UiO : University of Oslo

INF3320: Computer Graphics and Discrete Geometry

Polygonal Meshes

Christopher Dyken and Martin Reimers
Corrections and additions by André R. Brodtkorb

October 15, 2014

Polygonal Meshes

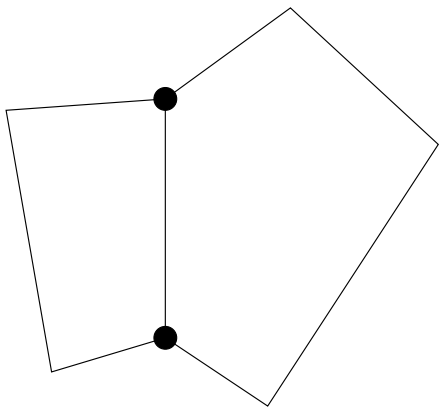
Akenine-Möller and Haines:

- Polygonal Techniques (Chapter 12)
- Sources of three-dimensional data (Chapter 12.1)
- Tessellation and triangulation (Chapter 12.2)
- Consolidation (Chapter 12.3)
- Simplification (Chapter 12.5)

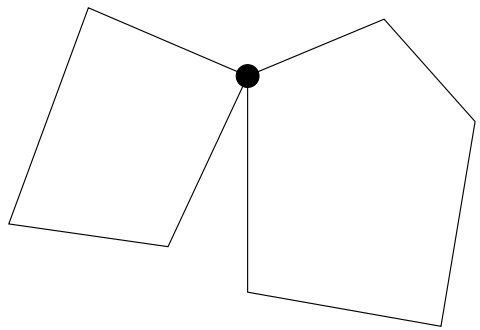
Polygonal meshes

Polygonal meshes

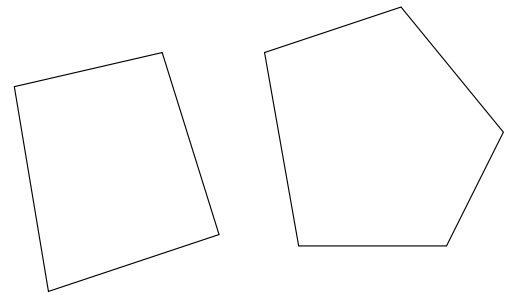
A *polygonal mesh* is a set of faces (polygons) in $\mathbb{R}^3$ such that the intersection between any pair of faces is either



a common edge



a common vertex,

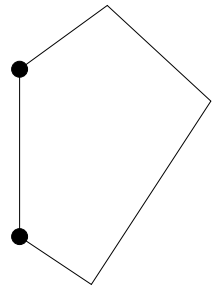


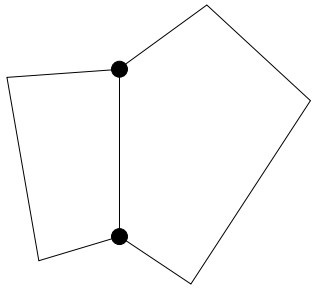
or empty

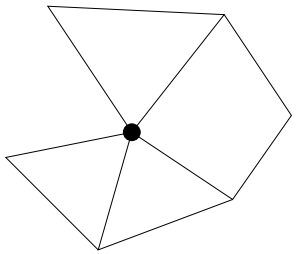
And...

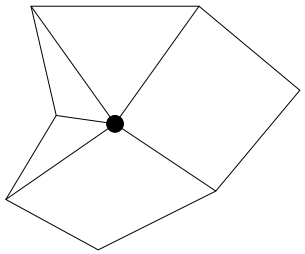
the union of the faces is a *manifold surface*:

Each edge belongs to either one or two faces.









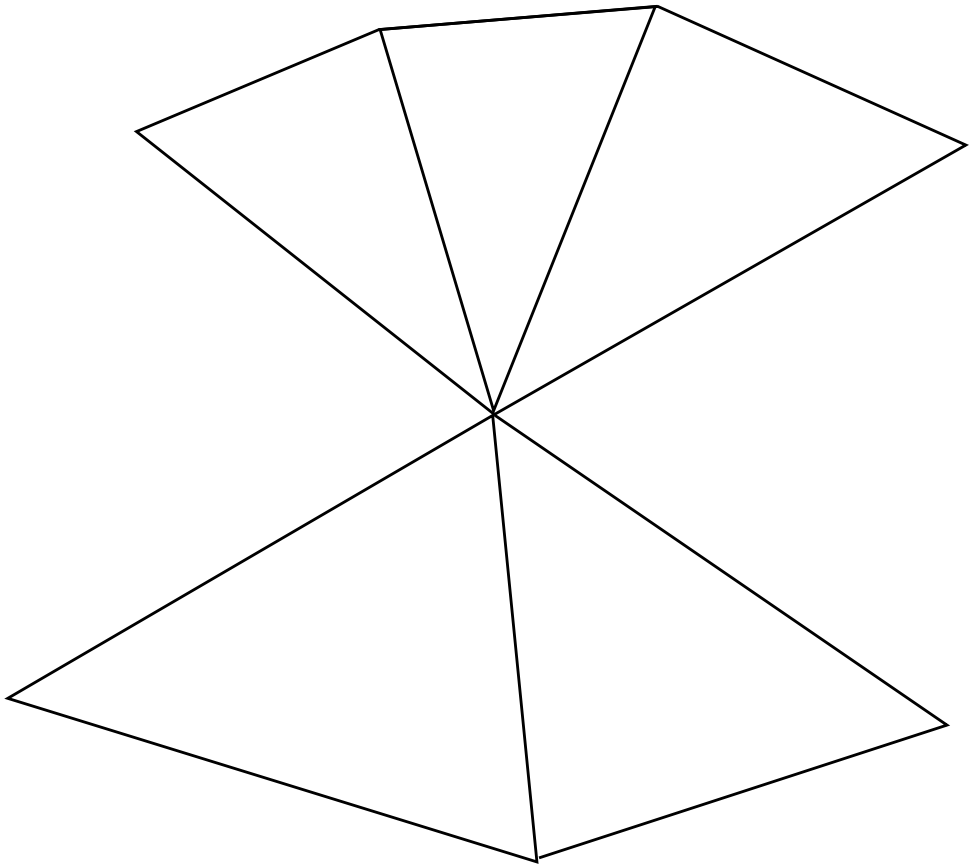
The faces incident on a vertex form an open or closed ‘fan’.

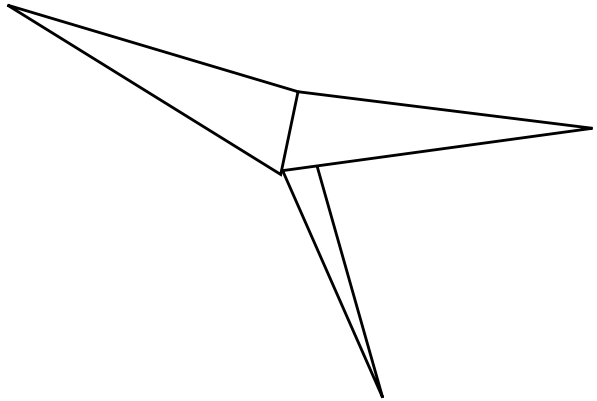
Which implies that the mesh looks locally like a surface.

An edge belonging to only one face lies on the *boundary*.

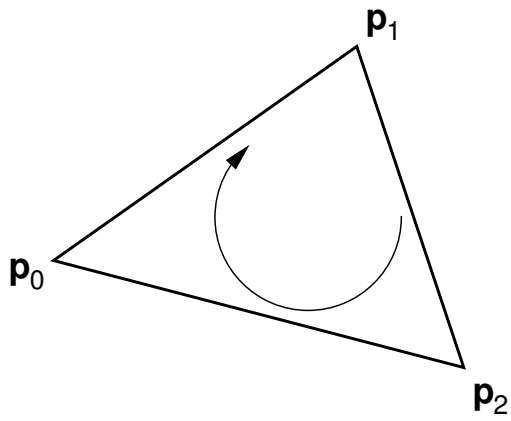
The boundary (if any) consists of one or more loops.

Non-manifold meshes

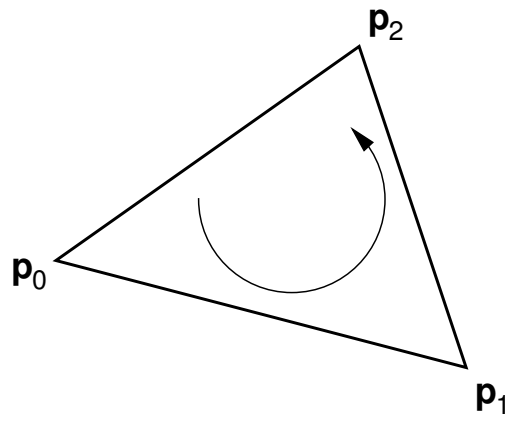




The *orientation* of a face is the cylic order of its incident vertices.

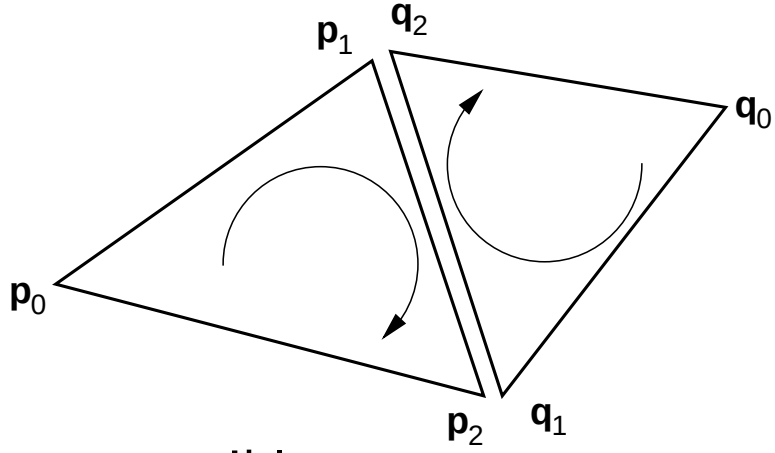


clockwise



counter-clockwise

The orientation of two adjacent faces is *compatible* if the two vertices of their common edge are in opposite order.

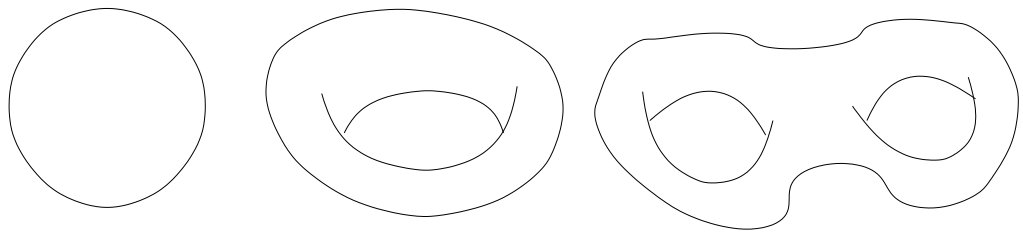


A mesh is said to be *orientable* if there exists a choice of face orientations that makes all pairs of adjacent faces compatible.

The Euler characteristic

The *genus* g of a mesh is the number of handles.

The *sphere*, *torus*, and the *double torus* have genus zero, one, and two.



The genus is also the maximum number cuttings along closed simple curves without disconnecting the surface.

The genus is an intrinsic property that can be found directly by counting the number of vertices, edges, and faces in a mesh.

The *Euler characteristic* denoted χ, is defined by

$$\chi \equiv v - e + f,$$

where v, e, f are the number of vertices, edges, and faces.

It characterizes the *topological type* of a mesh.

The genus and the Euler characteristic are related by

$$\chi = 2 - 2g,$$

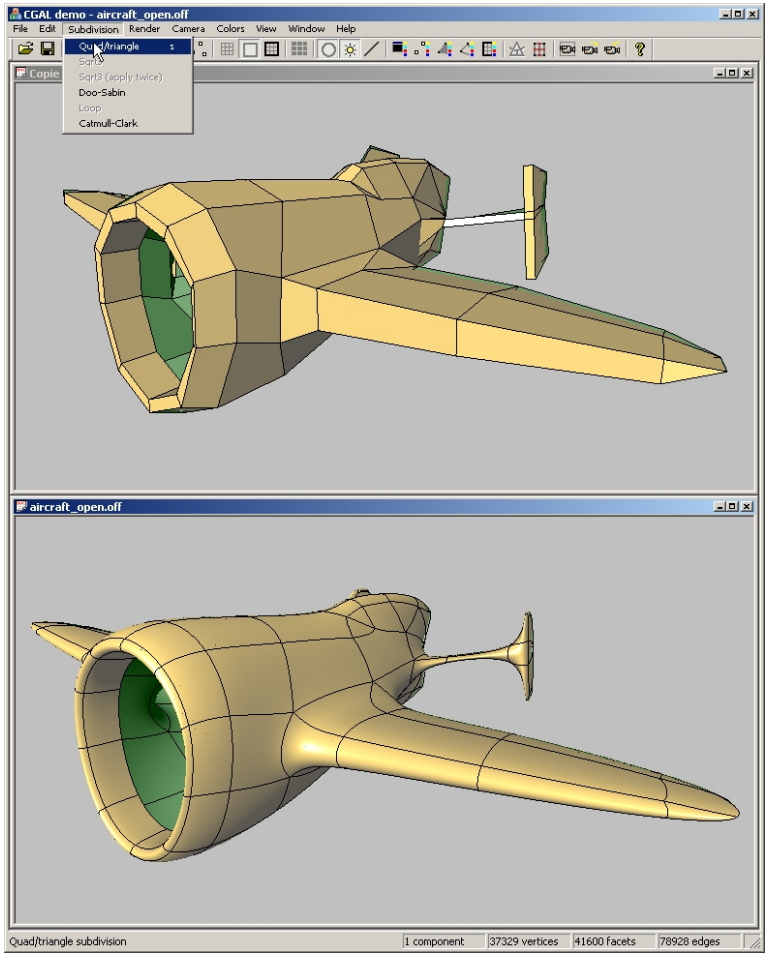
i.e.,

$$v - e + f = 2 - 2g$$

Surface with b boundary components: $\chi = 2 - 2g - b$

Polygonal meshes

A polygonal mesh can consist of different types of faces, typically



- Triangle meshes
- Quadrilateral (quad) meshes
- A mix between the two
- Arbitrary polygons

Data structures for 3D data and triangular meshes

Sources of three-dimensional data

3D data can come from many sources:

- Manual input
- Generating programs
- CAD models
- Digitizers, laser scanners
- Reconstructed from photographs, CT, MR, ...
- ...

3D data are required to model reality - great many applications

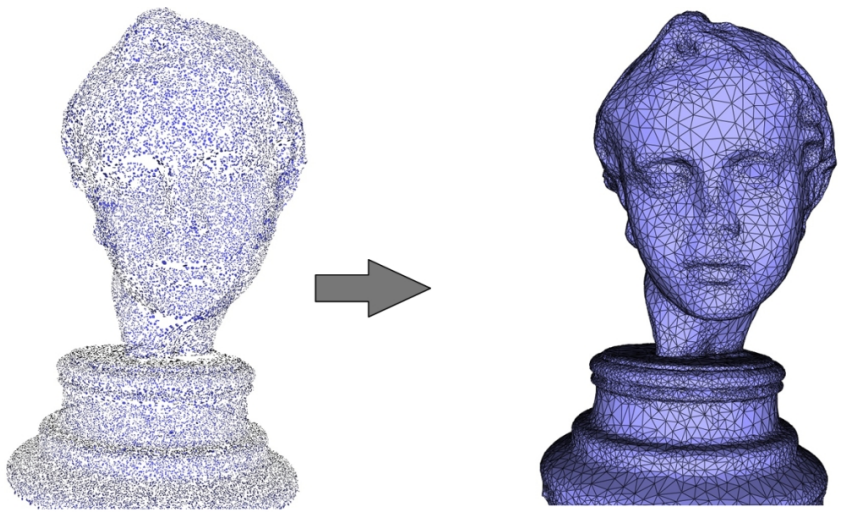
Organizing and manipulating geometric data remains a challenge and a big field of research

Point clouds

A *point cloud* is a set of points in 3D without topological information (no edges/triangles)

Rendering a point cloud is quite easy, draw vertices in OpenGL, or use more advanced techniques such as splatting

Organizing a point cloud into a surface mesh is a different matter ...



[CGAL]

Polygon soups

A *Polygon soup*[1] is a list of polygons, where each polygon contains the position of its vertices. If the model only contains triangles, it is called a *triangle soup*.

```
facet normal 0.00 0.00 1.00
outer loop
vertex  2.00  2.00  0.00
vertex -1.00  1.00  0.00
vertex  0.00 -1.00  0.00
endloop
endfacet
```

Rendering a triangle soup is quite easy, we simply run through the triangles and throw the vertices at OpenGL.

[1] The STL file format uses polygon soup to describe geometry.

Polygon soups have some serious drawbacks:

- A vertex is usually specified multiple times.
  $\Longrightarrow$ GPU vertex cache gives no benefit.
- No notion of whether two vertices are equal
- We have no info of which triangle contains a specific vertex.
- We have no info of which triangles are adjacent.

We can create an *indexed face set* from a polygon soup by

- identifying identical vertices (within a tolerance)
- enumerating unique vertices
- specifying geometry using the vertex indices

$\Longrightarrow$ *indexed face set*.

An *indexed face set*[2] is a list of vertices and a list of polygons which indexes the list of vertices.

OpenInventor file representing a tetrahedron:

```
#Inventor V2.1 ascii
VertexProperty { vertex [     0.0        0.0  1.73205,
                              0.0    1.63299 -0.57735,
                         -1.41421 -0.816497 -0.57735,
                          1.41421 -0.816497 -0.57735 ] }
IndexedFaceSet { coordIndex  [ 0, 1, 2, -1,
                               0, 2, 3, -1,
                               0, 3, 1, -1,
                               1, 3, 2, -1 ] }
```

Indexed Face Sets are important for rendering as well
... Vertex Buffer Objects, index arrays etc.

[2] The OpenInventor, VRML, and OBJ file formats are examples of file formats representing geometry as indexed face sets.

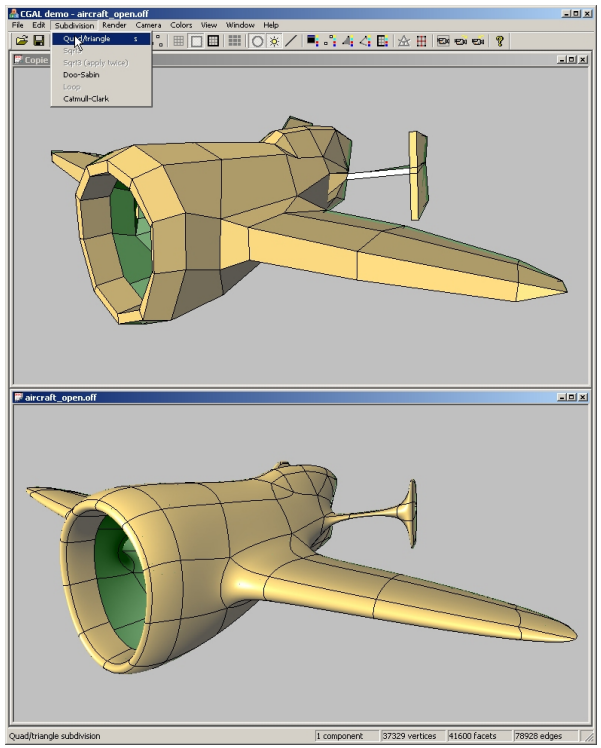
Connectivity

Polygons soups and lists of triangles have little information about the structure of the mesh/surface

Need *connectivity* (topology) - relating vertices, edges and faces to each other - in order to organize and work with meshes

Typical operations/queries

- Access vertices, edges and faces
- traverse a face
- get vertices or edges contained in a face
- get faces containing a vertex or edge
- get all neighbours of a face, edge or node
- traverse a mesh
- ...

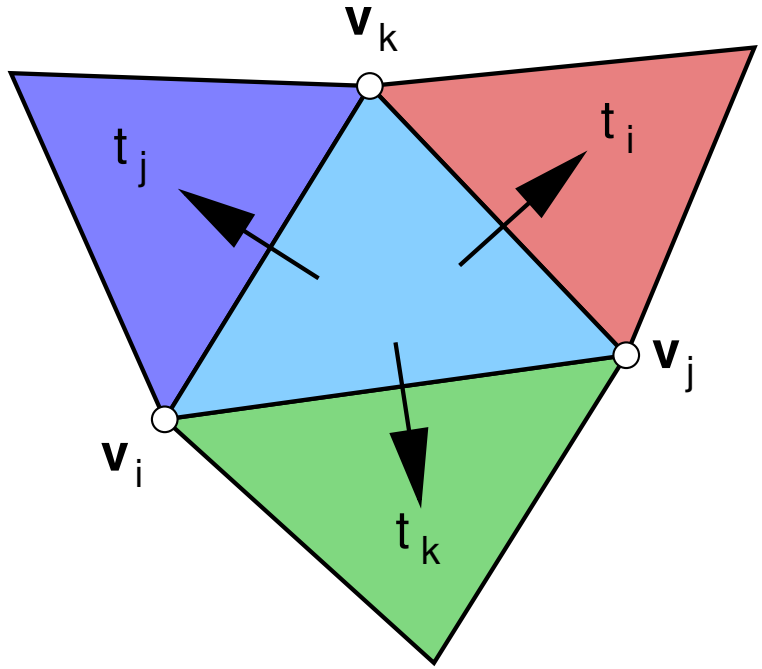


There are many ways to organize a polygonal mesh - Higher level data-structures required

Triangle based data structure with neighbours

Regular triangle list with pointers to neighbouring triangles.

A triangle $[\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k]$ has three extra fields, pointers to nbr triangles t_i, t_j, and t_k.

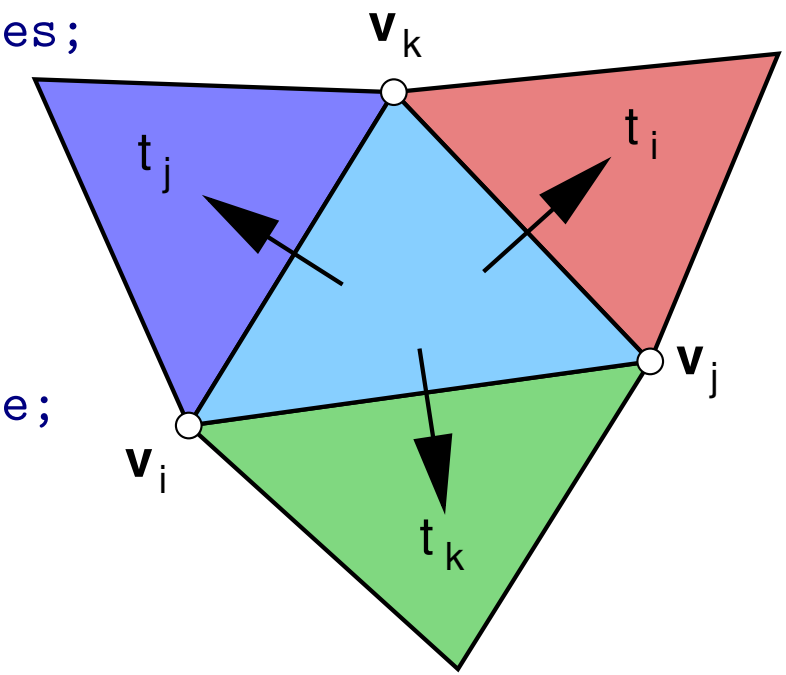


Common convention:
neighbour t_i corresponds to $\mathbf{v}_i$ is on the opposite side of the triangle, i.e., connecting to the edge $[\mathbf{v}_j, \mathbf{v}_k]$.

In addition: each vertex has a *leading triangle*

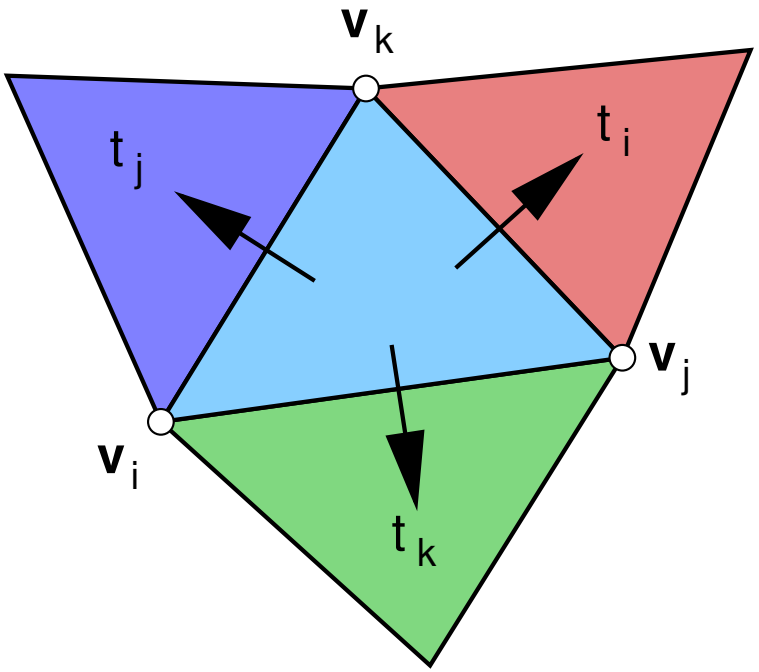
The data structure is something along the lines of:

```
class Triangulation {
    vector<Vertex*>   m_vertices;
    vector<Triangle*> m_triangles;
};

class Vertex {
    Vec3f     m_position;
    Triangle* m_leading_triangle;
};

class Triangle {
    Vertex*   m_vertices[3];
    Triangle* m_neighbours[3];
};
```



Mesh operations are relatively straightforward, e.g

- getTriangleNodes(tri)
- getTriangleNeighbours(tri)
- getNodeNeighbours(node): get leading triangle, traverse around node

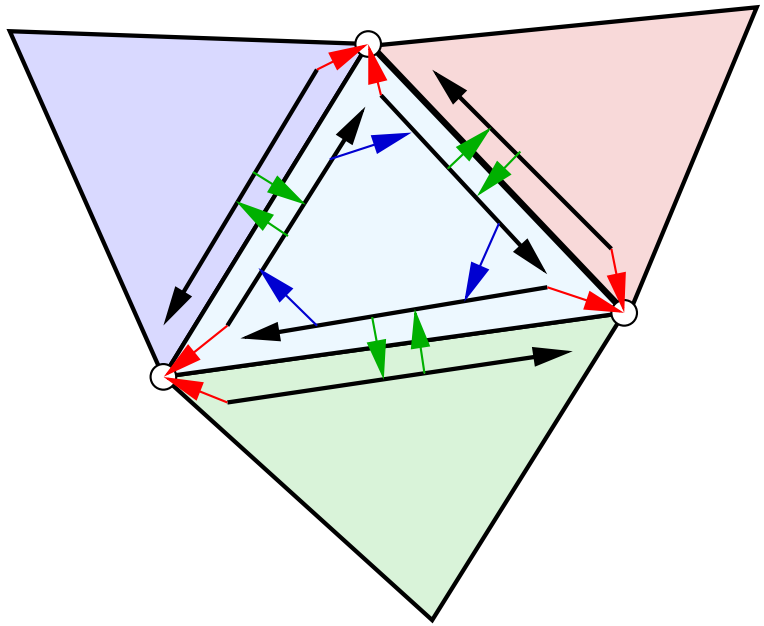


Half-edge based triangle structure

An alternative is the more flexible *half-edge* data structure.

Each triangle represented by 3 half-edges, each with 3 pointers:

- Source node.
- Next half-edge in the triangle loop.
- Twin half-edge in the adjacent triangle.

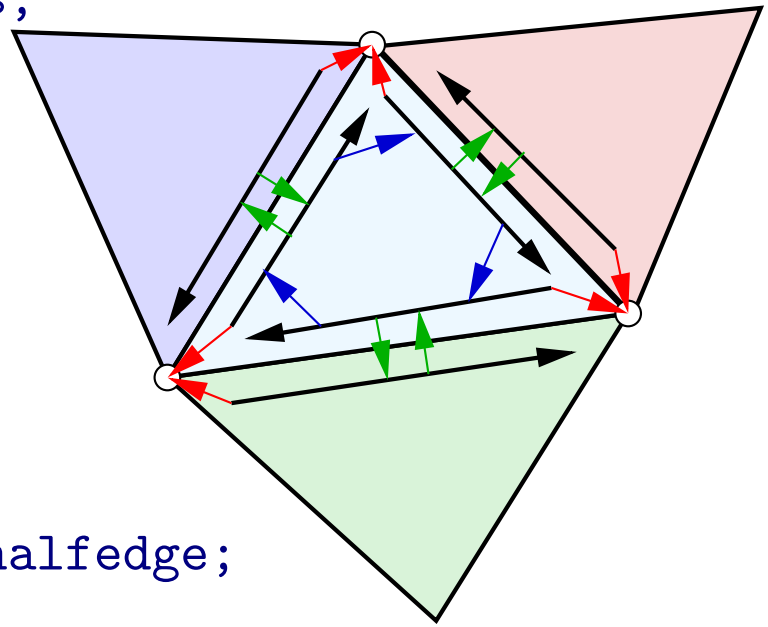


Each vertex has a pointer to a *leading half-edge*: one of the half-edges emanating from it.

Usually, we also add a triangle class to hold triangle-specific data.

Half-edge datastructure

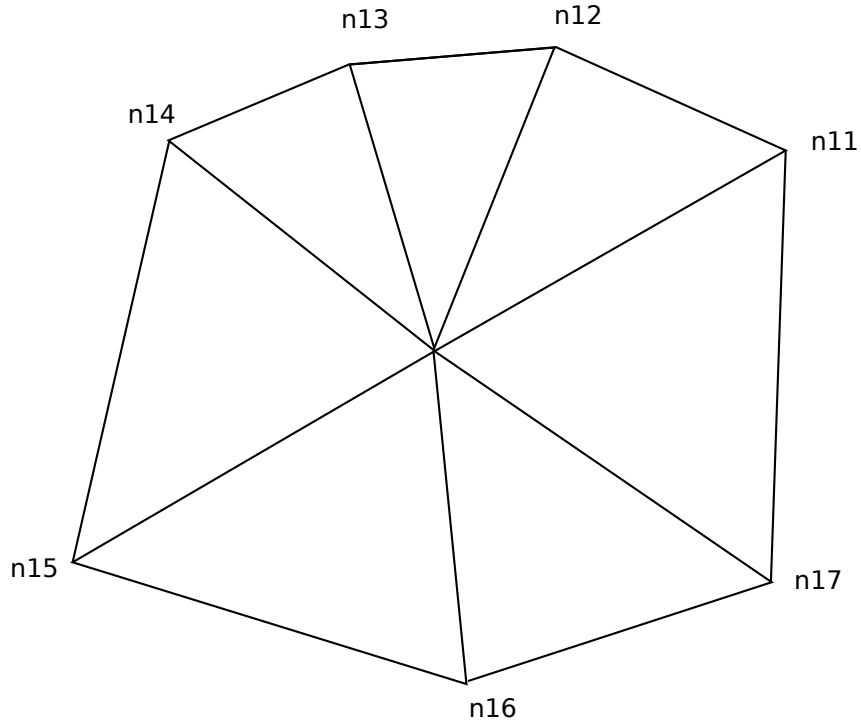
```
class Triangulation {
  vector<Vertex*>     m_vertices;
  vector<HalfEdge*>   m_halfedges;
  vector<Triangle*>   m_triangles;
};

class HalfEdge {
  Vertex*             m_source;
  Triangle*           m_triangle;
  HalfEdge*           m_next;
  HalfEdge*           m_twin;
};

class Vertex {
  Vec3f               m_position;
  HalfEdge*           m_leading_halfedge;
};

class Triangle {
  HalfEdge*           m_leading_halfedge;
};
```



Node-based data structure

Store for each node a list of its neighbour nodes (Cline & Renka)

```
 node     :     neighbor nodes
node 1   :    n11, n12, ....,n1m
node 2   :    n21, n22, ....,n2m
node 3   :    n31, n32, ....,n3m
...
```



- Very compact format
- ... but harder to work with
- No explicit triangles, two versions of each edge
- Some queries are simple (i.e getNeighbourNodes), others more complicated (i.e. getNeighbourTriangle)

Consolidation

Consolidation

Connectivity (mesh-topology) describes how triangles and vertices relate to each other:

- Which triangles share a given vertex.
- Which triangles are adjacent (share an edge).
- What are the neighbours of a vertex.
- ...

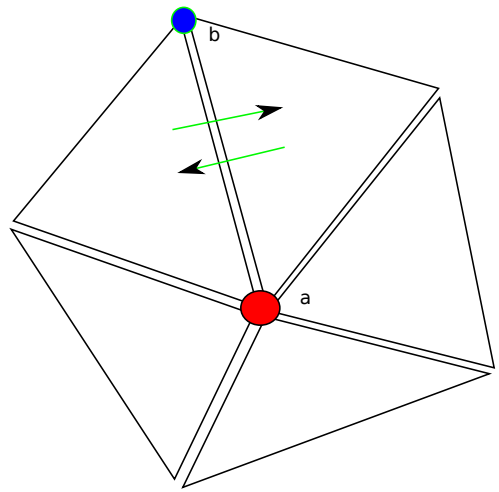
Consolidation is the process of finding and adjusting the connectivity (topology) of a mesh.

Typical algorithm:

1. Run over all faces, collect faces sharing an edge (or node)
2. For each edge (or node), connect sharing triangles

Consolidation - triangle based

1. For each vertex, make a list of faces containing it (for each face, add it to the list of each of its vertices)
2. For each list, connect faces that share two vertices
3. Choose one leading triangle



Many alternatives, e.g. use Standard Template Library (STL) maps

For each edge in each triangle create a struct

```
struct HData {
   int       a;   // smallest edge vertex index
   int       b;   // largest edge vertex index
   Triangle* tri; // pointer to the triangle
};
```

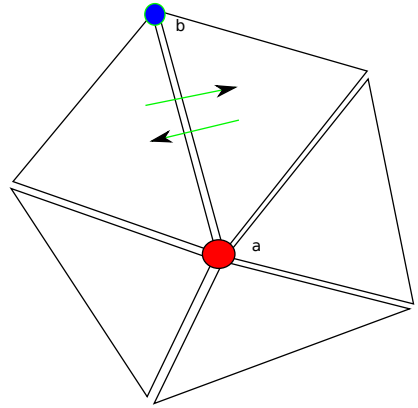
Put all these into a helper list

```
for each triangle t and for each i=0..2:
    int a = min(t[i], t[(i+1)%3]);
    int b = max(t[i], t[(i+1)%3]);
    helper.push_back(new HData(a, b, t));
```

Sort helper list lexicographically:

```
if(helper1.a == helper2.a)
    return helper1.b < helper2.b;
return helper1.a < helper2.a;
```

If two consecutive elements in `helper` have equal `a` and `b`, the two triangles the elements point to are adjacent along the edge (`a`,`b`).




```
//Loop through all edges
for(j=0; j<helper.size(); j++)
    //Find i so that i and i+1 are different edges
    for(i=j; i<helper.size()-1; i++)
        if(helper[i].a != helper[i+1].a ||
           helper[i].b != helper[i+1].b)
            break;

    if(i==j)
        // helper[j] points to a boundary edge
    else if (i==j+1)
        // connect the triangles of helper[i] and helper[j]
    else
        // j..i-1 are non-manifold edges
    j = i;
```

Mesh geometry

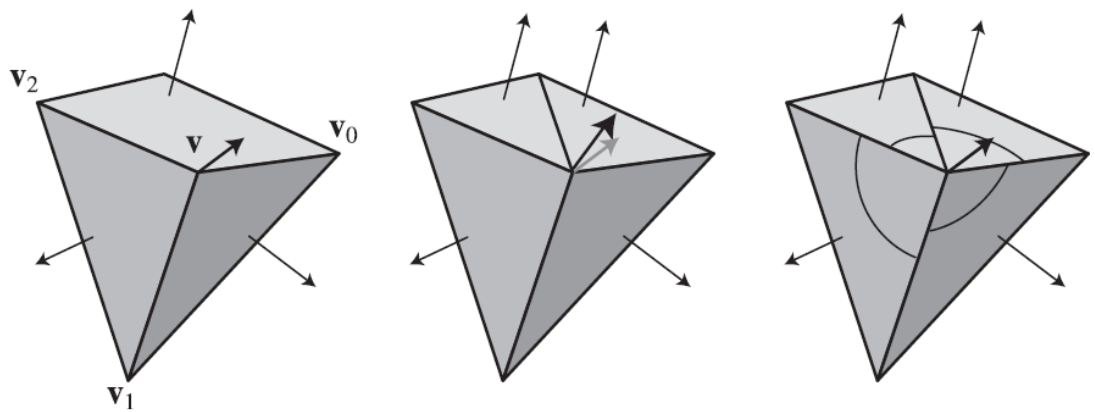
Normals

Triangle normals: $N_T = \frac{(\mathbf{v}_j - \mathbf{v}_i) \times (\mathbf{v}_k - \mathbf{v}_i)}{||(\mathbf{v}_j - \mathbf{v}_i) \times (\mathbf{v}_k - \mathbf{v}_i)||}$

Vertex normals: typically weighted average of nbr triangle normals

$$N_{\mathbf{v}_i} = \frac{\sum_{j \in TNbr_i} w_{ij} N_{T_j}}{\sum_{j \in TNbr_i} w_{ij}}$$

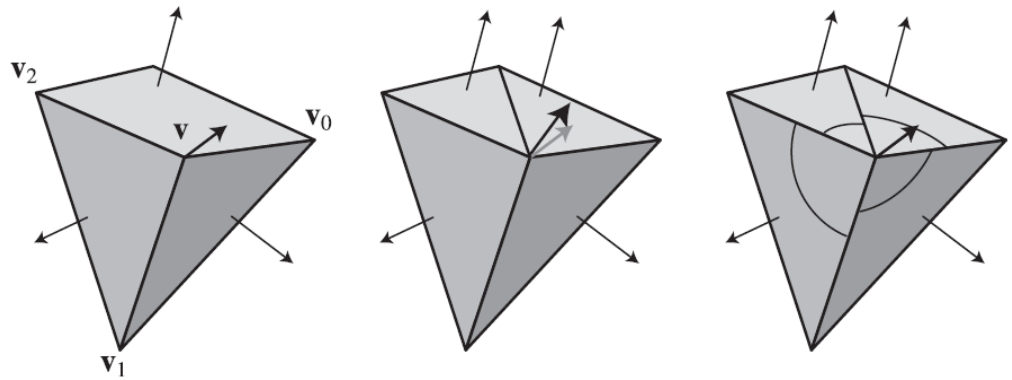
- Uniform: $w_{ij} = 1$
- Angle: $w_{ij} = \theta_{ij} = Angle(T_j, i)$
- Area: $w_{ij} = Area(T_j)$



Curvature, features

Curvature: measures how quickly the normal direction vary

Triangle meshes: discrete measures for curvature, e.g. dihedral angle (angle between neighbouring triangle normals)



Features/creases: regions with large curvature

Simplification

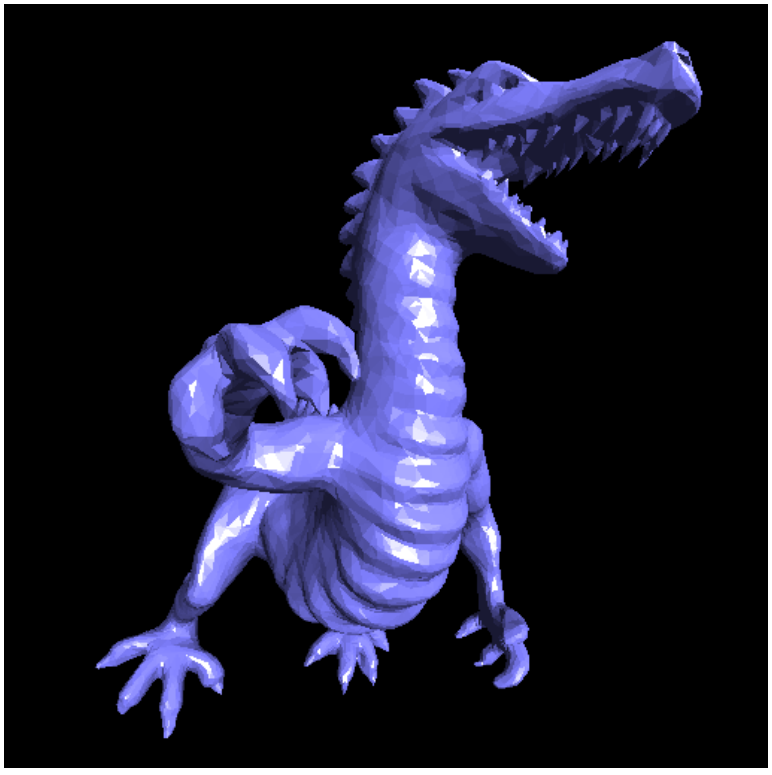
Level-Of-Detail (LOD)

Distance/size determines neccessary accuracy.

Level-Of-Detail (LOD)

Distance/size determines neccessary accuracy.

Challenge: continuous level-of-detail



20,000 triangles

1,000 triangles

Many real life meshes are huge (millions of triangles)!

Important to be able to reduce data or *simplify* the mesh.

1. *Static simplification*: creating seperate level of detail models before rendering begins.
2. *Dynamic simplification*: creating a continuous level of detail.
3. *View-dependent simplification*: variable level of detail within the model, used extensively in terrain rendering.

Both static and dynamic simplification is usually implemented using an *incremental* algorithm:

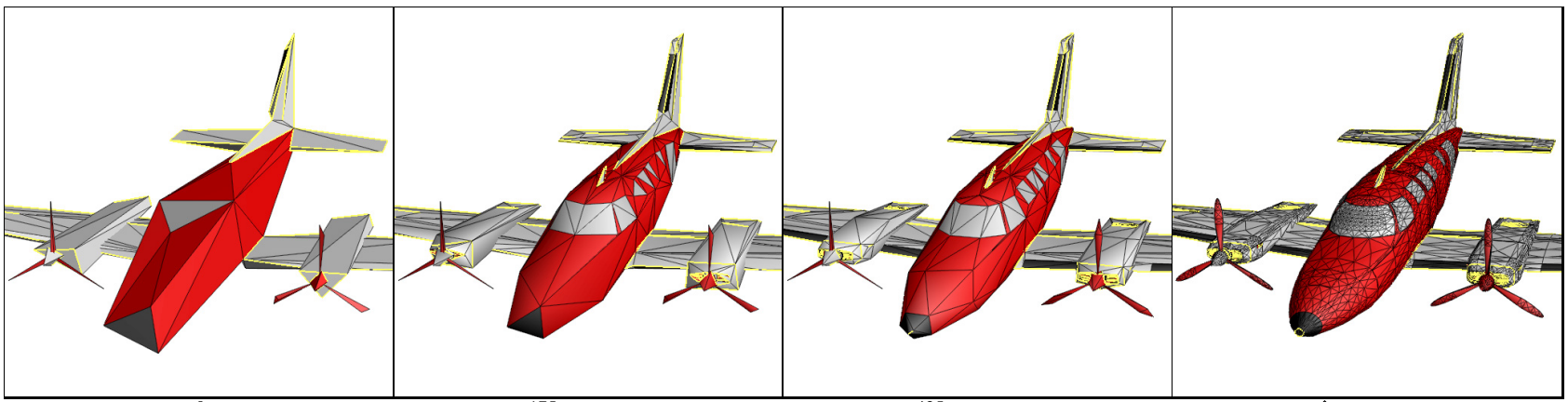
> *Remove one vertex at a time and repair the hole left by the removal.*

- Dynamic simplification: store sequence of point removals.
- Static simplification: remove a given number of vertices for each level-of-detail.

Ideally: remove maximal num. vertices so that the resulting mesh is still a good enough approximation to the original mesh

Example: progressive meshes

Continuous LOD: add/remove vertices one by one

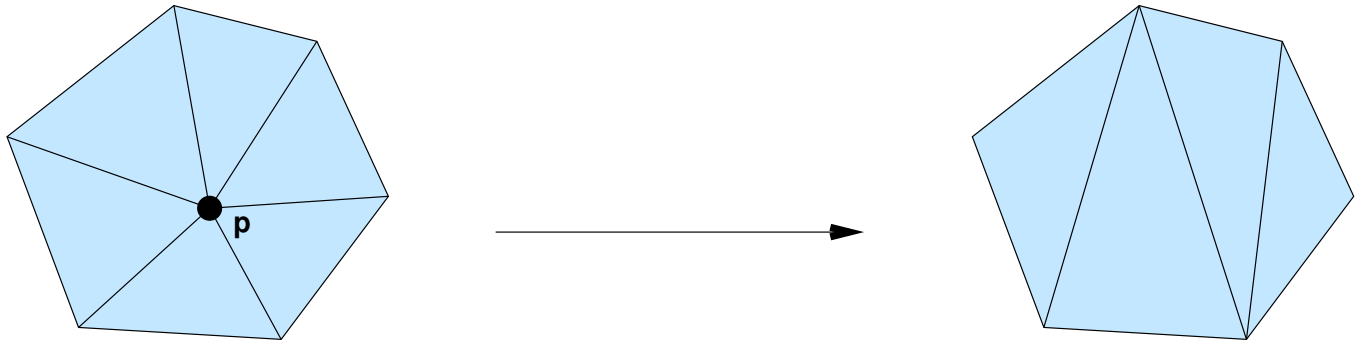


(a) Base mesh M^0 (150 faces) (b) Mesh M^{175} (500 faces) (c) Mesh M^{425} (1,000 faces) (d) Original $\hat{M} = M^n$ (13,546 faces)

[Hoppe]

Vertex removal

Remove a vertex **p** and retriangulate the hole.



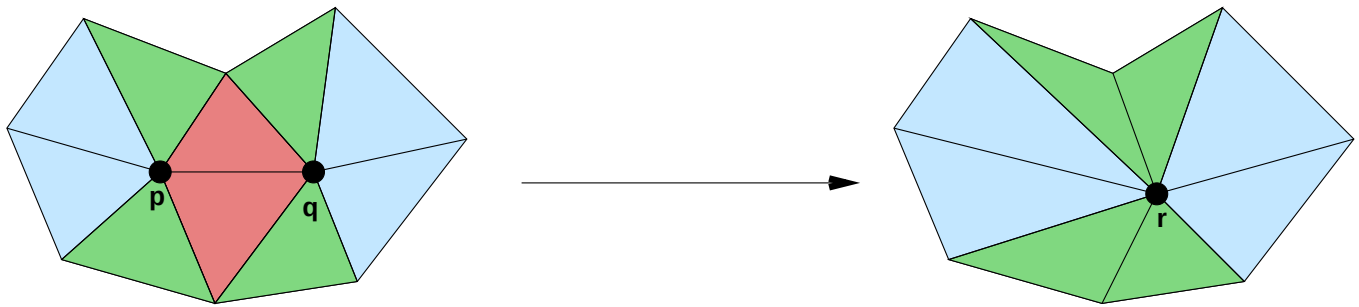
If there were k triangles/edges sharing **p**, there will be $k-2$ triangles and $k-3$ edges in the repaired mesh.

Since the number of vertices is reduced by 1, we see that the Euler characteristic $\chi = v - e + f$ is unchanged, reflecting the fact that vertex removal is a *Euler operation*.

There are a lot of choices of how to triangulate the resulting hole.

Edge collapse

To reduce the number of choices, we can choose one edge, and let this edge degenerate to a point, which is known as *edge-collapse*.



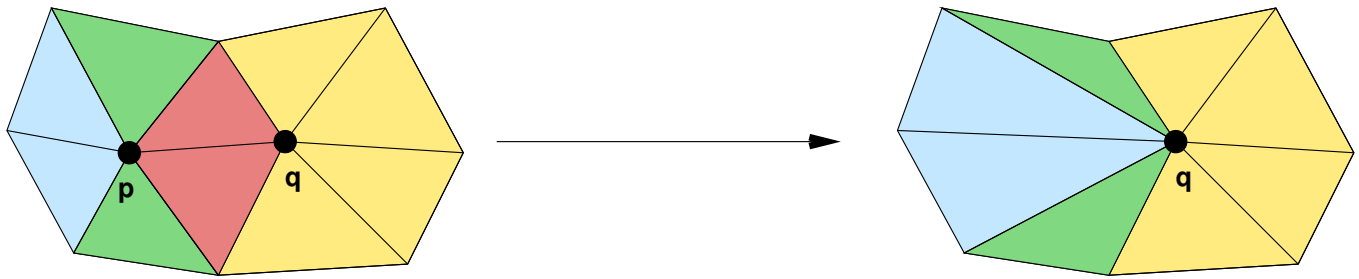
An *edge collapse* takes two neighbouring vertices **p** and **q** an collapses the edge betwen them to a new point **r**.

As a result, two triangles adjacent to the edge $[\mathbf{p}, \mathbf{q}]$ become degenerate and are removed from the mesh.

We must somehow determine the position of the new point **r**.

Half-edge collapse

To reduce the number of choices even more, *half-edge collapse* moves **p** to **q**. Again two triangles become degenerated and are removed.

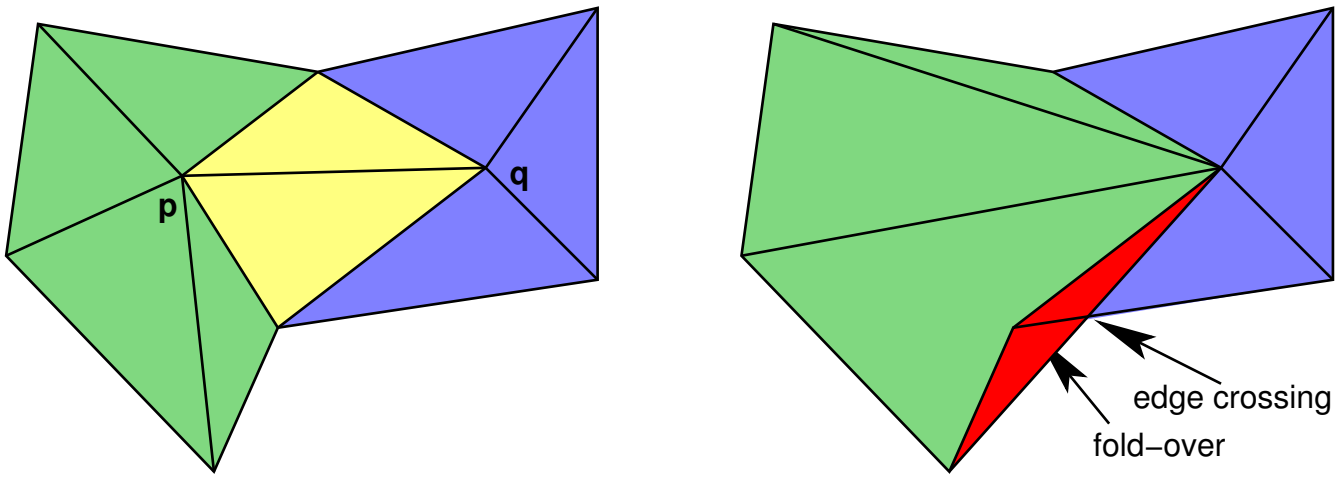


It can be thought of as a special case of edge collapse where the new position **r** is taken to be **q**.

It is also the special case of vertex removal in which the triangulation of the k-sided hole is generated by connecting all neighbouring vertices with **q** instead of **p**.

Fold-overs

Consider half-edge collapsing **p** to **q**:

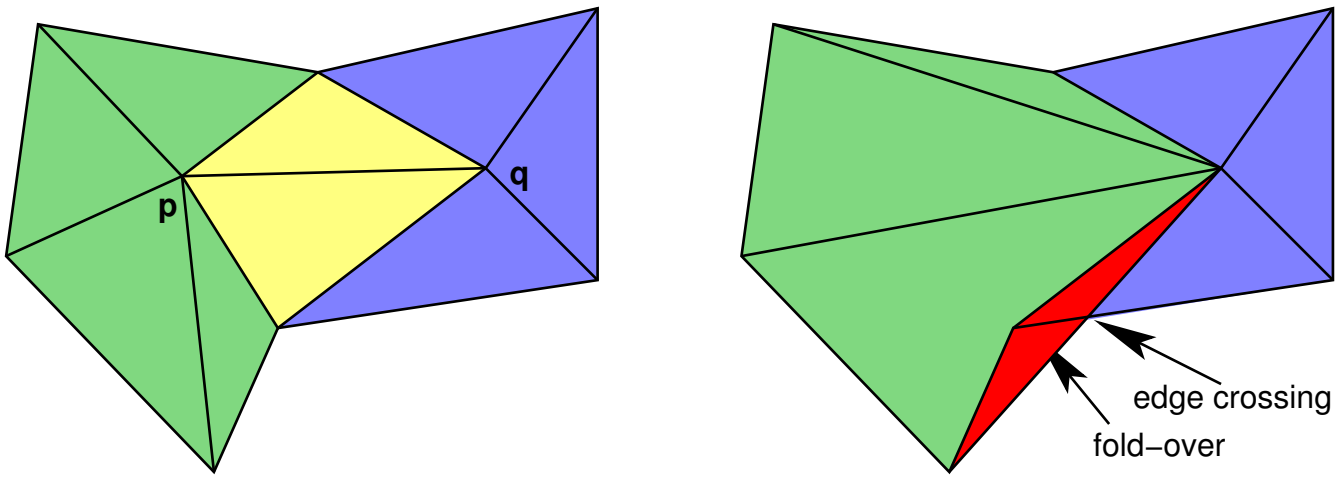


Get a fold-over (red triangle), which can make the mesh invalid (2D) or introduce artifacts (3D)

Detecting fold-overs in $\mathbb{R}^2$ is quite simple, but detecting fold-overs in $\mathbb{R}^3$ is difficult to do in a bullet-proof fashion.

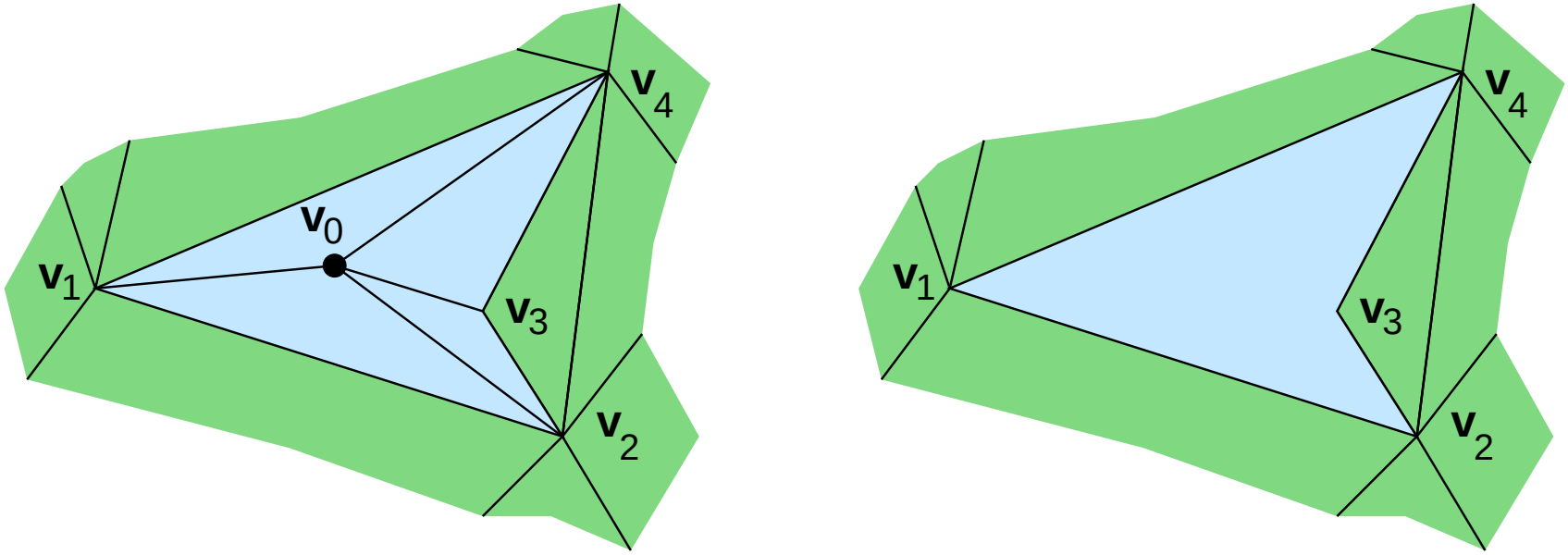
A strategy is to investigate triangle normals:

> *If a half-edge collapse is producing triangles where the normal vector of adjacent triangles differs too much, it is defined an illegal collapse.*



Keeping the graph simple

Consider removing $\mathbf{v}_0$, leaving a four-sided hole.



It would be fine to triangulate the hole by connecting $\mathbf{v}_1$ and $\mathbf{v}_3$.

We could *not* however use the alternative of connecting $\mathbf{v}_2$ and $\mathbf{v}_4$ since $\mathbf{v}_2$ and $\mathbf{v}_4$ are *already* neighbours, and thus, we get *four* triangles meeting at the edge $[\mathbf{v}_2, \mathbf{v}_4]$.

Connecting $\mathbf{v}_2$ and $\mathbf{v}_4$ would yield a non-simple graph (a graph is simple if no pair of vertices belong to more than one edge).

Most data structures require a valid mesh

Avoided by simply not allowing such connections by a simple rule:

Never connect two vertices that are already connected.

Situation tend to occur more frequently as the mesh is simplified.

At some point it is not possible to make *any* further vertex removals, e.g. if the mesh is a tetrahedron

Removal criteria

Greedy approach: remove the least important vertex sequentially, based on some *criterion*

The criterion could be based on:

- Approximation error (distance between the previous mesh and the new one)
- Vertex density (but many vertices needed in curved regions)
- Triangle aspect ratio (avoid long thin triangles, important for solving PDEs).

Implementation

For our chosen removal criterion we compute, for each candidate removal p, some measure $sig(p)$ of the how *significant* it is.

Once we have $sig(p)$ for every possible removal p in the mesh, we can remove the best removal p_*

Most (efficient) criteria are *local*, so that $sig(p)$ depends only on p and its (immediate) neighbours.

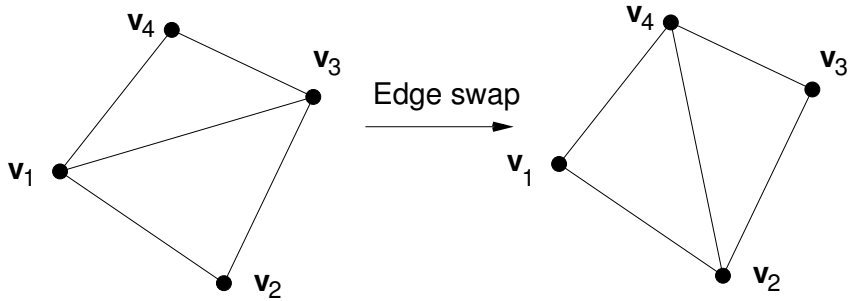
Use a *heap data structure* for best removal in $O(\log n)$

Mesh optimization

Mesh optimization

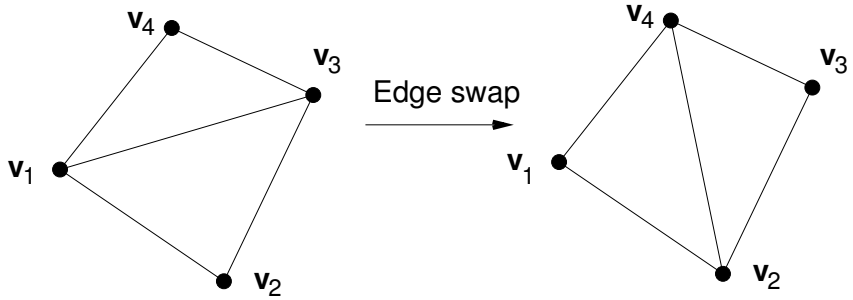
Conceptually simple: Change mesh topology to get better quality

The simplest change in connectivity we can make is to *swap* edges, i.e. diagonals of a *quadrilateral*.



Edge swap

In the figure below we swap the edge $[\mathbf{v}_1, \mathbf{v}_3]$ with $[\mathbf{v}_2, \mathbf{v}_4]$.



This has the effect of replacing the two triangles $[\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$ and $[\mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_4]$ by $[\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_4]$ and $[\mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4]$.

> *As with vertex removal, care must be taken not to invalidate the mesh by creating a non-simple graph.*

Thus if $\mathbf{v}_2$ and $\mathbf{v}_4$ on the left are already connected by an edge outside the quadrilateral, the swap is *illegal*.

Edge swapping

By applying several edge swaps we gradually change the original mesh into a better one, measured by a global *cost function*, e.g.

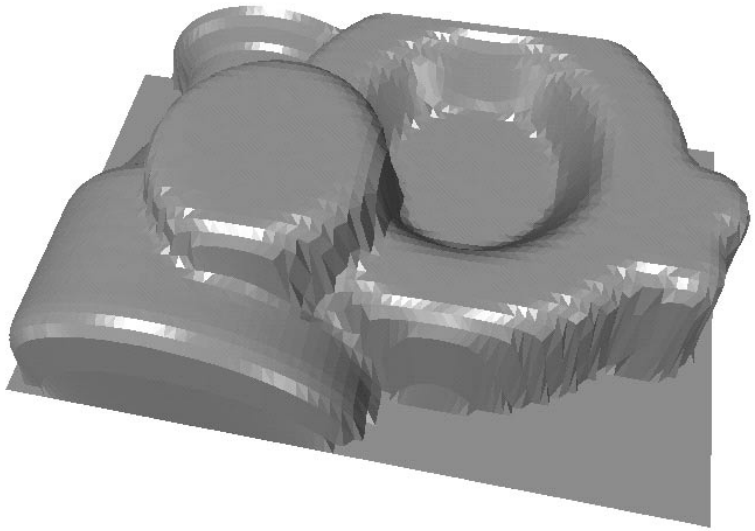
- mesh smoothness/fairness/curvature
- triangle aspect ratio

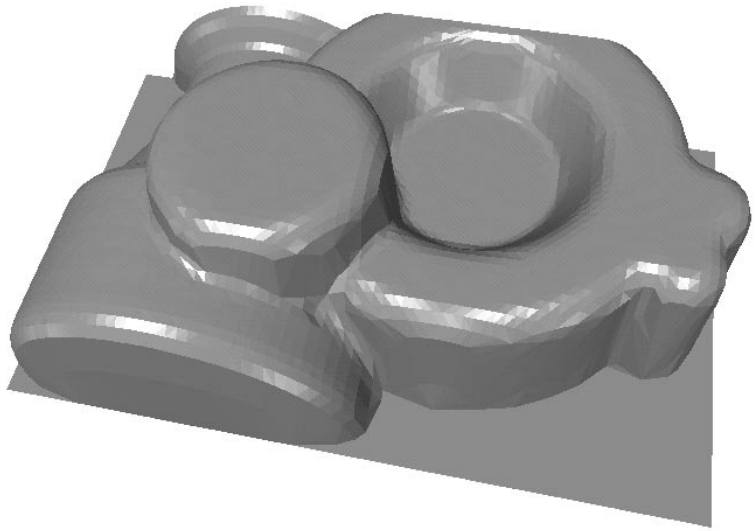
Greedy approach:

- Use a *swap criterion* (score) that ranks a swap wrt decrease in the cost function.
- Do the best legal swap until cost function cannot be decreased further.

Implemented much like the decimation algorithm.

Edge swapping example





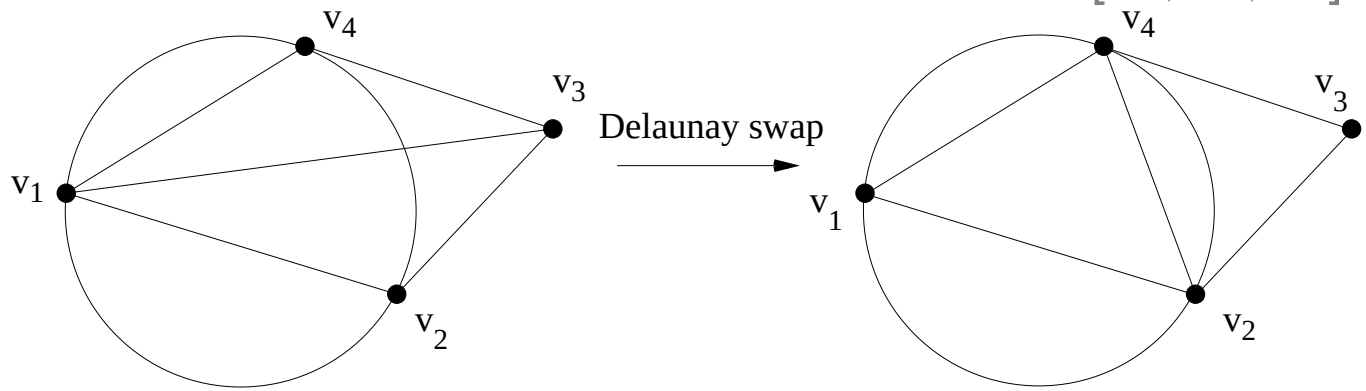
[Dyn et al.]

It is in general not possible to reach the global minimum.

Delaunay triangulations (2D)

The in-circle swap criterion for *planar triangle meshes*:

> *Swap* $[\mathbf{v}_1, \mathbf{v}_3]$ *if it is the diagonal of a convex quadrilateral and* $\mathbf{v}_3$ *lies outside the circumcircle of* $[\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_4]$.



Equivalently: the *max-min angle* criterion:

> *swap if the minimum of the six angles in the two triangles is increased.*

The beauty of these criterions is that it a locally optimal triangulation is in fact globally optimal.

The optimal triangulation is a *Delauanay triangulation*:

- the minimum of all the triangle angles is maximized,
- the interior of the circumcircle of each triangle is empty (contains no other vertices of the triangulation),
- if no set of three points are cocircular, the Delaunay triangulation is unique.

Voronoi diagram

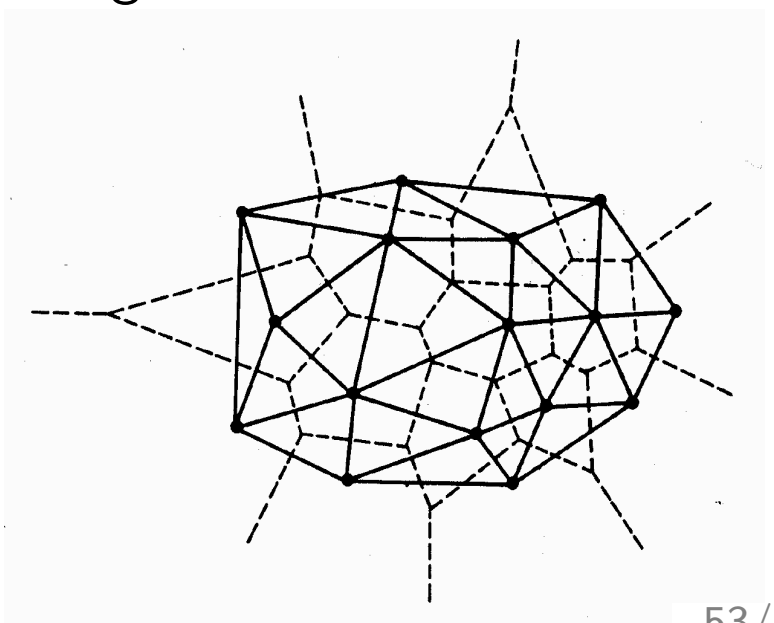
The Voronoi diagram of a set of planar points $\mathbf{p}_1, \ldots, \mathbf{p}_N$ is a collection of tiles:

Tile V_i is the set of all points in $\mathbb{R}^2$ that are closer to $\mathbf{p}_i$ than any other point $\mathbf{p}_j$, i.e.,

$$V_i = \{\mathbf{x} \in \mathbb{R}^2 : \quad \|\mathbf{x} - \mathbf{p}_i\| \leq \|x - \mathbf{p}_j\| \qquad \forall j \neq i\}.$$

The Voronoi diagram and Delaunay triangulation are *duals*:

Vertices of VD are circumcenters of DT



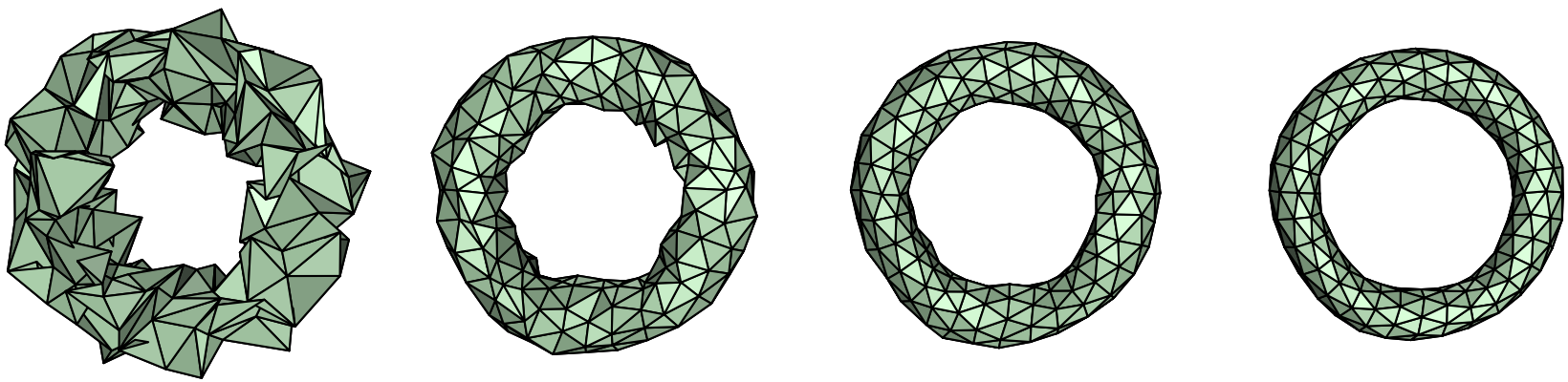
Mesh smoothing

Change vertex positions to get better

- mesh smoothness/fairness/curvature
- triangle aspect ratio

Typically done to remove noise

Laplacian smoothing: repeat $\mathbf{v}_i^{new} = \sum_{j \in N_i} \mathbf{v}_j / n$



Challenges:

- volume and feature preservation
- smoothing scale
- mesh topology independence

Mesh smoothing

original

noise removal
narrow spatial, narrow influence
($\sigma_f = 2$, $\sigma_g = 0.2$)

smooth small features
narrow spatial, wide influence
($\sigma_f = 2$, $\sigma_g = 4$)

smooth large features
wide spatial, wide influence
($\sigma_f = 4$, $\sigma_g = 4$)

[Jones et al.]

The last slide

Much more could be said about meshes:

- Mesh encoding and compression
- Feature detection
- Shortest paths, parameterization, ...

Some further pointers

- Computational Geometry Alg. Library (CGAL):
  `www.cgal.org/`
- Open Mesh:
  `www.openmesh.org/`
- Polygonal Mesh Modeling tutorial:
  `http://graphics.ethz.ch/Downloads/Publications/Tutorials/2008/Bot08a/eg08-tutorial.pdf`

Next time: Parametric curves and surfaces