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

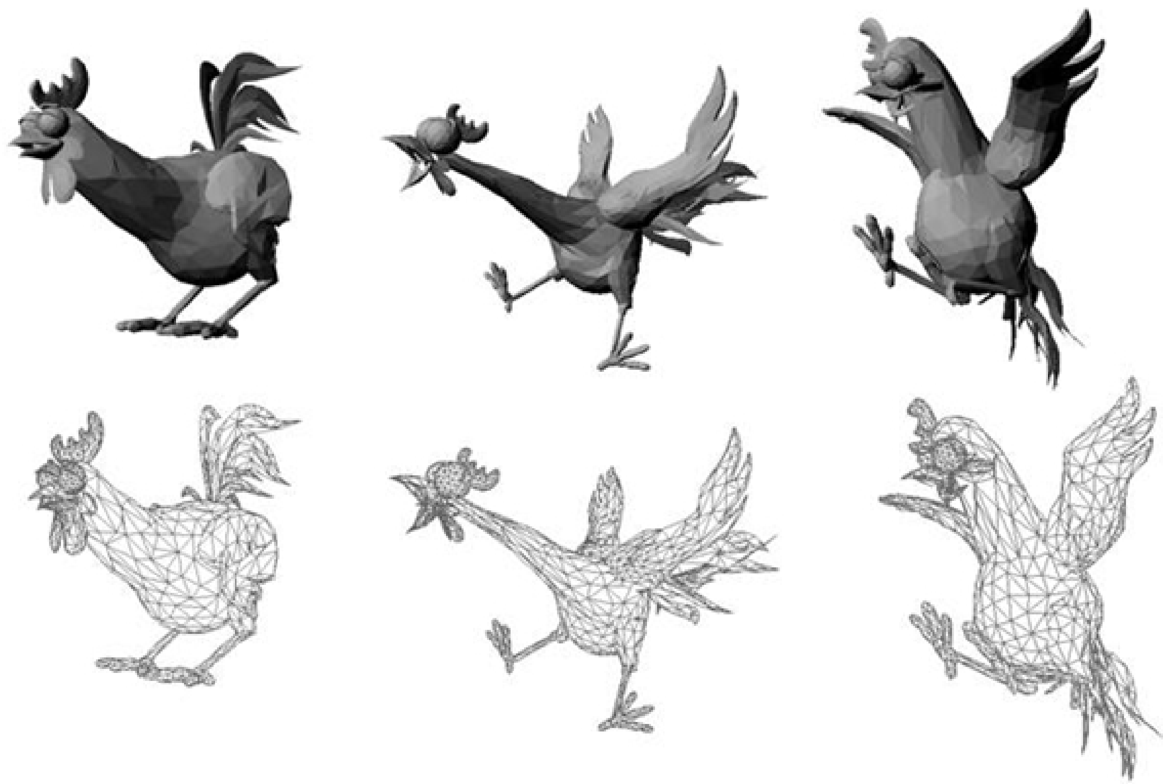
Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

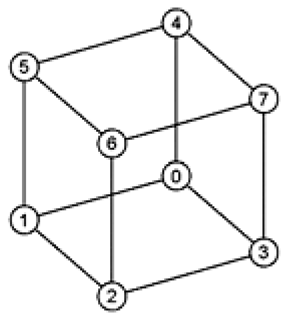
3D Applications

- 2D points: (x,y) coordinates
- 3D points: have (x,y,z) coordinates



Setting up 3D Applications

- Programming 3D similar to 2D
  1. Load representation of 3D object into data structure

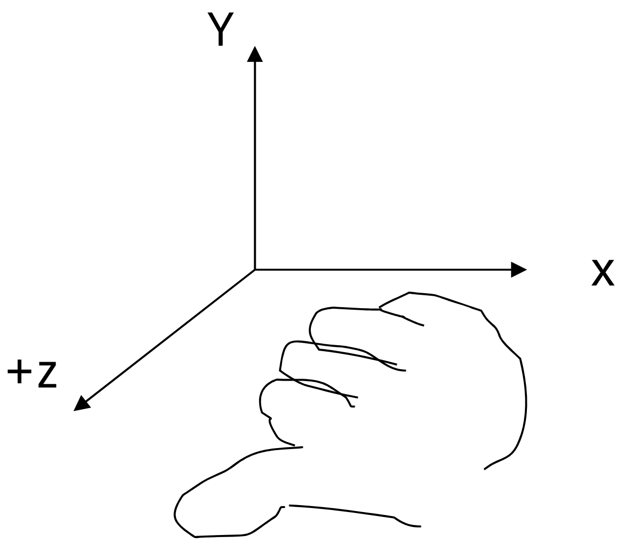


Each vertex has (x,y,z) coordinates.
Store as **vec3**, **glUniform3f NOT vec2**

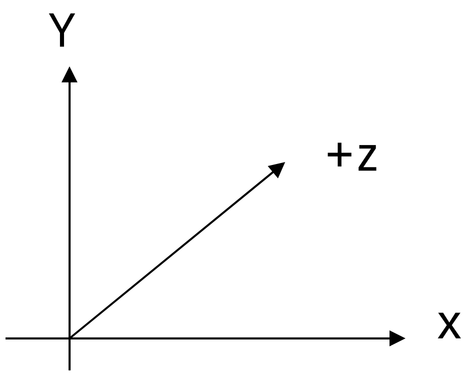
  2. Draw 3D object
  3. **Set up Hidden surface removal:** Correctly determine order in which primitives (triangles, faces) are rendered (e.g Blocked faces **NOT** drawn)

3D Coordinate Systems

- Vertex (x,y,z) positions specified on coordinate system
- OpenGL uses **right hand coordinate system**



Right hand coordinate system
Tip: sweep fingers x-y: thumb is z



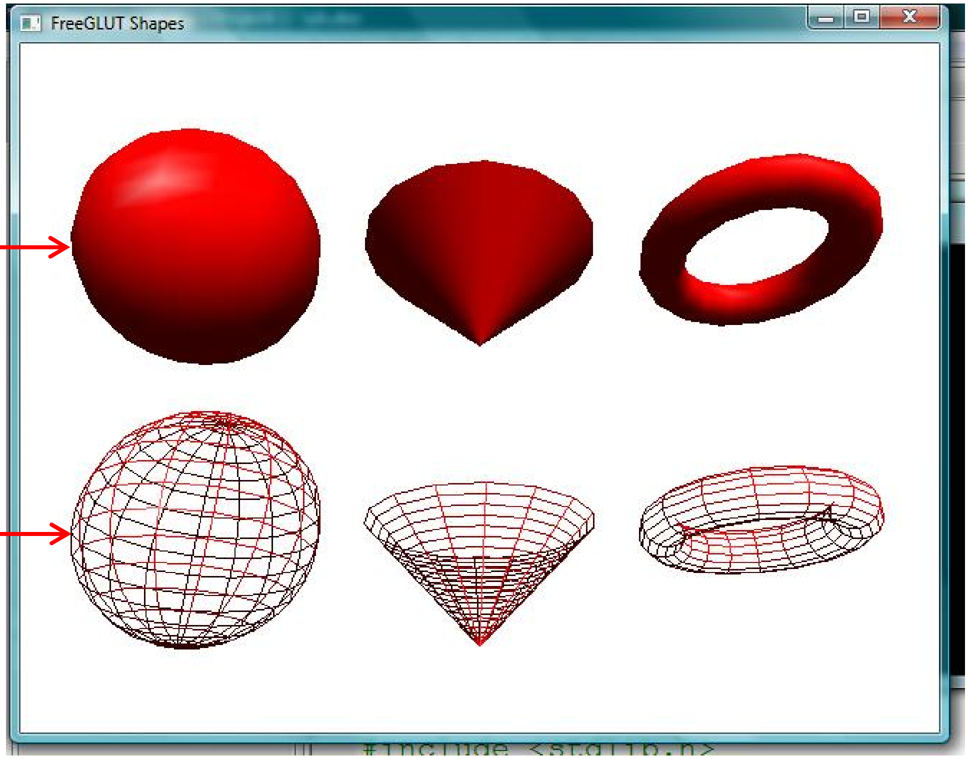
Left hand coordinate system
•Not used in OpenGL

Generating 3D Models: GLUT Models

- Make GLUT 3D calls in **OpenGL program** to generate vertices describing different shapes (Restrictive?)
- Two types of GLUT models:
  - Wireframe Models
  - Solid Models

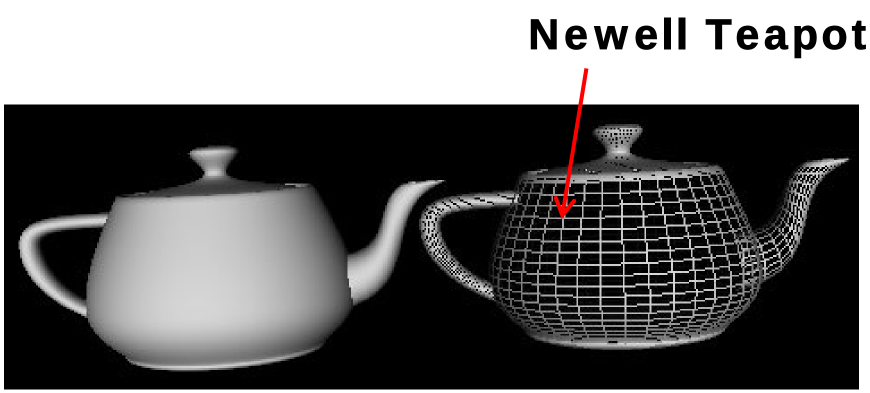
Solid models

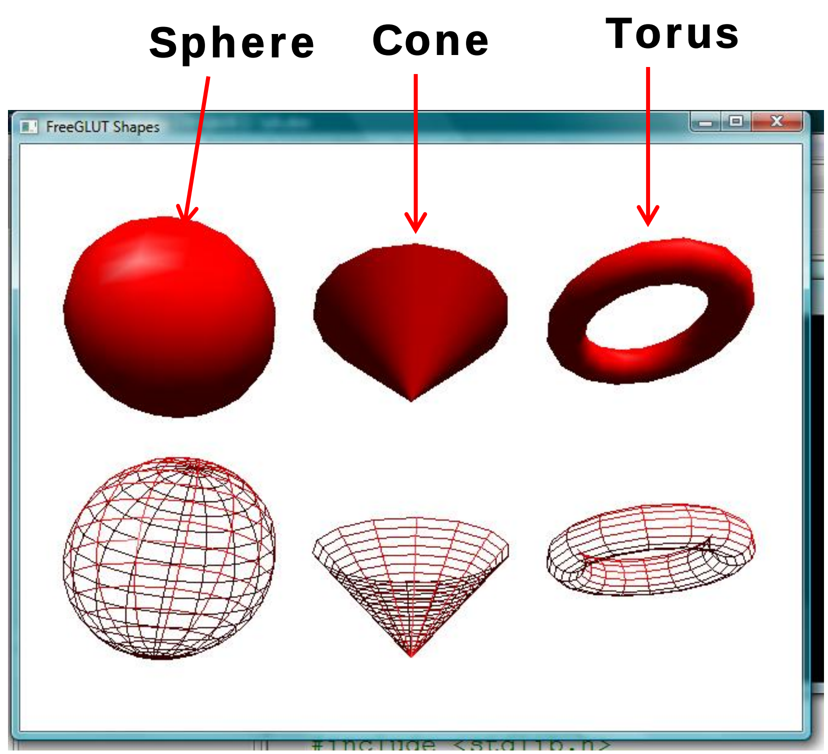
Wireframe models



3D Modeling: GLUT Models

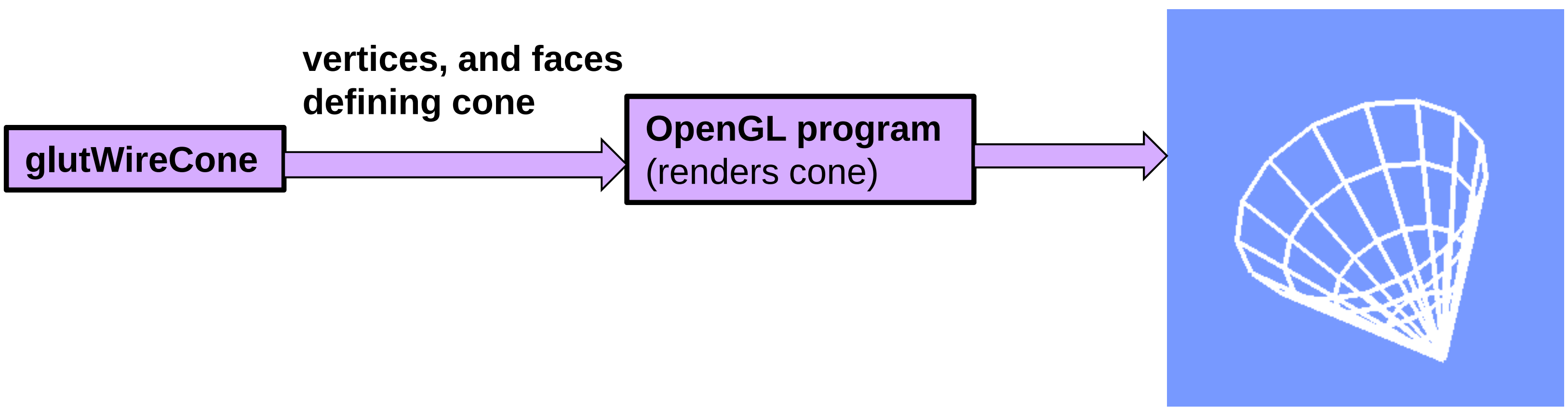
- Basic Shapes
  - **Cone:** glutWireCone( ), glutSolidCone( )
  - **Sphere:** glutWireSphere( ), glutSolidSphere( )
  - **Cube:** glutWireCube( ), glutSolidCube( )
- More advanced shapes:
  - Newell Teapot: (symbolic)
  - Dodecahedron, Torus





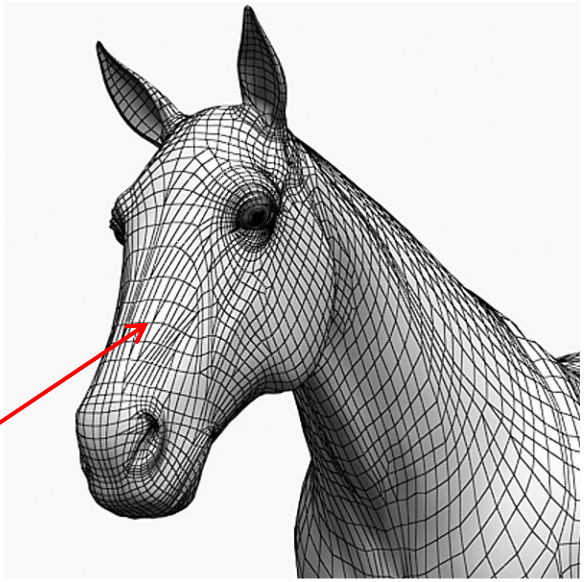
3D Modeling: GLUT Models

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



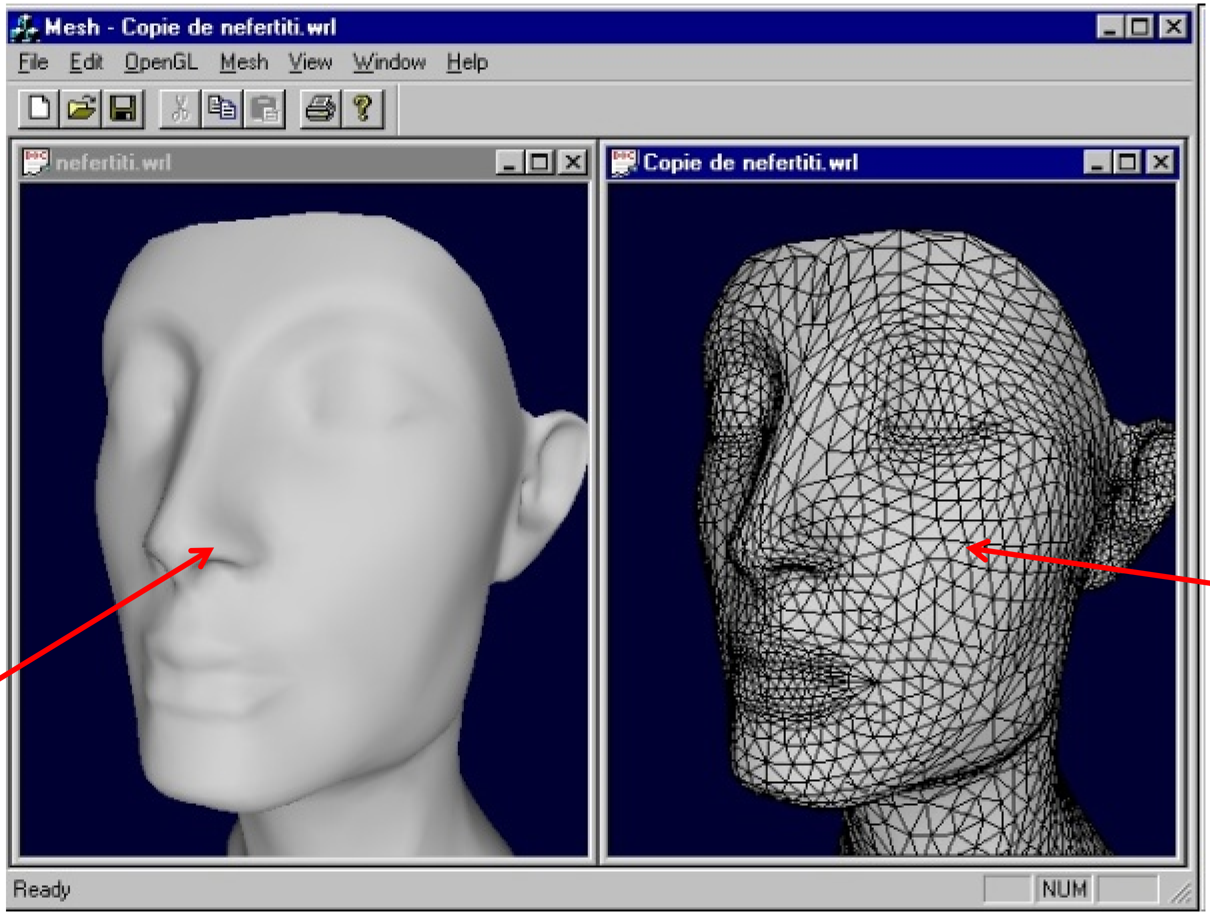
Polygonal Meshes

- Modeling with GLUT shapes (cube, sphere, etc) too restrictive
- Difficult to approach realism. E.g. model a horse
- 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



Each face of mesh is a polygon

Polygonal Mesh Example

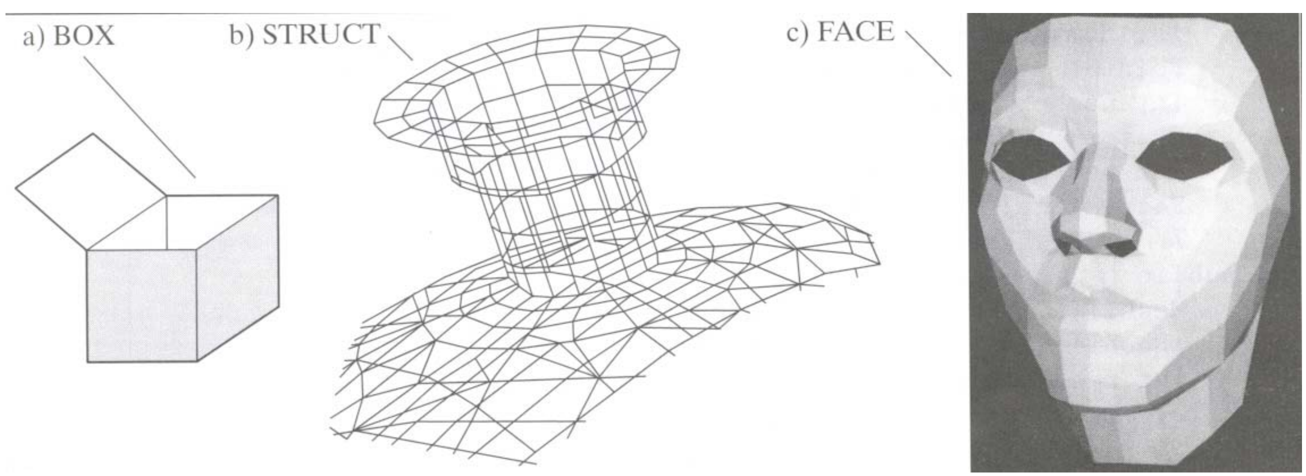


Smoothed Out with Shading (later)

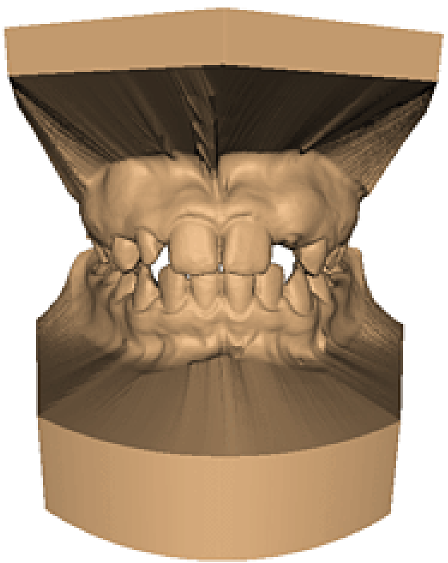
Mesh (wireframe)

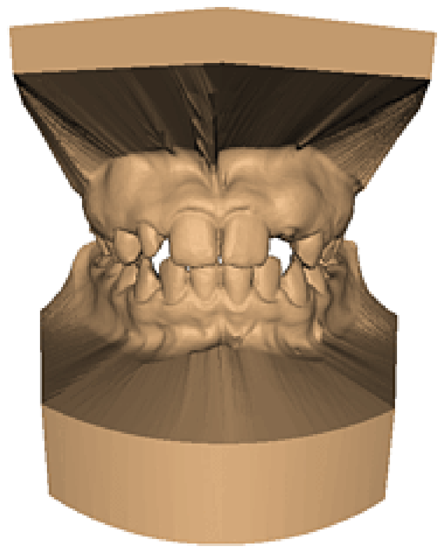
Polygonal Meshes

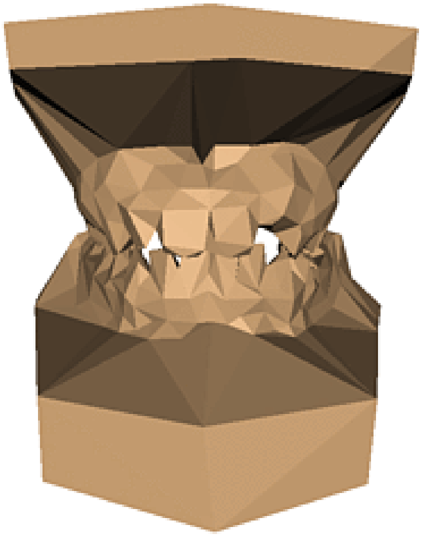
- Meshes now standard in graphics
- OpenGL
  - Good at drawing polygons, triangles
  - Mesh = sequence of polygons forming thin skin around object
- Simple meshes exact. (e.g barn)
- Complex meshes approximate (e.g. human face)



Meshes at Different Resolutions







Original: 424,000 triangles

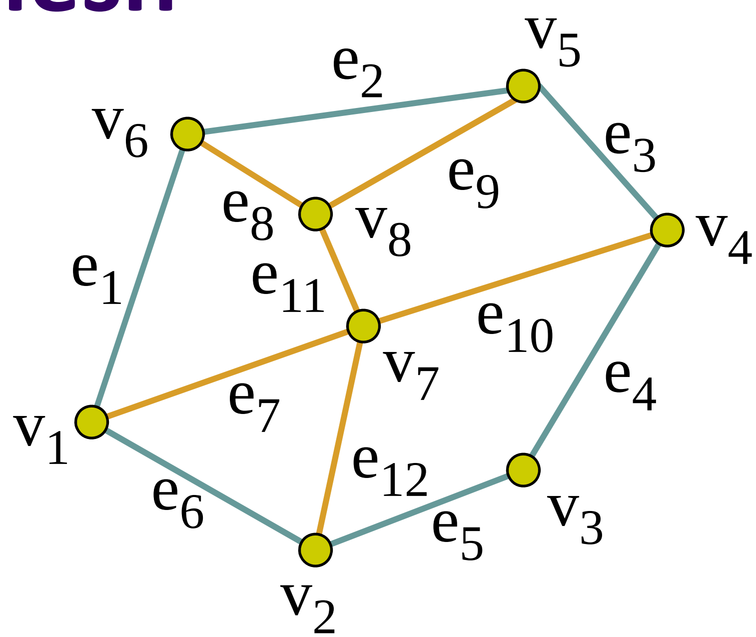
60,000 triangles (14%).

1000 triangles (0.2%)

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

Representing a Mesh

- Consider a mesh



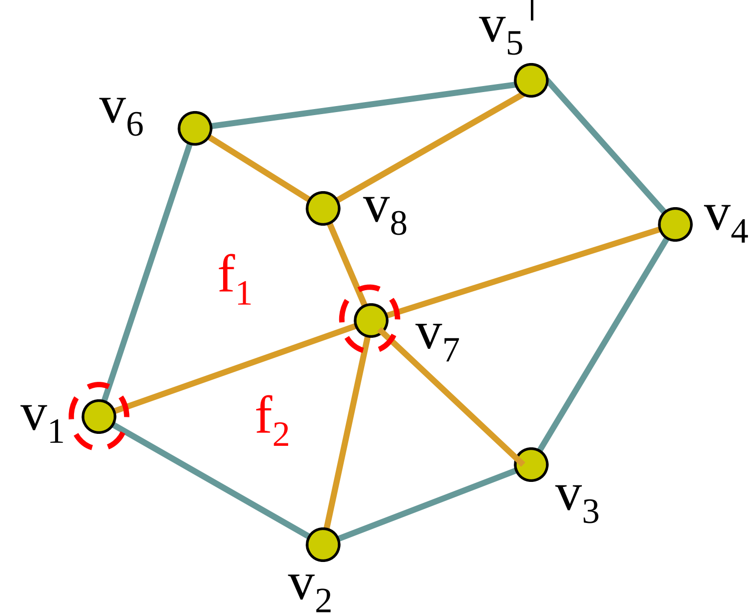
- There are 8 vertices and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges (shown in orange)
- Each vertex has a location $v_i = (x_i\ y_i\ z_i)$

Simple Representation

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

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

Issues with Simple Representation



- Declaring face f1

```
vertex[i]   = vec3(x1, y1, z1);
vertex[i+1] = vec3(x7, y7, z7);
vertex[i+2] = vec3(x8, y8, z8);
vertex[i+3] = vec3(x6, y6, z6);
```

- Declaring face f2

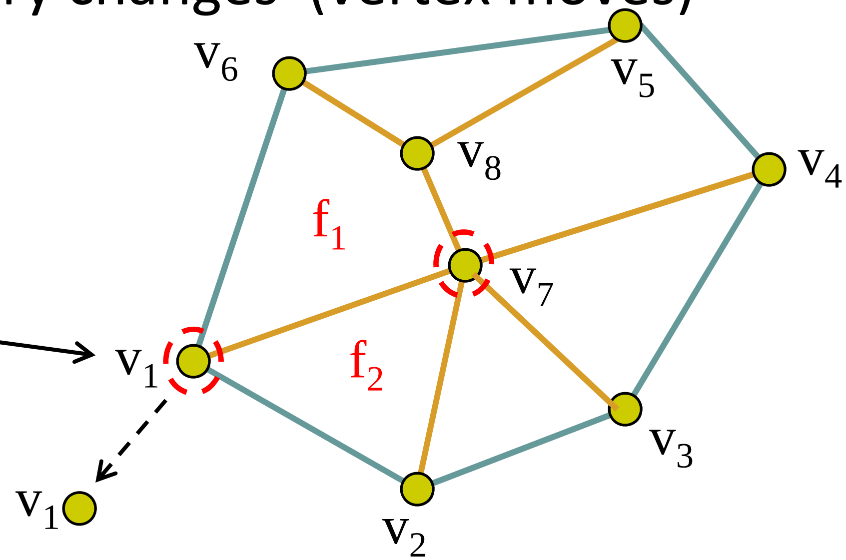
```
vertex[i]   = vec3(x1, y1, z1);
vertex[i+1] = vec3(x2, y2, z2);
vertex[i+2] = vec3(x7, y7, z7);
```

- Inefficient and unstructured
  - **Repeats:** vertices v1 and v7 repeated while declaring f1 and f2
  - Shared vertices shared declared multiple times
  - Delete vertex? Move vertex? Search for all occurences of vertex

Geometry vs Topology

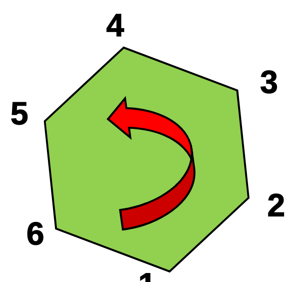
- Better data structures separate **geometry** from **topology**
  - **Geometry:** (x,y,z) locations of the vertices
  - **Topology:** How vertices and edges are connected
  - **Example:**
    - A polygon is **ordered list** of vertices
    - An edge connecting successive pairs of vertices
  - Topology holds even if geometry changes (vertex moves)

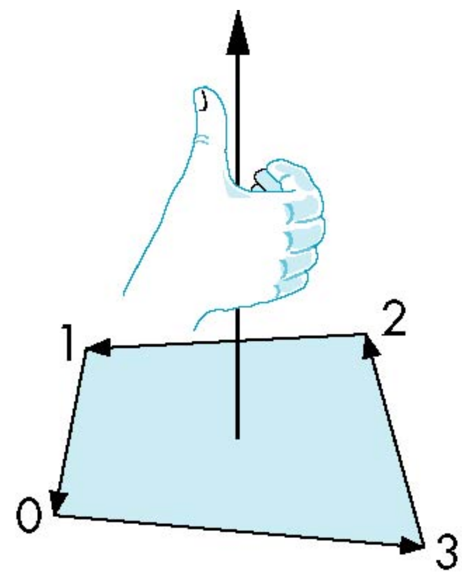
Example: even if we move (x,y,z) location of v1, v1 still connected to v6, v7 and v2



Polygon Traversal Convention

- Use the ***right-hand rule*** = **counter-clockwise** encirclement of outward-pointing normal
- Focus on direction of traversal
  - Orders $\{v_1, v_0, v_3\}$ and $\{v_3, v_2, v_1\}$ are same *(ccw)*
  - Order $\{v_1, v_2, v_3\}$ is different *(clockwise)*

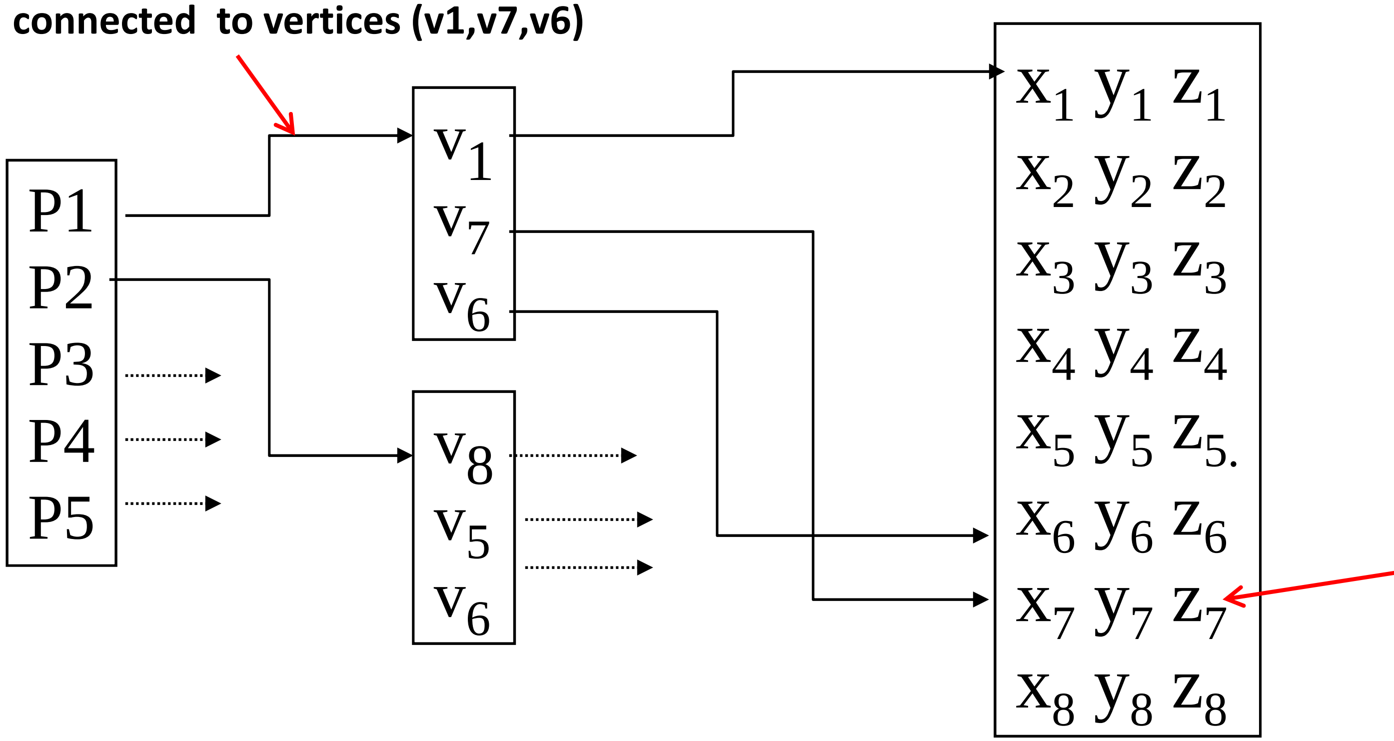




Vertex Lists

- **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)

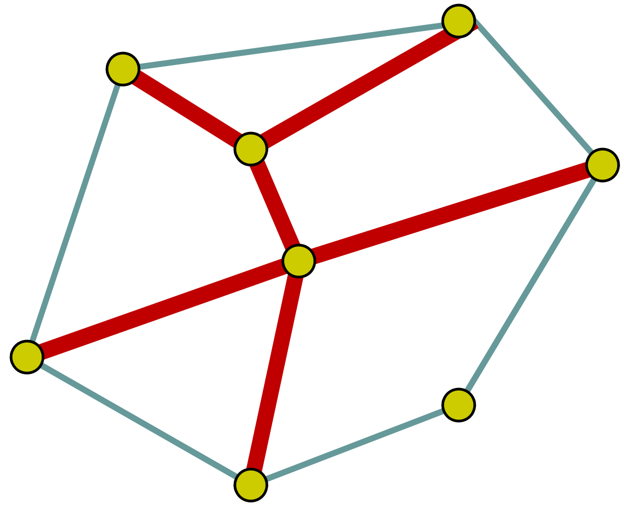
Topology example: Polygon P1 of mesh is connected to vertices (v1,v7,v6)



Geometry example: Vertex v7 coordinates are (x7,y7,z7). Note: If v7 moves, changed once in vertex list

Vertex List Issue: Shared Edges

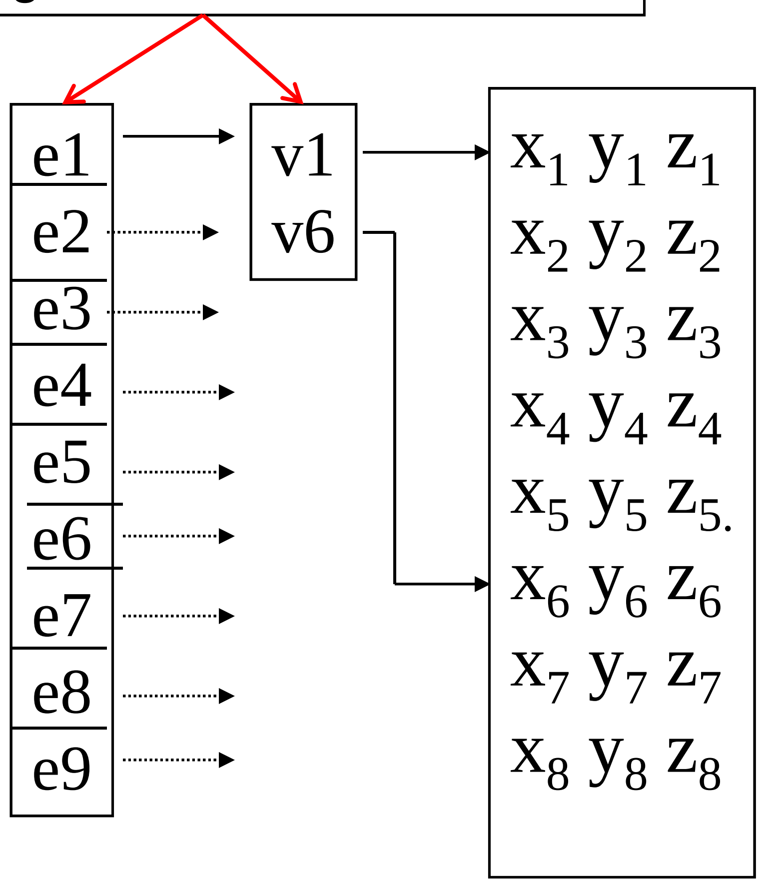
- Vertex lists draw filled polygons correctly
- If each polygon is drawn by its edges, shared edges are drawn twice

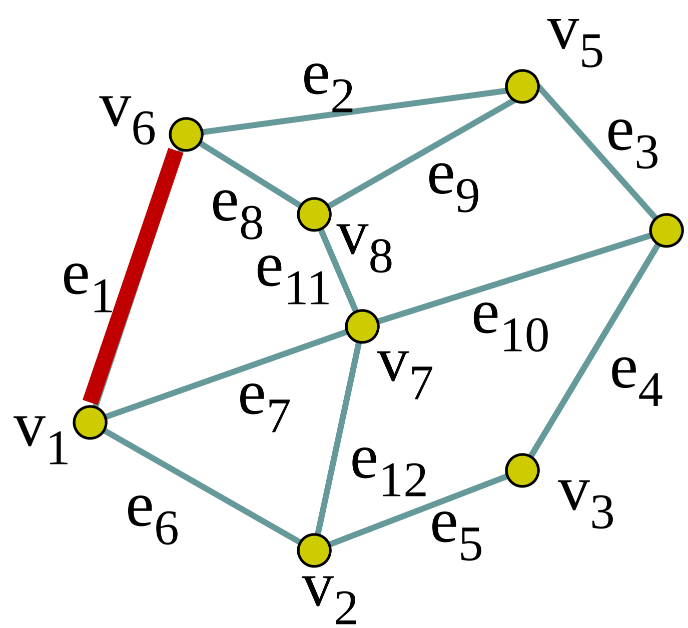


- **Alternatively:** Can store mesh by *edge list*

Edge List

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

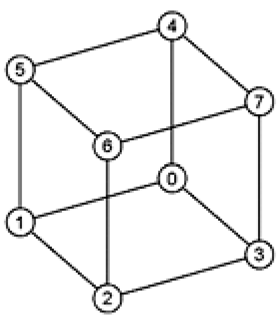




Note polygons are not represented

Modeling a Cube

- In 3D, declare vertices as (x,y,z) using **point3 v[3]**
- Define **global arrays** for vertices and colors



x y z

```
typedef vec3 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)};
```

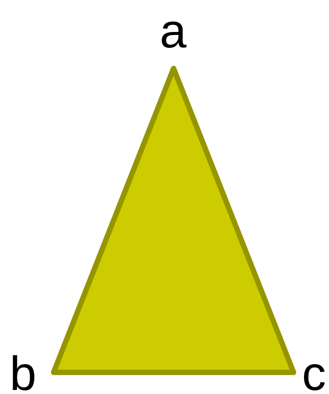
r g b

```
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 list of indices

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

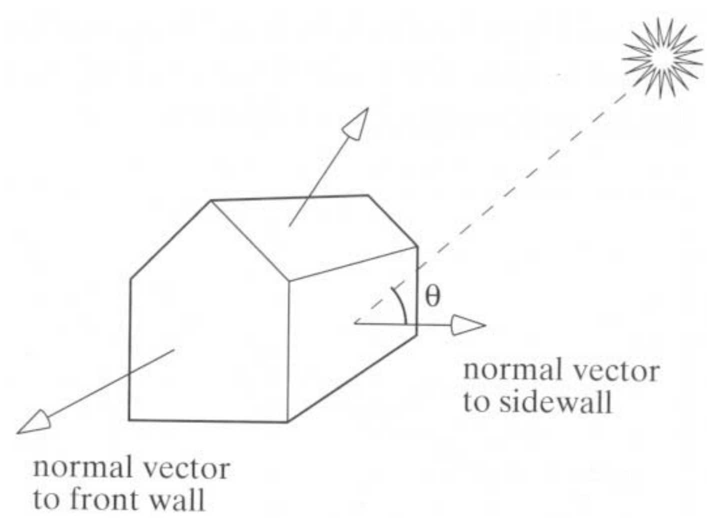
```
void triangle(int a, int b, int c, int d)
{
   vcolors[i] = colors[d];
   position[i] = vertices[a];
   vcolors[i+1] = colors[d]);
   position[i+1] = vertices[b];
   vcolors[i+2] = colors[d];
   position[i+2] = vertices[c];
   i+=3;
 }
```



Variables a, b, c are indices into vertex array
Variable d is index into color array
Note: Same face, so all three vertices have same color

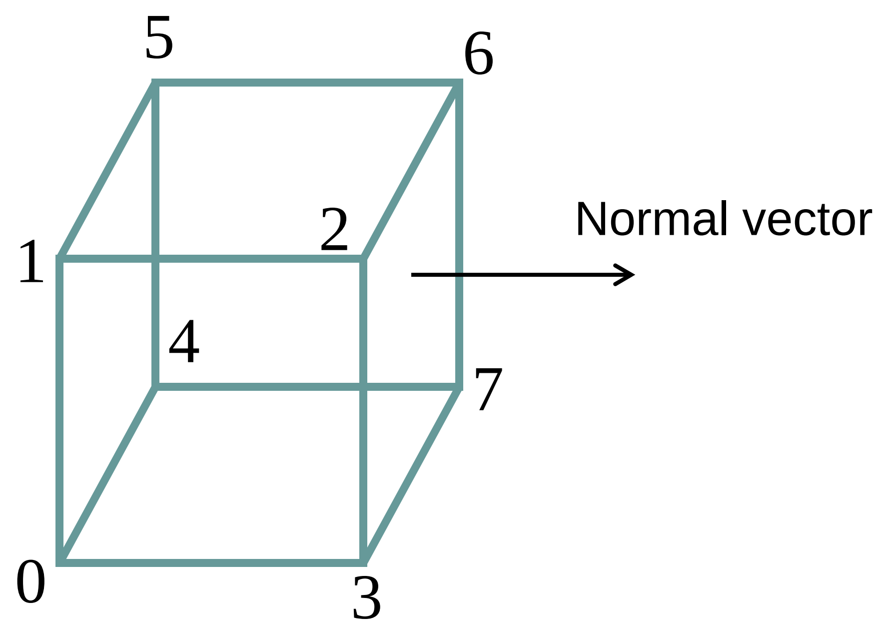
Normal Vector

- **Normal vector:** Direction each polygon is facing
- Each mesh polygon has a **normal vector**
- Normal vector used in shading
- **Normal vector • light vector** determines shading (Later)



Draw 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: vertices ordered (**counterclockwise**)
so that we obtain correct outward facing normals

Old Way for Storing Vertices: Inefficient

- Previously drew cube by its 6 faces using
  - 6 **glBegin**, 6 **glEnd**
  - 6 **glColor**
  - 24 **glVertex**
  - More commands if we use texture and lighting
  - E.g: to draw each face

```
glBegin(GL_QUAD)
      glVertex(x1, y1, z1);
      glVertex(x2, y2, z2);
      glVertex(x3, y3, z3);
      glVertex(x4, y4, z4);
glEnd( );
```

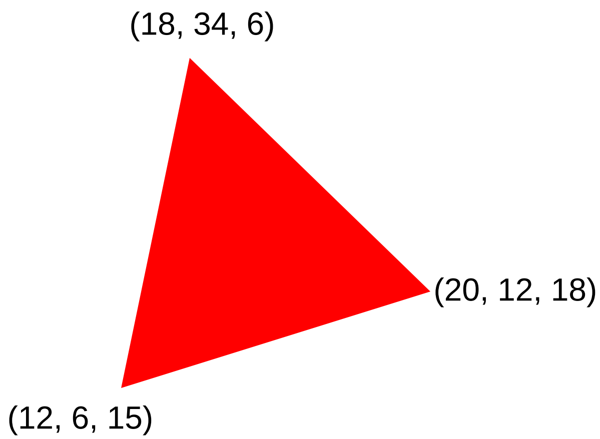
New Way: Vertex Representation and Storage

- We have declare vertex lists, edge lists and arrays
- But OpenGL expects meshes passed to have a specific structure
- We now study that structure….

Vertex Arrays

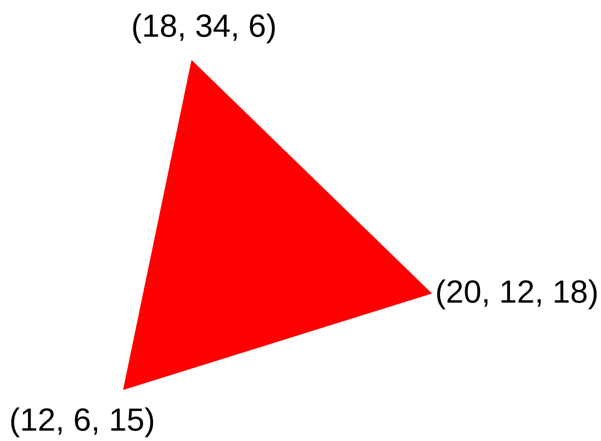
- **Previously:** OpenGL provided a facility called ***vertex arrays*** for storing rendering data
- Six types of arrays were supported initially
  - Vertices
  - Colors
  - Color indices
  - Normals
  - Texture coordinates
  - Edge flags
- Now vertex arrays can be used for **any attributes**

Vertex Attributes

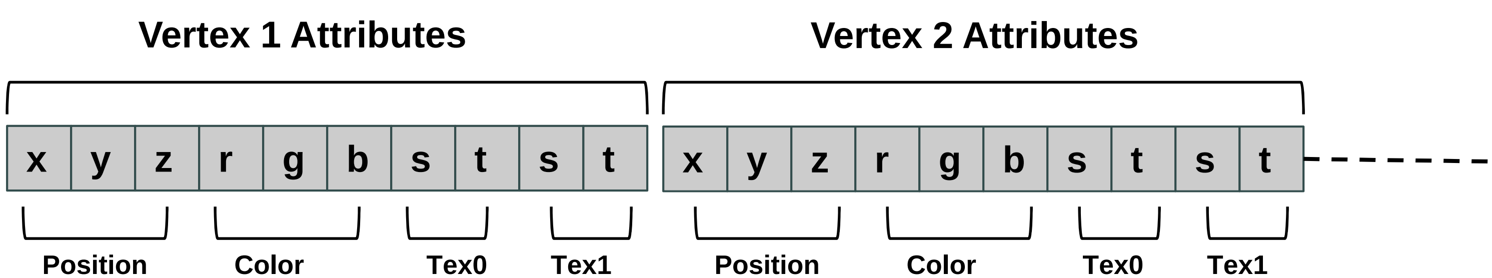


- Vertices can have attributes
  - Position (e.g 20, 12, 18)
  - Color (e.g. red)
  - Normal (x,y,z)
  - Texture coordinates

Vertex Attributes

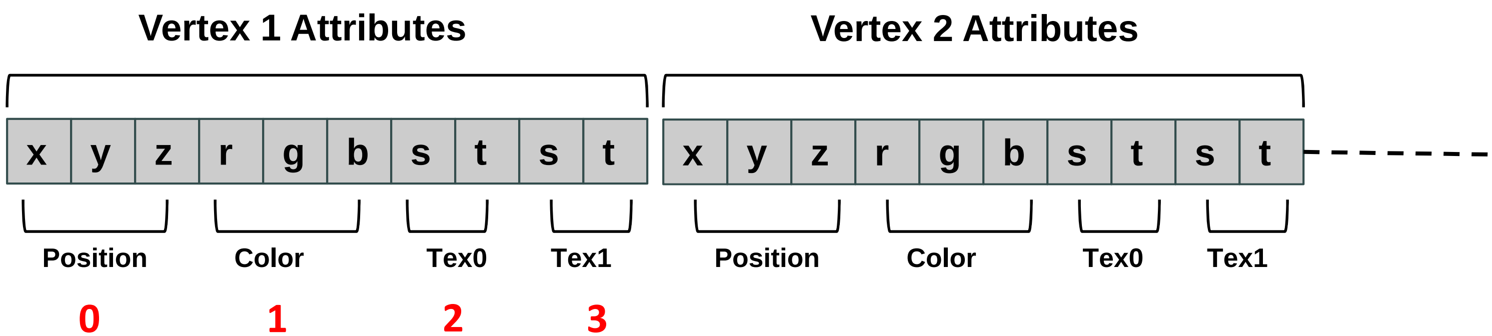


- Store vertex attributes in single Array (array of structures)



Declaring Array of Vertex Attributes

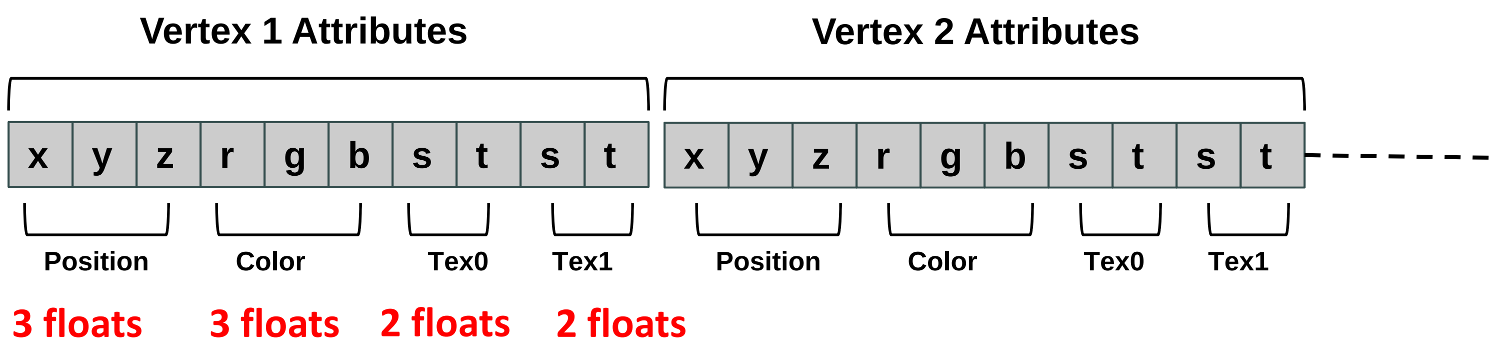
- Consider the following array of vertex attributes



- So we can define attribute positions (per vertex)

```
#define VERTEX_POS_INDEX           0
#define VERTEX_COLOR_INDEX         1
#define VERTEX_TEXCOORD0_INDX      2
#define VERTEX_TEXCOORD1_INDX      3
```

Declaring Array of Vertex Attributes

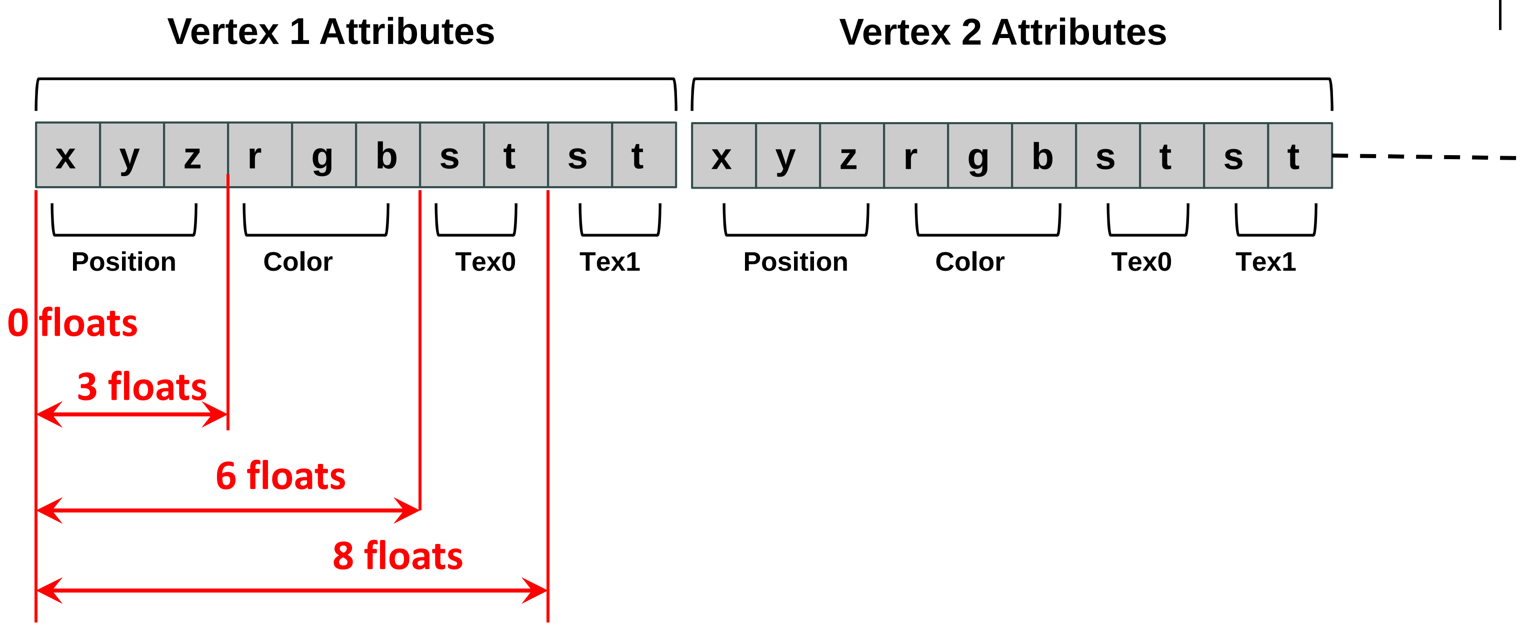


- Also define number of floats (storage) for each vertex attribute

```
#define VERTEX_POS_SIZE              3    // x, y and z
#define VERTEX_COLOR_SIZE            3    // r, g and b
#define VERTEX_TEXCOORD0_SIZE        2    // s and t
#define VERTEX_TEXCOORD1_SIZE        2    // s and t

#define VERTEX_ATTRIB_SIZE    VERTEX_POS_SIZE + VERTEX_COLOR_SIZE + \
                              VERTEX_TEXCOORD0_SIZE + \
                              VERTEX_TEXCOORD1_SIZE
```

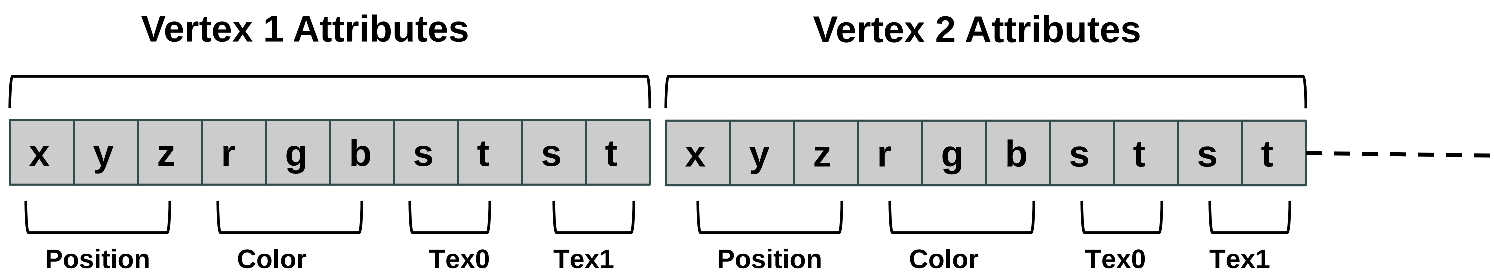
Declaring Array of Vertex Attributes



- Define offsets (# of floats) of each vertex attribute from beginning

```
#define VERTEX_POS_OFFSET           0
#define VERTEX_COLOR_OFFSET         3
#define VERTEX_TEXCOORD0_OFFSET     6
#define VERTEX_TEXCOORD1_OFFSET     8
```

Allocating Array of Vertex Attributes



Recall

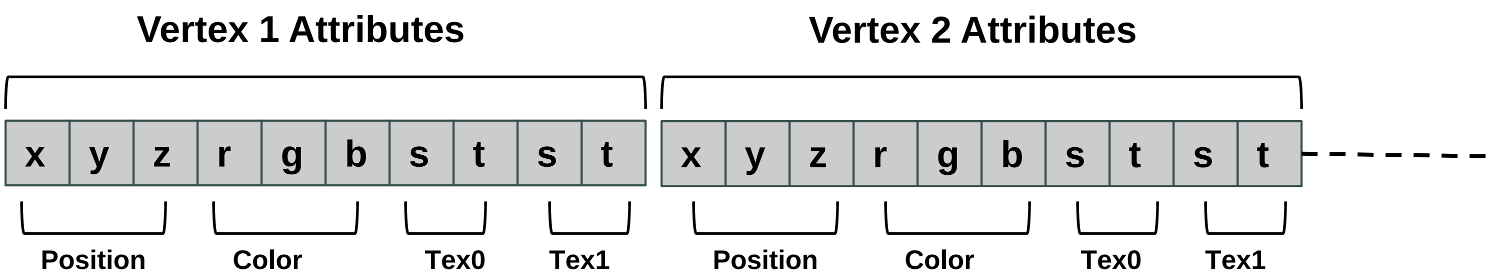
- Allocate memory for entire array of vertex attributes

```
#define VERTEX_ATTRIB_SIZE    VERTEX_POS_SIZE + VERTEX_COLOR_SIZE + \
                              VERTEX_TEXCOORD0_SIZE + \
                              VERTEX_TEXCOORD1_SIZE
```

```
float *p = malloc(numVertices * VERTEX_ATTRIB_SIZE * sizeof(float));
```

Allocate memory for all vertices

Specifying Array of Vertex Attributes



- `glVertexAttribPointer` used to specify vertex attributes
- Example: to specify vertex position attribute

Position 0

3 floats (x, y, z)

```
glVertexAttribPointer(VERTEX_POS_INDX, VERTEX_POS_SIZE,
                      GL_FLOAT, GL_FALSE,
                      VERTEX_ATTRIB_SIZE * sizeof(float), p);
```

Data should not Be normalized

Data is floats

Stride: distance between consecutive vertices

Pointer to data

```
glEnableVertexAttribArray(0);
```

- `do same for normal, tex0 and tex1`

References

- Angel and Shreiner, Interactive Computer Graphics, 6th edition, Chapter 3
- Hill and Kelley, Computer Graphics using OpenGL, 3rd edition