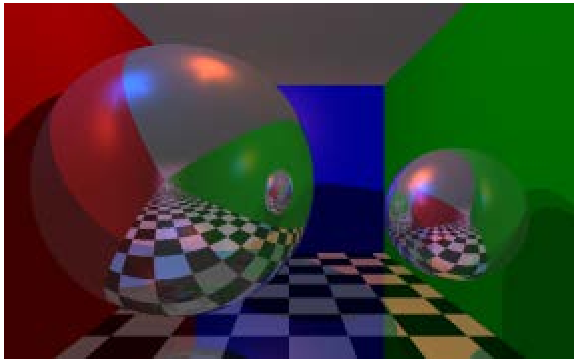
CSI 4105 Computer Graphics

Spring 2017

Lecture 6: Model Representation

Seon Joo Kim
Yonsei University



Announcement

- Project #2 (Due April 14^{th}, 11:55PM)

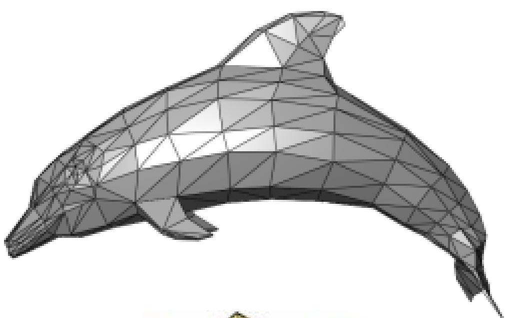
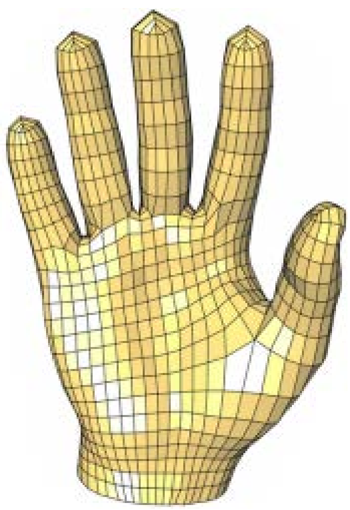
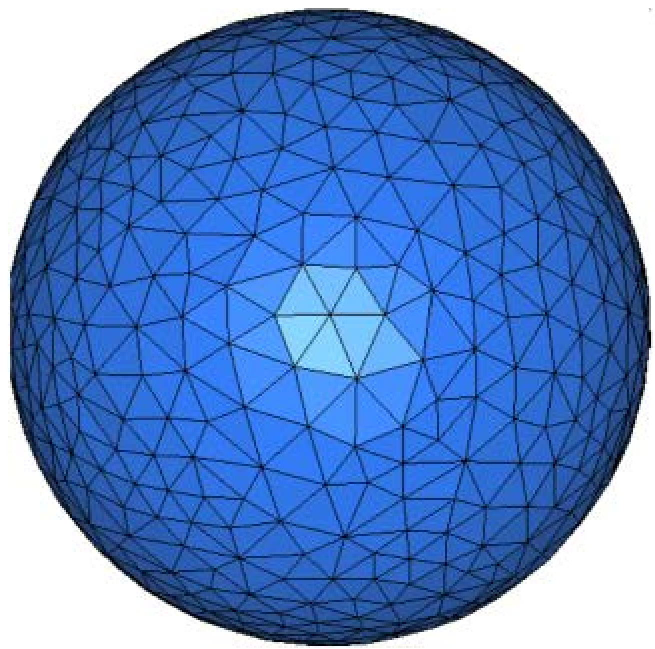
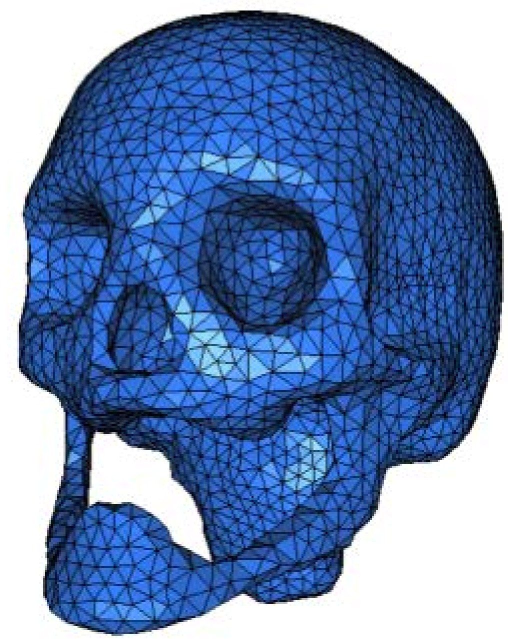
Objectives

- To overview common model representation
  - Vertex Lists
  - Efficient rendering

Model representation
(and efficient rendering)

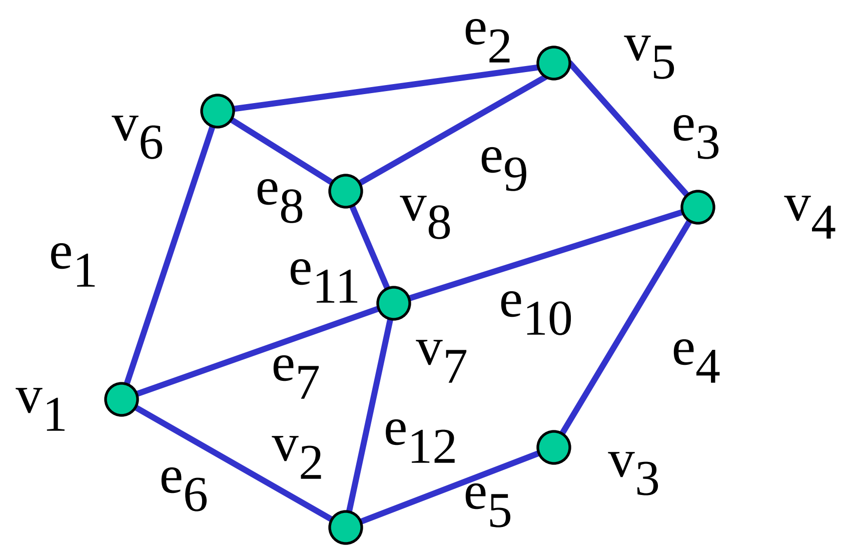
Mesh

- We use meshes of polygons to represent our models
  - As discussed before, the triangle is the optimal polygon (and preferred), but it doesn't have to be a triangle

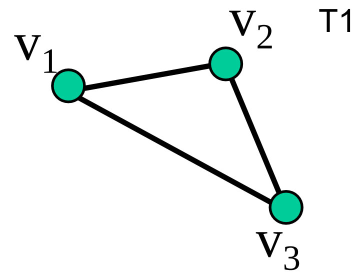
Representing a Mesh

- Look at this simple polygonal mesh



- There are 8 nodes and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges
- Each vertex has a location $v_i = (x_i\ y_i\ z_i)$
- We can think of it as a graph, vertices are connected by edges

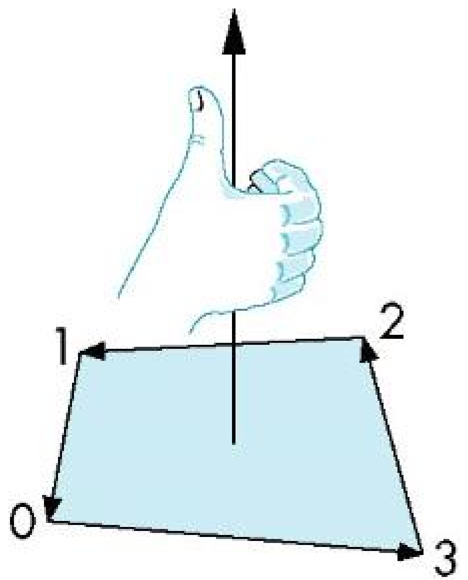
Inward and Outward Facing Polygons (The winding)



- For triangle T1, the order $\{v_1, v_3, v_2\}$ and $\{v_3, v_2, v_1\}$ are considered equivalent, but the order $\{v_1, v_2, v_3\}$ is considered different

- $\{v_1, v_3, v_2\}$ and $\{v_3, v_2, v_1\}$ describe an *outwardly facing* polygon

  -They use the *right-hand rule* that means counter-clockwise encirclement (or winding) results in an outward-pointing normal
  [your right hand has to wind counter-clockwise about the thumb]



- The order $\{v_1, v_2, v_3\}$ (clockwise winding) will result in a polygon with an inward facing normal
  - So, geometry is the same, but normal for that polygon will be different

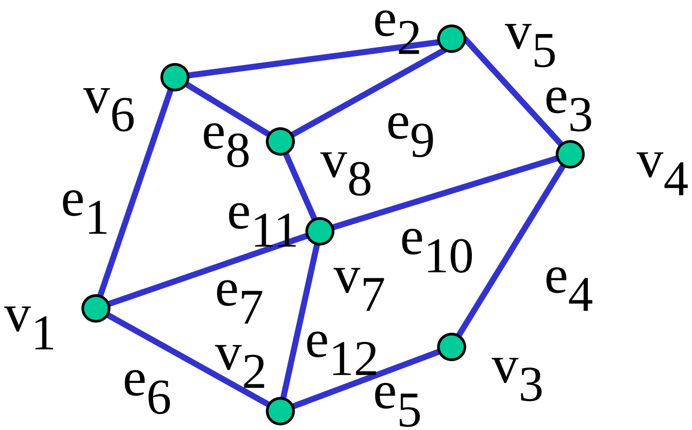
Switching the "Winding" in OpenGL

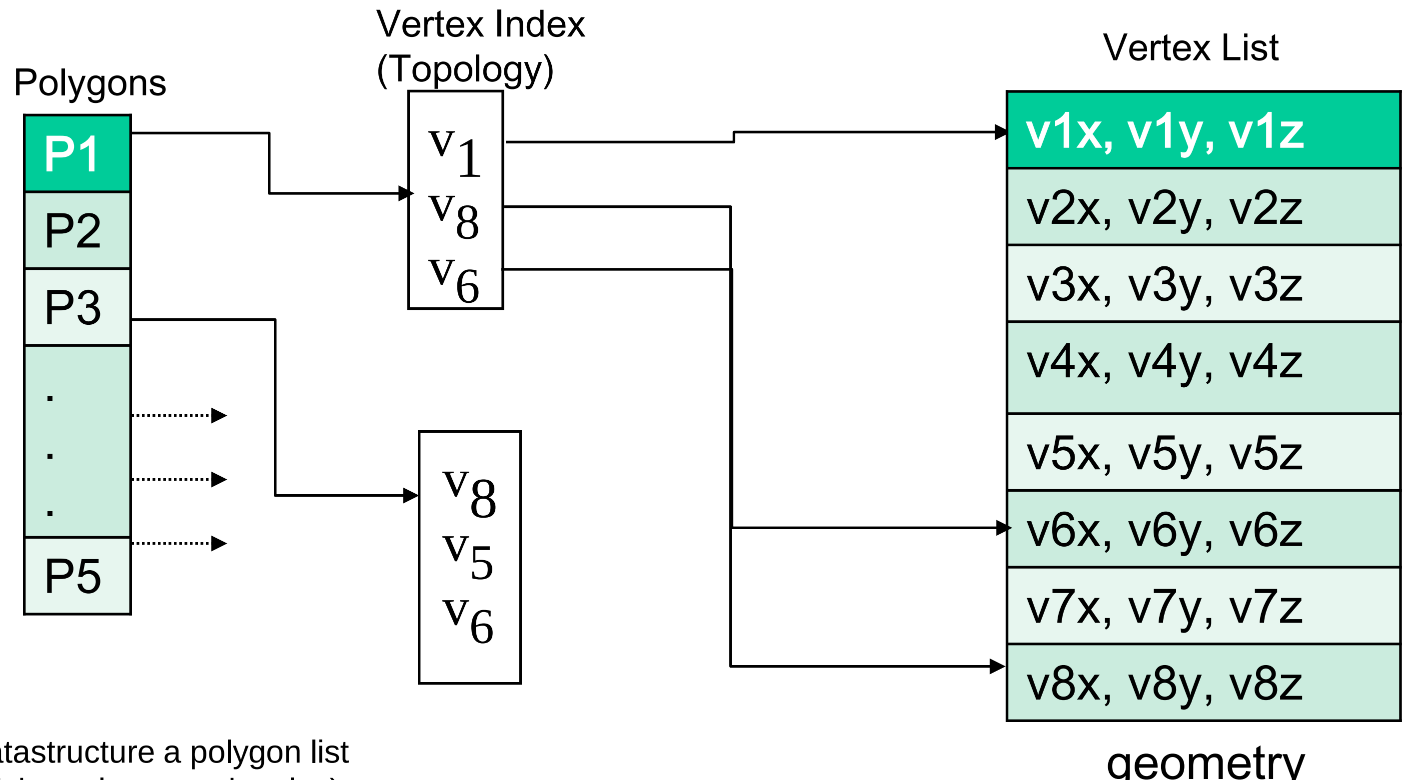
- OpenGL default is that outward facing polygons are specified in counter-clockwise fashion
- But you change this by:
  - `glFrontFace(GL_CW);      // Clockwise Winding = outward face`
  - `glFrontFace(GL_CCW);     // Counter-Clockwise Winding =`
    `// outward face`

- Classic example involving the Glut Teapot
  - glutTeapot is specified in clockwise winding (it is a bug!)
  - So, do this:
  - ```
    glFrontFace(GL_CW);         // switch to clockwise
    glutSolidTeapot(1.0);       // draw teapot
    glFrontFace(GL_CCW);        // switch back
    ```
    ```

Organize the data – vertex list + polygon list

- Put the vertex geometry in an array
- Introduce a polygon list
  - This list indexes to the corresponding vertices

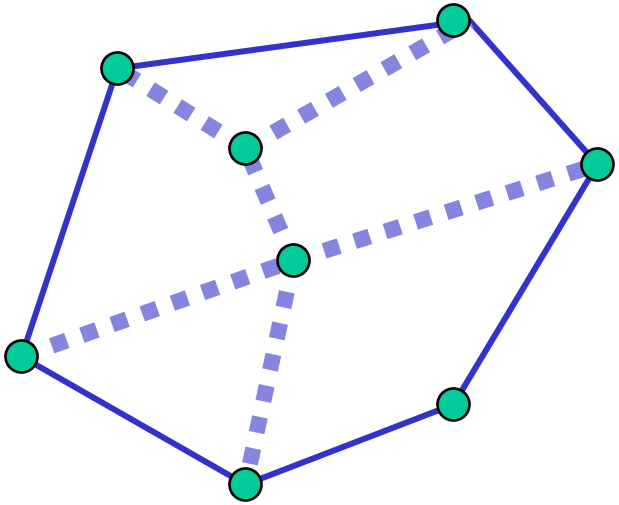




* We call this datastructure a polygon list or triangle list (if the polygons=triangles)
Sometimes the polygons are called “faces”

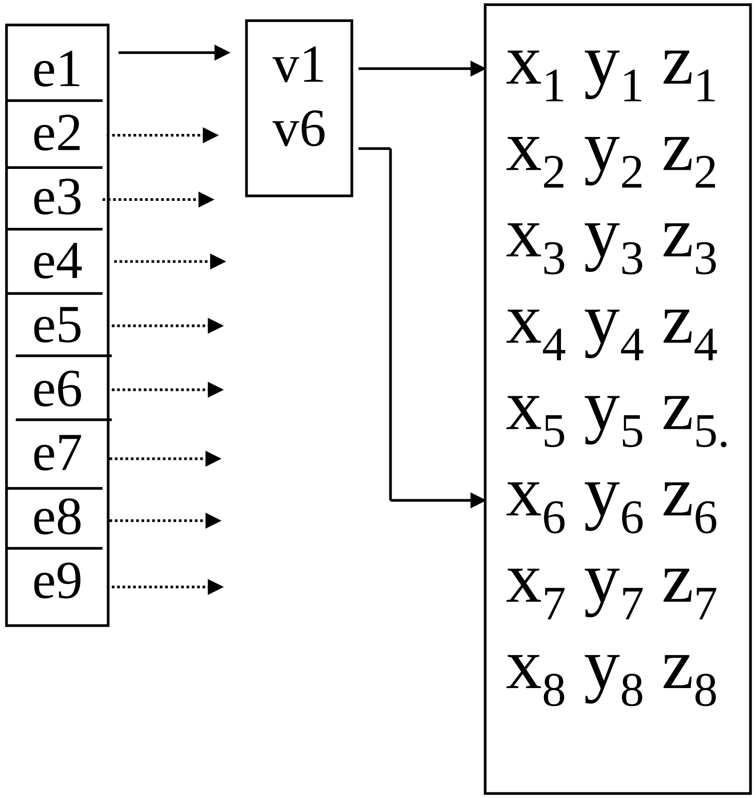
Shared Edges

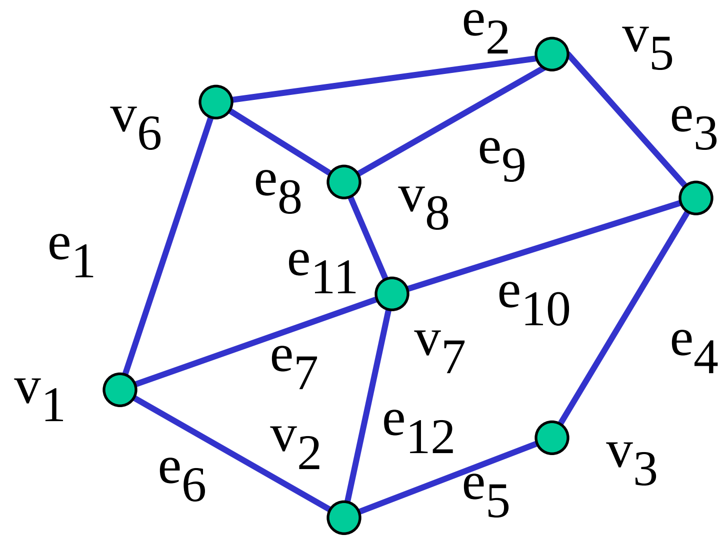
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



Edges (■ ■ ■) are drawn more than once if we store the mesh as a polygon list.

“Edge List” data structure





Note polygons are not represented

Edge list stores edges + vertices. Polygons are not represented!

For most OpenGL purposes, the polygon/triangle list is the most effective, but if you know you will draw lots of wireframe models, or require other types of processing [like cloth modeling], the edge representation can be very useful. Converting an edge-list to a polygon list is not trivial, so often start with a polygon list and derive the edge-list from it. Deriving the edge-list datastructure from a polygon-list data-structure is straight-forward.

Modeling a Cube (cube1.c)

Model a color cube for rotating cube program

Define global arrays for vertices and colors (one color per vertex)

```
GLfloat vertices[8][3] = {
    {-1,-1, 1}, {-1, 1, 1},
    { 1, 1, 1}, { 1,-1, 1},
    {-1,-1,-1}, {-1, 1,-1},
    { 1, 1,-1}, { 1,-1,-1}
};

GLfloat colors[8][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

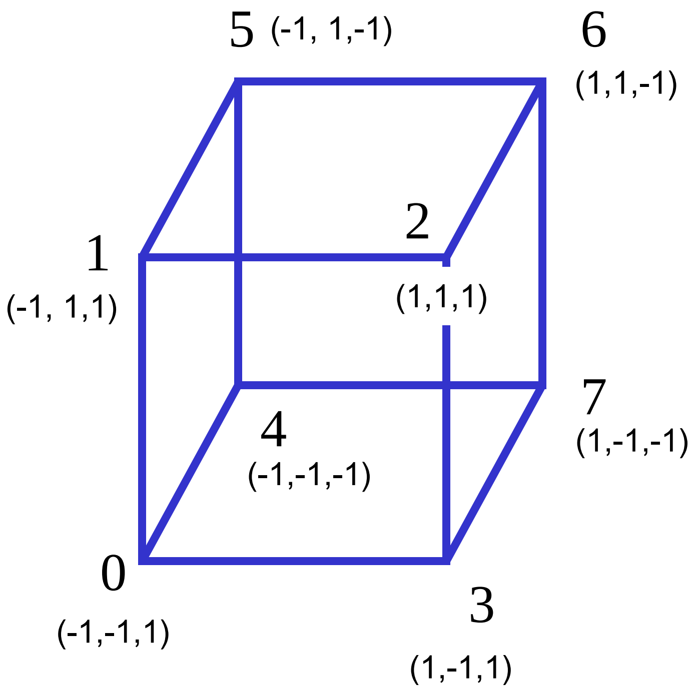
Draw a quadrilateral from a list of indices into the array **`vertices`**

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

}
```

Draw cube from faces

```
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);
}
```



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`**
  - 24 **`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 (cube2.c)

- 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!!

Also see: glDrawArray(..)

Comment on efficiency

- The drawElements (and related drawArray) is introduced in this class so you are aware of it

- I don’t expect you to use this in our assignments. The performance gain for the type of geometry we are rendering is minimum

- However, in a real world application efficient rendering and drawing would be very important!