Computer Graphics
CS 543 – Lecture 4 (Part 1)
Building 3D Models (Part 1)

Prof Emmanuel Agu

Computer Science Dept.

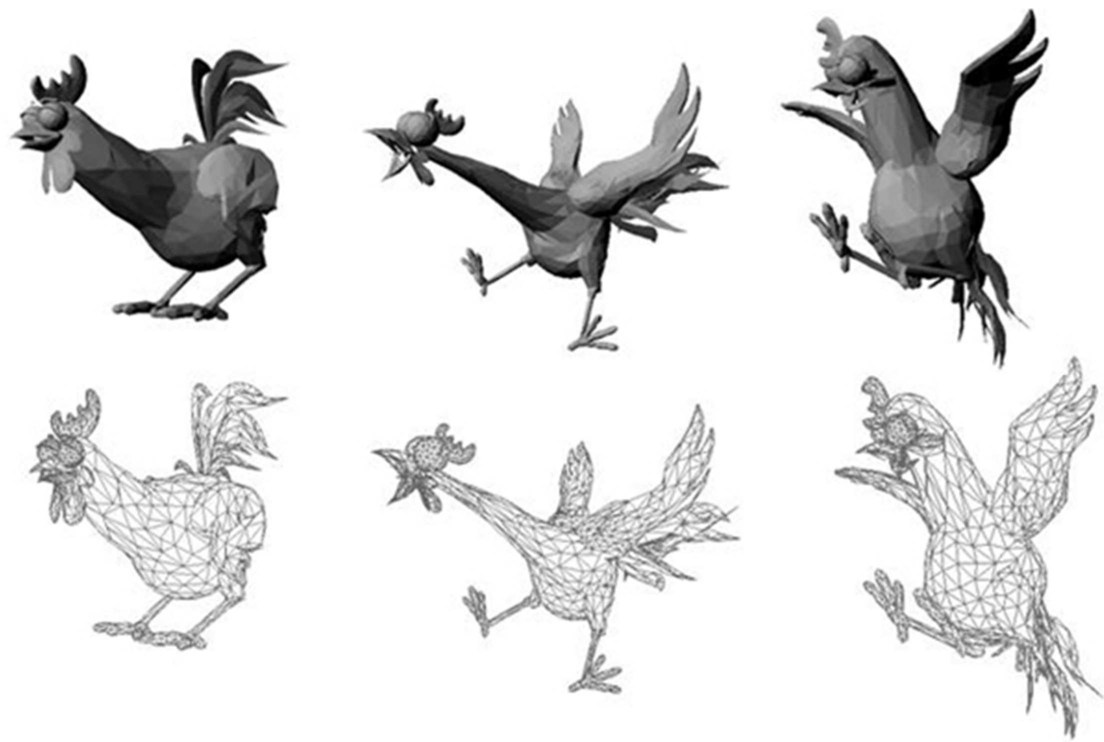
Worcester Polytechnic Institute (WPI)

Objectives

- Introduce simple data structures for building polygonal models
  - Vertex lists
  - Edge lists
- Deprecated OpenGL vertex arrays
- Drawing 3D objects

3D Applications

- 2D: points have (x,y) coordinates
- 3D: points have (x,y,z) coordinates
- In OpenGL, 2D graphics are special case of 3D graphics

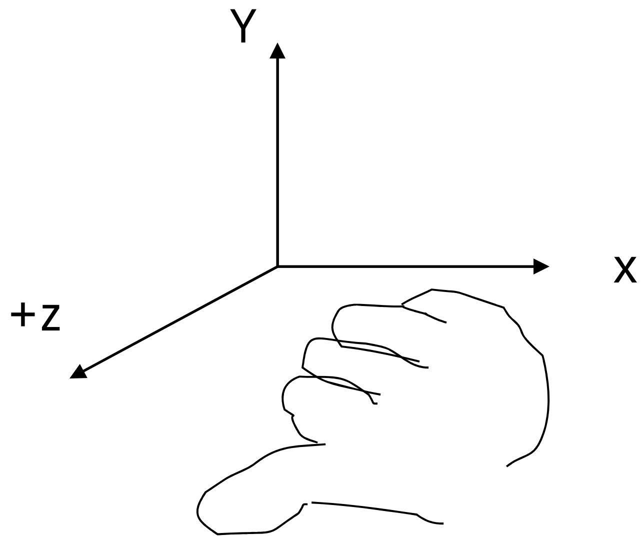


Setting up 3D Applications

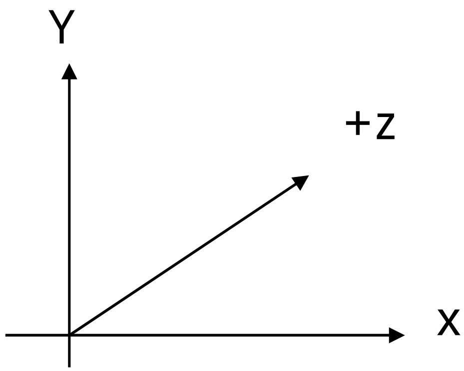
- Programming 3D, not many changes from 2D
  1. Load representation of 3D object into data structure
     - **Note:** Vertices stored as 3D points (x, y, z)
     - Use **`vec3, glUniform3f`** instead of **`vec2`**
  2. Draw 3D object
  3. **Hidden surface removal:** Correctly determine order in which primitives (triangles, faces) are rendered (Blocked faces **NOT** drawn)

3D Coordinate Systems

- Tip: sweep fingers x-y: thumb is z



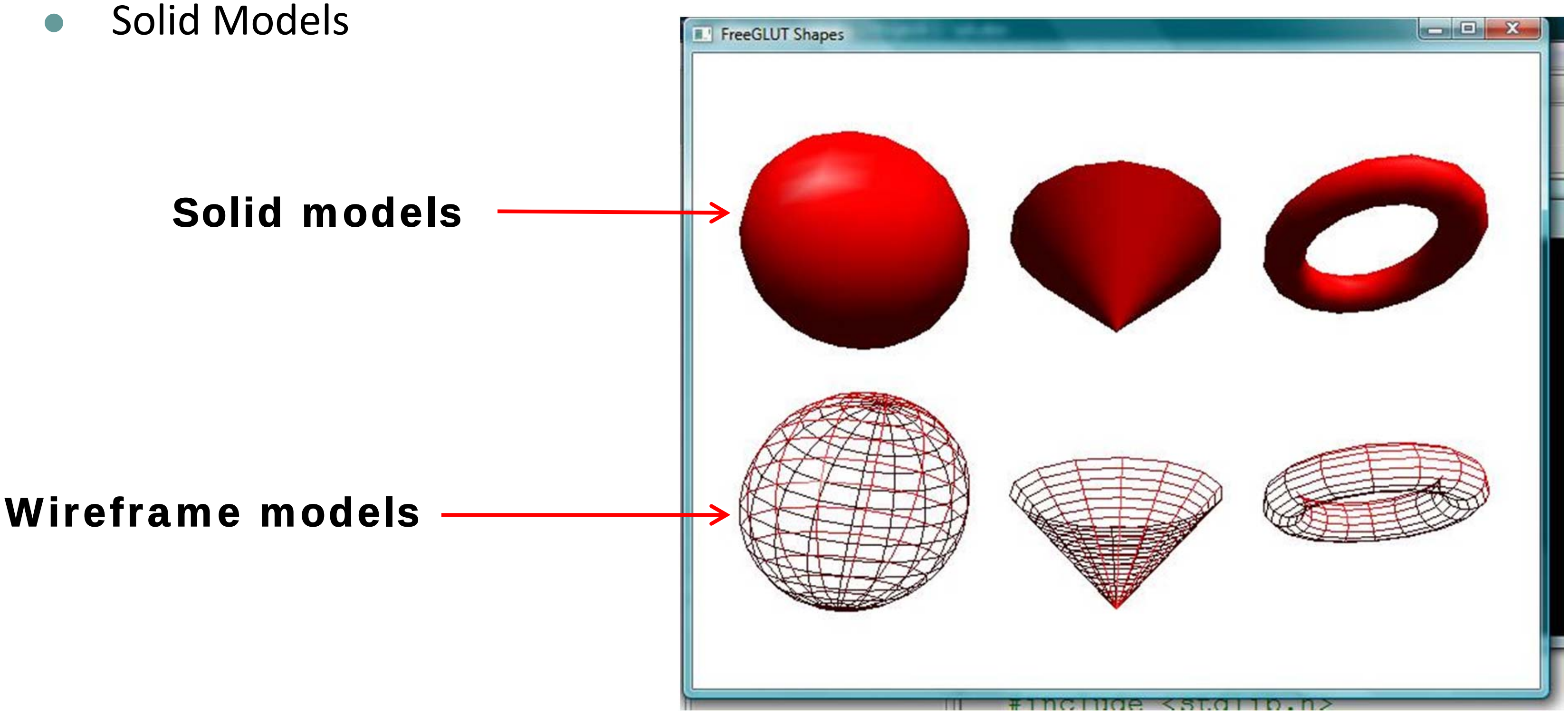
Right hand coordinate system



Left hand coordinate system
•Not used in this class and
•Not in OpenGL

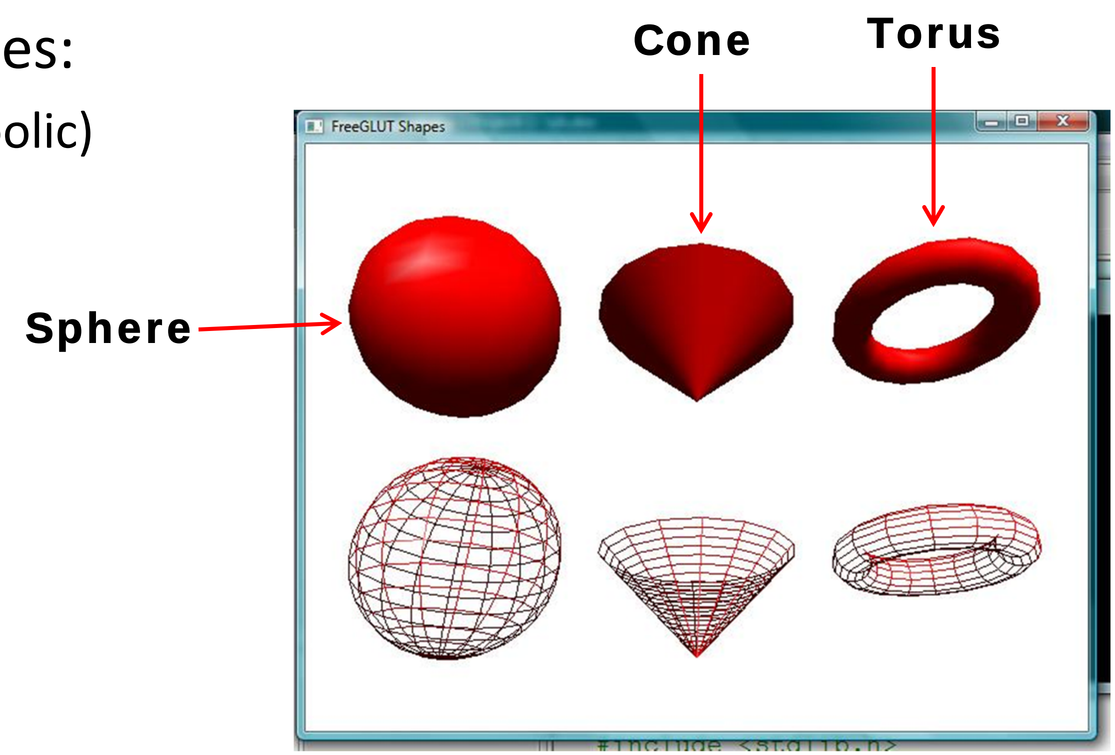
Generating 3D Models: GLUT Models

- One way of generating 3D shapes is by using GLUT 3D models (Restrictive?)
- **Note:** Simply make GLUT 3D calls in **application program** (Not shaders)
- Two main categories of GLUT models:
  - Wireframe Models
  - Solid Models



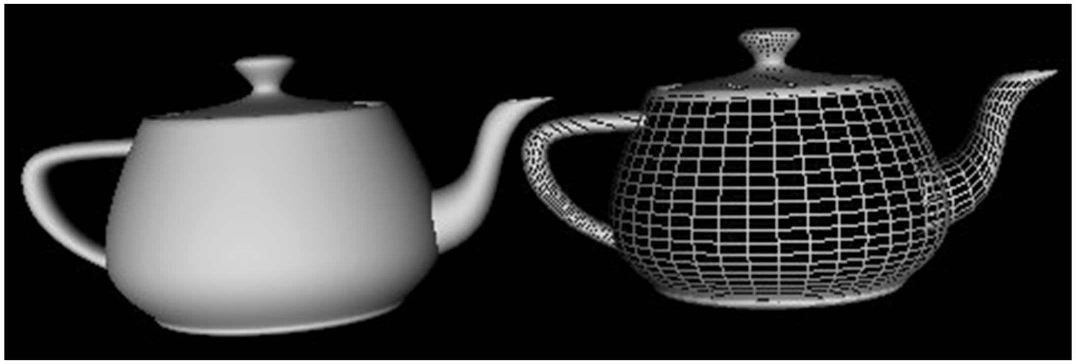
3D Modeling: GLUT Models

- Basic Shapes
  - **Cone:** glutWireCone( ), glutSolidCone( )
  - **Sphere:** glutWireSphere( ), glutSolidSphere( )
  - **Cube:** glutWireCube( ), glutSolidCube( )
- More advanced shapes:
  - Newell Teapot: (symbolic)
  - Dodecahedron, Torus



GLUT Models: glutwireTeapot( )

- Famous Utah Teapot has become an unofficial computer graphics mascot



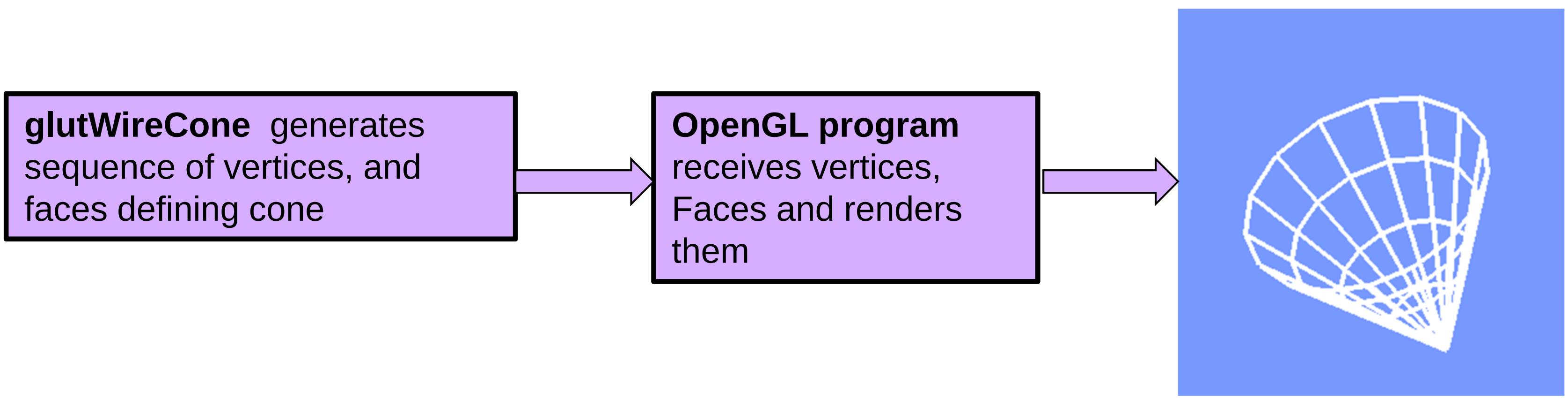
glutWireTeapot(0.5) - Create teapot of size 0.5, center positioned at (0,0,0)

Also glutSolidTeapot( )

You need to apply transformations to position, scale and rotate it

3D Modeling: GLUT Models

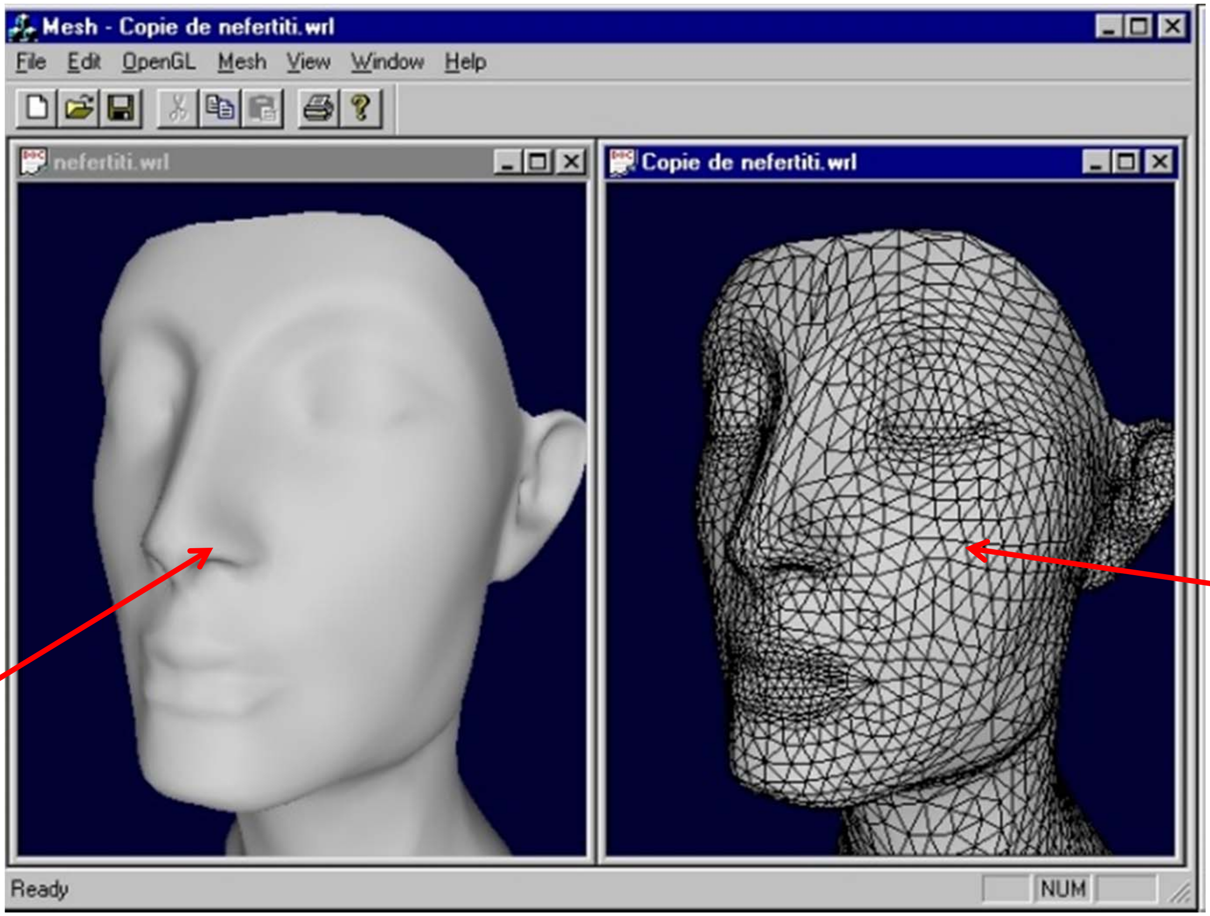
- Glut functions under the hood
  - generate sequence of points that define a shape
  - centered at 0.0
- **Example:** glutWireCone generates sequence of vertices, and faces defining cone and connectivity
- Generated vertices and faces passed to OpenGL for rendering



Polygonal Meshes

- Modeling with GLUT shapes (cube, sphere, etc) too restrictive
- Difficult to approach realism
- Other (preferred) way is using polygonal meshes:
  - Collection of polygons, or faces, that form “skin” of object
  - More flexible
  - Represents complex surfaces better
  - Examples:
    - Human face
    - Animal structures
    - Furniture, etc

Polygonal Mesh Example

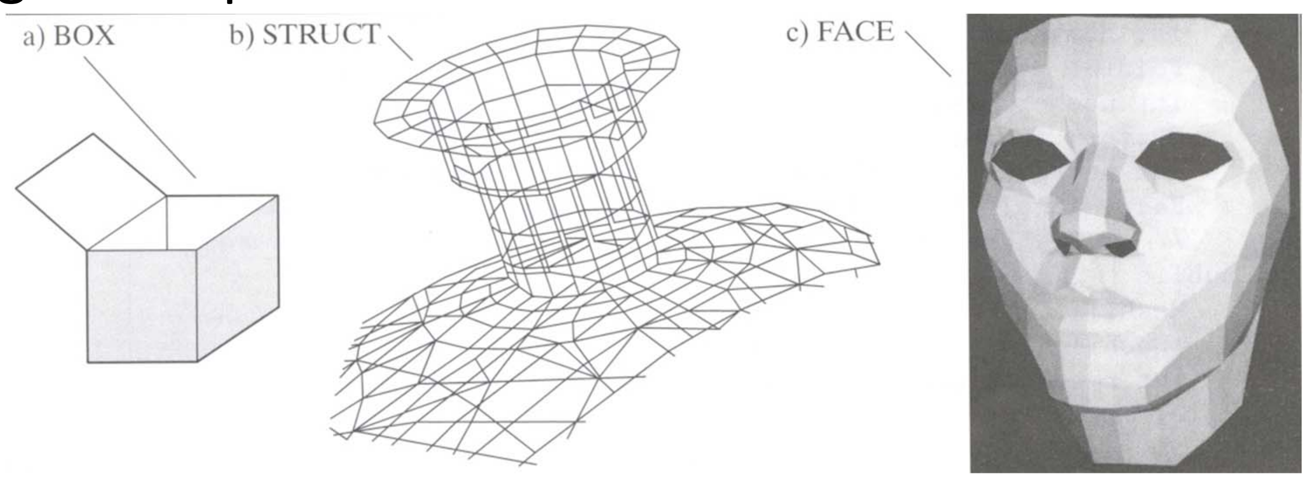


Smoothed Out with Shading (later)

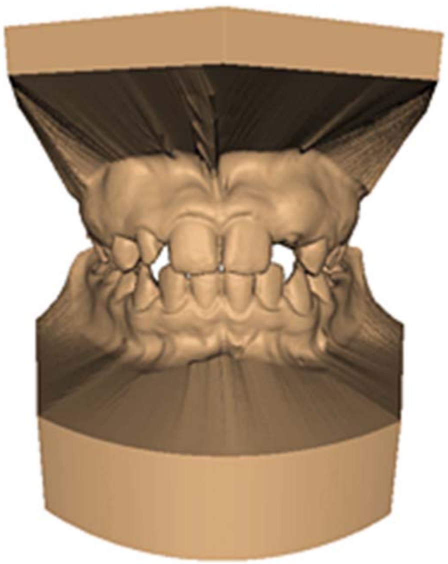
Mesh (wireframe)

Polygonal Meshes

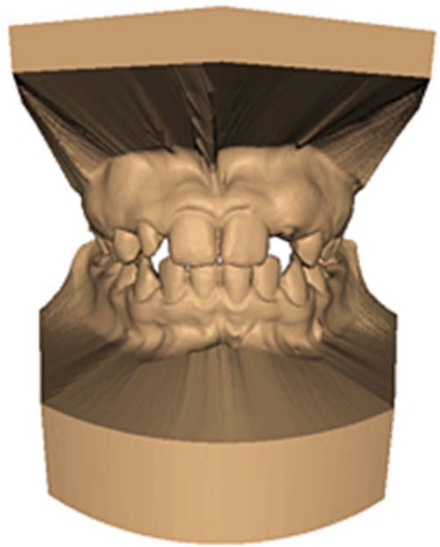
- Meshes now standard in graphics
- OpenGL
  - Good at drawing polygons, triangles
  - Mesh = sequence of polygons forming thin skin around object
- Simple meshes exact. (e.g barn)
- Complex meshes approximate (e.g. human face)
- Use shading technique later to smoothen



Meshes at Different Resolutions



Original: 424,000 triangles



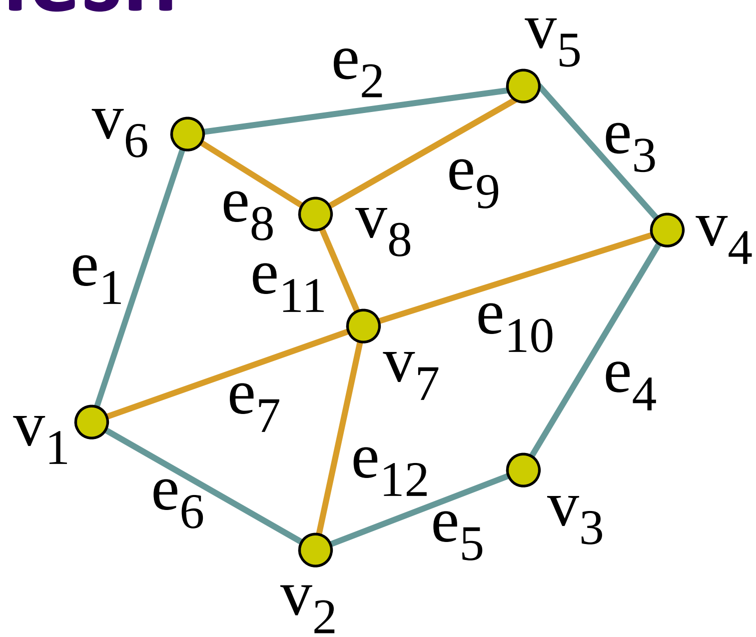
60,000 triangles (14%).

1000 triangles (0.2%)

(courtesy of Michael Garland and Data courtesy of Iris Development.)

Representing a Mesh

- Consider a mesh



- There are 8 vertices and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges (shown in orange)
- Each vertex has a location $v_i = (x_i\ y_i\ z_i)$

Simple Representation

- Define each polygon by (x,y,z) locations of its vertices
- Leads to OpenGL code such as

```
vertex[i]   = vec3(x1, y1, z1);
vertex[i+1] = vec3(x6, y6, z6);
vertex[i+2] = vec3(x7, y7, z7);
i+=3;
```

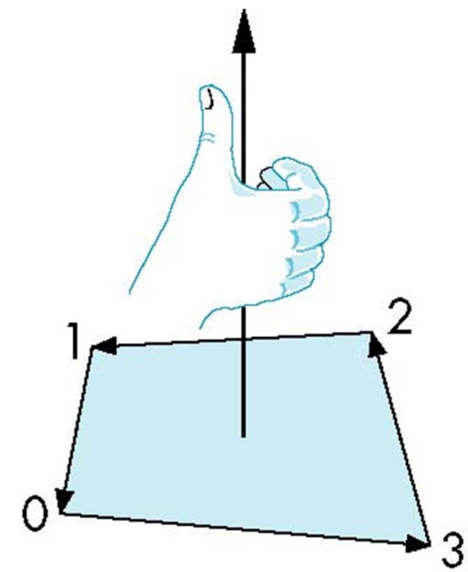
- Inefficient and unstructured
  - Vertices shared by many polygons are declared multiple times
  - Consider deleting vertex, moving vertex to new location
  - Must search for all occurrences

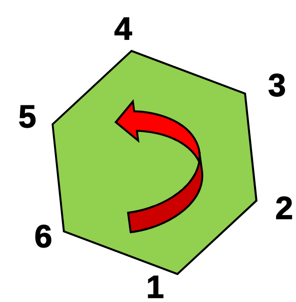
Geometry vs Topology

- Better data structures should separate geometry from topology
  - **Geometry:** (x,y,z) locations of the vertices
  - **Topology:** How vertices and edges are connected
  - **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 moves)

Polygon Traversal Convention

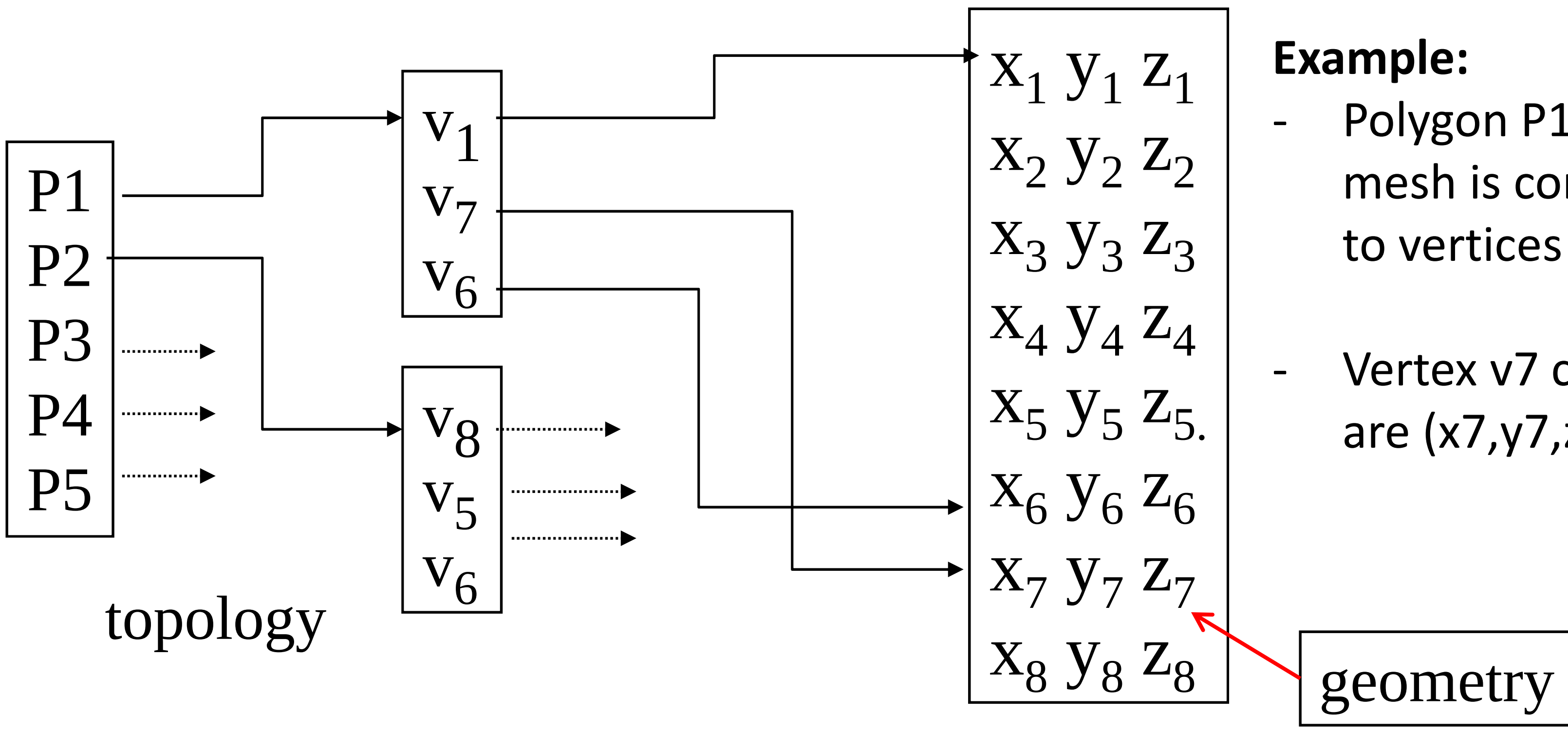
- Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal
- OpenGL can treat inward and outward facing polygons differently
- The order $\{v_1, v_6, v_7\}$ and $\{v_6, v_7, v_1\}$ are equivalent in that the same polygon will be rendered by OpenGL but the order $\{v_1, v_7, v_6\}$ is different
- The first two describe *outwardly facing* polygons





Vertex Lists

- **Vertex list:** (x,y,z) of vertices (its geometry) are put in array
- Use pointers from vertices into vertex list
- **Polygon list:** vertices connected to each polygon (face)

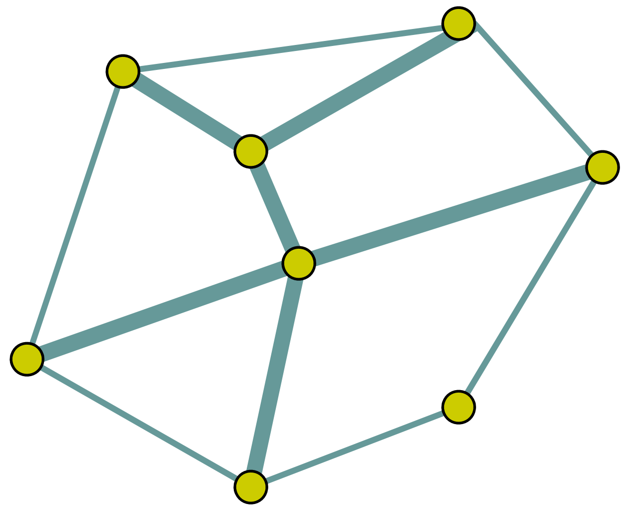


Example:

- Polygon P1 of mesh is connected to vertices (v1,v7,v6)
- Vertex v7 coordinates are (x7,y7,z7)

Shared Edges

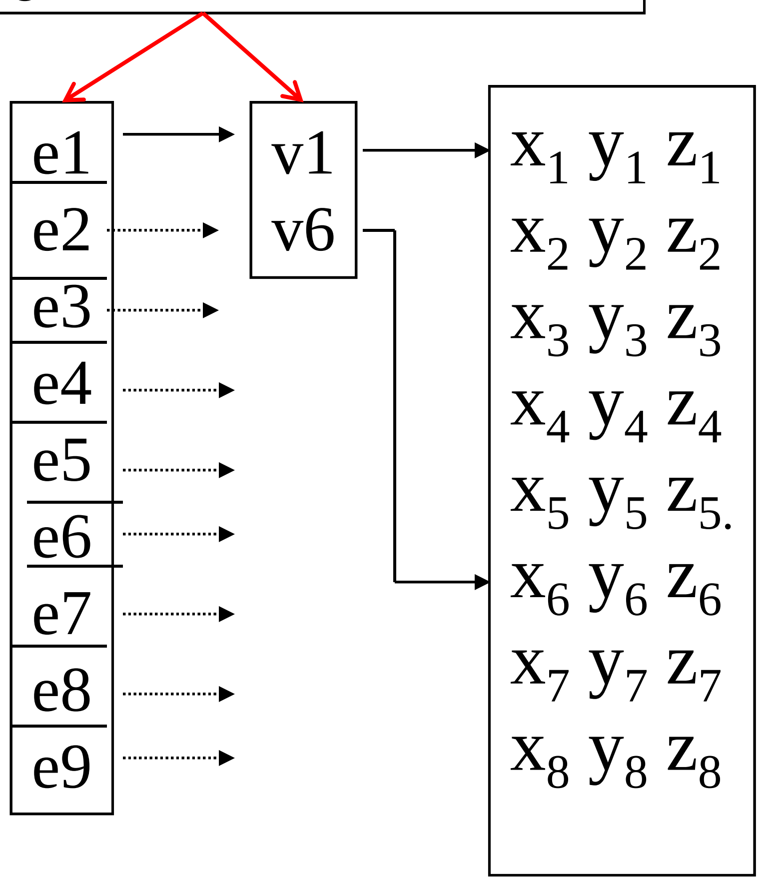
- Vertex lists draw filled polygons correctly
- If each polygon is drawn by its edges, shared edges are drawn twice

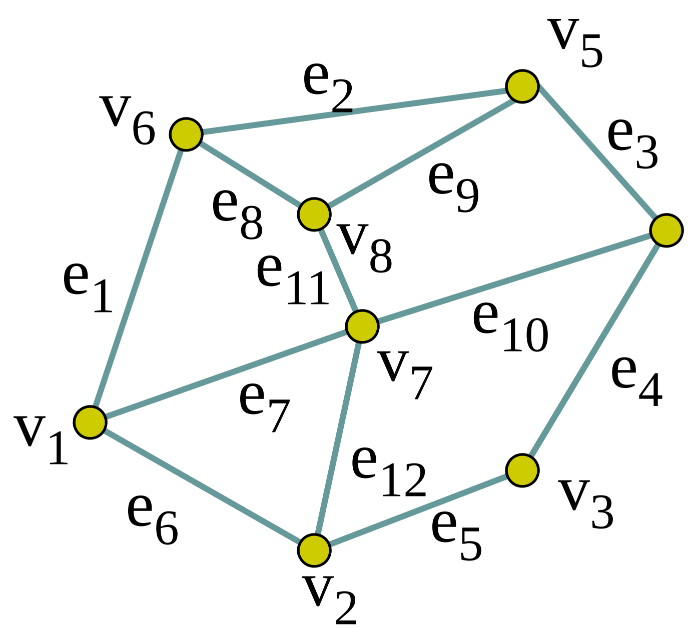


- **Alternatively:** Can store mesh by *edge list*

Edge List

Simply draw each edges once
E.g e1 connects v1 and v6

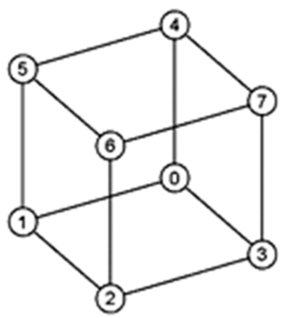




Note polygons are not represented

Modeling a Cube

- In 3D, declare vertices as (x,y,z) using **point3 v[3]**
- 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}); 
```

References

- Angel and Shreiner
- Hill and Kelley, appendix 4