Building Models

Adapted From:
Ed Angel
Professor of Emeritus of Computer Science
University of New Mexico
Objectives

Introduce simple data structures for building polygonal models

- Vertex lists
- Edge lists

Implementation with VBOs
Representing a Mesh

Consider a mesh

There are 6 nodes and 9 edges

4 interior polygons
3 interior (shared) edges

Each vertex has a location \( v_i = (x_i, y_i, z_i) \)
Simple Representation

Define each polygon by the geometric locations of its vertices
Leads to OpenGL code such as

\[
\begin{align*}
vertex[i] &= \text{vec3}(x_1, y_1, z_1); \\
vertex[i+1] &= \text{vec3}(x_2, y_2, z_2); \\
vertex[i+2] &= \text{vec3}(x_3, y_3, z_3); \\
i += 3; \\
vertex[i] &= \text{vec3}(x_2, y_2, z_2); \\
vertex[i+1] &= \text{vec3}(x_3, y_3, v_3); \\
vertex[i+2] &= \text{vec3}(x_4, y_4, v_4); \\
\end{align*}
\]

Inefficient and unstructured

Shared vertices are repeated
Consider moving a vertex to a new location
Geometry vs Topology

Generally it is a good idea to look for data structures that separate the geometry from the topology

- **Geometry**: locations of the vertices
- **Topology**: organization of the vertices and edges

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 Lists

Put the geometry in an array
Use pointers from the vertices into this array
Introduce a polygon list

P1  P2  P3  P4  P5

v1  v7  v6

v8  v5  v6

go to

x1 y1 z1
x2 y2 z2
x3 y3 z3
x4 y4 z4
x5 y5 z5
x6 y6 z6
x7 y7 z7
x8 y8 z8
Shared Edges

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

Can also store mesh by edge list
Edge List

<table>
<thead>
<tr>
<th>e1</th>
<th>v1</th>
<th>x1 y1 z1</th>
</tr>
</thead>
<tbody>
<tr>
<td>e2</td>
<td></td>
<td>x2 y2 z2</td>
</tr>
<tr>
<td>e3</td>
<td></td>
<td>x3 y3 z3</td>
</tr>
<tr>
<td>e4</td>
<td></td>
<td>x4 y4 z4</td>
</tr>
<tr>
<td>e5</td>
<td></td>
<td>x5 y5 z5</td>
</tr>
<tr>
<td>e6</td>
<td></td>
<td>x6 y6 z6</td>
</tr>
<tr>
<td>e7</td>
<td></td>
<td>x7 y7 z7</td>
</tr>
<tr>
<td>e8</td>
<td></td>
<td>x8 y8 z8</td>
</tr>
<tr>
<td>e9</td>
<td>v6</td>
<td></td>
</tr>
</tbody>
</table>
Cube Model Example
See 3D_ModelingDemo_VL
Inward and Outward Facing Polygons

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.

Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal.

OpenGL can treat inward and outward facing polygons differently.
Using VBOs for Indexed Data

Option I
- Stored vertices, colors, etc.
- Stored index info
- Manually turn indexed data into raw vertex data for primitives

Leads to duplicates and defeats the purpose of the indexing

Option II
- VBO with glDrawElements
glDraw Elements

Setup your VBO
Specify an *Index VBO*
use glDrawElements instead of glDrawArrays
Example

point2 vertices[3] = {
    point2(0.0, 0.0), point2(1.0, 0.0), point2(1.0, 1.0),
    point2(0.0, 1.0)};

int indices[6] = {0, 1, 2, 0, 2, 3}; // 2 triangles

/* vao stuff left out*/
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
    GL_STATIC_DRAW);

// Generate a buffer for the indices
    GLuint elementbuffer;
    glGenBuffers(1, &elementbuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices),
    &indices[0], GL_STATIC_DRAW);
myDisplay() {
    ...
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);

    // Draw the triangles!
    glDrawElements(
        GL_TRIANGLES, // kind of primitive to render
        NumIndices, // # indices to render
        GL_UNSIGNED_INT, // index type
        (void*)0 // Offset into the array buffer
    );
See 3D_ModelingDemo_IB