Computer Graphics (CS 4731)
Lecture 8: 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: Main Steps

- Programming 3D similar to 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 \texttt{vec3} NOT \texttt{vec2}
3D Coordinate Systems

- Vertex \((x, y, z)\) positions specified on coordinate system
- OpenGL uses **right hand coordinate system**
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 Models](image)
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 Meshes

- Mesh = sequence of polygons forming thin skin around object
- OpenGL Good at drawing polygons, triangles
- Meshes now standard in graphics
- Simple meshes exact. (e.g. barn)
- Complex meshes approximate (e.g. human face)
Same Mesh 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
  
  \[
  \begin{align*}
  \text{vertex}[i] &= \text{vec3}(x_1, y_1, z_1); \\
  \text{vertex}[i+1] &= \text{vec3}(x_7, y_7, z_7); \\
  \text{vertex}[i+2] &= \text{vec3}(x_8, y_8, z_8); \\
  \text{vertex}[i+3] &= \text{vec3}(x_6, y_6, z_6);
  \end{align*}
  \]

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

- Inefficient and unstructured
  
  - **Repeats:** vertices \textbf{v1 and v7 repeated} while declaring f1 and f2
  
  - Shared vertices shared declared multiple times
  
  - Delete vertex? Move vertex? Search for all occurrences of vertex
Geometry vs Topology

- **Geometry**: $(x,y,z)$ locations of the vertices
- **Topology**: How vertices and edges are connected

Good data structures separate **geometry** from **topology**

- **Example**:
  - A polygon is **ordered list** of vertices
  - An edge connects 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

- **Convention:** traverse vertices **counter-clockwise** around 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**)
- **Normal vector:** Direction each polygon is facing
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
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)
- **Later:** pass array to OpenGL, specify attributes, order, position using `glVertexAttribPointer`

---

### Vertex 1 Attributes

- **Position**
- **Color**
- **Tex0**
- **Tex1**

### Vertex 2 Attributes

- **Position**
- **Color**
- **Tex0**
- **Tex1**
Declaring Array of Vertex Attributes

- Consider the following array of vertex attributes

```
0 1 2 3
```

- So we can define attribute positions (per vertex)

```c
#define VERTEX_POS_INDEX 0
#define VERTEX_COLOR_INDEX 1
#define VERTEX_TEXCOORD0_INDEX 2
#define VERTEX_TEXCOORD1_INDEX 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

```c
#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

- 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));
```
Specifying Array of Vertex Attributes

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

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

- do same for normal, tex0 and tex1
Full Example: Rotating Cube in 3D

- **Desired Program behaviour:**
  - Draw colored cube
  - Continuous rotation about X, Y or Z axis
    - Idle function called repeatedly when nothing to do
    - Increment angle of rotation in idle function
  - Use 3-button mouse to change direction of rotation
    - Click left button -> rotate cube around X axis
    - Click middle button -> rotate cube around Y axis
    - Click right button -> rotate cube around Z axis

- **Use default camera**
  - If we don’t set camera, we get a default camera
  - Located at origin and points in the negative z direction
Cube Vertices

Declare array of (x,y,z,w) vertex positions for a unit cube centered at origin (Sides aligned with axes)

```c
point4 vertices[8] = {
  0 point4( -0.5, -0.5, 0.5, 1.0 ),
  1 point4( -0.5, 0.5, 0.5, 1.0 ),
  2 point4( 0.5, 0.5, 0.5, 1.0 ),
  3 point4( 0.5, -0.5, 0.5, 1.0 ),
  4 point4( -0.5, -0.5, -0.5, 1.0 ),
  5 point4( -0.5, 0.5, -0.5, 1.0 ),
  6 point4( 0.5, 0.5, -0.5, 1.0 ),
  7 point4( 0.5, -0.5, -0.5, 1.0 )
};
```

Declare array of vertex colors (set of RGBA colors vertex can have)

```c
color4 vertex_colors[8] = {
  color4( 0.0, 0.0, 0.0, 1.0 ),  // black
  color4( 1.0, 0.0, 0.0, 1.0 ),  // red
  color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
  color4( 0.0, 1.0, 0.0, 1.0 ),  // green
  color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
  color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
  color4( 1.0, 1.0, 1.0, 1.0 ),  // white
  color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```
Color Cube

// generate 6 quads,  
// sides of cube

void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}

point4 vertices[8] = {
    0 point4( -0.5, -0.5,  0.5, 1.0 ),
    1 point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    4 point4( -0.5, -0.5, -0.5, 1.0 ),
    5 point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};

Function **quad** is
Passed vertex indices
**Quad Function**

Quad Function generates two triangles \((a,b,c)\) and \((a,c,d)\) for each face and assigns colors to the vertices.

```c
int Index = 0;  // Index goes 0 to 5, one for each vertex of face
void quad( int a, int b, int c, int d )
{
  colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++;
  colors[Index] = vertex_colors[b]; points[Index] = vertices[b]; Index++;
  colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++;
  colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++;
  colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++;
  colors[Index] = vertex_colors[d]; points[Index] = vertices[d]; Index++;
}
```

- **quad 0** = points[0 - 5]
- **quad 1** = points[6 - 11]
- **quad 2** = points[12 - 17] ... etc

Points[] array to be Sent to GPU

Read from appropriate index of unique positions declared
Initialization

```c
void init()
{
    colorcube(); // Generates cube data in application using quads

    // Create a vertex array object
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

    // Create a buffer object and move data to GPU
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof(points) +
                 sizeof(colors), NULL, GL_STATIC_DRAW );

    Points[] array of vertex positions sent to GPU
    colors[] array of vertex colors sent to GPU
}
```
Initialization II

Send `points[ ]` and `colors[ ]` data to GPU separately using `glBufferSubData`

```c
glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(points), points );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(points), sizeof(colors), colors );
```

// Load vertex and fragment shaders and use the resulting shader program
GLuint program = InitShader( "vshader36.glsl", "fshader36.glsl" );
glUseProgram( program );
Initialization III

// set up vertex arrays

GLint vPosition = glGetUniformLocation( program, "vPosition" );
EnableVertexAttribArray( vPosition );
VertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0) );

GLint vColor = glGetUniformLocation( program, "vColor" );
EnableVertexAttribArray( vColor );
VertexAttribPointer( vColor, 4, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(sizeof(points)) );

theta = glGetUniformLocation( program, "theta" );

Want to Connect rotation variable theta
in program to variable in shader
Display Callback

```c
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT );

    glUniform3fv( theta, 1, theta );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );

    glutSwapBuffers();
}
```

Draw series of triangles forming cube
enum { Xaxis = 0, Yaxis = 1, Zaxis = 2, NumAxes = 3 };

void mouse( int button, int state, int x, int y )
{
    if ( state == GLUT_DOWN ) {
        switch( button ) {
            case GLUT_LEFT_BUTTON:  axis = Xaxis;  break;
            case GLUT_MIDDLE_BUTTON: axis = Yaxis;  break;
            case GLUT_RIGHT_BUTTON:  axis = Zaxis;  break;
        }
    }
}

Select axis (x,y,z) to rotate around
Using mouse click
Idle Callback

```c
void idle( void )
{
    theta[axis] += 0.01;

    if ( theta[axis] > 360.0 ) {
        theta[axis] -= 360.0;
    }

    glutPostRedisplay();
}
```

The `idle()` function is called whenever nothing to do.

Use it to increment rotation angle in steps of `theta = 0.01` around currently selected axis.

```c
void main( void )
{
    .......

    glutIdleFunc( idle );
    .......
}
```

Note: still need to:

- Apply rotation by `(theta)` in shader
References