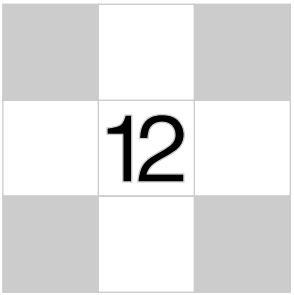


Data Structures for Graphics

Certain data structures seem to pop up repeatedly in graphics applications, perhaps because they address fundamental underlying ideas like surfaces, space, and scene structure. This chapter talks about several basic and unrelated categories of data structures that are among the most common and useful: mesh structures, spatial data structures, scene graphs, and tiled multidimensional arrays.

For meshes, we discuss the basic storage schemes used for storing static meshes and for transferring meshes to graphics APIs. We also discuss the winged-edge data structure (Baumgart, 1974) and the related half-edge structure, which are useful for managing models where the tessellation changes, such as in subdivision or model simplification. Although these methods generalize to arbitrary polygon meshes, we focus on the simpler case of triangle meshes here.

Next, the scene-graph data structure is presented. Various forms of this data structure are ubiquitous in graphics applications because they are so useful in managing objects and transformations. All new graphics APIs are designed to support scene graphs well.

For spatial data structures, we discuss three approaches to organizing models in 3D space—bounding volume hierarchies, hierarchical space subdivision, and uniform space subdivision—and the use of hierarchical space subdivision (BSP trees) for hidden surface removal. The same methods are also used for other purposes including geometry culling and collision detection.

Finally, the tiled multidimensional array is presented. Originally developed to help paging performance in applications where graphics data needed to be swapped in from disk, such structures are now crucial for memory locality on machines regardless of whether the array fits in main memory.

12.1 Triangle Meshes

Most real-world models are composed of complexes of triangles with shared vertices. These are usually known as *triangular meshes*, *triangle meshes*, or *triangular irregular networks* (TINs) and handling them efficiently is crucial to the performance of many graphics programs. The kind of efficiency that is important depends on the application. Meshes are stored on disk and in memory, and we'd like to minimize the amount of storage consumed. When meshes are transmitted across networks or from the CPU to the graphics system, they consume bandwidth, which is often even more precious than storage. In applications that perform operations on meshes, besides simply storing and drawing them—such as subdivision, mesh editing, mesh compression, or other operations—efficient access to adjacency information is crucial.

Triangle meshes are generally used to represent surfaces, so a mesh is not just a collection of unrelated triangles, but rather a network of triangles that connect to one another through shared vertices and edges to form a single continuous surface. This is a key insight about meshes: a mesh can be handled more efficiently than a collection of the same number of unrelated triangles.

The minimum information required for a triangle mesh is a set of triangles (triples of vertices) and the positions (in 3D space) of their vertices. But many, if not most, programs require the ability to store additional data at the vertices, edges, or faces to support texture mapping, shading, animation, and other operations. Vertex data is the most common: each vertex can have material parameters, texture coordinates, irradiances—any parameters whose values change across the surface. These parameters are then linearly interpolated across each triangle to define a continuous function over the whole surface of the mesh. However, it is also occasionally important to be able to store data per edge or per face.

12.1.1 Mesh Topology

The idea that meshes are surface-like can be formalized as constraints on the *mesh topology*—the way the triangles connect together, without regard for the vertex positions. Many algorithms will only work, or are much easier to implement, on a mesh with predictable connectivity. The simplest and most restrictive requirement on the topology of a mesh is for the surface to be a *manifold*. A manifold mesh is "watertight"—it has no gaps and separates the space on the inside of the surface from the space outside. It also looks like a surface everywhere on the mesh.

We'll leave the precise definitions to the mathematicians; see the chapter notes.

The term *manifold* comes from the mathematical field of topology: roughly speaking, a manifold (specifically a two-dimensional manifold, or 2-manifold) is

a surface in which a small neighborhood around any point could be smoothed out into a bit of flat surface. This idea is most clearly explained by counterexample: if an edge on a mesh has three triangles connected to it, the neighborhood of a point on the edge is different from the neighborhood of one of the points in the interior of one of the triangles, because it has an extra "fin" sticking out of it (Figure 12.1). If the edge has exactly two triangles attached to it, points on the edge have neighborhoods just like points in the interior, only with a crease down the middle. Similarly, if the triangles sharing a vertex are in a configuration like the left one in Figure 12.2, the neighborhood is like two pieces of surface glued together at the center, which can't be flattened without doubling it up. The vertex with the simpler neighborhood shown at right is just fine.

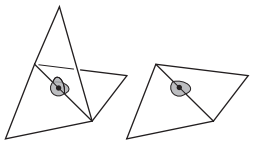


Figure 12.1. Non-manifold (left) and manifold (right) interior edges.

Many algorithms assume that meshes are manifold, and it's always a good idea to verify this property to prevent crashes or infinite loops if you are handed a malformed mesh as input. This verification boils down to checking that all edges are manifold and checking that all vertices are manifold by verifying the following conditions:

- Every edge is shared by exactly two triangles.
- Every vertex has a single, complete loop of triangles around it.

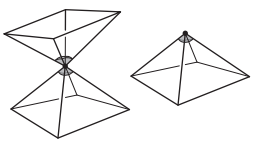


Figure 12.2. Non-manifold (left) and manifold (right) interior vertices.

Figure 12.1 illustrates how an edge can fail the first test by having too many triangles, and Figure 12.2 illustrates how a vertex can fail the second test by having two separate loops of triangles attached to it.

Manifold meshes are convenient, but sometimes it's necessary to allow meshes to have edges, or *boundaries*. Such meshes are not manifolds—a point on the boundary has a neighborhood that is cut off on one side. They are not necessarily watertight. However, we can relax the requirements of a manifold mesh to those for a *manifold with boundary* without causing problems for most mesh processing algorithms. The relaxed conditions are:

- Every edge is used by either one or two triangles.
- Every vertex connects to a single edge-connected set of triangles.

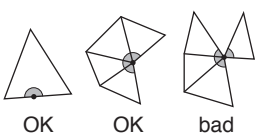


Figure 12.3. Conditions at the edge of a manifold with boundary.

Figure 12.3 illustrates these conditions: from left to right, there is an edge with one triangle, a vertex whose neighboring triangles are in a single edge-connected set, and a vertex with two disconnected sets of triangles attached to it.

Finally, in many applications it's important to be able to distinguish the "front" or "outside" of a surface from the "back" or "inside"—this is known as the *orientation* of the surface. For a single triangle we define orientation based on the order in which the vertices are listed: the front is the side from which the triangle's three vertices are arranged in counterclockwise order. A connected mesh is

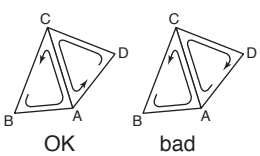


Figure 12.4. Triangles (B,A,C) and (D,C,A) are consistently oriented, whereas (B,A,C) and (A,C,D) are inconsistently oriented.

consistently oriented if its triangles all agree on which side is the front—and this is true if and only if every pair of adjacent triangles is consistently oriented.

In a consistently oriented pair of triangles, the two shared vertices appear in opposite orders in the two triangles' vertex lists (Figure 12.4). What's important is consistency of orientation—some systems define the front using clockwise rather than counterclockwise order.

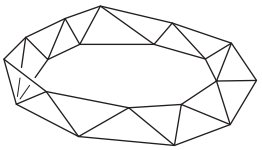


Figure 12.5. A triangulated Mobius band, which is not orientable.

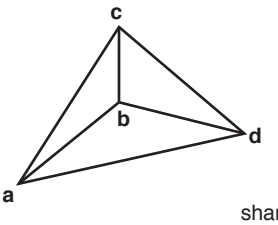
Any mesh that has non-manifold edges can't be oriented consistently. But it's also possible for a mesh to be a valid manifold with boundary (or even a manifold), and yet have no consistent way to orient the triangles—they are not *orientable* surfaces. An example is the Möbius band shown in Figure 12.5. This is rarely an issue in practice, however.

12.1.2 Indexed Mesh Storage

A simple triangular mesh is shown in Figure 12.6. You could store these three triangles as independent entities, each of this form:

```
Triangle {
  vector3 vertexPosition[3]
}
```

This would result in storing vertex **b** three times and the other vertices twice each for a total of nine stored points (three vertices for each of three triangles). Or you could instead arrange to share the common vertices and store only four, re-



separate triangles:

#	vertex 0	vertex 1	vertex 2
0	(a_x, a_y, a_z)	(b_x, b_y, b_z)	(c_x, c_y, c_z)
1	(b_x, b_y, b_z)	(d_x, d_y, d_z)	(c_x, c_y, c_z)
2	(a_x, a_y, a_z)	(d_x, d_y, d_z)	(b_x, b_y, b_z)

shared vertices:

triangles	
#	vertices
0	(0, 1, 2)
1	(1, 3, 2)
2	(0, 3, 1)

vertices	
#	position
0	(a_x, a_y, a_z)
1	(b_x, b_y, b_z)
2	(c_x, c_y, c_z)
3	(d_x, d_y, d_z)

Figure 12.6. A three-triangle mesh with four vertices, represented with separate triangles (left) and with shared vertices (right).

sulting in a *shared-vertex mesh*. Logically, this data structure has triangles which point to vertices which contain the vertex data:

```
Triangle {
  Vertex v[3]
}

Vertex {
  vector3 position   // or other vertex data
}
```

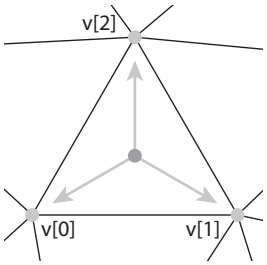


Figure 12.7. The triangle-to-vertex references in a shared-vertex mesh.

Note that the entries in the v array are references, or pointers, to Vertex objects; the vertices are not contained in the triangle.

In implementation, the vertices and triangles are normally stored in arrays, with the triangle-to-vertex references handled by storing array indices:

```
IndexedMesh {
  int tInd[nt][3]
  vector3 verts[nv]
}
```

The index of the kth vertex of the ith triangle is found in tInd[i][k], and the position of that vertex is stored in the corresponding row of the verts array; see Figure 12.8 for an example. This way of storing a shared-vertex mesh is an *indexed triangle mesh*.

Separate triangles or shared vertices will both work well. Is there a space advantage for sharing vertices? If our mesh has n_v vertices and n_t triangles, and if we assume that the data for floats, pointers, and ints all require the same storage (a dubious assumption), the space requirements are:

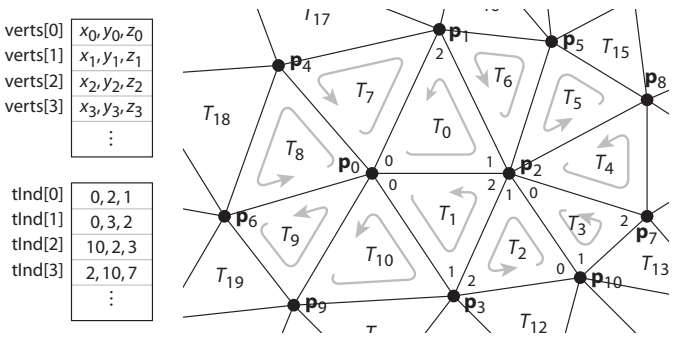


Figure 12.8. A larger triangle mesh, with part of its representation as an indexed triangle mesh.

- **Triangle.** Three vectors per triangle, for $9n_t$ units of storage;
- **IndexedMesh.** One vector per vertex and three ints per triangle, for $3n_v + 3n_t$ units of storage.

The relative storage requirements depend on the ratio of n_t to n_v.

As a rule of thumb, a large mesh has each vertex connected to about six triangles (although there can be any number for extreme cases). Since each triangle connects to three vertices, this means that there are generally twice as many triangles as vertices in a large mesh: $n_t \approx 2n_v$. Making this substitution, we can conclude that the storage requirements are $18n_v$ for the Triangle structure and $9n_v$ for IndexedMesh. Using shared vertices reduces storage requirements by about a factor of two; and this seems to hold in practice for most implementations.

Is this factor of two worth the complication? I think the answer is yes, and it becomes an even bigger win as soon as you start adding "properties" to the vertices.

12.1.3 Triangle Strips and Fans

Indexed meshes are the most common in-memory representation of triangle meshes, because they achieve a good balance of simplicity, convenience, and compactness. They are also commonly used to transfer meshes over networks and between the application and graphics pipeline. In applications where even more compactness is desirable, the triangle vertex indices (which take up two-thirds of the space in an indexed mesh with only positions at the vertices) can be expressed more efficiently using *triangle strips* and *triangle fans*.

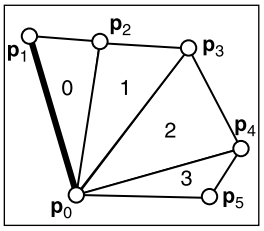


Figure 12.9. A triangle fan.

A triangle fan is shown in Figure 12.9. In an indexed mesh, the triangles array would contain [(0, 1, 2), (0, 2, 3), (0, 3, 4), (0, 4, 5)]. We are storing 12 vertex indices, although there are only six distinct vertices. In a triangle fan, all the triangles share one common vertex, and the other vertices generate a set of triangles like the vanes of a collapsible fan. The fan in the figure could be specified with the sequence $[0, 1, 2, 3, 4, 5]$: the first vertex establishes the center, and subsequently each pair of adjacent vertices (1-2, 2-3, etc.) creates a triangle.

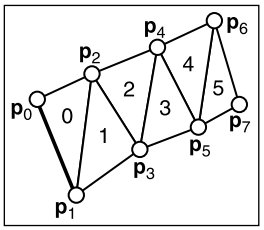


Figure 12.10. A triangle strip.

The triangle strip is a similar concept, but it is useful for a wider range of meshes. Here, vertices are added alternating top and bottom in a linear strip as shown in Figure 12.10. The triangle strip in the figure could be specified by the sequence [0 1 2 3 4 5 6 7], and every subsequence of three adjacent vertices (0-1-2, 1-2-3, etc.) creates a triangle. For consistent orientation, every other triangle needs to have its order reversed. In the example, this results in the triangles (0, 1, 2), (2, 1, 3), (2, 3, 4), (4, 3, 5), etc. For each new vertex that comes in, the oldest vertex is forgotten and the order of the two remaining vertices is swapped. See Figure 12.11 for a larger example.

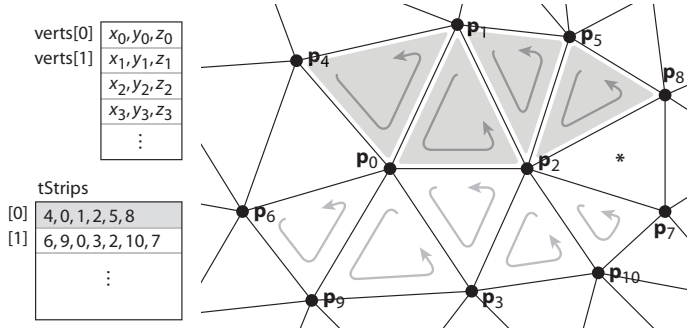


Figure 12.11. Two triangle strips in the context of a larger mesh. Note that neither strip can be extended to include the triangle marked with an asterisk.

In both strips and fans, $n + 2$ vertices suffice to describe n triangles—a substantial savings over the $3n$ vertices required by a standard indexed mesh. Long triangle strips will save approximately a factor of three if the program is vertex-bound.

It might seem that triangle strips are only useful if the strips are very long, but even relatively short strips already gain most of the benefits. The savings in storage space (for only the vertex indices) are as follows:

strip length	1	2	3	4	5	6	7	8	16	100	∞
relative size	1.00	0.67	0.56	0.50	0.47	0.44	0.43	0.42	0.38	0.34	0.33

So, in fact, there is a rather rapid diminishing return as the strips grow longer. Thus, even for an unstructured mesh, it is worthwhile to use some greedy algorithm to gather them into short strips.

12.1.4 Data Structures for Mesh Connectivity

Indexed meshes, strips, and fans are all good, compact representations for static meshes. However, they do not readily allow for meshes to be modified. In order to efficiently edit meshes, more complicated data structures are needed to efficiently answer queries such as:

- Given a triangle, what are the three adjacent triangles?
- Given an edge, which two triangles share it?

- Given a vertex, which faces share it?
- Given a vertex, which edges share it?

There are many data structures for triangle meshes, polygonal meshes, and polygonal meshes with holes (see the notes at the end of the chapter for references). In many applications the meshes are very large, so an efficient representation can be crucial.

The most straightforward, though bloated, implementation would be to have three types, *Vertex*, *Edge*, and *Triangle*, and to just store all the relationships directly:

```
Triangle {
  Vertex v[3]
  Edge e[3]
}

Edge {
  Vertex v[2]
  Triangle t[2]
}

Vertex {
  Triangle t[]
  Edge e[]
}
```

This lets us directly look up answers to the connectivity questions above, but because this information is all inter-related, it stores more than is really needed. Also, storing connectivity in vertices makes for variable-length data structures (since vertices can have arbitrary numbers of neighbors), which are generally less efficient to implement. Rather than committing to store all these relationships explicitly, it is best to define a class interface to answer these questions, behind which a more efficient data structure can hide. It turns out we can store only some of the connectivity and efficiently recover the other information when needed.

The fixed-size arrays in the Edge and Triangle classes suggest that it will be more efficient to store the connectivity information there. In fact, for polygon meshes, in which polygons have arbitrary numbers of edges and vertices, only edges have fixed-size connectivity information, which leads to many traditional mesh data structures being based on edges. But for triangle-only meshes, storing connectivity in the (less numerous) faces is appealing.

A good mesh data structure should be reasonably compact and allow efficient answers to all adjacency queries. Efficient means constant-time: the time to find

neighbors should not depend on the size of the mesh. We'll look at three data structures for meshes, one based on triangles and two based on edges.

The Triangle-Neighbor Structure

We can create a compact mesh data structure based on triangles by augmenting the basic shared-vertex mesh with pointers from the triangles to the three neighboring triangles, and a pointer from each vertex to one of the adjacent triangles (it doesn't matter which one); see Figure 12.12:

```
Triangle {
   Triangle nbr[3];
   Vertex v[3];
}

Vertex {
   // ... per-vertex data ...
   Triangle t; // any adjacent tri
}
```

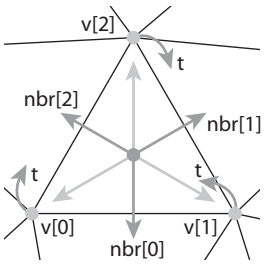


Figure 12.12. The references between triangles and vertices in the triangle neighbor structure.

In the array Triangle.nbr, the kth entry points to the neighboring triangle that shares vertices k and $k+1$. We call this structure the *triangle-neighbor structure*. Starting from standard indexed mesh arrays, it can be implemented with two additional arrays: one that stores the three neighbors of each triangle, and one that stores a single neighboring triangle for each vertex. (See Figure 12.13 for an example):

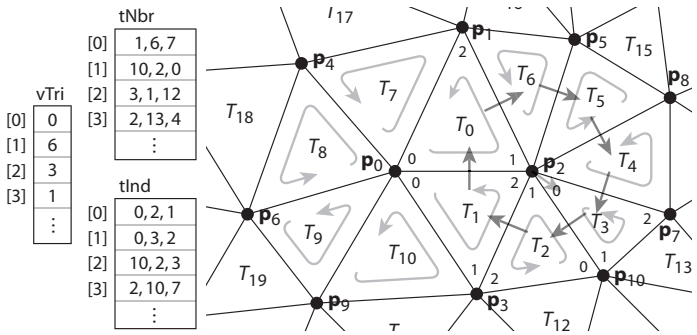


Figure 12.13. The triangle neighbor structure as encoded in arrays, and the sequence that is followed in traversing the neighboring triangles of vertex 2.

```
Mesh {
   // ... per-vertex data ...
   int tInd[nt][3]; // vertex indices
   int tNbr[nt][3]; // indices of neighbor triangles
   int vTri[nv]; // index of any adjacent triangle
}
```

Clearly the neighboring triangles and vertices of a triangle can be found directly in the data structure, but by using this triangle adjacency information carefully it is also possible to answer connectivity queries about vertices in constant time. The idea is to move from triangle to triangle, visiting only the triangles adjacent to the relevant vertex. If triangle t has vertex v as its kth vertex, then the triangle t.nbr[k] is the next triangle around v in the clockwise direction. This observation leads to the following algorithm to traverse all the triangles adjacent to a given vertex:

Of course, a real program would *do* something with the triangles as it found them.

```
TrianglesOfVertex(v) {
   t = v.t
   do {
      find i such that (t.v[i] == v)
      t = t.nbr[i]
   } while (t != v.t)
}
```

This operation finds each subsequent triangle in constant time—even though a search is required to find the position of the central vertex in each triangle's vertex list, the vertex lists have constant size so the search takes constant time. However, that search is awkward and requires extra branching.

A small refinement can avoid these searches. The problem is that once we follow a pointer from one triangle to the next, we don't know from which way we came: we have to search the triangle's vertices to find the vertex that connects back to the previous triangle. To solve this, instead of storing pointers to neighboring triangles, we can store pointers to specific edges of those triangles by storing an index with the pointer:

```
Triangle {
   Edge nbr[3];
   Vertex v[3];
}

Edge { // the i-th edge of triangle t
   Triangle t;
   int i;  // in {0,1,2}
}
```

```
Vertex {
   // ... per-vertex data ...
   Edge e;  // any edge leaving vertex
}
```

In practice the `Edge` is stored by borrowing two bits of storage from the triangle index t to store the edge index i, so that the total storage requirements remain the same.

In this structure the neighbor array for a triangle tells *which* of the neighboring triangles' edges are shared with the three edges of that triangle. With this extra information, we always know where to find the original triangle, which leads to an invariant of the data structure: for any jth edge of any triangle t,

$$t.\text{nbr}[j].\text{t.nbr}[t.\text{nbr}[j].\text{i}].\text{t} == t.$$

Knowing which edge we came in through lets us know immediately which edge to leave through in order to continue traversing around a vertex, leading to a streamlined algorithm:

```
TrianglesOfVertex(v) {
   {t, i} = v.e;
   do {
      {t, i} = t.nbr[i];
   } while (t != v.t);
}
```

The triangle-neighbor structure is quite compact. For a mesh with only vertex positions, we are storing four numbers (three coordinates and an edge) per vertex and six (three vertex indices and three edges) per face, for a total of $4n_v + 6n_t \approx 16n_v$ units of storage per vertex, compared with $9n_v$ for the basic indexed mesh.

The triangle neighbor structure as presented here works only for manifold meshes, because it depends on returning to the starting triangle to terminate the traversal of a vertex's neighbors, which will not happen at a boundary vertex that doesn't have a full cycle of triangles. However, it is not difficult to generalize it to manifolds with boundary, by introducing a suitable sentinel value (such as -1) for the neighbors of boundary triangles and taking care that the boundary vertices point to the most counterclockwise neighboring triangle, rather than to any arbitrary triangle.

The Winged-Edge Structure

One widely used mesh data structure that stores connectivity information at the edges instead of the faces is the *winged-edge* data structure. This data struc-

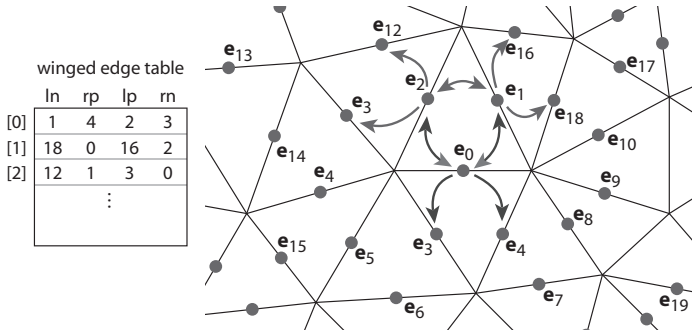
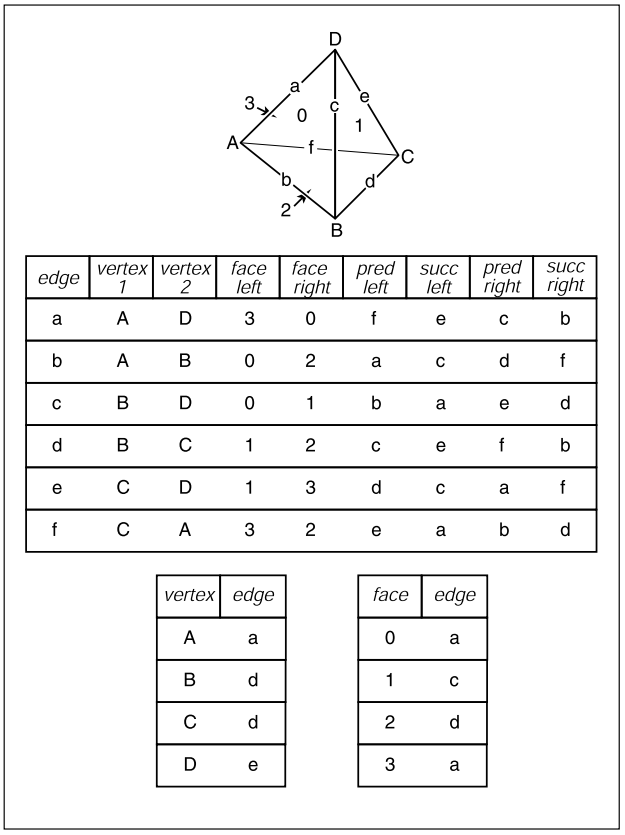


Figure 12.14. An example of a winged-edge mesh structure, stored in arrays.



edge	*vertex 1*	*vertex 2*	*face left*	*face right*	*pred left*	*succ left*	*pred right*	*succ right*
a	A	D	3	0	f	e	c	b
b	A	B	0	2	a	c	d	f
c	B	D	0	1	b	a	e	d
d	B	C	1	2	c	e	f	b
e	C	D	1	3	d	c	a	f
f	C	A	3	2	e	a	b	d

vertex	*edge*
A	a
B	d
C	d
D	e

face	*edge*
0	a
1	c
2	d
3	a

Figure 12.15. A tetrahedron and the associated elements for a winged-edge data structure. The two small tables are not unique; each vertex and face stores any one of the edges with which it is associated.

ture makes edges the first-class citizen of the data structure, as illustrated in Figures 12.14 and 12.15.

In a winged-edge mesh, each edge stores pointers to the two vertices it connects (the *head* and *tail* vertices), the two faces it is part of (the *left* and *right* faces), and, most importantly, the next and previous edges in the counterclockwise traversal of its left and right faces (Figure 12.16). Each vertex and face also stores a pointer to a single, arbitrary edge that connects to it:

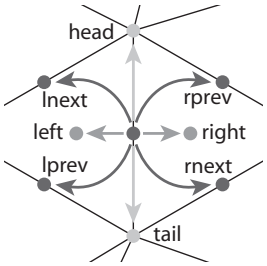


Figure 12.16. The references from an edge to the neighboring edges, faces, and vertices in the winged-edge structure.

```
Edge {
   Edge lprev, lnext, rprev, rnext;
   Vertex head, tail;
   Face left, right;
}

Face {
   // ... per-face data ...
   Edge e;  // any adjacent edge
}

Vertex {
   // ... per-vertex data ...
   Edge e;  // any incident edge
}
```

The winged-edge data structure supports constant-time access to the edges of a face or of a vertex, and from those edges the adjoining vertices or faces can be found:

```
EdgesOfVertex(v) {
   e = v.e;
   do {
      if (e.tail == v)
         e = e.lprev;
      else
         e = e.rprev;
   } while (e != v.e);
}

EdgesOfFace(f) {
   e = f.e;
   do {
      if (e.left == f)
         e = e.lnext;
      else
         e = e.rnext;
```

```
    } while (e != f.e);
}
```

These same algorithms and data structures will work equally well in a polygon mesh that isn't limited to triangles; this is one important advantage of edge-based structures.

As with any data structure, the winged-edge data structure makes a variety of time/space trade-offs. For example, we can eliminate the *prev* references. This makes it more difficult to traverse clockwise around faces or counterclockwise around vertices, but when we need to know the previous edge, we can always follow the successor edges in a circle until we get back to the original edge. This saves space, but it makes some operations slower. (See the chapter notes for more information on these tradeoffs).

The Half-Edge Structure

The winged-edge structure is quite elegant, but it has one remaining awkwardness—the need to constantly check which way the edge is oriented before moving to the next edge. This check is directly analogous to the search we saw in the basic version of the triangle neighbor structure: we are looking to find out whether we entered the present edge from the head or from the tail. The solution is also almost indistinguishable: rather than storing data for each edge, we store data for each *half-edge*. There is one half-edge for each of the two triangles that share an edge, and the two half-edges are oriented oppositely, each oriented consistently with its own triangle.

The data normally stored in an edge is split between the two half-edges. Each half-edge points to the face on its side of the edge and to the vertex at its head, and each contains the edge pointers for its face. It also points to its neighbor on the

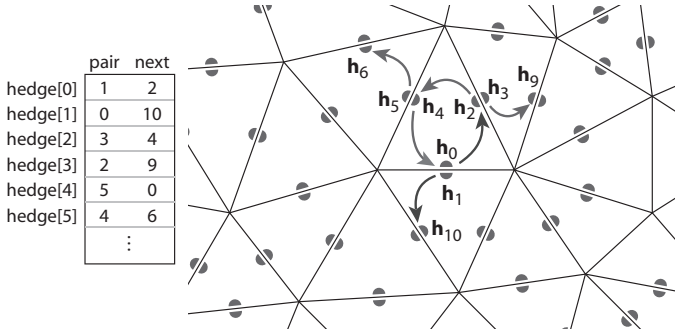


Figure 12.17. An example of a half-edge mesh structure, stored in arrays.

other side of the edge, from which the other half of the information can be found. Like the winged-edge, a half-edge can contain pointers to both the previous and next half-edges around its face, or only to the next half-edge. We'll show the example that uses a single pointer.

```
HEdge {
   HEdge pair, next;
   Vertex v;
   Face f;
}

Face {
   // ... per-face data ...
   HEdge h;  // any h-edge of this face
}

Vertex {
   // ... per-vertex data ...
   HEdge h;  // any h-edge pointing toward this vertex
}
```

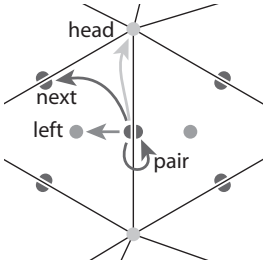


Figure 12.18. The references from a half-edge to its neighboring mesh components.

Traversing a half-edge structure is just like traversing a winged-edge structure except that we no longer need to check orientation, and we follow the *pair* pointer to access the edges in the opposite face.

```
EdgesOfVertex(v) {
   h = v.h;
   do {
      h = h.pair.next;
   } while (h != v.h);
}

EdgesOfFace(f) {
   h = f.h;
   do {
      h = h.next;
   } while (h != f.h);
}
```

The vertex traversal here is clockwise, which is necessary because of omitting the *prev* pointer from the structure.

Because half-edges are generally allocated in pairs (at least in a mesh with no boundaries), many implementations can do away with the *pair* pointers. For instance, in an implementation based on array indexing (such as shown in Figure 12.17), the array can be arranged so that an even-numbered edge i always pairs with edge $i + 1$ and an odd-numbered edge j always pairs with edge $j - 1$.

In addition to the simple traversal algorithms shown in this chapter, all three of these mesh topology structures can support "mesh surgery" operations of various sorts, such as splitting or collapsing vertices, swapping edges, adding or removing triangles, etc.

12.2 Scene Graphs

A triangle mesh manages a collection of triangles that constitute an object in a scene, but another universal problem in graphics applications is arranging the objects in the desired positions. As we saw in Chapter 6, this is done using transformations, but complex scenes can contain a great many transformations and organizing them well makes the scene much easier to manipulate. Most scenes admit to a hierarchical organization, and the transformations can be managed according to this hierarchy using a *scene graph*.

To motivate the scene-graph data structure, we will use the hinged pendulum shown in Figure 12.19. Consider how we would draw the top part of the pendulum:

$\mathbf{M}_1 = \text{rotate}(\theta)$
$\mathbf{M}_2 = \text{translate}(\mathbf{p})$
$\mathbf{M}_3 = \mathbf{M}_2\mathbf{M}_1$
Apply $\mathbf{M}_3$ to all points in upper pendulum

The bottom is more complicated, but we can take advantage of the fact that it is attached to the bottom of the upper pendulum at point $\mathbf{b}$ in the local coordinate system. First, we rotate the lower pendulum so that it is at an angle ϕ relative to

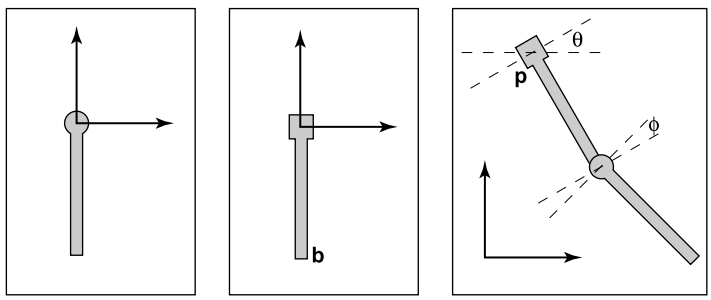


Figure 12.19. A hinged pendulum. On the left are the two pieces in their "local" coordinate systems. The hinge of the bottom piece is at point **b** and the attachment for the bottom piece is at its local origin. The degrees of freedom for the assembled object are the angles (θ,ϕ) and the location **p** of the top hinge.