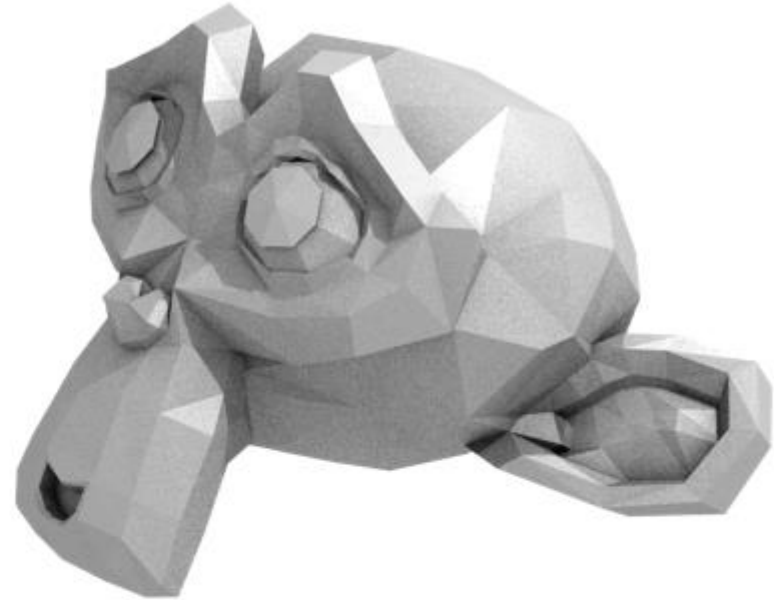
Meshes

Outline

- What is a mesh
  - Geometry vs topology
  - Manifolds
  - Orientation / compatability
  - Genus vs boundary

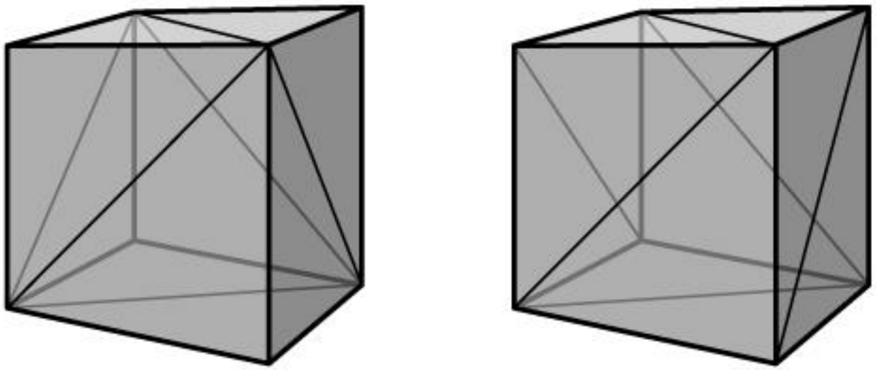
Mesh Definitions



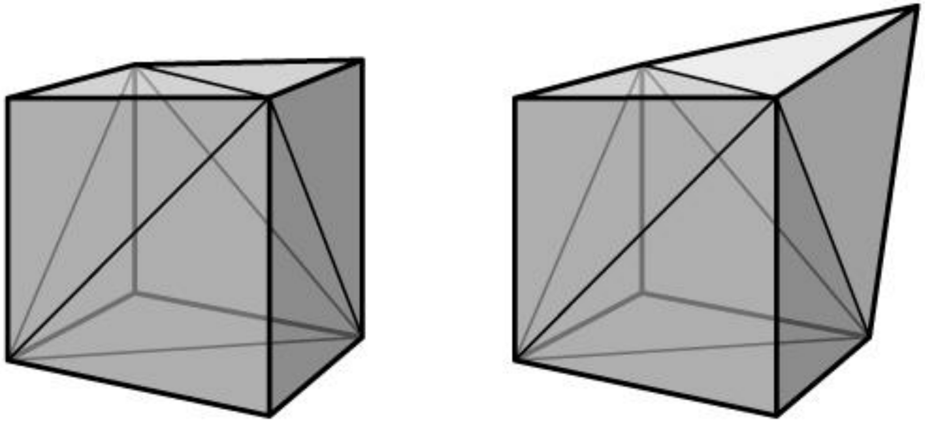
- A surface constructed from polygons that are joined by common edges.
- A mesh consists of ***vertices***, ***edges,*** and ***faces***
- Mesh ***connectivity*** (i.e., ***topology***) describes ***incidence relationships***, e.g., adjacent vertices, edges, faces.
- Mesh ***geometry*** describes positions and other geometric characteristics

Topology/Geometry Examples

- Same geometry, different mesh topology:

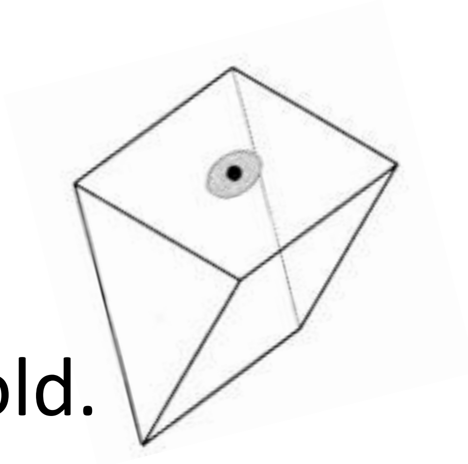


- Same mesh topology, different geometry:

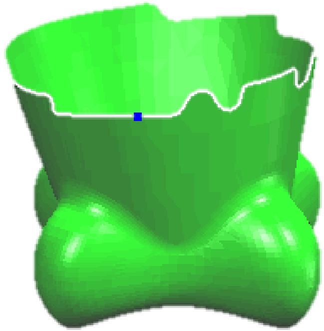


More Definitions

- A 2D **manifold** a topological space that locally resembles 2D Euclidean space. Meshes can be manifold or non-manifold.



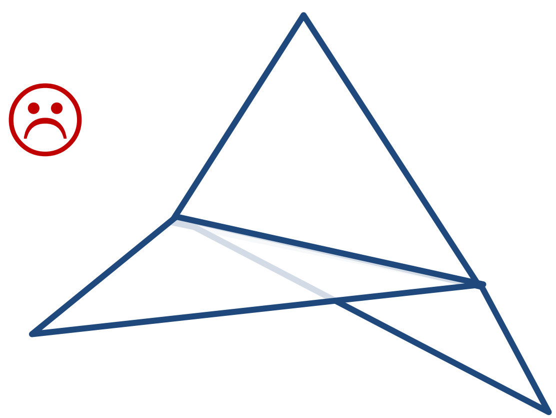
- If an edge belongs to only one polygon (i.e., one face) then it is on the ***boundary*** of the surface.

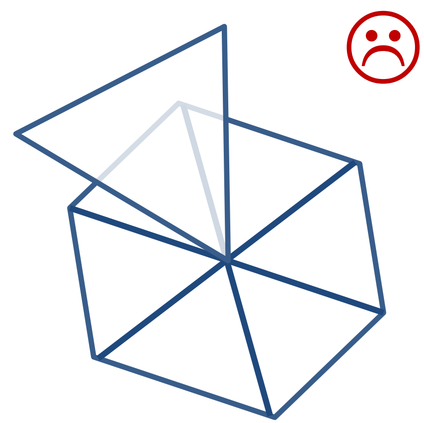


- If an edge belongs to 2 polygons then it is in the ***interior*** of the surface.

Non-manifold Examples

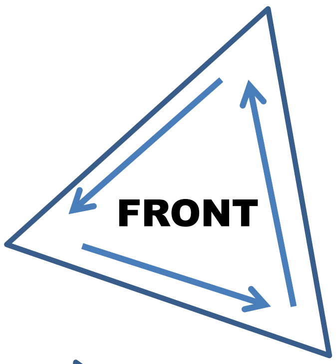
- If an edge belongs to more than 2 polygons then the mesh is ***non-manifold***
- Vertices must likewise have a single ***fan*** of incident faces for the mesh to be manifold

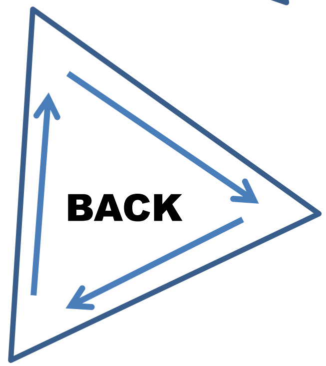


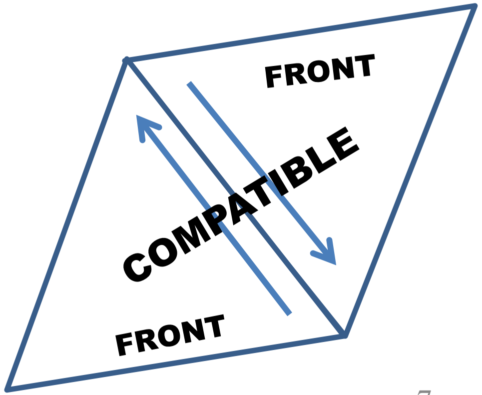


More Definitions

- The ***orientation*** of a face is a cyclic order of adjacent vertices.
- We define the ***front face*** as a counter clockwise order.
- The orientation of two adjacent faces are ***compatible*** if the two vertices of the shared edge are in opposite order.
- A manifold is **orientable** if all adjacent faces are compatible

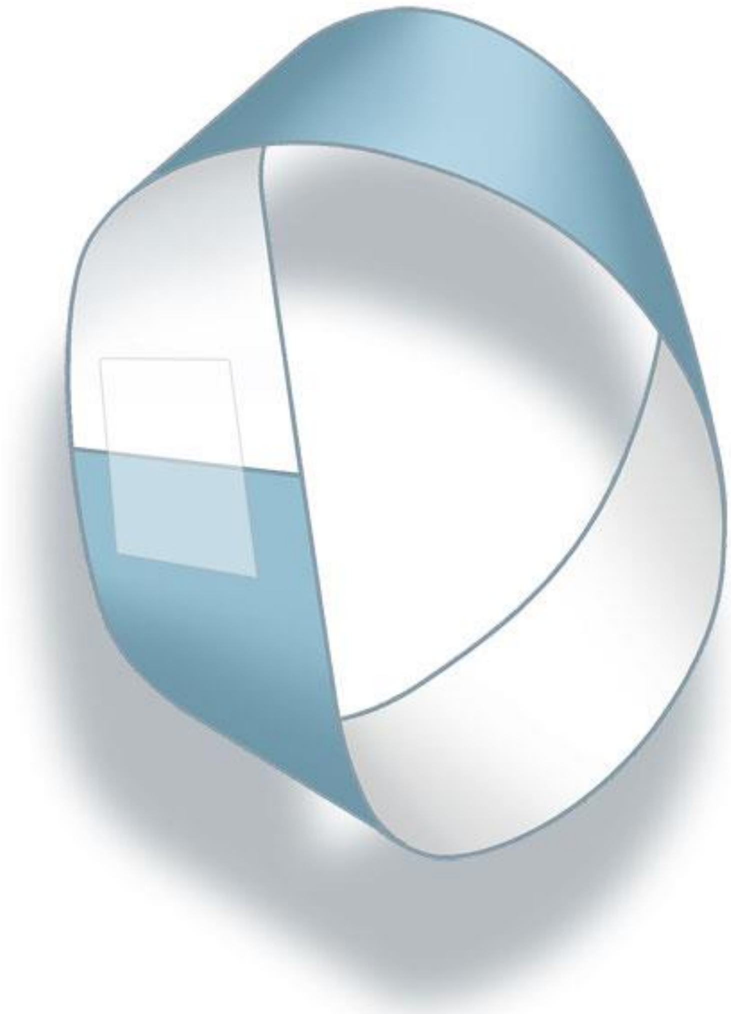






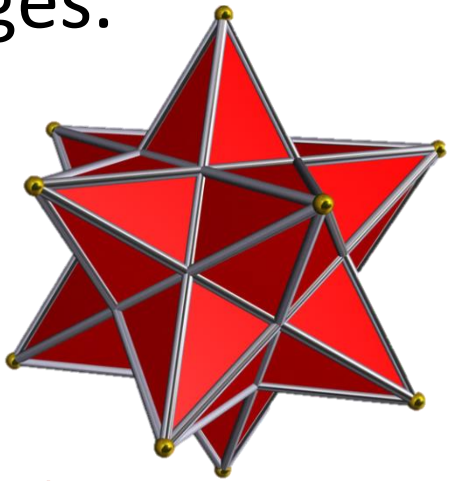
Non-Orientable Manifold

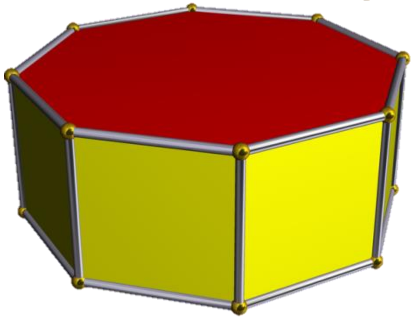
- A non-orientable manifold will have its front surface connected to its back surface.
- Example: Mobius strip.



Polyhedron

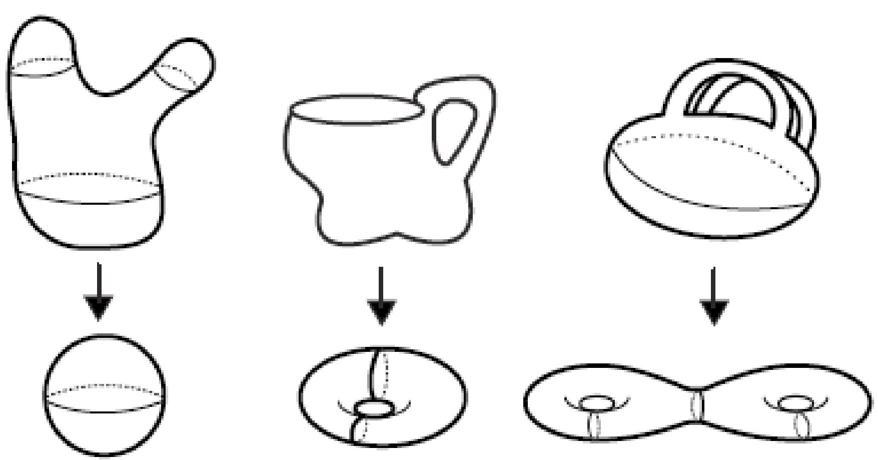
- A ***polyhedron*** is a closed orientable manifold (i.e., no boundaries), and *represents a volume*.
- Made up of flat faces and straight edges.
- Can be convex or non-convex.
- Meshes, by definition, are not smooth (i.e., they have flat faces and sharp edges).
  – We will see smooth surfaces later.





Genus

- The ***genus*** of a connected orientable surface is the maximum number of cuts that can be made along non-intersecting closed simple curves without rendering the resultant manifold disconnected.
  - Can think of genus as the number of “holes”.

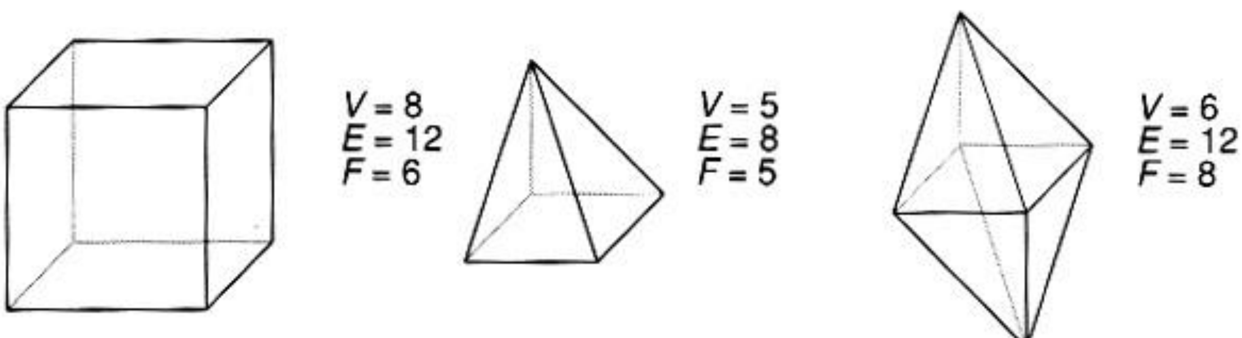


Euler Characteristic

- F = number of faces;
  V = number of vertices;
  E = number of edges.

Could specify ***convex polyhedron,*** because convex forces the mesh to be genus 0, while specifying a polyhedron forces the mesh to be orientable manifold without boundary!

- Euler: V – E + F = 2 for a ***zero genus polyhedron***
  - In general, it sums to small integer (more on next slide)
  - For triangles, have F:E:V is about 2:3:1 (more on this later)



[Foley et al.]

Euler Characteristic

- Generalization of Euler characteristic for orientable manifolds ***with*** boundary:

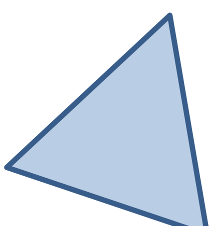
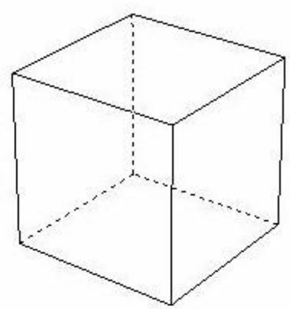
$$\underbrace{V - E + F}_{\chi} + 2g + \#\delta = 2$$

Number of boundaries

Genus

Euler Characteristic

- Try these examples:


- Consider a mesh, which is a closed orientable manifold (i.e., no boundary), and is defined entirely with quadrilaterals using 100 vertices, with each vertex having exactly degree 4.
  - How many edges are there?
  - How many faces are there?
  - What is the genus of the mesh?

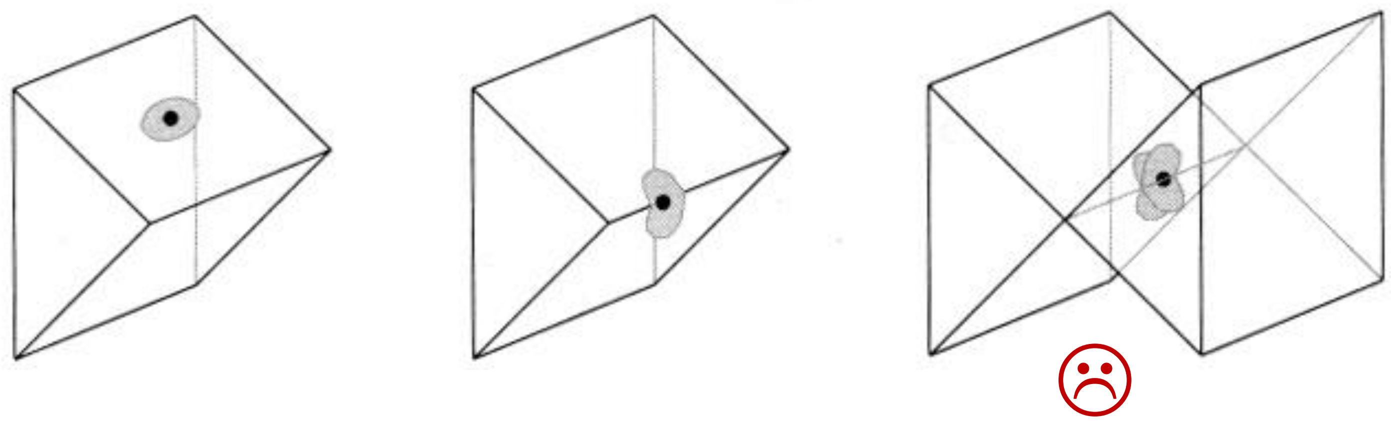
- A classic soccer ball has only pentagons and hexagons, and each vertex has degree three. How many pentagons does a soccer ball have? How many hexagons can a soccer ball have? Hint: assume that there are x pentagons and y hexagons and use the Euler Characteristic.

Validity of Triangle Meshes

- In many cases we care about the mesh being able to nicely bound a region of space.
- 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

Topological Validity

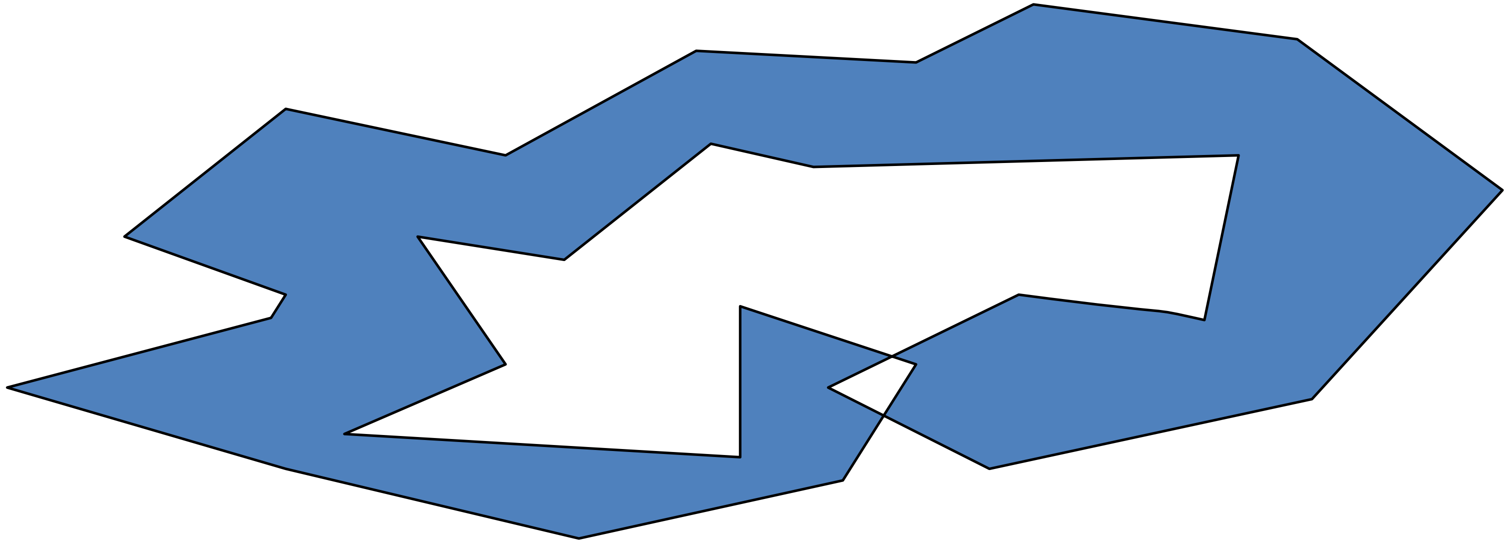
- Strongest property, and most simple: be a manifold
  - This means that no points should be "special"
  - Interior points are fine
  - Edge points: each edge should have exactly 2 triangles
  - Vertex points: each vertex should have one loop of triangles
    - not too hard to weaken this to allow boundaries



[Foley et al.]

Geometric Validity

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



Mesh Terminology Review

vertices
edges
faces
connectivity
topology
incidence
geometry
manifold

non-manifold
boundary
interior
triangle fan
face orientation
front face
back face
compatible triangles

orientable manifold
polyhedron
genus
convex polyhedron
Euler Characteristic
topological validity
geometric validity

2 Triangles/Vertex on average… Why?

- First let us ask how or if we can create a ***regular*** tiling of a surface with **n-gons**.
  - ***Regular*** means we want each vertex to have the same number of incident edges. This is known as ***degree*** or ***valence***.
  - We call a polygon with n sides an ***n-gon***.
- Note that the we are not concerned about **regular n-gons** (all sides and angles equal), and we'll focus on topology for now…

Topological approach to exploring options

- Zero genus (sphere-like) object with n-gons and with regular degree k vertices?
  - Each edge has 2 vertices, each vertex has k edges

$$2E = kV$$

  - Each edge has 2 faces, each face has n edges

$$2E = nF$$

  - Convex polyhedron must satisfy V-E+F=2

$$\frac{2}{k}E - E + \frac{2}{n}E = 2$$

$$\frac{1}{k} + \frac{1}{n} = \frac{1}{2} + \frac{1}{E} \qquad \frac{1}{k} + \frac{1}{n} > \frac{1}{2}$$

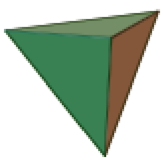
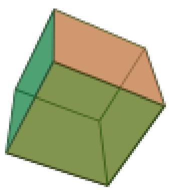
always positive

Topological approach to exploring options

- For what k and n is $\frac{1}{k} + \frac{1}{n} > \frac{1}{2}$?

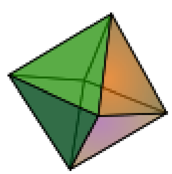
Note that k and n must be both at least 3

k=3 ? Need 1/n > 1/2 – 1/3 = 1/6

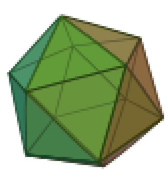

n=3,4,5 will work

k=4 ? Need 1/n > 1/2 – 1/4 = 1/4



n=3 will work

k=5 ? Need 1/n > 1/2 – 1/5 = 3/10



n=3 will work

We can work out the face count for each case using the Euler characteristic on the previous slide…

- Tetrahedron (4)
- Cube (6)
- Dodecahedron (12)
- Octagon (8)
- Icosahedron (20)

No other options exist!

Topological approach to exploring options

- Genus 1 (torus-like) object with n-gons and with regular degree k vertices?
  - Must satisfy V-E+F=0, i.e., $\frac{2}{k}E - E + \frac{2}{n}E = 0$
  - Edge count doesn't matter, can rearrange to get

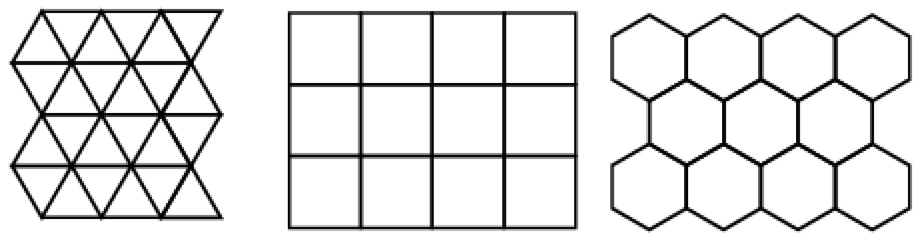
$$2n - nk + 2k = 0 \quad \rightarrow \quad k = \frac{2n}{n-2}$$

n=3 (triangles) k = 6

n=4 (quadrilaterals) k = 4

n=6 (hexagons) k = 3

No other integer solutions!



Thus, on average, large regular meshes of triangles, quads, hexagons have, 2, 1, and 0.5 faces per vertex respectively

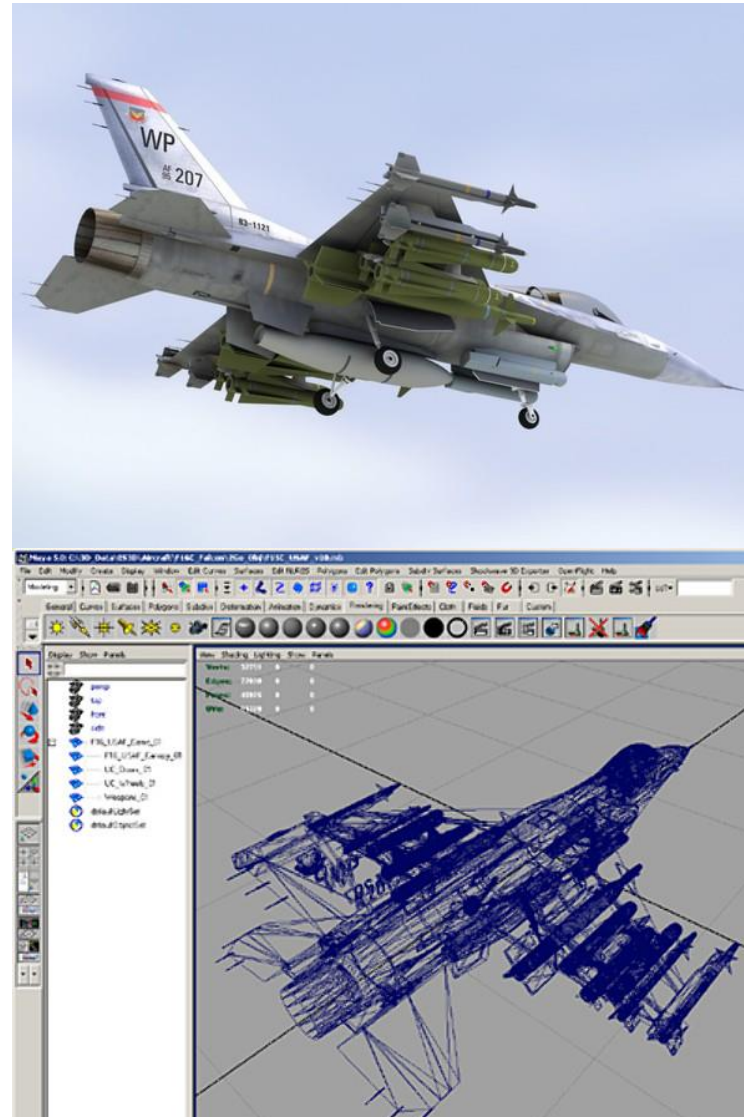
Where does geometry come from?

Where does geometry come from?

Artists

CAD models

[Mudbox]



[ES3DStudios]

Where does geometry come from?

Measurement

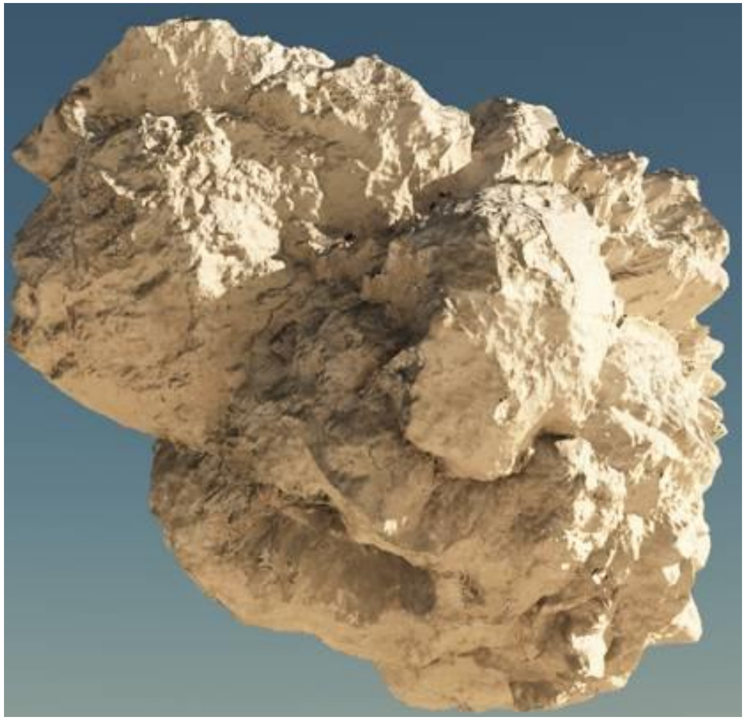
- Multi camera stereo
- Laser scans
- Medical scans

[Beeler et al 2010]

Where does geometry come from?

Procedural

- Noise
- Fractals
- L-systems
- Scripts

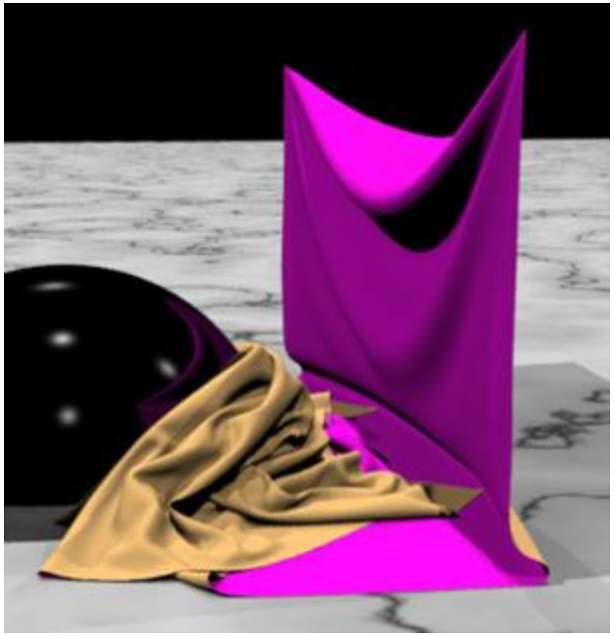


Where does geometry come from?

Physically Based Modeling

- Shapes produced through physics simulation

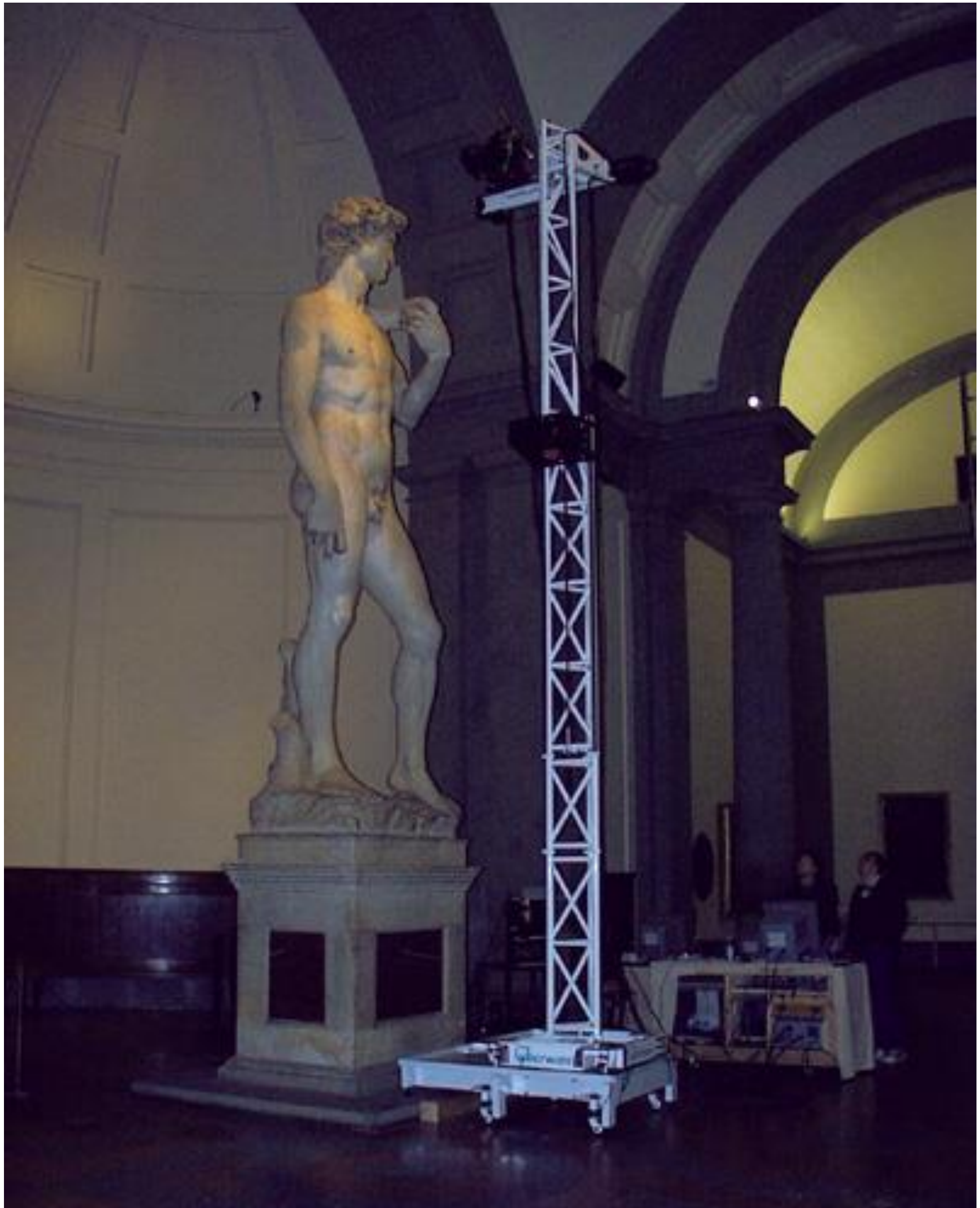
[Thürey et al. 2010]

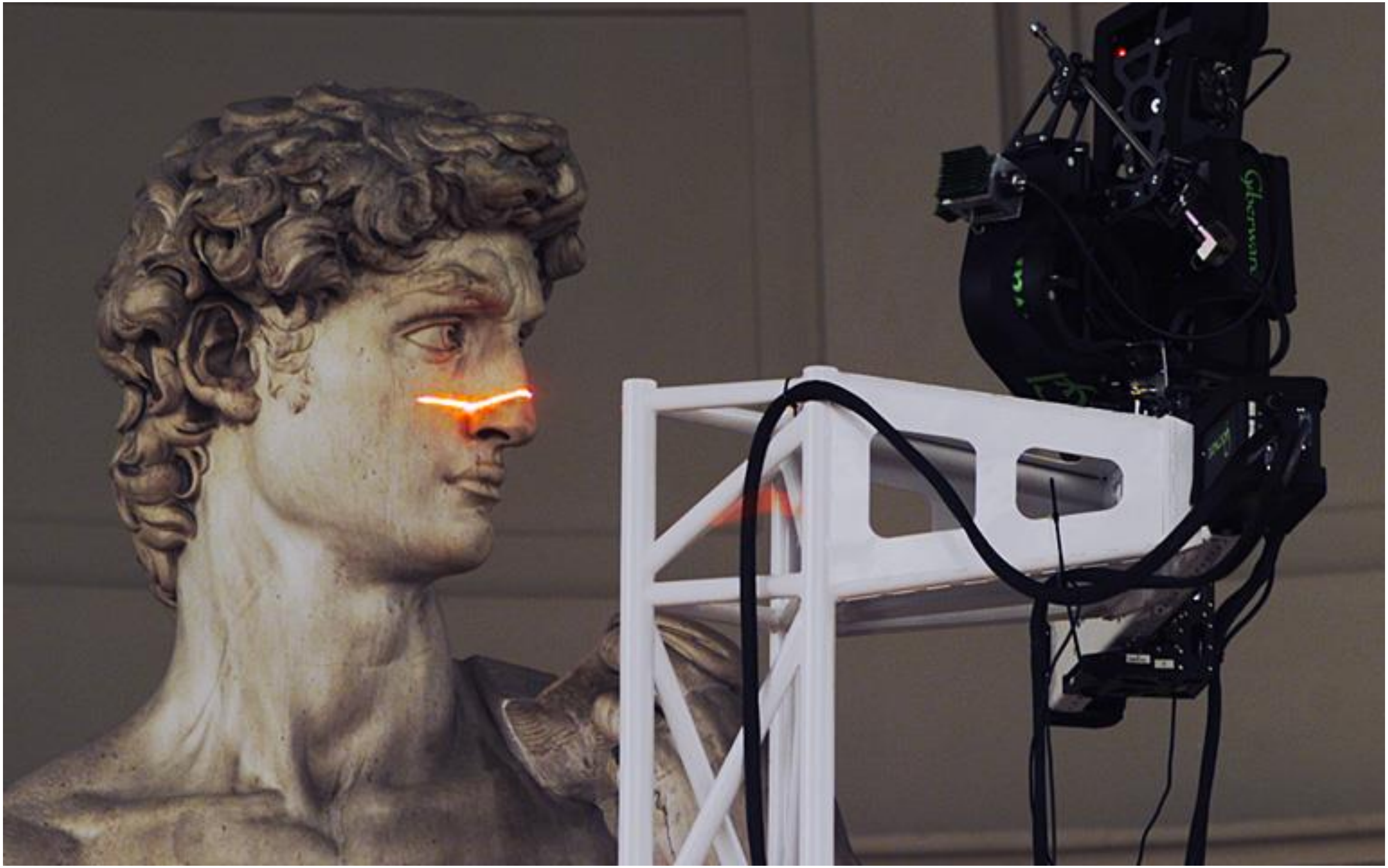


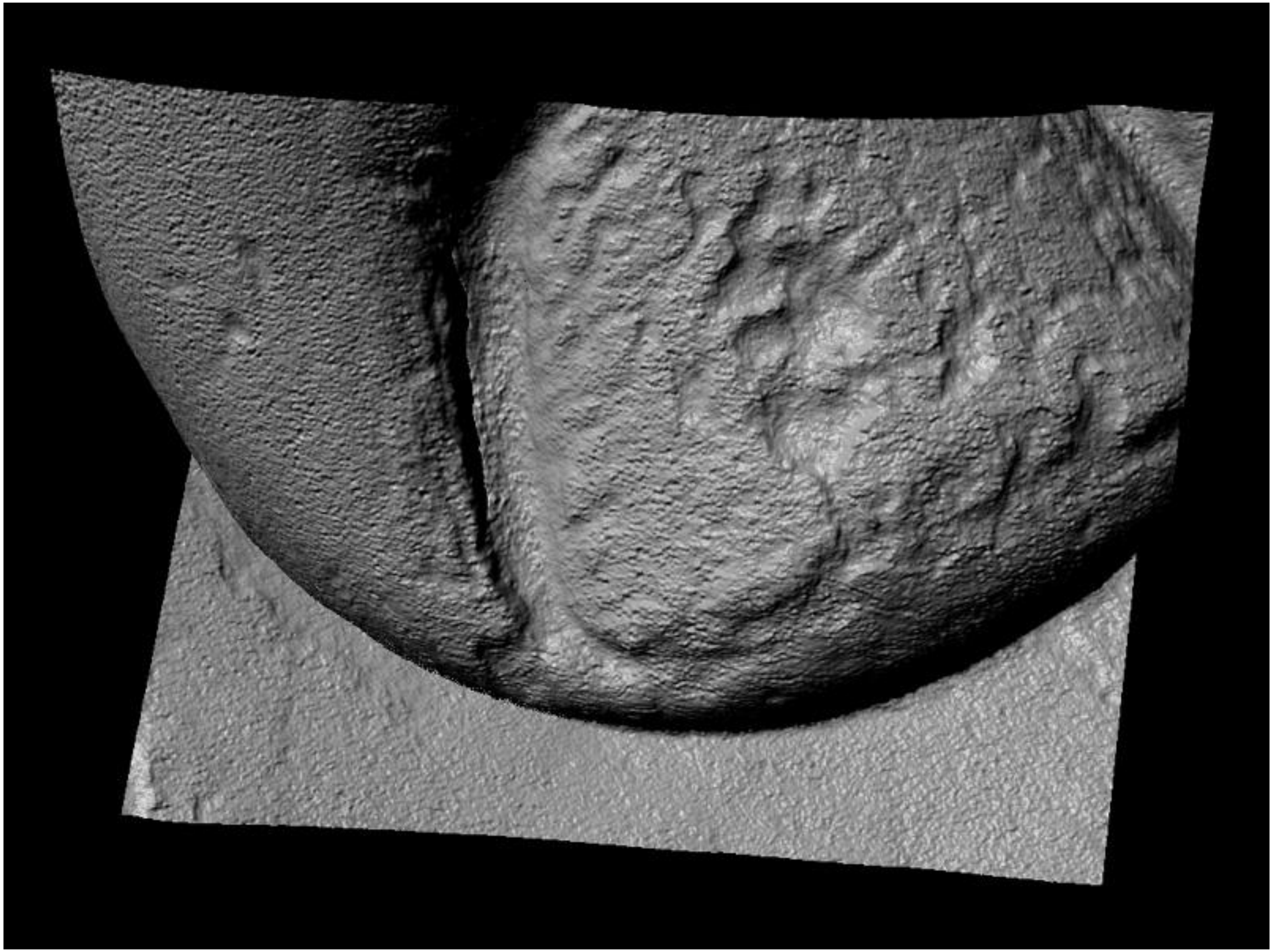
[Bridson et al. 2002]

More Issues

- Where do meshes come from?
  - Artists
    - Not always manifold
    - Often quadrilaterals
  - Scans (Laser, MRI, CAT, etc…)
    - Huge numbers of points
    - Complicated triangulation problems
    - Noise and topology problems
    - Level of detail problem for rendering (more on this later)
  - Procedural and physically based
    - May not be able to guarantee geometric validity
    - May also have topological issues







Representation of Triangle Meshes

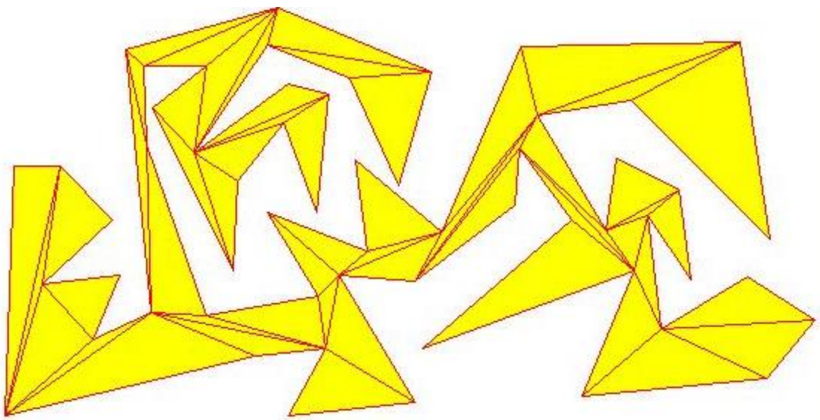
- Compactness
- Efficiency for rendering
  - enumerate all triangles as triples of 3D points
- Efficiency of queries
  - all vertices of a triangle
  - all triangles around a vertex
  - neighbouring triangles of a triangle
  - (need depends on application)
    - finding triangle strips
    - computing subdivision surfaces
    - mesh editing
- Question: what information might we care about?

Representations for triangle meshes

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

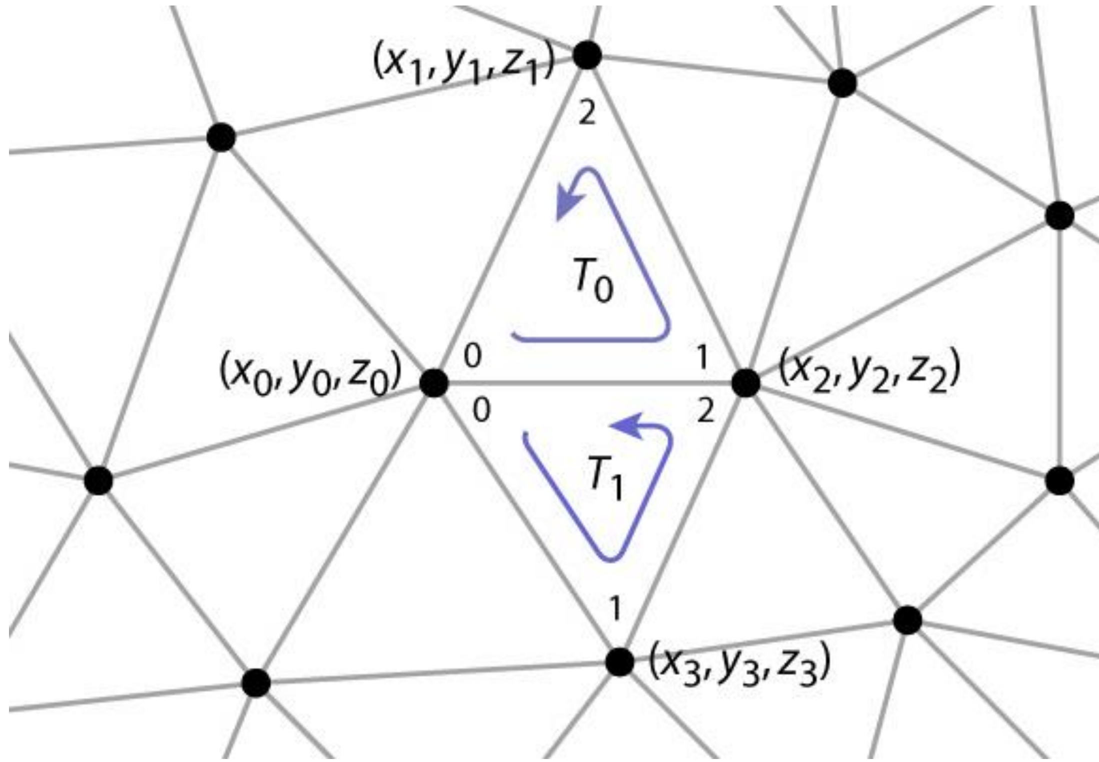
Issues

- Non triangular meshes?
  - Enforce planarity of non triangular faces?
  - Breaking up polygons into triangles for processing?
    - Tessellation / Triangulation
    - Tricky for non-convex shapes



Separate triangles

	[0]	[1]	[2]
tris[0]	x_0, y_0, z_0	x_2, y_2, z_2	x_1, y_1, z_1
tris[1]	x_0, y_0, z_0	x_3, y_3, z_3	x_2, y_2, z_2
	⋮	⋮	⋮



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 round-off
  - difficulty of finding neighbours at all

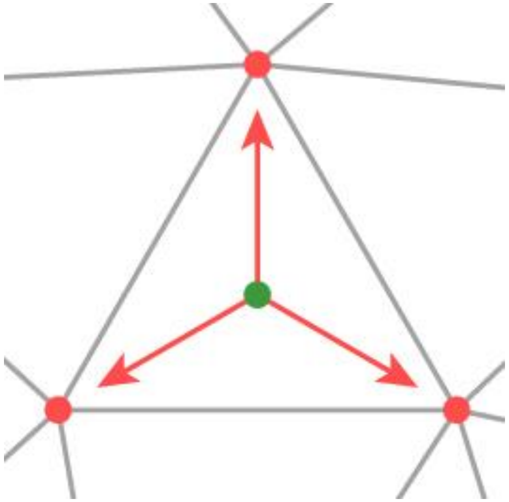
Indexed triangle set

- Store each vertex once
- Each triangle points to its three vertices

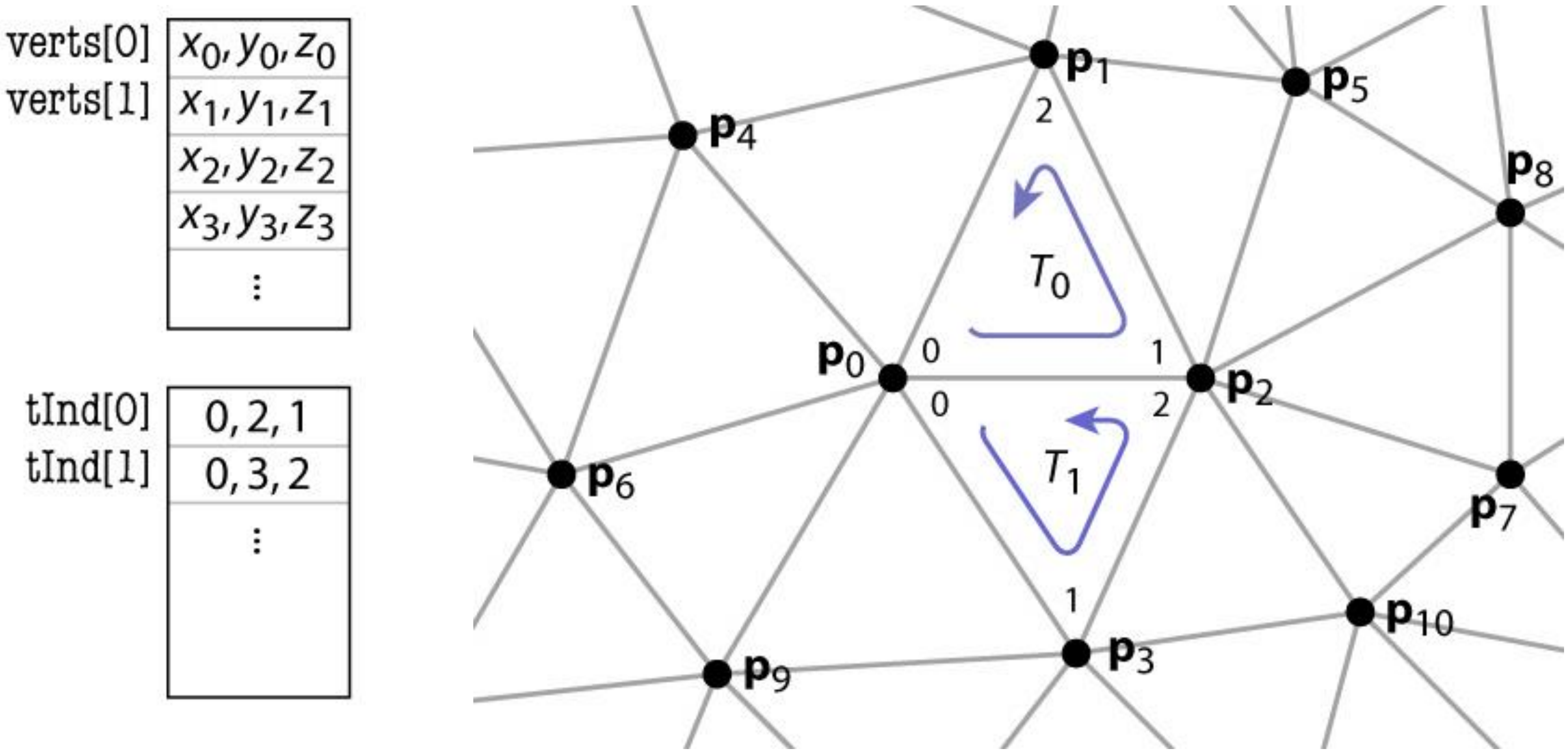
```
Triangle {
   Vertex vertex[3];
}

Vertex {
   float position[3];  // or other data
}
```

// … or …

```
Mesh {
   float verts[nv][3];  // vertex positions (or other data)
   int tInd[nt][3];  // vertex indices
}
```



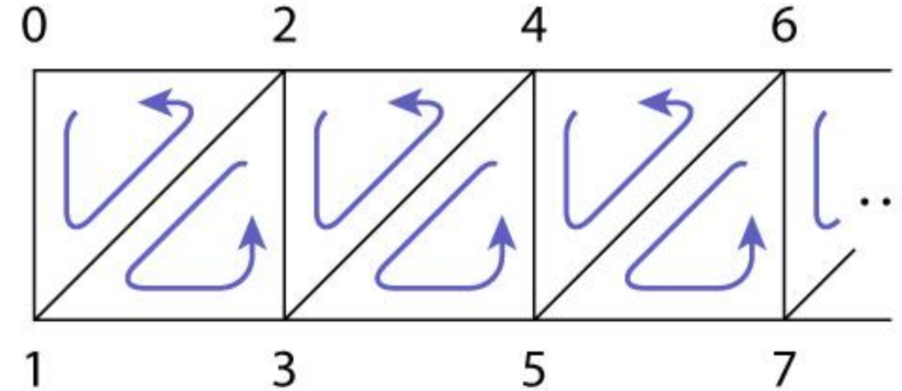
Indexed triangle set



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 neighbours is at least well defined

Triangle strips

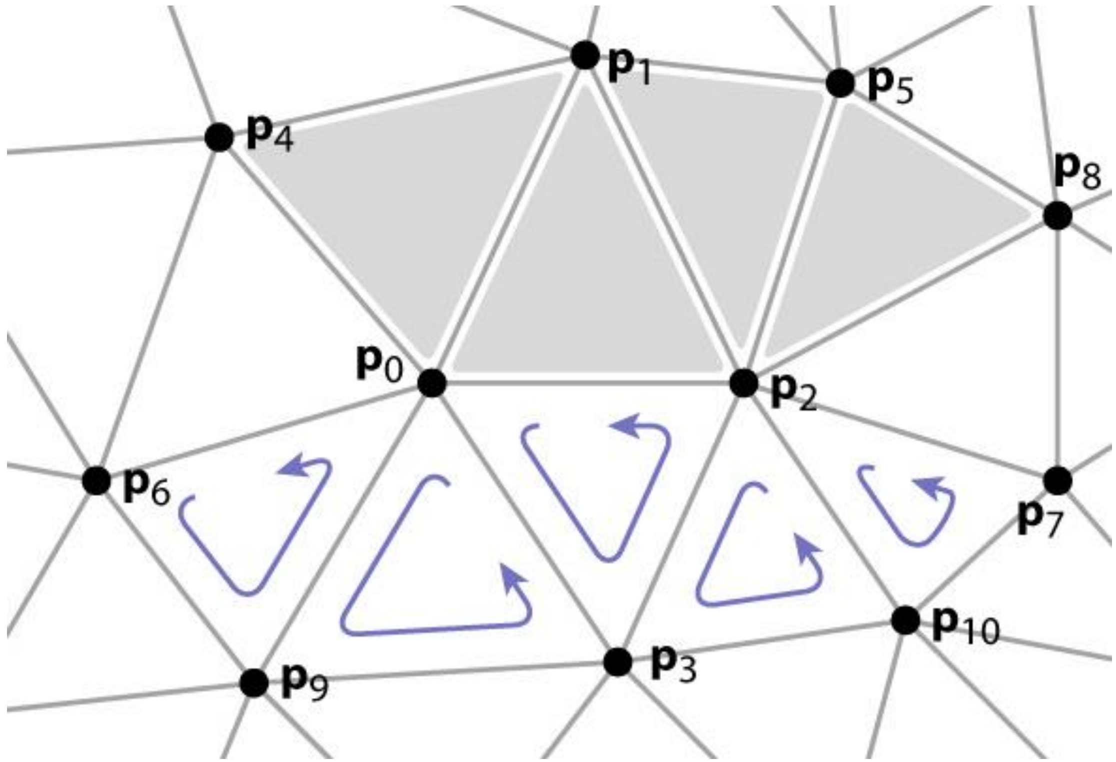


- Take advantage of the mesh property
  - each triangle is usually adjacent to the previous
  - let every vertex create a triangle by ***reusing two of the vertices*** of the previous triangle
  - every sequence of three vertices produces a triangle (but not in the same order)
  - e. g., 0, 1, 2, 3, 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
	⋮

tStrip[0]	4, 0 , 1, 2, 5, 8
tStrip[1]	6, 9, 0, 3, 2, 10, 7
	⋮

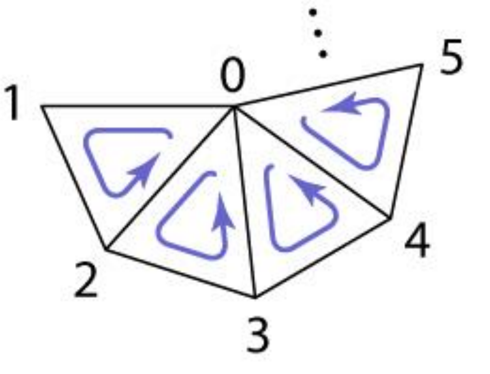


Triangle strips

- 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 + ∑) 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

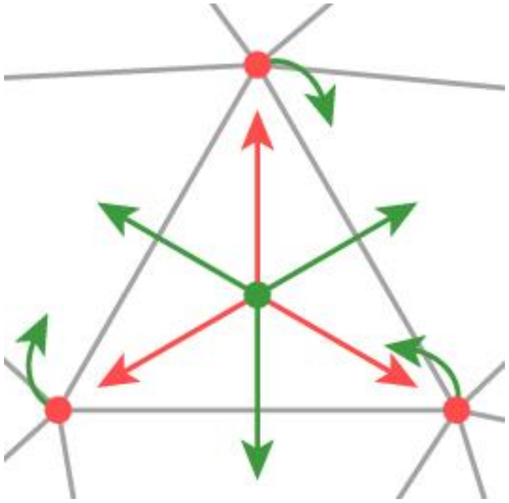


Drawing triangles in OpenGL

- Put per vertex data and indices into buffers
  - glGenBuffer(…)
  - glBindBuffer(GL_[ARRAY,ELEMENT_ARRAY]_BUFFER, …)
  - glBufferData(…)
  - glEnableClientState(GL_[VERTEX,NORMAL,…]_ARRAY)
  - gl[Vertex,Normal,…]Pointer(…)
  - glDrawElement(GL_TRIANGLES, …)
- https://www.opengl.org/wiki/VBO_-_just_examples

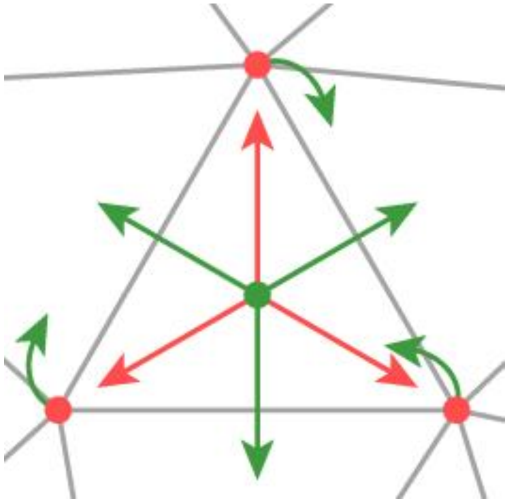
Triangle neighbour structure

- Extension to indexed triangle set
- Triangle points to its three neighbouring triangles
- Vertex points to a single neighbouring triangle
- Can now enumerate triangles around a vertex



Triangle neighbour structure

```
Triangle {
   Triangle nbr[3];
   Vertex vertex[3];
}
// t.neighbour[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 neighbour triangles
   int vTri[nv]; // index of any adjacent triangle
}
```

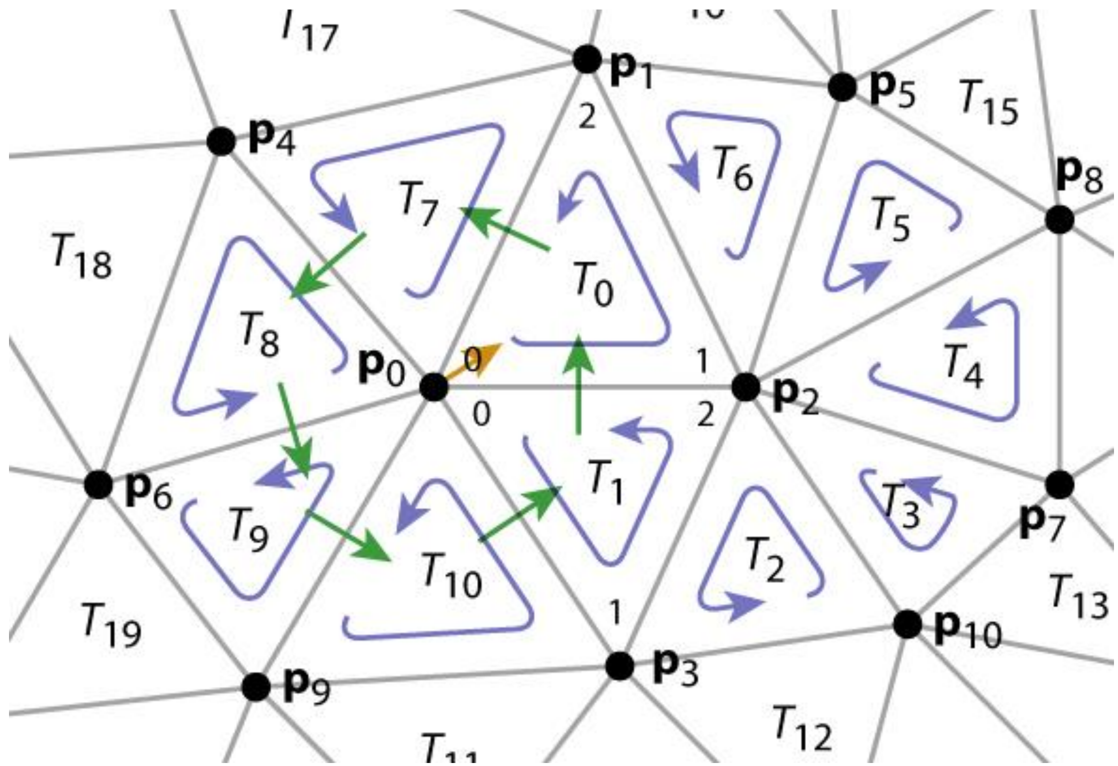


Triangle neighbour structure

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

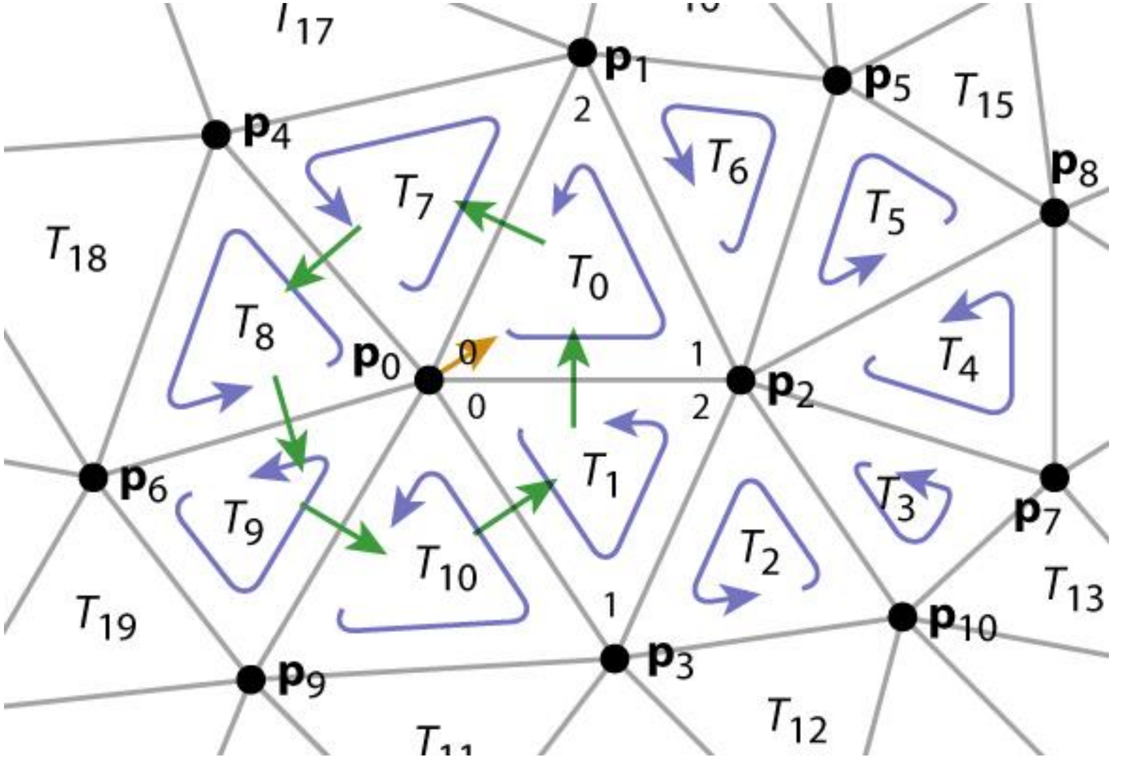
tNbr[0]	1, 6, 7
tNbr[1]	10, 2, 0
tNbr[2]	3, 1, 12
tNbr[3]	2, 13, 4
	⋮

tInd[0]	0, 2, 1
tInd[1]	0, 3, 2
tInd[2]	10, 2, 3
tInd[3]	2,10,7
	⋮



Triangle neighbour 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 neighbour structure

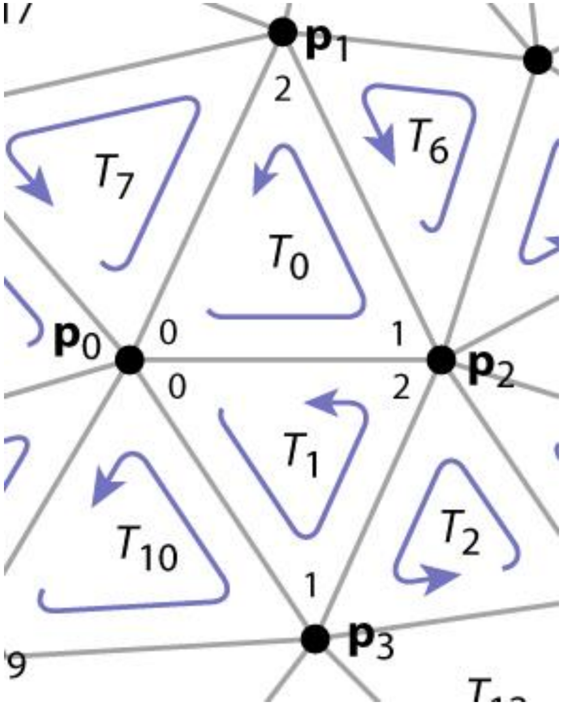
- indexed mesh was 36 bytes per vertex
- add an array of triples of indices (per triangle)
  - int[n_T][3]: about 24 bytes per vertex
    - 2 triangles per vertex (on average)
    - (3 indices x 4 bytes) per triangle
- add an array of representative triangle per vertex
  - int[n_V]: 4 bytes per vertex
- total storage: 64 bytes per vertex
  - still not as much as separate triangles

Triangle neighbour structure—refined

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

// if t.nbr[k].i == j
// then t.nbr[k].t.nbr[j] == t

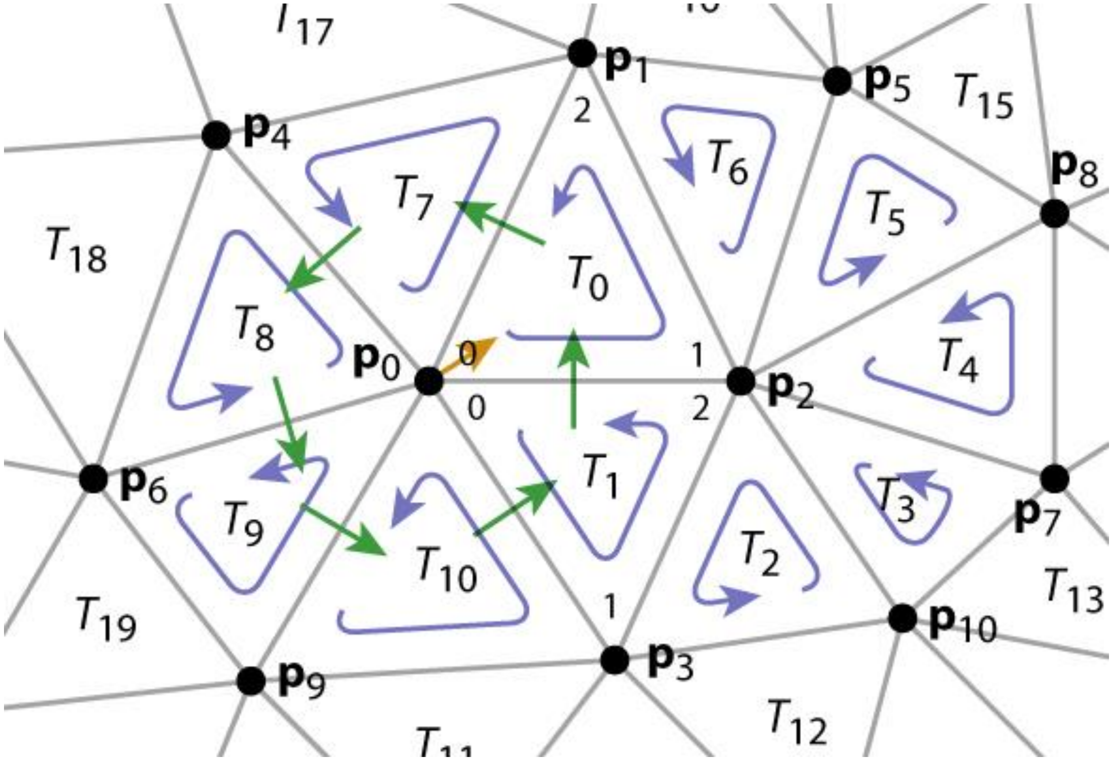
```
Edge {
    // the i-th edge of triangle t
    Triangle t;
    int i;  // in {0,1,2}
    // in practice t and i share 32 bits
}

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



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

Triangle neighbour structure

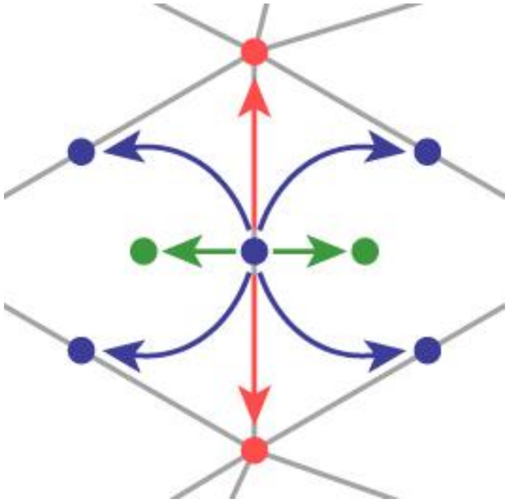
```
TrianglesOfVertex(v) {
   {t, i} = v.e;
   do {
      {t, i} = t.nbr[pred(i)];
   } while (t != v.t);
}

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 }

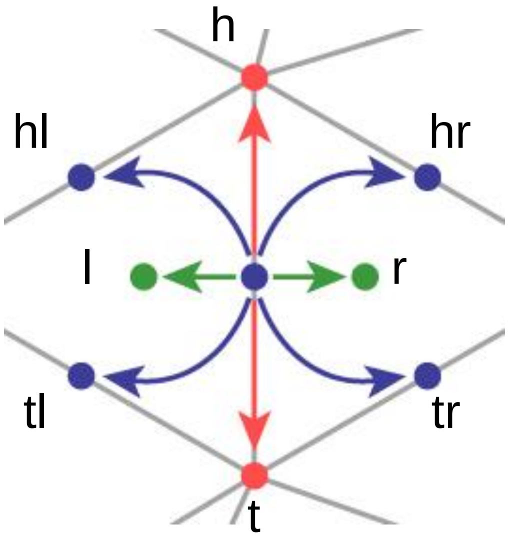
Winged-edge mesh

- 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 {
   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
}
```

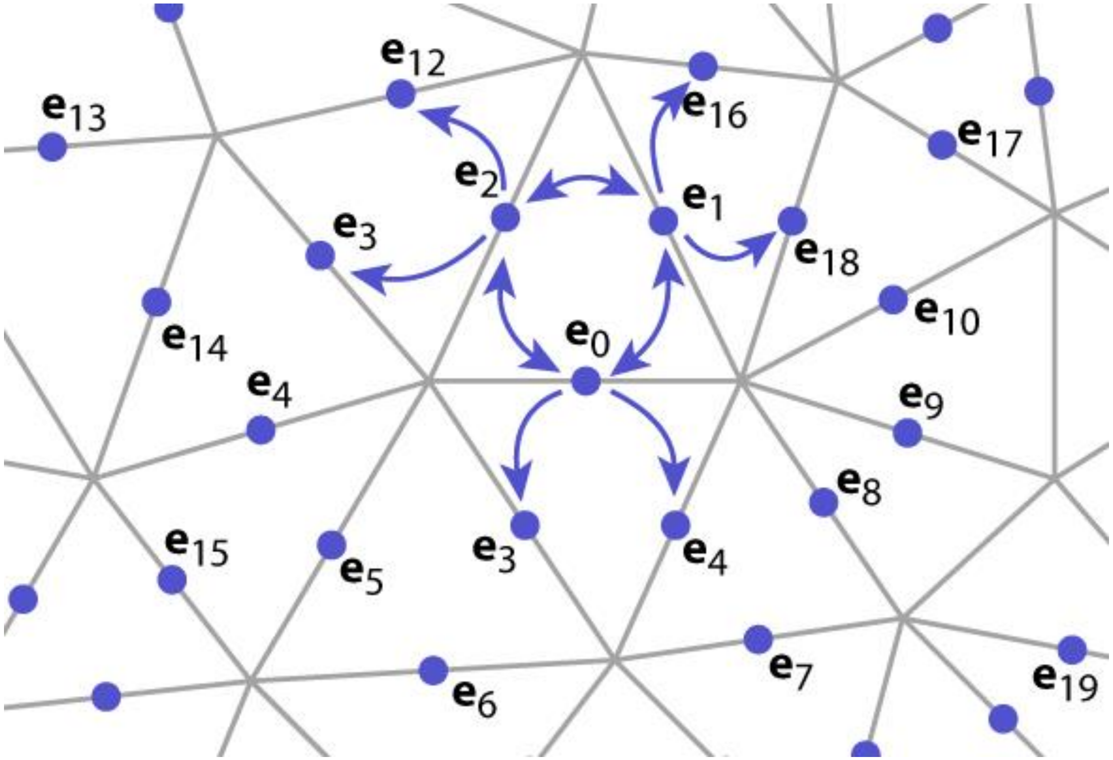


(think ***head*** and ***tail*** for ***h*** and ***t***, likewise, ***left*** and ***right*** for ***l*** and ***r***)

Winged-edge structure

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

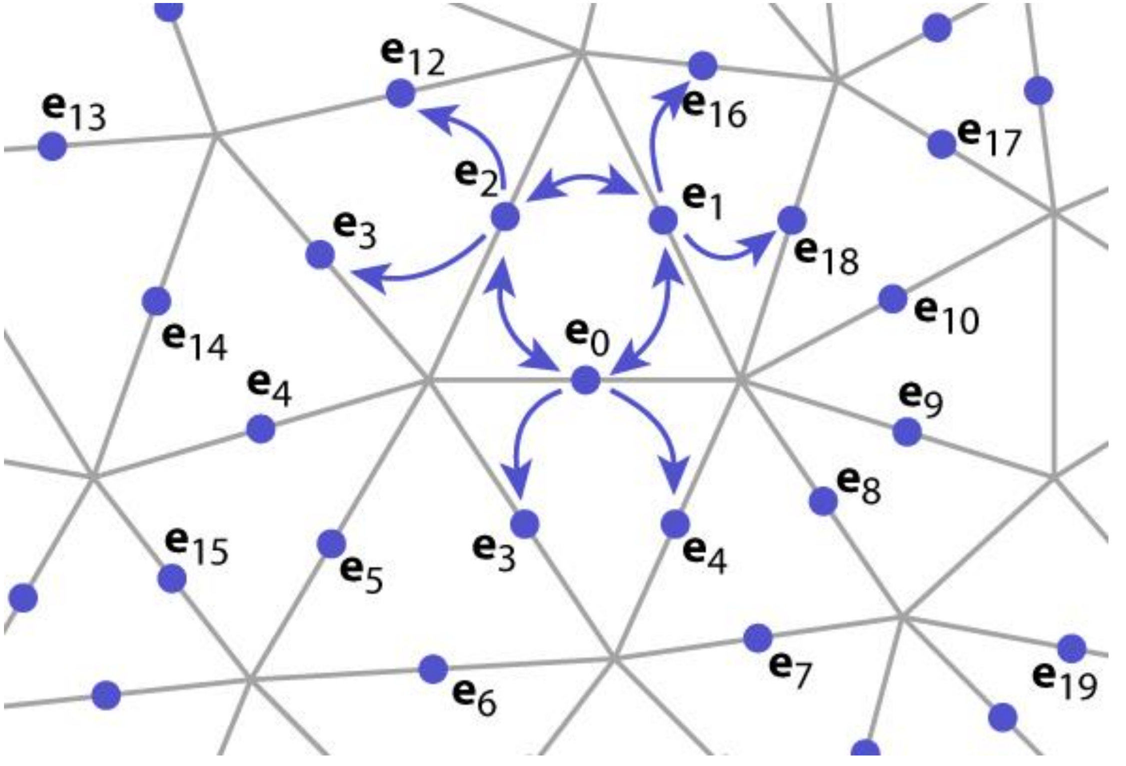
	hl	hr	tl	tr
edge[0]	1	4	2	3
edge[1]	18	0	16	2
edge[2]	12	1	3	0
		⋮		



Winged-edge structure

```
EdgesOfFace(f) {
    e = f.e;
    do {
        if (e.l == f) {
            e = e.hl;
        } else {
            e = e.tr;
        }
    } while (e != f.e);
}
```

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

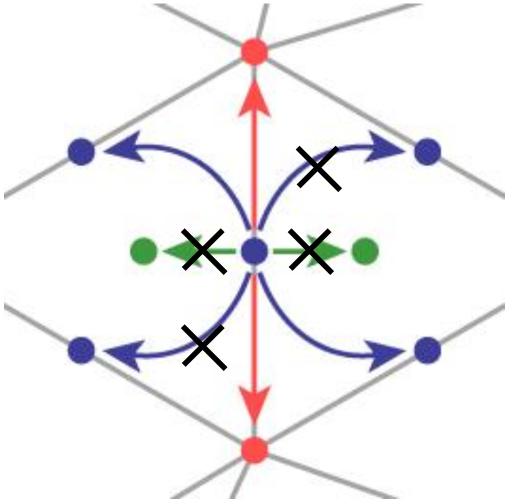


Winged-edge structure

- 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

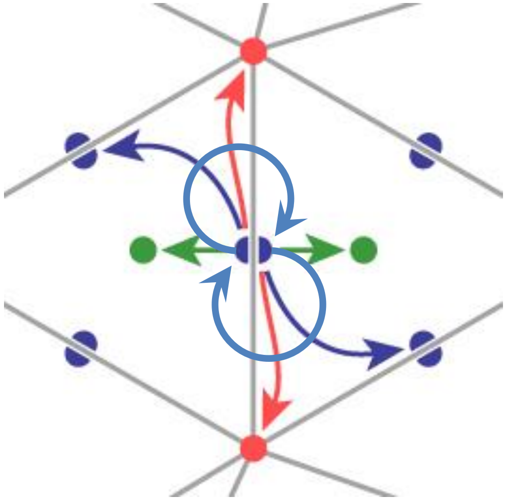
Winged-edge optimizations

- Omit faces if not needed
- Omit one edge pointer on each side
  - results in one-way traversal



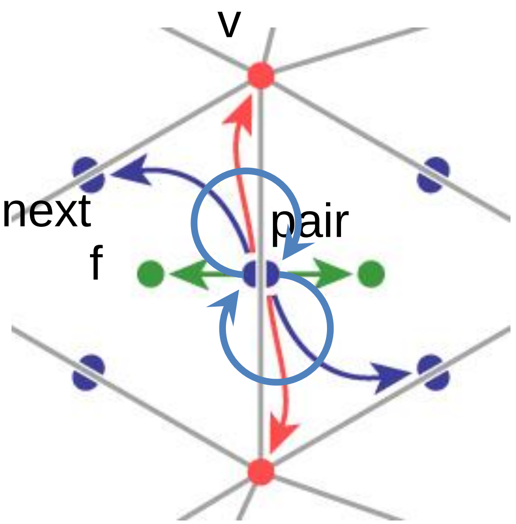
Half-edge structure

- 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

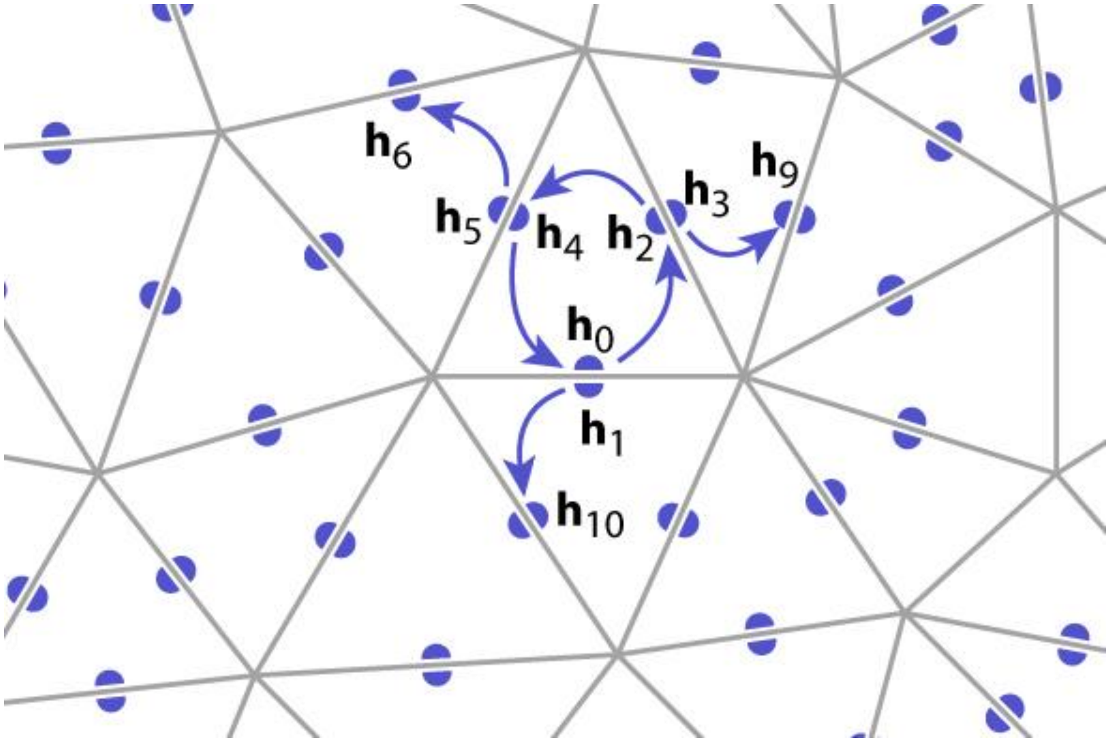
```
HEdge {
   HEdge pair; // also called twin, or opposite
   HEdge next;
   Vertex v;
   Face f;
}

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

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



Half-edge structure

```
EdgesOfFace(f) {
    h = f.h;
    do {
        h = h.next;
    } while (h != f.h);
}
```

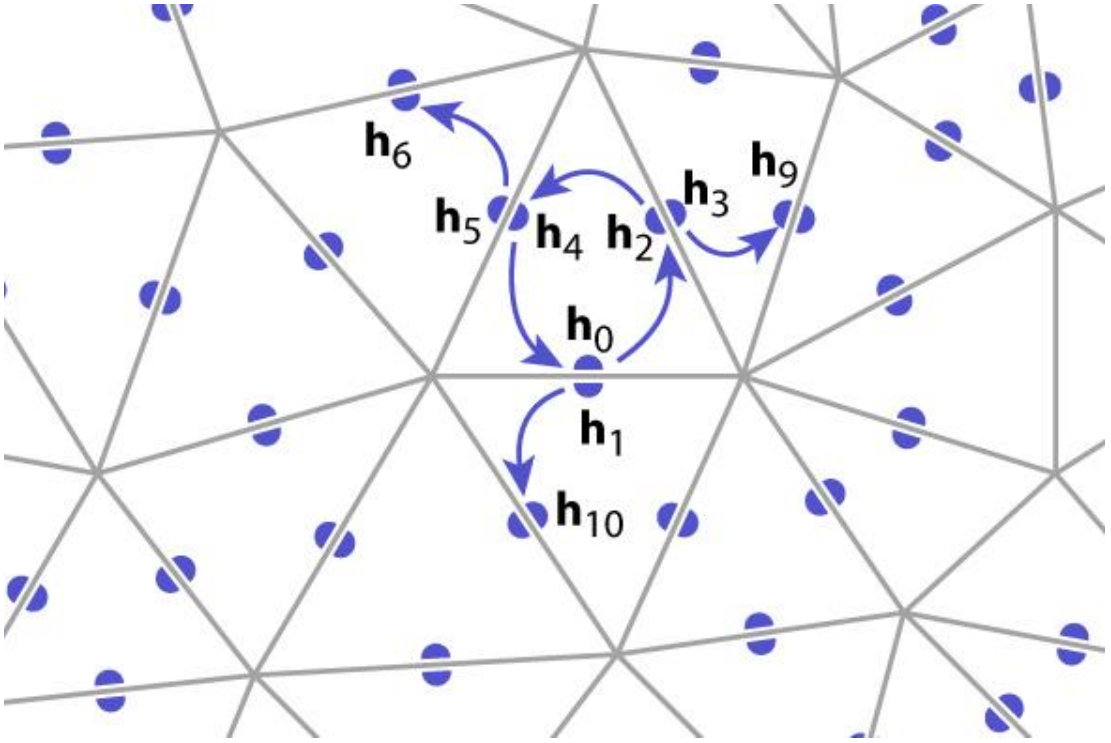
	pair	next
hedge[0]	1	2
hedge[1]	0	10
hedge[2]	3	4
hedge[3]	2	9
hedge[4]	5	0
hedge[5]	4	6
	⋮	



Half-edge structure

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

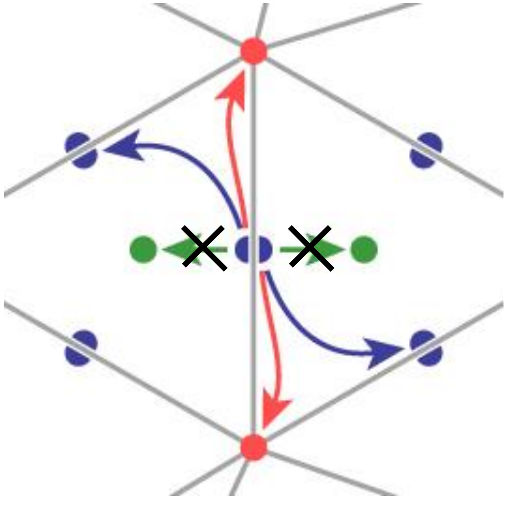


Half-edge structure

- 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

Half-edge optimizations

- Omit faces if not needed
- Use implicit pair pointers
  - they are allocated in pairs
  - they can be at even and odd indices in an array



Creating a Half Edge Data Structure

- Common format for storing a mesh on disk is an .obj file

```
v -2 -1 0
v  2 -1 0
v  0  1.732 0
f 1 2 3
```

One-indexed, i.e., indices do not start at zero!

```
v 1 -1 -1
v 1 -1 1
v -1 -1 1
v -1 -1 -1
v 1 1 -1
v 1 1 1
v -1 1 1
v -1 1 -1
f 5 1 4
f 5 4 8
f 3 7 8
f 3 8 4
f 2 6 3
f 6 7 3
f 1 5 2
f 5 6 2
f 5 8 6
f 8 7 6
f 1 2 3
f 1 3 4
```

- How do you create a HEDS from a polygon soup?