Triangle Meshes

Akine-Möller, Haines and Hoffman:
- Polygonal Techniques (Chapter 12)
- Sources of three-dimensional data (Chapter 12.1)
- Tesselllation and triangulation (Chapter 12.2)
- Consolidation (Chapter 12.3)
- Triangle fans, strips and meshes (Chapter 12.4)
- Simplification (Chapter 12.5)
Polygonal meshes

Data structures for triangular meshes

Simplification

Mesh optimization
Polygonal meshes
  Polygonal meshes
  The Euler characteristic
  Geometry and connectivity

Data structures for triangular meshes

Simplification

Mesh optimization
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 nothing, 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.

Edges belonging to only one face form the boundary (if any).

The boundary (if any) consists of one or more loops.
The **orientation** of a face is the cyclic order of its incident vertices.

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 of 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 on a mesh.
The **Euler characteristic** denoted $\chi$, is defined by

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

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

The Euler characteristic characterizes the topological type of a mesh.

The genus and the Euler characteristic are related by

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

i.e.,

$$v - e + f = 2 - 2g.$$
Geometry and connectivity

We can view the mesh as consisting of

1. **geometry**, which is the geometric position of vertices

2. **connectivity**, (or **topology**) which is how the vertices are connected together to form a polygonal shape.

The connectivity implicitly defines a set of **edges** connecting pairs of **neighbouring** vertices.

Also, two polygons sharing an edge are said to be **neighbouring** or **adjacent** polygons.
The connectivity defines two different graphs:

1. The vertices are the nodes in a graph where an edge connect two neighbouring vertices.
2. The polygons are the nodes in a graph where an edge connect two adjacent polygons.

1 is the **dual graph** of 2 and vice versa.
Polygonal meshes

Data structures for triangular meshes
- Polygon soups
- Indexed face set
- Connectivity and consolidation
- Triangle based data structure with neighbours
- Half-edge based data structure
- Example consolidation strategy

Simplification

Mesh optimization
Polygon soups

A **polygon soup** 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**.

An example (in .stl format):

```
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.
Polygon soups have some serious drawbacks:

- A vertex is usually specified multiple times. ⇒ GPU vertex cache gives no benefit.
- No notion of whether two vertices are the same.
- We have no info about which triangle use a specific vertex.
- We have no info about which triangles are adjacent.
Indexed face set

The **indexed face set** is a list of vertices and a list of polygons which indexes the list of vertices.

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

⇒ **indexed face set**.

Examples of file formats that represent geometry as indexed face sets include OpenInventor, VRML and the .msh files on the course web page.
OpenInventor file representing a tetrahedron:

```inventor
#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 ] }
```

In this format, polygonal faces may have any number of vertices, so the vertex indices for every polygon is terminated by $-1$. 
A .msh file representing the same tetrahedron:

4 4

0.0 0.0 1.73205
0.0 1.63299 -0.57735
-1.41421 -0.816497 -0.57735
1.41421 -0.816497 -0.57735

0 1 2
0 2 3
0 3 1
1 3 2

In this, simpler format, we only support triangles, so no \(-1\) is needed.
Indexed face sets are nice, but…

- Finding all triangles sharing a common vertex is $O(N)$.
- Finding all triangles for all vertices is $O(N^2)$.

→ Slow for e.g. $10^6$ vertices.

To avoid this, we store more connectivity information in the data structure, for example:

- Which triangles share a given vertex.
- Which triangles are adjacent (share an edge).
- Which vertices are adjacent to a given vertex.
- …
Consolidation is the process of finding and storing the connectivity of a mesh.

(Or at least some of it.)

We will first look at some data structures that store more connectivity information that the indexed face set, and then at one possible strategy for consolidation.
Triangle based data structure with neighbours

A simple, yet effective, way to encode connectivity is to augment a regular triangle list with pointers to neighbouring triangles.

I.e., the triangle \([v_i, v_j, v_k]\) has three extra fields, pointers to triangles \(t_i, t_j,\) and \(t_k\).

It is common to use the convention that the neighbour \(t_i\) corresponding to \(v_i\) is on the opposite side of the triangle, i.e., connecting to the edge \([v_j, v_k]\).

In addition, each vertex has a pointer to one of the triangles using that vertex. This triangle is known as the leading triangle.
class Triangulation {
    vector<Vertex*> m_vertices;
    vector<Triangle*> m_triangles;
};

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

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

The data structure is something along the lines of:
A note on the use of pointers

It is not necessary to use pointers in the above datastructure or the half-edge structure discussed later.

You need to refer to entities (triangles, vertices etc.) somehow.

Referring by index (like an indexed face set) is a useful alternative.
An alternative is the more flexible **half-edge** data structure.

Each triangle is split into three half-edges, and each half-edge has three pointers:

**Source** is the source node.

**Next** is the next half-edge in the triangle loop.

**Twin** is the twin half-edge in the adjacent triangle.

Each vertex has a pointer to the **leading half-edge**, which one of the half-edges emitting from that vertex.

Usually, we also add a triangle class to hold triangle-specific data.
class Triangulation {
    vector<Vertex*> m_vertices;
    vector<HalfEdge*> m_halfedges;
    vector<Triangle*> m_triangles;
};

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

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

class Triangulation::Triangle {
    HalfEdge* m_leading_halfedge;
};
Example consolidation strategy

For each edge in each triangle (i.e. each half-edge) create a struct

```c
struct foo {
    ...
    int lo;  // vertex index of edge with lowest index
    int hi;  // vertex index of edge with highest index
    Triangle* tri; // pointer to the triangle
};
```

where \(lo < hi\).

Put all these structs into a vector `helper`.

```c
for each triangle t and for each i=0..2:
    helper.push_back(t[i] < t[(i+1) % 3] ?
    foo(t[i], t[(i+1) % 3], t) :
    foo(t[(i+1) % 3], t[i], t));
```
Sort this lexicographically:

```cpp
...  
sort(helper.begin(), helper.end());  
...  
```

This requires a comparison operator to be defined:

```cpp
bool operator<(const foo& a, const foo& b)  
{
    if(a.lo == b.lo)  
        return a.hi < b.hi;  
    return a.lo < a.lo;
}  
```
Then, if two consecutive elements in helper have equal lo and hi, the two triangles the elements points to are adjacent along the edge (lo,hi).

```c
for(int j=0; j<helper.size(); ++j) {
    int i;
    for(i=j; i<helper.size()-1; ++i) {
        if(helper[i] < helper[i+1]) break;
    }
    if(i==j)
        // helper[j] points to an 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;
}
```
Polygonal meshes

Data structures for triangular meshes

Simplification
  Vertex removal
  Edge collapse
  Half-edge collapse
  Fold-overs
  Keeping the graph simple
  Removal criteria
  Implementation

Mesh optimization
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.
1. **Static simplification**: creating separate level of detail models before rendering begins.
2. **Dynamic simplification**: creating a continuous level of detail.
3. **View-dependent simplification**: level of detail various 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.

In dynamic simplification, we store the sequence of point removals. In static simplification, we remove a given amount of vertices for each level-of-detail level.

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.
Vertex removal

Remove a vertex \( p \) and retriangulate the hole.

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.

There are a many possible ways 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 \( q \) and collapses the edge between them to a new point \( r \).

As a result, two triangles adjacent to the edge \([p, q]\) 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 \( q \).
Consider half-edge collapsing \( p \) to \( q \):

Here we get a fold-over (red triangle), which violates the requirement that the intersection between any pair of faces is either a common edge, a common vertex or nothing.
I.e., this produces an illegal mesh.

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 probably an illegal collapse.*
Keeping the graph simple

Consider removing $v_0$, leaving a four-sided hole.

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 $v_4$ since they are already neighbours, and thus, we get two edges connecting the vertices $v_2, v_4$. 
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 possibilities tend to occur more and more frequently as the mesh is simplified.

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.
Removal criteria

The removal criteria associates a score with each vertex, and the vertex with the best score is removed, i.e. the criteria chooses which vertex to remove.

The score is determined by for example:

- Remove that vertex which results in the smallest distance between the previous mesh and the new one.
- Remove vertices where the density of vertices is highest, in order to get an even distribution of vertices.
- Try to maintain good aspect ratios in the triangles (avoiding long thin triangles).

This is particularly important if the mesh is to be used for differential equations, since often error estimates are based on triangle aspect ratio.
Implementation

For our chosen removal criterion we compute, for each vertex $p$, some measure $\text{sig}(p)$ of the how significant the vertex $p$ is.

Once we have $\text{sig}(p)$ for every vertex $p$ in the mesh, we can remove any vertex $p_*$ such that

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

Most (efficient) criteria are local, so that $\text{sig}(p)$ depends only on $p$ and its (immediate) neighbours.
A naive implementation of the decimation algorithm computes $\text{sig}(p)$ for each $p$ in the current mesh after every removal. If there are $N$ vertices in the original mesh, this costs $O(N^2)$ operations.

An enormous improvement is to employ a heap data structure, exploiting the fact that $\text{sig}(p)$ remains constant for the vast majority of the vertices $p$ in the current mesh. A heap is organized as a binary tree, with the vertex $p$ with the smallest $\text{sig}(p)$ value at the root of the heap.

As a preprocess, we compute $\text{sig}(p)$ for all vertices $p$ in the fine mesh and place the vertices $p$ in a heap ordered by the $\text{sig}(p)$ values.

Note that when using half-edge collapses, we could assign a significance to each ordered vertex pair $(p, q)$, and the heap would contain these pairs rather than the vertices themselves.
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 $\text{sig}$ 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.
Polygonal meshes

Data structures for triangular meshes

Simplification

Mesh optimization
- Mesh optimization
- Edge swap
- Delaunay triangulations
- Voronoi diagram
Mesh optimization

Mesh optimization is conceptually simpler than mesh simplification.

**Idea:** 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 \textbf{cost function} we wish to minimize and the \textbf{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.
The in-circle criterion is a common swap criterion for planar triangle meshes:

Swap the edge $[v_1, v_3]$ if it is the diagonal of a convex quadrilateral and $v_3$ lies outside the circumcircle of the triangle $[v_1, v_2, v_4]$. 
The in-circle criterion, which was proposed by Lawson, is equivalent to the max-min angle criterion:

\[ \text{swap if the minimum of the six angles in the two triangles is increased.} \]

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

The optimal triangulation is a Delaunay triangulation:

- the minimum of all the angles of its triangles 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 \( p_1, \ldots, p_N \) 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.,

\[
V_i = \{ x \in \mathbb{R}^2 : \| x - p_i \| \leq \| x - p_j \| \quad \forall j \neq i \}.
\]
The Voronoi diagram and Delaunay triangulation are **duals**: 

The figure shows the Delaunay 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.

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 shows various optimizations of a given toroidal-shaped triangle mesh.