CS4620/5620: Lecture 19

Meshes

Announcements

- Prelim on Monday
  - In class, closed book
- PPA 1 out
  - Class today, start early!

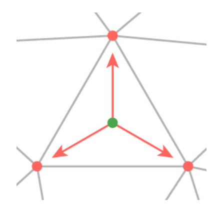
Representations for triangle meshes

- 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

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
  }
```



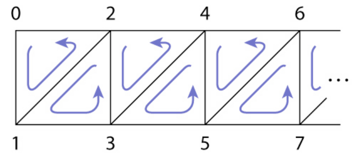
Indexed triangle set

- array of vertex positions
  - float[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

Representations for triangle meshes

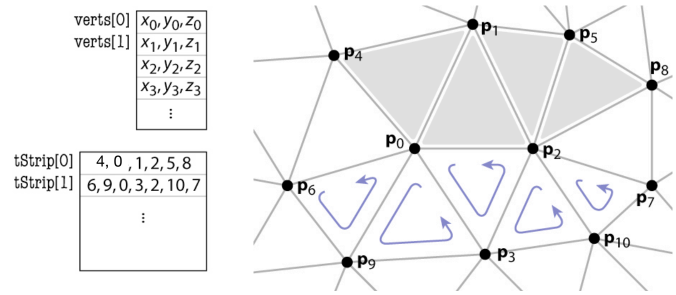
- Separate triangles
- Indexed triangle set
  - shared vertices
- Triangle strips and triangle fans
  - compression schemes for transmission to hardware
- Triangle-neighbor data structure
  - supports adjacency queries
- Winged-edge data structure
  - supports general polygon meshes

Triangle strips



- Take advantage of the mesh property
  - each triangle is usually adjacent to the previous
  - let every vertex create a triangle by reusing 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

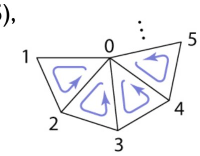


Triangle strips

- 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

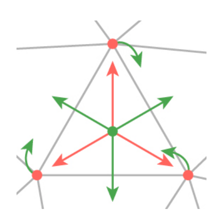


Representations for triangle meshes

- 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

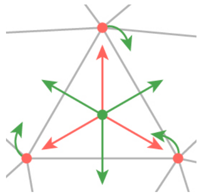
Triangle neighbor structure

- 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 neighbor structure

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

// t.nbr[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
  }
```

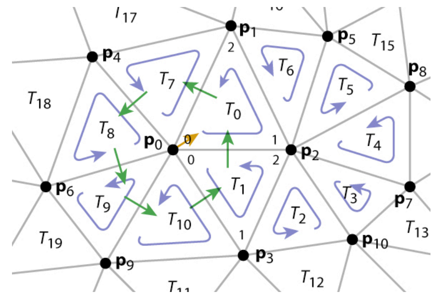


Triangle neighbor structure

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

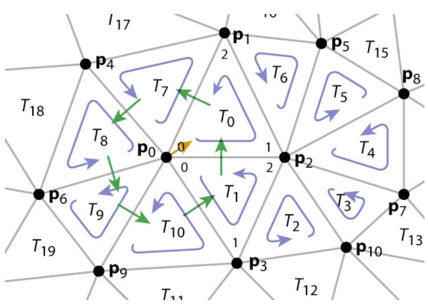
tNbr[0]	1, 6, 7
tNbr[1]	10, 2, 0
tNbr[2]	3, 1, 12
tNbr[3]	2, 13, 4
	⋮

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 {
    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;
```

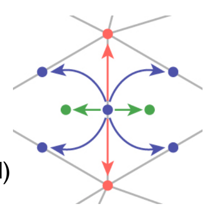


Triangle neighbor structure

- indexed mesh was 36 bytes per vertex
- add an array of triples of indices (per triangle)
  - int[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

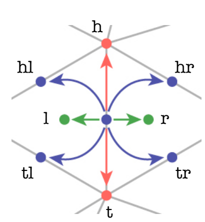
Winged-edge mesh

- 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 (head and tail)
  - left and right faces
- Each face or vertex points to one edge



Winged-edge mesh

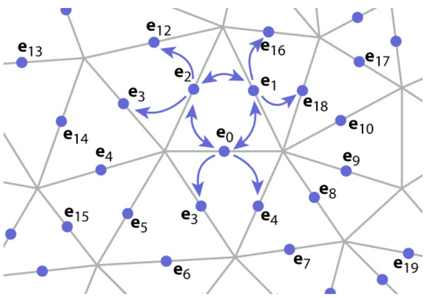
```
Edge {
  Edge hl, hr, tl, tr;
  Vertex h, t;
  Face l, r;
  }

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

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



Winged-edge structure

```
EdgesOfVFarcteex(fx)(v) {
  e = fv.e;
  do {
    if (e.lt == fv)
      e = e.thll;
    else
      e = e.thrr;
  } while (e != fv.e);
}
```

	hl	hr	tl	tr
edge[0]	1	4	2	3
edge[1]	18	0	16	2
edge[2]	12	1	3	0
	⋮			

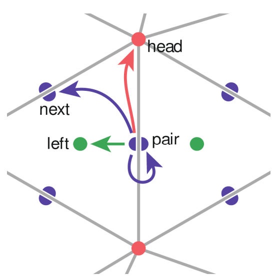


Winged-edge structure

- 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

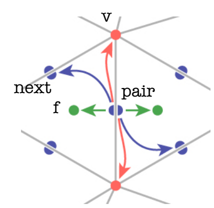
Half-edge structure

- 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

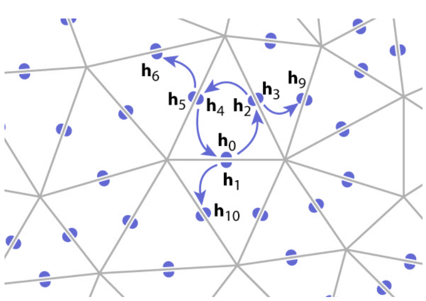
```
HEdge {
  HEdge pair, next;
  Vertex v;
  Face f;
  }

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

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



Half-edge structure

```
EdgesOfVFarcteex(fx)(v) {
  h = fv.h;
  do {
      h = h.next;pair;  // typo in text
    } while (h != fv.h);
}
```

	pair	next
hedge[0]	1	2
hedge[1]	0	10
hedge[2]	3	4
hedge[3]	2	9
hedge[4]	5	0
hedge[5]	4	6
	⋮	



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