Meshes and Manifolds

Computer Graphics
CMU 15-462/15-662
Last time: overview of geometry

- Many types of geometry in nature
- Demand sophisticated representations
- Two major categories:
  - IMPLICIT - “tests” if a point is in shape
  - EXPLICIT - directly “lists” points
- Lots of representations for both
- Today:
  - what is a surface, anyway?
  - nuts & bolts of polygon meshes
  - geometry processing / resampling
Manifold Assumption

- Today we’re going to introduce the idea of manifold geometry.
- Can be hard to understand motivation at first!
- So first, let’s revisit a more familiar example...
Bitmap Images, Revisited

To encode images, we used a *regular grid* of pixels:
But images are not fundamentally made of little squares:

Goyō Hashiguchi, *Kamisuki* (ca 1920)
So why did we choose a square grid?

...rather than dozens of possible alternatives?
Regular grids make life easy

- One reason: SIMPLICITY / EFFICIENCY
  - E.g., always have four neighbors
  - Easy to index, easy to filter...
  - Storage is just a list of numbers

- Another reason: GENERALITY
  - Can encode basically any image

- Are regular grids always the best choice for bitmap images?
  - No! E.g., suffer from anisotropy, don’t capture edges, ...
  - But more often than not are a pretty good choice

- Will see a similar story with geometry...
So, how should we encode surfaces?
Smooth Surfaces

- Intuitively, a *surface* is the boundary or "shell" of an object
  (Think about the candy shell, not the chocolate.)
- Surfaces are *manifold*:
  - If you zoom in far enough, can draw a regular coordinate grid
  - E.g., the Earth from space vs. from the ground
Isn’t every shape manifold?

- No, for instance:

Can’t draw ordinary 2D grid at center, no matter how close we get.
Examples—Manifold vs. Nonmanifold

Which of these shapes are manifold?

❌ ❌ ✔ ✔ ❌ ❌ ✔ ✔ ✔
A manifold polygon mesh has fans, not fins

- For polygonal surfaces just two easy conditions to check:
  1. Every edge is contained in only two polygons (no “fins”)
  2. The polygons containing each vertex make a single “fan”
What about boundary?

- The boundary is where the surface “ends.”
- E.g., waist & ankles on a pair of pants.
- Locally, looks like a \textit{half} disk
- Globally, each boundary forms a loop

- Polygon mesh:
  - one polygon per boundary edge
  - boundary vertex looks like “pacman”
Ok, but why is the manifold assumption *useful*?
Keep it Simple!

Same motivation as for images:
- make some assumptions about our geometry to keep data structures/algorithms simple and efficient
- in many common cases, doesn’t fundamentally limit what we can do with geometry

\[
\begin{array}{ccc}
(i,j-1) & (i,j) & (i+1,j) \\
(i-1,j) & (i,j) & (i+1,j) \\
(i,j+1) & & \\
\end{array}
\]
How do we actually encode all this data?
Warm up: storing numbers

Q: What data structures can we use to store a list of numbers?

One idea: use an *array* (constant time lookup, coherent access)

```
1.7  2.9  0.3  7.5  9.2  4.8  6.0  0.1
```

Alternative: use a linked list (linear lookup, incoherent access)

Q: Why bother with the linked list?

A: For one, we can easily insert numbers wherever we like...
Polygon Soup

- Most basic idea:
  - For each triangle, just store three coordinates
  - No other information about connectivity
  - Not much different from point cloud! ("Triangle cloud?")

- Pros:
  - Really stupidly simple

- Cons:
  - Redundant storage
  - Hard to do much beyond simply drawing the mesh on screen
  - Need spatial data structures (later) to find neighbors
Adjacency List (Array-like)

- Store triples of coordinates (x,y,z), tuples of indices
- E.g., tetrahedron:

<table>
<thead>
<tr>
<th>VERTICES</th>
<th>POLYGONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>x  y  z</td>
<td>i  j  k</td>
</tr>
<tr>
<td>0: -1 -1 -1</td>
<td>0  2  1</td>
</tr>
<tr>
<td>1:  1 -1  1</td>
<td>0  3  2</td>
</tr>
<tr>
<td>2:  1  1 -1</td>
<td>3  0  1</td>
</tr>
<tr>
<td>3: -1  1  1</td>
<td>3  1  2</td>
</tr>
</tbody>
</table>

Q: How do we find all the polygons touching vertex 2?

Ok, now consider a more complicated mesh:

Very expensive to find the neighboring polygons! (What’s the cost?)
Incidence Matrices

- If we want to know who our neighbors are, why not just store a list of neighbors?
- Can encode all neighbor information via *incidence matrices*
- E.g., tetrahedron:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>v0</th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>e2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>e3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>e4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>e5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge</th>
<th>e0</th>
<th>e1</th>
<th>e2</th>
<th>e3</th>
<th>e4</th>
<th>e5</th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>f1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VERTEX ↔ EDGE</th>
<th>FACE ↔ EDGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0 v1 v2 v3</td>
<td>e0 e1 e2 e3 e4 e5</td>
</tr>
<tr>
<td>e0 1 1 0 0</td>
<td>f0 1 0 0 1 0 1</td>
</tr>
<tr>
<td>e1 0 1 1 0</td>
<td>f1 0 1 0 0 1 1</td>
</tr>
<tr>
<td>e2 1 0 1 0</td>
<td>f2 1 1 1 0 0 0</td>
</tr>
<tr>
<td>e3 1 0 0 1</td>
<td>f3 0 0 1 1 1 0</td>
</tr>
<tr>
<td>e4 0 0 1 1</td>
<td></td>
</tr>
<tr>
<td>e5 0 1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

- 1 means “touches”; 0 means “does not touch”
- Instead of storing lots of 0’s, use *sparse matrices*
- Still large storage cost, but finding neighbors is now O(1)
- Hard to change connectivity, since we used fixed indices
- Bonus feature: mesh does not have to be manifold
Aside: Sparse Matrix Data Structures

- Ok, but how do we actually store a “sparse matrix”?
- Lots of possible data structures:
  - **Associative array from (row, column) to value**
    - easy to lookup/set entries, fast (e.g., hash table)
    - harder to do matrix operations (e.g., multiplication)
  - **Array of linked lists (one per row)**
    - conceptually simple
    - slow access time, incoherent memory access
  - **Compressed column format**—pack entries in list
    - hard to add/modify entries
    - fast for actual matrix operations
- In practice: often build up entries using an “easier” data structure, convert to compressed format for computation
Halfedge Data Structure (Linked-list-like)

- Store *some* information about neighbors
- Don’t need an exhaustive list; just a few key pointers
- Key idea: two *halfedges* act as “glue” between mesh elements:

```
struct Halfedge {
    Halfedge* twin;
    Halfedge* next;
    Vertex* vertex;
    Edge* edge;
    Face* face;
};
```

```
struct Vertex {
    Halfedge* halfedge;
};
```

```
struct Edge {
    Halfedge* halfedge;
};
```

```
struct Face {
    Halfedge* halfedge;
};
```

- Each vertex, edge face points to just *one* of its halfedges.
Halfedge makes mesh traversal easy

- Use “twin” and “next” pointers to move around mesh
- Use “vertex”, “edge”, and “face” pointers to grab element
- Example: visit all vertices of a face:
  ```c
  Halfedge* h = f->halfedge;
  do {
    h = h->next;
    // do something w/ h->vertex
  } while( h != f->halfedge );
  ```
- Example: visit all neighbors of a vertex:
  ```c
  Halfedge* h = v->halfedge;
  do {
    h = h->twin->next;
  } while( h != v->halfedge );
  ```
- Note: only makes sense if mesh is manifold!
Halfedge connectivity is *always* manifold

- Consider simplified halfedge data structure
- Require only “common-sense” conditions

```c
struct Halfedge {
    Halfedge *next, *twin;
};
```

- Keep following `next`, and you’ll get faces.
- Keep following `twin` and you’ll get edges.
- Keep following `next->twin` and you’ll get vertices.

Q: Why, therefore, is it impossible to encode the red figures?
Recall manifold conditions (fans not fins):
- every edge contained in two faces
- every vertex contained in one fan

These conditions say nothing about vertex positions! Just connectivity

Hence, can have perfectly good (manifold) connectivity, even if geometry is awful

In fact, sometimes you can have perfectly good manifold connectivity for which any vertex positions give “bad” geometry!

Can lead to confusion when debugging: mesh looks “bad”, even though connectivity is fine
Halfedge meshes are easy to edit

- Remember key feature of linked list: insert/delete elements
- Same story with halfedge mesh (“linked list on steroids”)
- E.g., for triangle meshes, several atomic operations:
  
  ![Diagram of halfedge mesh operations](image)

  - **flip**
  - **split**
  - **collapse**

- Must be careful to preserve manifoldness!
Edge Flip (Triangles)

- Triangles (a,b,c), (b,d,c) become (a,d,c), (a,b,d):

```
\begin{figure}[h]
\centering
\begin{tikzpicture}
    \draw[fill=blue!30] (0,0) -- (2,1) -- (0,2) -- (-2,1) -- cycle;
    \draw[->] (0,0) -- (2,1);
    \draw[->] (0,0) -- (0,2);
    \draw[->] (0,0) -- (-2,1);
    \draw[->] (0,0) -- (0,-1);
    \node at (0,0) [left] {a};
    \node at (2,1) [above] {c};
    \node at (0,2) [left] {b};
    \node at (-2,1) [below] {d};
    \node at (0,0) [left] {a};
    \node at (2,1) [above] {c};
    \node at (0,2) [left] {b};
    \node at (-2,1) [below] {d};
    \end{tikzpicture}
\end{figure}
```

- Long list of pointer reassignments (`edge->halfedge = ...`)
- However, no elements created/destroyed.
- Q: What happens if we flip twice?
- Challenge: can you implement edge flip such that pointers are unchanged after two flips?
Edge Split (Triangles)

- Insert midpoint $m$ of edge $(c,b)$, connect to get four triangles:

- This time, have to add new elements.
- Lots of pointer reassignments.
- Q: Can we “reverse” this operation?
Edge Collapse (Triangles)

- Replace edge \((b, c)\) with a single vertex \(m\):
- Now have to delete elements.
- Still lots of pointer assignments!
- Q: How would we implement this with an adjacency list?
- Any other good way to do it? (E.g., different data structure?)
Alternatives to Halfedge

- Many very similar data structures:
  - winged edge
  - corner table
  - quadedge
  - ...
- Each stores local neighborhood information
- Similar tradeoffs relative to simple polygon list:
  - **CONS**: additional storage, incoherent memory access
  - **PROS**: better access time for individual elements, intuitive traversal of local neighborhoods
- With some thought*, can design halfedge-type data structures with coherent data storage, support for non manifold connectivity, etc.

*see for instance [http://geometry-central.net/](http://geometry-central.net/)
## Comparison of Polygon Mesh Data Structures

<table>
<thead>
<tr>
<th></th>
<th>Adjacency List</th>
<th>Incidence Matrices</th>
<th>Halfedge Mesh</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant-time neighborhood access?</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>easy to add/remove mesh elements?</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>nonmanifold geometry?</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>

**Conclusion:** pick the right data structure for the job!
Ok, but what can we actually do with our fancy new data structures?
Subdivision Modeling
Subdivision Modeling

- Common modeling paradigm in modern 3D tools:
  - Coarse “control cage”
  - Perform local operations to control/edit shape
  - Global subdivision process determines final surface
Subdivision Modeling—Local Operations

- For general polygon meshes, we can dream up lots of local mesh operations that might be useful for modeling:

  ...and many, many more!
Next Time: Digital Geometry Processing

- Extend traditional digital signal processing (audio, video, etc.) to deal with geometric signals:
  - upsampling / downsampling / resampling / filtering ...
  - aliasing (reconstructed surface gives “false impression”)