Introduction to Geometry

Computer Graphics
CMU 15-462/15-662
Assignment 2

- Start building up “Scotty3D”; first part is 3D modeling
3D Modeling

- Don’t just make great software... make great art! :-)

(This mesh was created in Scotty3D in about 5 minutes... you can do much better!)
Increasing the complexity of our models

Transformations

Geometry

Materials, lighting, ...
Q: What is geometry?
A: Geometry is the study of two-column proofs.

Ceci n'est pas géométrie.

See: Paul Lockhart, “A Mathematician's Lament”
What is geometry?

“Earth”  “measure”

**ge•om•et•ry** /jēˈämətrē/ *n.*
1. The study of shapes, sizes, patterns, and positions.
2. The study of spaces where some quantity (lengths, angles, etc.) can be *measured*.

Plato: “...the earth is in appearance like one of those balls which have leather coverings in twelve pieces...”
How can we describe geometry?

**Implicit**

\[ x^2 + y^2 = 1 \]

**Linguistic**

"unit circle"

**Explicit**

\[(\cos \theta, \sin \theta)\]

**Tomographic**

(constant density)

**Dynamic**

\[ \frac{d^2}{dt^2} x = -x \]

**Symmetric**

rotate

**Discrete**

\[ n \to \infty \]
Given all these options, what’s the best way to encode geometry on a computer?
Examples of geometry
Examples of geometry
Examples of geometry
Examples of geometry
Examples of geometry
Examples of geometry
Examples of geometry
Examples of geometry
It’s a Jungle Out There!
No one “best” choice—geometry is hard!

“I hate meshes.
I cannot believe how hard this is.
Geometry is hard.”

—David Baraff
Senior Research Scientist
Pixar Animation Studios
Many ways to digitally encode geometry

- **EXPLICIT**
  - point cloud
  - polygon mesh
  - subdivision, NURBS
  - ...

- **IMPLICIT**
  - level set
  - algebraic surface
  - L-systems
  - ...

- Each choice best suited to a different task/type of geometry
“Implicit” Representations of Geometry

- Points aren’t known directly, but satisfy some relationship
- E.g., unit sphere is all points such that $x^2 + y^2 + z^2 = 1$
- More generally, $f(x, y, z) = 0$
Many implicit representations in graphics

- algebraic surfaces
- constructive solid geometry
- level set methods
- blobby surfaces
- fractals
- ...

(Will see some of these a bit later.)
"Explicit" Representations of Geometry

- All points are given directly
- E.g., points on sphere are $$(\cos(u) \sin(v), \sin(u) \sin(v), \cos(v))$$, for $0 \leq u < 2\pi$ and $0 \leq v \leq \pi$

- More generally: $f : \mathbb{R}^2 \rightarrow \mathbb{R}^3; (u, v) \mapsto (x, y, z)$

- (Might have a bunch of these maps, e.g., one per triangle!)
Many explicit representations in graphics

- triangle meshes
- polygon meshes
- subdivision surfaces
- NURBS
- point clouds
- ...

(Will see some of these a bit later.)
Ok, so we have many ways to represent surfaces.

But what is a surface anyway?
Manifold Assumption

- First, let’s define manifold geometry
- Can be hard to understand motivation at 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?

- ✔
- ✔
- ✔
- ✗
- ✗
Suppose we have a polygon mesh (an explicit representation)
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 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

<table>
<thead>
<tr>
<th></th>
<th>(i,j-1)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(i-1,j)</td>
<td>(i,j)</td>
<td>(i+1,j)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(i,j+1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Let’s talk about how to 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)

```
1.7  0.3  7.5  9.2  6.0  0.1
```

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></th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>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 ↔ EDGE</th>
<th>EDGE ↔ FACE</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
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 Edge
{
    Halfedge* halfedge;
};
```

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

```
struct Vertex
{
    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:
  ```cpp
  Halfedge* h = f->halfedge;
  do {
    h = h->next;
    // do something w/ h->vertex
  }
  while( h != f->halfedge );

- Example: visit all neighbors of a vertex:
  ```cpp
  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;
};
```

twin->twin == this
twin != this
every he is someone’s “next”

- 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?
Connectivity vs. Geometry

- 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

![non manifold connectivity?](image)

...or just a really skinny triangle?

![same connectivity, random vertex positions](image)
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:

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

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

- Triangles \((a,b,c), (b,d,c)\) become \((a,d,c), (a,b,d)\):  
  ![Diagram showing edge flip](image)
  
  - Long list of pointer reassignments \((\text{edge}\rightarrow\text{halfedge} = \ldots)\)
  - 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 \(m\):

  ![Diagram showing edge collapse](image)

  - 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

- 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!
Geometry Processing

- reconstruction
- filtering
- remeshing
- shape analysis
- parameterization
- compression
Geometry Processing: Upsampling

- Increase resolution via interpolation
- Images: e.g., bilinear, bicubic interpolation
- Polygon meshes:
  - subdivision
  - bilateral upsampling
  - …
Geometry Processing: Downsampling

- Decrease resolution; try to preserve shape/appearance
- Images: nearest-neighbor, bilinear, bicubic interpolation
- Point clouds: subsampling (just take fewer points!)
- Polygon meshes:
  - iterative decimation, variational shape approximation, ...
Geometry Processing: Resampling

- Modify sample distribution to improve quality
- Images: not an issue! (Pixels always stored on a regular grid)
- Meshes: shape of polygons is extremely important!
  - different notion of “quality” depending on task
  - e.g., visualization vs. solving equations
Geometry Processing: Filtering

- Remove noise, or emphasize important features (e.g., edges)
- Images: blurring, bilateral filter, edge detection, ...

- Polygon meshes:
  - curvature flow
  - bilateral filter
  - spectral filter
Geometry Processing: Compression

- Reduce storage size by eliminating redundant data/approximating unimportant data

- Images:
  - run-length, Huffman coding - lossless
  - cosine/wavelet (JPEG/MPEG) - lossy

- Polygon meshes:
  - compress geometry and connectivity
  - many techniques (lossy & lossless)
Geometry Processing: Shape Analysis

- Identify/understand important semantic features
- Images: computer vision, segmentation, face detection, ...
- Polygon meshes:
  - segmentation, correspondence, symmetry detection, ...
Remeshing is resampling

- Remember our discussion of aliasing
- Bad sampling makes signal appear different than it really is
- E.g., undersampled curve looks flat
- Geometry is no different!
  - undersampling destroys features
  - oversampling bad for performance
What makes a “good” mesh?

- One idea: good approximation of original shape!
- Keep only elements that contribute information about shape
- Add additional information where, e.g., curvature is large
Approximation of position is not enough!

- Just because the vertices of a mesh are close to the surface it approximates does not mean it’s a good approximation!
- Can still have wrong appearance, wrong area, wrong...
- Need to consider other factors*, e.g., close approximation of surface normals

What else makes a “good” triangle mesh?

- Another rule of thumb: triangle

  - E.g., all angles close to 60 degrees
  - More sophisticated condition: Delaunay (empty circumcircles)
    - often helps with numerical accuracy/stability
    - coincides with shockingly many other desirable properties
      (maximizes minimum angle, provides smoothest interpolation, guarantees maximum principle…)
  - Tradeoffs w/ good geometric approximation*
    - e.g., long & skinny might be “more efficient”

*see Shewchuk, “What is a Good Linear Element”
What else constitutes a “good” mesh?

- Another rule of thumb: regular vertex degree
- Degree 6 for triangle mesh, 4 for quad mesh

Why? Better polygon shape; more regular computation; smoother subdivision:

Fact: in general, can’t have regular vertex degree everywhere!
Next class sessions

- Subdivision + quadric error
- Geometric queries
- Many different ways to represent geometry (a late intro)

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feb 8</td>
<td>3D Rotations</td>
</tr>
<tr>
<td>Feb 13</td>
<td>Intro to Geometry / Halfedge Data Structure</td>
</tr>
<tr>
<td></td>
<td>Assignment 1.5 DUE</td>
</tr>
<tr>
<td></td>
<td>Assignment 2.0 OUT</td>
</tr>
<tr>
<td>Feb 15</td>
<td>Subdivision and Simplification</td>
</tr>
<tr>
<td>Feb 20</td>
<td>Geometric Queries</td>
</tr>
<tr>
<td></td>
<td>Assignment 2.0 DUE</td>
</tr>
<tr>
<td></td>
<td>Assignment 2.5 OUT</td>
</tr>
<tr>
<td>Feb 22</td>
<td>Midterm Review</td>
</tr>
<tr>
<td>Feb 27</td>
<td>MIDTERM</td>
</tr>
<tr>
<td>Mar 1</td>
<td>Other Geometric Representations</td>
</tr>
<tr>
<td>Mar 6</td>
<td>SPRING BREAK</td>
</tr>
<tr>
<td>Mar 8</td>
<td>SPRING BREAK</td>
</tr>
<tr>
<td>Mar 13</td>
<td>Spatial Data Structures</td>
</tr>
<tr>
<td></td>
<td>Assignment 3.0 OUT</td>
</tr>
<tr>
<td>Mar 15</td>
<td>Color</td>
</tr>
<tr>
<td></td>
<td>Assignment 2.5 DUE</td>
</tr>
</tbody>
</table>