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(x_1, x_1, x_1);
  glVertex3f(x_6, x_6, x_6);
  glVertex3f(x_7, x_7, x_7);
  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 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.

P1 → P2 → P3 → P4 → P5

v1, x1 y1 z1
v2, x2 y2 z2
v3, x3 y3 z3
v4, x4 y4 z4
v5, x5 y5 z5
v6, x6 y6 z6
v7, x7 y7 z7
v8, x8 y8 z8

topology
geometry

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

- Can store mesh by 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

```c
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}};
```

```c
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

```c
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

```c
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 straightforward 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);
  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:

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

Drawing the cube

- Method 1:
  
  ```
  for(i=0; i<6; i++) glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_BYTE, &cubeIndices[4*i]);
  ```
  
- Method 2:
  
  ```
  glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndices);
  ```
  
  Draws cube with 1 function call!!