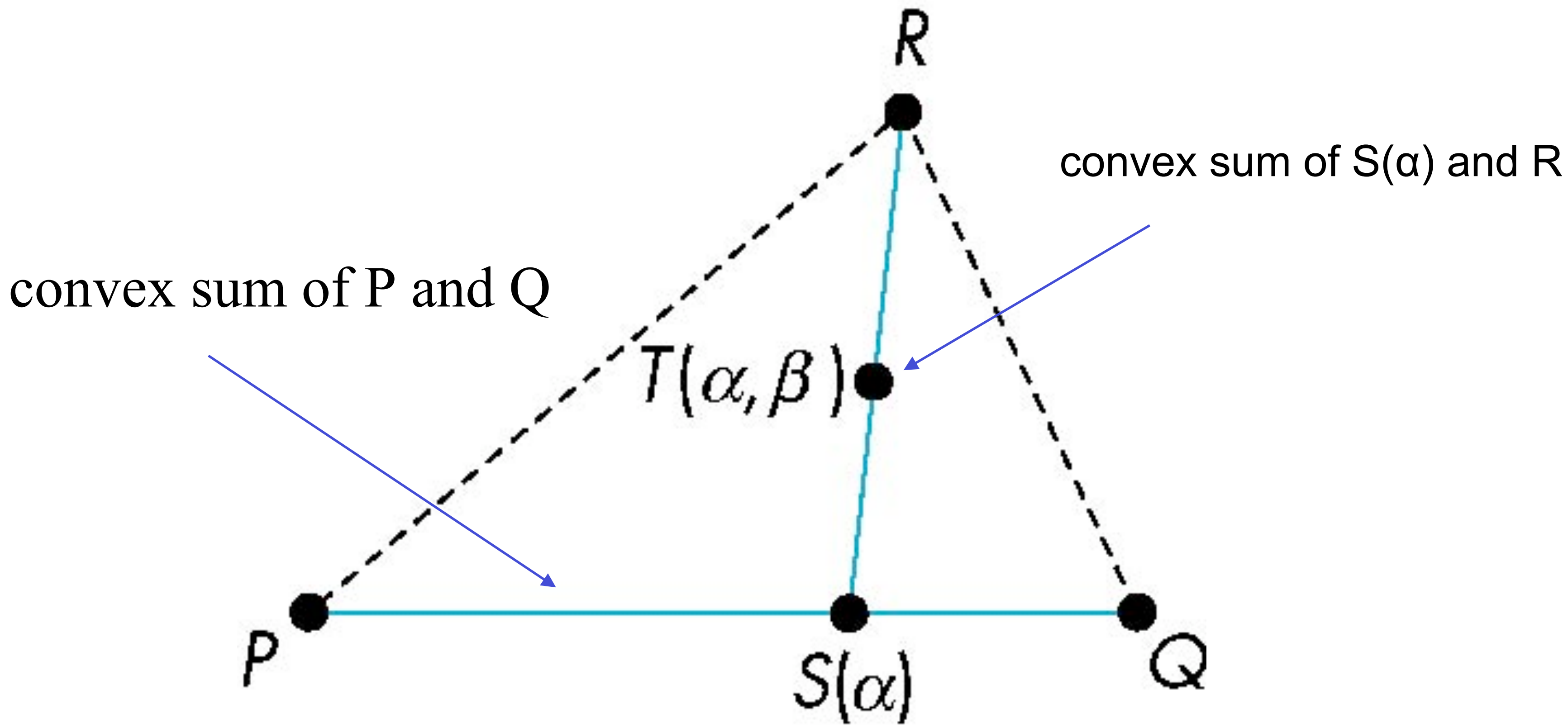




P(a,b)=R+au+bv

P(a,b)=R+a(Q-R)+b(P-Q)

Triangles



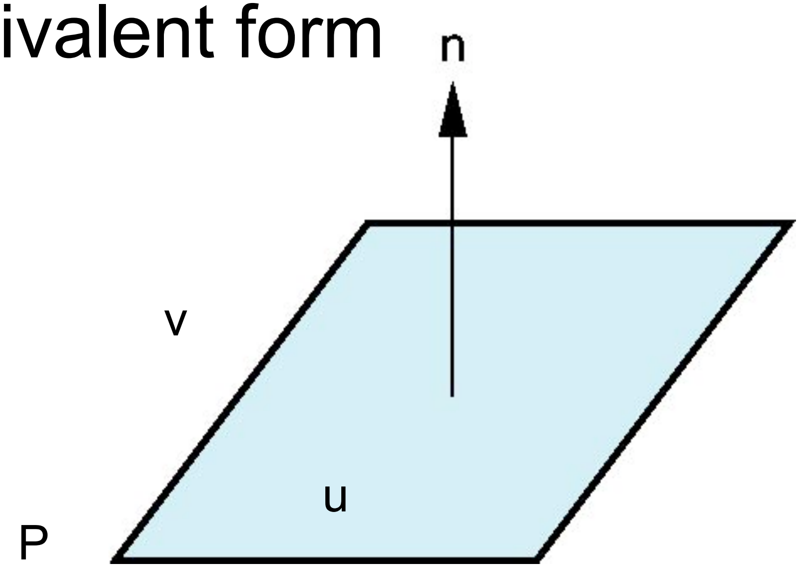
for 0<=α,β<=1, we get all points in triangle

Normals

- Every plane has a vector n normal (perpendicular, orthogonal) to it
- From point-two vector form $P(a,b)=R+au+bv$, we know we can use the cross product to find

$n = u \times v$ and the equivalent form

$(P(a)-P) \times n=0$



Figures from *Ed Angel: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

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 $v = a_1 v_1 + a_2 v_2 + \ldots + a_n v_n$
- The list of scalars $\{a_1, a_2, \ldots a_n\}$ is the *representation* of v with respect to the given basis
- We can write the representation as a row or column array of scalars

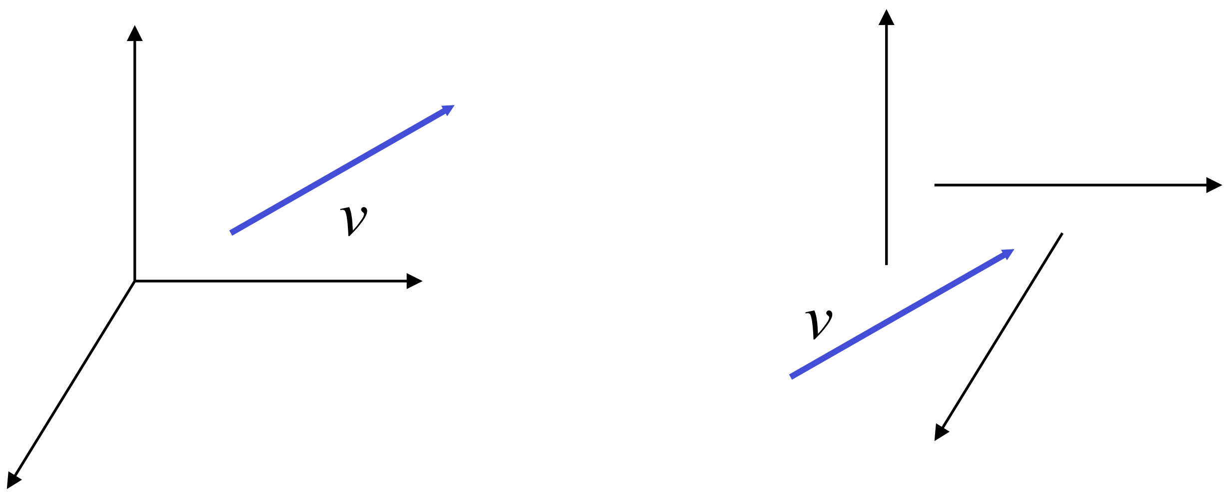
$$\mathbf{a} = [a_1 \quad a_2 \quad \ldots \quad a_n]^T = \begin{bmatrix} a_1 \\ a_2 \\ . \\ a_n \end{bmatrix}$$

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

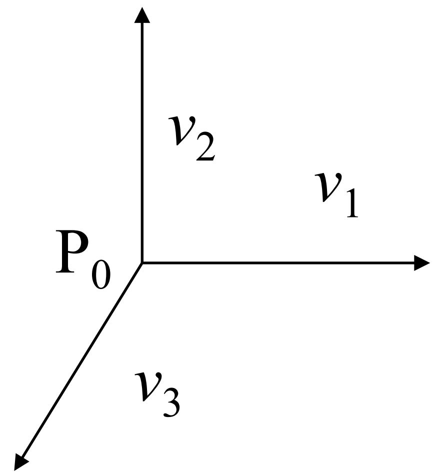
- Which is correct?



- 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 is a Frame

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

$$v = a_1 v_1 + a_2 v_2 + \ldots. + a_n v_n$$

- Every point can be written as

$$P = P_0 + b_1 v_1 + b_2 v_2 + \ldots. + b_n v_n$$

Points & Vectors

Consider the point and the vector

$$P = P_0 + b_1 v_1 + b_2 v_2 + \ldots + b_n v_n$$

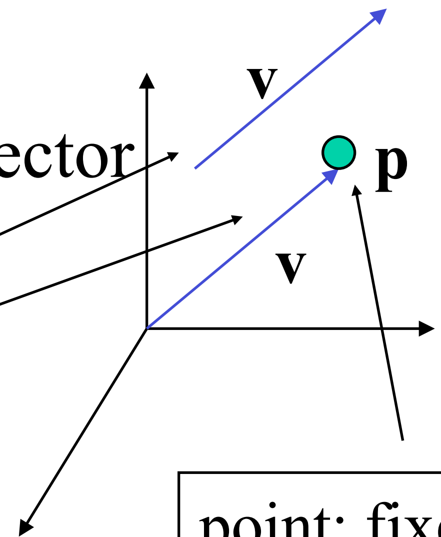
$$v = a_1 v_1 + a_2 v_2 + \ldots + a_n v_n$$

They appear to have the similar representations

$\mathbf{p}=[b_1\ b_2\ b_3]$ $\mathbf{v}=[a_1\ a_2\ a_3]$

which confuses the point with the vector

A vector has no position



Vector can be placed anywhere

point: fixed

Homogenizing Representation

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

$\mathrm{v} = a_1 v_1 + a_2 v_2 + a_3 v_3 = [a_1\ a_2\ a_3\ 0\ ]\ [v_1\ v_2\ v_3\ \ P_0]^{\mathrm{T}}$

$P = P_0 + b_1 v_1 + b_2 v_2 + b_3 v_3 = [b_1\ b_2\ b_3\ 1\ ]\ [v_1\ v_2\ v_3\ \ P_0]^{\mathrm{T}}$

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

$\mathbf{v} = [a_1\ a_2\ a_3\ 0\ ]^{\mathrm{T}}$

$\mathbf{p} = [b_1\ b_2\ b_3\ 1\ ]^{\mathrm{T}}$

Homogenous 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 $w>0$) by

$x = x'/w$

$y = y'/w$

$z = z'/w$

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]

Homogenous Coords & CG

- 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

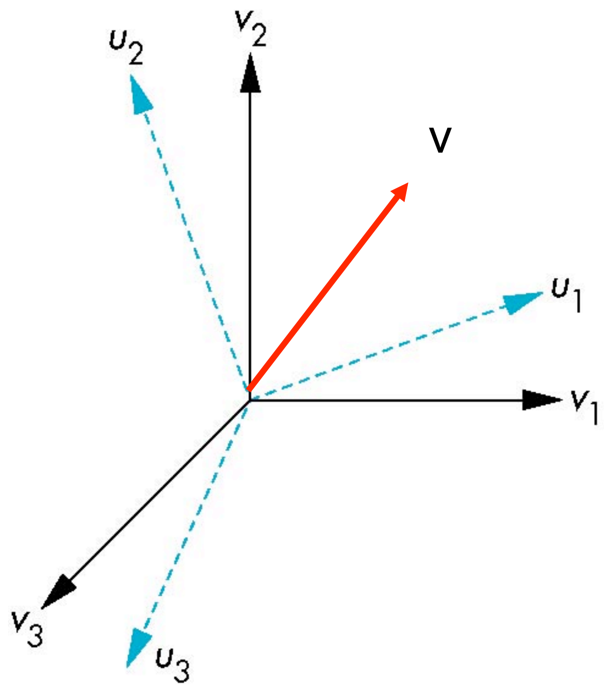
$$\mathbf{a}=[a_1\ a_2\ \ a_3\ ]$$
$$\mathbf{b}=[b_1\ \ b_2\ \ b_3]$$

where

$$v=a_1v_1+a_2v_2+a_3v_3=[a_1\ a_2\ a_3]\ [v_1\ v_2\ v_3]^T$$
$$=b_1u_1+b_2u_2+b_3u_3=[b_1\ b_2\ b_3]\ [u_1\ u_2\ u_3]^T$$

Representing 2nd basis in terms of 1st

Each of the basis vectors, u1,u2, u3, are vectors that can be represented in terms of the first basis v1,v2, v3,



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

Matrix Form

The coefficients define a 3 x 3 matrix

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

and the bases can be related by

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

see text (p179-) for numerical examples

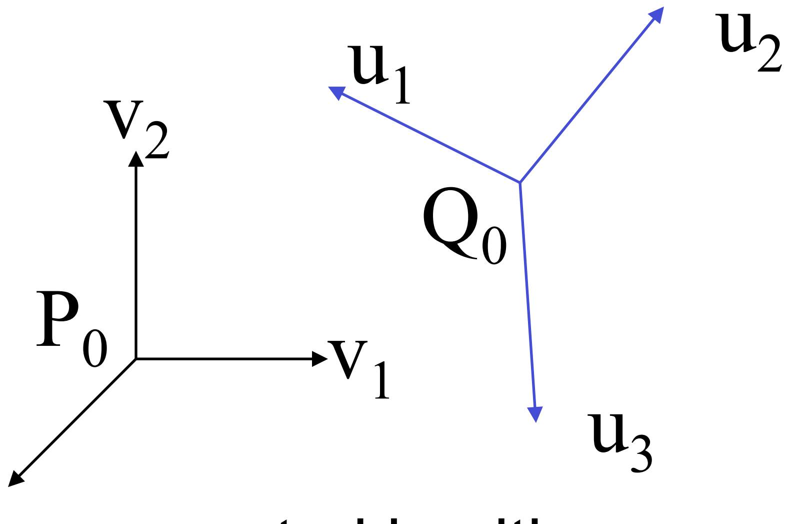
Change of Frames

- 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

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

defining a 4 x 4 matrix

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

Working with Representations

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

$\mathbf{a}=[a_1\ a_2\ \ a_3\ a_4\ ]$ in the first frame
$\mathbf{b}=[b_1\ b_2\ \ b_3\ b_4\ ]$ in the second frame

where $a_4=b_4=1$ for points and $a_4=b_4=0$ 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

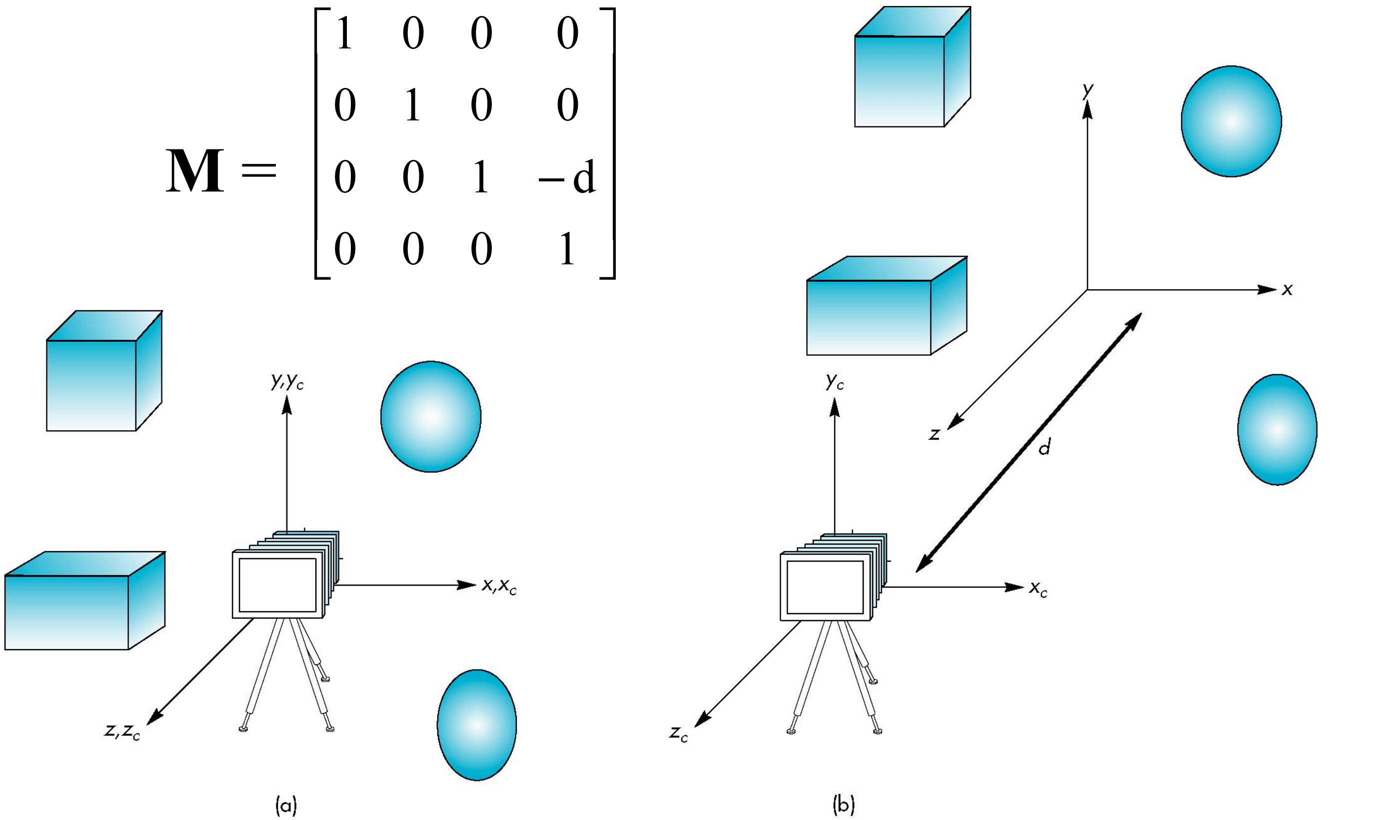
The World and Camera Frames

- 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

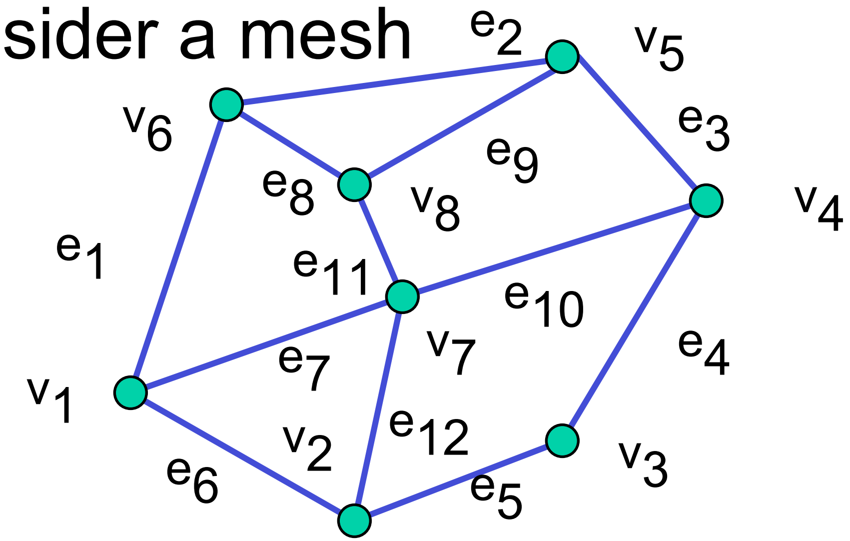
If objects are on both sides of z=0, we must move camera frame

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



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

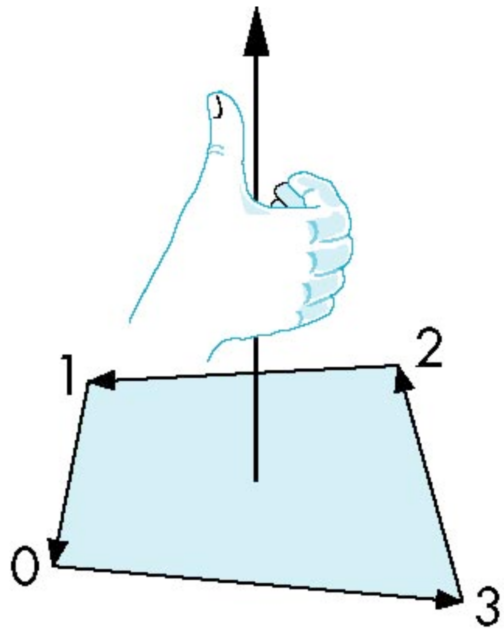
- Define each polygon by the geometric locations of its vertices
- Leads to OpenGL code such as

```
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 & 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

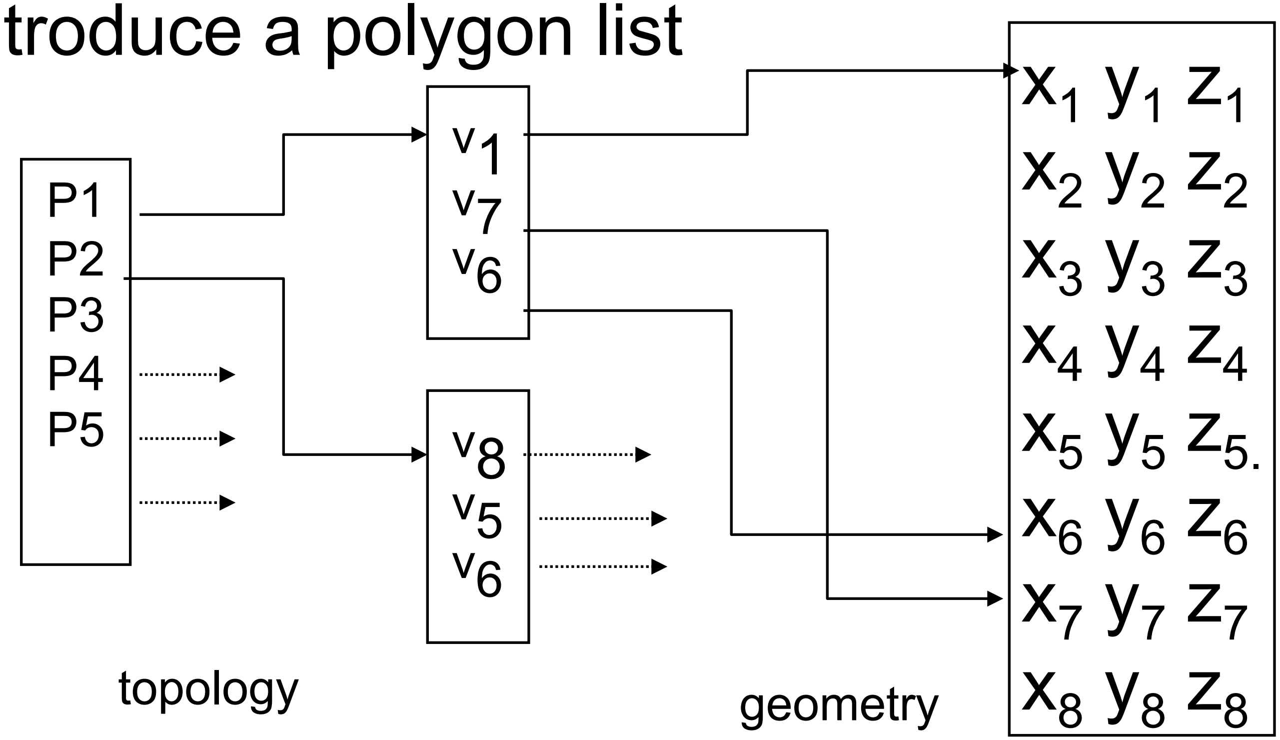


Geometry vs Topology

- 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

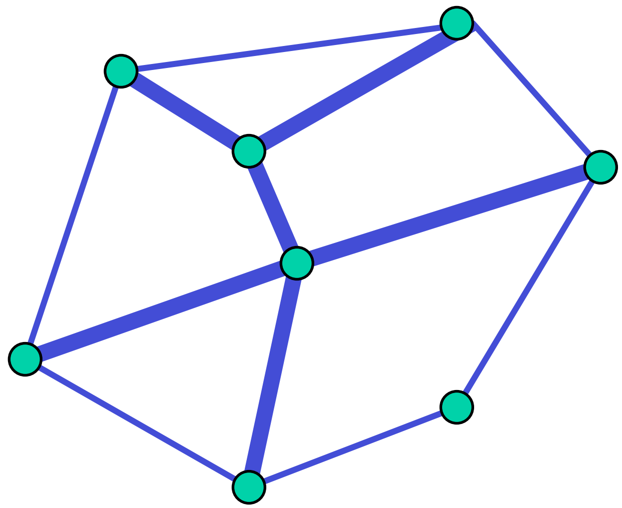
Vertex Lists

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



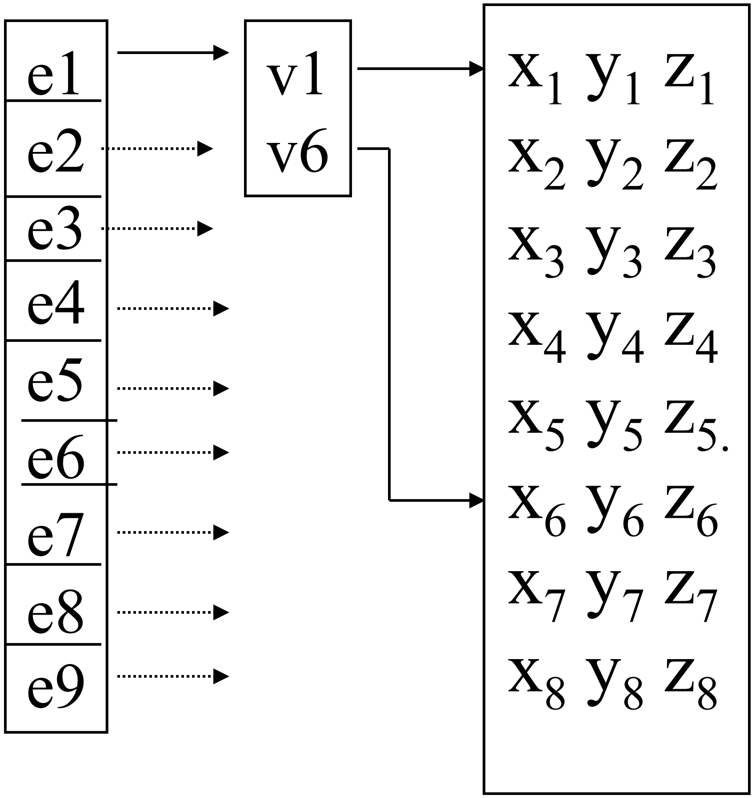
Shared Edges

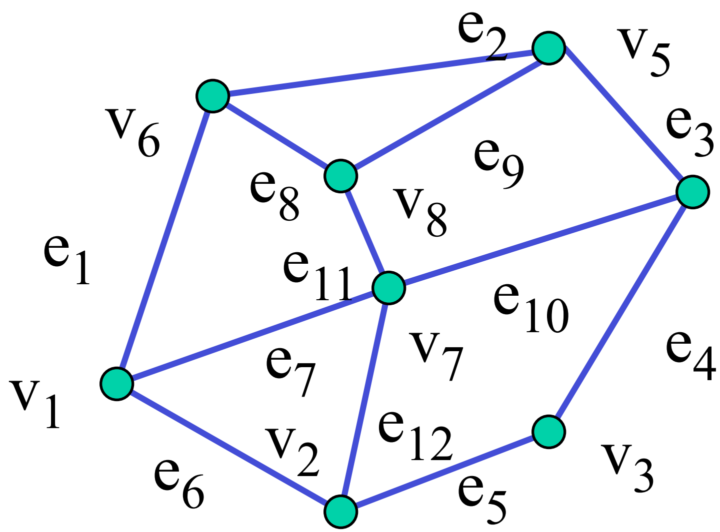
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- Can store mesh by *edge list*

Edge Lists





Note polygons are not represented

Modeling a Cube

Define global arrays for vertices and colors

```
typedef vex3 point3;
point3 vertices[] = {point3(-1.0,-1.0,-1.0),
  point3(1.0,-1.0,-1.0), point3(1.0,1.0,-1.0),
  point3(-1.0,1.0,-1.0), point3(-1.0,-1.0,1.0),
  point3(1.0,-1.0,1.0), point3(1.0,1.0,1.0),
  point3(-1.0,1.0,1.0)};

typedef vec3 color3;
color3 colors[] = {color3(0.0,0.0,0.0),
color3(1.0,0.0,0.0), color3(1.0,1.0,0.0),
color(0.0,1.0,0.0), color3(0.0,0.0,1.0),
color3(1.0,0.0,1.0), color3(1.0,1.0,1.0),
color3(0.0,1.0,1.0});
```

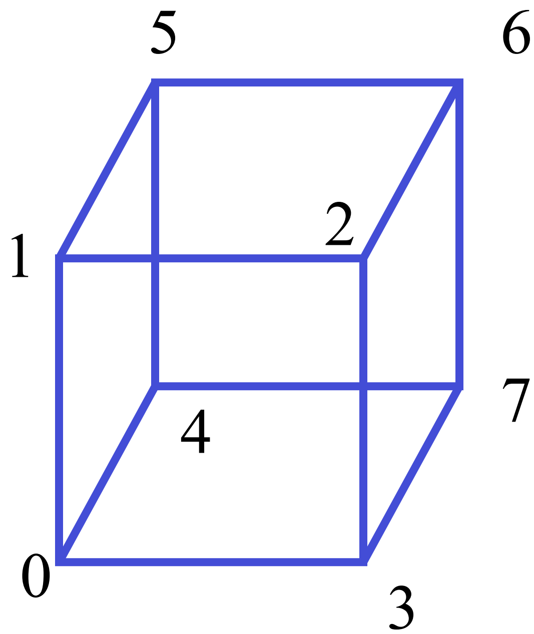
Drawing a Triangle from a List of Indices

Draw a triangle from a list of indices into the array **`vertices`** and assign a color to each index

```
void triangle(int a, int b, int c, int d)
{
    vcolors[i] = colors[d];
    position[i] = vertices[a];
    vcolors[i+1] = colors[d]);
    position[i+1] = vertices[b];
    vcolors[i+2] = colors[d];
    position[i+2] = vertices[c];
    i+=3;
}
```

Drawing a cube from Faces

```
void 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

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);
```

data array

3d arrays   stored as floats   data contiguous

```
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

- Old Method:

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

Draws cube with 1 function call!!

- Problem is that although we avoid many function calls, data are still on client side
- Solution:
  - no immediate mode
  - Vertex buffer object
  - Use glDrawArrays

Rotating the Cube

- Full example
- Model Colored Cube
- Use 3 button mouse to change direction of rotation
- Use idle function to increment angle of rotation

Cube Vertices

```
// Vertices of a unit cube centered at origin
//  sides aligned with axes
point4 vertices[8] = {
point4( -0.5, -0.5,  0.5, 1.0 ),
point4( -0.5,  0.5,  0.5, 1.0 ),
point4(  0.5,  0.5,  0.5, 1.0 ),
point4(  0.5, -0.5,  0.5, 1.0 ),
point4( -0.5, -0.5, -0.5, 1.0 ),
point4( -0.5,  0.5, -0.5, 1.0 ),
point4(  0.5,  0.5, -0.5, 1.0 ),
point4(  0.5, -0.5, -0.5, 1.0 )
};
```

Colors

```
// RGBA colors
color4 vertex_colors[8] = {
color4( 0.0, 0.0, 0.0, 1.0 ),  // black
color4( 1.0, 0.0, 0.0, 1.0 ),  // red
color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
color4( 0.0, 1.0, 0.0, 1.0 ),  // green
color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
color4( 1.0, 1.0, 1.0, 1.0 ),  // white
color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

Quad Function

```
// quad generates two triangles for each face and assigns colors
//    to the vertices
int Index = 0;
void quad( int a, int b, int c, int d )
{
colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++;
colors[Index] = vertex_colors[b]; points[Index] = vertices[b]; Index++;
colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++;
colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++;
colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++;
colors[Index] = vertex_colors[d]; points[Index] = vertices[d]; Index++;
}
```

Color Cube

```
// generate 12 triangles: 36 vertices and 36 colors
void
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 );
}
```

Initialization I

```
void
init()
{
    colorcube();

    // Create a vertex array object

    GLuint vao;
    glGenVertexArrays ( 1, &vao );
    glBindVertexArray ( vao );
```

Initialization II

```
// Create and initialize a buffer object
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof(points) +
       sizeof(colors), NULL, GL_STATIC_DRAW );
    glBufferSubData( GL_ARRAY_BUFFER, 0,
        sizeof(points), points );
    glBufferSubData( GL_ARRAY_BUFFER, sizeof(points),
        sizeof(colors), colors );
// Load shaders and use the resulting shader program
    GLuint program = InitShader( "vshader36.glsl", "fshader36.glsl" );
    glUseProgram( program );
```

Initialization III

```
// set up vertex arrays
    GLuint vPosition = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( vPosition );
    glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
                           BUFFER_OFFSET(0) );

    GLuint vColor = glGetAttribLocation( program, "vColor" );
    glEnableVertexAttribArray( vColor );
    glVertexAttribPointer( vColor, 4, GL_FLOAT, GL_FALSE, 0,
                           BUFFER_OFFSET(sizeof(points)) );

    theta = glGetUniformLocation( program, "theta" );
```

Display Callback

```
void
display( void )
{
    glClear( GL_COLOR_BUFFER_BIT
        |GL_DEPTH_BUFFER_BIT );

    glUniform3fv( theta, 1, theta );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );

    glutSwapBuffers();
}
```

Mouse Callback

```
void
mouse( int button, int state, int x, int y )
{
    if ( state == GLUT_DOWN ) {
        switch( button ) {
            case GLUT_LEFT_BUTTON:    Axis = Xaxis;  break;
            case GLUT_MIDDLE_BUTTON:  Axis = Yaxis;  break;
            case GLUT_RIGHT_BUTTON:   Axis = Zaxis;  break;
        }
    }
}
```

Idle Callback

```
void
idle( void )
{
    theta[axis] += 0.01;

    if ( theta[axis] > 360.0 ) {
        theta[axis] -= 360.0;
    }

    glutPostRedisplay();
}
```