Lecture 2
Triangle Mesh Representation and its Data Structure
Overview

- To study the storage and data structure of the widely used triangle mesh in representing 3D surfaces
  - triangle meshes can adaptively approximate the continuous surfaces using a finite number of vertices and triangles
  - a piecewise linear representation
Storage of Triangle Meshes: Polygon Soup

**Polygon (Triangle) Soup**: A collection of unorganized triangles
- Example: the Stereolithography (STL) Format (widely used in computer-aided design/manufacturing software) is a type of polygon soup

**Storage Cost:**
- If using $x$ (e.g., 32-bits or 4 bytes) bits to represent a vertex coordinate (float)
- Then each triangle needs $3 \times 3 \times 4 = 36$ bytes
- A mesh with $n$ triangles needs $36n$ bytes

**Pros and Cons:**
- Efficient rendering
- No connectivity info stored
- Inefficient for many geometric computing: e.g. traversing local adjacency information
- Vertex positions replicated as many times as the degree of the vertices
Storage of Triangle Meshes: Indexed Vertex Tables

Using an indexed vertex table, then a face table
- Examples: OFF, OBJ, VRML, M formats

Storage Cost:
- If using \( x \) (e.g., 32-bits or 4 bytes) bits to represent a vertex coordinate (float), and \( x \) bits to represent a vertex index (int)
- Then each vertex needs \( 3 \times 4 = 12 \) bytes, and each triangle needs \( 3 \times 4 = 12 \) bytes
- A mesh with \( n \) triangles needs \( 12n + n/2 \times 12 = 18n \) bytes

Pros and Cons:
- ✔ Efficient storage and rendering
- ❌ Inefficient for local traversal
Mesh Representation in Memory for Efficient Computation in CG Tasks

What operators do we usually need?

- Access to individual elements (vertices, edges, and faces): enumeration of all elements
- Local traversal, e.g.:
  - What are the edges in a given face;
  - What are the vertices in a given face or edge;
  - What are the one-ring primitives of a geometric primitive
    - E.g. incident faces/edges/vertices of a given vertex
    - E.g. incident faces of a given edge
- Example: Modifying the last page’s data structure for local traversal
  - For each face: store references to its 3 vertices + neighboring triangle
  - For each vertex: store 3 coordinates + a reference to its neighboring triangle
  - Used in CGAL for representing 2D Triangulation, 32 bytes / triangle
  - CGAL = an open-source computational geometry algorithm library
    - Google “CGAL”
  - Limitations:
    - But enumerating the one-ring vertices of a vertex is not easy
    - Not easily extendable to general/mixed polygonal meshes
Edge-based Data Structure

- A more generally used data structure, since the connectivity is a graph, directly relates to the mesh edges
- Many well known methods: winged-edge [Baumgart 72], quad-edge [Guibas and Stolfi 85], and variants [O’Rourke 94]
- *An example: Winged-edge structure
  - Each edge: stores references to its endpoint vertices + two incident faces + next and previous edge within the left and right faces
  - Each vertex: stores a reference to one of its incident edges
  - Each face: stores a reference to one of its incident edges
  - **Storage Cost**: A mesh with $n$ faces needs $60n$ bytes
- Limitations: Still not easy for some local traversal
  - e.g. to traverse the one-ring of a vertex, how do you know if it is the first or second vertex of an edge?

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>vertex</td>
</tr>
<tr>
<td>Position</td>
<td></td>
</tr>
<tr>
<td>EdgeRef</td>
<td>edge</td>
</tr>
<tr>
<td>Face</td>
<td></td>
</tr>
<tr>
<td>EdgeRef</td>
<td>next</td>
</tr>
<tr>
<td></td>
<td>prev</td>
</tr>
</tbody>
</table>
Half-Edge Data Structure

• (What?) A common way to represent triangular mesh for geometric processing
  • We first focus on triangle-mesh, (it works for general polygonal mesh).
  • 3D analogy: half-face data structure for tetrahedral mesh

• (Why?) Effective for maintaining incidence information of vertices
  • Efficient local traversal
  • Relatively low spatial cost
  • Supporting dynamic local updates/manipulations (edge collapse, vertex split, etc.)

• (Resources?) Codes are provided. After this class, please go through them carefully, we will work on it during the whole semester.
Consider each edge by splitting it into two **halfedges**

- Primitives: Face, Edge, **Halfedge**, Vertex

- Store all adjacency information between primitives on **halfedges**

  - Each edge has 2 **halfedges** (the boundary edge has only 1)
Half-Edge Data Structure (cont.)

- Halfedges are oriented consistently in counterclockwise order around each face
- Each halfedge designates a unique corner on each face (can be used to store texture coordinates, later in texture mapping)

  - On each halfedge, we store:
    - the vertex it points to (its target);
    - the face this halfedge locates;
    - the next halfedge on the face;
    - the previous halfedge on the face;
    - *(1)* its twin halfedge;

  - For each vertex: store one of its incident incoming halfedges
  - For each face: store one of its halfedges
  - *(2)* for each edge: store its two halfedges

- *(1)* and *(2)*: keep either one and we can get the other easily

- Storage Costs:
  - A mesh with \( n \) triangles needs ?? Bytes
  - Hint: \# of halfedges \( H \) is about 6 times of \( V \)
Half-Edge Structure Implementation

Read through the provided source codes for the implementation of halfedge data structure:

- Check Halfedge.h, each halfedge class stores:
  - → target(): the target vertex;
  - → face(): adjacent face;
  - → next(): the next halfedge on the face;
  - → prev(): the previous halfedge in the face;

And you can use some other implementation:
- → twin(): its twin halfedge;
- → source(): the source vertex;

Check Edge.h, Vertex.h, Face.h, and finally Mesh.h
Half-Edge Data Structure (example)

1). In Mesh.h, four containers used to store primitives:

<table>
<thead>
<tr>
<th>Container Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Vertex Container*</td>
<td>v1 … v6</td>
</tr>
<tr>
<td>The Half-Edge Container</td>
<td>[v1,v2], [v2,v3], [v3,v1], [v1,v3], [v3,v4], …</td>
</tr>
<tr>
<td>The Edge Container</td>
<td>[v1,v3], [v1,v2], [v2,v3], [v1,v4], [v3,v4], …</td>
</tr>
</tbody>
</table>

[v₁, v₂] means: a halfedge from v₁ to v₂

Halfedges: [v₁, v₂] ≠ [v₂, v₁]

The orientation of all the halfedges should be consistent: CounterClockWise (CCW) in our configuration

Note*: the container could be array, list, binary search tree… (it depends, our sample codes used list)
Half-Edge Data Structure (example)

1). Containers store primitives:

<table>
<thead>
<tr>
<th>Container Type</th>
<th>Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Vertex Container*</td>
<td>v1 … v6</td>
</tr>
<tr>
<td>The Half-Edge Container</td>
<td>[v1,v2], [v2,v3], [v3,v1], [v1,v3], [v3,v4], …</td>
</tr>
<tr>
<td>The Edge Container</td>
<td>[v1,v3], [v1,v2], [v2,v3], [v1,v4], [v3,v4], …</td>
</tr>
</tbody>
</table>

2). Relationship between primitives:
Using Half-Edge Data Structure

1. How to check whether a vertex/edge/face is on the boundary?
2. How to track the boundary?
3. How to find your one-ring neighbor?
4. How to do subdivision/simplification…?
Using Half-Edge Data Structure  
– Local Rotations

- **Rotation Operations** defined on **halfedge**:
  1. `clw_rotate_about_target()`: e.g. `[v4, v3]` to `[v6, v3]`
     ```
     \text{he} \rightarrow \text{next()} \rightarrow \text{twin()}
     ```
  2. `clw_rotate_about_source()`: e.g. `[v3, v2]` to `[v3, v1]`
     ```
     \text{he} \rightarrow \text{twin()} \rightarrow \text{next()}
     ```
  3. `ccw_rotate_about_target()`: e.g. `[v6, v3]` to `[v4, v3]`
     ```
     \text{he} \rightarrow \text{twin()} \rightarrow \text{prev()}
     ```
  4. `ccw_rotate_about_source()`: e.g. `[v3, v1]` to `[v3, v2]`
     ```
     \text{he} \rightarrow \text{prev()} \rightarrow \text{twin()}
     ```

- **Rotation Operations** defined on **boundary vertices**:
  1. `most_clw_in_halfedge()`: e.g. for v4, it is `[v2, v4]`
     ```
     \text{Let } \text{he} = v \rightarrow \text{he()}, \text{ then keep doing clw rotation about its target}
     ```
  2. `most_ccw_in_halfedge()`: e.g. for v4, it is `[v6, v4]`
     ```
     \text{Let } \text{he} = v \rightarrow \text{he()}, \text{ then keep doing ccw rotation about its target}
     ```
  3. `most_clw_out_halfedge()`: e.g. for v4, it is `[v4, v1]`
  4. `most_ccw_out_halfedge()`: e.g. for v4, it is `[v4, v1]`

- **What about interior vertices?**
  - Not well defined. Return any in/out halfedge
Using Half-Edge Data Structure

Mesh subdivision/simplification…?

Subdivision (refining)

Simplification (coarsening)
Half-Edge Data Structure (cont.)

1) Read “iterators.h” to see how you can do local traversal
2) Read “mesh.h”, to see how you can get access to primitives
3) *Go through “mesh→read()” method, to see how the halfedge data structure is constructed from indexed vertex-face table.
Some 3D Models in Polygonal Meshes

- “m” format
  - Some models (with “.mesh” or “.m” as extension)
  - A small openGL viewer “MViewer.exe” (you can drag your downloaded “.m” file into it directly)
- “obj” format
  - Two models with .obj extension
  - You will be asked to write a program similar to “Mviewer” in homework 1 and 2

- Many 3D shapes/data available online (but in various formats):
  - Aim@Shape Repository: [http://shapes.aim-at-shape.net/index.php](http://shapes.aim-at-shape.net/index.php)
  - Google 3D warehouse