Introduction to Computer Graphics

Geometry Objects

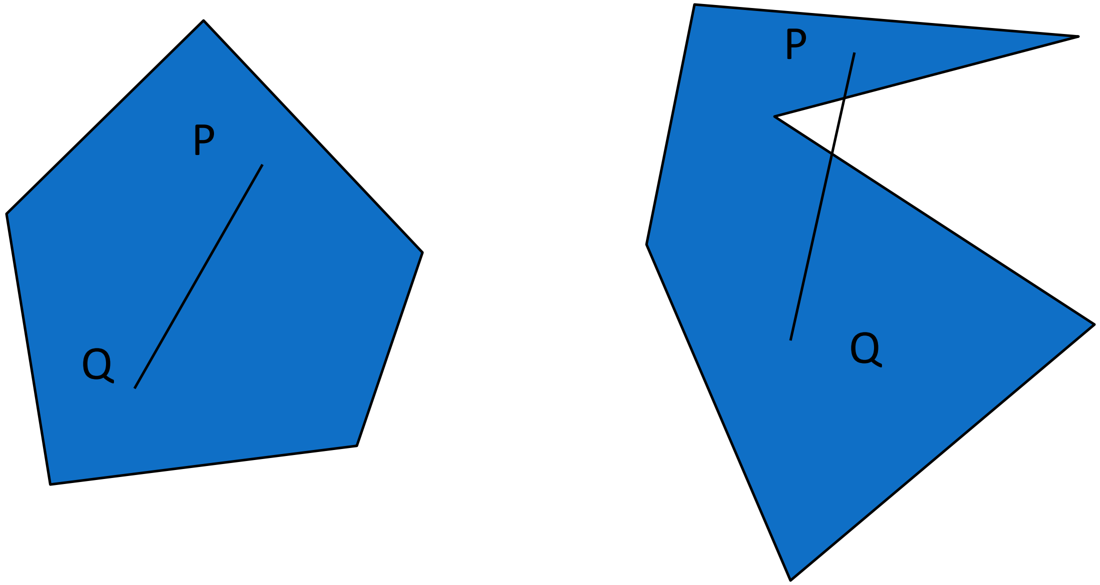
葉奕成 I-Cheng (Garrett) Yeh

Basic Elements

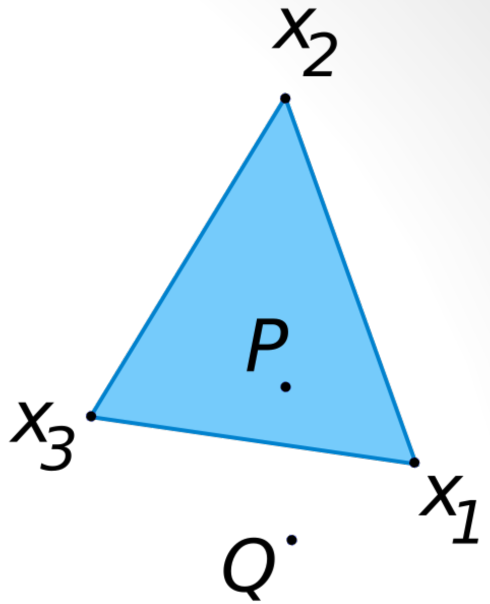
- Geometry is the study of the relationships among objects in an *n*-dimensional space
  - In computer graphics, we are interested in objects that exist in three dimensions
- Want a minimum set of primitives from which we can build more sophisticated objects
- We will need three basic elements
  - Scalars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

Convexity

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



Affine Sums



- A convex combination is a linear combination of points
- Consider the “sum”

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

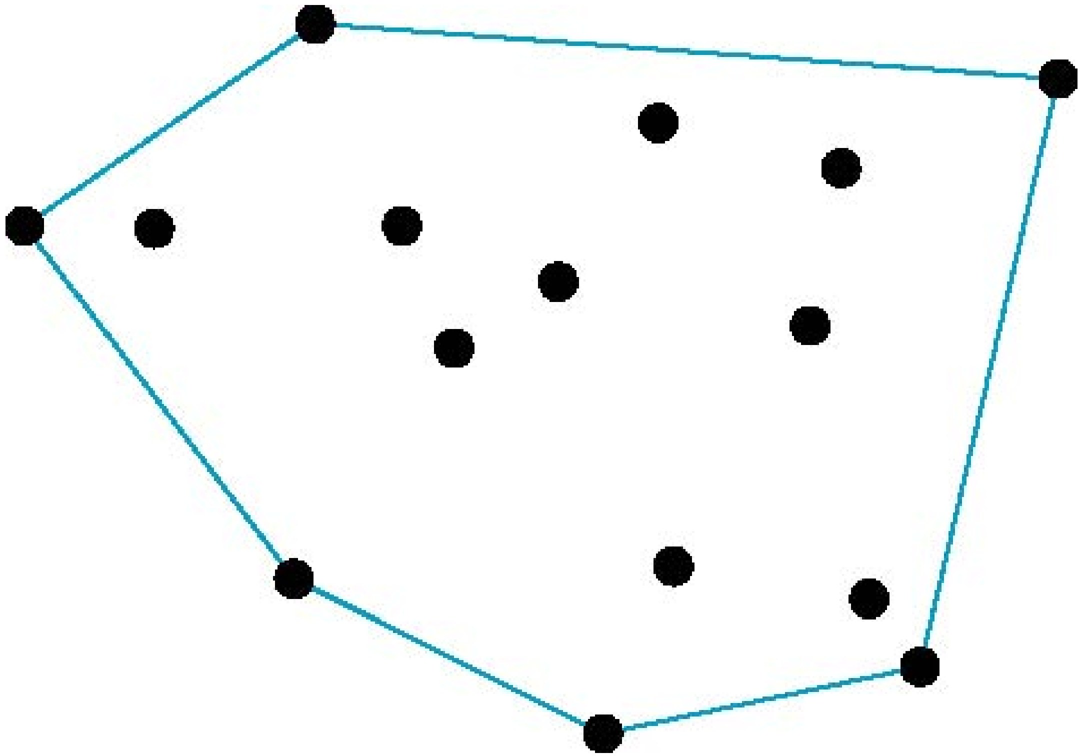
Can show by induction that this sum makes sense if and only if $\alpha_1+\alpha_2+\ldots\ldots\alpha_n=1$

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

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

Convex Hull

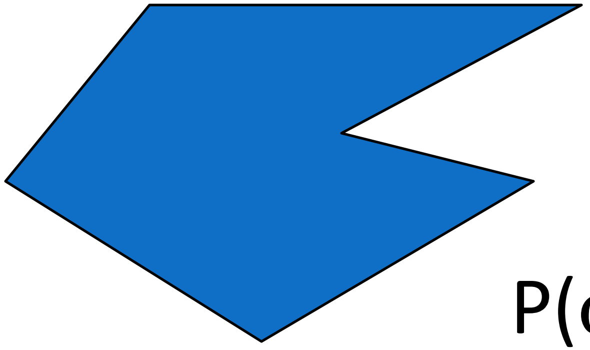
- Smallest convex object containing $P_1, P_2, \ldots\ldots P_n$
- Formed by “shrink wrapping” points



Curves and Surfaces

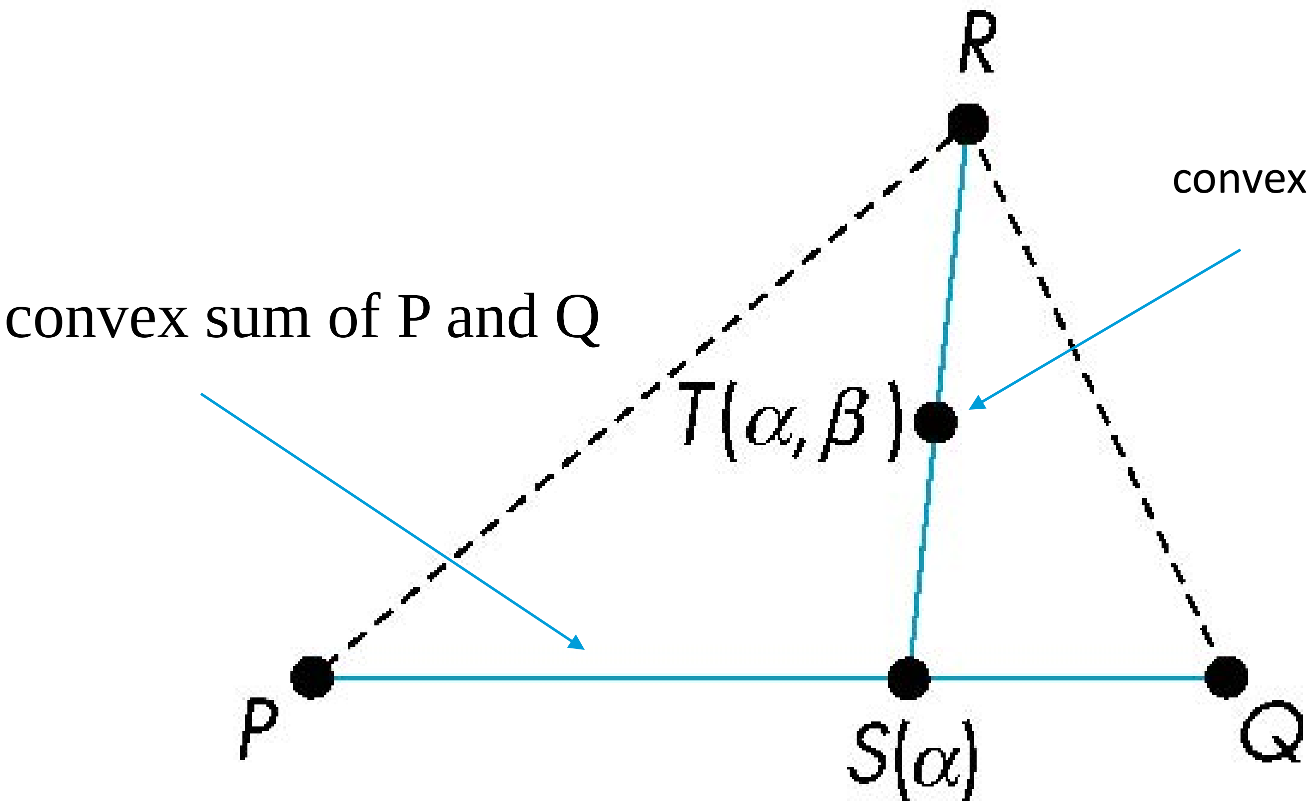
- Curves are one parameter entities of the form $P(\alpha)$ where the function is nonlinear
- Surfaces are formed from two-parameter functions $P(\alpha, \beta)$
  - Linear functions give planes and polygons

$P(\alpha)$



$P(\alpha, \beta)$

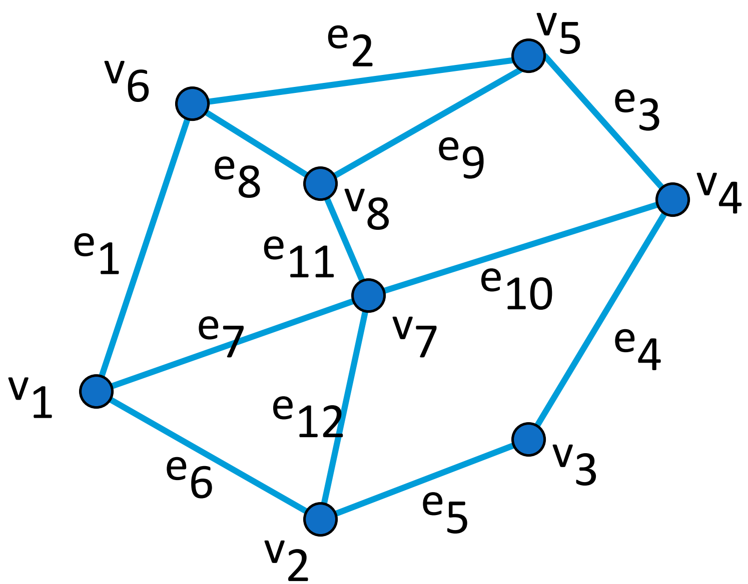
Triangles



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

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

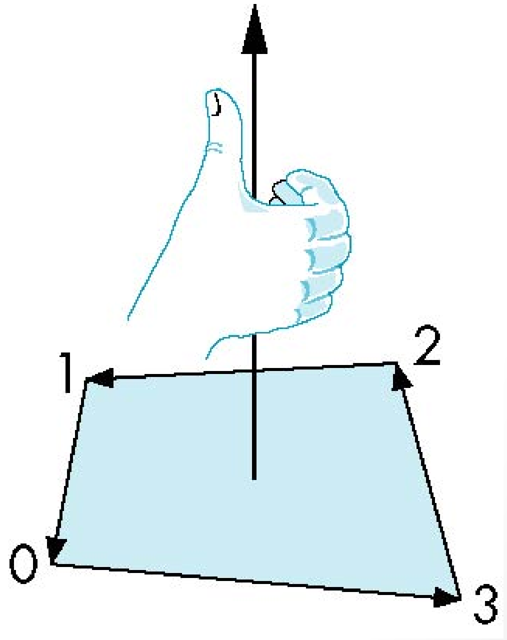
- List all polygons by their geometric locations
- Leads to OpenGL code such as

```
glBegin(GL_POLYGON);
    glVertex3f(x1, y1, z1);
    glVertex3f(x6, y6, z6);
    glVertex3f(x7, y7, z7);
glEnd();
```

- Inefficient and unstructured
  - Consider moving a vertex to a new locations, the *connectivity* of this mesh may been broken.

Inward and Outward Facing Polygons

- The order $\{v_1, v_2, v_7\}$ and $\{v_2, v_7, v_1\}$ are equivalent in that the same polygon will be rendered by OpenGL but the order $\{v_1, v_7, v_2\}$ is different
- The first two describe *outwardly facing* polygons
- Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal
- OpenGL treats inward and outward facing polygons differently

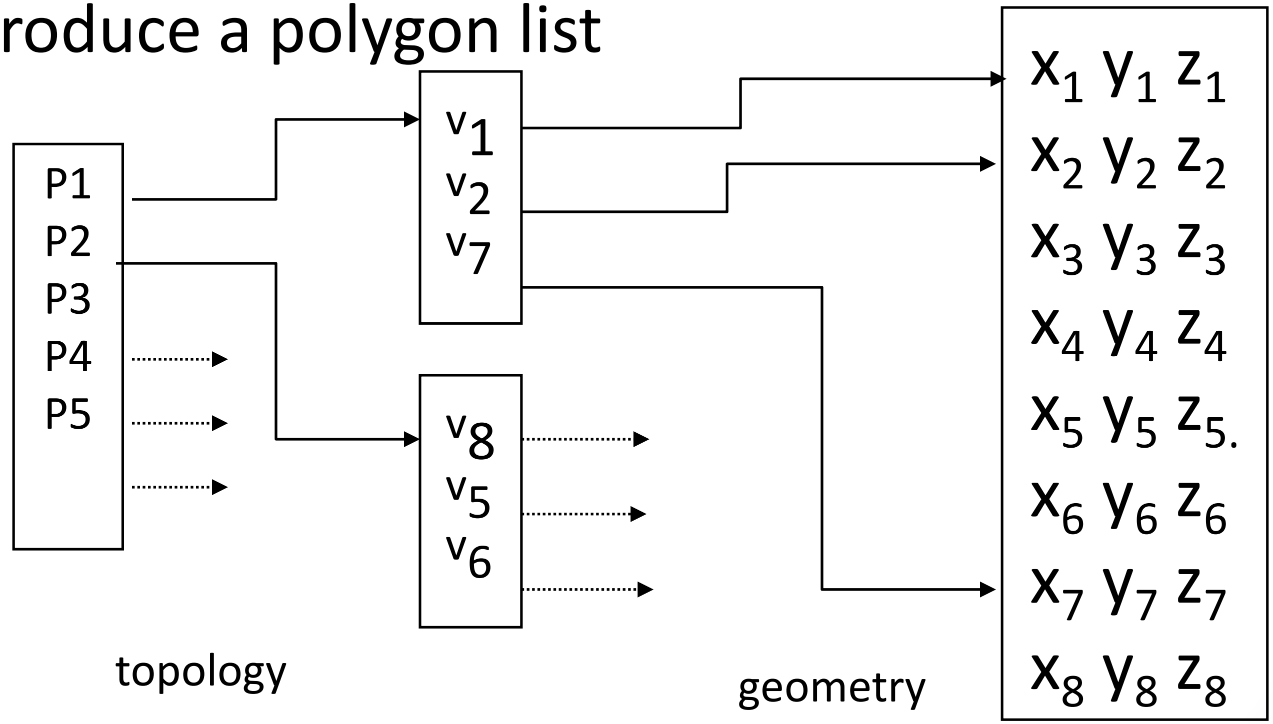


Geometry vs Topology

- 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

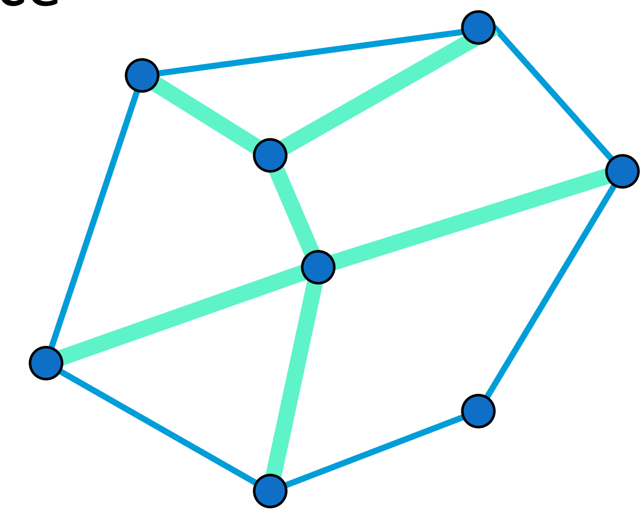
Vertex Lists

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



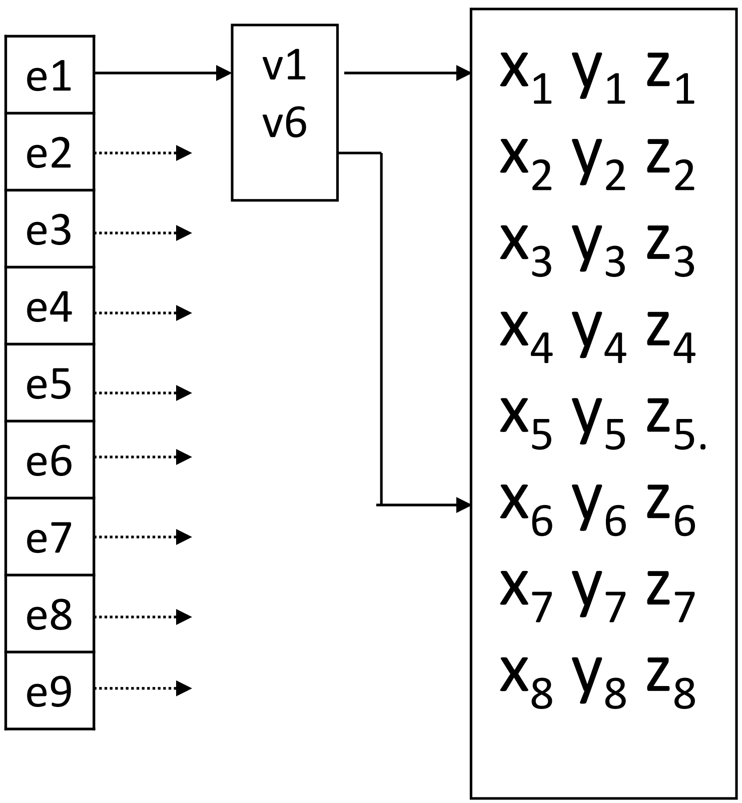
Shared Edges

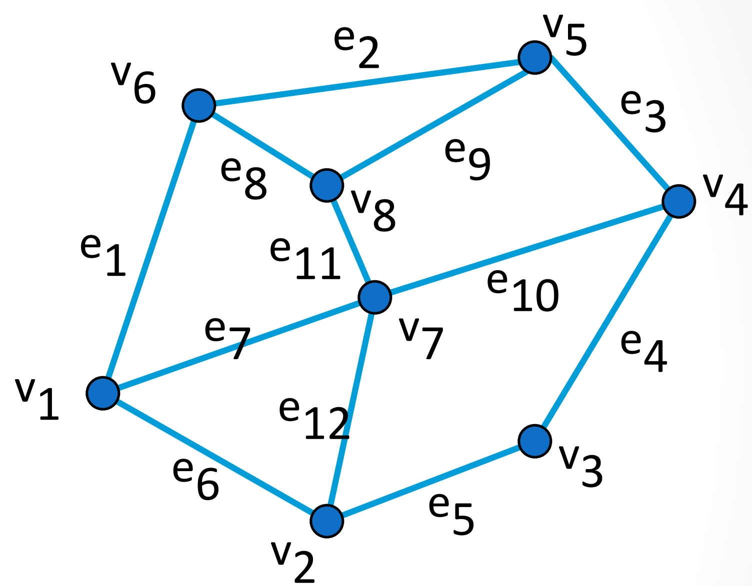
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- 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}};
```

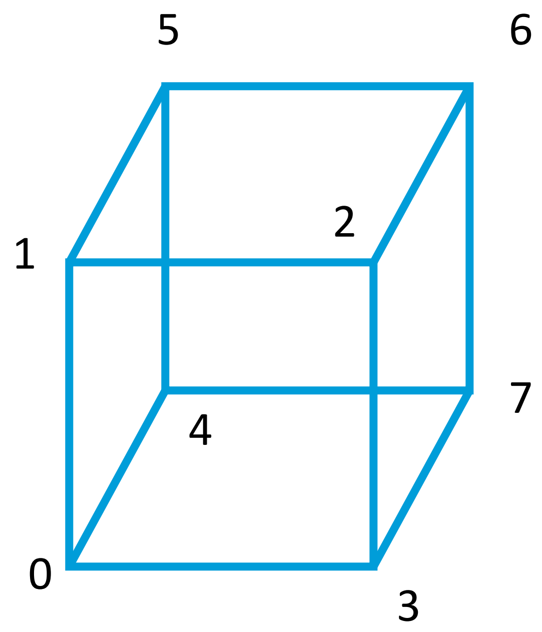
Drawing a Polygon from a List of Indices

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

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 x **`glBegin`**, 6 x **`glEnd`**
  - 6 x **`glColor`**
  - 24 x **`glVertex`**
  - More if we use texture (*texcoord*) and lighting (*normal*)

Vertex Arrays

- OpenGL provides a facility called *vertex arrays* that allow 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 positions

stored as floats

offset b/w each vertex

data array pointer

```
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* !!

SUGGESTION!

OR

OBJECTION?

Let's stop here,

TAKE A BREAK