1 Triangle Soup

Each triangle has three vertices. Each vertex has three coordinates. Let's represent the triangles using an array with n rows. In i-th row, list the coordinates of the three vertices bounding the i-th triangle. That's the total of 9 floats per triangle.

2 Triangle table, vertex table, adjacency table

Triangle soup works OK in some cases (e.g. if all you care about is just to render the triangles) but is inconvenient if the triangles are to be processed in any way. Almost any surface processing algorithm requires many accesses to neighbors of triangles or vertices of the mesh. Finding e.g. triangles that share an edge with a given triangle in a triangle soup is expensive: basically, it requires exhaustive search over all triangles. Therefore, in most cases, representations that support efficient neighborhood queries are preferred over the plain triangle soup representation.

2.1 Manifold Meshes

Most of the time, we will be interested in nice, regular tilings of surfaces bounding 3D volumes. In this case, it is desirable to make sure the triangles form a manifold mesh. A mesh is *manifold* if all its vertices are manifold vertices. A manifold vertex is a vertex such that its incident triangles form a cyclic fan around it. A mesh is *manifold with boundary* if all vertices are manifold or boundary manifold. A boundary manifold vertex is a vertex whose incident triangles form a (not necessarily circular) fan. See Figure 2 for examples of manifold and non-manifold vertices.

Note that in a manifold mesh no edge can have more than two incident triangles.

2.2 Tables

A possible representation of a manifold mesh which provides easy access to neighbors consists of three tables:

- **Vertex table.** It lists all vertices of the mesh, three floats (coordinates) per row. It has as many rows as there are vertices. Clearly, it can store more per-vertex info if required in a particular application (like normals, texture coordinates etc). Note that by listing the vertices in this table, we give them unique numbers (identifiers) between 0 and $m-1$ (m=number of vertices).

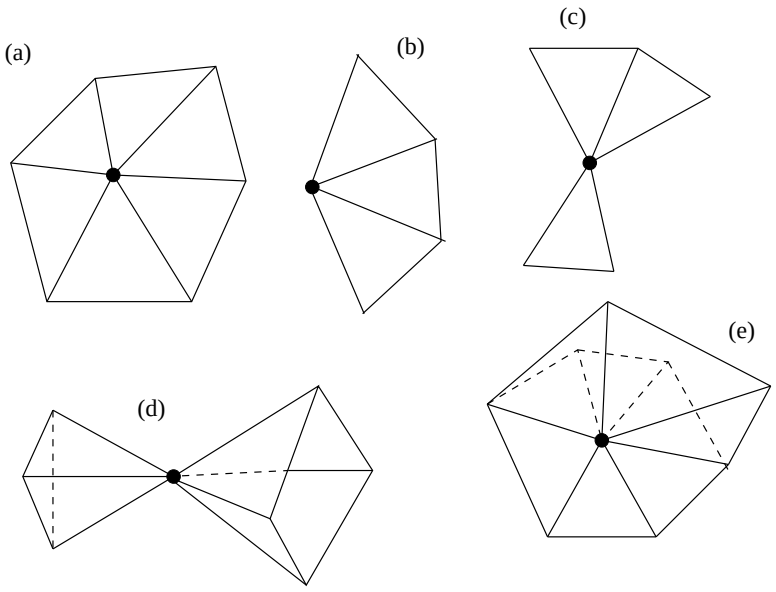


Figure 1: (a) a manifold vertex, (b) a boundary manifold vertex (c)-(e) non-manifold vertices

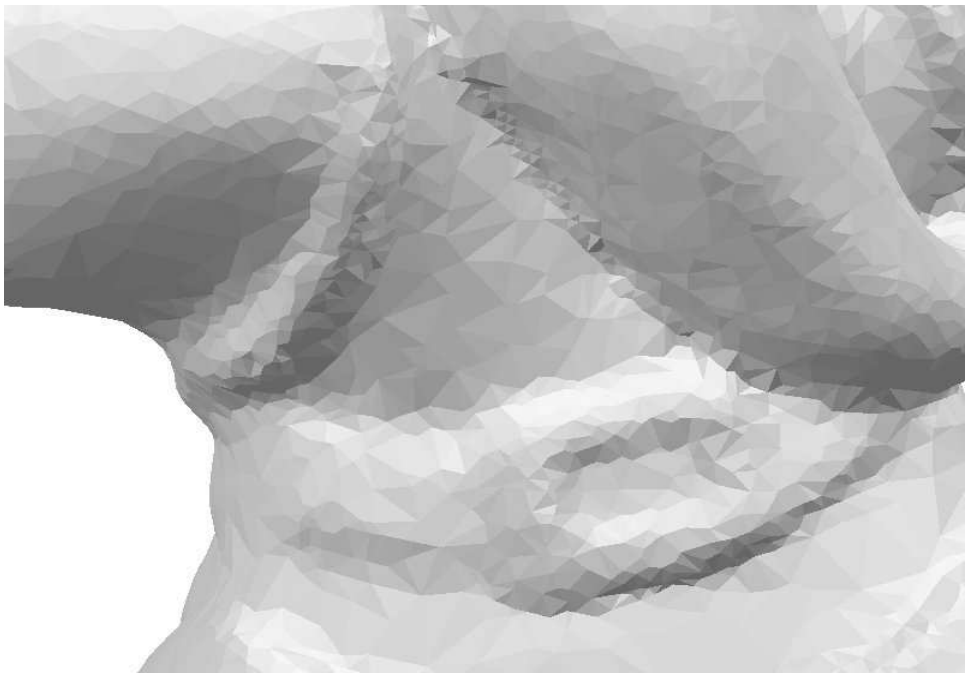
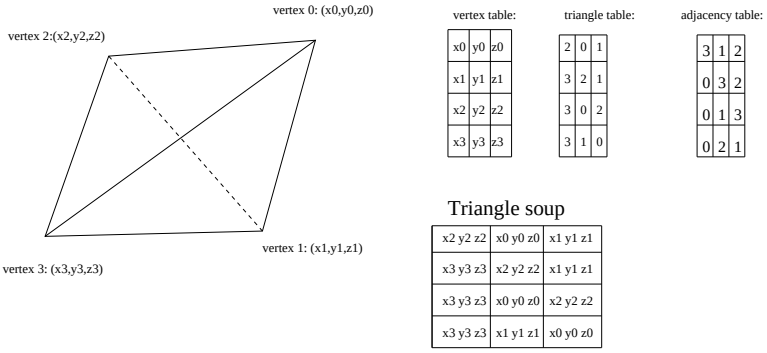


Figure 2: A zoom into a manifold mesh

- **Triangle table.** For each triangle, we store three integers, the IDs of its vertices. They can be thought of as indices into the vertex table. Again, this table induces a numbering of triangles. For each triangle, it also fixes an ordering of its vertices (this is the order in which they appear in the corresponding row).

- **Adjacency table.** The dimensions of this table are the same as the dimensions of the triangle table. In row i and column j, it keeps the ID of the triangle adjacent to triangle i across the edge opposite to its j-th vertex.

Here is how the tables look for the boundary of a tetrahedron (4 triangles):



2.3 Triangle table from triangle soup

Constructing just any triangle table from triangle soup is easy: just read the vertices as they appear in the soup table and put them in this order into the vertex table. Then, just use the consecutive integers as entries of the triangle table. For the tetrahedron mesh, this would lead to tables

$$\text{Vertex table}: \begin{bmatrix} x_2 & y_2 & z_2 \\ x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_3 & y_3 & z_3 \\ x_2 & y_2 & z_2 \\ x_1 & y_1 & z_1 \\ x_3 & y_3 & z_3 \\ x_0 & y_0 & z_0 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_1 & y_1 & z_1 \\ x_0 & y_0 & z_0 \end{bmatrix}, \qquad \text{triangle table}: \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}.$$

Even though this is a valid way of converting a triangle soup to triangle table, it misses the purpose of triangle table. It is very uneconomical (there are a lot

of repetitions in the vertex table). Also, the triangle table itself does not tell much about the structure of the mesh (in particular, it's impossible to figure out which triangles are neighbors without peeking into the vertex table). This is why we'll want to build the vertex/triangle tables so that the no two rows in the vertex table are the same (every vertex appears in it just once).

How to achieve this efficiently? An (worst-case optimal) $O(n \log n)$ algorithm works as follows. Build a new table $\mathcal{T}$, with three times as many rows as there are triangles in the soup: scan the rows of the soup table and generate three rows of $\mathcal{T}$ for each row. If row i of the soup table is

$$\left[\begin{array}{ccc} (x_i, y_i, z_i) & (x'_i, y'_i, z'_i) & (x''_i, y''_i, z''_i) \end{array} \right]$$

then the three corresponding rows of $\mathcal{T}$ will be

$$\left[\begin{array}{ccccc} x_i & y_i & z_i & i & 0 \\ x'_i & y'_i & z'_i & i & 1 \\ x''_i & y''_i & z''_i & i & 2 \end{array} \right].$$

Thus, the first three columns of $\mathcal{T}$ hold the coordinates of the consecutive vertices which appear in the soup table (in fact they are the same as the uneconomical vertex table we constructed previously). For each row, the last two entries are essentially the 'address' (in the soup table) of the vertex specified by the first three entries. Now, take the table $\mathcal{T}$ and sort its rows lexicographically. As a result of sorting, vertices with the same coordinates will end up in adjacent rows (since coordinates are most significant entries in each row). Now, read the rows of the sorted table. For each row, use the current vertex ID (initialized to zero at startup) as entry (i, j) of the triangle table (where i and j are the fourth and fifth entries in the row). Increment the current vertex ID every time you move to a new row and discover that its first three entries are different from the first three entries of the previous row.

2.4 Adjacency table from triangle table

Of course, the minimal assumption we have to make to be able to build adjacency table at all is that every edge has at most two incident triangles. Otherwise, we may have more than one neighbor of a triangle across a specific edge and we may need to use list of adjacent triangles rather than just one entry in the adjacency table. Adjacency table can be constructed from triangle table in linear time. This can be done in two steps.

First, one constructs a list of incident triangles for each vertex. This can be done by initializing the incident lists to null and then, for each triangle, adding it to the incidence lists of all its vertices.

Having constructed the incidence lists, we fill the adjacency table. The idea is to go over the vertices and, for each vertex v, use its incident triangle list to figure out which pairs of triangles are adjacent along an edge whose one endpoint is v. Notice that we already know that all triangles in the v's incidence list have v as a vertex. We need to compute pairs of triangles belonging to that list

that share one more vertex. To do that, we'll utilize lists of triangles (we'll call them just 'lists' as opposed to 'incidence lists' that were constructed in step 1), allocated and initialized (only once) to empty at startup (one for each vertex). Go over all triangles in v's incidence list and, for each triangle, add it to the lists of its two vertices other than v. Whenever we add a second triangle to a list of a vertex, call it w, we have determined a pair of adjacent (across the edge $v\bar{w}$) triangles, and we can fill in two entries of the adjacency table. Note that no list will ever get longer than two (this would mean that there is an edge with 3 incident triangles). After we have processed all triangles in the v's incidence list, we can clear the lists (by going over all triangles in v's incidence list again and clearing the lists of its vertices other than v).

The first step clearly requires linear time. The second stage requires time proportional to the total length of all incidence lists. Since each triangle belongs to three incidence lists (the ones corresponding to its vertices only), its total cost is also linear.