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.

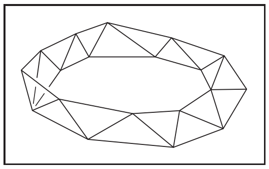


Figure 12.5. A triangulated Möbius band, which is not orientable.

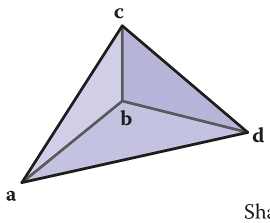
Any mesh that has non-manifold edges can't be oriented consistently. But it's also possible for a mesh to be a valid manifold with boundary (or even a manifold), and yet have no consistent way to orient the triangles—they are not *orientable* surfaces. An example is the Möbius band shown in Figure 12.5.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



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	
#	*Vertices*	#	*Position*
0	(0, 1, 2)	0	(a_x, a_y, a_z)
1	(1, 3, 2)	1	(b_x, b_y, b_z)
2	(0, 3, 1)	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).

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
}
```

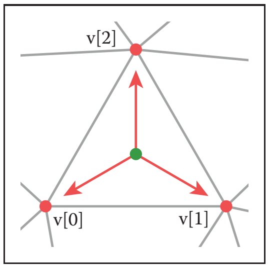


Figure 12.7. The triangle-to-vertex references in a shared-vertex mesh.

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 as follows:

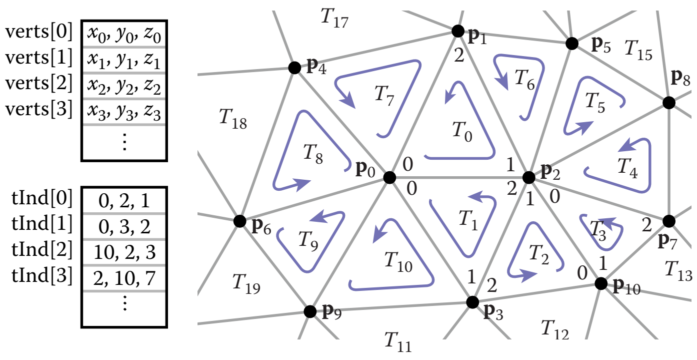

verts[0] x_0, y_0, z_0
verts[1] x_1, y_1, z_1
verts[2] x_2, y_2, z_2
verts[3] x_3, y_3, z_3
⋮
tInd[0] 0, 2, 1
tInd[1] 0, 3, 2
tInd[2] 10, 2, 3
tInd[3] 2, 10, 7
⋮
$\mathbf{p}_0$ $\mathbf{p}_1$ $\mathbf{p}_2$ $\mathbf{p}_3$ $\mathbf{p}_4$ $\mathbf{p}_5$ $\mathbf{p}_6$ $\mathbf{p}_7$ $\mathbf{p}_8$ $\mathbf{p}_9$ $\mathbf{p}_{10}$
T_0 T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 T_9 T_{10} T_{11} T_{12} T_{13} T_{15} T_{17} T_{18} T_{19}

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*.

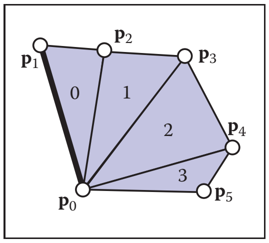


Figure 12.9. A triangle fan.

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.

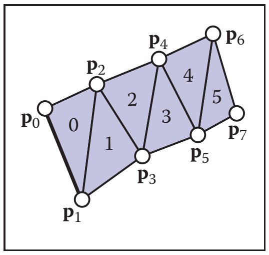


Figure 12.10. A triangle strip.

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.