CSC 433/533 Computer Graphics

Joshua Levine
josh@email.arizona.edu

Lecture 14 Triangle Meshes

Oct. 14, 2019

Today's Agenda

- Reminders:
  - A04 any questions?
  - WARNING: Midterm in one week!!
- Goals for today:
  - Discuss how complex shapes are stored and rendered

Midterm

- Introduction
  - Basics (Ch.1)
  - Color and Images (Ch.3)
  - Perception (Ch.19)
- Image Processing
  - Signal Processing Concepts (Ch.9)
  - Tone Reproduction (Ch.21.1-21.3)
  - Composition (Ch.20.1-20.2)
- Ray Tracing Concepts
  - Vectors (Ch.2.1-2.4)
  - Ray Tracing (Ch.4)
  - Shading (Ch.10)
  - Advanced Ray Tracing (Ch.13)
  - Global Illumination (Ch.23)
- Shapes
  - Meshes (Ch.12.1)
  - Accel. Structures (Ch.12.3)

Midterm Format

- Closed book, closed notes: Only need a pen or pencil.
- 12 short answer questions (very similar in format to written questions on homework)
  - 5-6 points each (65% total)
- 3 long answer questions that require explaining a concept more in depth
  - 10-15 points each (35% total)

Modeling Complex Shapes

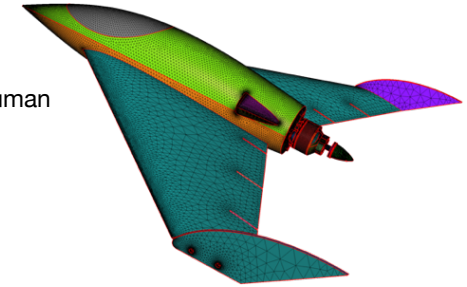
Recall: Shape Models That We Have So Far

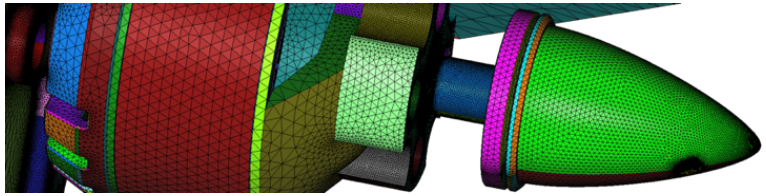
- Implicit Shapes ($f(\mathbf{p}) = 0$ for all $\mathbf{p}$ on shape):
  - Sphere: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0$
  - Plane: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$
- Parametric Shapes ($\mathbf{p}(t)$ is a point on shape for all t):
  - Rays: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Triangles: $\mathbf{p}$ $= \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$

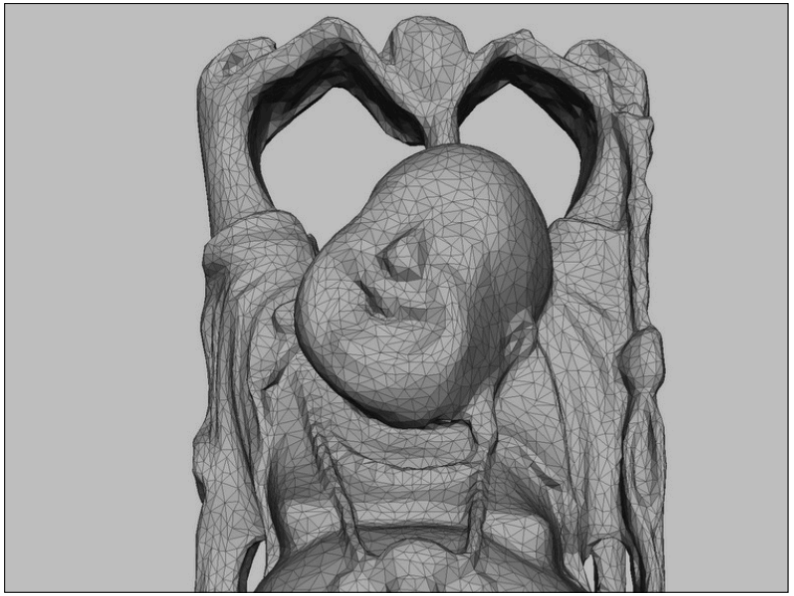
Triangle Meshes

- Are used in a huge number of applications
- Can be used to represent complex shapes by breaking them into simple (perhaps the simplest) two-dimensional elements

- Patrick Laug & Houman Borouchaki 2013
- 1,844,460 triangles

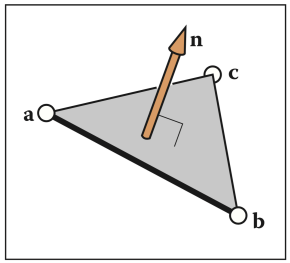






Definition of Triangles

- 3 **vertices** (points $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ in 3D space)
- The normal of the triangle is a vector, $\mathbf{n}$, that points to its front side
- Convention: vertices listed in counter-clockwise order from the “front” of the triangle

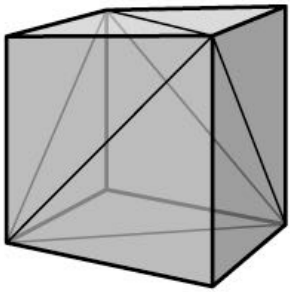


Definition of Triangle Meshes

- In short, a collection of triangles in 3D space that are connected to form a surface
  - Terminology: vertices, edges, triangles
  - Surface is piecewise planar, except where two triangle meet which forms a crease and their shared edge
  - Meshes are often a piecewise approximation of a smooth surface — graphics can hide the artifacts

A Simple Mesh

- How many vertices? How many triangles?



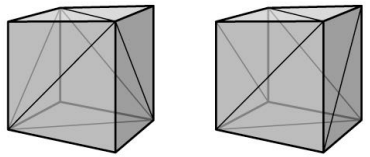
Mesh Topology

Two Considerations for Meshes

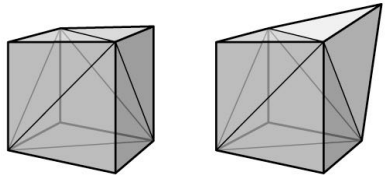
- We typically care about the mesh being a good approximation to a surface:
  - This leads to questions of **mesh geometry**, e.g.: How many triangles? where to place their vertices?
- We also care about how these triangles are connected
  - This leads to questions of **mesh topology**, e.g.: Are there holes in the mesh? How do triangles intersect?
  - Mesh topology can affect assumptions on algorithms that process meshes

Topology vs. Geometry

- Same geometry, different topology



- Same topology, different geometry

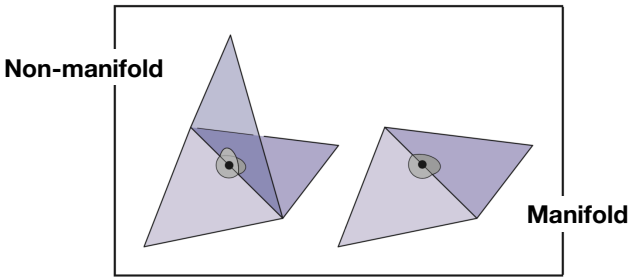


Topological Validity

- Meshes that approximate surfaces should be manifolds
- Definition: A (2-dimensional) **manifold** is a space where every point locally appears to be 2-dimensional space
  - 3 cases: points that are on edges, points that are vertices, and points that are interior to triangles.

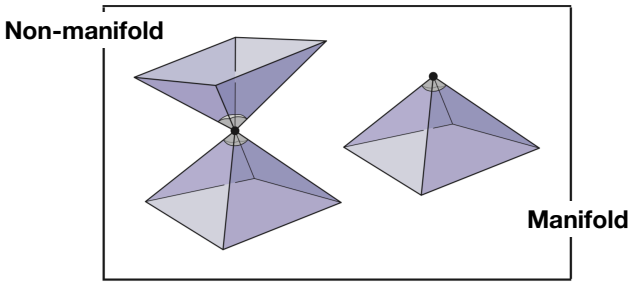
When is a Mesh a Manifold?

- Definition: A (2-dimensional) manifold is a space where every point locally appears to be 2-dimensional space
- Implication: Every edge is shared by exactly two triangles



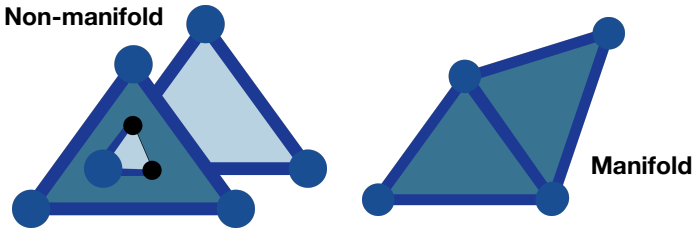
When is a Mesh a Manifold?

- Definition: A (2-dimensional) manifold is a space where every point locally appears to be 2-dimensional space
- Implication: Every vertex has a single, complete loop of triangles around it



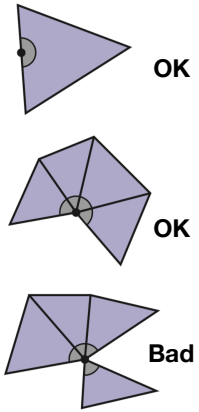
When is a Mesh a Manifold?

- Definition: A (2-dimensional) manifold is a space where every point locally appears to be 2-dimensional space
- Implication: Triangles only intersect at vertices and edges



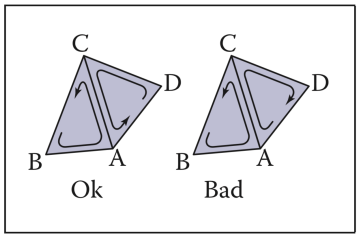
Manifolds with Boundary

- Sometimes, we relax the manifold condition to allow meshes with boundaries.
- Every point on a **manifold with boundary** either locally appears to be 2-dimensional space or 2-dimensional half-space
  - Every edge is used by either one or two triangles
  - Every vertex connects to a single edge-connected set of triangles

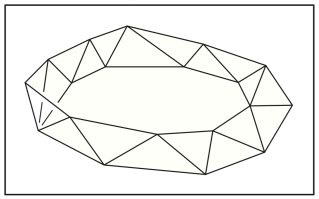


Consistent Orientation

- In many applications, all triangles facing the same way is important
  - Can be used to distinguish inside from outside.
- If consistent: neighboring triangles will appear to disagree on the order of vertices on their shared edge



Möbius strip: Non-orientable



Simple Representations of Triangle Meshes

Important Concerns w/ Representing Triangle Meshes

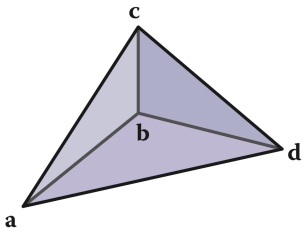
- Efficiency of storage size
  - Many representations store redundant information
- Efficiency of access
  - How quickly can we get the information we need for rendering?
  - How quickly can we get neighborhood information, for mesh modification?

Using Separate Triangles

- Use a simple structure to store each triangle:

```
Triangle {
  vertexPositions[3];  //Vec3
};
```

- Store a triangle mesh using an array of `Triangle`
- Problems?



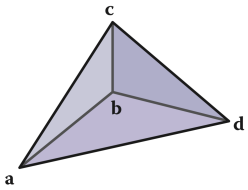
#	Vertex 0	Vertex 1	Vertex 2
0	(a_x, a_y, a_z)	(b_x, b_y, b_z)	(c_x, c_y, c_z)
1	(b_x, b_y, b_z)	(d_x, d_y, d_z)	(c_x, c_y, c_z)
2	(a_x, a_y, a_z)	(d_x, d_y, d_z)	(b_x, b_y, b_z)

Using Indexed Meshes

- Triangles share a common list of vertices, storing only references/ pointers:

```
Triangle {
  vertices[3]; //object reference or int
};

Vertex {
  position;    //Vec3
};
```

- Store a triangle mesh using two arrays, one of `Vertex` and the other of `Triangle`

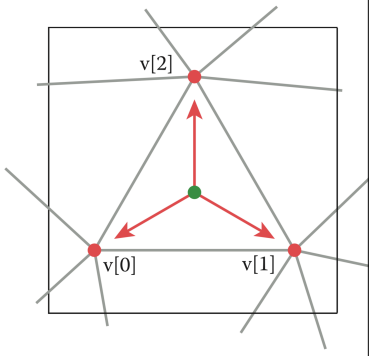


Triangles	
#	Vertices
0	(0, 1, 2)
1	(1, 3, 2)
2	(0, 3, 1)

Vertices	
#	Position
0	(a_x, a_y, a_z)
1	(b_x, b_y, b_z)
2	(c_x, c_y, c_z)
3	(d_x, d_y, d_z)

Using Indexed Meshes

- Each triangle thus tracks references to the vertices associated with it

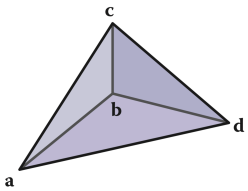


Using Indexed Meshes

- Alternatively one can store using array indices directly:

```
IndexedMesh {
  vertices[num_verts];   //Vec3
  triIndices[num_tris];  //int
};
```

- Plus, it is easy (or at least easier) to see which two triangles share an edge.



Triangles	
#	Vertices
0	(0, 1, 2)
1	(1, 3, 2)
2	(0, 3, 1)

Vertices	
#	Position
0	(a_x, a_y, a_z)
1	(b_x, b_y, b_z)
2	(c_x, c_y, c_z)
3	(d_x, d_y, d_z)

Storage Requirements

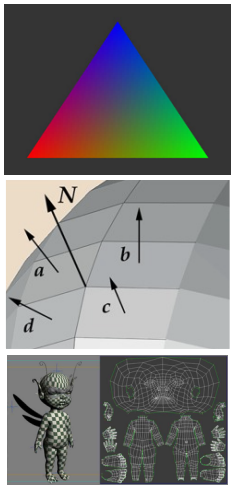
- Let n_v be the number of vertices and n_t be the number of triangles.
  - A single vertex requires 12 bytes (3 floats, 4 bytes/float)
- Separate triangles:
  - 3 vertices per triangle => $12*3*n_t = 36*n_t$ bytes
- Indexed Meshes
  - 3 integers per triangle, 3 floats per vertex => $12*n_t + 12*n_v$ bytes

Storage Requirements

- For large meshes, $n_t \approx 2n_v$
- Separate triangles:
  - $36*n_t = 72*n_v$ bytes
- Indexed Meshes
  - $12*n_t + 12*n_v = 36*n_v$ bytes

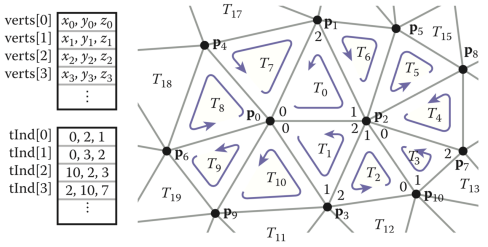
Data on Meshes

- Typically, we store a variety of data on meshes as well
- Can store this on vertices, triangles, or even edges
- Examples:
  - Colors stored on vertices
  - Normals stored on faces
  - Texture coordinates stored on vertices
- Information stored on vertices is typically interpolated with barycentric coordinates



Mesh File Formats: *.obj

- Widely used format for indexed meshes
- Supports additional data stored on vertices and polygons

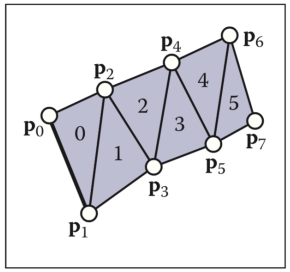



```
#sample .obj file

v 0.000 0.000 0.000
v 0.500 0.809 0.309
v 1.000 0.000 -0.309
v 0.583 -0.720 0.225
v -0.630 0.750 0.025
...

f 1 3 2
f 1 4 3
f 11 3 4
f 3 11 7
...
```

More Efficient Representations

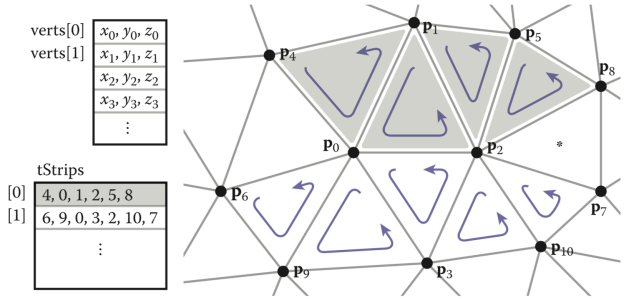
Triangle Strips

- Idea: Rely on the mesh property and group triangles that share common vertices
- Create a new triangle by reusing the last two vertices in the strip
- [0,1,2,3,4,5,6,7] specifies the sequence on the right with triangles (0,1,2), (1,2,3), (2,3,4) …
- Have to invert every other for consistent orientation



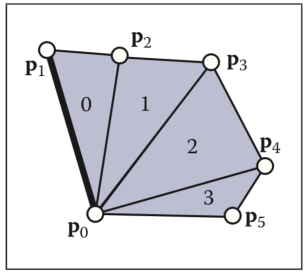
Triangle Strips

- Complex meshes store list of strips
- How long of a strip to use?



Triangle Fans

- Same idea as triangle strips, but keep the earliest vertex in the list instead of the last two
- [0,1,2,3,4,5] specifies the sequence on the right with triangles (0,1,2), (0,2,3), (0,3,4), …



Mesh Data Structures and Queries

Queries on Meshes

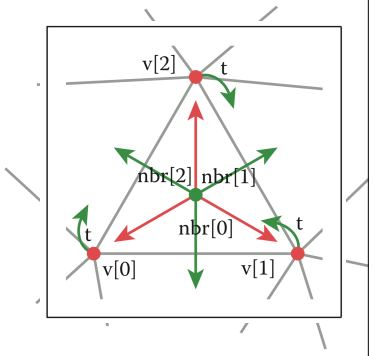
- For face, find all:
  - Vertices
  - Edges
  - Adjacent faces
- For vertex, find all:
  - Incident edges
  - Incident triangles
  - Neighboring vertices
- For edge, find:
  - Two adjacent faces
  - Two adjacent vertices

```
Triangle {
  v[3];   //Vertex
  e[3];   //Edge
  adj[3]; //Triangle
}

Vertex {
  t[];    //Triangle
  e[];    //Edge
  adj[];  //Vertex
}

Edge {
  v[2];   //Vertex
  t[2];   //Triangle
}
```

Can we do better?

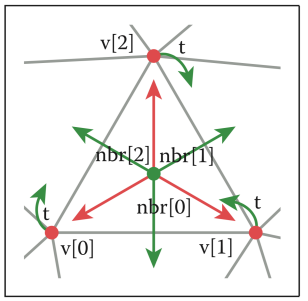
Triangle-Neighbor Structure

- Let's try first extending the indexed mesh structure for sharing vertices
- Add pointers, `nbr[]`, to 3 neighboring triangles
- Add a single pointer, `t`, for each vertex to one of its adjacent triangles
- Can now enumerate triangles adjacent to vertices

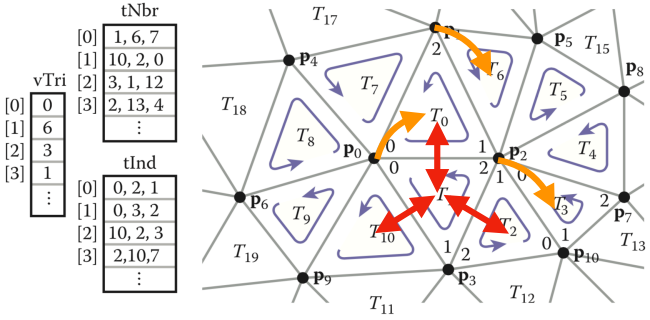


Triangle-Neighbor Structure

```
Triangle {
  v[3];    //Vertex
  nbr[3];  //Triangle
}

Vertex {
  ...
  t;       //Triangle
}

...or...

IndexedMesh {
  ...
  tInd[num_tris];  //int[3]
  tNbr[num_tris];  //int[3]
  vTri[num_verts]; //int
};
```



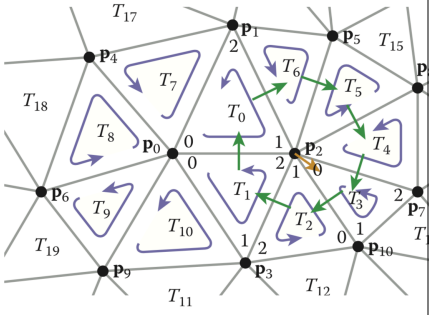
Triangle-Neighbor Structure



Triangle-Neighbor Structure

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

- Can optimize by storing pointers to neighboring edges

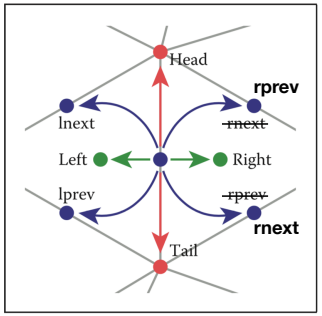


Triangle-Neighbor Structure

- Recall that indexed meshes needed $36*n_v$ bytes and $n_t \approx 2n_v$
- We added an array of triples of indices (per triangle)
  - This increases storage by $3*4*n_t$ or $24*n_v$ bytes
- We also added an array of representative triangle per vertex
  - This increases storage by $4*n_v$ bytes
- Total storage: 36 + 24 + 4 = 64 bytes per vertex
  - Still not as much as separate triangles

Winged-Edge Structure

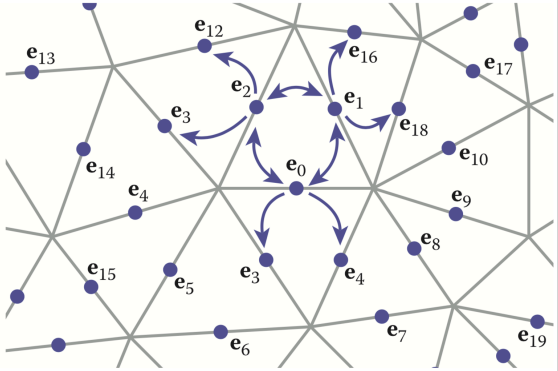
- Widely used mesh structure that focuses on edges instead of triangles
- Edges store pointers to:
  - Head/Tail vertices
  - Left/Right triangles
  - Left/Right “next” edges
  - Left/Right “previous” edges
- Each vertex/triangle stores one pointer to some edge



Winged-Edge Structure

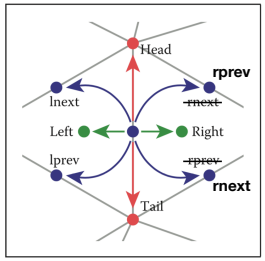
Winged-edge table

	ln	rp	lp	rn
[0]	1	4	2	3
[1]	18	0	16	2
[2]	12	1	3	0
	⋮			



Winged-Edge Structure

```
Edge {
  lprev, lnext, rprev, rnext; //Edge
  head, tail;                 //Vertex
  left, right;                //Face
}

Face {
  ...
  e;  //Edge
}

Vertex {
  ...
  e;  //Edge
}
```



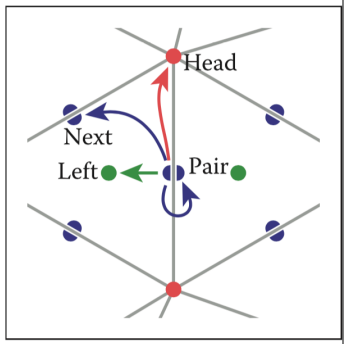

```
EdgesOfVertex(v) {
  e = v.e;
  do {
    if (e.tail == v) {
      e = e.lprev;
    } else {
      e = e.rprev;
    }
  } while (e != v.e);
}
```

Winged-Edge Storage Requirements

- Vertex data: 3 floats for position, 1 int for edge reference
  - $4*4 = 16n_v$ bytes
- Face data: 1 int for edge reference
  - $4*1 = 4n_t = 8n_v$ bytes.
- Edge data, 8 ints for references
  - $n_e \approx 3n_v$
  - $8*4*3 = 96n_v$ bytes.
- In total, $120n_v$ bytes.

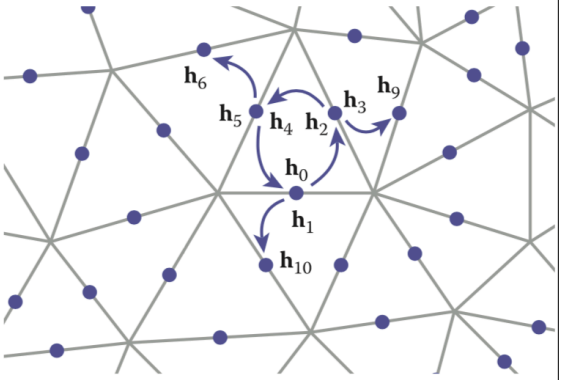
Half-Edge Structure

- Simplifies winged-edge, removes awkwardness of checking which way edges are oriented
- Each **half-edge** store pointers to:
  - Head vertex
  - Left triangle
  - Left “next” edge
  - The opposite “pair” half-edge
- Each vertex/triangle stores one pointer to a half-edge



Half-Edge Structure

	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

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

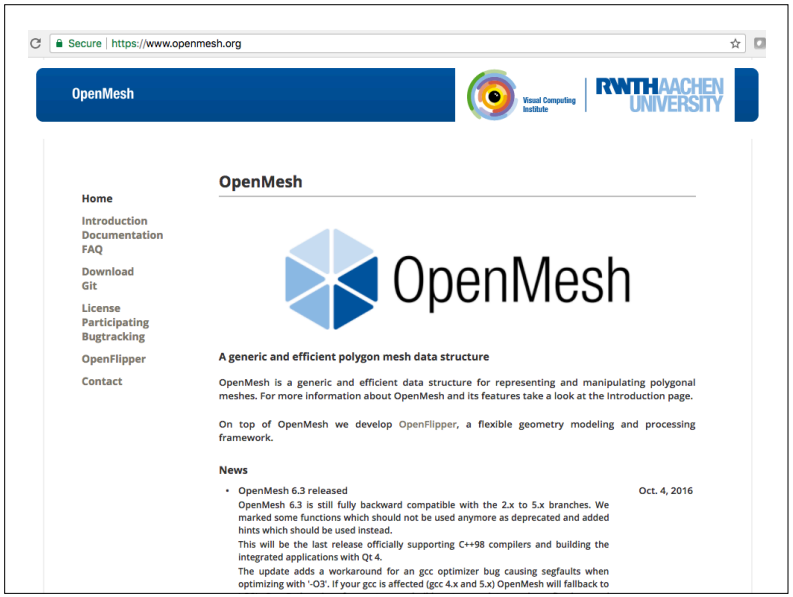
```
EdgesOfVertex(v) {
  h = v.h;
  do {
    h = h.next.pair;
  } while (h != v.h);
}
```

```
EdgesOfVertex(v) {
  e = v.e;
  do {
    if (e.tail == v) {
      e = e.lprev;
    } else {
      e = e.rprev;
    }
  } while (e != v.e);
}
```

Winged-Edge Implementation

Half-Edge Storage Requirements

- Vertex data: 3 floats for position, 1 int for edge reference
  - $4*4 = 16n_v$ bytes
- Face data: 1 int for edge reference
  - $4*1 = 4*n_t = 8n_v$ bytes.
- Edge data, 4 ints for references, but store a pair of half edges for each edge
  - $n_h \approx 6n_v$
  - $8*4*6 = 96n_v$ bytes.
- In total, $120n_v$ bytes.



Lec15 Required Reading

- FOCG, Ch. 12.3

Reminder: Assignment 04

Assigned: Wednesday, Oct. 9
Written Due: Monday, Oct. 23, 4:59:59 pm
Program Due: Wednesday, Oct. 28, 4:59:59 pm