Geometry Processing

Computer Graphics
CMU 15-462/15-662
Last time — Adjacency List (Array-like)

- Store triples of coordinates \((x,y,z)\), tuples of indices

- E.g., tetrahedron:

<table>
<thead>
<tr>
<th>VERTEXES</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:

  ~1 billion polygons

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

Each vertex, edge face points to just one of its halfedges.
## 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?
Geometry Processing: Reconstruction

- Given samples of geometry, reconstruct surface
- What are “samples”? Many possibilities:
  - points, points & normals, ...
  - image pairs / sets (multi-view stereo)
  - line density integrals (MRI/CT scans)
- How do you get a surface? Many techniques:
  - silhouette-based (visual hull)
  - Voronoi-based (e.g., power crust)
  - PDE-based (e.g., Poisson reconstruction)
  - Radon transform / isosurfacing (marching cubes)
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, ...

![Image of a rabbit going through different stages of downsampling process](image)
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, ...

Extrinsic symmetry

Intrinsic symmetry
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!
A2L1 Flip Edge

Given a non-boundary edge, rotate it to the vertices one step counterclockwise from its current endpoints.

\[
\text{flip_edge}(e);
\]

A2L2 Split Edge

Given an edge, add a vertex in the middle and create an edge from the newly-added vertex to the next-in-ccw-order-from-the-edge vertex in the face.

\[
v = \text{split_edge}(e); \quad h = v\rightarrow\text{halfedge};
\]

Split edge does not subdivide boundary faces.

\[
v = \text{split_edge}(e);
\]

CMU 15-462/662
A2.0 Diagrams

A2L3 Collapse Edge

Merge the two endpoints of an edge at the midpoint of the edge. This will combine all the edges to both endpoints into a new vertex (be careful to not have two edges overlapping).

```
v = collapse_edge(e);
```

If collapsing the edge would result in an invalid mesh, you should instead do nothing and return `std::nullopt`. Otherwise returns the newly collapsed vertex.

Examples of edge cases that we will be testing would be

```
\[\begin{array}{c}
\hline
| & | & |
\hline
- & \rightarrow & -
\end{array}\]
```

(figure 8, and any generalizations of this)
What is a valid mesh?

A `Halfedge_Mesh` structure is a sea of pointers. It is surprisingly easy to tangle these pointers into a mess that fails to represent a manifold mesh (or, really, a consistent mesh at all). In order to provide guidance on what constitutes a reasonable setting for a mesh, we define a valid mesh as one for which the following properties hold:

1. The mesh is self-contained. All pointers are to elements in the vertices, edges, faces, or halfedges lists.
2. Edges correspond to twinned halfedges. That is, for every edge $e$, $e\rightarrow\text{halfedge}\rightarrow\text{twin}^n$ is a cycle of exactly two halfedges, and these are exactly the halfedges with $n\rightarrow\text{edge}$ equal to $e$.
3. Faces correspond to cycles of halfedges. That is, for every face $f$, $f\rightarrow\text{halfedge}\rightarrow\text{next}^n$ is a cycle of at least three halfedges, and these are exactly the halfedges with $n\rightarrow\text{face}$ equal to $f$.
4. Vertices correspond to stars of halfedges. That is, for every vertex $v$, $v\rightarrow\text{halfedge}\rightarrow\text{twin}\rightarrow\text{next}^n$ is a cycle of at least two halfedges, and these are exactly the halfedges with $n\rightarrow\text{vertex}$ equal to $v$.
5. Vertices are not orphaned, nor is the surface adjacent to them hourglass-shaped. That is, vertices are adjacent to at least one non-boundary face and at most one boundary face.
6. Edges not orphaned. That is, edges are adjacent to at least one non-boundary face.
7. Faces are simple. That is, faces touch each vertex and edge at most once.

These properties are checked by the `Halfedge_Mesh::validate()` function, which returns `nullopt` if the mesh is valid or an element reference and an explanatory message if something is wrong with the mesh.
Mesh Processing:

“If the surface resulting from an operation can be represented by some valid mesh, then run the operation and produce a valid mesh representing the result.”
A few things to consider —
Remeshing as 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!
Upsampling via Subdivision
Upsampling via Subdivision

- Repeatedly split each element into smaller pieces
- Replace vertex positions with weighted average of neighbors

Main considerations:
- interpolating vs. approximating
- limit surface continuity ($C^1$, $C^2$, ...)
- behavior at irregular vertices

Many options:
- Quad: Catmull-Clark
- Triangle: Loop, Butterfly, Sqrt(3)
Catmull-Clark Subdivision

- **Step 0:** split every polygon (any # of sides) into quadrilaterals:

- **New vertex positions are weighted combination of old ones:**

  **STEP 1: Face coords**
  \[ \frac{1}{n} \sum_{i} p_i \]

  **STEP 2: Edge coords**
  \[ (a + b + c + d) / 4 \]

  **STEP 3: Vertex coords**

  **New vertex coords:**
  \[ \frac{Q + 2R + (n - 3)S}{n} \]

  - **n** – vertex degree
  - **Q** – average of face coords around vertex
  - **R** – average of edge coords around vertex
  - **S** – original vertex position
Catmull-Clark on quad mesh

- Few irregular vertices
  → Smoothly-varying surface normals

- Smooth reflection lines

- Smooth caustics
Catmull-Clark on triangle mesh

many irregular vertices
⇒ erratic surface normals

jagged reflection lines

jagged caustics
Loop Subdivision

- Alternative subdivision scheme for triangle meshes
- Curvature is continuous away from irregular vertices ("$C^2$")

Algorithm:
- Split each triangle into four
- Assign new vertex positions according to weights:

$$n: \text{vertex degree}$$

$$u: \frac{3}{16} \text{ if } n=3, \frac{3}{8n} \text{ otherwise}$$
Loop Subdivision via Edge Operations

- First, split edges of original mesh in any order:

- Next, flip new edges that touch a new & old vertex:

(Don’t forget to update vertex positions!)

Images cribbed from Denis Zorin.
Downsampling
(i.e., what if we want fewer triangles?)
Simplification via Edge Collapse

- One popular scheme: iteratively collapse edges

- Greedy algorithm:
  - assign each edge a cost
  - collapse edge with least cost
  - repeat until target number of elements is reached

- Particularly effective cost function: quadric error metric*

*invented at CMU (Garland & Heckbert 1997)
Quadric Error Metric

- Approximate distance to a collection of triangles
- Q: Distance to plane w/ normal $n$ passing through point $p$?
- A: $\text{dist}(x) = \langle n, x \rangle - \langle n, p \rangle = \langle n, x - p \rangle$
- Quadric error is then sum of squared point-to-plane distances:

\[
Q(x) := \sum_{i=1}^{k} \langle n_i, x - p \rangle^2
\]

\[
Q = \frac{1}{8}
\]

\[
Q = \frac{1}{2}
\]

\[
Q = 1
\]

\[
Q = 0
\]
Quadric Error - Homogeneous Coordinates

- Suppose in coordinates we have
  - a query point \( \mathbf{x} = (x, y, z) \)
  - a normal \( \mathbf{n} = (a, b, c) \)
  - an offset \( d := -\langle \mathbf{n}, \mathbf{p} \rangle \)
- In homogeneous coordinates, let
  - \( \mathbf{u} := (x, y, z, 1) \)
  - \( \mathbf{v} := (a, b, c, d) \)
- Signed distance to plane is then just \( \langle \mathbf{u}, \mathbf{v} \rangle = ax + by + cz + d \)
- Squared distance is \( \langle \mathbf{u}, \mathbf{v} \rangle^2 = \mathbf{u}^T(\mathbf{v}\mathbf{v}^T)\mathbf{u} =: \mathbf{u}^T\mathbf{Ku} \)
- Matrix \( \mathbf{K} = \mathbf{vv}^T \) encodes squared distance to plane

Key idea: sum of matrices \( \mathbf{K} \leftrightarrow \) distance to union of planes

\[
\mathbf{u}^T\mathbf{K}_1\mathbf{u} + \mathbf{u}^T\mathbf{K}_2\mathbf{u} = \mathbf{u}^T(\mathbf{K}_1 + \mathbf{K}_2)\mathbf{u}
\]
Quadric Error of Edge Collapse

- How much does it cost to collapse an edge $e_{ij}$?
- Idea: compute midpoint $m$, measure error $Q(m) = m^T(K_i + K_j)m$
- Error becomes “score” for $e_{ij}$, determining priority

Better idea: find point $x$ that minimizes error!

Ok, but how do we minimize quadric error?
Suppose you have a function $f(x) = ax^2 + bx + c$.

**Q:** What does the graph of this function look like?

**Could also look like this!**

**Q:** How do we find the minimum?

**A:** Find where the function looks “flat” if we zoom in really close.

**I.e., find point $x$ where 1st derivative vanishes:**

$$f'(x) = 0$$

$$2ax + b = 0$$

$$x = -b/2a$$

*(What does $x$ describe for the second function?)*
Minimizing Quadratic Polynomial

- Not much harder to minimize a quadratic polynomial in $n$ variables
- Can always write in terms of a symmetric matrix $A$
- E.g., in 2D: $f(x, y) = ax^2 + bxy + cy^2 + dx + ey + g$

$$ x = \begin{bmatrix} x \\ y \end{bmatrix} \quad A = \begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix} \quad u = \begin{bmatrix} d \\ e \end{bmatrix} $$

$$ f(x, y) = x^T Ax + u^T x + g $$

(will have this same form for any $n$)

**Q:** How do we find a critical point (min/max/saddle)?

**A:** Set derivative to zero!

$$ 2Ax + u = 0 $$

$$ x = -\frac{1}{2} A^{-1} u $$

(compare with our 1D solution)

$$ x = -\frac{b}{2a} $$

(Can you show this is true, at least in 2D?)
**Positive Definite Quadratic Form**

- Just like our 1D parabola, critical point is not always a min!
- **Q:** In 2D, 3D, nD, when do we get a minimum?
- **A:** When matrix $A$ is positive-definite:

$$ x^T A x > 0 \quad \forall \ x $$

- **1D:** Must have $x a x = ax^2 > 0$. In other words: $a$ is positive!
- **2D:** Graph of function looks like a “bowl”:

Positive-definiteness extremely important in computer graphics: means we can find minimizers by solving linear equations. Starting point for many algorithms (geometry processing, simulation, ...)
Minimizing Quadric Error

- Find “best” point for edge collapse by minimizing quadratic form
  \[
  \min_{u \in \mathbb{R}^4} u^T Ku
  \]

- Already know fourth (homogeneous) coordinate for a point is 1
- So, break up our quadratic function into two pieces:

\[
\begin{bmatrix}
  x^T & 1
\end{bmatrix}
\begin{bmatrix}
  B & w \\
  w^T & d^2
\end{bmatrix}
\begin{bmatrix}
  x \\
  1
\end{bmatrix} = x^T B x + 2 w^T x + d^2
\]

- Now we have a quadratic polynomial in the unknown position \( x \in \mathbb{R}^3 \)
- Can minimize as before:

\[
2Bx + 2w = 0 \quad \iff \quad x = -B^{-1}w
\]

Q: Why should \( B \) be positive-definite?
Quadric Error Simplification: Final Algorithm

- Compute $K$ for each triangle (squared distance to plane)
- Set $K_i$ at each vertex to sum of $K$s from incident triangles
- For each edge $e_{ij}$:
  - set $K_{ij} = K_i + K_j$
  - find point $x$ minimizing error, set cost to $K_{ij}(x)$
- Until we reach target number of triangles:
  - collapse edge $e_{ij}$ with smallest cost to optimal point $x$
  - set quadric at new vertex to $K_{ij}$
  - update cost of edges touching new vertex
- More details in assignment writeup!
Quadric Simplification—Flipped Triangles

- Depending on where we put the new vertex, one of the new triangles might be “flipped” (normal points in instead of out):

- Easy solution: for each triangle $ijk$ touching collapsed vertex $i$, consider normals $N_{ijk}$ and $N_{kjl}$ (where $kjl$ is other triangle containing edge $jk$)

- If $\langle N_{ijk}, N_{kjl} \rangle$ is negative, don’t collapse this edge!
What if we’re happy with the number of triangles, but want to improve quality?
How do we make a mesh “more Delaunay”?

- Already have a good tool: edge flips!
- If $\alpha + \beta > \pi$, flip it!

- FACT: in 2D, flipping edges eventually yields Delaunay mesh
- Theory: worst case $O(n^2)$; doesn’t always work for surfaces in 3D
- Practice: simple, effective way to improve mesh quality
Alternatively: how do we improve degree?

- Same tool: edge flips!
- If total deviation from degree-6 gets smaller, flip it!

FACT: average degree approaches 6 as number of elements increases

Iterative edge flipping acts like “discrete diffusion” of degree

No (known) guarantees; works well in practice

Total deviation: \[ |d_i - 6| + |d_j - 6| + |d_k - 6| + |d_l - 6| \]
How do we make a triangles “more round”?

- Delaunay doesn’t guarantee triangles are “round” (angles near 60°)
- Can often improve shape by centering vertices:

  ![Diagram showing triangle smoothing](attachment:triangle_smoothing.png)

- Simple version of technique called “Laplacian smoothing”
- On surface: move only in tangent direction
- How? Remove normal component from update vector
Isotropic Remeshing Algorithm

- Try to make triangles uniform shape & size
- Repeat four steps:
  - Split any edge over 4/3rds mean edge length
  - Collapse any edge less than 4/5ths mean edge length
  - Flip edges to improve vertex degree
  - Center vertices tangentially

Based on: Botsch & Kobbelt, “A Remeshing Approach to Multiresolution Modeling”
What can go wrong when you resample a signal?
Danger of Resampling

Q: What happens if we repeatedly resample an image?

A: Signal quality degrades!
Danger of Resampling

Q: What happens if we repeatedly resample a mesh?

A: Signal also degrades!
But wait: we have the original signal (mesh). Why not just project each new sample point onto the closest point of the original mesh?
Next Time: Geometric Queries

- Q: Given a point, in space, how do we find the closest point on a surface? Are we inside or outside the surface? How do we find intersection of two triangles? Etc.

- Do implicit/explicit representations make such tasks easier?

- What’s the cost of the naïve algorithm, and how do we accelerate such queries for large meshes?

- So many questions!