Mesh Data Structures

The efficiency and memory consumption of the geometric modeling algorithms presented in this book largely depend on the underlying surface mesh data structures. This chapter provides a brief overview of the most common data structures of the wide variety described in the literature.

Choosing a mesh data structure requires taking into account topological as well as algorithmic considerations:

- Topological requirements. Which kinds of meshes need to be represented by the data structure? Can we rely on 2-manifold meshes, or do we need to represent complex edges and singular vertices (see Section 1.3.3)? Can we restrict ourselves to pure triangle meshes, or do we need to represent arbitrary polygonal meshes? Are the meshes regular, semi-regular, irregular (see Chapter 6)? Do we want to build up a hierarchy of increasingly refined meshes (see Section 1.3.2)?

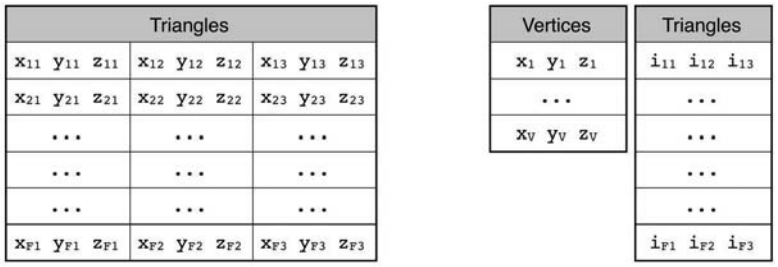
- Algorithmic requirements. Which kinds of algorithms will be operating on the data structure? Do we simply want to render the mesh, or do we need efficient access to local neighborhoods of vertices, edges, and faces? Will the mesh be static or will its geometry and/or connectivity change over time? Do we need to associate additional data with vertices, edges, and faces of the mesh? Do we have special requirements in terms of memory consumption (i.e., are the data sets massive)?

Evaluating a data structure requires measuring various criteria such as (a) time to construct it during preprocessing, (b) time to answer a specific query, (c) time to perform a specific operation, and (d) memory consumption and redundancy. While it is not uncommon to design a data structure specialized to a particular algorithm, there are a number of data structures common to several algorithms in geometry processing, which we review in this chapter.

2.1 Face-Based Data Structures

The simplest way to represent a surface mesh consists of storing a set of *individual* polygonal faces represented by their vertex positions (the so-called *face-set*). For the simpler case of triangular meshes this requires storing three vertex positions per face (see Figure 2.1 (left)). Using 32-bit single precision numbers to represent vertex coordinates, this requires $3 \cdot 3 \cdot 4 = 36$ bytes per triangle. Since due to Euler's formula (Equation (1.5)) the number of faces F is about twice the number of vertices V, this data structure consumes on average 72 bytes/vertex. As it does not represent the mesh connectivity, it is commonly referred to as *triangle soup* or polygon soup. Some data exchange formats, such as stereolithography (STL), use this representation as a common denominator.

However, this representation is not sufficient for most applications: connectivity information cannot be accessed explicitly, and vertices and associated data are replicated as many times as the degree of the vertices. The latter redundancy can be avoided by a so-called *indexed face set* or *shared-vertex* data structure, which stores an array of vertices and encodes polygons as sets of indices into this array (see Figure 2.1 (right)). For the case of triangle meshes, and using 32 bits to store vertex coordinates and face indices, this representation requires 12 bytes for each vertex and for each triangle, i.e., it consumes on average 12 bytes/vertex + 12 bytes/face = 36 bytes/vertex, which is only half of the face-set structure.



Triangles		
x_{11} y_{11} z_{11}	x_{12} y_{12} z_{12}	x_{13} y_{13} z_{13}
x_{21} y_{21} z_{21}	x_{22} y_{22} z_{22}	x_{23} y_{23} z_{23}
...	...	...
...	...	...
...	...	...
x_{F1} y_{F1} z_{F1}	x_{F2} y_{F2} z_{F2}	x_{F3} y_{F3} z_{F3}

Vertices
x_1 y_1 z_1
...
x_V y_V z_V

Triangles
i_{11} i_{12} i_{13}
...
...
...
...
i_{F1} i_{F2} i_{F3}

Figure 2.1. Face-set data structure (left) and indexed face-set data structure (right) for triangle meshes.

Because it is simple and efficient in storage, this representation is used in many file formats such as OFF, OBJ, and VRML. Similarly, it is relevant for a class of efficient rendering algorithms that assume static data (OpenGL vertex arrays; see [Shreiner and Khronos OpenGL ARB Working Group 09]).

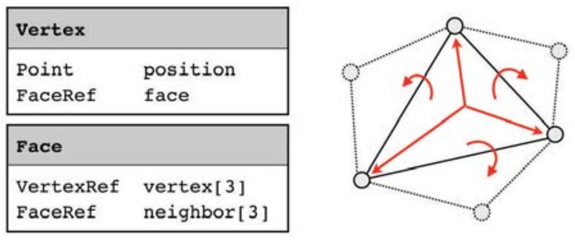
However, without additional connectivity information, this data structure requires expensive searches to recover the local adjacency information of a vertex and hence is not efficient enough for most algorithms.

This is a minimal set of operations frequently used by most algorithms:

- Access to individual vertices, edges, and faces. This includes the enumeration of all elements in unspecified order.
- Oriented traversal of the edges of a face, which refers to finding the next edge (or previous edge) in a face. With additional access to vertices, for example, the rendering of faces is enabled.
- Access to the incident faces of an edge. Depending on the orientation, this is either the left or right face in the manifold case. This enables access to neighboring faces.
- Given an edge, access to its two endpoint vertices.
- Given a vertex, at least one incident face or edge must be accessible. Then for manifold meshes all other elements in the so-called *one-ring neighborhood* of a vertex can be enumerated (i.e., all incident faces or edges and neighboring vertices).

These operations, which enable both local and global traversal of the mesh, relate vertices, edges, and faces of the mesh by connectivity information (and orientation).

We now review several data structures devised for fast traversal of surface meshes.



```
Vertex
Point       position
FaceRef     face

Face
VertexRef   vertex[3]
FaceRef     neighbor[3]
```

Figure 2.2. Connectivity information stored in a face-based data structure.

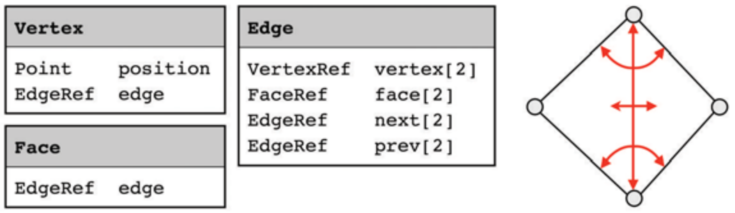
A standard face-based data structure for triangle meshes that includes connectivity information consists of storing, for each face, references to its three vertices as well as references to its neighboring triangles. Each vertex stores a reference to one of its incident faces in addition to its 3D position (see Figure 2.2). Based on this connectivity information one can circulate around a vertex in order to enumerate its one-ring neighborhood, and perform all other operations listed above. This representation is used, for instance, for the 2D triangulation data structures of CGAL [CGAL 09] and consumes only 24 bytes/face + 16 bytes/vertex = 64 bytes/vertex.

However, this data structure also has some drawbacks. First, it does not explicitly store edges, so, for example, no data can be attached to edges. Second, enumerating the one-ring of a center vertex requires a large number of case distinctions (is the center vertex the first, second, or third vertex of the current triangle?). Finally, if this data structure is to be used for general polygonal meshes, the data type for faces no longer has constant size, which makes the implementation more complex and less efficient.

2.2 Edge-Based Data Structures

Data structures for general polygon meshes are logically edge-based, since the connectivity primarily relates to the mesh edges. Well-known edge-based data structures are the *winged-edge* [Baumgart 72] and *quad-edge* [Guibas and Stolfi 85] data structures in different variants (see, for instance, [O'Rourke 94]).

The winged-edge structure is depicted in Figure 2.3. Each edge stores references to its endpoint vertices, to its two incident faces, and to the next and previous edge within the left and right face, respectively. Vertices and faces store a reference to one of its incident edges. In total, this leads to a memory consumption of 16 bytes/vertex + 32 bytes/edge + 4 bytes/face = 120 bytes/vertex (since $F \approx 2V$ and $E \approx 3V$ due to the Euler formula in Equation (1.5)).



```
Vertex
Point      position
EdgeRef    edge

Face
EdgeRef    edge

Edge
VertexRef  vertex[2]
FaceRef    face[2]
EdgeRef    next[2]
EdgeRef    prev[2]
```

Figure 2.3. Connectivity information stored in an edge-based data structure.

While an edge-based data structure can represent arbitrary polygonal meshes, traversing the one-ring still requires case distinctions (is the center vertex the first or second vertex of an edge?). This issue is finally addressed by halfedge data structures, as described in the next section.

2.3 Halfedge-Based Data Structure

Halfedge data structures [Mantyla 88, Kettner 99] avoid the case distinctions of edge-based data structures by splitting each (unoriented) edge into two oriented *halfedges*, as depicted in Figure 2.4. This data structure is able to represent arbitrary polygonal meshes that are subsets of orientable (combinatorial) 2-manifolds (no complex edges and vertices, see Figure 1.6).

In a halfedge data structure, halfedges are oriented consistently in counterclockwise order around each face and along each boundary. Each boundary may therefore be seen here as an empty face of potentially high degree. As a by-product, each halfedge designates a unique corner (a non-shared vertex in a face) and hence attributes such as texture coordinates or normals can be stored per corner.

For each halfedge we store a reference to

- the vertex it points to,
- its adjacent face (a zero pointer if it is a boundary halfedge),
- the next halfedge of the face or boundary (in counterclockwise direction),
- the previous halfedge in the face, and
- its opposite (or inverse) halfedge.

Note that the opposite halfedge does not have to be stored if two opposing halfedges are always grouped in pairs and stored in subsequent array

Vertex	
Point	position
HalfedgeRef	halfedge

Face	
HalfedgeRef	halfedge

Halfedge	
VertexRef	vertex
FaceRef	face
HalfedgeRef	next
HalfedgeRef	prev
HalfedgeRef	opposite

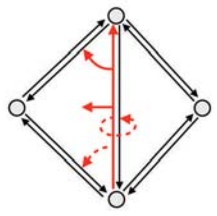


Figure 2.4. Connectivity information stored in an halfedge-based data structure.

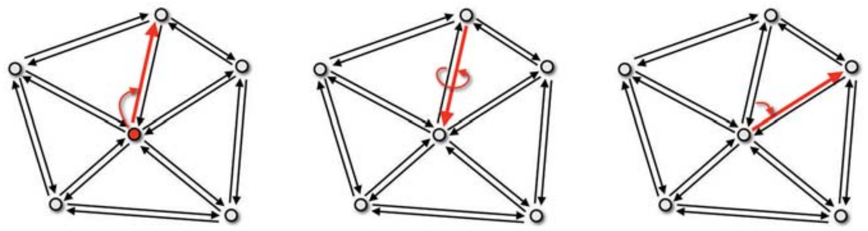


Figure 2.5. The one-ring neighbors of the center vertex can be enumerated by starting with an outgoing halfedge of the center (left), and then repeatedly rotating clockwise by stepping to the opposite halfedge (center) and next halfedge (right) until the first halfedge is reached again.

locations `halfedges`[`i`] and `halfedges`[`i+1`]. The opposite halfedge is then given implicitly by addition modulo 2. Moreover, we obtain an explicit representation for "full" edges as a pair of two halfedges, which is important when we want to associate data with edges rather than with halfedges. The reference to the previous halfedge in a face may also be omitted, since it can be found by stepping along the next halfedge references.

Additionally, each face stores a reference to one of its halfedges, and each vertex stores an outgoing halfedge. Since the number of halfedges H is about six times the number of vertices V, the total memory consumption is 16 bytes/vertex + 20 bytes/halfedge + 4 bytes/face = 144 bytes/veretx. Not explicitly storing the previous and opposite halfedge reduces the memory costs to 96 bytes/vertex.

A halfedge data structure enables us to enumerate for each element (i.e., vertex, edge, halfedge, or face) all of its adjacent elements. In particular, the one-ring neighborhood of a given vertex can now be enumerated without inefficient case distinctions, as shown in Figure 2.5 and in the pseudocode below.

```
void enumerate\_one\_ring(VertexRef center, Function func)
{
HalfedgeRef h = outgoing\_halfedge(center);
HalfedgeRef hstop = h;
  do {
    VertexRef v = vertex(h);
    func(v); // process vertex v
    h = next\_halfedge( opposite\_halfedge(h) );
  } while (h != hstop);
}
```

The implementation of the references (e.g., `HalfedgeRef`) can be realized, for instance, by using pointers or indices. In practice, index representations (see, e.g., Section 2.4) are more flexible even though memory access is indirect: using indices into data arrays enables efficient memory relocation (and simpler and more compact memory management) and *all* attributes of a vertex (edge, halfedge, face) can be identified by the same index.

2.4 Directed-Edge Data Structure

The directed-edge data structure [Campagna et al. 98] is a memory-efficient variant of the halfedge data structure that is particularly designed for triangle meshes. It is based on indices that reference each element in the mesh (vertex, face, or halfedge). The indexing follows certain rules that *implicitly* encode some of the connectivity information of the triangle mesh. Instead of pairing opposite halfedges (as proposed in the previous section), this data structure groups the three halfedges belonging to a common triangle.

To be more precise, let f be the index of a face. Then, the indices of its three halfedges are given as

$$\text{halfedge}(f, i) = 3f + i, \quad i = 0, 1, 2.$$

Now let h be the index of a halfedge. Then, the index of its adjacent face and its index within that face are simply given by

$$\text{face}(h) = h/3, \qquad \text{face\_index}(h) = h \bmod 3.$$

The index of h's next halfedge can be computed as $(h + 1) \bmod 3$. The remaining parts of the connectivity have to be stored explicitly: each vertex stores its position and the index to an outgoing halfedge; each halfedge stores the index of its opposite halfedge and the index of its vertex. This leads to a memory consumption of only 16 bytes/vertex + 8 bytes/halfedge = 64 bytes/vertex, which is just as much as the simple face-based structure of Section 2.1, although the directed edges data structure offers much more functionality.

Directed edges can represent all triangle meshes that can be represented by a general halfedge data structure. Note, however, that the boundaries are handled by special (e.g., negative) indices indicating that the opposite halfedge is invalid. Traversing boundary loops is more expensive, since there is no atomic operation to enumerate the next boundary edge. For a general halfedge structure, this can efficiently be accessed by the `next` halfedge along the boundary.

Although we have here described the directed-edge data structure for pure triangle meshes, an adaption to pure quad meshes is straightforward. However, it is not possible to mix triangles and quads, or to represent general polygonal meshes. The main benefit of directed edges is its memory efficiency. Its drawbacks are (a) the restriction to pure triangle/quad meshes and (b) the lack of an explicit representation of edges.

2.5 Summary and Further Reading

Carefully designed data structures are central for geometry processing algorithms based on polygonal meshes. For most algorithms presented in this book we recommend halfedge data structures, or directed-edge data structures as a special case for triangle meshes. While implementing such data structures may look like a simple programming exercise at a first glance, it is actually much harder to achieve a good balance between versatility, memory consumption, and computational efficiency. For those reasons we recommend using existing implementations that provide a number of generic features and that have been matured over time. Some of the publicly available implementations include CGAL,[1] OpenMesh,[2] and MeshLab.[3]

For a detailed overview and comparison of different mesh data structures we refer the reader to [Kettner 99], and to [Floriani and Hui 03, Floriani and Hui 05] for data structures for *non-manifold* meshes. For further reading there are a number of data structures that are specialized for a variety of tasks and size of data, such as processing massive meshes [Isenburg and Lindstrom 05] and view-dependent rendering of massive meshes [Cignoni et al. 04]. Finally, we point the reader to data structures that offer a trade-off between low memory consumption and full access [Kallmann and Thalmann 01, Aleardi et al. 08].

[1]CGAL: http://www.cgal.org
[2]OpenMesh: http://www.openmesh.org
[3]MeshLab: http://www.meshlab.org