

1 Triangles, arrays etc

A 3D point can be represented as three floats (x,y and z coordinates). To specify a triangle we may just list its vertices (3 points/9 floats total). In case several triangles are needed, we can arrange all this information in a table, whose each row lists three vertices bounding a triangle.

Alternatively, we can use two tables, one for storing all the vertex coordinates and the other – for storing identifiers of triples of vertices bounding a triangle. Here is an example:

The second way:

vertex table:		
x0	y0	z0
x1	y1	z1
x2	y2	z2
x3	y3	z3

triangle table:		
2	0	1
3	2	1
3	0	2
3	1	0

adjacency table:		
3	1	2
0	3	2
0	1	3
0	2	1

The first way

x2 y2 z2	x0 y0 z0	x1 y1 z1
x3 y3 z3	x2 y2 z2	x1 y1 z1
x3 y3 z3	x0 y0 z0	x2 y2 z2
x3 y3 z3	x1 y1 z1	x0 y0 z0

1.1 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 manifold triangle mesh satisfies the following two conditions:

1. Every edge has exactly two incident triangles
2. Triangles incident upon a vertex form a circular 'fan'.

Manifold meshes are easier to deal with than general ones. For example, storing their adjacency information (see next section) becomes much easier: after all, we know that there is only one triangle adjacent to any other one across an edge.

1.2 Adjacency

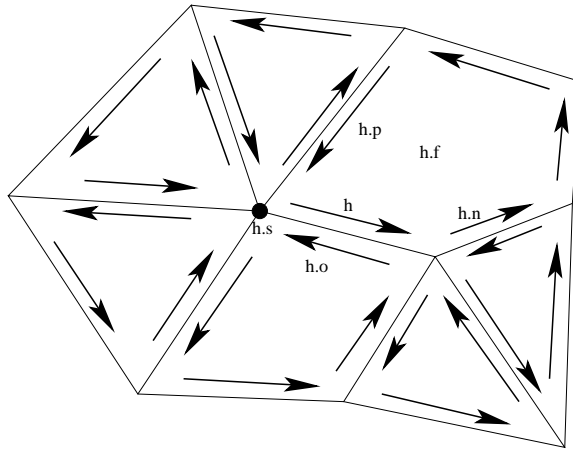
Sometimes it is desirable to have the adjacency information for our mesh handy. In such cases, we may additionally want to use the adjacency table as a part of our representation. The adjacency table lists, for each triangle, the identifiers of triangles adjacent to each triangle in the mesh. An example is shown in the

Figure. For example, the top row of the adjacency table contains 3, 1 and 2. This means that the triangle adjacent across the edge opposite to the first vertex of the triangle 0 is triangle 3. From the triangle table, we can read that it has vertices 3, 1 and 0 and therefore indeed shares the vertices 0 and 1 with the triangle 0.

Similarly, for example, the ID of the triangle adjacent to the triangle 2 across the edge opposite to its third vertex can be immediately found from the adjacency table: it's 3 and the vertices shared by the two triangles are vertex 3 and 0 (the first two vertices of triangle 2).

2 Half-edge data structure

It is something that works also for polygons, not only for triangles. The basic building block can be viewed as an arrow running along an edge. The way the arrows are constructed is as follows. For each face (polygon) we draw arrows around that face (along the 'inside' side of edges) in the counter-clockwise direction. This leads to an arrangement of arrows like that shown in the Figure below.



Notice that all the edges in the interior (i.e. having two faces on both sides) have two arrows along them, running in different directions. This is where the name half-edge comes from (edge=two arrows, so an arrow=edge/2 :)). Every exterior edge (i.e. one having just one face next to it) has only one arrow next to it. Each of the half edges h is stored as a record, having the following fields (see the figure above):

1. the reference to the next half edge, $h.n$
2. the reference to the previous half-edge, $h.p$
3. the reference to the opposite half-edge, $h.o$ (**nil** if there is no opposite)

4. the reference to the starting vertex, *h.s*
5. the reference to its face, *h.f*.

Besides half-edges, we also want to store vertex data (normals, colors etc), using a specially designed record data structure. One of the fields we are going to include in the vertex data structure is a reference to one of the half edges out of that vertex (say, *v.h*). The face record, used to store the faces and their properties, will include a reference to one of the half-edges around the face as one of the fields (*f.h*).

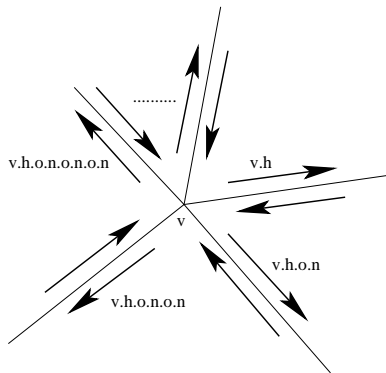
There a lot of things which can be done with the half-edge data structure. For example, list *all* the half edges around a vertex *v*:

```

i := v.h;
do
  output(i); i := i.o.n;
until i=v.h;

```

How does it work? Here is a figure:



You are encouraged to write a similar loop which, for example:

1. outputs all vertices adjacent to a given one
2. outputs all faces adjacent to a given face across an edge
3. outputs all half edges bounding a face