Chapter 8

A Simple Way to Describe Shape in 2D and 3D

8.1 Introduction

We now turn to a discussion of the **triangle mesh,** the most widely used representation of shape in graphics. Triangle meshes consist of many triangles joined along their edges to form a surface (see Figure 8.1). Other meshes, in which the basic elements are quads (quadrilaterals), or other polygons are sometimes used, but there can be problems associated with them. For instance, it's easy to create a quadrilateral whose four vertices do not all lie in a plane; how should the interior be filled in? For triangles, this is not a problem: There's always a plane containing any three vertices. Because triangle meshes are so widespread, we concentrate on them in this chapter.

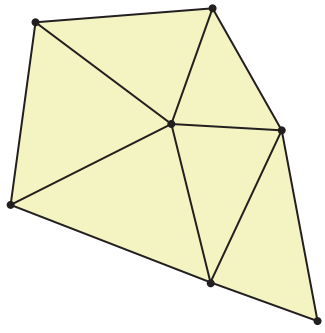

Figure 8.1: A triangle mesh, consisting of vertices, edges, and triangular faces.

It's easy to see how to create certain shapes with triangle meshes. Starting with any polyhedron, for instance, we can subdivide the faces into triangles. Figure 8.2 shows this for the cube. For more complex shapes, it's possible to *approximate* the shape with a mesh. One way to do this is to find the locations of many points on the shape, and then connect adjacent locations with a mesh structure. Such an approximation, if the points are close enough to one another, can look very much like a smooth surface. Consider the case of the icosahedron, which looks a lot like a sphere: Each point of the icosahedral mesh is very close to a point of the sphere, and vice versa. Similarly, each normal vector to a triangle mesh is very close to a vector normal to the sphere at a corresponding point, and vice versa. There is a distinction, however: The function that assigns normal vectors to points of the sphere is continuous, while for the icosahedron, it's **piecewise constant** (the normal vector doesn't change as you move about on a triangular facet). This distinction can be important when we try to consider the reflection of light from surfaces described with planar facets.

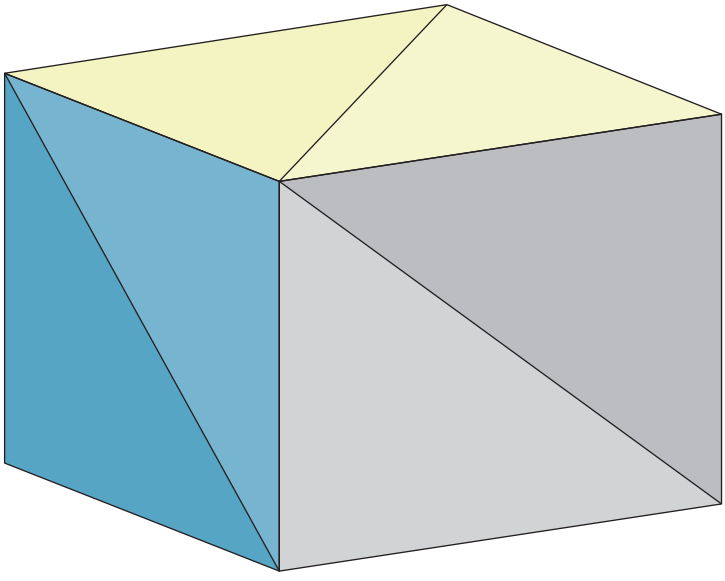

Figure 8.2: A triangle mesh that has the geometry of a cube.

One of the nicest properties of triangle meshes is their uniformity. This uniformity allows us to apply various operations to them with guarantees that have relatively simple proofs; it also makes it easy to try simple ideas. Among the most interesting operations we can perform on a mesh is **subdivision,** in which a single triangle is replaced by several smaller triangles in a fairly simple way. (There are many subdivision algorithms, some of which we'll discuss in Chapter 22.) Typically subdivision is used to smooth out a mesh that has sharp points or edges, so as to approach a limit surface that's fairly smooth. Of course, repeated subdivision operations increase the triangle count substantially; this can have a major impact on rendering performance.

Another important operation on meshes is **simplification,** in which a mesh is replaced by another mesh that's similar to it, topologically or geometrically, but has a more compact structure. If this is done repeatedly, one can arrive at a collection of simpler and simpler representations of the same surface, suitable for viewing at greater and greater distances. (A $10,000$-polygon object that covers only a single display pixel can almost certainly be rendered with *fewer* polygons, for instance—a simplified mesh is ideal here.) Hoppe [Hop96, Hop98] has studied this problem extensively.

Meshes are in common use in part because we are familiar with the geometry of triangles. Not every object in the world is well suited to mesh representation. Certain shapes, for instance, are characterized by having geometric detail at every scale (e.g., mica or fractured marble). Others have structure that is uniform in a way particularly unsuited for mesh representation, like hair, whose bent tubular structure can be far more compactly represented than with a mesh approximation.

Nonetheless, many research laboratories and commercial companies have managed to produce a great many successful images using an approach in which all shapes were approximated by triangle meshes.

8.2 “Meshes” in 2D: Polylines

The analog of a triangle mesh in space, taken one dimension lower, is a collection of line segments in the plane. (The space in which we work is one dimension lower, and the objects we’re working with are one dimension lower: line segments instead of triangles.) We’ll discuss these briefly as an introduction to mesh structures. We’ll call one of these a 1D mesh.

A 1D mesh (see Figure 8.3) consists of **vertices** and **edges,** which are line segments joining the vertices. Because the line segment between two vertices is completely determined by the vertices themselves, we can describe such a structure in two parts.

- A listing of the vertices and their locations. Typically the vertices are denoted by small integers; their locations are points in the plane.
- A listing of the edges, consisting of a collection of ordered pairs of vertices.

The following tables describe a simple 1D mesh:

Vertices	
1	(0, 0)
2	(0.5, 0)
3	(1.5, 1)
4	(0, 2.0)
5	(3, 0)
6	(4, 0)

Edges	
1	(1, 2)
2	(2, 3)
3	(3, 4)
4	(4, 1)
5	(5, 6)

This data structure has an interesting property: The **topology** of the mesh (which edges meet which other edges) is encoded in the Edges table, while the geometry is encoded in the Vertices table. If you adjusted one of the entries in the Vertices table a little, the number of connected components, for instance, would not vary.

One might argue that if you moved the vertices enough, then two edges might intersect when they didn’t intersect before. That’s true, but such intersections can be removed by adjusting the vertices; the fact that the edge $(1, 2)$ intersects the edge $(2, 3)$ cannot be altered by moving the vertices.

Indeed, one can treat the edge table (together with a listing of the vertex indices, in case some vertex is not in any edge) as describing an abstract graph (in the sense of graph theory). From this one can compute things like the Euler characteristic, the number of components, etc.

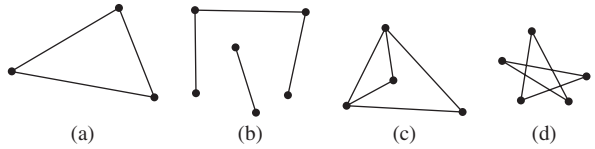

Figure 8.3: A 1D mesh consists of a collection of vertices and straight-line edges between them. The ones that most often interest us (shown in (a) and (b)) have one or two edges meeting each vertex, and no two edges intersect except at a vertex. But there are also meshes with more than two edges at some vertices, like (c), and ones where edges do intersect at nonvertex points (d).

8.2.1 Boundaries

The **boundary** of a 1D mesh defined as a *formal sum* of the vertices of the mesh in which the coefficient of each vertex is determined as follows: Each edge from vertex i to vertex j adds $+1$ to the coefficient for j, and -1 to the coefficient for i; we sometimes write that the boundary of the edge ij is $j - i$. Applying this to the mesh above, we find that the boundary formal sum is (reading edge by edge, and writing v_i for the ith vertex)

$$(v_2 - v_1) + (v_3 - v_2) + (v_4 - v_3) + (v_1 - v_4) + (v_6 - v_5), \tag{8.1}$$

which simplifies to $v_6 - v_5$. Informally, we say that the boundary consists of vertices 5 and 6.

The reason for the formalism arises when we consider more interesting meshes, like the one shown in Figure 8.4. The boundary in this case consists of $v_1 + v_2 + v_3 + v_4 + v_5 - 5v_0$.

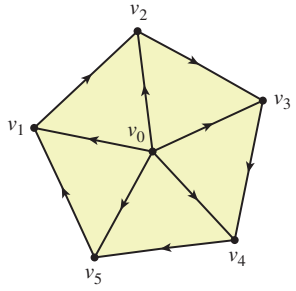

Figure 8.4: A wagon-wheel-shaped mesh. An arrow from vertex i to vertex j indicates that (i, j) is an edge of the mesh, and not (j, i).

A 1D mesh whose boundary is zero (i.e., the formal sum in which all coefficients are zero) has the property that it's easy to define "inside" and "outside" by a rule like the winding number rule for polygons in the plane. Such a mesh is called **closed.**

A 1D mesh where each vertex has degree 2 (i.e., where each vertex has an arriving edge and a leaving edge) is said to be a **manifold mesh:** In the abstract graph, every point has a neighborhood (a set of all points sufficiently near it) that resembles a small part of the real number line. A point in the interior of an edge, for example, has the edge interior as such a neighborhood. A vertex has the union of the interiors of the two adjacent edges, together with the vertex itself, as such a neighborhood.

We use the term "manifold mesh" to suggest that such meshes are like *manifolds,* which we will not formally define; there are many books that introduce the idea of manifolds with the appropriate supporting mathematics [dC76, GP10]. Informally, however, an n-dimensional manifold is an object M with the property that for any point $p \in M$, there's a neighborhood of p (i.e., a set of all points in M close to p, defined appropriately) that looks like the set $\{\mathbf{x} \in \mathbf{R}^n : \|\mathbf{x}\| < 1\}$ (the "open ball") in $\mathbf{R}^n$. "Looks like" means that there's a continuous map from the ball to the neighborhood and back. (These continuous maps are also required to be "consistent" with one another wherever their domains overlap; the precise details are beyond the scope of this book.) For example, the unit circle in the plane is a 1-manifold because one neighborhood of the point with angle coordinate θ consists of all points with coordinates $\theta - 0.1$ to $\theta + 0.1$; the correspondence to the unit ball in $\mathbf{R}$ (i.e., the open interval $-1 < x < 1$) is $u \mapsto 10(u - \theta)$. Similarly, familiar smooth surfaces in 3-space like the sphere, or the surface of a donut, are 2-manifolds. An atlas (i.e., a book showing maps of the whole world) is a kind of demonstration that the sphere is a manifold: Each page of the atlas gives a correspondence between some region of the globe (e.g., Western Europe) and a portion of the plane (i.e., the page of the atlas that shows Western Europe).

Shapes with corners (like a cube) can also be manifolds, but they are not *smooth* manifolds, which is what's usually meant when the term is used informally—a continuous map that takes a small region around the corner of the cube and sends it to the plane ends up distorting things too much for all the required conditions for "smoothness" to hold.

Self-intersecting shapes like a figure eight in the plane fail to be manifolds, because any small neighborhood of the crossing point in the middle of the figure eight looks like a small letter “x,” which cannot be made to correspond to a unit interval in a bicontinuous way. The technicalities in defining manifolds are quite subtle (indeed, it took several decades for them to eventually settle into their modern form). Fortunately for us, the shapes we use in graphics are generally “polyhedral manifolds.” The definitions are still subtle, but there are key theorems that tell us that in the one- and two-dimensional cases, we can instead verify that something’s a manifold by using far simpler methods. Those simpler methods are precisely the content of our notions of a manifold vertex-and-edge mesh, and a manifold triangle mesh (which we’ll define shortly).

Your goal, in reading the definitions here, should not be to gain a deep understanding that you could use to prove theorems—that requires a much more thorough treatment. But you should, when done, be able to say, “Sure, I can look at a simple mesh and identify it as a manifold mesh, or perhaps manifold-with-boundary mesh.”

Manifold meshes are commonplace and particularly easy to work with (and to prove theorems about). Note that the definition does not say that a manifold mesh must have only one connected component; indeed, a mesh consisting of two nonintersecting triangles is a valid manifold mesh. So is the empty mesh.

The meshes we’ve described, in which each edge is an *ordered* pair, are called **oriented meshes;** if we had described edges by unordered pairs, we would have had an **unoriented mesh,** in which case the definition of “boundary” would have made no sense. We’ll have no further use for such meshes, however.

8.2.2 A Data Structure for 1D Meshes

To prepare for the discussion of 2D meshes in 3-space, we’ll describe a data structure for 1D meshes in 2-space first, one which has strong analogies with more complex structures. The components are

- A vertex table, consisting of vertex indices and the associated points in the plane
- An edge table, consisting of ordered pairs of edges
- A **neighbor-list table,** consisting of an ordered cyclic list of edges meeting a vertex so that one can read off the list of edges at a vertex (in counterclockwise order)

We already encountered such a structure when we discussed subdivision of curves (see Figure 8.5) in Chapter 4.

The operations supported by this data structure (and their implementations) are as follows.

- Insert a vertex: Add it to the vertex table; leave other tables untouched.
- Insert an edge (i,j): Add it to the vertex table and the edge table (both $O(1)$); add it to the neighbor-list table both at vertex i and at vertex j. Insertion in the neighbor list for vertex i can take $O(e)$ time, where e is the number of edges in the table (one must insert it in the right place in the counterclockwise ordering of edges around vertex i, which might contain all e edges). In a *manifold* mesh, however, where there can be, at most, two edges at a vertex, this operation is $O(1)$.

- Get the edges meeting vertex i. (This is $O(1)$, since it consists of the neighbor list for vertex i.)
- Given a vertex i and an edge e containing i, find the other end of e.
- Given an edge e, find its two endpoints.
- Delete an edge. This requires deletion from the edge table, and if the edge is (i,j), deleting the edge from the neighbor lists for i and j, which may be an $O(e)$ operation, but in the manifold case is $O(1)$.

The choice of how to implement the vertex and edge tables depends on the expected use. An array implementation is convenient if there will be no deletions. But if there *will* be deletions, one must do one of the following.

- Mark array entries as invalid somehow.
- Shift the array contents to "fill in" when an item is deleted (which requires updating indices stored in other tables).

The marked-entries approach can create large but mostly empty tables if there are many insertions and deletions so that the "list all vertices" or "list all edges" operations become slow. The content-shifting approach actually works quite well, however. To start with a simple case, if there are n vertices and we want to delete vertex n, we just declare the end of the array to be the $n-1$st element, and delete all references to vertex n in other tables. To delete a different vertex—say, the second one—we reduce the problem to the earlier case: We exchange the second and the nth vertices, and then delete the nth. This requires replacing all references to vertex index 2 with vertex index n, and vice versa, in the other tables, but that's fairly straightforward.

Note that the choice to store the edges that meet a vertex is also application-dependent. It makes finding all vertices of topological distance one from vertex i very fast, but at the cost of making edge addition somewhat slow in the worst case. If you do not anticipate needing to find the neighbors of vertex i, maintaining a neighbor list is pointless. Similarly, the choice to store the neighbor lists in a counterclockwise-sorted order is useful primarily if one is interested not only in the structure of the 1D mesh, but also in the structure of the 2D regions into which it divides the plane; if these are of no interest, then the neighbor lists can be stored in hash tables or other similarly efficient structures (or in a two-element array, in the case of manifold meshes).

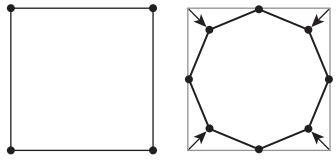

Figure 8.5: The square at left is subdivided to become the octagon at right. Note that for each vertex of the square, there's a vertex in the octagon, and for each edge *of the square, its midpoint is a vertex of the octagon as well.*

8.3 Meshes in 3D

The situation in 3D is analogous to that in 2D: To describe a mesh, we list the vertices and the triangles of the mesh. What about the edges? The tradition in graphics has been to infer the edges from the triangles: If vertices i, j, and k form a triangle, then the edges (i,j), (j,k), and (k,i) are assumed to be part of the mesh structure. This means that one cannot have dangling edges (see Figure 8.6), although isolated vertices are still allowed. The general descriptions of mesh structures (consisting of vertices, edges, triangles, tetrahedra, etc., with coordinates associated only to the vertices, and then extended to higher-dimensional pieces by interpolation) have been at the foundation of topology for more than 100 years [Spa66]; the student interested in such structures should consult the topology literature to avoid reinventing things.

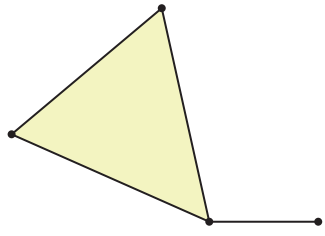

Figure 8.6: A triangle with a dangling edge, like the one shown here, cannot be represented by our mesh structure.

In graphics, however, the structure of a vertex table and "face table" or "triangle table" is well established. As in the 1D case, insertion of vertices and triangles is fast, and deletion of vertices is slow (because all associated triangles must be found and deleted). If we store a neighbor list for each vertex, deletion becomes faster. Because the neighbor list of triangles meeting a vertex is unordered, the insertion cost is low, but the deletion cost is high. When additional constraints are imposed on the mesh, the triangle list for a vertex can be ordered, slightly increasing insertion and deletion costs, but simplifying other operations like finding the two faces adjacent to an edge.

What about edges? While we cannot insert an edge, we can ask questions like "Is this pair of vertices an edge of the mesh?" In other words, is it the edge of some triangle of the mesh? And given a triangle with vertices i, j, and k, we can ask, "What are the other triangles that contain the edge (i,j)? These questions are $O(T)$, in the sense that answering them requires an exhaustive search of the triangle list. In special cases, which we discuss in the next section, they can be made faster.

If you are eager, at this point, to get on with making objects and pictures of objects, you can safely skip the remainder of this chapter and use the vertex- and triangle-table structure for meshes until you encounter problems with space or efficiency, at which point the remaining sections will be of use to you. If, on the other hand, you'd like to know more about how to work with meshes effectively, read on.

8.3.1 Manifold Meshes

A finite 2D mesh is a **manifold mesh** if the edges and triangles meeting a vertex v can be arranged in a cyclic order $t_1, e_1, t_2, e_2, \ldots, t_n, e_n$ without repetitions such that edge e_i is an edge of triangles t_i and t_{i+1} (indices taken mod n). This implies that for each edge, there are exactly two faces that contain it.

We can store a manifold mesh in a data structure analogous to the one we described for 1D meshes, consisting of a vertex table, a triangle table, and a neighbor-list table.

The neighbor list for vertex i consists of the triangles surrounding vertex i, in some cyclic order (so the kth and $k+1$st triangles in the list share an edge). (One can no longer disambiguate between the two possible cyclic orderings around a vertex with a notion like "counterclockwise," unfortunately, unless the manifold is *oriented,* which we describe in the next section.)

Manifold meshes unfortunately don't admit insertions or deletions of triangles: Any insertion or deletion ruins the manifold property. But it *is* easy to find the vertices adjacent to a given vertex (i.e., given a vertex index i, we can find all vertex indices j such that (i,j) is an edge): We simply take the set of all vertices of all triangles in the neighbor list for vertex i, and then remove vertex i.

It's also fairly easy, given a triangle containing edge (i,j), to find the other triangle containing that edge.

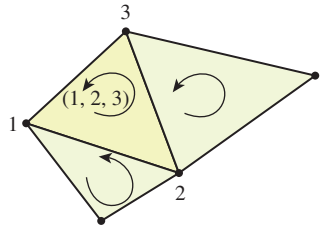

Figure 8.7: Two adjacent triangles in a mesh with consistent normal vectors (i.e., the normal vector tips are on the "same side" of the mesh). Note that the edge (i,j) is an edge of one triangle, but (j,i) is an edge of the other. In general, in a consistently oriented mesh, each edge appears twice, in opposite directions.

8.3.1.1 Orientation

We'll often have reason to care about the *orientation* of triangles in a mesh (see Figure 8.8) so that the triangles $(1,2,3)$ and $(2,1,3)$ are considered different (the triples are listings of vertex indices). One use of an orientation is in the determination of a *normal vector:* If the vertices of a nondegenerate triangle

(i.e., one with a nonzero area) are at locations P_i, P_j, and P_k, then we can compute $(P_j - P_i) \times (P_k - P_i)$, which is a vector perpendicular to the plane of the triangle.[1] Note that if we exchange vertices P_j and P_k, the resultant vector is negated. Since we often use the normal to a triangle in a mesh to determine what's "inside" or "outside" the mesh, the ordering of the vertices is critical.

If two adjacent triangles (see Figure 8.7) have consistently oriented normal vectors, then the edge they share will appear as (i,j) in one triangle and as (j,i) in the other. And if a manifold mesh can have its triangles oriented so that this is true, it can also be given consistently oriented normal vectors. This is a nontrivial theorem from combinatorial topology.

When a manifold is oriented, the triangles around a vertex have a natural order. Suppose that surrounding vertex 1 there are the triangles $(5,1,2)$, $(4,3,1)$, $(1,5,4)$, and $(1,3,2)$. We can cyclically permute each to put vertex 1 first: $(1,2,5)$, $(1,4,3)$, $(1,5,4)$, and $(1,3,2)$. Now starting with the first triangle, we can consider its "first" and "last" edges, $(1,2)$ and $(5,1)$. We choose, as our next triangle, the one whose first edge is the opposite, $(1,5)$, of this last edge. That's $(1,5,4)$. And the last edge of $(1,5,4)$ is $(4,1)$; $(1,4)$ is the first edge of $(1,4,3)$, whose last edge is $(3,1)$; $(3,1)$ is the first edge of $(1,3,2)$, and we've ordered the triangles: $(1,2,5)$, $(1,5,4)$, $(1,4,3)$, $(1,3,2)$.

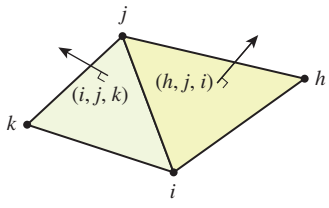

Figure 8.8: The triangles of this mesh are oriented*; the circular arrows indicate the cyclic ordering of the vertices. Note that the vertex triples* $(1,2,3)$ *and* $(2,3,1)$ *indicate the same oriented triangle (i.e., there are three equivalent descriptions of every oriented triangle).*

8.3.1.2 Boundaries

More interesting than manifold meshes (and more common as well) are meshes whose vertices are manifold or **boundarylike** (see Figure 8.9), in the sense that instead of the adjacent triangles forming a cycle, they form a chain, whose first and last elements share only one of their edges with other triangles in the chain; the *other* edge of the first triangle that meets the vertex is contained *only* in the first triangle, and not in any other triangle of the mesh. (The same condition applies to the last triangle.) This unshared edge is called a **boundary edge,** and the vertex is called a **boundary vertex.** Vertices that are not boundary vertices are called **interior vertices.**

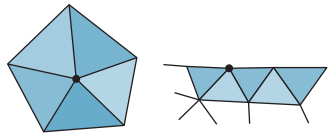

Figure 8.9: (left) A manifold vertex has a cycle of triangles around it; (right) by contrast, a boundarylike vertex v has a chain of triangles around it—the starting and ending triangles of the chain each share only one of their edges with other triangles of the chain. The other edges of those triangles that contain v are boundary edges.

8.3.1.3 Boundaries and Oriented 2D Meshes

Just as we defined the boundary of an edge from vertex i to vertex j to be the formal sum $v_j - v_i$, we can define the boundary of a triangle in a mesh with vertices i, j, and k to consist of the formal sum of edges

$$(i,j) + (j,k) + (k,i). \tag{8.2}$$

Furthermore, we can define an algebra on these formal sums in which the vertex (i,j) is identified with $-1(j,i)$ so that the boundary above could be written

$$(i,j) + (j,k) - (i,k) \tag{8.3}$$

instead. We can define the boundary of a collection of oriented triangles as the formal sum of their boundaries.

For an oriented manifold mesh, this boundary will be zero (i.e., the coefficient of each edge will be zero), because if (i,j) is part of the boundary of one face, then $(j,i) = -(i,j)$ is part of the boundary of another.

1. The description of the cross product and further discussion of normal vectors was given in Chapter 7.

For an oriented manifold-with-boundary mesh, the boundary will consist of exactly the edges we identified above as "boundary edges." In general, an oriented mesh with no boundary edges is called **closed.**

8.3.1.4 Operations on Manifold-with-Boundary Meshes

Manifold-with-boundary meshes support operations like vertex and face insertion and deletion. The efficiency of each operation depends on the implementation. If the representation is simply vertex tables and face tables, then insertion is $O(1)$, and deletion (after the "exchange with the last item in the list" trick) is too. If we maintain a neighbor list for each vertex, then insertion becomes $O(T)$, where T is the number of triangles, as does deletion.

Note that computing the boundary of such a mesh can be done in $O(T)$ time. (Use a hash table to count the number of times each edge appears, with sign. If the edge appears zero times, delete it from the hash table.) But one can also maintain a record of the boundary during insertions and deletions so that reporting it at any time is an $O(1)$ operation.

8.3.2 Nonmanifold Meshes

Just as in the 1D case, we sometimes encounter shapes that are not well represented by manifolds or manifolds with boundary. The two cubes shown in Figure 8.10 share a vertex which is a nonmanifold vertex; two cubes sharing an edge are similarly nonmanifold. There is an important difference between the two cases, however: It's easy to encounter a nonmanifold vertex in the course of constructing a manifold with boundary (see Figure 8.11). But once we construct a nonmanifold edge (one with three or more faces meeting it), it can never become a manifold edge through further additions.

Each vertex in the **directed-edge** structure also contains a reference to one of the edges containing it (see Figure 8.12). This allows one to compute the neighborhood of the vertex (all edges and triangles that meet it) in time proportional to the size of the neighborhood.

Campagna et al. [CKS98] show how this structure can be extended to handle non-manifold vertices and edges, and how to trade time for space by simplifying the structure for very large meshes.

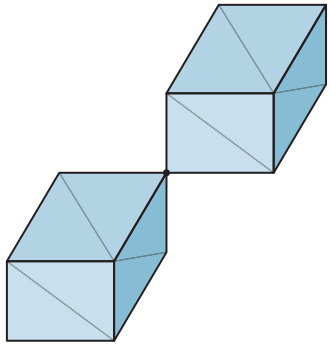

Figure 8.10: The shared vertex is nonmanifold: No neighborhood looks planar.

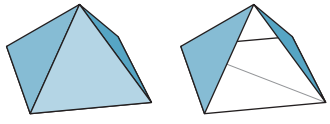

Figure 8.11: The pyramid at left has six faces; the bottom square is divided into two triangles that you cannot see. If we construct the pyramid so that after four triangles have been added, it appears as shown at the right, then the apex vertex is neither an interior vertex nor a boundary vertex according to the definitions, so this shape is neither a manifold nor a manifold with boundary. Once we add another face, it becomes a manifold with boundary, and when we add the last face, it becomes a manifold.

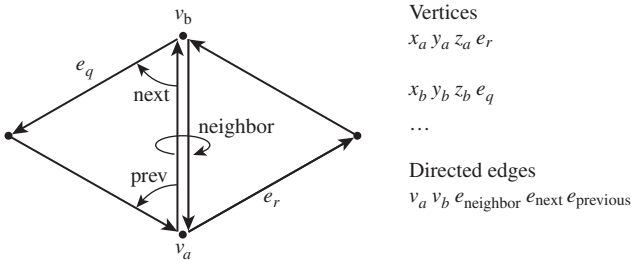

Figure 8.12: The directed-edge data structure (following Figures 4 and 5 of [CKS98]). A directed edge stores a reference to its starting and ending vertex, to the previous and next edges, and to its neighbor. For each real edge of the mesh, there are two directed edges, in opposite directions. Each vertex stores its coordinates and a reference to one of the directed edges that leaves it.

For more general planar meshes, in which the faces may not be triangles, one can use the **winged-edge** data structure [Bau72], which associates to each edge of the mesh the next edge of the face to its right, the next edge of the face to its left, and the previous edges of each of these as well (see Figure 8.13). This suffices to recover all the faces and edges, provided that all faces are simply connected (i.e., no face has a ringlike shape, like the surface of a moat). The winged-edge structure also stores, for each vertex, its *xyz*-coordinates, and a reference to one of the edges that it lies on (from which one can find all other edges). For each face, the structure stores a reference to one edge of the face (from which one can find all the others).

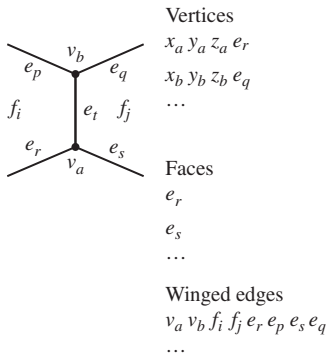

Figure 8.13: The winged-edge data structure records a "previous" and "next" pointer for the faces on each side of each edge.

8.3.3 Memory Requirements for Mesh Structures

Each of the structures we've described for representing meshes has certain memory requirements. Assuming 4-byte floating-point representations and 4-byte integers, we can compare their memory requirements as done by Campagna et al. [CKS98].

The vertex-table-and-triangle-table approach takes $12V$ bytes for the V vertices and $12T$ bytes for the T triangles. How are the number of vertices and triangles related? For a mesh representing a closed surface, Euler's formula tells us that $V - E + F = 2 - 2g$, where g is the **genus** of the surface.[2] Assuming further that every vertex is actually part of some triangle (so that the vertex table does not contain lots of unused vertices), and that the mesh is closed, we can simplify this: Each triangle has three edges, and each edge is shared by two triangles. So the number of edges, E, is $\frac{3}{2}T$. Thus,

$$V - \frac{3}{2}T + T = 2 - 2g, \tag{8.4}$$

which simplifies to

$$V - \frac{1}{2}T = 2 - 2g. \tag{8.5}$$

For low-genus surfaces with fine tessellations, the right-hand side is negligible compared to the left, so we find that the number of triangles is approximately twice the number of vertices; this gets us a total of $12(T + V) \approx 12(3V) = 36V \approx 18T$ bytes of storage. For the remaining mesh representations, we'll assume that we're representing closed manifolds of low genus so that we can replace V with $\frac{T}{2}$ and vice versa.

For the winged-edge structure, each vertex uses 16 bytes (three floats and an edge reference); each face uses four bytes (one edge reference), and each edge has four edge references, two face references, and two vertex references, for a total of eight references or 32 bytes. The total, again assuming we use *triangular faces only,* is $16V + 32E + 4T \approx 8T + 32\frac{3}{2}T + 4T = 60T$.

For the directed-edge data structure (in the store-all-references form), each vertex uses 12 bytes for coordinates and four for an edge reference. Each directed

2. The genus of a closed surface is, informally, the number of holes in it. A sphere has genus zero, a torus has genus one, a two-holed torus has genus two, etc. A slice of Swiss cheese tends to have quite high genus.

edge contains references to two vertices and three edges, using 20 bytes. Triangular faces are not explicitly represented. So the memory use is

$$16V + 40E \approx 8T + 60T = 68T \tag{8.6}$$

bytes. Note that in the analysis above, we assumed that a vertex or edge reference required only a single byte; for more complex meshes, this byte count may have to to be increased to roughly $\lceil \log_2(\frac{3T}{2}) \rceil$.

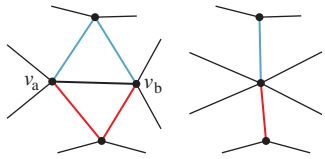

Figure 8.14: The edge from vertex a to vertex b is collapsed; the edge itself and the two adjacent faces are removed from the data structure; and the other two edges of the upper face, drawn in blue, become one, as do the two edges of the lower face, drawn in red. Nothing else changes. The two vertices $\mathbf{v}_a$ and $\mathbf{v}_b$ become a single vertex.

8.3.4 A Few Mesh Operations

One of the advantages of triangle meshes is that their homogeneity makes certain operations easy to perform. For manifold meshes, the homogeneity is even greater. In mesh simplification, for instance, one of the standard operations is an **edge collapse,** in which one edge is shrunk until it has length zero, resulting in the two adjacent triangles disappearing. In mesh **beautification** (where we try to make a mesh have nearly equilateral triangles and other nice properties), the **edge-swap** operation helps turn two long and skinny triangles into two more nearly equilateral ones. Both involve minimal operations on the data structure itself.

8.3.5 Edge Collapse

In the **edge-collapse** operation (see Figure 8.14), a single edge of a mesh is removed [HDD+93]. The two triangles that contain this edge are both eliminated, and the other two edges of each of them become a single edge in the new mesh. The vertices at the end of the eliminated segment become a single vertex.

The description above is purely topological; there's a geometric question as well: When we merge the two vertices, we must choose a *location* for the merged vertex. The location we choose depends on the goal of our simplification (see Figure 8.15). If computation is at a premium, simply using one of the old vertices as the new one is very fast. If we want to preserve some sort of shape, averaging the two vertices is easy. If such an averaging process moves a lot of points, and this will be visually distracting, we can choose a new location that minimizes the average or extreme distance between the old and new meshes. There is no one "right answer." As in most of graphics, the choice you make depends on your intended use of the data structure.

Figure 8.15: Different geometric choices for an edge collapse in 2D. The edge AB is collapsed; one can (a) place the collapsed vertex at A for computational simplicity, (b) at the midpoint of the segment AB, or (c) at a point Q that minimizes the maximum (or average) distance from every old mesh point to the nearest new mesh point. Other goals are possible as well.

8.3.6 Edge Swap

Meshes that get distorted or deformed in the course of an application's use of them may eventually get so deformed that individual triangles are long and skinny. Such triangles are characterized by their bad *aspect ratios*. In general, one can define an **aspect ratio** for a planar shape (see Figure 8.16) by finding, among all rectangles that enclose the object and touch it on all four sides (these are called **bounding boxes**), the one whose length-to-width ratio is greatest. This ratio is then called the aspect ratio. High-aspect-ratio triangles produce bad artifacts in many situations, so it's nice to be able to eliminate them when possible. An **edge-swap** operation (see Figure 8.17) can convert two adjacent high-aspect-ratio triangles to two with lower aspect ratios. (It can also, done in reverse, do the opposite: Selecting the right edge to swap in order to beautify a mesh requires examining the impact of each possible swap.)

Notice that the swap removes two triangles from the mesh structure and replaces them with two others. In the simple vertex- and triangle-table structure, this operation is trivial. In the directed-edge structure, the implementation is more complex, because (a) some vertex may point to the edge that's being swapped out, and this edge pointer needs to be found and replaced; and (b) it makes sense to replace the two old triangles with the two new ones, but there's substantial shuffling of directed edges in this process, and making sure that the new directed edges point to the correct new triangles is messy.

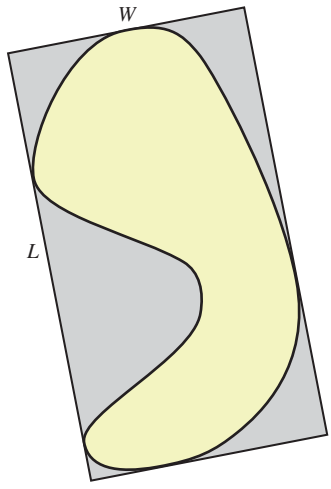

Figure 8.16: The aspect ratio of a planar shape is determined by finding the bounding box for the object for which the ratio of length to width is greatest.

8.4 Discussion and Further Reading

Triangle mesh representations and other representations for nontriangle meshes, for planar graphs, and for **simplicial complices**—assemblies of vertices, edges, triangles, tetrahedra, etc.—are widely studied in areas other than graphics; each representation is tuned to the application area. We've described a few representations here that are particularly suited to work in graphics, but those who develop CAD programs, for instance, may have to deal with computing the union and intersections of shapes represented by meshes. Unfortunately, the union of two manifold meshes (e.g., two cubes) may not be a manifold mesh (if the cubes share just one vertex, or just one edge), so structures suitable for nonmanifold representations may be essential. Those working in finite-element modeling of mechanical structures or fluid flow have their own constraints, such as the need for triangles and tetrahedra to be nicely shaped (no small angles in any triangles), or to have their size vary depending on the region in which they lie (e.g., turbulent flow may require a fine triangulation, while smooth flow may be adequately represented by a coarse one).

For most elementary graphics, the vertex- and triangle-table representations are adequate; their incredible simplicity makes them very versatile, and if you're implementing your own mesh structures, they're very easy to program properly. As your needs evolve, more complex structures may be suitable; be certain that you evaluate the more complex structures to ensure that their complexity solves your particular problem.

One structure we've completely ignored is the "list each triangle separately as a triple of *xyz*-coordinates structure." Although this is simple, it has so many disadvantages that we can never recommend its use. In particular, the only way to tell whether two triangles or edges share a vertex is with floating-point comparisons: If you move one copy of a vertex, you must move all others if you want to preserve the adjacency structure of the mesh; and determining any sort of adjacency information is, in general, $O(T)$.

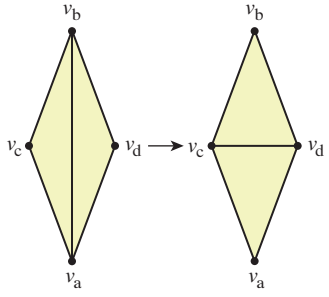

Figure 8.17: The triangles adjacent to the edge from $\mathbf{v}_a$ to $\mathbf{v}_b$ both have bad aspect ratios. By replacing the edge $\mathbf{v}_a\mathbf{v}_b$ with the edge $\mathbf{v}_c\mathbf{v}_d$, we get a new pair of triangles whose aspect ratios are better.

8.5 Exercises

Exercise 8.1: Suppose you know that triangle (i, j, k) is one of the triangles of a manifold mesh that's represented by a vertex table and a face table. Then edge (i, j) is in the mesh. Describe how to find the other triangle containing edge (i, j). Express the running time of this operation in terms of T, the number of triangles in the mesh. Draw an exemplar class of meshes that shows that the upper bound you found is actually realized in practice.

Exercise 8.2: Implement a form of the 1D mesh structure that's suitable for implementing a *subdivision* operation on *manifold* meshes. Given such a mesh, M, the associated subdivision mesh M' has one vertex for each vertex v of M: If the neighbors of v are u and w, the new vertex is at location $\frac{\alpha}{2}u + \frac{\alpha}{2}w + (1-\alpha)v$. M' also has one vertex for each *edge* of M: If the edge is between vertices t and u, the associated vertex of M' is at $\frac{1}{2}(t+u)$. Vertices in M' are connected by an edge if their associated vertices and/or edges in M meet (i.e., the vertex associated to the edge from u to v is connected to the vertex associated to u and to the vertex associated to v). Figure 8.5 shows an example. The parameter α determines the nature of the subdivision. The example shown in the figure uses $\alpha = 0.5$; on repeated subdivision, the square becomes a smoother and smoother curve. What happens for other values of α? Use the standard 2D test bed to make a program with which you can experiment. Note: Not every manifold mesh has just a single connected component.

Exercise 8.3: Draw examples to show how adding a triangle to a mesh can cause each of the four changes described in the nonmanifold vertex representation.

Exercise 8.4: Write pseudocode explaining how, given a vertex reference in a directed-edge data structure, to determine the list of all directed edges leaving that vertex in time proportional to the output size.

Exercise 8.5: Explain, in pseudocode, how, given a reference to a face in the winged-edge data structure, one can find all the edges of the face.

Exercise 8.6: Suppose that M is a connected manifold mesh with no boundary. M may be *orientable* without being *oriented:* It's possible that there's a consistent orientation of the faces of M, but that some faces are oriented inconsistently.
(a) Assuming that M is connected, describe an algorithm, based on depth-first search, for determining whether M is orientable.
(b) If M *is* orientable, explain why there are, at most, two possible orientations. Hint: Your algorithm may show why, once a single triangle orientation is chosen, all other triangle orientations are determined.
(c) If M is orientable, explain why there are exactly two orientations of M.
(d) Now suppose that M is not connected, but has $k \geq 2$ components. How many orientations of M are there?

Exercise 8.7: The 2D test bed was designed to aid in the study of things like meshes. Use it to build a program for drawing polylines in a 2D plane, getting mouse clicks, and reporting the closest vertex to a click (perhaps by changing the color of the vertex).

Exercise 8.8: Add a feature so that a shift-click on a vertex initializes edge drawing: The starting vertex is highlighted, and the next vertex clicked is connected to the starting vertex with an edge; if there's already an edge between the two, it should be deleted. And if the next click is *not* on a vertex, a vertex is created there and an edge from the preselected vertex to the new one is added. Modify the program to handle 2D meshes (i.e., vertices and *triangles*) by allowing the user to control-click on three vertices to create a triangle (or delete it if it already exists).

This page intentionally left blank

Chapter 9

Functions on Meshes

9.1 Introduction

In mathematics, functions are often described by an algebraic expression, like $f(x) = x^2 + 1$. Sometimes, on the other hand, they're **tabulated,** that is, the values for each possible argument are listed, as in

$$f : \{1, 2, 3\} \to \{0, 9\}; \tag{9.1}$$

$$f(1) = f(2) = 0; f(3) = 9. \tag{9.2}$$

A third, and very common, way to describe a function is to give its values at particular points and tell how to *interpolate* between these known values. For instance, we might plot the temperature at noon and midnight of each day of a week; such a plot consists of 15 distinct dots (see Figure 9.1). But we could also make a guess about the temperatures at times between each of these, saying, for instance, that if it was 60° at noon and 24° at midnight, that drop of 36° took place at a steady rate of 3° per hour. In other words, we would be **linearly interpolating** to define the function for *all* times rather than just at noon and midnight each day. The resultant function, defined on the whole week rather than just the 15 special times, is a connect-the-dots version of the original.

Let's now write that out in equations. Suppose that $t_0 < t_1 < t_2 < \ldots < t_n$ are the times at which the temperature is known, and that $f_0, f_1, \ldots, f_n$ are the temperatures in degrees Fahrenheit at those times.

Then

$$f : [t_0, t_n] \to \mathbf{R} : t \mapsto (1 - s)f_i + sf_{i+1} \tag{9.3}$$

where

$$t_i \leq t \leq t_{i+1} \quad \text{and} \tag{9.4}$$

$$s = \frac{t - t_i}{t_{i+1} - t_i}. \tag{9.5}$$

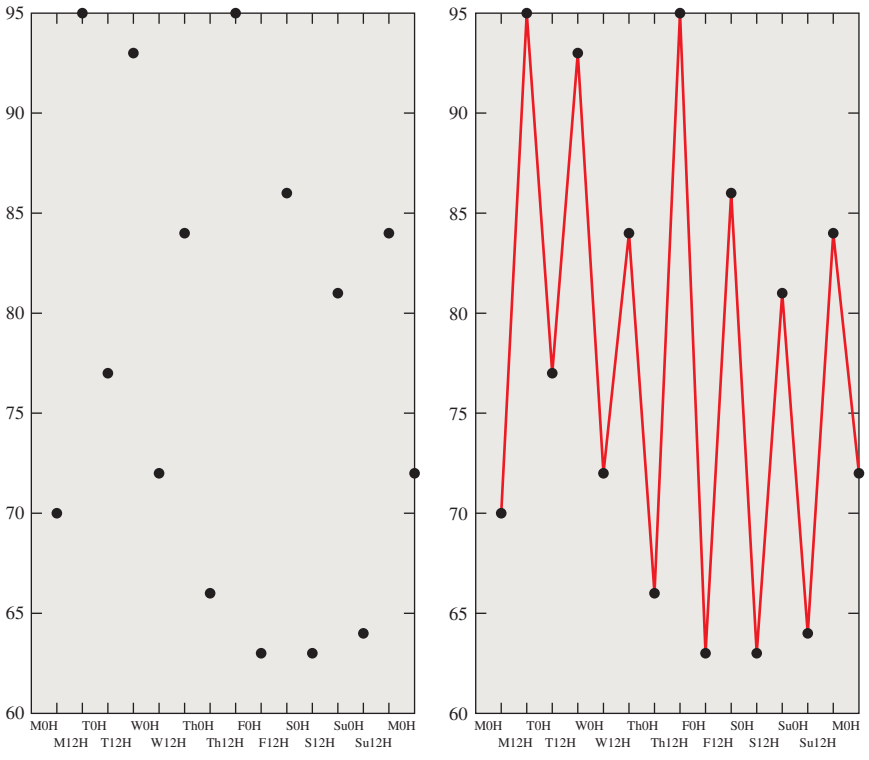

Figure 9.1: The temperature at noon and midnight for each day in a week. The domain of this function consists of 15 points. When we linearly interpolate the values between pairs of adjacent points, we get a function defined on the whole week—a connect-the-dots version of the original.

Further suppose that the times are measured in 12-hour units, starting at midnight Sunday, so $t_0 = 0, t_1 = 1$, etc.

In this formulation i is the index of the interval containing t, and s describes the fraction of the way from t_i to t_{i+1} where t lies (when $t = t_i$, s is zero; when $t = t_{i+1}$, s is one).

Inline Exercise 9.1: Suppose that $t_0 = 0, t_1 = 1$, etc., and that $f_0 = 7, f_1 = 3$, and $f_2 = 4$; evaluate $f(1.2)$ by hand. If we changed f_0 to 9, would it change the value you computed? Why or why not?

Just as we can have barycentric coordinates on a triangle (see Section 7.9.1), we can place them on an interval $[p, q]$ as well. The first coordinate varies from one at p to zero at q; the second varies from zero at p to one at q. Their sum is **everywhere one** that is, for every point of the interval $[p, q]$, the sum is one (see Exercise 9.8). With these barycentric coordinates, we can write a slightly more symmetric version of the formula above:

$$f(t) = c_0(t) f_i + c_1(t) f_{i+1}, \tag{9.6}$$

where $c_0(t)$ is the first barycentric coordinate of t and $c_1(t)$ is the other.

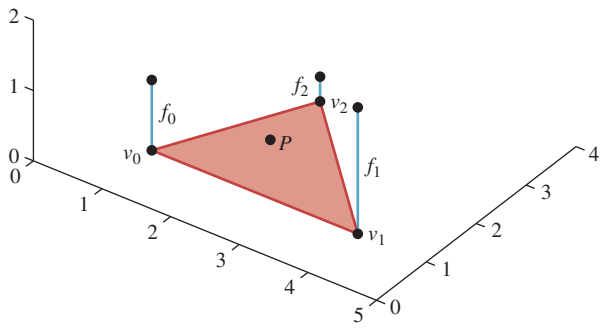

Figure 9.2: The point P is in the triangle with vertices v_0, v_1, and v_2, where the function has values f_0, f_1, and f_2, respectively. What value should we assign to the point P?

This "continuous extension" of f has analogs for other situations. Before we look at those, let's examine some of the properties. First, the value of the interpolated function at noon or midnight each day (t_i) is just f_i; it doesn't depend at all on the values at noon or midnight on the other days. Second, the value at noon on one day influences the shape of the graph only for the 12 hours before and the 12 hours after. Third, the interpolated function is in fact continuous.

Now let's consider interpolating values given at discrete points on a surface. Since we often use triangle meshes in graphics to represent surfaces, let's suppose that we have a function whose values are known at the vertices of the mesh; the value at vertex i is f_i. How can we "fill in" values at the other points of all the triangles in the mesh?

By analogy, we use barycentric coordinates. Consider a point P in a triangle with vertices v_0, v_1, and v_2 (see Figure 9.2); we can define

$$f(P) = c_0 f_0 + c_1 f_1 + c_2 f_2 \tag{9.7}$$

where (c_0, c_1, c_2) are the barycentric coordinates of P with respect to v_0, v_1, and v_2.

Once again, the interpolated function has several nice properties. First, the value at the vertex v_i is just f_i; the value along the edge from v_i to v_j (assuming they are adjacent) depends only on f_i and f_j; thus, for a point q on such an edge, it doesn't matter which of the two triangles sharing the edge from v_i to v_j is used to compute $f(q)$—the answer will be the same! Second, the value f_i at vertex v_i again influences other values only **locally,** that is, only on triangles that contain the vertex v_i. Third, the interpolated function is in fact continuous.

The remainder of this chapter investigates this idea of interpolating across faces, its relationship to barycentric coordinates, and some applications.

9.2 Code for Barycentric Interpolation

The discussion so far has been somewhat abstract; we'll now write some code to implement these ideas. Let's start with a simple task.

Input:

- A triangle mesh in the form of an $n \times 3$ table, `vtable`, of vertices
- A $k \times 3$ table, `ftable`, of triangular faces, where each row of `ftable` contains three indices into `vtable`

- An $n \times 1$ table, `fntable`, of function values at the vertices of the mesh
- A point P of the mesh, expressed by giving the index, t, of the triangle in which the point lies, and its barycentric coordinates α, β, γ in that triangle

Output:

- The value of the interpolated function at P

The first thing to realize is that only the tth row of `ftable` is relevant to our problem: The point P lies in the tth triangle; the other triangles might as well not exist. If the tth triangle has vertex indices $i0$, $i1$, and $i2$, then only those entries in `vtable` matter. With this in mind, our code is quite simple:

```
1  double meshinterp(double[,] vtable, int[,] ftable,
2    double[] fntable, int t, double alpha, double beta, double gamma)
3  {
4    int i0 = ftable[t, 0];
5    int i1 = ftable[t, 1];
6    int i2 = ftable[t, 2];
7    double fn0 = fntable[i0];
8    double fn1 = fntable[i1];
9    double fn2 = fntable[i2];
10   return alpha*fn0 + beta*fn1 + gamma*fn2;
11 }
```

Now suppose that P is given differently: We are given the coordinates of P in 3-space rather than the barycentric coordinates, and we are given the index t of the triangle to which P belongs, and we need to find the barycentric coordinates α, β, and γ. If we say that the vertices of the triangle t are A, B, and C, we want to have

$$\alpha A_x + \beta B_x + \gamma C_x = P_x \tag{9.8}$$

where the subscript x indicates the first coordinate of a point; we must also satisfy the same equations for y and z. But there's one more equation that has to hold: $\alpha + \beta + \gamma = 1$. We can rewrite that in a form that's analogous to the others:

$$\alpha 1 + \beta 1 + \gamma 1 = 1. \tag{9.9}$$

Now our system of equations becomes

$$\begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}. \tag{9.10}$$

There's really no solution here except to directly solve the system of equations. The problem is that we have four equations in three unknowns, and most solvers want to work with *square* matrices rather than rectangular ones. (Section 7.9.2 presented an alternative approach to this problem using some precomputation, but that precomputation amounts to doing much of the work of solving the system of equations.)

The good news is that the four equations are in fact redundant: The fact that P is *given* to us as a point of the triangle ensures that if we simply solve the first three equations, the fourth will hold. That assurance, however, is purely mathematical—computationally, it may happen that small errors creep in. There are several viable approaches.

- Express P as a convex combination of *four* points of $\mathbf{R}^4$: the three already written and a fourth, with coordinates n_x, n_y, n_z, and 0, where $\mathbf{n}$ is the normal vector to the triangle. The expression will have a fourth coefficient, δ, in the solution, representing the degree to which P is *not* in the plane of A, B, and C. We ignore this and scale up the computed α, β, and γ accordingly, using $\alpha/(1-\delta), \beta/(1-\delta)$, and $\gamma/(1-\delta)$ as the barycentric coordinates. This is a good solution (in the sense that if the numerical error in P is entirely in the $\mathbf{n}$ direction, the method produces the correct result), but it requires solving a 4×4 system of equations.
- Delete the fourth row of the system in Equation 9.10; adjust α, β, and γ to sum to one by dividing each by $\alpha + \beta + \gamma$. This reduces the problem to a 3×3 system, but it lacks the promise of correctness for normal-only errors.
- Use the pseudoinverse to solve the overconstrained system (see Section 10.3.9). This has the advantage that it's already part of many numerical linear algebra systems, and that it works even when the triangle is degenerate (i.e., the three points are collinear), in the sense that *if* P lies in the triangle, then the method produces numbers α, β, and γ with $\alpha A + \beta B + \gamma C = P$, even though the solution in this case is not unique. Better still, if P is not in the plane of A, B, and C, the solution returned will have the property that $\alpha A + \beta B + \gamma C$ is the point in the plane of A, B, and C that's closest to P. This is therefore the ideal.

So the restated problem looks like this.
Input:

- A triangle mesh in the form of an $n \times 3$ table, `vtable`, of vertices
- A $k \times 3$ table, `ftable`, of triangles, where each row of `ftable` contains three indices into `vtable`
- A point P of the mesh, expressed by giving the index, t, of the triangle in which the point lies, and its coordinates in 3-space

Output:

- The value of the barycentric coordinates of P with respect to the vertices of the kth triangle

And our revised solution is this:

```
1  double[3] barycentricCoordinates(double[,] vtable,
2     int[,] ftable, int t, double p[3])
3  {
4     int i0 = ftable[t, 0];
5     int i1 = ftable[t, 1];
6     int i2 = ftable[t, 2];
7     double[,] m = new double[4, 3];
8     for (int j = 0; j < 3; j++) {
9        for (int i = 0; i < 3; i++) {
10         m[i,j] = vtable[ftable[t, j], i];
11       }
12       m[3,j] = 1;
13    }
14
15    k = pseudoInverse(m);
16    return matrixVectorProduct(k, p);
17 }
```

Here we've assumed the existence of a matrix-vector product procedure and a pseudoinverse procedure, as provided by most numerical packages.

The two procedures above can be combined, of course, to produce the function value at a point P that's specified in xyz-coordinates.

Input:

- A triangle mesh in the form of an $n \times 3$ table, `vtable`, of vertices
- A $k \times 3$ table, `ftable`, of triangles, where each row of `ftable` contains three indices into `vtable`
- An $n \times 1$ table, `fntable`, of function values at the vertices of the mesh
- A point P of the mesh, expressed by giving the index, t, of the triangle in which the point lies, and its coordinates in 3-space

Output:

- The value of the function defined by the function table at the point P

```
1 double meshinterp2(double[,] vtable, int[,] ftable, double[] fntable,
2 int t, double p[3])
3 {
4   double[] barycentricCoords =
5      barycentricCoordinates(vtable, ftable, t, p);
6   return meshinterp2(vtable, ftable, fntable, t,
7      barycentricCoords[0], barycentricCoords[1], barycentricCoords[2]);
8 }
```

Of course, the same idea can be applied to the ray-intersect-triangle code of Section 7.9.2, where we first computed the barycentric coordinates (α, β, γ) of the ray-triangle intersection point Q with respect to the triangle ABC, and then computed the point Q itself as the barycentric weighted average of A, B, and C. If instead we had function values f_A,f_B, and f_C at those points, we could have computed the value at Q as $f_Q = \alpha f_A + \beta f_B + \gamma f_C$. Notice that this means we can compute the interpolated function value f_Q at the intersection point without ever computing the intersection point itself!

Computing barycentric coordinates (α, β, γ) for a point P of a triangle ABC, where $A, B, C \in \mathbf{R^2}$, is somewhat simpler than the corresponding problem in 3-space (see Figure 9.3). We know that the lines of constant α are parallel to BC. If we let $\mathbf{n} = (C - B)^{\perp}$, then the function defined by $f(P) = (P - B) \cdot \mathbf{n}$ is *also* constant on lines parallel to $B - C$. Scaling this down by $f(A)$ gives us the function we need: It's zero on line BC, and it's one at A. So we let

$$g : \mathbf{R^2} \to \mathbf{R} : P \mapsto \frac{(P - B) \cdot \mathbf{n}}{(A - B) \cdot \mathbf{n}}, \tag{9.11}$$

and the value of $g(P)$ is just α. A similar computation works for β and γ.

The resultant code looks like this:

```
1 double[3] barycenter2D(Point P, Point A, Point B, Point C)
2 C[2])
3 {
4   double[] result = new double[3];
5   result[0] = helper(P, A, B, C);
6   result[1] = helper(P, B, C, A);
7   result[2] = helper(P, C, A, B);
8   return result;
```

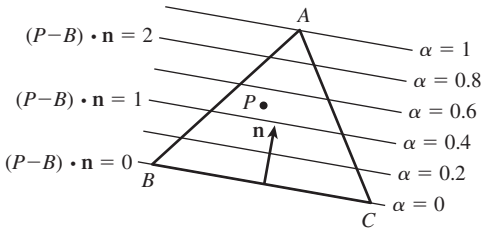

Figure 9.3: To write the point P as $\alpha A + \beta B + \gamma C$, we can use a trick. For points on line BC, we know $\alpha = 0$; on any line parallel to BC, α is also constant. We can compute the projection of $P - B$ onto the vector **n** *that's perpendicular to BC; this also gives a linear function that's constant on lines parallel to BC. If we scale this function so that its value at A is* 1*, we must have the function α.*

```
9  double helper(Point P, Point A, Point B, Point C)
10 {
11    Vector n = C - B;
12    double t = n.X;
13    n.X = -n.Y; // rotate C-B counterclockwise 90 degrees
14    n.Y = t;
15    return dot(P - B, n) / dot(A - B, n);
16 }
```

Of course, if the triangle is degenerate (e.g., if A lies on line BC), then the dot product in the denominator of the helper procedure will be zero; then again, in this situation the barycentric coordinates are not well defined. In production code, one needs to check for such cases; it would be typical, in such a case, to express P as a convex combination of two of the three vertices.

9.2.1 A Different View of Linear Interpolation

One way to understand the interpolated function is to realize that the interpolation process is *linear*. Suppose we have two sets of values, $\{f_i\}$ and $\{g_i\}$, associated to the vertices, and we interpolate them with functions F and G on the whole mesh. If we now try to interpolate the values $\{f_i + g_i\}$, the resultant function will equal $F + G$. That is to say, we can regard barycentric interpolation on the mesh as a function from "sets of vertex values" to "continuous functions on the mesh." Supposing there are n vertices, this gives a function

$$I : \mathbf{R}^n \to C(M) \tag{9.12}$$

where $C(M)$ is the set of all continuous functions on the mesh M. What we've just said is that

$$I(f + g) = I(f) + I(g) \tag{9.13}$$

where f denotes the set of values $\{f_1, f_2, \ldots, f_n\}$, and similarly for g; the other linearity rule—that $I(\alpha f) = \alpha I(f)$ for any real number α—also holds.

A good way to understand a linear function is to examine what it does to a *basis*. The standard basis for $\mathbf{R}^n$ consists of elements that are all zero except for a single entry that's one. Each such basis vector corresponds to interpolating a function that's zero at all vertices except one—say, v—and is one at v (see Figure 9.4).

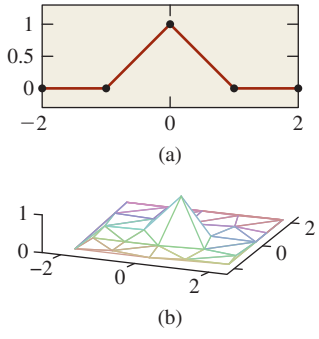

Figure 9.4: (a) The 2D interpolating basis function is tent-shaped near its center; (b) in 3D, for a mesh in the xy-plane, we can graph the function in z and again see a tentlike graph that drops off to zero by the time we reach any triangle that does not contain v.

The resultant interpolant is a **basis function** that has a graph that looks tentlike, with the peak of the tent sitting above the vertex where the value is one.

If we add up all these basis functions, the result is the function that interpolates the values $1, 1, \ldots, 1$ at all the vertices; this turns out to be the constant function 1. Why? Because *the barycentric coordinates of every point in every triangle sum to one.*

One might look at these tent-shaped functions and complain that they're not very continuous in the sense that they are continuous but not differentiable. Wouldn't it be nicer to use basis functions that looked like the ones in Figure 9.5? It would, but it turns out to be more difficult to have *both* the smoothness property *and* the property that the interpolant for the "all ones" set of values is the constant function 1. We'll discuss this further in Chapter 22.

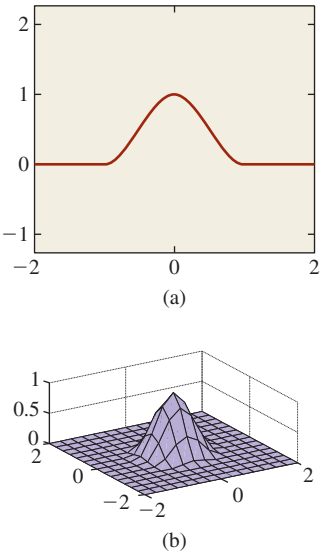

Figure 9.5: Basis functions for (a) 2D and (b) 3D interpolations that are smoother than the barycentric-interpolation functions.

9.2.1.1 Terminology for Meshes

This section introduces a few terms that are useful in discussing meshes. First, the vertices, edges, and faces of a mesh are all called **simplices**. Simplices come in categories: A vertex is a 0-simplex, an edge is a 1-simplex, and a face is a 2-simplex. Simplices contain their boundaries, so a 2-simplex in a mesh contains its three edges and a 1-simplex contains its two endpoints.

The **star** of a vertex (see Figure 9.6) is the set of triangles that contain that vertex. More generally, the star of a simplex is the set of all simplices that contain it.

The boundary of the star of a vertex is called the **link** of the vertex. This is useful in describing functions like the tent functions above: We can say that the tent has value 1 at the vertex v, has nonzero values only on the star of v, and is zero on the link of v.

There's a notion of "distance" in a mesh based on edge paths between vertices: The distance from v to w is the smallest number of edges in any chain of edges from v to w. Thus, all the vertices in the link of v have a mesh distance of one from v.

The sets of vertices at various distances from v have names as well. The 1-ring is the set of vertices whose distance from v is one or less; the 2-ring is the set of vertices whose distance from v is two or less, etc.

9.2.2 Scanline Interpolation

Frequently in graphics we need to compute some value at each point of a triangle; for example, we often compute an RGB color triple at each vertex of a triangle, and then interpolate the results over the interior (perhaps because doing the

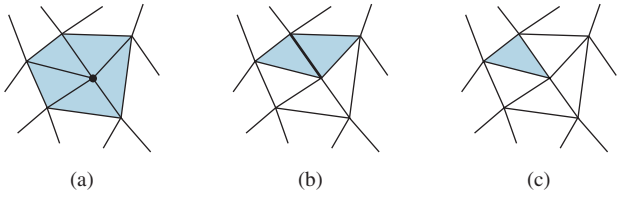

Figure 9.6: The star of a simplex. (a) The star of a vertex is the set of triangles containing it, (b) the star of an edge is the two triangles containing it, and (c) the star of a triangle is the triangle itself.

computation that produced the RGB triple at the vertices would be prohibitive if carried out for every interior point).

In the 1980s, when raster graphics (pixel-based graphics) technology was new, **scanline rendering** was popular. In scanline rendering, each horizontal line of the screen was treated individually, and all the triangles that met a line were processed to produce a row of pixel values, after which the renderer moved on to the next line. Usually the new scanline intersected many of the same triangles as the previous one, and so lots of data reuse was possible. Figure 9.7 shows a typical situation: At row 3, only the orange pixel meets the triangle; at row 4, the two blue pixels do. At row 6, the four gray pixels meet the triangle, and after that the intersecting span begins to shrink.

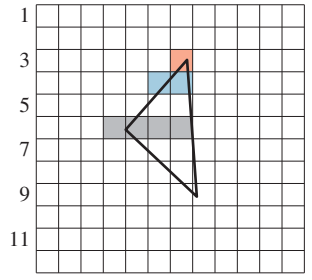

Figure 9.7: The processing of a single triangle by a scanline renderer.

One scheme that was used for interpolating RGB triples was to interpolate the vertex values along each edge of the triangle, and then to interpolate these across each scanline.

It's not difficult to show that this results in the same interpolant as the barycentric method we described (see Exercise 9.4).

But now suppose that we apply this method to a more interesting shape, like a quadrilateral. It's easy to see that the two congruent squares in Figure 9.8, with gray values of $0, 40, 0$, and 40 (in a range of 0 to 40) assigned to their vertices in clockwise order, lead to different interpolated values for the points P and P', the first being 20 and the second being 40.

The interpolated values in each configuration look decent, but when one makes an animation by rotating a shape, the interior coloring or shading seems to "swim" in a way that's very distracting.

What went wrong?

The *problem* was to infer values at the interior points of a polygon, given values at the vertices. But the *solution* depends on something unrelated to the problem, namely the scanlines in the output; this dependence shows up as an artifact in the results. The difficulty is that the solution is not based on the mathematics and physics of the problem, but rather on the computational constraints on the *solution*. When you limit the class of solutions that you're willing to examine *a priori,* there's always a chance that the best solution will be excluded. Of course, sometimes there's a good reason to constrain the allowable solutions, but in

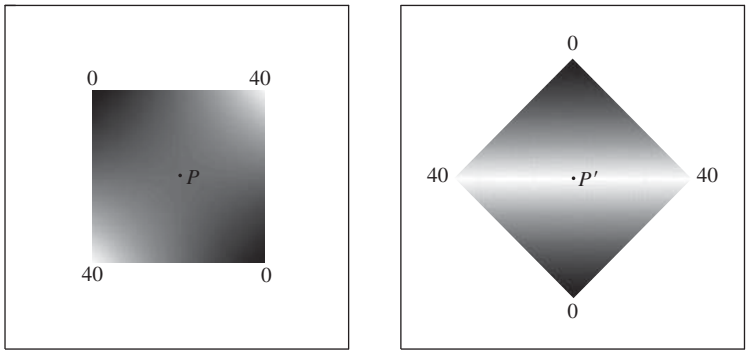

Figure 9.8: The congruent squares in these two renderings have the same gray values at corresponding vertices, but the scanline-interpolated gray values at P and P' differ.

doing so, you should be aware of the consequences. We embody these ideas in a principle:

> ✓ **THE DIVISION OF MODELING PRINCIPLE:** Separate the mathematical and/or physical model of a phenomenon from the numerical model used to represent it.

Attempting to computationally solve a problem often involves three separate choices. The first is an *understanding* of the problem; the second is a choice of *mathematical tools*; the third is a choice of *computational method*. For example, in trying to model ocean waves, we have to first look at what's known about them. There are large rolling waves and there are waves that crest and break. The first step is to decide which of these we want to simulate. Suppose we choose to simulate rolling (nonbreaking) waves. Then we can represent the surface of the water with a function, $y = f(t, x, z)$, expressing the height of the water at the point (x, z) at time t. Books on oceanography give differential equations describing how f changes over time. When it comes to *solving* the differential equation, there are many possible approaches. Finite-element methods, finite-difference methods, spectral methods, and many others can be used. If we choose, for example, to represent f as a sum of products of sines and cosines of x and z, then solving the differential equations becomes a process of solving systems of equations in the coefficients in the sums. If we limit our attention to a finite number of terms, then this problem becomes tractable.

But having made this choice, we can see certain influences: Because there's a highest-frequency sine or cosine in our sum, there's a smallest possible wavelength that can be represented. If we want our model to contain ripples smaller than this, the numerical choice prevents it. And if, having solved the problem, we decide we'd like to make waves that actually crest and break, our mathematical choice precludes it. We can, of course, add a "breaking wave" adjustment to the output of our model, altering the shapes of certain wave crests, but it should come as no surprise that such an ad hoc addition is more likely to produce problems than good results. Furthermore, it will make debugging of our system almost impossible, because without a clear notion of the "right solution," it's very difficult to detect an error conclusively.

The idea of carefully structuring your models and separating the mathematical from the numerical is discussed at length by Barzel [Bar92].

9.3 Limitations of Piecewise Linear Extension

The method of extending a function on a triangle mesh from vertices to the interiors of triangles is called **piecewise linear extension.** By looking at the basis functions—the tent-shaped graphs—we can see that the graphs of such extensions will have sharp corners and edges. Depending on the application, these artifacts may be significant. For instance, if we interpolate gray values across a triangle mesh, the human eye tends to notice the second-order discontinuities at triangle edges: When the gray values change linearly across the triangle interiors, things look fine; when the rate of change changes at a triangle edge, the eye picks it out.

Some people tend to see "bands" near such discontinuities; the effect is called *Mach banding* (see Section 1.7).

If we use piecewise linear interpolation in animation, having computed the positions of objects at certain "key" times, then between these times objects move with constant velocities, and hence zero acceleration; all the acceleration is concentrated in the key moments. This can be very distracting.

9.3.1 Dependence on Mesh Structure

If we have a polyhedral shape with nontriangular faces, we can triangulate each face to get a triangle mesh. Then we can, as before, interpolate function values at the vertices over the triangular faces. But the results of this interpolation can vary wildly depending on the particular triangulation. It's easiest to see this with a very simple example (see Figure 9.9) in which a function defined on the corners of a square is extended to the interior in two different ways. The results are evidently triangulation-dependent.

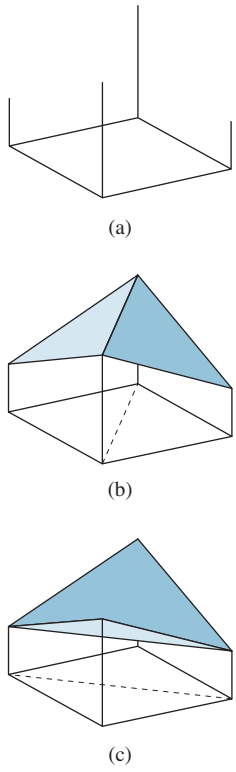

Figure 9.9: (a) A square with heights assigned at the four corners; (b) one piecewise linear interpolation of these values; and (c) a different interpolation of the same values.

9.4 Smoother Extensions

As we hinted above, taking function values at the vertices of a mesh and trying to find *smoothly* interpolated values over the interior of the mesh is a difficult task. Part of the difficulty arises in defining what it means to be a smooth function on a mesh. If the mesh happens to lie in the *xy*-plane, it's easy enough: We can use the ordinary definition of smoothness (existence of various derivatives) on the plane. But when the mesh is simply a polyhedral surface in 3-space (e.g., like a dodecahedron), it's no longer clear how to measure smoothness.

Of course, if we replace the dodecahedron with the sphere that passes through its vertices, then defining smoothness is once again relatively easy. Each point of the dodecahedron corresponds to a point on the surrounding sphere (e.g., by radial projection), and we can declare a function that's smooth on the sphere to be smooth on the dodecahedron as well. Unfortunately, finding a smooth shape that passes through the vertices of a polyhedron is itself an instance of the extension problem: We have a function (the *xyz*-coordinates of a point) defined at each vertex of the mesh; we'd like a function (the *xyz*-coordinates of the smooth-surface points) that's defined on the interiors of triangles. Such a function is what a solution to the smooth interpolation problem would give us. Thus, in suggesting that we use a smooth approximating shape, we haven't really simplified the problem at all.

A partial solution to this is provided by creating a sequence of meshes through a process called **subdivision** of the original surface. These subdivided meshes converge, in the limit, to a fairly smooth surface. We'll discuss this further in Chapter 22.

9.4.1 Nonconvex Spaces

The piecewise linear extension technique works when the values at the vertices are real numbers; it's easy to extend this to tuples of real numbers (just do the extension on one coordinate at a time). It's also easy to apply it to other spaces in which convex combinations, that is,

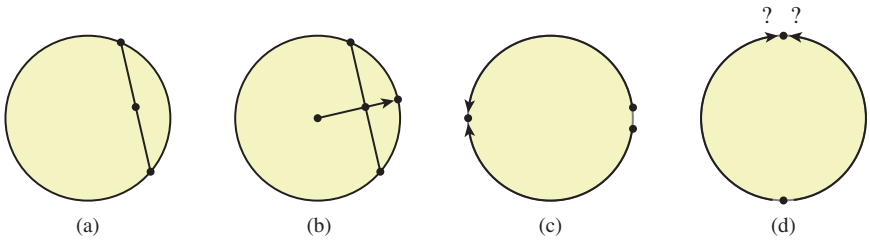

Figure 9.10: (a) If we take convex combinations of points on a circle in $\mathbf{R^2}$*, the result is a point in* $\mathbf{R^2}$*, but it is not generally on the circle; (b) if we radially project back to the circle, it works better … but the value is undefined when the original convex combination falls at the origin. (c) If we do angular interpolation, the point halfway between* 355° *and* 5° *turns out to be* 180°*. (d) If we try angular interpolation along the shortest route, our blend becomes undefined when the points are opposites.*

$$c_i f_i + c_j f_j + c_k f_k, \text{ where } c_1 + c_2 + c_3 = 1 \text{ and } c_1, c_2, c_3 \geq 0, \tag{9.14}$$

make sense. For example, if you have a 2×2 symmetric matrix associated to each vertex, you can perform barycentric blending of the matrices, because a convex combination of symmetric matrices is still a symmetric matrix.

Unfortunately, there are many interesting spaces in which convex combinations either don't make sense or fail to be defined for certain cases. The exemplar is the circle $\mathbf{S^1}$. If you treat the circle as a subset of $\mathbf{R^2}$, then (see Figure 9.10) forming convex combinations of two points makes sense … but the result is a point in the unit disk $\mathbf{D^2} \subset \mathbf{R^2}$, and generally *not* a point on $\mathbf{S^1}$.

The usual attempt at solving this is to "re-project" back to the circle, replacing the convex combination point C with $C/\|C\|$. Unfortunately, this fails when C turns out to be the origin. The problem is not one that can be solved by any clever programming trick.


It's a fairly deep theorem of topology that if

$$h : \mathbf{D^2} \to \mathbf{S^1} \tag{9.15}$$

has the property that $h(p) = p$ for all points of $\mathbf{S^1}$, then h *must* be discontinuous somewhere.

Another possible approach is to treat the values as angles, and just interpolate them. Doing this directly leads to some oddities, though: Blending some points in $\mathbf{S^1}$ that are very close (like 350° and 10°) yields a point (180° in this case) that's far from both. Doing so by "interpolating along the shorter arc" addresses *that* problem, but it introduces a new one: There are two shortest arcs between opposite points on the circle, so the answer is not well defined.


Once again, a theorem from topology explains the problem. If we simply consider the problem of finding a *halfway* point between two others, then we're seeking a function

$$H : \mathbf{S^1} \times \mathbf{S^1} \to \mathbf{S^1} \tag{9.16}$$

with certain properties. For instance, we want H to be continuous, and we want $H(p,p) = p$ for every point $p \in S^1$, and we want $H(p,q) = H(q,p)$, because "the halfway point between p and q" should be the same as "the halfway point between q and p." It turns out that even these two simple conditions are too much: No such function exists.

The situation is even worse, however: Interpolation between pairs of points, if it were possible, would let us extend our function's domain from vertices to edges of the mesh. Suppose that somehow we were given such an extension; could we *then* extend continuously over triangles? Alas, no. The study of when such extensions exist is a part of homotopy theory, and particularly of obstruction theory [MS74]. We mention this not because we expect you to learn obstruction theory, but because we hope that the existence of theorems like the ones above will dissuade you from trying to find ad hoc methods for extending functions whose codomains don't have simple enough topology.

9.4.2 Which Interpolation Method Should I Really Use?

The problem of interpolating over the interior of a triangle applies in many situations. If we are interpolating a color value, then linear interpolation in some space where equal color differences correspond to equal representation distances (see Chapter 28) makes sense. If we are interpolating a unit normal vector value, then linear interpolation is surely wrong, because the interpolated normal will generally not end up a unit vector, and if you use it in a computation that relies on unit normals, you'll get the wrong answer. And if the value is something discrete, like an object identifier, then interpolation makes no sense at all.

All of this seems to lead to the answer, "It depends!" That's true, but there's something deeper here:

> ✓ **THE MEANING PRINCIPLE:** For every number that appears in a graphics program, you need to know the semantics, the **meaning,** of that number.

Sometimes this meaning is given by units ("That number represents speed in meters per second"); sometimes it helps you place a bound on possible values for a variable ("This is a solid angle,[1] so it should be between 0 and $4\pi \approx 12.5$" or "This is a unit normal vector, so its length should be 1.0"); and sometimes the meaning is discrete ("This represents the number of paths that have ended at this pixel"). It's important to distinguish the *meaning* of the number from its representation. For instance, we often are interested in the **coverage** of a pixel (how much of a small square is covered by some shape) as a number α between zero and one, but α is sometimes represented as an 8-bit unsigned integer, that is, an integer between 0 and 255. Despite the discrete nature of the representation, it makes perfectly good sense to average two coverage values (although representing the average in the 8-bit form may introduce a roundoff error, of course).

9.5 Functions Multiply Defined at Vertices

Until now, we've discussed the very common case of functions that have a single value at each vertex and need to be interpolated across faces. But another situation arises frequently in graphics: a function where there's a value at each vertex *for each triangle that meets that vertex*. Consider, for instance, the colored octahedron in Figure 9.11. Each triangle has a color gradient across it, but no two

1. Solid angles are discussed in Chapter 26.

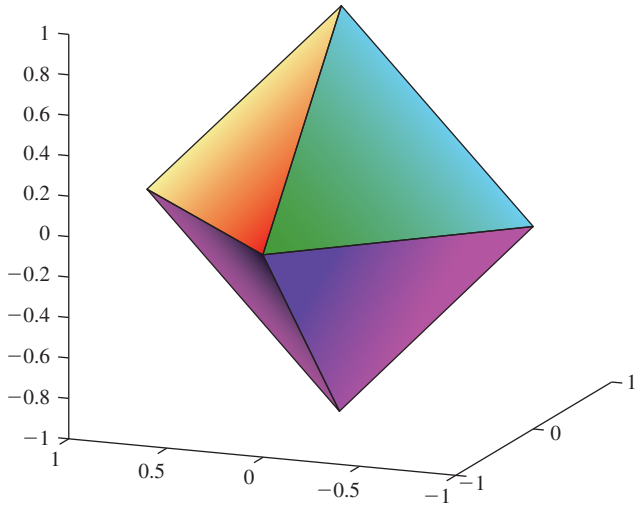

Figure 9.11: An octahedron on which each face has a different color gradient. At each vertex of the octahedron, we need to store four different colors, one for each of the faces.

triangles have the same color at each vertex. The way to generate a color function for this shape is to consider a function defined on a larger domain than the vertices. Consider all vertex-triangle pairs in which the triangle contains the vertex, that is,

$$Q = \{(v,t) : v \in t\} \subset V \times T, \tag{9.17}$$

where V and T are the sets of vertices and triangles of the mesh, respectively. Until now, we've taken a function defined h in V and extended it to all points of the mesh. We now instead define a function on Q and extend this to all points of the mesh. A point in a triangle t with vertices i, j, and k gets its value by a barycentric interpolation of the values $h(i,t)$, $h(j,t)$, and $h(k,t)$.

This leads to one important problem: A point on the edge (i,j) is a point of two different triangles. What color should it have? The answer is "It depends." From a strictly mathematical standpoint, there's no single correct answer; there will be a color associated to the interpolation of colors on one face, and a different one associated to the interpolation of colors on the other face. Neither is implicitly "right." The best we can do is to say that interpolation defines a function on

$$U = \{(P,t) : P \in t\} \subset M \times T, \tag{9.18}$$

where P is a point of the mesh M; that is, for each point P *and* each triangle t that contains it, we get a value $h(P,t)$. Since most points are in only one triangle, the second argument is generally redundant. But for those in more than one triangle, the function value is defined only with reference to the triangle under consideration.

9.6 Application: Texture Mapping

We mentioned in Chapter 1 that models are often described not only by geometry, but by **textures** as well: To each point of an object, we can associate some property (surface color is a common one), and this property is then used in rendering the

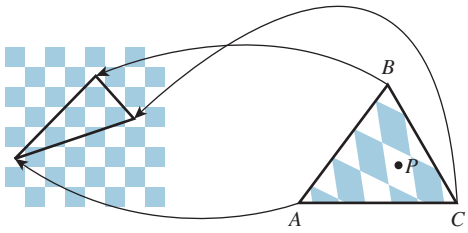

Figure 9.12: The point P of the triangle $T = \triangle ABC$ has its color determined by a texture map. The points A, B, and C have been assigned to points in the checkerboard image, as shown by the arrows; the point P corresponds to a point in a white square, so its texture color is white.

object. At a high level, when we are determining the color of a pixel in a rendering we typically base that computation on information about the underlying object that appears at that pixel; if it's a triangle of some mesh, we use, for instance, the orientation of that triangle in determining how brightly lit the point is by whatever illumination is in the scene. In some cases, the triangle may have been assigned a single color (i.e., its appearance under illumination by white light), which we also use, but in others there may be a color associated to each vertex, as in the previous section, and we can interpolate to get a color at the point of interest. Very often, however, the vertices of the triangle are associated to locations in a **texture map,** which is typically some $n \times k$ image; it's easy to think of the triangle as having been stretched and deformed to sit in the image. The color of the point of interest is then determined by looking at its location in the texture map, as in Figure 9.12, and finding the color there.

9.6.1 Assignment of Texture Coordinates

When we say that sometimes the vertices of the triangle are associated to locations in a texture map, it's natural to ask, "How did they get associated?" The answer is "by the person who created the model." There are some simple models for which the association is particularly easy. If we start with an $n \times k$ grid of triangles as shown in Figure 9.13, top, with $n = 6$ and $k = 8$, we can associate to each vertex (i, j) of this mesh a point in 3-space by setting

$$\theta = 2\pi j/(k-1) \tag{9.19}$$

$$\phi = \phi = -\frac{\pi}{2} + \pi i/(n-1) \tag{9.20}$$

$$X = \cos(\theta)\cos(\phi) \tag{9.21}$$

$$Y = \sin(\phi) \tag{9.22}$$

$$Z = \sin(\theta)\cos(\phi), \tag{9.23}$$

that is, letting θ and ϕ denote longitude and latitude, respectively. The approximately spherical shape that results is shown in the middle. We also have a 100×200 texture image of an "unprojected" map of the Earth (the vertical coordinate is proportional to latitude; the horizontal proportional to longitude). We assign texture coordinates $100i/(n-1), 200j/(k-1)$ to the vertex at position (i, j), and the resultant globe is shown rendered with the texture map at the bottom.

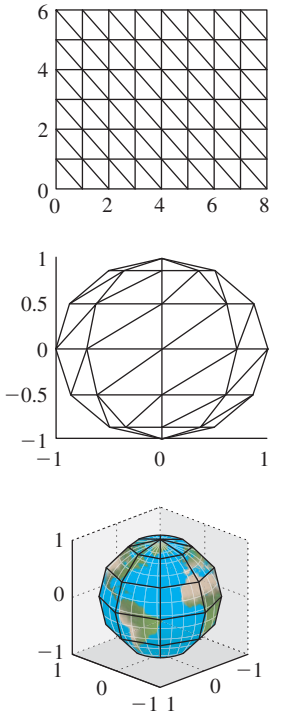

Figure 9.13: Texture-mapping a globe.

In this case, the way we created the mesh made it natural to create the texture coordinates at the same time. There is one troubling aspect of this approach, though: The texture coordinates depend on the number of pixels in the image that we use as our world map. If we created this shape and decided that the result looked bad, we might want to use a higher-resolution image, but that would entail changing the texture coordinates as well. Because of this, texture coordinates are usually specified as numbers between zero and one, representing a fraction of the way up[2] or across the image, so texture coordinates $(0.75, 0.5)$ correspond to a point three-quarters of the way up the texture image (regardless of size) and midway across, left to right.

It's commonplace to name these texture coordinates u and v so that a typical vertex now has five attributes: x, y, z, u, and v. Sometimes texture coordinates are referred to as ***uv*-coordinates.**

9.6.2 Details of Texture Mapping

If we have a mesh with texture coordinates assigned at each vertex, how do we determine the texture coordinates at some location within a triangle? *We use the techniques of this chapter, one coordinate at a time.* For instance, we have a u-coordinate at every vertex; such an assignment of a real value at every vertex uniquely defines a piecewise linear function at every point of the mesh: If P is a point of the triangle ABC, and the u-coordinates of A, B, and C are u_A, u_B, and u_C, we can determine a u-coordinate for P using barycentric coordinates. We write P in the form

$$P = \alpha A + \beta B + \gamma B \tag{9.24}$$

and then define

$$u(P) = \alpha u_A + \beta u_B + \gamma u_C. \tag{9.25}$$

We can do the same thing for v, and this uniquely determines the uv-coordinates for P.

If the triangle ABC happens to cover many pixels, we'll do this computation—find the barycentric coordinates and use them to combine the texture coordinates from the vertices—many times. Fortunately, the regularity of pixel spacing makes this repeated computation particularly easy to do in hardware, as described in Chapter 38.

9.6.3 Texture-Mapping Problems

If a triangle has texture coordinates that make it cover a large area of the texture image, but the triangle itself, when rendered, occupies a relatively small portion of the final image, then each pixel in the final image (thought of as a little square) corresponds to *many* pixels in the texture image. Our approach finds a *single* point in the texture image for each final image pixel, but perhaps it seems that the right thing to do would be to blend together many texture image pixels to get a combined result. If we do *not* do this, we get an effect called **texture aliasing,** which we'll discuss further in Chapters 17, 20 and 38. If we *do* blend together texture

2. Or down, if the image pixels are indexed from top to bottom, as often happens.

image pixels for each pixel to be rendered, the texturing process becomes very slow. One way to address this is through precomputation; we'll discuss a particular form of this precomputation, called **MIP mapping,** in Chapter 20.

9.7 Discussion

The main idea of this chapter is so simple that many graphics researchers tend to not even notice it: A real-valued function on the vertices of a mesh can be extended piecewise linearly to a real-valued function on the whole mesh, via the barycentric coordinates on each triangle. In a triangle with vertices v_i, v_j, and v_k, where the values are f_i,f_j, and f_k, the value at the point whose barycentric coordinates are c_i, c_j, and c_k is $c_i f_i + c_j f_j + c_k f_k$. This extension from vertices to interior is so ingrained that it's done without mention in countless graphics papers. There *are* other possible extensions from values at vertices to values on whole triangles, some of which are discussed in Chapter 22, but this piecewise linear interpolation dominates.

This same piecewise linear interpolation method works for functions with other codomains (e.g., $\mathbf{R^2}$ or $\mathbf{R^3}$), as long as those codomains support the idea of a "convex combination." For domains that do not (like the circle, or the sphere, or the set of 3×3 rotation matrices) there may be no reasonable way to extend the function over triangles.

Solving problems like this interpolation from vertices in the abstract helps avoid implementation-dependent artifacts. If we had studied the problem in the context of interpolating values represented by 8-bit integers across the interior of a triangle in a GPU, we might have gotten distracted by the low bit-count representation, rather than trying to understand the general problem and then adapt the solution to our particular constraints. This is another instance of the Approximate the Solution principle.

One important application of piecewise linear interpolation is texture mapping, in which a property of a surface is associated to each vertex, and these property values are linearly interpolated over the interiors of triangles. If the property is "a location in a texture image," then the interpolated values can be used to add finely detailed variations in color to the object. Chapter 20 discusses this.

9.8 Exercises

Exercise 9.1: The basis functions we described in this chapter, shown in Figure 9.4(a), not only *correspond* to a basis for $\mathbf{R}^n$, they also *form* a basis of another vector space—a subspace of the vector space of all continuous functions on the domain $[1, n]$. To justify this claim, show that these functions are in fact linearly independent *as functions*.

Exercise 9.2: (a) Draw a tetrahedron; pick a vertex and draw its link and its star. Suppose v and w are distinct vertices of the tetrahedron. What is the intersection of the star of v and the star of w?
(b) Draw an octahedron, and answer the same question when v is the top vertex and w is the bottom one.

Exercise 9.3: Think about a mesh that contains vertices, edges, triangles, and tetrahedra. You could call this a *solid* mesh rather than a surface mesh. Consider a

nonboundary vertex of such a mesh. What is the topology of the star of the vertex? What is the topology of the link of the vertex?

Exercise 9.4: Show that the interpolation of values across a triangle using the scanline method is the same as the one defined using the barycentric method. Hint: Show that if the triangle is in the xy-plane, then each of the methods defines a function of the form $f(x, y) = Ax + By + C$. Now suppose you have two such functions with the same values at the three vertices of a triangle. Explain why they must take the same values at all interior points as well.

Exercise 9.5: The function H of Equation 9.16 cannot exist; here's a reason why. Suppose that it did. Define a new function

$$K : [0, 2\pi] \times [0, 2\pi] \to [0, 2\pi] : (\theta, \phi) \mapsto H(\theta, \phi), \tag{9.26}$$

where we're implicitly using the correspondence of the number θ in the interval $[0, 2\pi]$ with the point $(\cos\theta, \sin\theta)$ of $\mathbf{S^1}$. Now consider the loop in the domain of K consisting of three straight lines, from $(0, 0)$ to $(2\pi, 0)$, from $(2\pi, 0)$ to $(2\pi, 2\pi)$, and from $(2\pi, 2\pi)$ back to $(0, 0)$.

(a) Draw this path.

(b) For each point p of this path, $K(p)$ is an element of $\mathbf{S^1}$. Restricting K to this path gives a map from the path to $\mathbf{S^1} \subset \mathbf{R^2}$. We can compute the winding number of this path about the origin in $\mathbf{R^2}$. Explain, using the assumed properties of H, why the winding numbers of the first two parts of the path must be equal.

(c) Explain why the winding number of the last part must be one.

(d) Conclude that the total winding number must be odd.

(e) Now consider shrinking the triangular loop toward the center of the triangle. The winding number will be a continuous integer-valued function of the triangle size. Explain why this means the winding number must be constant.

(f) When the triangle has shrunk to a single point, explain why the winding number must be zero.

(g) Explain why this is a contradiction.

Exercise 9.6: Use the 2D test bed to create a program to experiment with texture mapping. Display, on the left, a 100×100 checkerboard image, with 10×10 squares. Atop this, draw a triangle whose vertices are draggable. On the right, draw a fixed equilateral triangle above a 100×100 grid of tiny squares (representing display pixels). For each of these display pixels, compute and store the barycentric coordinates of its center (with respect to the equilateral triangle). Using the locations of the three draggable vertices in the checkerboard as texture coordinates, compute, for each display pixel within the equilateral triangle, the uv-coordinates of the pixel center, and then use these uv-coordinates to determine the pixel color from the checkerboard texture image (see Figure 9.14). Experiment with mapping the equilateral triangle to a tiny triangle in the texture image, and to a large one; experiment with mapping it to a tall and narrow triangle. What problems do you observe?

Exercise 9.7: Suppose that you have a polyline in the plane with vertices $P_0, P_1, \ldots, P_n$ and you want to "resample" it, placing multiple points on each edge, equally spaced, for a total of $k + 1$ points $Q_0 = P_0, \ldots, Q_k = P_n$.

(a) Write a program to do this: First compute the total length L of the polyline, then place the Qs along the polyline so that they're separated by L/k. This will require special handling at each vertex of the original polyline.

(b) When you're done, you'll notice that if n and k are nearly equal, many "corners" tend to get cut off the original polyline. It's natural to say, "I want equally

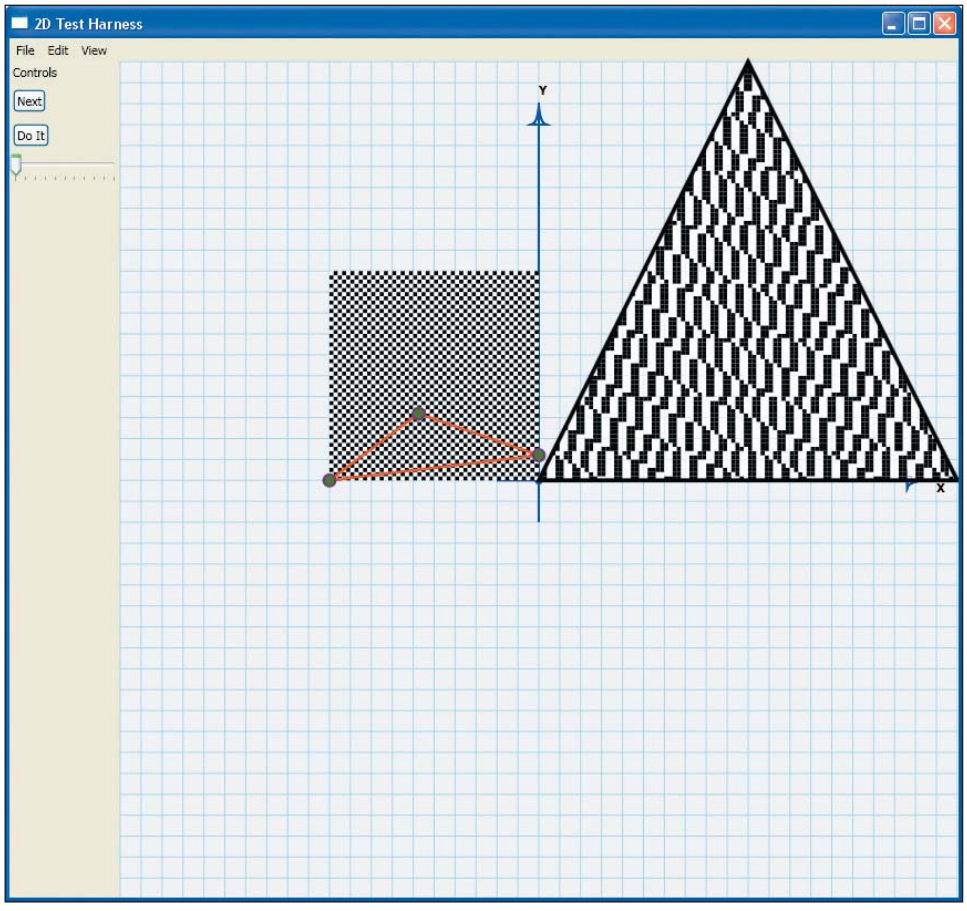

Figure 9.14: A screen capture for the texture-mapping program of Exercise 9.6, showing the texture on the left, a large triangle, and the locations of the triangle's vertices in texture coordinates atop the texture. The resultant texturing on the triangle's interior is shown on the right.

spaced points, but I want them to include all the original points!" That's generally not possible, but we can come close. Suppose that the shortest edge of the original polygon has length s. Show that you can place approximately L/s points $Q_0, Q_1, \ldots$ on the original polyline, including all the points $P_0, \ldots, P_n$, with the property that the ratio of the greatest gap between adjacent points and the smallest gap is no more than 2.
(c) Suppose you let yourself place CL/s points with the same constraints as in the previous part, for some C greater than one. Estimate the max-min gap ratio in terms of C.

Exercise 9.8: Consider the interval $[p, q]$ where $p \neq q$. If we define $\alpha(x) = \frac{x-p}{q-p}$ and $\beta(x) = \frac{x-q}{p-q}$, then α and β are called the **barycentric coordinates of** x.
(a) Show that for $x \in [p, q]$, both $\alpha(x)$ and $\beta(x)$ are between 0 and 1.
(b) Show that $\alpha(x) + \beta(x) = 1$.
◈ (c) Clearly α and β can be defined on the rest of the real line, and these definitions depend on p and q; if we call them α_{pq} and β_{pq}, then we can, for another

interval $[p', q']$, define corresponding barycentric coordinates. How are $\alpha_{pq}(x)$ and $\alpha_{p',q'}(x)$ related?

Exercise 9.9: Suppose you have a nondegenerate triangle in 3-space with vertices P_0, P_1, and P_2 so that $\mathbf{v}_1 = P_1 - P_0$ and $\mathbf{v}_2 = P_2 - P_0$ are nonzero and nonparallel. Further, suppose that we have values $f_0, f_1, f_2 \in \mathbf{R}$ associated with the three vertices. Barycentric interpolation of these values over the triangle defines a function that can be written in the form

$$f(P) = f_0 + (P - P_0) \cdot \mathbf{w} \tag{9.27}$$

for some vector $\mathbf{w}$. We'll see this in two steps: First, we'll compute a possible value for $\mathbf{w}$, and then we'll show that if it has this value, f actually matches the given values at the vertices.

(a) Show that the vector $\mathbf{w}$ must satisfy $\mathbf{v}_i \cdot \mathbf{w} = f_i - f_0$ for $i = 1, 2$ for the function defined by Equation 9.27 to satisfy $f(P_1) = f_1$ and $f(P_2) = f_2$.

(b) Let $\mathbf{S}$ be the matrix whose columns are the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$. Show that the conditions of part (a) can be rewritten in the form

$$\mathbf{S}^\mathbf{T}\mathbf{w} = \begin{bmatrix} f_1 - f_0 \\ f_2 - f_0 \end{bmatrix}, \tag{9.28}$$

and that therefore $\mathbf{w}$ must also satisfy

$$\mathbf{S}\mathbf{S}^\mathbf{T}\mathbf{w} = \mathbf{S} \begin{bmatrix} f_1 - f_0 \\ f_2 - f_0 \end{bmatrix}. \tag{9.29}$$

(c) Explain why $\mathbf{S}\mathbf{S}^\mathbf{T}$ must be invertible.

(d) Conclude that $\mathbf{w} = (\mathbf{S}\mathbf{S}^\mathbf{T})^{-1}\mathbf{S} \begin{bmatrix} f_1 - f_0 \\ f_2 - f_0 \end{bmatrix}$.

(e) Verify that if we use this formula for $\mathbf{w}$, then $f(P_i) = f_i$ for $i = 0, 1, 2$.

(f) Suppose that $\mathbf{w}' = \mathbf{w} + \alpha\mathbf{n}$, where $\mathbf{n} = \mathbf{v}_1 \times \mathbf{v}_2$ is the normal vector to the triangle. Show that we can replace $\mathbf{w}$ with $\mathbf{w}'$ in the formula for f and still have $f(P_i) = f_i$ for $i = 0, 1, 2$.