Triangle meshes

CS 4620 Lecture 7
spheres

approximate sphere
finite element analysis
Ottawa Convention Center
Notation

• \( n_T = \#\text{tris}; n_V = \#\text{verts}; n_E = \#\text{edges} \)

• Euler: \( n_V - n_E + n_T = 2 \) for a simple closed surface
  – and in general sums to small integer
  – argument for implication that \( n_T:n_E:n_V \) is about 2:3:1
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)
• 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:
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!
  – caution: not always possible

[Diagram of topological validity with 'OK' and 'bad' examples]
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

<table>
<thead>
<tr>
<th>tris[0]</th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₀, y₀, z₀</td>
<td>x₂, y₂, z₂</td>
<td>x₁, y₁, z₁</td>
<td></td>
</tr>
<tr>
<td>x₀, y₀, z₀</td>
<td>x₃, y₃, z₃</td>
<td>x₂, y₂, z₂</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Diagram showing separate triangles T₀ and T₁ with vertices (x₀,y₀,z₀), (x₁,y₁,z₁), (x₂,y₂,z₂), and (x₃,y₃,z₃).
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

• 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] | x₀, y₀, z₀ |
| verts[1] | x₁, y₁, z₁ |
|          | x₂, y₂, z₂ |
|          | x₃, y₃, z₃ |
|          | ⋮          |

| tInd[0] | 0, 2, 1   |
| tInd[1] | 0, 3, 2   |
|          | ⋮          |

Diagram showing triangles $T₀$ and $T₁$. Points $p₀$, $p₁$, $p₂$, $p₃$, $p₅$, $p₆$, $p₇$, and $p₉$ are connected by edges to form the triangles.
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
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

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

| tStrip[0] | 4, 0, 1, 2, 5, 8 |
| tStrip[1] | 6, 9, 0, 3, 2, 10, 7 |
|           | $\vdots$       |
Triangle strips

The image shows a diagram of triangle strips with corresponding vertices and triangle strips stored in arrays `verts` and `tStrip`. The vertices are stored in the `verts` array, and the triangle strips are stored in the `tStrip` array.
Triangle strips

- array of vertex positions
  - float[n_V][3]: 12 bytes per vertex
    - (3 coordinates x 4 bytes) per vertex
- array of index lists
  - int[n_S][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 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

```
| 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
Triangle neighbor structure
Triangle neighbor structure

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

T0.nbr[0] = { T1, 2 }
T1.nbr[2] = { T0, 0 }
V0.e = { T1, 0 }
Triangle neighbor structure

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;

T_0.nbr[0] = { T_1, 2 }
T_1.nbr[2] = { T_0, 0 }
V_0.e = { T_1, 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-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-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

<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>

...
Winged-edge structure

EdgesOfFace(f) {
    e = f.e;
    do {
        if (e.l == f)
            e = e.hl;
        else
            e = e.tr;
    } while (e != f.e);
}
Winged-edge structure

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
  - \text{int}[n_E][8]: about 96 bytes per vertex
    - 3 edges per vertex (on average)
    - (8 indices \times 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

- 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

- 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
Half-edge structure

<table>
<thead>
<tr>
<th>hedge[0]</th>
<th>pair</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>hedge[1]</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>hedge[2]</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>hedge[3]</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>hedge[4]</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>hedge[5]</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

...
Half-edge structure
Half-edge structure

```c
EdgesOfFace(f) {
    h = f.h;
    do {
        h = h.next;
    } while (h != f.h);
}
```
Half-edge structure

EdgesOfVertex(v) {
    h = v.h;
    do {
        h = h.next.pair;
    } while (h != v.h);
}

<table>
<thead>
<tr>
<th>pair</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>hedge[0]</td>
<td>1 2</td>
</tr>
<tr>
<td>hedge[1]</td>
<td>0 10</td>
</tr>
<tr>
<td>hedge[2]</td>
<td>3 4</td>
</tr>
<tr>
<td>hedge[3]</td>
<td>2 9</td>
</tr>
<tr>
<td>hedge[4]</td>
<td>5 0</td>
</tr>
<tr>
<td>hedge[5]</td>
<td>4 6</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