Mesh Data Structures

Adam Mally
CIS 277 Spring 2015
University of Pennsylvania
Meshes You’ve Made

- Meshes built with Vertex Buffer Objects are just vertex positions arranged into faces
  - Given a set of vertices, build faces by grouping the vertices
- Advantage: Very little storage space, no element duplication
- Disadvantage: No adjacency information
  - i.e. there is no concept of two adjacent faces, only vertices
Polygon Mesh Implementation

- Generally speaking, we build a linked structure to explicitly connect faces, edges, and vertices
  - Advantages: fast, efficient, can support arbitrary polygons
  - Disadvantages: Somewhat complex to implement, must maintain pointers to the necessary neighboring components
Well-Formed Surfaces

- When you have a mesh structure, you generally want its structure to be “clean”
- Components must intersect “properly”
  - Faces are disjoint, OR share either a vertex or the edge between two vertices
  - Each edge is incident to exactly two vertices
  - Each edge is incident to at most two faces
Well-Formed Surfaces

- Local topology must also be “proper”
  - The “neighborhood” of a vertex should permit stretching and bending, but not tearing
- Global topology should be connected, closed, and bounded
Simple Data Structures

- List of polygon faces
- List of edges
- Vertices and index-based faces
List of Polygons

- Store a list of polygon faces, each containing a set of explicit vertex coordinates
- No explicit edges
- No explicit face adjacency
  - You might figure out what faces are adjacent by checking for pairs of shared vertices

A = <0,1>, <3,3>, <2,0>
B = <2,0>, <3,3>, <4, 0.5>
C = <4, 0.5>, <3, 3>, <5, 2.5>
List of Edges

- Store a list of vertex pairs, where each pair defines an edge
- Again, no explicit faces or adjacency
  - Two vertices at the same coordinates are discrete
Index-based faces

- Like we discussed previously, this is the data structure you used to render your geometry with vertex buffer objects.
- As with the previous two data structures, there is no adjacency information.

Positions Buffer:  
- \(<0,1,0,1>\)
- \(<1,1,0,1>\)
- \(<1,0,0,1>\)
- \(<0,0,0,1>\)

Indices Buffer:  
- 0
- 1
- 2
- 3

V0 \(<0,1,0,1>\)  
V1 \(<1,1,0,1>\)  
V2 \(<1,0,0,1>\)  
V3 \(<0,0,0,1>\)

T1 = \([V0, V1, V2]\)
T2 = \([V0, V2, V3]\)
Computational Complexity

● We want a mesh data structure that helps us do the following
  ○ Easily add/remove/modify elements in 3D
  ○ Have a low access time when finding elements
    ■ Usually linear time
    ■ Avoid searching for things like neighbors
  ○ Have constant-size storage (i.e. no usage of std::vectors)
  ○ Implicitly prevent “bad” mesh structures
● The data structures we’ve already covered don’t do all of this
● So, in addition to geometric data, let’s also store topological information like adjacency and connectivity
Naïve Adjacency

- Each element has a list of pointers to ALL adjacent elements
  - Queries depend on the local complexity of the mesh
  - Each element (face/edge/vertex) does not have a fixed size!
  - Slow to construct, tedious to maintain all the pointers during operations on the mesh
**Half-Edge Data Structure**

- Composed of vertices, faces, and half-edges
- Half-edges are **directed** edges that form a ring around a particular face
- A half-edge stores the following information:
  - The **face** to its left
  - The **next** half-edge in the ring
  - The **symmetric** half-edge on the face adjacent to this half-edge’s face
  - The **vertex** between this half-edge and the next half-edge
Incomplete Half-Edge Meshes

- If an edge lies at a boundary (i.e. it only touches one face), both half-edges are still needed
  - The outer half-edge just has a NULL face pointer
- Take note that the external boundary of this mesh is still linked in a ring
More Half-Edge Mesh Attributes

- A face stores a single pointer to any one of the half-edges that loops around it.
- A vertex stores a single pointer to an arbitrary half-edge that points to it.
- Combined with the many pointers stored in a half-edge, we have a data structure that we can traverse starting from any arbitrary component!
Half-Edge Traversal: Clicker

Generally speaking, what sort of loop could we best use to count the number of edges surrounding a face?

1. FOR
2. FOR EACH
3. WHILE
Half-Edge Traversal

- Given some half-edge, how do we find the other vertex that bounds the edge?
Given some face, how do we find all adjacent faces?
Half-Edge Advantages

- Fixed size of data structure elements
- Data:
  - Geometric information stored at vertices
  - Attribute information (e.g. color) stored in any relevant component
  - Topological information in half-edges only!
- Structure enforces “proper” topology (i.e. you can’t have Mobius strips)
- Time complexity
  - Linear for all local information (e.g. gathering lists of faces, edges, or vertices)
  - Independent of the overall mesh complexity
Half-Edge Operations: Split Edge

- **Input:** HE1 or HE2
  - They’re each one half of the same full edge, so the half-edge with which you begin doesn’t really matter
- **Goal:** Insert a new vertex between V1 and V2
Half-Edge Operations: Split Edge

- Step 1: Create the new vertex $V_3$
  - $V_3$’s position is the average of $V_1$’s and $V_2$’s
Half-Edge Operations: Split Edge

- Step 1: Create the new vertex V3
- Step 2: Create the two new half-edges HE1B and HE2B needed to surround V3
  - Step 2A: Make sure that HE1B points to V1 and that HE2B points to V2
    - Since HE1 and HE2 already point to these vertices, this is very easy
  - Step 2B: Also set the correct face pointers for HE1B and HE2B
Half-Edge Operations: Split Edge

- **Step 1**: Create the new vertex $V_3$
- **Step 2**: Create the two new half-edges $HE_{1B}$ and $HE_{2B}$ needed to surround $V_3$
  - **Step 2A**: Make sure that $HE_{1B}$ points to $V_1$ and that $HE_{2B}$ points to $V_2$
  - **Step 2B**: Also set the correct face pointers for $HE_{1B}$ and $HE_{2B}$
- **Step 3**: Adjust the sym, next, and vert pointers of $HE_1$, $HE_2$, $HE_{1B}$, and $HE_{2B}$ so the data structure flows as it did before
Half-Edge Operations: Split Edge

- Step 3: Adjust the sym, next, and vert pointers of HE1, HE2, HE1B, and HE2B so the data structure flows as it did before
- Explicitly:
  - HE1B.next = HE1.next
  - HE2B.next = HE2.next
Half-Edge Operations: Split Edge

- Step 3: Adjust the `sym`, `next`, and `vert` pointers of `HE1`, `HE2`, `HE1B`, and `HE2B` so the data structure flows as it did before.

- Explicitly:
  - `HE1B.next = HE1.next`
  - `HE2B.next = HE2.next`
  - `HE1.next = HE1B`
  - `HE2.next = HE2B`
Half-Edge Operations: Split Edge

- Step 3: Adjust the `sym`, `next`, and `vert` pointers of HE1, HE2, HE1B, and HE2B so the data structure flows as it did before
- Explicitly:
  - HE1B.next = HE1.next
  - HE2B.next = HE2.next
  - HE1.next = HE1B
  - HE2.next = HE2B
  - HE1.vert = HE2.vert = V3
Half-Edge Operations: Split Edge

- Step 3: Adjust the `sym`, `next`, and `vert` pointers of `HE1`, `HE2`, `HE1B`, and `HE2B` so the data structure flows as it did before.
- Explicitly:
  - `HE1B.next = HE1.next`
  - `HE2B.next = HE2.next`
  - `HE1.next = HE1B`
  - `HE2.next = HE2B`
  - `HE1.vert = HE2.vert = V3`
  - `HE1.sym = HE2B, HE2B.sym = HE1`
  - `HE2.sym = HE1B, HE1B.sym = HE2`
Half-Edge Operations: Triangulate

- Input: some quadrangle “FACE1”
- Goal: Divide FACE1 into two triangles
  - Note: HE_0 is the arbitrary half-edge to which FACE1 points
Half-Edge Operations: Triangulate

- Step 1: Create two new half-edges HE_A and HE_B
  - HE_A points to HE_0.vert
  - HE_B points to HE_0.next.next.vert
  - HE_A and HE_B have each other for syms
Half-Edge Operations: Triangulate

- Step 1: Create two new half-edges HE_A and HE_B
- Step 2: Create a second face FACE2
  - HE_A, HE_0.next, and HE_0.next.next now all point to FACE2
  - HE_B points to FACE1
  - FACE2’s arbitrary half-edge pointer can be HE_A.
Half-Edge Operations: Triangulate

- Step 1: Create two new half-edges HE_A and HE_B
- Step 2: Create a second face FACE2
- Step 3: Fix up the next pointers for our half-edges
  - HE_B.next = HE_0.next.next.next
  - HE_0.next.next.next = HE_A
  - HE_A.next = HE_0.next
  - HE_0.next = HE_B