

CS7491 Notes: Tuesday February 3, 2004

Urs Bischoff (urs@cc)

Triangle Meshes

These notes are related to the PowerPoint slides that were presented in class.

Project 2: Triangulation of point clouds in 3D. We should use a similar approach to the 2D case that was shown on the slides. Instead of a disk, we should use a ball that has to intersect three points. The orientation of the triangle is determined by the middle point of the ball.

How should we pick a radius for the disk / ball? Choosing a random radius would be one possibility. But most probably it is better to determine a radius in a pre-processing step. We could compute the average of the average distances to the nearest three neighbors of all points.

Somebody asked if the computation of the three nearest neighbors isn't already part of the original problem that has to be solved? This question could be rephrased: let's assume we have an oracle that knows the nearest neighbor of a point for zero cost. Would this help us for the algorithm? It may help, but it is not so clear how, because we don't know anything about other neighbor points.

Corners, incidence and adjacency: So far we have only generated a triangle soup, i.e. triangles that don't know anything about each other. We want to have connectivity information.

Spanning Trees: There is a relation between the VST and the TST. They are dual, which means that an edge is either in the VST or in the TST. I can cut the mesh at edges and build the VST. I can triangulate this tree; this means that I add the triangles that belong to these cut edges. The spanning tree of the dual graph of this mesh represents the TST. This idea also leads to a proof of the Euler formula. Because we know that an edge of a triangle mesh is either an edge in the VST ("cut edges") or in the TST ("unfolding edges"), we just have to count all edges in both graphs to get the total number of edges in the triangle mesh. Thus, we get $E = (V-1) + (T-1)$. The rest of the proof can be found on the slides.

Connectivity of triangles: As stated above, a triangle soup is normally not a useful representation. It is expensive and we have no connectivity information. One way to represent the connectivity is a **triangle strip**. Each triangle that is connected to an existing triangle strip is defined by an additional vertex. Graphic API's (e.g. OpenGL) normally support triangle strips. However, some API's only allow adding one on the left side, then one on the right, then on the left etc. So the question is if it is always possible to create a strip with left/right/left/right/...

V-Table: A straightforward way to represent the vertices would be an $n \times 3$ -array. Each row in this array contains the three vertices of a triangle. However, we want to use a $3 \times l$ -

array. In this array, the three vertices of a triangle are in consecutive entries in the array. Why do we want to organize it like this (actually both tables contain the same information)? The advantage of the second approach is that we easily get unique identifier for the corners.

O-Table: How can we walk from one triangle to the next triangle? We do it by walking from one corner c to the opposite corner $c.o$. The opposite corners can be found in the O-table.

This O-table has to be precomputed. How can we do this? One possibility is to compare all edges between two corners. If the edges of two corner pairs are the same, we know that they share an edge and we can add an entry to the corner table. Unfortunately, this easy algorithm is an $O(n^2)$ -algorithm.

A more efficient algorithm was presented in class. That algorithm uses bucket sort, to sort (vertex, vertex, corner)-triples. For normal triangle meshes the buckets should be very small, because a vertex normally has a limited number of incident triangles. So, bucket sort should be an efficient sorting algorithm.

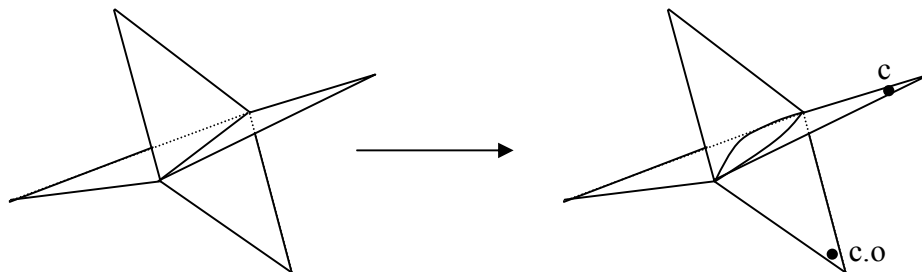
Non-manifold meshes: How can you use the O-table if you have a non-manifold mesh? In a non-manifold mesh the opposite corner can be ambiguous. We can find out whether a mesh is non-manifold when we compute the O-table:

If a mesh is manifold, we only find pairs c and $c.o$.

If we can not pair a corner with an opposite corner, the triangle has a border edge.

If we find more than two opposite corners, we have a non-manifold edge.

Make a T-mesh an orientable manifold: Problematic areas (non-manifold areas) are self-intersections of the mesh. One way to solve this problem is to cut the edges at self-intersections. The following illustration shows the idea. Of course, we do not change the geometry of the vertices. We just assign one unambiguous opposite corner to a corner.



(Jarek's paper "Matchmaker" discusses this problem)

In our project it can happen that we have the same triangle twice. It is possible that we triangulate three points once from “inside” and once from “outside”. This means that each side of such a triangle has its own corners.