Triangle meshes
Examples

Spheres

Approximate sphere

Based on slides courtesy of Steve Marschner

Rineau & Yvinec
CGAL manual

Andrzej Barabasz
Examples

Finite element analysis
Examples
A small triangle mesh

12 triangles, 8 vertices
A large mesh

10 million triangles from a high-resolution 3D scan
About a trillion triangles from automatically processed satellite and aerial photography
Triangles

• Defined by three vertices
• Lies in the plane containing those vertices
• Normal of the plane is normal of the triangle
• Conventions (for this class, not everyone agrees):
  – Vertices are counter-clockwise as seen from the “outside” or “front”
  – Surface normal points towards the outside (“outward facing normals”)
Triangle meshes

• A bunch of triangles in 3D space that are connected together to form a surface

• Geometrically, a mesh is a piecewise planar surface
  – Almost everywhere, it is planar
  – Exceptions are at the edges where triangles join

• Often, it is a piecewise planar approximation of a smooth surface
  – In this case the creases between triangles are artifacts—we do not want to see them
Representation of triangle meshes

- Compactness
- Efficiency for rendering
  - Enumerate all triangles as triples of 3D points
- Efficiency of queries
  - All vertices of a triangle
  - All triangles around a vertex
  - Neighboring triangles of a triangle
  - Application dependent
    - Finding triangle strips
    - Computing subdivision surfaces
    - Mesh editing
Representations for triangle meshes

• Separate triangles
• Indexed triangle set
  – Shared vertices
• Triangle strips and triangle fans
  – Compression schemes for fast transmission
• Triangle-neighbor data structure
  – supports adjacency queries
• Winged-edge data structure
  – supports general polygon meshes
Separate triangles

<table>
<thead>
<tr>
<th>tris[0]</th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tris[0]</td>
<td>$x_0, y_0, z_0$</td>
<td>$x_2, y_2, z_2$</td>
<td>$x_1, y_1, z_1$</td>
</tr>
<tr>
<td>tris[1]</td>
<td>$x_0, y_0, z_0$</td>
<td>$x_3, y_3, z_3$</td>
<td>$x_2, y_2, z_2$</td>
</tr>
<tr>
<td></td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
</tbody>
</table>
Separate triangles

• Array of triples of points
  – float[n_T][3][3]: about 72 bytes per vertex
• 2 triangles per vertex (on average)
• 3 vertices per triangle
• 3 coordinates per vertex
• 4 bytes per coordinate (float)

• Various problems
  – Wastes space (each vertex stored 6 times)
  – Cracks due to roundoff
  – Difficult to find neighbors, if at all
Indexed triangle set

- Store each vertex once
- Each triangle points to its three vertices

Triangle {
    Vertex vertex[3];
}

Vertex {
    float position[3]; // or other data
}

// ... or ...

Mesh {
    float verts[nv][3]; // vertex positions (or other data)
    int tInd[nt][3]; // vertex indices
}
Indexed triangle set

| verts[0] | $x_0, y_0, z_0$ |
| verts[1] | $x_1, y_1, z_1$ |
|          | $x_2, y_2, z_2$ |
|          | $x_3, y_3, z_3$ |
|          | $\vdots$        |

| tInd[0]  | 0, 2, 1 |
| tInd[1]  | 0, 3, 2 |
|          | $\vdots$ |
Estimating storage space

- $n_T = \#\text{tris}; n_V = \#\text{verts}; n_E = \#\text{edges}$
- Euler: $n_V - n_E + n_T = 2$ for a simple closed surface
  - In general, sums to small integer
  - Argument for implication that $n_T:n_E:n_V$ is about 2:3:1
Indexed triangle set

• Array of vertex positions
  – float[$n_V$][3]: 12 bytes per vertex
    • (3 coordinates x 4 bytes) per vertex
• Array of triples of indices (per triangle)
  – int[$n_T$][3]: about 24 bytes per vertex
    • 2 triangles per vertex (on average)
    • (3 indices x 4 bytes) per triangle
• Total storage: 36 bytes per vertex (factor of 2 savings)
• Represents topology and geometry separately
• Finding neighbors is at least well defined
Data on meshes

• We often need to store additional information besides just the geometry
• Store additional data at faces, vertices, or edges
• Examples
  – Colors stored on faces, for faceted objects
  – Information about sharp creases stored at edges
  – Any quantity that varies continuously (without sudden changes, or discontinuities) gets stored at vertices
Key types of vertex data

• Surface normals
  – When a mesh is approximating a curved surface, store normals at vertices

• Texture coordinates
  – 2D coordinates that tell you how to paste images on the surface

• Positions
  – At some level this is just another piece of data
  – Position varies continuously between vertices
Differential geometry

- **Tangent plane**
  - At a point on a smooth surface in 3D there is a unique plane tangent to the surface called the tangent plane.

- **Normal vector**
  - Vector perpendicular to a surface (that is, to the tangent plane).
  - Only unique for smooth surfaces (not at corners or edges).
Surface parameterization

• A surface in 3D is a two-dimensional entity
• Sometimes, we need 2D coordinates for points on the surface
• Defining these coordinates is *parameterizing* the surface
• Examples:
  – Cartesian coordinates on a rectangle (or other planar shape)
  – Cylindrical coordinates \((\theta, y)\) on a cylinder
  – Latitude and longitude on the Earth’s (ellipsoid) surface
  – Spherical coordinates \((\theta, \phi)\) on a sphere
Example: unit sphere

• Position:
  
  \[ x = \cos \theta \sin \phi \]
  
  \[ y = \sin \theta \]
  
  \[ z = \cos \theta \cos \phi \]

• Normal is also position
How to think about vertex normals

• Piecewise planar approximation converges pretty quickly to the smooth geometry as the number of triangles increases
  – For mathematicians: error is $O(h^2)$

• But the surface normals do not converge so well
  – Normal is constant over each triangle, with discontinuous jumps across edges
  – For mathematicians: error is only $O(h)$

• Better: store the “real” normal at each vertex, and interpolate to get normals that vary gradually across triangles
Interpolated normals—2D example

• Approximating circle with increasingly many segments
• Max error in position error drops by factor of 4 at each step
• Max error in normal only drops by factor of 2
Triangle strips

• Take advantage of the mesh property
  – Each triangle is usually adjacent to the previous
  – Let every vertex create a triangle by reusing the second and third vertices of the previous triangle
  – Every sequence of three vertices produces a triangle (but not in the same order)
  – e.g., 0, 1, 2, 3, 4, 5, 6, 7, ... leads to (0 1 2), (2 1 3), (2 3 4), (4 3 5), (4 5 6), (6 5 7), ...
  – For long strips, this requires about one index per triangle
Triangle strips

\begin{align*}
\text{verts[0]} & \begin{bmatrix} x_0, y_0, z_0 \\
x_1, y_1, z_1 \\
x_2, y_2, z_2 \\
x_3, y_3, z_3 \\
\vdots 
\end{bmatrix} \\
\text{verts[1]} & \begin{bmatrix} 
\end{bmatrix}
\end{align*}

\begin{align*}
\text{tStrip[0]} & \begin{bmatrix} 
4, 0, 1, 2, 5, 8 
\end{bmatrix} \\
\text{tStrip[1]} & \begin{bmatrix} 
6, 9, 0, 3, 2, 10, 7 
\end{bmatrix} \\
\vdots & 
\end{align*}
Triangle strips

• Array of vertex positions
  – float[nV][3]: 12 bytes per vertex
    • (3 coordinates x 4 bytes) per vertex

• Array of index lists
  – int[nS][variable]: 2 + n indices per strip
  – On average, (1 + \( \sum \)) indices per triangle (assuming long strips)
    • 2 triangles per vertex (on average)
    • About 4 bytes per triangle (on average)

• Total is 20 bytes per vertex (limiting best case)
  – Factor of 3.6 over separate triangles; 1.8 over indexed mesh
Triangle fans

• Same idea as triangle strips, but keep oldest rather than newest
  – Every sequence of three vertices produces a triangle
  – e.g., 0, 1, 2, 3, 4, 5, ... leads to
    (0 1 2), (0 2 3), (0 3 4), (0 4 5), ...
  – For long fans, this requires about one index per triangle

• Memory considerations exactly the same as triangle strip
Topology vs. geometry

• Two completely separate issues:
  – Mesh topology: how the triangles are connected (ignoring the positions entirely)
  – Geometry: where the triangles are in 3D space
Topology/geometry examples

• Same geometry, different mesh topology

• Same mesh topology, different geometry
Validity of triangle meshes

• In many cases we care about the mesh being able to bound a region of space nicely

• In other cases we want triangle meshes to fulfill assumptions of algorithms that will operate on them (and may fail on malformed input)

• Again, two completely separate issues
  – Topology: how the triangles are connected (ignoring the positions entirely)
  – Geometry: where the triangles are in 3D space
Topological validity

• Strongest property: be a manifold
  – This means that no points should be "special"
  – Interior points are fine
  – Edge points: each edge must have exactly 2 triangles
  – Vertex points: each vertex must have one loop of triangles

• Slightly looser: manifold with boundary
  – Weaken rules to allow boundaries
Topological validity

• Consistent orientation
  – Which side is the “front” or “outside” of the surface and which is the “back” or “inside”?
  – Rule: you are on the outside when you see the vertices in counter-clockwise order
  – In mesh, neighboring triangles should agree about which side is the front!
  – Warning: not always possible

non-orientable
Geometric validity

- Generally, want non-self-intersecting surface
- In general, this is hard to guarantee
  - Far-apart parts of mesh might intersect
Triangle neighbor structure

• Extension to indexed triangle set
• Triangle points to its three neighboring triangles
• Vertex points to a single neighboring triangle
• Can now enumerate triangles around a vertex
Triangle neighbor structure

Triangle {
    Triangle nbr[3];
    Vertex vertex[3];
}

// t.neighbor[i] is adjacent
// across the edge from i to i+1

Vertex {
    // ... per-vertex data ...
    Triangle t; // any adjacent tri
}

// ... or ...

Mesh {
    // ... per-vertex data ...
    int tInd[nt][3]; // vertex indices
    int tNbr[nt][3]; // indices of neighbor triangles
    int vTri[nv]; // index of any adjacent triangle
}
Triangle neighbor structure

| vTri[0] | 0 |
| vTri[1] | 6 |
| vTri[2] | 1 |
| vTri[3] | 1 |

| tNbr[0] | 1, 6, 7 |
| tNbr[1] | 10, 2, 0 |
| tNbr[2] | 3, 1, 12 |
| tNbr[3] | 2, 13, 4 |

| tInd[0] | 0, 2, 1 |
| tInd[1] | 0, 3, 2 |
| tInd[2] | 10, 2, 3 |
| tInd[3] | 2, 10, 7 |
Triangle neighbor structure

TrianglesOfVertex(v) {
    t = v.t;
    do {
        find t.vertex[i] == v;
        t = t.nbr[pred(i)];
    } while (t != v.t);
}

pred(i) = (i+2) % 3;
succ(i) = (i+1) % 3;
Triangle neighbor structure

• Indexed mesh was 36 bytes per vertex
• Add an array of triples of indices (per triangle)
  – int[$n_T$][3]: about 24 bytes per vertex
    • 2 triangles per vertex (on average)
    • (3 indices x 4 bytes) per triangle
• Add an array of representative triangle per vertex
  – int[$n_V$]: 4 bytes per vertex
• Total storage: 64 bytes per vertex
  – Still not as much as separate triangles
Triangle neighbor structure—refined

Triangle {
    Edge nbr[3];
    Vertex vertex[3];
}

// if t.nbr[i].i == j
// then t.nbr[i].t.nbr[j] == t

Edge {
    // the i-th edge of triangle t
    Triangle t;
    int i; // in {0,1,2}
    // in practice t and i share 32 bits
}

Vertex {
    // ... per-vertex data ...
    Edge e; // any edge leaving vertex
}
Triangle neighbor structure


def TrianglesOfVertex(v):
    {t, i} = v.e;
    do {
        {t, i} = t.nbr[pred(i)];
    } while (t != v.t);

    pred(i) = (i+2) % 3;
    succ(i) = (i+1) % 3;

T0.nbr[0] = { T1, 2 }
T1.nbr[2] = { T0, 0 }
V0.e = { T1, 0 }
Winged-edge mesh

- Edge-centric rather than face-centric
  - Therefore also works for polygon meshes
- Each (oriented) edge points to:
  - Left and right forward edges
  - Left and right backward edges
  - Front and back vertices
  - Left and right faces
- Each face or vertex points to one edge
Winged-edge mesh

Edge {
    Edge hl, hr, tl, tr;
    Vertex h, t;
    Face l, r;
}

Face {
    // per-face data
    Edge e;  // any adjacent edge
}

Vertex {
    // per-vertex data
    Edge e;  // any incident edge
}
Winged-edge structure

EdgesOfFace(f) {
    e = f.e;
    do {
        if (e.t == f)
            e = e.tl;
        else
            e = e.hr;
    } while (e != f.e);
}

EdgesOfVertex(v) {
    e = v.e;
    do {
        if (e.t == v)
            e = e.tl;
        else
            e = e.hr;
    } while (e != v.e);
}

<table>
<thead>
<tr>
<th></th>
<th>hl</th>
<th>hr</th>
<th>tl</th>
<th>tr</th>
</tr>
</thead>
<tbody>
<tr>
<td>edge[0]</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>edge[1]</td>
<td>18</td>
<td>0</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>edge[2]</td>
<td>12</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

CSE 167, Winter 2018
Winged-edge structure

- Array of vertex positions: 12 bytes/vert
- Array of 8-tuples of indices (per edge)
  - Head/tail left/right edges + head/tail verts + left/right tris
  - $\text{int}[n_E][8]$: about 96 bytes per vertex
    - 3 edges per vertex (on average)
    - (8 indices x 4 bytes) per edge
- Add a representative edge per vertex
  - $\text{int}[n_V]$: 4 bytes per vertex
- Total storage: 112 bytes per vertex
  - But it is cleaner and generalizes to polygon meshes
Winged-edge optimizations

• Omit faces if not needed
• Omit one edge pointer on each side
  – Results in one-way traversal
Half-edge structure

• Simplifies, cleans up winged edge
  – Still works for polygon meshes
• Each half-edge points to:
  – Next edge (left forward)
  – Next vertex (front)
  – The face (left)
  – The opposite half-edge
• Each face or vertex points to one half-edge
Half-edge structure

HEdge {
    HEdge pair, next;
    Vertex v;
    Face f;
}

Face {
    // per-face data
    HEdge h; // any adjacent h-edge
}

Vertex {
    // per-vertex data
    HEdge h; // any incident h-edge
}
Half-edge structure

```c
EdgesOfFace(f) {
    h = f.h;
    do {
        h = h.next; pair;
    } while (h != f.h);
}
```

```plaintext
<table>
<thead>
<tr>
<th>pair</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
```
Half-edge structure

• Array of vertex positions: 12 bytes/vert
• Array of 4-tuples of indices (per h-edge)
  – Next, pair h-edges + head vert + left tri
  – int[2n_E][4]: about 96 bytes per vertex
    • 6 h-edges per vertex (on average)
    • (4 indices x 4 bytes) per h-edge
• Add a representative h-edge per vertex
  – int[n_V]: 4 bytes per vertex
• Total storage: 112 bytes per vertex
Half-edge optimizations

- Omit faces if not needed
- Use implicit pair pointers
  - They are allocated in pairs
  - They are even and odd in an array