CS4620/5620: Lecture 17

Meshes

Announcements

• Prelim next Thursday
  – In the evening, closed book
  – Including material of this week
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[nT][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
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

Triangle strips

- Take advantage of the mesh property
  - each triangle is usually adjacent to the previous
  - let every vertex create a triangle by reusing two 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 4 5),
  – for long fans, this requires about one index per triangle
• Memory considerations exactly the same as triangle strip

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
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.nbr[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
}
Triangle neighbor structure

TrianglesOfVertex(v) {
    t = v.t;
    do {
      i = (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[nT][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[nV]: 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].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 = { T0, 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;

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

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>hl</td>
<td>hr</td>
<td>tl</td>
<td>tr</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
**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
- add a representative h-edge per face
  - int[n_f]: 4 bytes per face
- total storage: 120 bytes per vertex
  - but it is powerful and generalizes to polygon meshes

**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);
}

<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
- add a representative h-edge per face
  - int[n_f]: 4 bytes per face
- total storage: 120 bytes per vertex

Discussion

- Winged edge and half edge
  - Very powerful
  - Efficient access for queries
  - Local changes in meshes, so good for dynamics
  - But, large memory
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