The Half-Edge Data Structure

By Max McGuire
07 August 2000

Introduction

A common way to represent a polygon mesh is a shared list of vertices and a list of faces storing pointers for its vertices. This representation is both convenient and efficient for many purposes, however in some domains it proves ineffective.

Mesh simplification, for example, often requires collapsing an edge into a single vertex. This operation requires deleting the faces bordering the edge and updating the faces which shared the vertices at end points of the edge. This type of polygonal "surgery" requires us to discover adjaceny relationships between components of the mesh, such as the faces and the vertices. While we can certainly implement these operations on the simple mesh representation mentioned above, they will most likely be costly; many will require a search through the entire list of faces or vertices, or possibly even both.

Other types of adjacency queries on a polygon mesh include:

Which faces use this vertex?
Which edges use this vertex?
Which faces border this edge?
Which edges border this face?
Which faces are adjacent to this face?

To implement these types of adjacency queries efficiently, more sophisticated boundary representations (b-reps) have been developed which explicitly model the vertices, edges, and faces of the mesh with additional adjacency information stored inside.

One of the most common of these types of representations is the winged-edge data structure where edges are augmented with pointers to the two vertices they touch, the two faces bordering them, and pointers to four of the edges which emanate from the end points. This structure allows us to determine which faces or vertices border an edge in constant time, however other types of queries can require more expensive processing.

The half-edge data structure is a slightly more sophisticated b-rep which allows all of the queries listed above (as well as others) to be performed in constant time (*). In addition, even though we are including adjacency information in the faces, vertices and edges, their size remains fixed (no dynamic arrays are used) as well as reasonably compact.

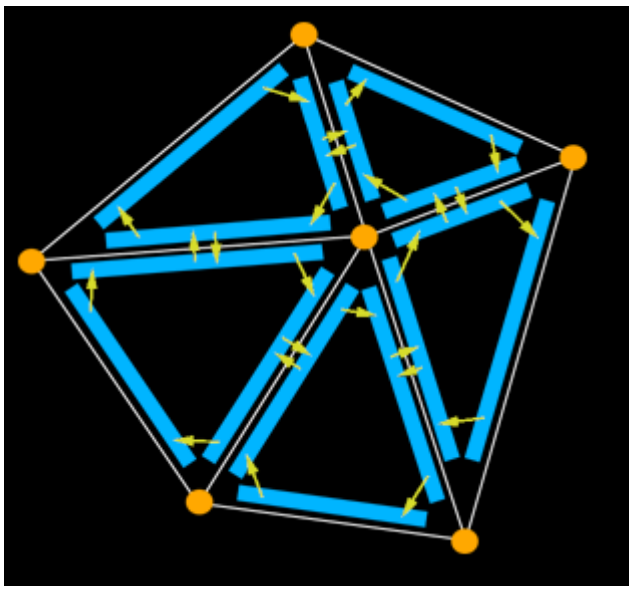
These properties make the half-edge data structure an excellent choice for many applications, however it is only capable of representing manifold surfaces, which in some cases can prove prohibitive. Mathematically defined, a manifold is a surface where every point is surrounded by a small area which has the topology of a disc. For the purpose of a polygon mesh, this means that every edge is bordered by exactly two faces; t-junctions, internal polygons, and breaks in the mesh are not allowed.

(*) More precisely, constant time per piece of information gathered. For instance when querying all edges adjacent to a vertex, the operation will be linear in the number of edges adjacent to the vertex, but constant time per-edge.

Structure

The half-edge data structure is called that because instead of storing the edges of the mesh, we store half-edges. As the name implies, a half-edge is a half of an edge and is constructed by splitting an edge down its length. We'll call the two half-edges that make up an edge a pair. Half-edges are directed and the two edges of a pair have opposite directions.

The diagram below shows a small section of a half-edge representation of a triangle mesh. The yellow dots are the vertices of the mesh and the light blue bars are the half-edges. The arrows in the diagram represent pointers, although in order to keep the diagram from getting too cluttered, some of them have been ommited.



As you can see in the diagram, the half-edges that border a face form a circular linked list around its perimeter. This list can either be oriented clockwise or counter-clockwise around the face just as long as the same convention is used throughout. Each of the half-edges in the loop stores a pointer to the face it borders (not shown in the diagram), the vertex at its end point (also not shown) and a pointer to its pair. It might look something like this in C:

```
struct HE_edge
{
    HE_vert* vert;   // vertex at the end of the half-edge
    HE_edge* pair;   // oppositely oriented adjacent half-edge
    HE_face* face;   // face the half-edge borders
    HE_edge* next;   // next half-edge around the face
};
```

Vertices in the half-edge data structure store their x, y, and z position as well as a pointer to exactly one of the half-edges, which use the vertex as its starting point. At any given vertex there will be more than one half-edge we could choose for this, but we only need one and it doesn't matter which one it is. We'll see why later on when the querying methods are explained. In C the vertex structure looks like this:

```
struct HE_vert
{
   float x;
   float y;
   float z;

   HE_edge* edge;  // one of the half-edges emantating from the vertex
};
```

For a bare-bones version of the half-edge data structure, a face only needs to store a pointer to one of the half-edges which borders it. In a more practical implementation we'd probably store information about textures, normals, etc. in the faces as well. The half-edge pointer in the face is similar to the pointer in the vertex structure in that although there are multiple half-edges bordering each face, we only need to store one of them, and it doesn't matter which one. Here's the face structure in C:

```
struct HE_face
{
   HE_edge* edge;  // one of the half-edges bordering the face
};
```

Adjacency Queries

The answers to most adjacency queries are stored directly in the data structures for the edges, vertices and faces. For example, the faces or vertices which border a half-edge can easily be found like this:

```
HE_vert* vert1 = edge->vert;
HE_vert* vert2 = edge->pair->vert;

HE_face* face1 = edge->face;
HE_face* face2 = edge->pair->face;
```

A slightly more complex example is iterating over the half edges adjacent to a face. Since the half-edges around a face form a circular linked list, and the face structure stores a pointer to one of these half-edges, we do it like this:

```
HE_edge* edge = face->edge;
do {
   // do something with edge
   edge = edge->next;
} while (edge != face->edge);
```

Similarly, we might be interested in iterating over the edges or faces which are adjacent to a particular vertex. Referring back to the diagram, you may see that in addition to the circular linked lists around the borders of the faces, the pointers also form loops around the vertices. The iterating process is the same for discovering the adjacent edges or faces to a vertex; here it is in C:

```
HE_edge* edge = vert->edge;
do {
    // do something with edge, edge->pair or edge->face
    edge = edge->pair->next;
} while (edge != vert->edge);
```

Note that in these iterating examples checks for null pointers are not included. This is because of the restriction on the surface being manifold; in order for this requirement to be fulfilled, all of the pointers must be valid.