Lecture 10:

Triangle Meshes: Simplification and Optimization

Topics:

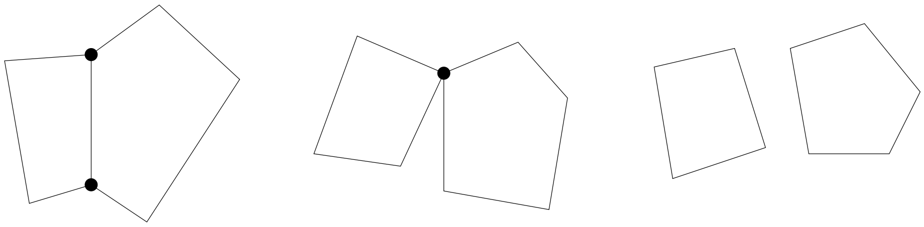
1. Triangle meshes
2. Mesh data structures
3. Mesh simplification
4. Mesh optimization

Michael S. Floater, Oslo, Oct. 2002

Polygonal meshes

A **polygonal mesh** is a set of faces (polygons) in $\mathbb{R}^3$ such that

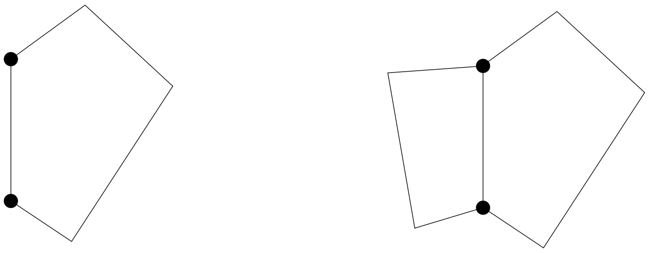
(i) the intersection between any pair of faces is either a common edge, a common vertex, or nothing:



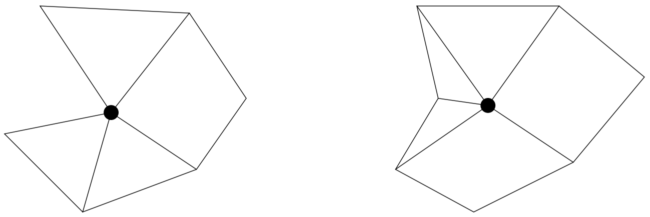
and

(ii) the union of the faces is a manifold surface (with or without boundary).

Condition (ii) implies that the mesh looks locally like a surface. It requires that (a) each edge belongs to either one or two faces:



and (b) the faces incident on a vertex form an open or closed ‘fan’:



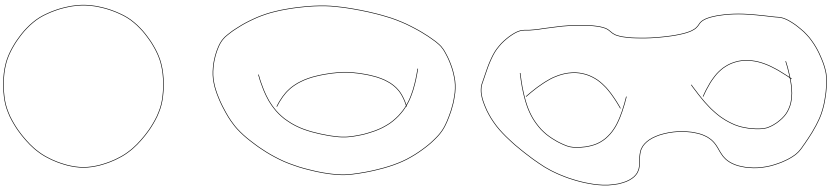
Edges belonging to only one face form the **boundary** (if any) of the mesh. The boundary (if any) consists of one or more loops.

The orientation of a face is the cylic order of its incident vertices. There are two possible orientations of each face: clockwise and anti-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 formula** describes the relationship between the number of vertices v, edges e, faces f, and the topological type of an orientable manifold mesh. The topological type is given by the **Euler characteristic** χ, and the Euler formula is

$$\chi := v - e + f.$$

A mesh is said to have **genus** g if it can be cut along $2g$ closed loops without disconnecting it. The **sphere** and **torus** have genus zero and one respectively. The **double torus** has genus two etc. Any mesh of genus g can be continuously deformed into a sphere with g handles.



If s is the number of connected components, g the genus and b the number of boundary loops, it can be shown that

$$\chi = 2(s - g) - b.$$

In the special case of a closed manifold triangular mesh each edge has exactly two incident triangles and each triangle has three incident edges, thus

$$2e = 3f.$$

This yields

$$2v - f = 2\chi,$$

and as χ is typically small,

$$f \approx 2v.$$

A mesh is sometimes called **simple** if it is connected, orientable, of genus zero, and has one boundary loop.

Geometry and connectivity. We can view the mesh as consisting of **geometry**, i.e., the vertices, and **connectivity**, i.e. the graph which specifies which pairs of vertices are neighbours. Two vertices are **neighbours** if they are the end points of an edge. The graph can be written as $G(V, E)$. Each vertex in the mesh can be represented by an integer index i in V and each edge can be represented as a pair (i, j) in E.

Mesh data structures

It is usual to represent the mesh geometry as a vector-valued array of floats or doubles:

```
x0 y0 z0
x1 y1 z1
...
xm ym zm
```

The connectivity is often defined by a **face vertex incidence table**. For each face an oriented cyclic list of indices of the incident vertices is given. In VRML the `IndexedFaceSet` defines the face-vertex incidence table by one list of indices, where the -1 separates different faces:

```
i0 j0 k0 l0 -1
i1 j1 k1 -1
i2 j2 k2 l2 m2 -1
....
in jn kn -1
```

The order of the vertex indices in a face defines the orientation. For a triangle mesh, no -1 is needed as every face has three vertices:

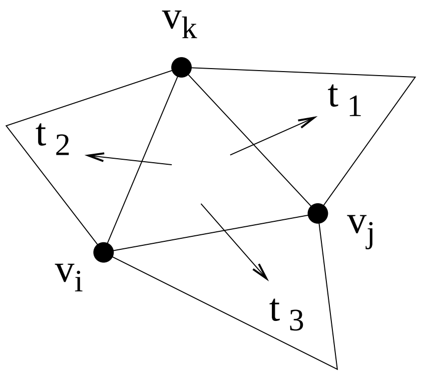
```
i0 j0 k0
i1 j1 k1
...
in jn kn
```

This simple data structure could be organized in three C++ classes: `Triangle`, `Node`, and `TriangleMesh`. The class `TriangleMesh` would consist of two arrays, one of `Triangle`'s and the other of `Node`'s. The following diagram indicates that `Triangle` has (three) pointers to `Node`.

Triangle → Node

This data structure is fine for simple visualization of the triangles. However, it is not good when we need neighbourhood information. We might for example wish to estimate normals at the vertices for use in Gouraud shading. This requires finding all triangles which contain a give vertex. The only way to find these triangles is to check **all** triangles in the triangle array. Thus if there are N triangles, finding one neighbourhood costs $O(N)$ time. Repeating this for each node, and since the number of nodes is about $N/2$, we end up with an $O(N^2)$ algorithm for estimating all the normals. For N in the region of 10^6 this is not an option.

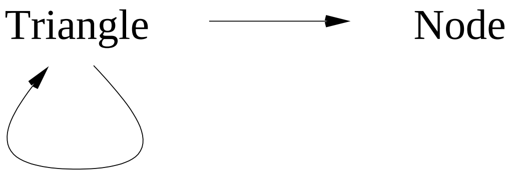
The solution is to enrich the data structure by adding further pointers. One possibility is to add three pointers to the neighbouring triangles in the `Triangle` class.



There are then six data members in the `Triangle` class:

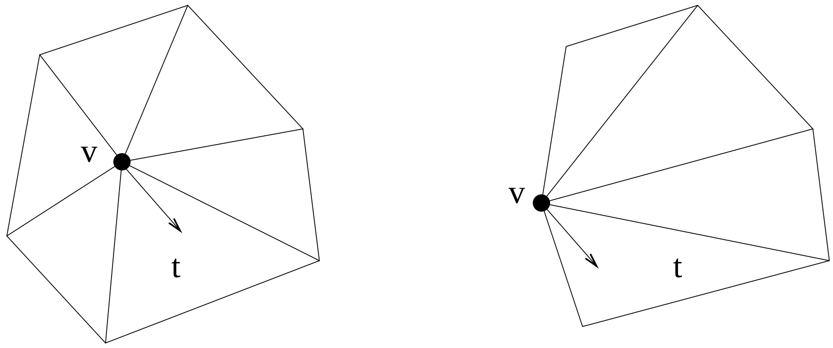
```
i j k  t1 t2 t3
```

with the convention that the first/second/third triangle pointer refers to the triangle opposite the first/second/third vertex. The dependencies are:



If we traverse the vertices by traversing the triangles, and for each triangle checking which of the three vertices have been visited, we can use the three triangle pointers to find all incident triangles to each vertex. Thus, for example, all normals in the mesh can now be estimated in $O(N)$ time. Note that when building this enhanced data structure from the original, all the triangle pointers can be found in $O(N)$ time.

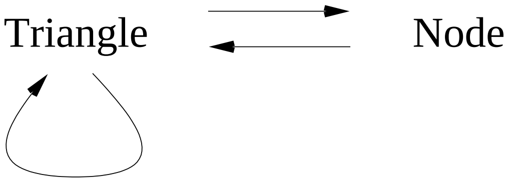
A further help to traversing vertices is to add pointers to the `Node` class. One can simply add a list of pointers to all incident triangles. Another option which uses less space is to add just **one** pointer to one of the containing triangles.



The enhanced `Node` class now take the form

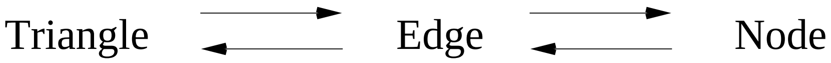
```
x y z  t
```

and the data structure now has the dependencies:



It is now easy to write a `getNeighbours(int i)` routine which returns (in $O(1)$ time) an ordered list of the neighbouring nodes of the given i-th node.

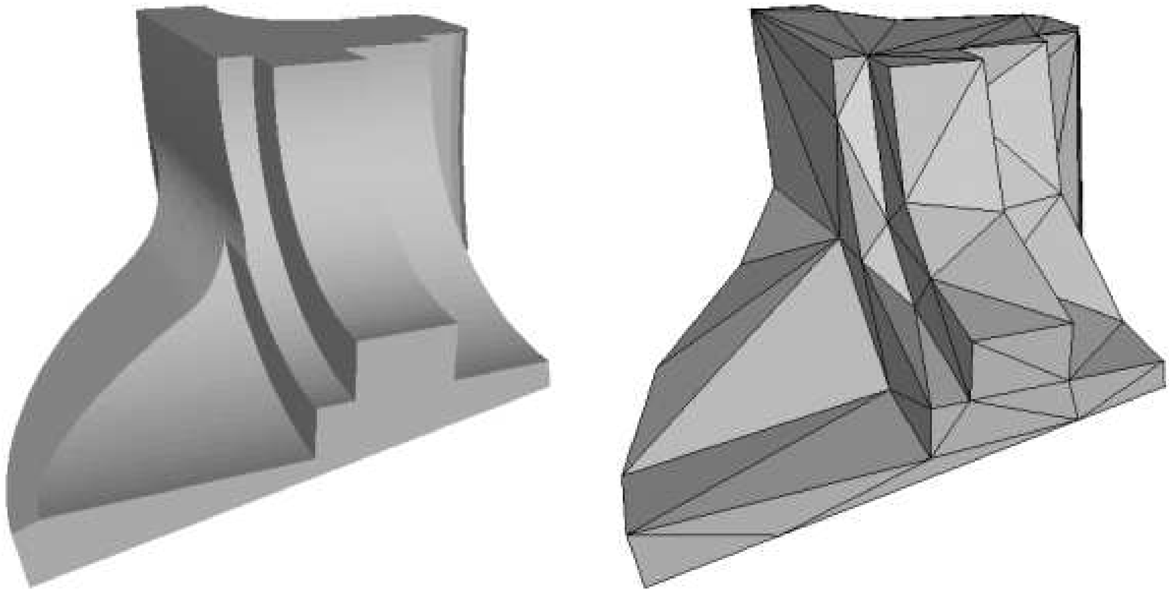
There are many possible data structures for triangle meshes, and all have advantages and disadvantages. Many data structures include an additional `Edge` class and one or more of the dependencies:



Some data structures use ‘half-edges’ instead of edges.

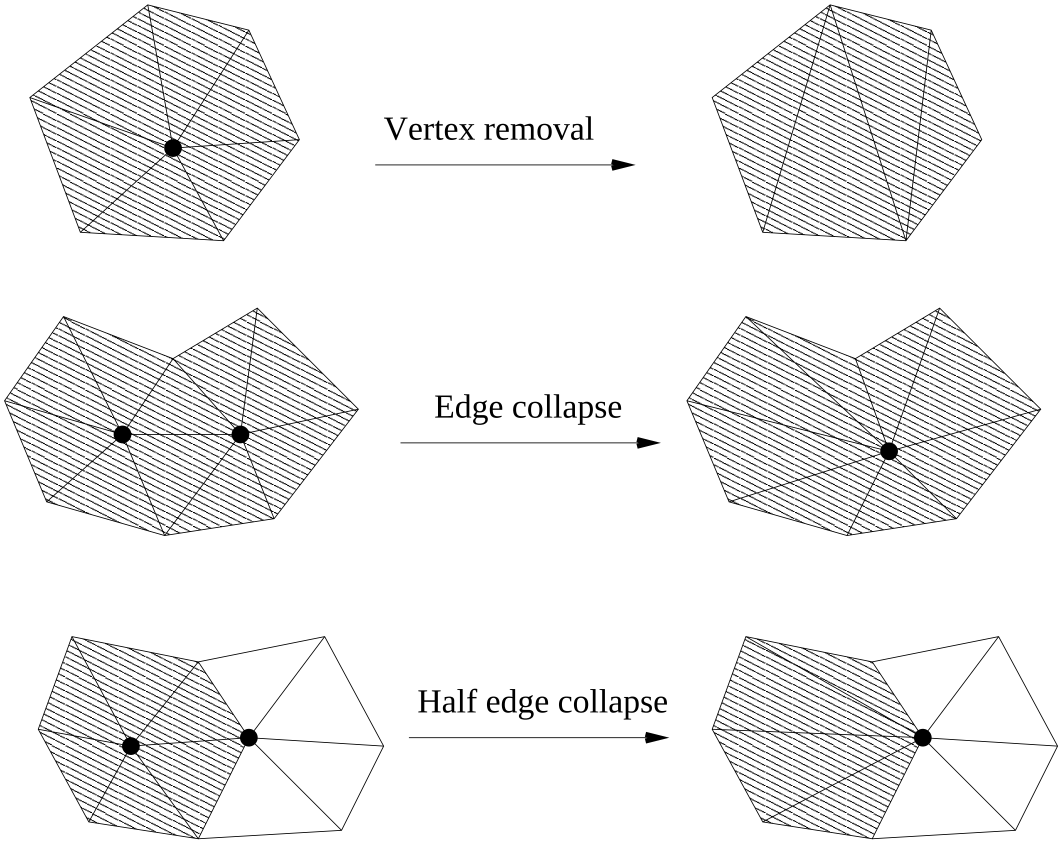
Mesh simplification

Frequently we encounter triangle meshes with very large numbers of triangles and vertices; these data sets often come from scanning millions of points from real objects. It is therefore important to be able to reduce or **simplify** the mesh if necessary.



A common way to simplify a mesh is to apply an **incremental** algorithm, removing one vertex at a time and repairing the hole left by the removal. Ideally, we want to remove as many vertices as we can so that the remaining **coarse** mesh is still a good enough approximation to the original **fine** mesh.

There are three common ways to remove a vertex and repair the hole: **vertex removal**, **edge collapse** and **half-edge collapse**.



Vertex removal simply means retriangulating the hole left by removing a vertex p. If there were k triangles sharing p, there will be $k-2$ in the repaired mesh. Note that the number of edges incident on p was also k and is now $k-3$. 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 an **Euler** operation; it does not change the topology of the mesh. We will not study Non-Euler operations.

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 become degenerate and are removed from the mesh. This is again an Euler operation.

A third, and popular, Euler operation is a **half-edge collapse** in which for some ordered pair of neighbouring vertices (p, q), we 'move' p to q. Again two triangles become degenerate 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.

Removal criteria. In each removal step, the best candidate for removal is determined by some user-specified criterion. For example, we might remove that vertex which results in the smallest distance between the previous mesh and the new one. We might instead remove vertices where the density of vertices is highest, in order to get an even distribution of vertices. A third alternative is to try to maintain good aspect ratios in the triangles (avoiding long thin triangles). One could try to combine several of these criteria.

Implementation. For our chosen removal criterion we compute, for each vertex p, some measure $sig(p)$ of the how **significant** the vertex p is. Once we have $sig(p)$ for every vertex p in the mesh, we can remove any vertex p_* such that

$$sig(p_*) = \min_{p \in V} sig(p).$$

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

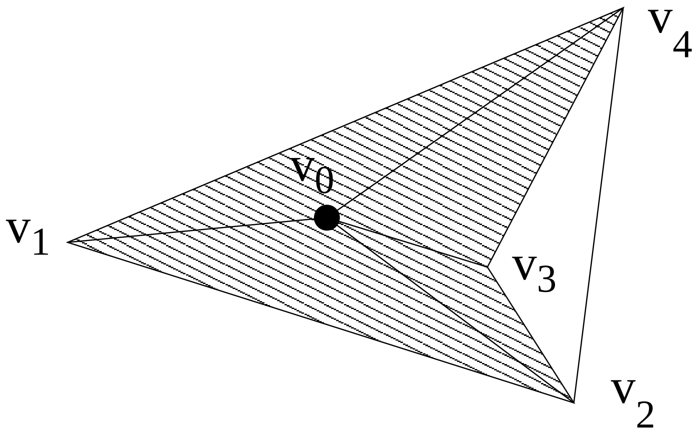
Use of heap. A naive implementation of the decimation algorithm computes $sig(p)$ for each p in the current mesh after every removal. If there are N vertices in the orginal mesh, this costs $O(N^2)$ operations. An enormous improvement is to employ a heap data structure, exploiting the fact that $sig(p)$ remains constant for the vast majority of the vertices p in the current mesh. As a preprocess, we compute $sig(p)$ for all vertices p in the fine mesh and place the vertices p in a heap together with the $sig(p)$ values. A heap is organized as a binary tree, with the vertex p with the smallest $sig(p)$ value at the root of the heap.

We then start decimating. At each step, the vertex to remove is at the root of the heap and so we remove it both from the mesh (and repair the hole) and from the heap. We then only need to update the heap with the new significance values of the vertices that were neighbours of p.

Popping the root of the heap costs only $\log(N)$ operations (since the heap has depth $\log(N)$), and similarly updating one sig value costs $\log(N)$. Thus one whole vertex removal costs only $\log(N)$ operations (assuming the number of neighbours is bounded). Thus the total decimation algorithm costs only $O(N \log(N))$ operations.

Note that when using half-edge collapses, we could assign a significance to each ordered vertex pair (p, q), and the heap would contains these pairs rather than the vertices themselves.

A final word of caution. Care has to be taken when removing vertices. Consider removing v_0 in the figure below, leaving a four-sided hole, shaded.



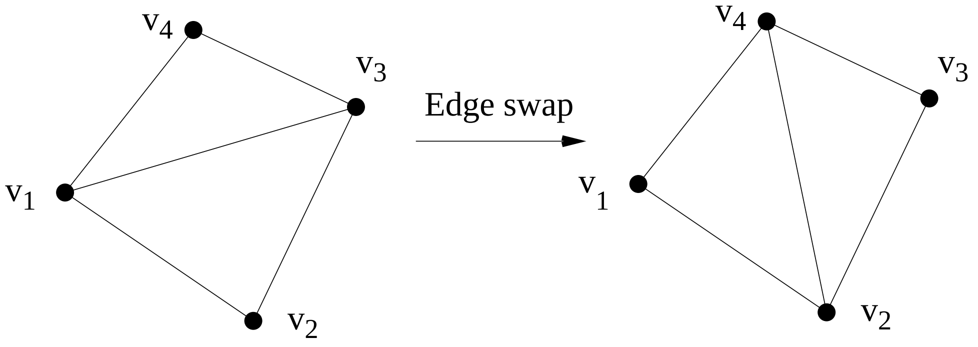
It would be fine to triangulate the hole by connecting v_1 and v_3. We could **not** however use the alternative of connecting v_2 and v_4, the reason being that $v2$ and v_4 are **already** neighbours in the mesh. Thus connecting v_2 and v_4 would yield a non-simple graph (a graph is simple if no pair of vertices belong to more than one edge). Most data structures for triangle meshes are designed on the assumption that the graph is simple.

Such problems are easily avoided by simply not allowing such connections. The rule to remember in vertex removals is **never connect two vertices that are already connected**.

Such possiblities tend to occur more and more frequently as the mesh is simplified. Of course, if we simplify until there are only a handful of vertices left, it may not be possible to make **any** further vertex removals, e.g. if the mesh is a tetrahedron, with just four vertices.

Mesh optimization

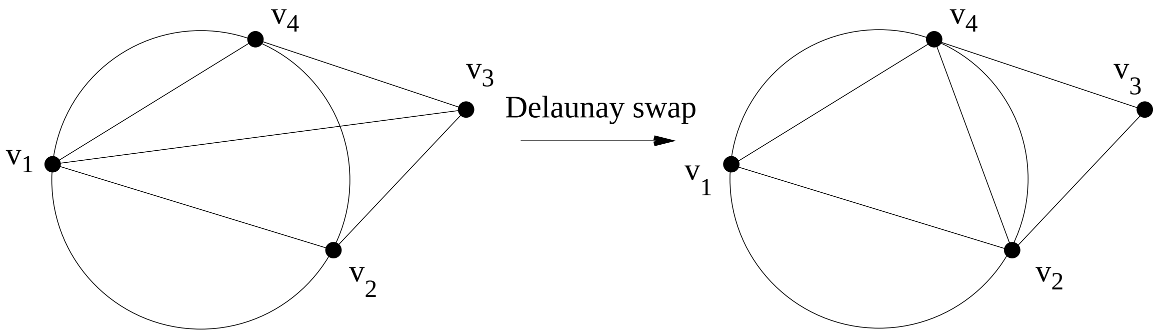
Mesh optimization is conceptually simpler than mesh simplification. The idea is simple. Without changing the vertices, can we change the connectivity of the mesh in order to achieve a better quality surface? The usual approach is **edge swapping**. Each pair of triangles sharing a common edge form a **quadrilateral**. The simplest change in connectivity we can make is to swap the given diagonal of a quadrilateral with the other one. In the figure below we swap the edge $[v_1, v_3]$ with $[v_2, v_4]$. This has the effect of replacing the two triangles $[v_1, v_2, v_3]$ and $[v_1, v_3, v_4]$ by $[v_1, v_2, v_4]$ and $[v_2, v_3, v_4]$.



As with vertex removal, care must be taken not to invalidate the mesh by creating a non-simple graph. Thus if v_2 and v_4 on the left are already connected by an edge outside the quadrilateral, we cannot perform the swap.

By applying several edge swaps we gradually change the original mesh into a better one. We choose some **cost function** we wish to minimize and the **swap criterion** is then simply whether the swap decreases the cost function. We keep on swapping edges which result in a decreased cost function, until no further decreases are possible. Usually it is not possible to guarantee that the global minimum can be reached by a sequence of swaps, but the result is often a better mesh anyway.

For **planar** triangle meshes, there is a very well known swap criterion, called the **Delaunay criterion**. We swap the edge $[v_1, v_3]$ (of a **convex** quadrilateral) if the vertex v_3 lies outside the circumcircle of the triangle $[v_1, v_2, v_4]$, as is in the following figure.



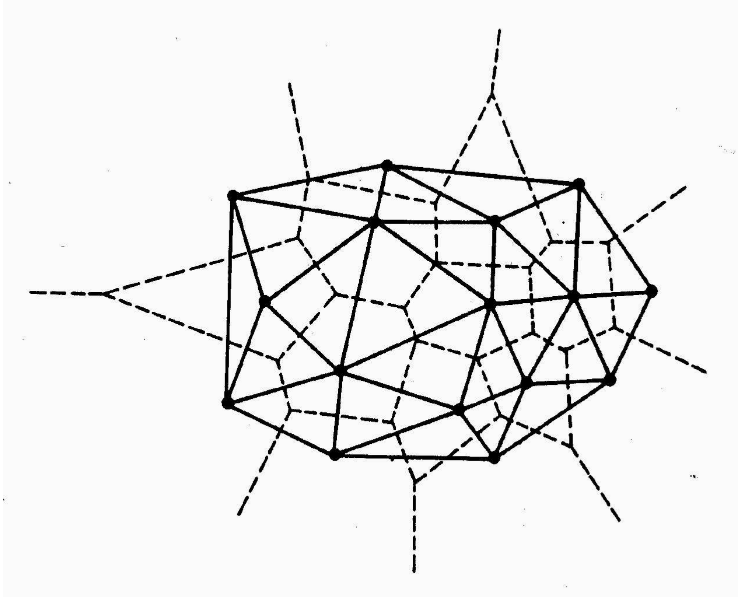
This swapping procedure was proposed by Lawson and it is equivalent to the **max-min angle** criterion; we swap if the minimum of the six angles in the two triangles is increased.

The beauty of the Delaunay swap criterion is that it always leads to a unique triangulation of the planar points (at least if there are no cocircular points). When no more swaps can be made, the triangle mesh is a **Delauanay triangulation** of the plane, i.e., a triangulation in which the interior of the circumcircle of each triangle is empty (contains no other vertices of the triangulation). A Delaunay triangulation has the property that the minimum of all the angles of its triangles is maximized. Thus the Delauany swap criterion tends to give 'well-shaped' triangles where possible.

Delaunay triangulation have several nice properties. A Delauanay triangulation of a set of planar points is the dual graph of their **Voronoi diagram**. The Voronoi diagram of a set of planar points $p_1, \ldots, p_N$ is a collection of tiles. There is one tile V_i associated with each point p_i. The i-th tile V_i is simply the set of all points in $\mathbb{R}^2$ that are closer to p_i than any other point p_j, i.e.,

$$V_i = \{x \in \mathbb{R}^2 : \|x - p_i\| \leq \|x - p_j\| \; \forall j \neq i\}.$$

The figure below shows the Delauanay triangulation (solid) of a set of planar points and their Voronoi diagram (dashed).



For triangle meshes in $\mathbb{R}^3$ the circumcircle criterion no longer makes sense, and though the max-min angle criterion could be used, a unique solution is no longer guaranteed.

In fact in the $\mathbb{R}^3$ case we often use optimization criteria which reflect the geometry of the surface. For example we might optimize the 'smoothness' of the mesh, by minimizing the angle between the normal directions of adjacent triangles, or by minimizing some discrete measure of curvature. The figure below show various optimizations of a given toroidal-shaped triangle mesh.