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 3D set up
- Introduce simple data structures for 3D models
  - Vertex lists
  - Edge lists
- Deprecated OpenGL vertex arrays
- Drawing 3D objects
3D Applications

- 2D points: (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
  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)

Each vertex has (x,y,z) coordinates. Store as vec3, glUniform3f NOT vec2
3D Coordinate Systems

- All vertex \((x, y, z)\) positions are with respect to a coordinate system
- OpenGL uses 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

- One way of generating 3D shapes is by using GLUT 3D models (Restrictive?)

- **Note:** Simply make GLUT 3D calls in *OpenGL program* to generate vertices describing different shapes

- Two types 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: 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
- Example: `glutWireCone` generates sequence of vertices, and faces defining `cone` and connectivity
- Generated vertices and faces passed to OpenGL for rendering

```
glutWireCone generates sequence of vertices, and faces defining cone

OpenGL program receives vertices and Faces, renders them
```
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

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

```cpp
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
  
  \[
  \text{vertex}[i] = \text{vec3}(x_1, y_1, z_1); \\
  \text{vertex}[i+1] = \text{vec3}(x_7, y_7, z_7); \\
  \text{vertex}[i+2] = \text{vec3}(x_8, y_8, z_8); \\
  \text{vertex}[i+3] = \text{vec3}(x_6, y_6, z_6);
  \]

- Declaring face f2
  
  \[
  \text{vertex}[i] = \text{vec3}(x_1, y_1, z_1); \\
  \text{vertex}[i+1] = \text{vec3}(x_2, y_2, z_2); \\
  \text{vertex}[i+2] = \text{vec3}(x_7, y_7, z_7);
  \]

- Inefficient and unstructured
  
  - In example, vertices v1 and v7 repeated while declaring f1 and f2
  - Vertices shared by many polygons are declared multiple times
  - Consider deleting vertex, moving vertex to new location
  - Must search for all faces in which vertex occurs
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 an **ordered list** of vertices with 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 \(v_1\), \(v_1\) still connected to \(v_6, v_7\) and \(v_2\)
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 \( \{v_1, v_0, v_3\} \) and \( \{v_3, v_2, v_1\} \) are equivalent in same polygon, rendered same way rendered by OpenGL
- But order of \( \{v_1, v_2, v_3\} \) 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)

**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 edge 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

```c
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)};

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

```c
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)
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
References