4

Convex Hulls in Three Dimensions

The focus of this chapter is algorithms for constructing the convex hull of a set of points in three dimensions (Section 4.2). We will also touch on related issues: properties of polyhedra (Section 4.1), how to represent polyhedra (Section 4.4), and a brief exploration of higher dimensions (Section 4.6). Finally, several related topics will be explored via a series of exercises (Section 4.7). The centerpiece of the chapter is the most complex implementation in the book: code for constructing the three-dimensional hull via the incremental algorithm (Section 4.3).

4.1. POLYHEDRA

4.1.1. Introduction

A polyhedron is the natural generalization of a two-dimensional polygon to three-dimensions: It is a region of space whose boundary is composed of a finite number of flat polygonal faces, any pair of which are either disjoint or meet at edges and vertices. This description is vague, and it is a surprisingly delicate task to make it capture just the right class of objects. Since our primary concern in this chapter is convex polyhedra, which are simpler than general polyhedra, we could avoid a precise definition of polyhedra. But facing the difficulties helps develop three-dimensional geometric intuition, an invaluable skill for understanding computational geometry.

We concentrate on specifying the boundary or surface of a polyhedron. It is composed of three types of geometric objects: zero-dimensional vertices (points), one-dimensional edges (segments), and two-dimensional faces (polygons). It is a useful simplification to demand that the faces be *convex* polygons, which we defined to be bounded (Section 1.1.1). This is no loss of generality since any nonconvex face can be partitioned into convex ones, although we must then allow adjacent faces to be coplanar. What constitutes a valid polyhedral surface can be specified by conditions on how the components relate to one another. We impose three types of conditions: The components intersect "properly," the local topology is "proper," and the global topology is "proper." We now expand each of these constraints.

1. Components intersect "properly."
   For each pair of faces, we require that either
   (a) they are disjoint, or
   (b) they have a single vertex in common, or
   (c) they have two vertices, and the edge joining them, in common.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

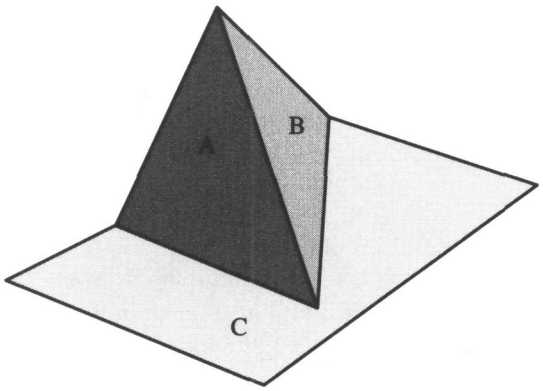


FIGURE 4.1 Faces A and B meet C improperly even though they do not penetrate C.

This is where the assumption that faces are convex simplifies the conditions. Improper intersections include not only penetrating faces, but also faces touching in the "wrong" way; see Figure 4.1. There is no need to specify conditions on the intersection of edges and vertices, as the condition on faces covers them also. Thus an improper intersection of a pair of edges implies an improper intersection of faces.

2. Local topology is "proper."

The local topology is what the surface looks like in the vicinity of a point. This notion has been made precise via the notion of *neighborhoods*: arbitrarily small portions (open regions) of the surface surrounding a point. We seek to exclude the three objects shown in Figure 4.2. In all three examples in that figure, there are points that have neighborhoods that are not topological two-dimensional disks. The technical way to capture the constraint is to require the neighborhoods of every point on the surface to be "homeomorphic" to a disk. A *homeomorphism* between two regions permits stretching and bending, but no tearing.[1] A fly on the surface would find the neighborhood of every point to be topologically like a disk. A surface for which this is true for every point is called a *2-manifold*, a class more general than the boundaries of polyhedra.

We have expressed the condition geometrically, but it is useful to view it combinatorially also. Suppose we triangulate the polygonal faces. Then every vertex is the apex of a number of triangles. Define the *link* of a vertex v to be the collection of edges opposite v in all the triangles incident to v.[2] Thus the link is in a sense the combinatorial neighborhood of v. For a legal triangulated polyhedron, we require that the link of every vertex be a simple, closed polygonal path. The link for the circled vertex in Figure 4.2(b), for example, is not such a path.

[1] Two sets are *homeomorphic* if there is a mapping between them that is one-to-one and continuous, and whose inverse is also continuous. See, e.g., Mendelson (1990, pp. 90–1). This concept is different from a *homomorphism*.

[2] My discussion here is indebted to that of Giblin (1977, pp. 51–3).

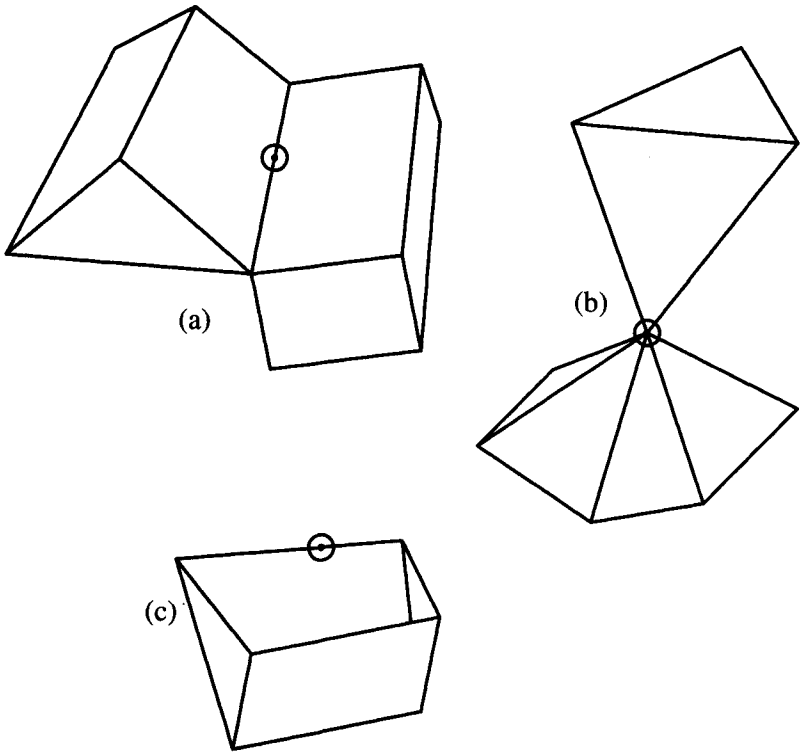


FIGURE 4.2 Three objects that are not polyhedra. In all three cases, a neighborhood of the circled point is not homeomorphic to an open disk. In (a) the point lies both on the top surface shown and on a similar surface underneath. Object (c) is not closed, so the indicated point's neighborhood is a half-disk.

One consequence of this condition is that every edge is shared by exactly two faces.

3. Global topology is "proper."

We would like the surface to be connected, closed, and bounded. So we require that the surface be connected in the sense that from any point, one may walk to any other on the surface. This can be stated combinatorially by requiring that the *1-skeleton*, the graph of edges and vertices, be connected. Note that this excludes, for instance, a cube with a "floating" internal cubical cavity. Together with stipulating a finite number of faces, our previous conditions already imply closed and bounded, although this is perhaps not self-evident (Exercise 4.1.6[1]).

One might be inclined to rule out "holes" in the definition of polyhedron, holes in the sense of "channels" from one side of the surface to the other that do not disconnect the exterior (unlike cavities). Should a torus (a shape like a doughnut) be a polyhedron? We adopt the usual terminology and permit polyhedra to have an arbitrary number of such holes. The number of holes is called the *genus* of the surface. Normally we will only consider polyhedra with genus zero: those topologically equivalent to the surface of a sphere.

In summary, the boundary of a polyhedron is a finite collection of planar, bounded convex polygonal faces such that

1. the faces intersect properly (as in (1) above);
2. the neighborhood of every point is topologically an open disk, or (equivalently) the link of every vertex is a simple polygonal chain; and
3. the surface is connected, or (equivalently) the 1-skeleton is connected.

The boundary is closed and encloses a bounded region of space. Every edge is shared by exactly two faces; these faces are called *adjacent*.

Convex polyhedra are called *polytopes*, or sometimes 3-polytopes to emphasize their three-dimensionality.[3] A polytope is a polyhedron that is convex in that the segment connecting any two of its points is inside. Just as convex polygons can be characterized by the local requirement that each vertex be convex, polytopes can be specified locally by requiring that all *dihedral* angles be convex ($\leq \pi$). Dihedral angles are the internal angles in space at an edge between the planes containing its two incident faces. For any polytope, the sum of the face angles around each vertex are at most 2π, but this condition does not alone imply convexity (Exercise 4.1.6[5]).

It is important for building intuition and testing out ideas to become intimately familiar with a few polyhedra. We therefore take time to discuss the five Platonic solids.

4.1.2. Regular Polytopes

A *regular polygon* is one with equal sides and equal angles: equilateral triangle, square, regular pentagon, regular hexagon, and so on. Clearly there are an infinite variety of regular polygons, one for each n. It is natural to examine *regular polyhedra*; they are convex, so they are often called *regular polytopes*. The greatest regularity one can impose is that all faces are congruent regular polygons, and the number of faces incident to each vertex is the same for all vertices. It turns out that these conditions imply equal dihedral angles, so that need not be included in the definition.

The surprising implication of these regularity conditions is that there are only five distinct types of regular polytopes! These are known as the *Platonic solids*, since they are discussed in Plato's *Timaeus*.[4]

We now prove that there are exactly five regular polytopes. The proof is pleasingly elementary. The intuition is that the internal angles of a regular polygon grow large with the number of vertices of the polygon, but there is only so much room to pack these angles around each vertex.

Let p be the number of vertices per face; so each face is a regular p-gon. The sum of the faces angles for one p-gon is $\pi(p-2)$ (Corollary 1.2.5), so each face angle is $1/p$-th of this, $\pi(1-2/p)$.

Let v be the number of faces meeting at a vertex. The key constraint is that the sum of the face angles meeting at a vertex is less than 2π, in order for the polyhedron to be

[3] The notation in the literature is unfortunately not standardized. I define a polytope to be convex and bounded, and a polyhedron to be bounded. Some define a polytope to be bounded but permit a polyhedron to be unbounded. Some do not require a polytope to be convex. Often polytopes have arbitrary dimensions.

[4] It seems that the constructions in Plato may originate with the Pythagoreans (Heath 1956, Vol. 2, p. 98). See Malkevitch (1988) and Cromwell (1997) for the history of polyhedra.

Table 4.1. Legal p/v values.

p	v	$(p-2)(v-2)$	Name	Description
3	3	1	Tetrahedron	3 triangles at each vertex
4	3	2	Cube	3 squares at each vertex
3	4	2	Octahedron	4 triangles at each vertex
5	3	3	Dodecahedron	3 pentagons at each vertex
3	5	3	Icosahedron	5 triangles at each vertex

convex.[5] This can be seen intuitively by noticing that if the polyhedron surface is flat in the vicinity of a vertex, the sum of the angles is exactly 2π; and the sum of angles at a needle-sharp vertex is quite small. So the angle sum is in the range $(0, 2\pi)$. Thus we have v angles, each $\pi(1-2/p)$, which must sum to less than 2π. We transform this inequality with a series of algebraic manipulations to reach a particularly convenient form:

$$v\pi(1-2/p) < 2\pi, \tag{4.1}$$

$$1-2/p < 2/v,$$

$$pv < 2v+2p,$$

$$pv-2v-2p+4 < 4,$$

$$(p-2)(v-2) < 4. \tag{4.2}$$

Both p and v are of course integers. Because a polygon must have at least three sides, $p \geq 3$. It is perhaps less obvious that $v \geq 3$: At least three faces must meet at each vertex, for no "solid angle" could be formed at v with only two faces. These constraints suffice to limit the possibilities to those listed in Table 4.1. For example, $p = 4$ and $v = 4$ leads to $(p-2)(v-2) = 4$, violating the inequality. And indeed if four squares are pasted at a vertex, they must be coplanar, and this case cannot lead to a polyhedron.

It is not immediately evident why the listed p and v values lead to the objects claimed: These numbers provide local information, from which the global structure must be inferred. We will not take the time to perform this deduction; examining the five polytopes in Figure 4.3[6] quickly reveals they realize the $\{p, v\}$ numbers.[7] Counting vertices, edges, and faces leads to the numbers in Table 4.2.

The Greek prefixes in the names refer to the number of faces: tetra = 4, octa = 8, dodeca = 12, icosa = 20. Sometimes a cube is called a "hexahedron"!

4.1.3. Euler's Formula

In 1758 Leonard Euler noticed a remarkable regularity in the numbers of vertices, edges, and faces of a polyhedron of genus zero: The number of vertices and faces together

[5] We only consider "real" vertices, at which the face angles sum to strictly less than 2π.

[6] This figure and most of the three-dimensional illustrations in the book were generated using *Mathematica*.

[7] This pair of numbers is called the *Schläfi symbol* for the polyhedron (Coxeter 1973, p. 14).

Table 4.2. Number of Vertices, Edges, and Faces of the five regular polytopes.

Name	$\{p, v\}$	V	E	F
Tetrahedron	{3, 3}	4	6	4
Cube	{4, 3}	8	12	6
Octahedron	{3, 4}	6	12	8
Dodecahedron	{5, 3}	20	30	12
Icosahedron	{3, 5}	12	30	20

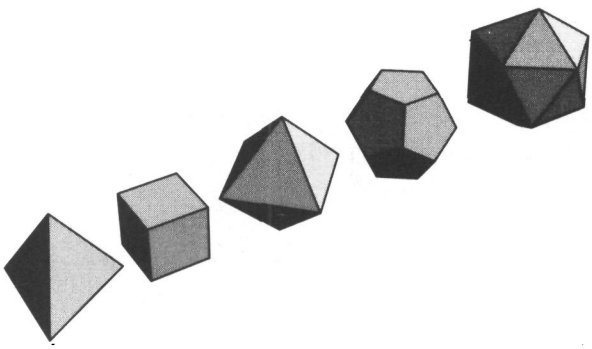


FIGURE 4.3 The five Platonic solids (left to right): tetrahedron, cube, octahedron, dodecahedron, and icosahedron.

is always two more than the number of edges; and this is true for *all* polyhedra. So a cube has 8 vertices and 6 faces, and $8 + 6 = 14$ is two more than its 12 edges. And the remaining regular polytopes can be seen to satisfy the same relationship. If we let V, E, and F be the number of vertices, edges, and faces respectively of a polyhedron, then what is now known as *Euler's formula* is:

$$V - E + F = 2\,. \tag{4.3}$$

One might think that recognizing this regularity is no great achievement, but Euler had to first "invent" the notions of vertex and edge to formulate his conjecture. It was many years before mathematicians developed a rigorous proof,[8] although with modern methods it is not too difficult. We now turn to a proof.

4.1.4. Proof of Euler's Formula

Our proof comprises three parts:

1. Converting the polyhedron surface to a plane graph.
2. The theorem for trees.
3. Proof by induction.

[8] See Lakatos (1976) for the fascinating history of this theorem.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

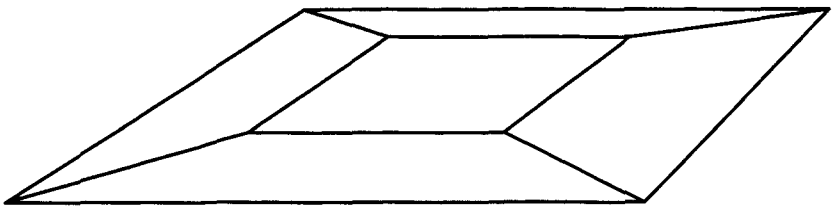


FIGURE 4.4 The 1-skeleton of a cube, obtained by flattening to a plane.

We first "flatten" the polyhedron surface P onto a plane, perhaps with considerable distortion, by the following procedure. Imagine the surface is made of a pliable material. Choose an arbitrary face f of P and remove it, leaving a hole in the surface. Now stretch the hole wider and wider until it becomes much larger than the original size of P. It should be intuitively plausible that one can then flatten the surface onto the plane, resulting in a *plane* graph G (the 1-skeleton of the polytope):[9] a graph embedded in the plane without edge crossings, whose nodes derive from vertices of P, and whose arcs derive from edges of P. The edges of f become the outer boundary of G. Each face of P except for f becomes a bounded face of G; f becomes the exterior, unbounded face of G. Figure 4.4 illustrates the graph that results from flattening a cube. Thus if we count this exterior face of G as a true face (which is the usual convention), then the vertices, edges, and faces of P are in one-to-one correspondence with those of G. This permits us to concentrate on proving Euler's formula for plane graphs.

The second step is to prove the formula in the highly restricted case where G is a tree. Of course a tree could never result from stretching a polyhedron, but this is a useful tool for the final step of the proof. So suppose G is a tree of V vertices and E edges. It is a property of trees that $V = E + 1$, a fact we assume for the proof. A tree bounds or delimits only one face, the exterior face, so $F = 1$. Now Euler's formula is immediate:

$$V - E + F = (E + 1) - E + 1 = 2.$$

The third and final step of the proof is by induction on the number of edges. Suppose Euler's formula is true for all connected graphs with no more than $E - 1$ edges, and let G be a graph of V, E, and F vertices, edges, and faces. If G is a tree, we are done by the previous argument without even using induction. So suppose G has a cycle, and let e be an edge of G in some cycle. The graph $G' = G \setminus e$ is connected,[10] with V vertices, $E - 1$ edges, and (here is the crux) $F - 1$ faces: Removal of e must join two faces into one. By the induction hypothesis,

$$V - (E - 1) + (F - 1) = 2 = V - E + F,$$

and we are finished.

[9]Note that this flattening would not work for genus greater than zero.

[10]$G \setminus e$ is the graph G with edge e removed.

4.1.5. Consequence: Linearity

We now show that Euler's formula implies that the number of vertices, edges, and faces of a polytope are linearly related: If $V = n$, then $E = O(n)$ and $F = O(n)$. This will permit us to use "n" rather loosely in complexity analyses involving polyhedra.

Because we seek to establish an upper bound on E and F as a function of $V = n$, it is safe to triangulate every face of the polytope, for this will only increase E and F without affecting V. So for the remainder of this argument we assume the polytope is *simplicial*: All of its faces are triangles.[11] If we count the edges face by face, then because each face has three edges, we get $3F$. But since each edge is shared by two faces, this double-counts the edges. So $3F = 2E$. Now substitution into Euler's formula establishes the linear bounds:

$$\begin{aligned} V - E + F &= 2, \\ V - E + 2E/3 &= 2, \\ V - 2 &= E/3, \\ E = 3V - 6 < 3V &= 3n = O(n), \qquad (4.4) \\ F = 2E/3 = 2V - 4 < 2V &= 2n = O(n)\ . \qquad (4.5) \end{aligned}$$

We summarize in a theorem for later reference:

Theorem 4.1.1. *For a polyhedron with $V = n$, E, and F vertices, edges, and faces respectively, $V - E + F = 2$, and both E and F are $O(n)$.*

Cromwell (1997) is a good source for further information on polyhedra.

4.1.6. Exercises

1. *Closed and bounded.* Argue that the definition of a polyhedron in the text guarantees that it is closed and bounded.
2. *Flawed definition 1.* Here is a "flawed" definition of polyhedron; call the objects so defined polyhedra$_1$. Find objects that are polyhedra$_1$ but are not polyhedra according to the definition in the text.

   A *polyhedron*$_1$ is a region of space bounded by a finite set of polygons such that every polygon shares at least one edge with some other polygon, and every edge is shared by exactly two polygons.
3. *Flawed definition 2.* Do the same as the previous exercise, but with this definition:

   A *polyhedron*$_2$ is a region of space bounded by a finite set of polygons such that every edge of a polygon is shared by exactly one other polygon, and no subset of these polygons has the same property.
4. *Cuboctahedron* [easy]. Verify Euler's formula for the *cuboctahedron*: a polytope formed by slicing off each corner of a unit cube in such a fashion that each corner is sliced down to an equilateral triangle of side length $\sqrt{2}/2$, and each face of the cube becomes a diamond: a square again of side length $\sqrt{2}/2$. Make a rough sketch of the polytope first.

[11] A triangle is a two-dimensional *simplex*, and thus the name "simplicial."

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

5. *Milk carton* (Saul Simhon). Find an example of a nonconvex polyhedron such that the sum of the face angles around each vertex is no more than 2π.
6. *Euler's formula for nonzero genus.* There is a version of Euler's Formula for polyhedra of any genus. Guess the formula based on empirical evidence for genus 1: polyhedra topologically equivalent to a torus.
7. *No 7-edge polyhedron* [easy]. Prove that there is no polyhedron with exactly seven edges.
8. *Polyhedra in FV-space.* Show that to every integer pair (F, V) satisfying

$$\frac{1}{2}F + 2 \leq V \leq 2F - 4$$

there exists a simple polyhedron of F faces and V vertices.

9. *Polyhedral torus* [difficult]. What is the fewest number of triangles needed to build a polyhedral torus? (Certainly four triangles are not enough.) What is the fewest number of vertices? Design a polyhedral torus, attempting to minimize the combinatorial size of the surface measured in these two ways.
10. *Gauss–Bonnet theorem.* Compute the total sum of the face angles at all the vertices of a few polyhedra (of genus 0), and formulate a conjecture.

4.2. HULL ALGORITHMS

Algorithms for constructing the hull in three dimensions are much more complex than two-dimensional algorithms, and the coverage here will be necessarily uneven. We will only mention gift wrapping, and talk through divide and conquer at a high level. The bulk of this chapter plunges into the incremental algorithm in full detail. Section 4.5 will sketch a randomized version of the incremental algorithm.

4.2.1. Gift Wrapping

As mentioned previously, the gift-wrapping algorithm was invented to work in arbitrary dimensions (Chand & Kapur 1970). The three-dimensional version is a direct generalization of the two-dimensional algorithm. At any step, a connected portion of the hull is constructed. A face F on the boundary of this partial hull is selected, and an edge e of this face whose second adjacent face remains to be found is also selected. The plane π containing F is "bent" over e toward the set until the first point p is encountered. Then conv $\{p, e\}$ is a new triangular face of the hull, and the wrapping can continue. As in two dimensions, p can be characterized by the minimum turning angle from π. A careful implementation can achieve $O(n^2)$ time complexity: $O(n)$ work per face, and as we just saw in Theorem 4.1.1, the number of faces is $O(n)$. And as in two dimensions, this algorithm has the advantage of being output-size sensitive: $O(nF)$ for a hull of F faces.

4.2.2. Divide and Conquer

The only lower bound for constructing the hull in three dimensions is the same as for two dimensions: $\Omega(n \log n)$ (Section 3.6). The question then naturally arises if this complexity is achievable in three dimensions, as it is in two dimensions. We mentioned in

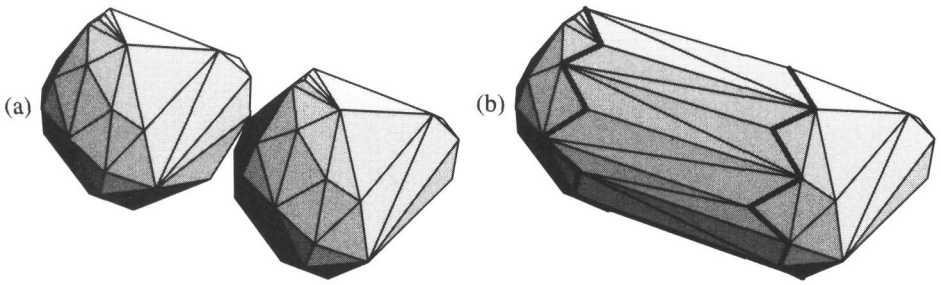


FIGURE 4.5 (a) Polytopes prior to merge. In this example, A and B are congruent, although that will not be true in general. (b) conv $\{A \cup B\}$. The dark edges show the "shadow boundary": the boundary of the newly added faces.

the previous chapter that although several of the two-dimensional algorithms extend (with complications) to three dimensions, the only one to achieve optimal $O(n \log n)$ time is the divide-and-conquer algorithm of Preparata & Hong (1977).[12] This algorithm is both theoretically important and quite beautiful. It is, however, rather difficult to implement, and it is not used as frequently in practice as other asymptotically slower algorithms, such as the incremental algorithm (Section 4.2.4), or those that "only" guarantee expected $O(n \log n)$ performance (Section 4.5). In this section I will describe the algorithm at a level one step above implementation details. Greater detail may be found in Preparata & Shamos (1985), Edelsbrunner (1987), and Day (1990).

The paradigm is the same as in two dimensions: Sort the points by their x coordinate, divide into two sets, recursively construct the hull of each half, and merge. The merge must be accomplished in $O(n)$ time to achieve the desired $O(n \log n)$ bound. All the work is in the merge, and we concentrate solely on this.

Let A and B be the two hulls to be merged. The hull of $A \cup B$ will add a single "band" of faces with the topology of a cylinder without endcaps. See Figure 4.5(b). The number of these faces will be linear in the size of the two polytopes: Each face uses at least one edge of either A or B, so the number of faces is no more than the total number of edges. Thus it is feasible to perform the merge in linear time, as long as each face can be added in constant time (on average).

Let π be a plane that supports A and B from below, touching A at the vertex a and B at the vertex b. To make the exposition simpler, assume that a and b are the only points of contact. Then π contains the line L determined by ab. Now "crease" the plane along L and rotate half of it about L until it bumps into one of the two polytopes. See Figure 4.6. A crucial observation is that if it first bumps into point c on polytope A (say), then ac must be an edge of A. In other words, the first point c hit by π must be a neighbor of either a or b. This limits the vertices that need to be examined to determine the next to be bumped. We highlight this important fact as a lemma, but we do not prove it.

Lemma 4.2.1. *When plane π is rotated about the line through ab as described above, the first point c to be hit is a vertex adjacent to either a or b.*

[12] Preparata & Shamos (1985) contains important corrections to the original paper.

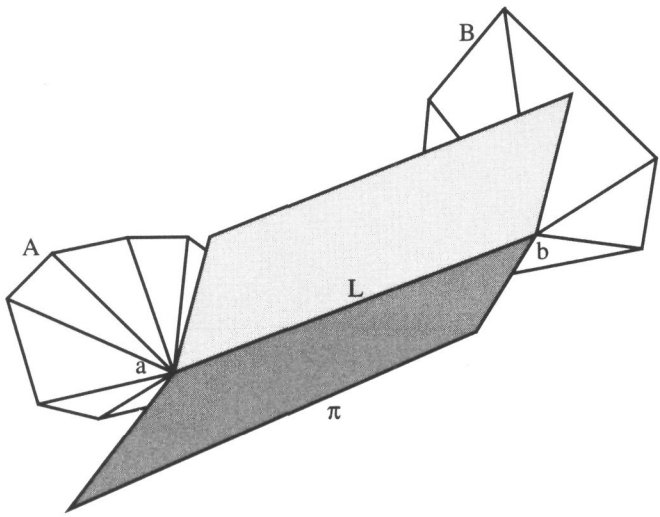


FIGURE 4.6 Plane π is creased along L and bent toward the polytopes A and B (only the faces incident to a and b are shown).

Once π hits c, one triangular face of the merging band has been found: (a, b, c). Now the procedure is repeated, but this time around the line through cb (if $c \in A$). The wrapping stops when it closes upon itself.

Thus what needs to be shown is that the point c can be found in constant time on average, so that the cost of merging is linear over the entire band.

Let's first examine a naive search of all neighbors of a and b. We can easily define the "angle" of any candidate c as the angle that π must be turned around ab from its initial position to hit c. Let α be the vertex adjacent to a with the smallest angle; α is the "A-winner." Let β be the vertex adjacent to b with the smallest angle, the "B-winner." The ultimate winner c is either α or β, whichever has the smaller angle. Clearly the winner c can be found in time proportional to the number of neighbors of a and b.

Two difficulties arise immediately. First, finding one winner might require examining $\Omega(n)$ candidate vertices, because a or b might h ve many neighbors. And to completely wrap A and B with a band of faces, $\Omega(n)$ wir er computations might be required, leading to a quadratic merge time. Although we cannot circumvent the fact that finding a single winner might cost $\Omega(n)$, this is not as damaging as might first appear, because we only need to achieve constant time per winner *on average*, amortized over all winner computations for one merge step. We will see that indeed this can be done.

The second difficulty is that if α is the winner, the work just done to find β might be wasted. Imagine a situation where the candidate on A wins many times in a row, so that b remains fixed. Suppose further that b has many neighbors. Then if we discard the search that obtains the loser β each time, and repeat it for each A winner, again we will be led to quadratic merge time.

Fortunately this repeated search can be avoided because of the following monotonicity property. Let α_i and β_i be the A- and B-winners respectively at the ith iteration of the wrapping.

Table 4.3. Wedge.

Index	(x, y, z)
0	(−20, 10, 5)
1	(−20, 20, 5)
2	(−5, 10, 5)
3	(−5, 20, 5)
4	(−5, 10, 8)
5	(−5, 20, 8)

Lemma 4.2.2. *If α_i is the winner, then the B-winner at the next iteration, β_{i+1}, is counterclockwise of β_i around b.*

Of course a symmetric statement holds with the roles of A and B reversed.

This means that each loop either results in the ultimate winner, in which case its work will not have to be repeated, or it advances around the pivoting vertex, an advance that will not have to be retracted and explored again. Therefore if we "charge" the work to the examined edges, each edge will be charged at most twice (once from each endpoint). Thus the wrapping can be accomplished in linear time.

Discarding Hidden Faces

After wrapping around A and B with a cylinder of faces, it only remains to discard the faces hidden by the wrapped band to complete the merge. Unfortunately the wrapping process does not immediately tell us which faces of A are visible from some point of B, and vice versa; it is just these faces that should be deleted. But the wrapping does discover all the "shadow boundary" edges: those edges of A and B touched by one of the wrapped faces, shown dark in Figure 4.5(b). (If all of B were a light source, the shadow boundary on A marks the division between light and dark; and symmetrically the shadow boundary on B separates light from dark when A is luminous.) Intuitively one could imagine "snipping" along these edges in the data structure and detaching the hidden caps of A and B.

I implemented just this procedure, and it would occasionally fail for reasons that were mysterious to me. It was not until Edelsbrunner (1987, p. 175) examined the algorithm closely that the flaw became evident: Contrary to intuition, the shadow boundary edges on A do not necessarily form a simple cycle on A (and similarly for B)! This is illustrated in Figure 4.7. Figure 4.7(a) shows two polytopes prior to merging: A is a flat wedge, extending 10 units in the y dimension; B is a tall box, 6 units in the y dimension. The coordinates of their vertices are displayed in Tables 4.3 and 4.4.

The hull conv $\{A \cup B\}$ is shown in Figure 4.7(b). The vertices of A on the shadow boundary occur in the order (0, 2, 4, 0, 1, 3, 5, 1) (drawn dark in the figure), forming

Table 4.4. Block.

Index	(x, y, z)
6	(10, 18, 20)
7	(20, 18, 20)
8	(20, 12, 20)
9	(10, 12, 20)
10	(10, 18, −10)
11	(20, 18, −10)
12	(20, 12, −10)
13	(10, 12, −10)

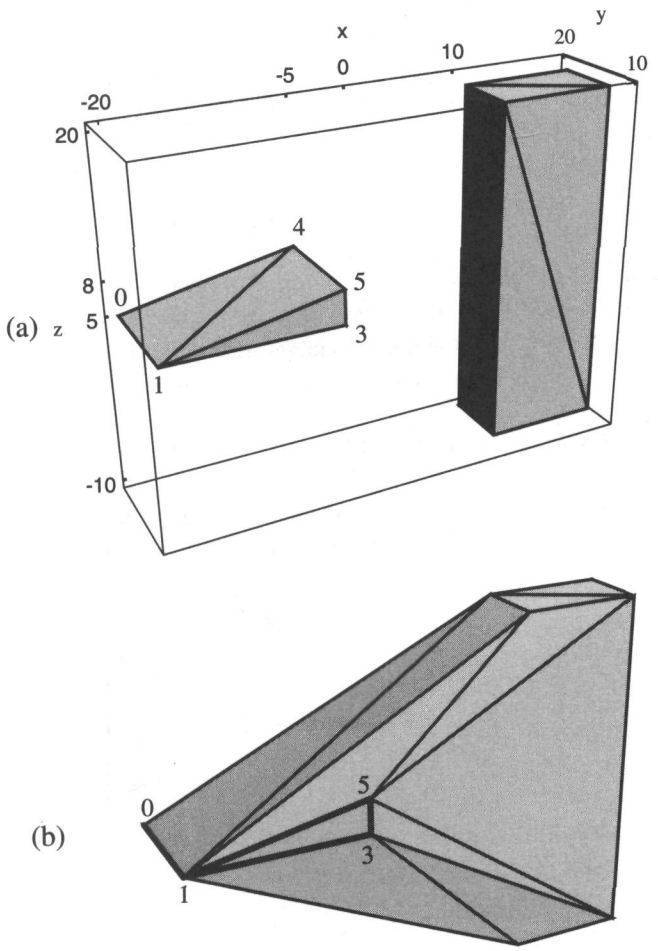


FIGURE 4.7 (a) Wedge and block prior to merging; (b) hull of wedge and block.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

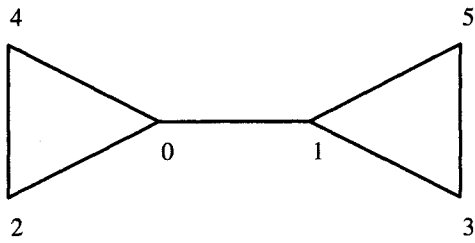


FIGURE 4.8 Topology of shadow boundary edges for Figure 4.7(b).

the topological "barbell" shown in Figure 4.8. Note that this sequence touches p_0 and p_1 twice, and so is not simple.

Despite this unexpected complication, the hidden faces form a connected cap (Exercise 4.2.3[3]) and can be found by a search from the shadow boundary, for example by depth-first search.

My discussion has been somewhat meandering, but I hope it conveys something of both the delicacy and the beauty of the algorithm. Given the complexity of the task of constructing the three-dimensional hull, I find it delightfully surprising that an $O(n \log n)$ algorithm exists.

4.2.3. Exercises

1. *Winning angle.* Detail the computation of the A-winner. Assume you know a and b, and you have accessible all of a's neighbors on A sorted counterclockwise about a.
2. *Degeneracies.* Discuss some difficulties that might arise for the divide-and-conquer algorithm with points that are not in general position: more than two collinear and/or more than three coplanar.
3. *Deleted faces.* Prove that the faces deleted from A during the merge step form a connected set.
4. *Topology of shadow boundary* (Michael Goodrich). Construct an example of two polytopes A and B such that the shadow boundary on A in conv $\{A \cup B\}$ has the topology shown in Figure 4.9.

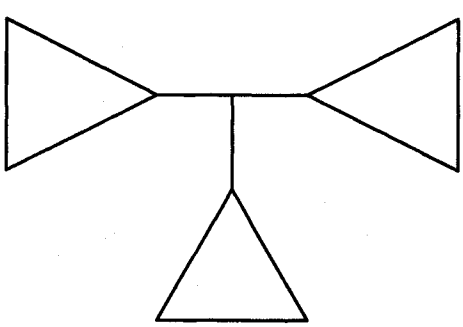


FIGURE 4.9 Can this graph be realized as a shadow boundary?

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

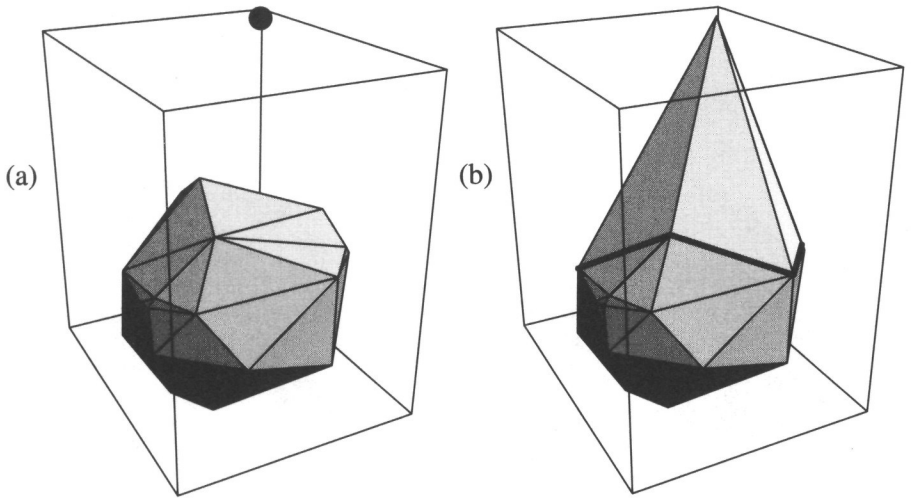


FIGURE 4.10 Viewpoint one: (a) H_{i-1} before adding point in corner; (b) after: H_i.

4.2.4. Incremental Algorithm

The overall structure of the three-dimensional incremental algorithm[13] is identical to that of the two-dimensional version (Section 3.7): At the ith iteration, compute $H_i \leftarrow \text{conv}(H_{i-1} \cup p_i)$. And again the problem of computing the new hull naturally divides into two cases. Let $p = p_i$ and $Q = H_{i-1}$. Decide if $p \in Q$. If so, discard p; if not, compute the cone tangent to Q whose apex is p, and construct the new hull.

The test $p \in Q$ can be made in the same fashion as in two dimensions: p is inside Q iff p is to the positive side of every plane determined by a face of Q. The left-of-triangle test is based on the volume of the determined tetrahedron, just as the left-of-segment test is based on the area of the triangle. If all faces are oriented consistently, the volumes must all have the same sign (positive under our conventions). This test clearly can be accomplished in time proportional to the number of faces of Q, which as we saw in the previous section, is $O(n)$.

When p is outside Q, the problem becomes more difficult, as the hull will be altered. Recall that in the two-dimensional incremental algorithm, the alteration required finding two tangents from p to Q (Figure 3.10). In three dimensions, there are tangent planes rather than tangent lines. These planes bound a *cone* of triangle faces, each of whose apex is p, and whose base is an edge e of Q. An example is shown in Figures 4.10 and 4.11. Figure 4.10 shows H_{i-1} and H_i from one point of view, and Figure 4.11 shows the same example from a different viewpoint. We now discuss how these cone faces can be constructed.

Imagine standing at p and looking toward Q. Assuming for the moment that no faces are viewed edge-on, the interior of each face of Q is either visible or not visible from

[13]This algorithm is sometimes called the "beneath-beyond" method when used to construct the hull in arbitrary dimensions. It seems to have been first discussed in print around 1981, by Seidel (1986) and Kallay (1984) (as cited in Preparata & Shamos (1985)).

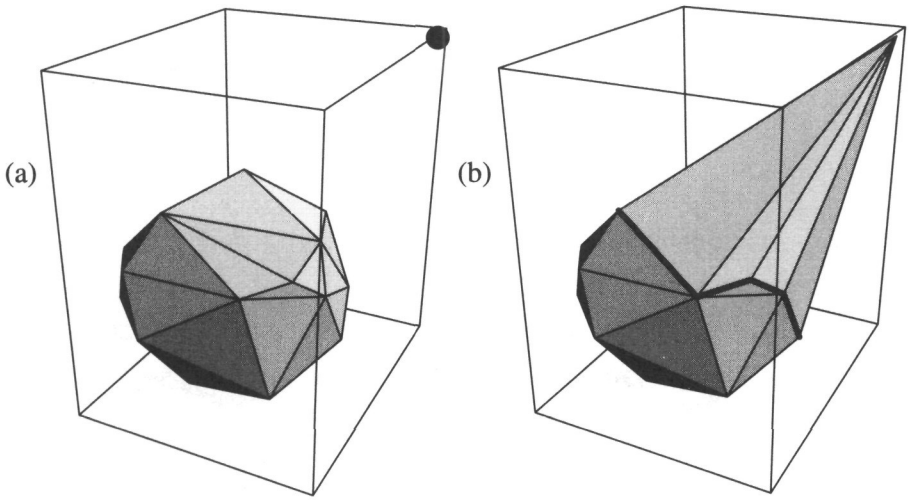


FIGURE 4.11 Viewpoint two: (a) H_{i-1} before adding point in corner; (b) after: H_i.

p. It should be clear that the visible faces are precisely those that are to be discarded in moving from $Q = H_{i-1}$ to H_i. Moreover, the edges on the border of the visible region are precisely those that become the bases of cone faces apexed at p. For suppose e is an edge of Q such that the plane determined by e and p is tangent to Q. Edge e is adjacent to two faces, one of which is visible from p, and one of which is not. Therefore, e is on the border of the visible region. An equivalent way to view this is to think of a light source placed at p. Then the visible region is that portion of Q illuminated, and the border edges are those between the light and dark regions, analogous to the shadow boundary edges in Section 4.2.2.

From this discussion, it is evident that if we can determine which faces of Q are visible from p and which are not, then we will know enough to find the border edges and therefore construct the cone, and we will know which faces to discard. We now need a precise definition of visibility.

Define a face to be *visible* from p iff some point x interior to f is visible from p, that is, px does not intersect Q except at x: $px \cap Q = \{x\}$. Note that under this definition, seeing only an edge of a face does not render the face visible, and faces seen edge-on are also considered invisible. Whether a triangle face (a, b, c) is visible from p can be determined from the signed volume of the tetrahedron (a, b, c, p): It is visible iff the volume is strictly negative. (This sign convention will be discussed in Section 4.3.2 below.)

We can now outline the algorithm based on the visibility calculation; see Algorithm 4.1. Of course many details remain to be explained, but the basics of the algorithm should be clear.

Complexity Analysis

Recall by Theorem 4.1.1 that $F = O(n)$ and $E = O(n)$, where n is the number of vertices of the polytope, so the loops over faces and edges are linear. Since these loops are embedded inside a loop that iterates n times, the total complexity is quadratic: $O(n^2)$.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

```
Algorithm: 3D INCREMENTAL ALGORITHM
Initialize H_3 to tetrahedron (p_0, p_1, p_2, p_3).
    for i = 4, ..., n - 1 do
    for each face f of H_{i-1} do
        Compute volume of tetrahedron determined by f and p_i.
        Mark f visible iff volume < 0.
    if no faces are visible
        then Discard p_i (it is inside H_{i-1}).
        else
            for each border edge e of H_{i-1} do
                Construct cone face determined by e and p_i.
            for each visible face f do
                Delete f.
            Update H_i.
```

Algorithm 4.1 Incremental algorithm, three dimensions.

4.3. IMPLEMENTATION OF INCREMENTAL ALGORITHM

Although the incremental algorithm is conceptually clean, an implementation is nontrivial. Nevertheless, in this section we plunge into a complete description of an implementation, the most complex presented in this book. The details left out of our high-level description above will be included when the code is presented. Those uninterested in the code should skip to the discussion of volume overflow in Section 4.3.5.

4.3.1. Data Structures

It is not obvious how best to represent the surface of a polyhedron, and several sophisticated suggestions have been made in the literature. We will examine a few of these ideas in Section 4.4. Here we will opt for very simple structures, which are limited in their applicability. In particular we will assume the surface of our polytope is triangulated: Every face is a triangle. This will simplify our data structures at the expense of producing an awkward representation for any polytope that is not triangulated, for example, a cube. Also, our data structure will not possess the symmetry that some others have, and it will force some operations to be a bit awkward. Despite these various drawbacks, I think it is the easiest to comprehend.

Structure Definitions. There are three primary data types: vertices, edges, and faces. All the vertices are doubly linked into a circular list, as are all the edges, and all the faces. These lists have the same structure as the list of polygon vertices used in Chapter 1 (Code 1.2). The ordering of the elements in the list has no significance; so these lists should be thought more as sets than as lists. Each element of these lists is a fixed-size structure containing relevant information, including links into the other lists. The vertex structure contains the (integer) coordinates of the vertex. It contains no pointers to its incident edges nor its incident faces. (Note that inclusion of such pointers would not

be straightforward, because a vertex may be incident to an arbitrary number of edges and faces.) The edge structure contains pointers to the two vertices that are endpoints of the edge and pointers to the two adjacent faces. The ordering of both of these pairs is arbitrary; more sophisticated data structures enforce an ordering. The face structure contains pointers to the three vertices forming the corners of the triangular face, as well as pointers to the three edges. Note that it is here that we exploit our assumption that all faces are triangles. The basic fields of the three structures are shown in Code 4.1. The structures will need to contain other miscellaneous fields, which will be discussed shortly.

```
struct tVertexStructure {
  int     v[3];
  int     vnum;
  tVertex next, prev;
};

struct tEdgeStructure {
  tFace   adjface[2];
  tVertex endpts[2];
  tEdge   next, prev;
};

struct tFaceStructure {
  tEdge   edge[3];
  tVertex vertex[3];
  tFace   next, prev;
};
```

Code 4.1 Three primary `structs`.

Each of the three primary structures has three associated type names, beginning with `t` as per our convention. The vertex structure is `tVertexStructure`; this name is used only in the declarations. The type `struct tVertexStructure` is given the name `tsVertex`; this name is used only when allocating storage, as an argument to `sizeof`. Finally, the type used throughout the code is `tVertex`, a pointer to an element in the vertex list. The edge and face structures have similar associated names. These names are established with `typedefs` preceding the structure declarations; see Code 4.2.

Example of Data Structures. We will illustrate the convex hull code with a running example, constructing the hull of eight points comprising the corners of a cube. One of the polytopes created enroute to the final cube has five vertices, and we use this to illustrate the data structures. Call the polytope P_5.

The vertex list contains all the input points; not all are referenced by the edge and face lists. The cube has edge length 10 and is in the positive orthant[14] of the coordinate

[14] An *orthant* is the intersection of three mutually orthogonal halfspaces, the natural generalization of "quadrant" to three dimensions.

Table 4.5. Vertex list.

Vertex	Coordinates
v_0	(0, 0, 0)
v_1	(0, 10, 0)
v_2	(10, 10, 0)
v_3	(10, 0, 0)
v_4	(0, 0, 10)
v_5	(0, 10, 10)
v_6	(10, 10, 10)
v_7	(10, 0, 10)

system. The indices assigned here to the vertices (and edges and faces) play no role in the code, as all references are conducted via pointers.

```
typedef struct tVertexStructure tsVertex;
typedef struct tVertexStructure tsVertex;

typedef struct tEdgeStructure tsEdge;
typedef tsEdge *tEdge;

typedef struct tFaceStructure tsFace;
typedef tsFace *tFace;
```

Code 4.2 Structure typedefs.

The polytope P_5 consists of 9 edges and 6 faces. The three lists in Tables 4.5–4.7 are shown exactly as they are constructed by the code. The indices on the v, e, and f labels indicate the order in which the records were created. Note that the face list contains no f_1 or f_4; both were created and deleted before the illustrated snapshot of the data structures.

A view of the polytope is shown in Figure 4.12. Faces f_2, f_5, and f_6 are visible; f_0 is on the xy-plane. The two "back" faces, f_3 and f_7, are coplanar, forming a square face of the cube, (v_0, v_1, v_5, v_4).

An important property of the face data structure that is maintained at all times is that the vertices in field `vertex` are ordered counterclockwise, so that the right-hand rule yields a vector normal to the face pointing exterior to the polytope. Thus f_6's vertices occur in the order (v_4, v_2, v_5). The same counterclockwise ordering is maintained for the `edge` field. Thus the ordering of f_6's edges is (e_4, e_6, e_8). The code often exploits the counterclockwise ordering of the vertices, but by happenstance never needs to use the counterclockwise ordering of the edges. The edge ordering is maintained by judicious swaps nevertheless, for aesthetics, and for potential uses beyond those presented here.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

Table 4.6. Edge list.

Edge	Endpoints	Adjacent faces
e_0	(v_0, v_2)	(f_2, f_0)
e_1	(v_1, v_0)	(f_3, f_0)
e_2	(v_2, v_1)	(f_5, f_0)
e_3	(v_0, v_4)	(f_2, f_3)
e_4	(v_2, v_4)	(f_2, f_6)
e_5	(v_1, v_4)	(f_3, f_7)
e_6	(v_2, v_5)	(f_5, f_6)
e_7	(v_1, v_5)	(f_5, f_7)
e_8	(v_4, v_5)	(f_6, f_7)

Table 4.7. Face list.

Face	Vertices	Edges
f_0	(v_0, v_1, v_2)	(e_0, e_1, e_2)
f_2	(v_0, v_2, v_4)	(e_0, e_4, e_3)
f_3	(v_1, v_0, v_4)	(e_1, e_3, e_5)
f_5	(v_2, v_1, v_5)	(e_2, e_6, e_6)
f_6	(v_4, v_2, v_5)	(e_4, e_6, e_8)
f_7	(v_1, v_4, v_5)	(e_5, e_8, e_7)

Head Pointers. At all times a global "head" pointer is maintained to each of the three lists, initialized to NULL, just as in Chapter 1 (Code 1.2). See Code 4.3.

```
tVertex vertices  =  NULL;
tEdge edges       =  NULL;
tFace faces       =  NULL;
```

Code 4.3 Head Pointers.

Loops over all vertices, edges, or faces all have the same basic structure, previously shown in Code 1.3. This looping structure assumes that the lists are nonempty, which is indeed the case immediately after the initial polytope is built.

Basic List Processing. Four basic list processing routines are needed for each of the three data structures: allocation of a new element (NEW), freeing an element's memory (FREE), adding a new element to the list (ADD), and deleting an old element (DELETE).

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

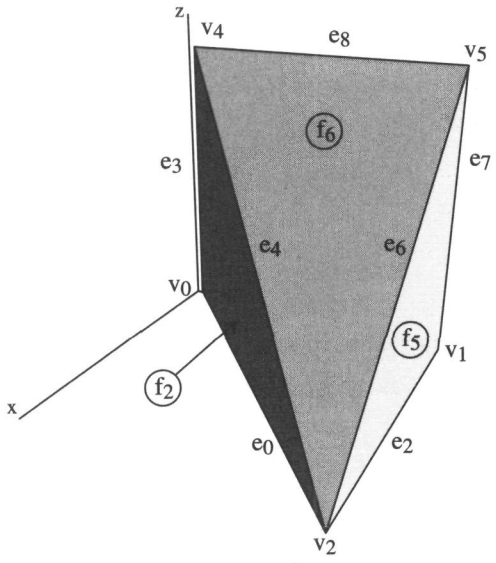


FIGURE 4.12 A view of P_5, with labels.

The first three were used in Chapters 1–3; see Code 1.4. Note that the NEW macro works for all three structures: The type, which will be one of `tsVertex`, `tsEdge`, or `tsFace`, is passed as an argument. The fourth macro DELETE is shown in Code 4.4. DELETE must manage the `head` pointer in case the cell `p` to which it points is the one deleted. In that case, it advances it to `head->next`.

```
#define DELETE( head, p ) if ( head ) {\
   if ( head == head->next ) \
       head = NULL; \
   else if ( p == head ) \
       head = head->next; \
   p->next->prev = p->prev; \
   p->prev->next = p->next; \
   FREE( p ); \
}
```

Code 4.4 DELETE macro.

`structs`: Full Detail. The fields of the basic data structures are augmented by several flags and auxiliary pointers, presented in Code 4.5 and 4.6 with the full structure definitions. The additional fields are all commented, and each will be explained further when first used. Each data structure has a corresponding `MakeNull` routine, which creates a new cell, initializes it, and adds it to the appropriate list. See Code 4.7.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

4.3.2. Example: Cube

In this section the running of the program is illustrated with the example started in the previous section, with input the eight corners of a cube. We will discuss each section of the code as it becomes relevant.

```
/* Define vertex indices. */
#define X  0
#define Y  1
#define Z  2

/* Define Boolean type. */
typedef enum {FALSE, TRUE }  bool;

/* Define flags. */
#define ONHULL        TRUE
#define REMOVED       TRUE
#define VISIBLE       TRUE
#define PROCESSED     TRUE
```

Code 4.5 Defines.

```
struct tVertexStructure {
  int       v[3];
  int       vnum;
  tEdge     duplicate;    /* pointer to incident cone edge (or NULL) */
  bool      onhull;       /* T iff point on hull. */
  bool      mark;         /* T iff point already processed. */
  tVertex   next, prev;
};

struct tEdgeStructure {
  tFace     adjface[2];
  tVertex   endpts[2];
  tFace     newface;      /* pointer to incident cone face. */
  bool      delete;       /* T iff edge should be delete. */
  tEdge     next, prev;
};

struct tFaceStructure {
  tEdge     edge[3];
  tVertex   vertex[3];
  bool      visible;      /* T iff face visible from new point. */
  tFace     next, prev;
};
```

Code 4.6 Full vertex, edge, and face structures.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

```
tVertex MakeNullVertex( void )
{
  tVertex v;

  NEW( v, tsVertex );
  v->duplicate = NULL;
  v->onhull = !ONHULL;
  v->mark = !PROCESSED;
  ADD( vertices, v );

  return v;
}
tEdge  MakeNullEdge( void )
{
  tEdge e;

  NEW( e, tsEdge );
  e->adjface[0] = e->adjface[1] = e->newface = NULL;
  e->endpts[0] = e->endpts[1] = NULL;
  e->delete = !REMOVED;
  ADD( edges, e );
  return e;
}
tFace  MakeNullFace( void )
{
  tFace f;
  int   i;

  NEW( f, tsFace);
  for ( i=0; i < 3; ++i ) {
     f->edge[i] = NULL;
     f->vertex[i] = NULL;
  }
  f->visible = !VISIBLE;
  ADD( faces, f );
  return f;
}
```

Code 4.7 Full vertex, edge, and face structures.

Main. The work is separated into four sections at the top level (Code 4.8): read, create initial polytope, construct the hull, and print.

The code will be discussed in as linear an order as is possible. Code 4.9 shows a list of which routine calls which, with a comment number indicating the order in which they are discussed.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

```
main( int argc, char *argv[] )
{
    /* (Flags etc. not shown here.) */

    ReadVertices();
    DoubleTriangle();
    ConstructHull();
    Print();
}
```

Code 4.8 `main`.

```
/* 1 */    ReadVertices()
                MakeNullVertex()
/* 2 */    DoubleTriangle()
/* 3 */            Collinear()
/* 4 */            MakeFace()
                        MakeNullEdge()
                        MakeNullFace()
                VolumeSign()
/* 5 */    ConstructHull()
/* 6 */            AddOne()
/* 7 */              VolumeSign()
/* 8 */            MakeConeFace()
                        MakeNullEdge()
                        MakeNullFace()
/* 9  */                MakeCcw()
/* 10 */           Cleanup()
/* 12 */                CleanEdges()
/* 11 */                CleanFaces()
/* 13 */                CleanVertices()
                Print()
```

Code 4.9 Who calls whom. Comments indicate the order of discussion.

`ReadVertices`. The input file for the cube example is:

0	0	0
0	10	0
10	10	0
10	0	0
0	0	10
0	10	10
10	10	10
10	0	10

The vertices are labeled $v_0, \ldots, v_7$ in the above order, as displayed previously in Table 4.5. They are read in and formed into the vertex list with the straightforward

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

procedures ReadVertices (Code 4.10) and MakeNullVertex (Code 4.7). The meaning of the various fields of each vertex record will be explained later.

```
void    ReadVertices( void )
{
  tVertex    v;
  int        x, y, z;
  int        vnum = 0;

  while ( scanf ("%d %d %d", &x, &y, &z ) != EOF ) {
    v = MakeNullVertex();
    v->v[X] = x;
    v->v[Y] = y;
    v->v[Z] = z;
    v->vnum = vnum++;
    }
  }
}
```

Code 4.10 ReadVertices.

DoubleTriangle. The next and first substantial step is to create the initial polytope. It is natural to start with a tetrahedron, as in Algorithm 4.1, but I have found it a bit easier to start with a doubly covered triangle (a *d-triangle* henceforth[15]), a polyhedron with three vertices and two faces identical except in the order of their vertices. Although this is not a polyhedron according to the definition in Section 4.1, it has the same local incidence structure as a polyhedron, which suffices for the code's immediate purposes.

Given that the goal is construction of a d-triangle, one might think this task is trivial; but in fact the code is complicated and messy, for several reasons. First, it is not adequate to use simply the first three points in the vertex list, as those points might be collinear. Although we can tolerate the degeneracy of double coverage, a face with zero area will form zero-volume tetrahedra with subsequent points, something we cannot tolerate. So we must first find three noncollinear points. Of course, an assumption of general position would permit us to avoid this unpleasantness, but even the vertices of a cube are not in general position. Second, the data structures need to be constructed to have the appropriate properties. In particular, the counterclockwise ordering of the vertices in each face record must be ensured. This also seems unavoidable. Third, the data structures are somewhat unwieldy. I have no doubt this is avoidable with more sophisticated data structures.

The d-triangle is constructed in three stages:

1. Three noncollinear points (v_0, v_1, v_2) are found.
2. The two triangle faces f_0 and f_1 are created and linked.
3. A fourth point v_3 not coplanar with (v_0, v_1, v_2) is found.

[15]Technically, a "bihedron."

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

```
void DoubleTriangle( void )
{
  tVertex    v0, v1, v2, v3, t;
  tFace      f0, f1 = NULL;
  tEdge      e0, e1, e2, s;
  int        vol;

  /* Find 3 noncollinear points. */
  v0 = vertices;
  while ( Collinear( v0, v0->next, v0->next->next ) )
     if ( ( v0 = v0->next ) == vertices )
        printf("DoubleTriangle: All points are Collinear!\n"),
        exit(0);
  v1 = v0->next;   v2 = v1->next;

  /* Mark the vertices as processed. */
  v0->mark=PROCESSED; v1->mark=PROCESSED; v2->mark=PROCESSED;

  /* Create the two "twin" faces. */
  f0 = MakeFace( v0, v1, v2, f1 );
  f1 = MakeFace( v2, v1, v0, f0 );

  /* Link adjacent face fields. */
  f0->edge[0]->adjface[1] = f1;
  f0->edge[1]->adjface[1] = f1;
  f0->edge[2]->adjface[1] = f1;
  f1->edge[0]->adjface[1] = f0;
  f1->edge[1]->adjface[1] = f0;
  f1->edge[2]->adjface[1] = f0;

  /* Find a fourth, noncoplanar point to form tetrahedron. */
  v3 = v2->next;
  vol = VolumeSign( f0, v3 );
  while ( !vol ) {
     if ( ( v3 = v3->next ) == v0 )
        printf("DoubleTriangle: All points are coplanar!\n"),
        exit(0);
     vol = VolumeSign( f0, v3 );
  }

  /* Insure that v3 will be the first added. */
  vertices = v3;
}
```

Code 4.11 DoubleTriangle.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

We now discuss each stage of `DoubleTriangle` (Code 4.11) in more detail.

1. Three noncollinear points. It suffices to check all triples of three consecutive points in the vertex list. For if not all points are collinear, at least one of these triples must be noncollinear. Collinearity is checked by the same method used in Chapter 1, but now because the points are in three dimensions, we cannot rely solely on the z coordinate of the cross product. The area of the triangle determined by the three points is zero iff each component of the cross product in Equation (1.1) is zero. This is implemented in Code 4.12.

```
bool  Collinear( tVertex a, tVertex b, tVertex c )
{  return
        ( c->v[Z] - a->v[Z] )*( b->v[Y] - a->v[Y] )-
        ( b->v[Z] - a->v[Z] )*( c->v[Y] - a->v[Y] )== 0
      &&( b->v[Z] - a->v[Z] )*( c->v[X] - a->v[X] )-
        ( b->v[X] - a->v[X] )*( c->v[Z] - a->v[Z] )== 0
      &&( b->v[X] - a->v[X] )*( c->v[Y] - a->v[Y] )-
        ( b->v[Y] - a->v[Y] )*( c->v[X] - a->v[X] )== 0 ;
}
```

Code 4.12 `Collinear`.

2. Face construction. Each face is created by an ad hoc routine `MakeFace`, which takes three vertex pointers as input and one face pointer `fold` (Code 4.13). It constructs a face pointing to those three vertices. If the face pointer `fold` is not `NULL`, it uses it to access the edge pointers. This is tricky but not deep: The goal is to fill the face record with three vertex pointers in the order passed, and with three edge pointers, either constructed de novo (for the first triangle) or copied from `fold` (for the second triangle), and finally to link the `adjface` fields of each edge. Note that achieving an initially correct orientation for each face is easy: One face uses (v_0, v_1, v_2) and the other (v_2, v_1, v_0).
3. Fourth noncoplanar point. A noncoplanar point is found by searching for a v_3 such that the volume of the tetrahedron (v_0, v_1, v_2, v_3) is nonzero. Once this is found, the head pointer is repositioned to v_3 so that this will be the first point added. This strategy is used so that we can be assured of reaching a legitimate nonzero-volume polyhedron on the next step. To permit it to grow in a plane would make orientation computations difficult.

When `DoubleTriangle` is run on our cube example, the first three vertices tried are noncollinear: v_0, v_1, v_2 (in fact, no three points of the input are collinear). Faces f_0 and f_1 are then constructed; f_1 will be deleted later in the processing. The first candidate tried for `v3` is v_3 (Table 4.5), which is in fact coplanar with (v_0, v_1, v_2). (We will discuss `VolumeSign` shortly.) The head pointer `vertices` is set to v_4, which is not coplanar, and the stage is set for the first point to be added by the incremental algorithm.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

```
tFace MakeFace( tVertex v0, tVertex v1, tVertex v2, tFace fold )
{
  tFace    f;
  tEdge    e0, e1, e2;

  /* Create edges of the initial triangle. */
  if( !fold ) {
    e0 = MakeNullEdge();
    e1 = MakeNullEdge();
    e2 = MakeNullEdge();
  }
  else { /* Copy from fold, in reverse order. */
    e0 = fold->edge[2];
    e1 = fold->edge[1];
    e2 = fold->edge[0];
  }
  e0->endpts[0] = v0;     e0->endpts[1] = v1;
  e1->endpts[0] = v1;     e1->endpts[1] = v2;
  e2->endpts[0] = v2;     e2->endpts[1] = v0;

  /* Create face for triangle. */
  f = MakeNullFace();
  f->edge[0]   = e0; f->edge[1]   = e1; f->edge[2]   = e2;
  f->vertex[0] = v0; f->vertex[1] = v1; f->vertex[2] = v2;

  /* Link edges to face. */
  e0->adjface[0] = e1->adjface[0] = e2->adjface[0] = f;

  return f;
}
```

Code 4.13 `MakeFace`.

`ConstructHull`. We now come to the heart of the algorithm. It is instructive to note how much "peripheral" code is needed to reach this point. The routine `ConstructHull` (Code 4.14) is called by `main` after the initial polytope is constructed, and it simply adds each point one at a time with the function `AddOne`. One minor feature to note: The entire list of vertices is processed using the field `v->mark` to avoid points already processed. It would not be possible to simply pick up in the vertex list where the initial `DoubleTriangle` procedure left off, because the vertices comprising that d-triangle might be spread out in the list.

After each point is added to the previous hull, an important routine `CleanUp` is called. This deletes superfluous parts of the data structure and prepares for the next iteration. We discuss this in detail below.

`AddOne`. The primary work of the algorithm is accomplished in the procedure `AddOne` (Code 4.15), which adds a single point `p` to the hull, constructing the new cone of faces

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

```
void   ConstructHull( void )
{
  tVertex    v, vnext;
  int        vol;
  v = vertices;
  do {
     vnext = v->next;
     if ( !v->mark ) {
        v->mark = PROCESSED;
        AddOne( v );
        CleanUp();
     }
     v = vnext;
  } while ( v != vertices );
}
```

Code 4.14 `ConstructHull`.

if `p` is exterior. There are two steps to this procedure:

1. Determine which faces of the previously constructed hull are "visible" to p. Recall that face f is visible to p iff p lies strictly in the positive halfspace determined by f, where, as usual, the positive side is determined by the counterclockwise orientation of f. The strictness condition is a crucial subtlety: We do not consider a face visible if p illuminates it edge-on.

   The visibility condition is determined by a volume calculation (discussed below): f is visible from p iff the volume of the tetrahedron determined by f and p is negative.

   If no face is visible from p, then p must lie inside the hull, and it is marked for subsequent deletion.
2. Add a cone of faces to p. The portion of the polytope visible from p forms a connected region on the surface. The interior of this region must be deleted, and the cone connected to its boundary. Each edge of the hull is examined in turn.[16] Those edges whose two adjacent faces are both marked visible are known to be interior to the visible region. They are marked for subsequent deletion (but are not deleted yet). Edges with just one adjacent visible face are known to be on the border of the visible region. These are precisely the ones that form the base of a new triangle face apexed at p. The (considerable) work of constructing this new face is handled by `MakeConeFace`..

   One tricky aspect of this code is that we are looping over all edges at the same time as new edges are being added to the list by `MakeConeFace` (as we will see). Recall that all edges are inserted immediately prior to the head of the list, `edges`. Thus the newly created edges are reprocessed by the loop. But both halves of the

[16] One could imagine representing this region when it is marked, and then only looping over the appropriate edges. See Exercise 4.3.6[6].

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

```
bool  AddOne( tVertex p )
{
  tFace    f;
  tEdge    e, temp;
  bool     vis = FALSE;

  /* Mark faces visible from p. */
  f = faces;
  do {
     if ( VolumeSign( f, p ) < 0 ) {
         f->visible = VISIBLE;
         vis = TRUE;
     }
     f = f->next;
  } while ( f != faces );

  /* If no faces are visible from p, then p is inside the hull. */
  if ( !vis ) {
     p->onhull = !ONHULL;
     return FALSE;
  }

  /* Mark edges in interior of visible region for deletion.
     Erect a newface based on each border edge. */
  e = edges;
  do {
     temp = e->next;
     if ( e->adjface[0]->visible && e->adjface[1]->visible )
        /* e interior: mark for deletion. */
        e->delete = REMOVED;
     else if ( e->adjface[0]->visible || e->adjface[1]->visible )
        /* e border: make a new face. */
        e->newface = MakeConeFace( e, p );
     e = temp;
  } while ( e != edges );
  return TRUE;
}
```

Code 4.15 AddOne.

if-statement fail for these edges, because their adjacent faces are created with their `visible` flag set to `FALSE`.

`AddOne` is written to return `TRUE` or `FALSE` depending on whether the hull is modified or not, but the version of the code shown does not use this Boolean value.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

`VolumeSign`. Recall from Section 1.3.8 that the volume of the tetrahedron whose vertices are (a, b, c, d) is 1/6-th of the determinant

$$\begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}. \tag{4.6}$$

The volume can be computed by a straightforward expansion of this determinant into an algebraic expression. We choose to express the computation differently from the expansion in Equation (1.16) as that used in `VolumeSign` (Code 4.16) is algebraically equivalent but uses fewer multiplications. It derives from translating the tetrahedron so that the p-corner is placed at the origin. The individual coordinates are tediously assigned to many distinct variables to make it easier to transcribe the volume equation without error.

The reader will note that the code does something strange: It takes integer coordinates as input, converts to floating point for the computation, and finally returns an integer in $\{-1, 0, +1\}$. We defer discussion of the reason for this circumlocution to Section 4.3.5.

```
int  VolumeSign( tFace f, tVertex p )
{
  double vol;
  double ax, ay, az, bx, by, bz, cx, cy, cz;

  ax = f->vertex[0]->v[X] - p->v[X];
  ay = f->vertex[0]->v[Y] - p->v[Y];
  az = f->vertex[0]->v[Z] - p->v[Z];
  bx = f->vertex[1]->v[X] - p->v[X];
  by = f->vertex[1]->v[Y] - p->v[Y];
  bz = f->vertex[1]->v[Z] - p->v[Z];
  cx = f->vertex[2]->v[X] - p->v[X];
  cy = f->vertex[2]->v[Y] - p->v[Y];
  cz = f->vertex[2]->v[Z] - p->v[Z];

  vol =   ax * (by*cz - bz*cy)
        + ay * (bz*cx - bx*cz)
        + az * (bx*cy - by*cx);

  /* The volume should be an integer. */
  if      ( vol > 0.5 )     return 1;
  else if ( vol < -0.5 )    return -1;
  else                      return 0;
}
```

Code 4.16 `VolumeSign`.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

Recall that the volume is positive when p is on the negative side of f, with the positive side determined by the right-hand rule. Consider adding the point $v_6 = (10, 10, 10)$ to the polytope P_5 in Figure 4.12. It can see face f_6, whose vertices in counterclockwise order "from the outside" are (v_4, v_2, v_5). The determinant of f_6 and v_6 is.

$$\begin{vmatrix} 0 & 0 & 10 & 1 \\ 10 & 10 & 0 & 1 \\ 0 & 10 & 10 & 1 \\ 10 & 10 & 10 & 1 \end{vmatrix} = -1{,}000 < 0. \tag{4.7}$$

This negative volume is interpreted in `AddOne` as indicating that v_6 can see f_6.

`AddOne`: *Cube Example.* Before discussing the routines employed by `AddOne`, we illustrate its functioning with the cube example. The first three vertices in the vertex list were marked by `DoubleTriangle`: v_0, v_1, v_2. As discussed previously, the head pointer is moved to v_4 because v_3 is coplanar with those first three vertices. The vertices are then added in the order v_4, v_5, v_6, v_7, v_3. Let P_i be the polytope after adding vertex v_i. The polytopes are then produced in the order P_2, P_4, P_5, P_6, P_7, and P_3. They are shown in Figure 4.13(a)–(f).

Let us look at the P_5 to P_6 transition, caused by the addition of v_6. As is evident from Figure 4.13(c) (see also Figure 4.12), v_6 can only see the face: $f_6 = (v_4, v_2, v_5)$. The visibility calculation computes the volume of the tetrahedra formed by v_6 with all the faces of P_6, returning -1 for f_6 (as we just detailed), $+1$ for faces f_0, f_3, and f_7, and 0 for f_2 and f_5. Note that the code does not mark the two coplanar faces f_2 and f_5 as visible, per our definition of visibility.

The second part of `AddOne` finds no edges in the interior of the visible region, since it consists solely of f_6. And it finds that each of f_6's edges, (e_4, e_6, e_8), are border edges, and so constructs three new faces with those as bases: f_8, f_9, and f_{10}. Initially these faces are linked into the field `e->newface`, permitting the old hull data structure to maintain its integrity as the cone is being added. This permits the old structure to be interrogated by `MakeConeFace` while the new is being built. Only after the entire cone is attached are the data structures cleaned up with `CleanUp`.

Coplanarity Revisited. To return to the issue of coplanarity, note that if we considered f_2 visible from v_6, then two of f_2's edges (e_0 and e_3) would become boundary edges, and e_4 would be interior to the visible region. The cone would then be based on four edges rather than three. So our decision to treat coplanar faces as invisible makes the visible region, and therefore the new cone, as small as possible.

There are two reasons for treating `vol==0` faces as invisible:

1. The changes to the data structure are minimized, since, as just explained, the visible region is minimized.
2. Points that fall on a face of the old hull are discarded.

Note that if we treated zero-volume faces as visible, a point in the interior of a face would see that face and thus would end up needlessly fracturing it into new faces.

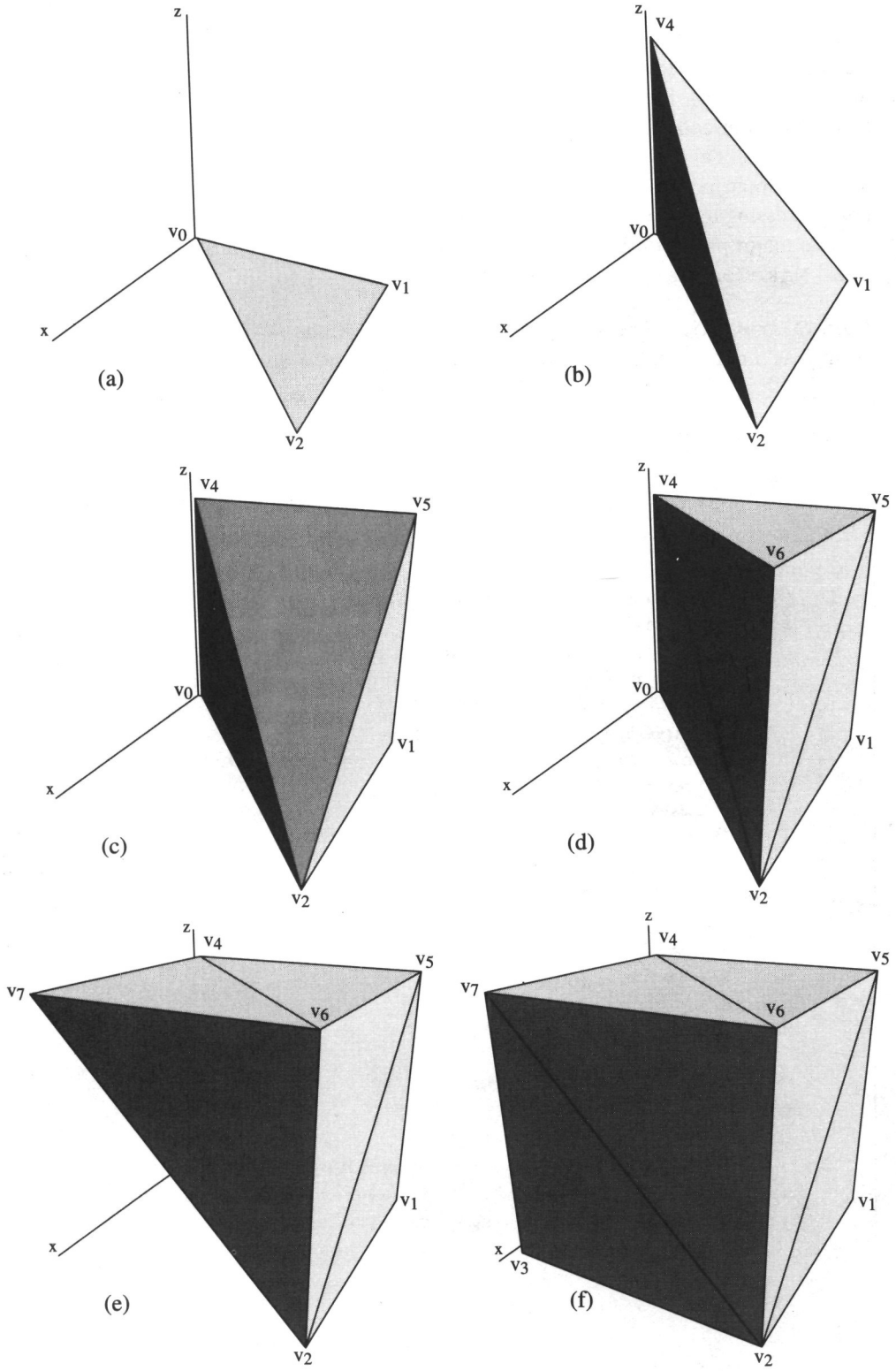


FIGURE 4.13 (a) P_2; (b) P_4; (c) P_5; (d) P_6; (e) P_7; (f) P_3.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

Although this treatment of visibility avoids inserting new points in the interior of old faces, it does not avoid all unnecessary coplanar points: If the interior point is encountered in the construction first, it will never be deleted later. An unfortunate consequence is that, unlike our code for Graham's two-dimensional hull algorithm in Section 3.5, the three-dimensional hull code can produce different outputs for different permutations of the same input points. Invariance with respect to permutations could be achieved by postprocessing to delete unnecessary coplanar points (Exercise 4.3.6[11]).

Two major pieces of the code remain to be explained, both managing the data structures: `MakeConeFace` and `CleanUp`.

`MakeConeFace`. The routine `MakeConeFace` (Code 4.17) takes an edge e and a point p as input and creates a new face spanned by e and p and two new edges between

```
tFace  MakeConeFace( tEdge e, tVertex p )
{
    tEdge    new_edge[2];
    tFace    new_face;
    int      i, j;

    /* Make two new edges (if they don't already exist). */
    for ( i=0; i < 2; ++i )
        /* If the edge exists, copy it into new_edge. */
        if ( !( new_edge[i] = e->endpts[i]->duplicate) ) {
            /* Otherwise (duplicate is NULL), MakeNullEdge. */
            new_edge[i] = MakeNullEdge();
            new_edge[i]->endpts[0] = e->endpts[i];
            new_edge[i]->endpts[1] = p;
            e->endpts[i]->duplicate = new_edge[i];
        }

    /* Make the new face. */
    new_face = MakeNullFace();
    new_face->edge[0] = e;
    new_face->edge[1] = new_edge[0];
    new_face->edge[2] = new_edge[1];
    MakeCcw( new_face, e, p );

    /* Set the adjacent face pointers. */
    for ( i=0; i < 2; ++i )
        for ( j=0; j < 2; ++j )
            /* Only one NULL link should be set to new_face. */
            if ( !new_edge[i]->adjface[j] ) {
                new_edge[i]->adjface[j] = new_face;
                break;
            }
    return new_face;
}
```

Code 4.17 `MakeConeFace`.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

p and the endpoints of e. A pointer to the face is returned, and the created structures are linked together properly.

This is mostly straightforward, but there are two complications. First, the creation of duplicate edges must be avoided. Because we have opted not to structure the border of the visible region, the faces of the cone are constructed in an arbitrary order. Once one face of the cone and its edges have been created, subsequent faces might share two, one, or no edges with previously created faces.

The mechanism we use to detect this is as follows. Each time an edge e_i is created with one end at p and the other at a vertex v on the old hull, a field of v's record, `v->duplicate`, points to e_i. For any vertex not incident to a constructed cone edge, the `duplicate` field is `NULL`. Note that each vertex is incident to at most one cone edge.

For every edge e on the border of the visible region, a new face f is always created. But a new edge e for f is only created if the `duplicate` field of the v-endpoint of e is `NULL`. If one is not `NULL`, then the already-created cone edge pointed to by that field is used to fill the appropriate edge field of f.

The second complication in `MakeConeFace` is the need to arrange the array elements in the `vertex` field of f in counterclockwise order. This is handled by the somewhat tricky routine `MakeCcw`. The basic idea is simple: Assuming that the old hull has its faces oriented properly, make the new faces consistent with the old orientation. In particular, a cone face f can inherit the same orientation as the visible face adjacent to the edge e of the old hull that forms its base. This follows because the new face hides the old and is in a sense a replacement for it; so it naturally assumes the same orientation.

It is here that the most awkward aspect of our choice of data structure makes itself evident. Because e is oriented arbitrarily, we have to figure out how e is directed with respect to the orientation of the visible face, that is, which vertex pointer i of the visible face points to the "base" `[0]`-end of e. We can then anchor decisions from this index i. Although not needed in the code as displayed, we also swap the edges of the new face f to follow the same orientation. Because e was set to be `edge[0]` in `MakeConeFace`, we swap `edge[1]` with `edge[2]` when they run against the orientation of the visible face. See Code 4.18.

`CleanUp`. Just prior to calling `CleanUp` after `AddOne`, the new hull has been constructed: All the faces and edges and one new vertex are linked to each other and to the old structures properly. However, the cone is "glued on" to the old structures via the `newface` fields of edges on the border of the visible region. Moreover, the portion of the old hull that is now inside the cone needs to be deleted. The purpose of `CleanUp` is to "clean up" the three data structures to represent the new hull exactly and only, thereby preparing the structures for the next iteration.

This task is less straightforward than one might expect. We partition the work into three natural groups (Code 4.19): cleaning up the vertex, the edge, and the face lists. But the order in which the three are processed is important. It easiest to decide which faces are to be deleted: those marked `f->visible`. Edges to delete require an inference, made earlier and recorded in `e->delete`: Both adjacent faces are visible. Vertices to delete require the most work: These vertices have no incident edges on the new hull.

We first describe `CleanFaces` (Code 4.20), which is a straight deletion of all faces marked `visible`, meaning visible from the new point just added, and therefore

```
void    MakeCcw( tFace f, tEdge e, tVertex p )
{
  tFace    fv;   /* The visible face adjacent to e */
  int      i;    /* Index of e->endpoint[0] in fv. */
  tEdge    s;    /* Temporary, for swapping */

  if  ( e->adjface[0]->visible )
       fv = e->adjface[0];
  else fv = e->adjface[1];

  /* Set vertex[0] & [1] of f to have the same orientation
     as do the corresponding vertices of fv. */
  for ( i=0; fv->vertex[i] != e->endpts[0]; ++i )
    ;
  /* Orient f the same as fv. */
  if ( fv->vertex[ (i+1) % 3 ] != e->endpts[1] ) {
     f->vertex[0] = e->endpts[1];
     f->vertex[1] = e->endpts[0];
  }
  else {
     f->vertex[0] = e->endpts[0];
     f->vertex[1] = e->endpts[1];
      SWAP( s, f->edge[1], f->edge[2] );
  }

  f->vertex[2] = p;
}
#define SWAP(t,x,y) { t = x; x = y; y = t; }
```

Code 4.18 `MakeCcw`.

```
void    CleanUp( void )
{
   CleanEdges();
   CleanFaces();
   CleanVertices();
}
```

Code 4.19 `CleanUp`.

inside the new hull. There is one minor coding feature to note. Normally our loops over all elements of a list start with the head and stop the `do-while` when the head is encountered again. But suppose, for example, that the first two elements A and B of the `faces` list are both `visible`, and so should be deleted. Starting with `f = faces`, the element `f` = A is deleted, `f` is set to B, and the `DELETE` macro revises `faces` to point

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

to B also. Now if we used the standard loop termination `while( f != faces )`, it would appear that we are finished when in fact we are not.

This problem is skirted by repeatedly deleting the head of the list (if appropriate) and only starting the general loop when we are assured that reencountering the head of the list really does indicate proper loop termination. The same strategy is used for deletion in `CleanEdges` and `CleanVertices`.

```
void    CleanFaces( void )
{
   tFace  f;     /* Primary pointer into face list. */
   tFace  t;     /* Temporary pointer, for deleting. */

  while ( faces && faces->visible ) {
     f = faces;
     DELETE( faces, f );
  }
  f = faces->next;
  do {
     if ( f->visible ) {
        t = f;
        f = f->next;
        DELETE( faces, t );
     }
     else f = f->next;
  } while ( f != faces );
}
```

Code 4.20 `CleanFaces`.

Recall that it is the border edges of the visible region to which the newly added cone is attached. For each of these border edges, `CleanEdges` (Code 4.21) copies `newface` into the appropriate `adjface` field. The reason that `CleanEdges` is called prior to `CleanFaces` is that we need to access the `visible` field of the adjacent faces to decide which to overwrite. So the old faces must be around to properly integrate the new.

Second, `CleanEdges` deletes all edges that were previously marked for deletion (by the routine `AddOne`).

The vertices to delete are not flagged by any routine invoked earlier. But we have called `CleanEdges` first so that we can infer that a vertex is strictly in the interior of the visible region if it has no incident edges: Those interior edges have all been deleted by now. Hence in `CleanVertices` (Code 4.22) we run through the edge list, marking each vertex that is an endpoint as on the hull in the `v->onhull` field. And then a vertex loop deletes all those points already processed but not on the hull. Finally, the various flags in the vertex record are reset.

This completes the description of the code. As should be evident, there is a significant gap between the relatively straightforward algorithm and the reality of an actual

implementation. We continue discussing a few more "real" implementation issues in the next three subsections.

```
void    CleanEdges( void )
{
   tEdge e;   /* Primary index into edge list. */
   tEdge t;   /* Temporary edge pointer. */

   /* Integrate the newfaces into the data structure. */
   /* Check every edge. */
   e = edges;
   do {
      if ( e->newface ) {
         if ( e->adjface[0]->visible )
            e->adjface[0] = e->newface;
         else e->adjface[1] = e->newface;
         e->newface = NULL;
      }
      e = e->next;
   } while ( e != edges );

   /* Delete any edges marked for deletion. */
   while ( edges && edges->delete ) {
      e = edges;
      DELETE( edges, e );
   }
   e = edges->next;
   do {
      if ( e->delete ) {
         t = e;
         e = e->next;
         DELETE( edges, t );
      }
     else e = e->next;
   } while ( e != edges );
}
```

Code 4.21 `CleanEdges`.

4.3.3. Checks

It is not feasible to hope that a program as complex as the foregoing will work correctly upon first implementation. I have spared the reader the debugging printout statements, which are turned on by a command-line flag. Another part of the code not shown is perhaps more worthy of discussion: consistency checks. Again via a command-line flag, we can invoke functions that comb through the data structures checking for various properties known to hold if all is copacetic. The current set of checks used are:

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

1. Face orientations: Check that the endpoints of each edge occur in opposite orders in the two faces adjacent to that edge.
2. Convexity: Check that each face of the hull forms a nonnegative volume with each vertex of the hull.
3. Euler's relations: Check that $F = 2V - 4$ (Equation 4.5)) and $2E = 3V$.

These tests are run after each iteration. They are very slow, but receiving a clean bill of health from these gives some confidence in the program.

```
void    CleanVertices( void )
{
   tEdge      e;
   tVertex    v, t;

   /* Mark all vertices incident to some undeleted edge as on the hull. */
   e = edges;
   do {
      e->endpts[0]->onhull = e->endpts[1]->onhull = ONHULL;
      e = e->next;
   } while (e != edges);

   /* Delete all vertices that have been processed but are not on the hull. */
   while ( vertices && vertices->mark && !vertices->onhull ) {
      v = vertices;
      DELETE( vertices, v );
   }
   v = vertices->next;
   do {
      if ( v->mark && !v->onhull ) {
         t = v;
         v = v->next;
         DELETE( vertices, t )
      }
      else v = v->next;
   } while ( v != vertices );

   /* Reset flags. */
   v = vertices;
   do {
      v->duplicate = NULL;
      v->onhull = !ONHULL;
      v = v->next;
   } while ( v != vertices );
}
```

Code 4.22 `CleanVertices`.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

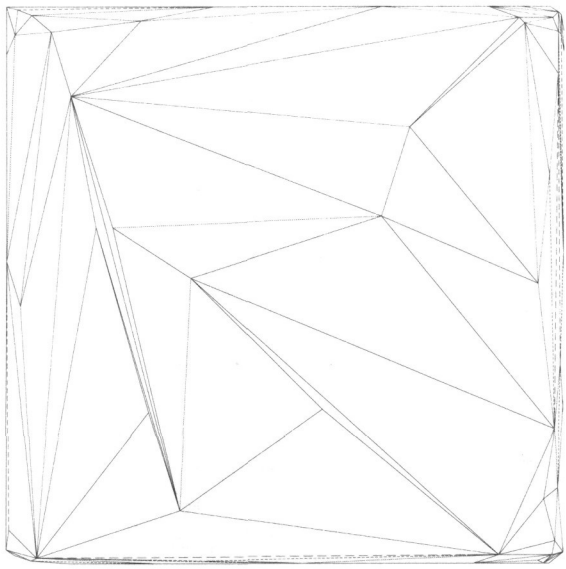


FIGURE 4.14 Hull of 10,000 points in a cube.

4.3.4. Performance

The program is fundamentally quadratic, but its performance varies greatly depending on the data. We present data here for two extreme cases: random points uniformly distributed inside a cube and random points uniformly distributed near the surface of a sphere. Figures 4.14 and 4.15 show examples for $n = 10{,}000$. Most of the points in a cube do not end up on the hull, whereas a large portion of the points near the sphere surface are part of the hull. In Figure 4.14, the hull has 124 vertices, so 9,876

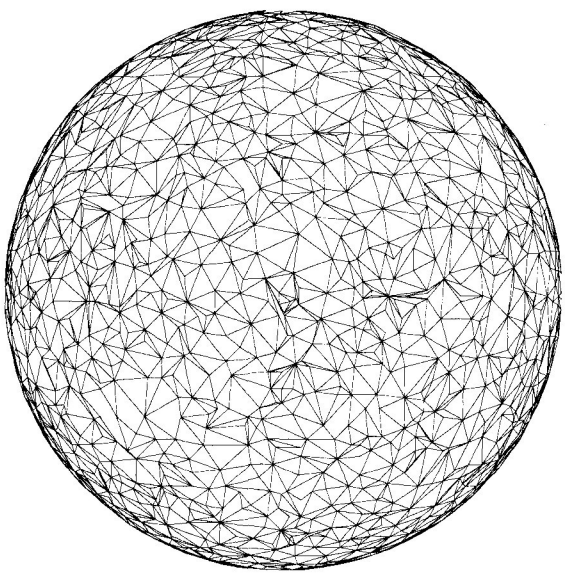


FIGURE 4.15 Hull of 10,000 points near the surface of a sphere.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

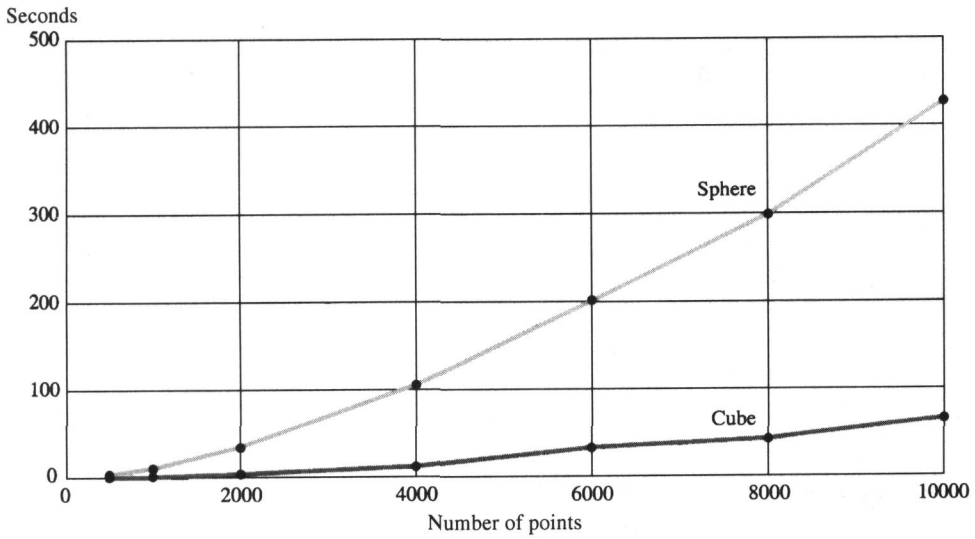


FIGURE 4.16 Runtimes for random points in a cube and near a sphere surface.

points of the 10,000 were interior. The hull in Figure 4.15 has 2,356 vertices; the other 7,644 points were within 2% of the sphere radius of the surface. The sphere points were generated from random vectors of length $r = 100$, whose tips were then rounded to integer coordinates; about three quarters of the lengths of these truncated vectors exceed 99.[17]

Figure 4.16 shows the computation time for the two cases for various n up to 10,000. The times are in seconds on an Silicon Graphics 133 MHz Indy workstation. The superlinear growth is evident in the sphere curve and barely discernible in the cube curve.

4.3.5. Volume Overflow

All the geometry in the code just presented is concentrated in one spot: the volume computation. We have insisted on integer coordinates for the points so that we can be sure this computation is correct. But now we have to face an unpleasant reality: Even computing the volume with integer arithmetic is not guaranteed to give the correct result, due to the possibility of overflow! On most current machines,[18] signed integers use 32 bits and can represent numbers from $-2^{31} = -2147483648$ to $2^{31} - 1 = 2147483647$: about two billion, $\pm 2.1 \times 10^9$. When a computation (e.g., addition or multiplication) exceeds these bounds, the C program proceeds without a complaint (unlike division by zero, integer overflow is not detected and reported back to the C program). Rather the 32 bits are just interpreted as a normal signed integer, which usually means that numbers that exceed $2^{31} - 1$ slightly "wrap around" to negative integers.

[17]The code for generating random points, `sphere.c` and `cube.c`, is included in the software distribution for this book (see the Preface).

[18]Some machines (e.g., Silicon Graphics workstations) have hardware for 64-bit integer computations.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

This does not affect many programs, because the numbers used never become very large. But our critical volume computation multiplies three coordinates together. To make this evident, the fully expanded determinant in Equation 4.6 is:

$$\begin{aligned} &-b_x c_y d_z + a_x c_y d_z + b_y c_x d_z - a_y c_x d_z - a_x b_y d_z \\ &+a_y b_x d_z + b_x c_z d_y - a_x c_z d_y - b_z c_x d_y + a_z c_x d_y \\ &+a_x b_z d_y - a_z b_x d_y - b_y c_z d_x + a_y c_z d_x + b_z c_y d_x \\ &-a_z c_y d_x - a_y b_z d_x + a_z b_y d_x + a_x b_y c_z - a_y b_x c_z \\ &-a_x b_z c_y + a_z b_x c_y + a_y b_z c_x - a_z b_y c_x. \end{aligned} \tag{4.8}$$

The generic term of the computation is abc, where a, b, and c are each one of the three coordinates of various points.

Let us explore the "safe range" of this computation. Because of the many terms, the freedom of compilers to reorganize the computation, and the possible cancellations of even incorrect calculations, this is not an easy question to answer. The smallest example on which I could make the computation err uses coordinates of only ± 512. The idea behind this example is that a regular tetrahedron maximizes its volume among all tetrahedra with fixed maximum edge length. So start with the regular tetrahedron T defined by $(1, 1, 1)$, $(1, -1, -1)$, $(-1, 1, -1)$, and $(-1, -1, 1)$, which is formed by four vertices of a cube centered on the origin. Scaled by a constant c, the volume of this tetrahedron is $16c^3$. With $c = 2^9 = 512$, the volume is $2^{3(9)+4} = 2^{31}$. Thus,

$$\begin{vmatrix} 512 & 512 & 512 & 1 \\ 512 & -512 & -512 & 1 \\ -512 & 512 & -512 & 1 \\ -512 & -512 & 512 & 1 \end{vmatrix} = 2^{31} = 2147483648. \tag{4.9}$$

However, evaluating Equation (4.8) results in the value $-2147483648 = -2^{31}$.[19]

To have the computation in error with such small coordinate values severely limits the usefulness of the code. Fortunately there is a way to extend the safe range of the computation on contemporary machines without much additional effort. It is based on the fact that most machines allocate `doubles` 64 bits, over 50 of which are used for the mantissa (i.e., not the exponent).[20] So curiously, integer calculations can be performed more accurately with floating-point numbers! In particular, the example above that failed in integer arithmetic is correctly computed when the computations use floating-point arithmetic.

Using `doubles`, however, only shifts the precision problem elsewhere. For example, the four points $(3, 0, 0)$, $(0, 3, 0)$, $(0, 0, 3)$, and $(1, 1, 1)$ are coplanar; the fourth is the

[19]The precise value of the incorrect result is machine dependent.

[20]The IEEE 754 standard is followed by many machines; it requires at least 53 bits for the mantissa.

centroid of the triangle determined by the first three. Scaling these points by c produces this determinant for the volume:

$$\begin{vmatrix} 3c & 0 & 0 & 1 \\ 0 & 3c & 0 & 1 \\ 0 & 0 & 3c & 1 \\ c & c & c & 1 \end{vmatrix} = (3c)^3 - 3((3c)^2)c = 0. \tag{4.10}$$

With $c = 200001 \approx 2 \times 10^5$, evaluation of Equation (4.8) with all variables `doubles` results in a volume of 16![21] The reason is that some intermediate terms in the calculation are as large as

$$(3c)^3 = 600003^3 = 216003240016200027 \approx 2.2 \times 10^{17},$$

which cannot be represented exactly in the 54 bits available on my machine, because

$$2^{54} = 18014398509481984 \approx 1.8 \times 10^{16}.$$

Code 4.16 does not compute the volume following Equation (4.8), but rather it uses a more efficient factoring, even more efficient than that presented in Chapter 1 (Equation (1.15)). Here efficiency is measured in terms of the number of multiplications, which are more time consuming on most machines than addition or subtraction. The `VolumeSign` code in fact computes the above determinant correctly, as cancellations prevent any terms from needing more than 54 bits.

But again, this reorganization only pushes off the "crash horizon" a bit more; terms are still composed of three coordinate differences multiplied. With $c = 800000001 \approx 8 \times 10^8$, the computation, in `doubles`, yields a volume of -1.16453×10^{27} rather than 0. Some intermediate computations run as high as $(10^9)^3 = 10^{27}$, which exceeds the 10^{16} that can be precisely represented with a 54-bit mantissa. I do not know the exact safe range of Code 4.16, but coordinate values to about 10^6 should give exact results on most machines (Exercise 4.3.6[12]).

One final point about the `VolumeSign` code needs to be made: It returns only the sign of the volume, not the volume itself. This is all that is needed for the visibility tests;[22] more importantly, converting a correct `double` volume to an `int` for return might cause the result to be garbled by the type conversion.

There is no easy solution to the fundamental problem faced here, an instance of what has become known as *robust computation*. Here are several coping strategies:

1. Report arithmetic overflows. C++ permits defining a class of numbers so that overflow will be reported. Other languages also report overflows. This does not extend the range of the code, but at least the user will know when it fails.

[21] This is again machine dependent; in this case, the number was calculated on a Sun Sparcstation.
[22] Exercise 4.7[7] requires the volume itself.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

2. Use higher precision arithmetic. Machines are now offering 64-bit integer computations, which extend the range of the volume computation to more comfortable levels.
3. Use bignums. Some languages, such as LISP and Mathematica, use arbitrary precision arithmetic, often called "bignums." The problem disappears in these languages, although they are often not the most convenient to mesh with other applications. Recently a number of arbitrary-precision expression packages have become available (Yap 1997), some specifically targeted toward geometric computations. The LEDA library is perhaps the most ambitious and widely used (Mehlhorn & Näher 1995).
4. Incorporate special determinant code. The critical need for accurate determinant evaluations has led to considerable research on this topic. An example of a recent achievement is a method of Clarkson (1992) that permits the sign of a determinant to be evaluated with just a few more bits than are used for the coordinates. The idea is to focus on getting the sign right, while making no attempt to find the exact value of the determinant (in our case, the volume). This permits avoiding the coordinate multiplications that forced our computation to need roughly three times as many bits as the coordinates.[23]

All of the issues faced with the volume computation occur in the area computation used in Chapter 1, except in more muted form because coordinates are only squared rather than cubed. Nevertheless it make sense to use an `AreaSign` function paralleling the `VolumeSign` function just discussed, and for the very same reasons. Consequently, the function shown in Code 4.23 is used throughout the code distributed with this book wherever only the sign of the area is needed (e.g., this would not suffice for the centroid computation in Exercise 1.6.8[5]). Note how integers are forced to `doubles` so that the multiplication has more bits available to it. We'll return to this point in Section 7.2.

```
int    AreaSign( tPointi a, tPointi b, tPointi c )
{
    double area2;

    area2=( b[0] - a[0] ) * (double)( c[1] - a[1] ) -
          ( c[0] - a[0] ) * (double)( b[1] - a[1] );

    /* The area should be an integer. */
    if       ( area2 > 0.5 )     return 1;
    else if  ( area2 < -0.5 )    return -1;
    else                         return 0;
}
```

Code 4.23 `AreaSign`.

[23] See Bronnimann & Yvinec (1997) for further details and Shewchuk (1996) and Avnaim, Boissonnat, Devillers, Preparata & Yvinec (1997) for similar results.

4.3.6. Exercises

1. *Explore* `chull.c` [programming]. Learn how to use `chull.c` and related routines. There are three main programs: `chull`, `sphere`, and `cube`. `sphere n` outputs n random points near the surface of a sphere. `cube n` outputs n random points inside a cube. `chull` reads points from standard input and outputs their convex hull. The output of `sphere` or `cube` may be piped directly into `chull`: `sphere 100 | chull`. See the lead comment for details of input and output formatting conventions and other relevant information. Although `chull` produces Postscript output, it can be modified easily for other graphics displays.
2. *Measure time complexity* [programming]. Measure the time complexity of `chull` by timing its execution on random data produced by `sphere` and `cube`. You may use the Unix function `time`; see `man time`. Make sure you don't time the point generation routines – only time `chull`. Compare the times on your machine with those shown in Figure 4.16.
3. *Profile* [programming]. Analyze where `chull` is spending most of its time with the Unix "profiling" tools. Compile with a `-p` flag and then run the utility `prof`. See the manual pages.
4. *Speed up* `chull` [programming]. David Dobkin sped up my code by a factor of five in some cases with various improvements. Suggest some improvements and implement them.
5. *Distributed volume computation* [programming]. If the volume computation is viewed as area times height, some savings can be achieved by computing the area normal a for each face f, and then calculating the height of the tetrahedron by dotting a vector from the face to p with a (where p is the point being added to the hull). Implement this change and see how much it speeds up the code.
6. *Visibility region.* Prove that the visibility region (the region of Q visible from p) is connected (compare Exercise 4.2.3[3]). Prove that the boundary edges of the visibility region form a simple cycle (in contrast to the situation in Figure 4.7). Suggest code improvements based on this property.
7. *Criticize data structures.* Point out as many weaknesses of the data structures that you can think of. In each case, suggest alternatives.
8. *Consistency checks.* Think of a way the data structure could be incorrect that would not be detected by the consistency checks discussed in Section 4.3.3. Design a check that would catch this.
9. *Faces with many vertices.* Design a data structure that allows faces to have an arbitrary number of vertices.
10. *Distinct points.* Does the code work when not all input points are distinct?
11. *Deleting coplanar points* [programming]. Postprocess the hull data structure to delete unnecessary coplanar points.
12. *Volume range* [open]. For a machine that allocates L bits to its floating-point mantissas, determine an integer m such that if all vertex coordinates are within the range $[-m, +m]$, then the result of `VolumeSign` is correct.
13. *Volume and* `doubles` [programming]. Find an example for which the double computation of `VolumeSign` is incorrect on your machine, and which uses coordinates whose absolute value is as small as possible.
14. *Break the code* [programming]. Find an example set of (noncoplanar) points for which the output of `chull.c` is incorrect, but where all volume computations are correct. Notify the author.

4.4. POLYHEDRAL BOUNDARY REPRESENTATIONS

Representing the boundaries of polyhedra and more general objects has developed into an important subspecialty within computer graphics, geometric modeling, and computational geometry. In this section I will sketch three representations more sophisticated than that used in Section 4.3.1. In particular, these representations do not require faces to be triangles. This immediately raises the issue of how to represent faces: Can fixed-length records be used, or must we resort to variable-length lists?

Our goal in this section is merely to indicate a few issues; no attempt will be made at comprehensive coverage.

4.4.1. Winged-Edge Data Structure

One of the first representations developed, and still popular, is Baumgart's *winged-edge* representation (Baumgart 1975). The focus of this data structure is the edge. Each vertex points to an arbitrary one of its incident edges, and each face points to an arbitrary one of its bounding edges. An edge record for e consists of eight pointers: to the two endpoints of e, v_0 and v_1; to the two faces adjacent to e, f_0 and f_1, left and right respectively of v_0v_1; and to four edges (the "wings" of e): e_0^- and e_0^+, edges incident to v_0, clockwise and counterclockwise of e respectively; and e_1^- and e_1^+, edges incident to v_1. See Figure 4.17. Note that all three structures are constant size, a useful feature.

As an example of the use of the data structure, the edges bounding a face f may be found by retrieving the sole edge e stored in f's record, and then following the e^+ edges around f until e is again encountered. However, because e is oriented arbitrarily, it is necessary to check if f is left or right of e to decide whether the e_1^+ or e_0^+ edge should be followed.

4.4.2. Twin-Edge Data Structure

Data structures in which the orientation of an edge is arbitrary force extra effort to determine its local orientation for certain operations. We saw this with the code `MakeCcw`

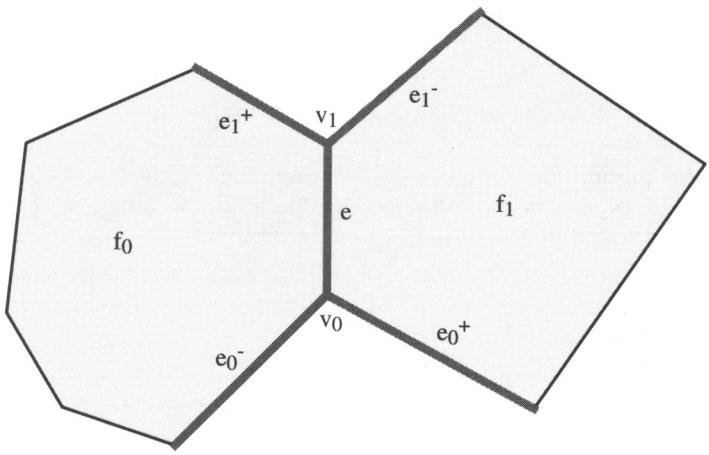


FIGURE 4.17 The winged-edge data structure.

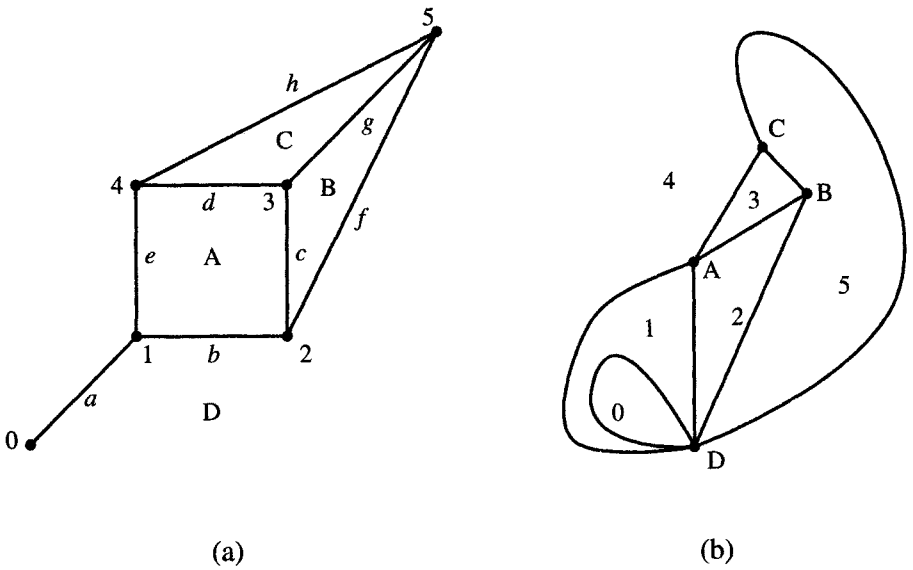
https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

(Code 4.18), and it resurfaced in the winged-edge structure above. A clean solution to this is to represent each edge as two oppositely directed "half" edges, sometimes called "twin edges." Each face points to an arbitrary one of its bounding half edges, which are linked into a circular list. Each vertex points to an arbitary incident half edge. Each half edge points to the unique face it bounds, to the next and previous edges around the face boundary, and to its twin edge, the other half shared with the adjacent face. Given f_0 and one of its bounding half edges e, the adjacent face f_1 is found via the face pointer of twin(e). The small increase in space and update complexity paid by representing each edge twice is often recouped in simpler code for some functions. For example, traversing the edges of a face is trivial with this data structure.

4.4.3. Quad-Edge Data Structure

Guibas and Stolfi invented an alluring data structure they call the *quad-edge* structure (Guibas & Stolfi 1985), which although more complex in the abstract, in fact simplifies many operations and algorithms. It has the advantage of being extremely general, representing any subdivision of 2-manifolds (Section 4.1.1) permitting distinctions between the two sides of a surface, allowing the two endpoints of an edge to be the same vertex, permitting dangling edges, etc.

Each edge record is part of four circular lists: for the two endpoints, and for the two adjacent faces. Thus it contains four pointers. Additional information may be included (an above/below bit, geometric information, etc.) depending upon the application. An example is shown in Figures 4.18 and 4.19. Figure 4.18(a) shows a plane graph. Note that it is not a polyhedral graph (one derivable from a polyhedron) but is rather more general. There are three interior faces, A, B, and C, with D the exterior face. The eight edges are labeled $a, \ldots, h$, and the six vertices $0, \ldots, 5$. Figure 4.19 shows the corresponding quad-edge structure, with each edge record represented by a cross, the



(a) (b)

FIGURE 4.18 (a) A plane graph to be represented; (b) its dual graph.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

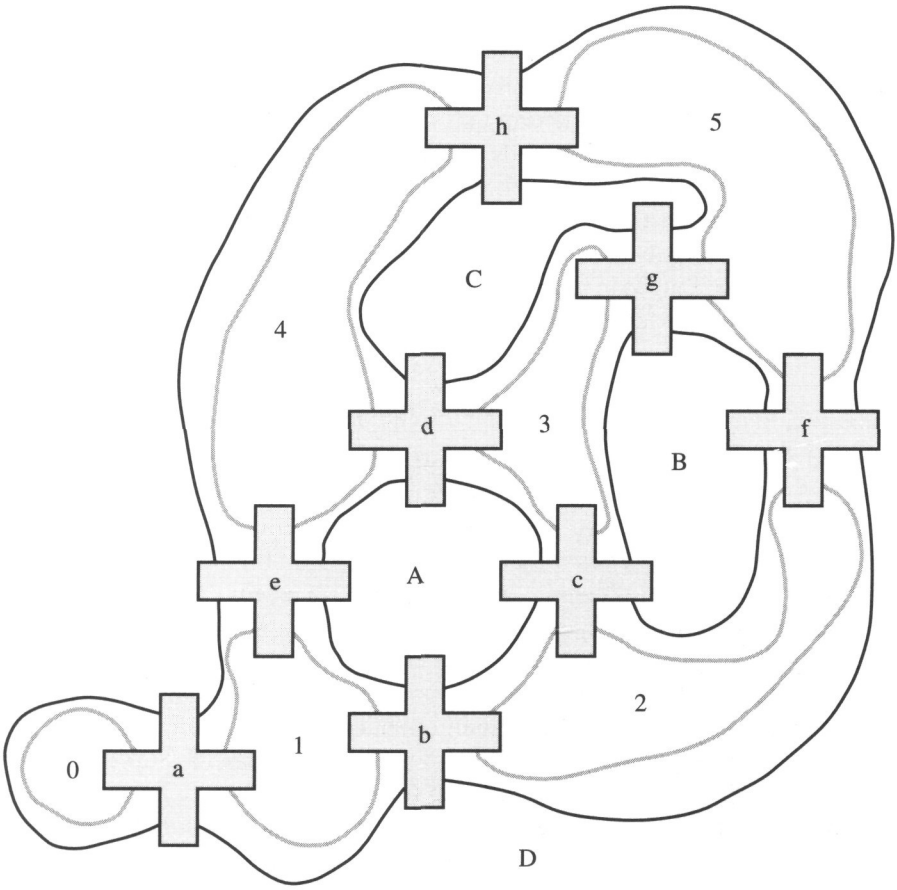


FIGURE 4.19 The quad-edge data structure for the graph in Figure 4.18. Dark cycles represent faces, and light cycles vertices.

four arms corresponding to the four pointers. The face cycles are drawn with dark lines; the vertex cycles are drawn with light lines. For example, face A is the ring of edges (b, c, d, e), and vertex 3 is the ring (c, g, d). Note that the dangling edge a is modeled in a pleasingly consistent way, appearing twice on the cycle for the exterior face D.

As with the winged-edge data structure, vertices and faces have minimal representations: Each is assigned to an arbitrary edge on their ring. The true representation of a vertex or face is this ring; the edge pointer just gives access to the ring.

One of the most beautiful aspects of this structure is that it encodes the dual subdivision automatically. We discussed triangulation duals in Chapter 1 (Section 1.2.3). The *dual* of a general plane graph G assigns a node to each face and an arc for each edge between adjacent faces. The "exterior face" is also assigned a node, and it connects to every face with an exterior boundary edge. This has the consequence that every vertex in G is surrounded by a cycle of face nodes in the dual, as shown in Figure 4.18(b). The dual subdivision is achieved in a quad-edge structure simply by interpreting the light cycles as faces and the dark cycles as vertices in Figure 4.19: No computation is necessary! We will encounter dual graphs again in the next chapter (Section 5.2.2).

4.4.4. Exercises

1. *Winged-edge: edges incident to vertex.* Given a vertex v and a winged-edge data structure, describe how to create a sorted list of all edges incident to v.
2. *Quad-edge: enumeration of edges.* Given one edge and a quad-edge data structure, describe a method of enumerating all edges in the subdivision.
3. *Twin-edge: implementation* [programming]. Modify the `chull.c` data structures (Code 4.1) so that each edge is a half edge, and each half edge points to its twin edge.

4.5. RANDOMIZED INCREMENTAL ALGORITHM

We have described an optimal $O(n \log n)$ algorithm and a practical $O(n^2)$ algorithm. The question naturally arises: Is there a practical $O(n \log n)$ algorithm? This is not merely an academic question. There are applications that require repeated computations of hulls of many points, for example, collision detection in environments consisting of complex polyhedral models. Fortunately, there is a randomized algorithm, due to Clarkson & Shor (1989), that achieves $O(n \log n)$ expected time. Recall from Section 2.4.1 that this means that the algorithm achieves this time complexity with high probability on any input.

We sketch the algorithm here. It is a variant of the incremental algorithm, a variant that on first blush seems like it might be inferior to that algorithm. Recall from Algorithm 4.1 that the faces of H_{i-1} visible from the next point p_i to be added are found by computing the volume of the tetrahedron determined by p_i and each face f. This $O(n)$ check is performed $O(n)$ times, yielding the overall $O(n^2)$ complexity. The Clarkson–Shor algorithm avoids the brute-force search of all faces to determine which are visible. It does this by maintaining in a data structure (called the *conflict graph*) two complementary sets of information: one for each face f of H_{i-1}, which of the yet-to-be-added points $p_i, p_{i+1}, \ldots, p_n$ can see it; and another for each such point p_k, the collection of faces it can see.[24] Although this seems to destroy the simplicity of the incremental algorithm, which only deals with one point at a time, this extra information makes finding the visible faces easy. For when p_i is added, the set of faces it can see (i.e., with which it is "in conflict") is immediately available from the conflict graph.

Of course now the conflict graph must be updated in each iteration. Removing information about deleted faces is easy. The only difficult part is adding information about the new "cone" faces incident to p_i. Let $f = \text{conv}\{e, p_i\}$ be one such new face, based on a polytope edge e on the border between the faces visible and invisible from p_i. The key observation is that if p_k sees f, then it must have been able to see either (or both) of the two faces adjacent to e on H_{i-1} (see Figure 4.10 and 4.11).

Although this gives a hint of how to update the conflict graph at each iteration, it is not at all clear that the overall complexity is improved. It requires a subtle analysis to establish $O(n \log n)$ expected complexity (see, e.g., Mulmuley (1994, Sec. 3.2) or de Berg, et al. (1997, Sec. 11.2)). Fortunately the subtlety of the analysis does not make the algorithm itself any more complicated.

[24] Thus the conflict graph is *bipartite*: All arcs are between face nodes and point nodes.

4.5.1. Exercises

1. *Conflict updates.* Prove the claim above: that if p_k sees $f = \text{conv}\{e, p_i\}$, then it must have been able to see either (or both) of the two faces adjacent to e. Use this to detail an efficient update procedure.
2. *Implementation* [programming]. Modify `chull.c` to maintain a conflict graph. Test it and see if the graph update overhead is compensated by the search reduction for $n \approx 10^5$.

4.6. HIGHER DIMENSIONS

Although we will not cover computational geometry in dimensions beyond three in this book, it would be remiss not even to mention this fertile and important area. This section (together with brief mentions elsewhere) will constitute our nod in this direction.

It is an intellectual challenge to appreciate higher-dimensional geometry, and the reader will only get a taste here. Banchoff (1990) and Rucker (1984) are good sources for more thorough explications.

It is best to approach higher dimensions by analogy with lower dimensions, preferably attaining a running start for your intuition by examining zero-, one-, two-, and three-dimensional examples before leaping into hyperspace.

4.6.1. Coordinates

A point on a number line can be represented by a single number: its value, or location. This can be viewed as a one-dimensional point, since the space in which it is located, the line, is one dimensional. A point in two dimensions can be specified by two coordinates (x, y), and in three dimensions by three coordinates (x, y, z). The leap here is easy: A point in four dimensions requires four coordinates for specification, say (x, y, z, t). If we think of (x, y, z) as space coordinates and t as time, then the four numbers specify an event in both space and time. Besides the use of four dimensions for space–time, there are many other possible spaces of higher dimensions. Just to contrive one example, we could represent the key sartorial characteristics of a person by height, sleeve length, inseam length, and neck and waist circumferences. Then each person could be viewed as a point in a five-dimensional space: *(height, arm, leg, neck, waist).*

Unfortunately the bare consideration of coordinates yields little insight into higher-dimensional geometry. For that we turn to the hypercube.

4.6.2. Hypercube

A zero-dimensional cube is a point. A one-dimensional cube is a line segment. A two-dimensional cube is a square. A three-dimensional cube is a normal cube. Before leaping into four dimensions, let's gather some statistics:

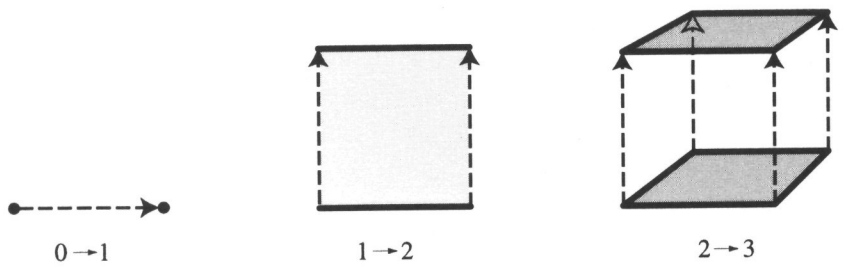


FIGURE 4.20 A cube can be viewed as a square swept through the third dimension.

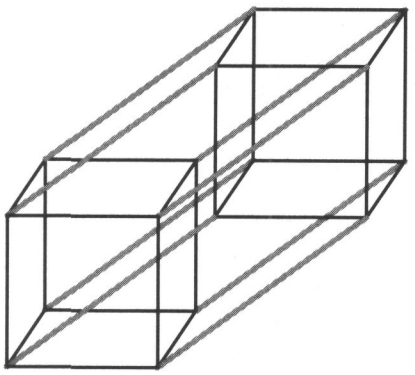


FIGURE 4.21 The edges of a hypercube. The shaded edges represent the sweep in the fourth dimension, connecting two copies of a three-dimensional cube.

Dim d	Name	V_d	E_d
0	point	1	0
1	segment	2	1
2	square	4	4
3	cube	8	12
4	hypercube	16	32
d	d-cube	2^d	$2E_{d-1} + V_{d-1}$

We can view a cube in dimension d as built from two copies of cubes in dimension $d-1$, as follows: Take a zero-dimensional cube (a point) and stretch it in a second dimension, producing a one-dimensional cube (a segment). Slide a segment orthogonal to itself to sweep out a square. Raise a square perpendicular to its plane to sweep out a cube. See Figure 4.20. Now comes the leap. Start with a cube of 8 vertices and 12 edges. Sweep it into a fourth dimension, dragging new edges between the original cube's vertices and the final cube. The new object is a *hypercube*, a four-dimensional cube:[25] Sixteen vertices from the start and stop cubes (8 from each) and 32 edges (12 from each, plus 8 new ones). See Figure 4.21. Note that the number of edges E_d is twice the number in one lower dimension, $2E_{d-1}$, plus the number of vertices V_{d-1}.

[25]Some authors use "hypercube" to indicate a cube in arbitrary dimensions.

https://doi.org/10.1017/CBO9780511804120.005 Published online by Cambridge University Press

Coordinates for the vertices of a generic hypercube can be generated conveniently by the binary digits of the first 2^d integers:

$$\begin{array}{ll} 0 \to (0,0,0,0) & 8 \to (1,0,0,0) \\ 1 \to (0,0,0,1) & 9 \to (1,0,0,1) \\ 2 \to (0,0,1,0) & 10 \to (1,0,1,0) \\ 3 \to (0,0,1,1) & 11 \to (1,0,1,1) \\ 4 \to (0,1,0,0) & 12 \to (1,1,0,0) \\ 5 \to (0,1,0,1) & 13 \to (1,1,0,1) \\ 6 \to (0,1,1,0) & 14 \to (1,1,1,0) \\ 7 \to (0,1,1,1) & 15 \to (1,1,1,1). \end{array} \tag{4.11}$$

The hypercube is the convex hull of these 16 points.

4.6.3. Regular Polytopes

We saw how there are exactly five distinct regular polytopes in three dimensions. In four dimensions there are precisely six regular polytopes. One is the hypercube. But there are surprises: One of the regular polytopes is known as the 600-cell; it is composed of 600 tetrahedral "facets"! It was not until the nineteenth century that the list of four-dimensional regular polytopes was completed, approximately 2,000 years after the three-dimensional polytopes were constructed. In each dimension $d \geq 5$, there are just three regular polytopes, the generalizations of the tetrahedron, the cube, and the octahedron. See Coxeter (1973).

4.6.4. Hull in Higher Dimensions

Much research has been invested in algorithms for constructing the convex hull of a set of points in higher dimensions. This problem arises in a surprisingly wide variety of contexts. Here we touch on three. First, the probability for a certain type of program to branch one way rather than another at a conditional can be modeled as a ratio of volumes of polytopes in a number of dimensions dependent upon the complexity of the code (Cohen & Hickey 1979). Second, the computation of the "antipenumbra" of a convex light source (the volume of space from which some, but not all, of the light source can be seen) can be approached by computing the hull of points in five dimensions (Teller 1992).[26] Third, triangulations of points in three dimensions can be constructed from convex hulls in four dimensions, a beautiful connection we will describe in Section 5.7.2. Such triangulations are needed in a plethora of applications. For example, dynamic stress analysis of three-dimensional objects solves differential equations by discretizing the object into small cells, often tetrahedra. This requires triangulating a collection of points on the surface of the object. Because of this and other connections between three and four dimensions, the convex hull in four dimensions is

[26]The five dimensions arise when the lines containing edges of polyhedra are converted to Plücker coordinates, which represent a directed line with a six-tuple. Removing a scale factor maps these into five dimensions.

in considerable demand, and a number of high-quality software packages have been developed (Amenta 1997).

There is, unfortunately, a fundamental obstruction to obtaining efficient algorithms: The structure of the hull is so complicated that just printing it out sets a stiff lower bound on algorithms. Klee (1980) proved that the hull of n points in d dimensions can have $\Omega(n^{\lfloor d/2 \rfloor})$ facets. Hence in particular, the hull in $d = 4$ dimensions can have quadratic size, and no $O(n \log n)$ algorithm is possible. Nevertheless, algorithms have been developed that are as efficient as possible under the circumstances: worst-case $O(n \log n + n^{\lfloor d/2 \rfloor})$. Moreover, output-size sensitive algorithms are available: One achieves $O(ndF)$ time to produce the F facets (Avis & Fukuda 1992).

4.6.5. Exercises

1. *Simplicies.* A *simplex* is the generalization of a triangle and tetrahedron to arbitrary dimensions. Guess how many vertices, $(d-1)$-dimensional facets, and $(d-2)$-dimensional "ridges" a simplex in d dimensions has. A *ridge* is the higher-dimensional analog of an edge in three dimensions.
2. *Volume of hypersphere.* What is the volume of a unit-radius sphere in four dimensions? Try to generalize to d dimensions. What is the limit of the volume as $d \to \infty$?

4.7. ADDITIONAL EXERCISES

1. *Diameter and width.* This is a generalization of Exercise 3.9.3[3].
   (a) Construct a polytope of n vertices whose diameter (largest distance between any two points) is realized by as many distinct pairs of points as possible.
   (b) Construct a polytope of n vertices that has as many distinct antipodal pairs of points as possible. *Antipodal points* are points that admit parallel planes of support: planes that touch at the points and have the hull to one side.
   (c) Characterize the contacts that may realize the width of a polytope, where the *width* is the smallest distance between parallel planes of support. Each plane of support may touch a face (f), an edge (e) (but not a face), or a vertex (v) (but not an edge). Which of the six possible combinations, (v, v), (v, e), (v, f), (e, e), (e, f), (f, f), can realize the width?
2. *GEB.* The cover of *Gödel, Escher, Bach* (Hofstadter 1979) shows a solid piece of carved wood, which casts the letters "G," "E," and "B" as shadows in three orthogonal directions.
   (a) Can all triples of letters be achieved as shadows of a solid, connected object? Make any reasonable assumptions on the shapes of the letters. If so, supply an argument. If not, exhibit triples that cannot be mutually realized.
   (b) Given three orthogonal polygons, design an algorithm for computing a shape that will have those polygons as shadows (see Figure 4.22), or report that no such shape exists. Keep your algorithm description at a high level, focusing on the method, not the details of implementation. Analyze your algorithm's time complexity as a function of the number of vertices n of the polygons (assume they all have about the same number of vertices).

      Discuss whether your algorithm might be modified to handle nonorthogonal polygons; it may be that it cannot.

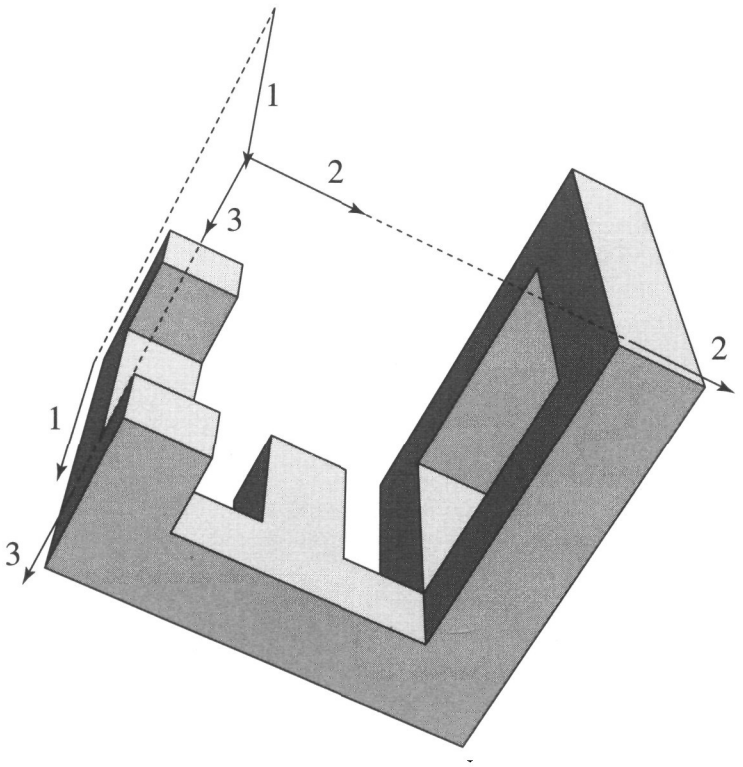


FIGURE 4.22 An orthogonal polyhedron whose shadow in each of the three labeled directions is an orthogonally polygonal letter of the alphabet.

3. *Polytope to tetrahedra.* For a polytope of V, E, and F vertices, edges, and faces, how many tetrahedra T result when it is partitioned into tetrahedra, partitioned in such a way that all edges of the tetrahedra have polytope vertices as endpoints? Is T determined by V, E, and F? If so, provide a formula; if not, provide upper and lower bounds on T.
4. *Stable polytopes.* Design an algorithm to decide if a polytope resting on a given face is stable or will fall over (cf. Exercise 1.6.8[5]).
5. *Shortest path on a cube's surface.* Design a method for finding the shortest path between two points x and y on the surface of a cube, where the path lies on the surface. This is the shortest path for a fly walking between x and y.
6. *Triangle $\cap$ cube.* When a triangle in three dimensions is intersected with the closed region bound by a cube, the result is a polygon P. This is a common computation in graphics, "clipping" a triangle to a cubical viewing space. What is the largest number of vertices P can have for any triangle?
7. *Volume of a polyhedron* [programming]. Compute the volume of a polyhedron in a manner analogous to that used in Chapter 1 to compute the area of a polygon: Choose an arbitrary point p (e.g., the 0th vertex), and compute the signed volume of the tetrahedron formed by p and each triangle t. The sum of the tetrahedra volumes is the volume of the polyhedron.

   Input the polyhedron as follows: Read in V, the number of vertices, and then the coordinates of the vertices. Read in F, the number of (triangular) faces, and then three vertex indices for each face.