Polygon Meshes

Shirley & Marschner,
Ch12.1, "Triangle Meshes"
**Aspects of 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)
- two completely separate issues:
  - 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:

**Euler’s Formula**

- \( n_V = \#\text{verts}; \ n_E = \#\text{edges}; \ n_F = \#\text{faces} \)
- Euler’s Formula for a convex polyhedron:
  \[ n_V - n_E + n_F = 2 \]
- Other meshes often sum to small integer
  - argument for implication that \( n_V, n_E, n_F \) is about 1:3:2
  - Consider semi-regular subdivision meshes

**Examples of simple convex polyhedra**

<table>
<thead>
<tr>
<th>Name</th>
<th>Image</th>
<th>Vertices</th>
<th>Edges</th>
<th>Faces</th>
<th>Euler characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tetrahedron</td>
<td><img src="image" alt="Tetrahedron" /></td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Hexahedron or cube</td>
<td><img src="image" alt="Hexahedron" /></td>
<td>8</td>
<td>12</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Octahedron</td>
<td><img src="image" alt="Octahedron" /></td>
<td>6</td>
<td>12</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Dodecahedron</td>
<td><img src="image" alt="Dodecahedron" /></td>
<td>20</td>
<td>30</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>Icosahedron</td>
<td><img src="image" alt="Icosahedron" /></td>
<td>12</td>
<td>30</td>
<td>20</td>
<td>2</td>
</tr>
</tbody>
</table>

http://en.wikipedia.org/wiki/Euler_characteristic
Examples of simple convex polyhedra

![Buckyball](http://idav.ucdavis.edu/~okreylos/BuckyballStick.gif)

\[ V = 60 \quad E = 90 \quad F = 32 \text{ (12 pentagons + 20 hexagons)} \]

\[ V - E + F = 60 - 90 + 32 = 2 \]

Examples (nonconvex polyhedra!)

<table>
<thead>
<tr>
<th>Name</th>
<th>Image</th>
<th>Vertices</th>
<th>Edges</th>
<th>Faces</th>
<th>Euler characteristic: ( V - E + F )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tetrahemihexahedron</td>
<td><img src="http://en.wikipedia.org/wiki/Tetrahedron" alt="Image" /></td>
<td>6</td>
<td>12</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Octahemioctahedron</td>
<td><img src="http://en.wikipedia.org/wiki/Octahedron" alt="Image" /></td>
<td>12</td>
<td>24</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Cubohemioctahedron</td>
<td><img src="http://en.wikipedia.org/wiki/Cuboctahedron" alt="Image" /></td>
<td>12</td>
<td>24</td>
<td>10</td>
<td>-2</td>
</tr>
<tr>
<td>Great icosahedron</td>
<td><img src="http://en.wikipedia.org/wiki/Icosahedron" alt="Image" /></td>
<td>12</td>
<td>30</td>
<td>20</td>
<td>2</td>
</tr>
</tbody>
</table>

Topological validity

- Strongest property, and most simple: be a manifold
  - this means that no points should be "special"
  - interior points are fine
  - edge points: each edge should have exactly 2 triangles
  - vertex points: each vertex should have one loop of triangles
    - not too hard to weaken this to allow boundaries

[Foley et al.]
Geometric validity

- Generally want non-self-intersecting surface
- Hard to guarantee in general
  - because far-apart parts of mesh might intersect

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
  - (need depends on application)
    - 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 transmission to hardware
- Triangle-neighbor data structure
  - supports adjacency queries
- Winged-edge data structure
  - supports general polygon meshes

Separate triangles
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
  - difficulty of finding neighbors 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

- array of vertex positions
  - float[nV][3]: 12 bytes per vertex
    - (3 coordinates x 4 bytes) per vertex
- array of triples of indices (per triangle)
  - int[nT][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

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**

- 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 + ε) 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 3 5)
  - for long fans, this requires about one index per triangle
- Memory considerations exactly the same as triangle strip

**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

- 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

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

// t.neighbor[1] 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
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_T]: 4 bytes per vertex
- total storage: 64 bytes per vertex
  - still not as much as separate triangles

Triangle neighbor structure—refined

```c
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
}
```

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

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.l == f)
            e = e.hl;
        else
            e = e.tr;
    } 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);
}

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
  - 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
  - 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 (next)
  - next vertex (head)
  - the face (left)
  - the opposite half-edge (pair)
- 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
}
EdgesOfFace(f) {
    h = f.h;
    do {
        h = h.next;
    } while (h != f.h);
}

EdgesOfVertex(v) {
    h = v.h;
    do {
        h = h.next.pair; // typo in text
    } while (h != v.h);
}
**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