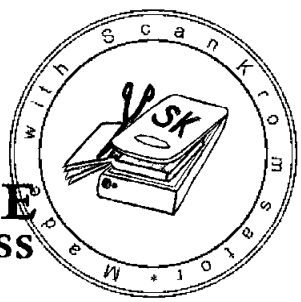
COMPUTATIONAL GEOMETRY IN C

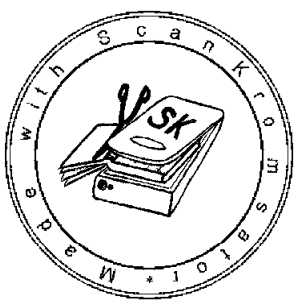
SECOND EDITION

JOSEPH O'ROURKE

CAMBRIDGE UNIVERSITY PRESS



Contents



Preface

Computational geometry broadly construed is the study of algorithms for solving geometric problems on a computer. The emphasis in this text is on the design of such algorithms, with somewhat less attention paid to analysis of performance. I have in several cases carried out the design to the level of working C programs, which are discussed in detail.

There are many brands of geometry, and what has become known as "computational geometry," covered in this book, is primarily discrete and combinatorial geometry. Thus polygons play a much larger role in this book than do regions with curved boundaries. Much of the work on continuous curves and surfaces falls under the rubrics of "geometric modeling" or "solid modeling," a field with its own conferences and texts,[1] distinct from computational geometry. Of course there is substantial overlap, and there is no fundamental reason for the fields to be partitioned this way; indeed they seem to be merging to some extent.

The field of computational geometry is a mere twenty years old as of this writing, if one takes M. I. Shamos's thesis (Shamos 1978) as its inception. Now there are annual conferences, journals, texts, and a thriving community of researchers with common interests.

Topics Covered

I consider the "core" concerns of computational geometry to be polygon partitioning (including triangulation), convex hulls, Voronoi diagrams, arrangements of lines, geometric searching, and motion planning. These topics from the chapters of this book. The field is not so settled that this list can be considered a consensus; other researchers would define the core differently.

Many textbooks include far more material than can be covered in one semester. This is not such a text. I usually cover about 80% of the text with undergraduates in one 40 class-hour semester and all of the text with graduate students. In order to touch on each of the core topics, I find it necessary to oscillate the level of detail, only sketching some algorithms while detailing others. Which ones are sketched and which detailed is a personal choice that I can only justify by my classroom experiences.

Prerequisites

The material in this text should be accessible to students with only minimal preparation. Discrete mathematics, calculus, and linear algebra suffice for mathematics. In fact very

[1] E.g., Hoffmann (1989) and Mortenson (1990).

little calculus or linear algebra is used in the text, and the enterprising student can learn the little needed on the fly. In computer science, a course in programming and exposure to data structures is enough (Computer Science I and II at many schools). I do not presume a course in algorithms, only familiarity with the "big-O" notation. I teach this material to college juniors and seniors, mostly computer science and mathematics majors.

I hasten to add that the book can be fruitfully studied by those who have no programming experience at all, simply by skipping all the implementation sections. Those who know some programming language, but not C, can easily appreciate the implementation discussions even if they cannot read the code. All code is available in Java as well as C, although only C is discussed in the body of the text.

When teaching this material to both computer science and mathematics majors, I offer them a choice of projects that permits those with programming skills to write code and those with theoretical inclinations to avoid programming.

Although written to be accessible to undergraduates, my experience is that the material can form the basis of a challenging graduate course as well. I have tried to mix elementary explanations with references to the latest literature. Footnotes provide technical details and citations. A number of the exercises pose open problems. It is not difficult to supplement the text with research articles drawn from the 300 bibliographic references, effectively upgrading the material to the graduate level.

Implementations

Not all algorithms discussed in the book are provided with implementations. Full code for twelve algorithms is included:[2]

- Area of a polygon.
- Triangulating a polygon.
- Convex hull in two dimensions.
- Convex hull in three dimensions.
- Delaunay triangulation.
- Segment/ray–segment intersection.
- Segment/ray–triangle intersection.
- Point in polygon.
- Point in polyhedron.
- Intersecting convex polygons.
- Minkowski convolution with a convex polygon.
- Multilink robot arm reachability.

Researchers in industry coming to this book for working code for their favorite algorithms may be disappointed: They may seek an algorithm to find the minimum spanning circle for a set of points and find it as an exercise.[3] The presented code should be viewed as samples of geometry programs. I hope I have chosen a representative set of algorithms to implement; much room is left for student projects.

[2]The distribution also includes code to generate random points in a cube (Figure 4.14), random points on a sphere (Figure 4.15), and uniformly distributed points on a sphere (the book cover image).

[3]Exercise 5.5.6[12].

All the C code in the book is available by anonymous ftp from `cs.smith.edu` (131.229.222.23), in the directory /pub/compgeom.[4] I regularly update the files in this directory to correct errors and incorporate improvements. The Java versions of all programs are in the same directory.

Exercises

There are approximately 250 exercises sprinkled throughout the text. They range from easy extensions of the text to quite difficult problems to "open" problems. These latter are an exciting feature of such a fresh field; students can reach the frontier of knowledge so quickly that even undergraduates can hope to solve problems that no one has managed to crack yet. Indeed I have written several papers with undergraduates as a result of their work on homework problems I posed.[5] Not all open problems are necessarily difficult; some are simply awaiting the requisite attention.

Exercises are sporadically marked "[easy]" or "[difficult]," but the lack of these notations should not be read to imply that neither apply. Those marked "[programming]" require programming skills, and those marked "[open]" are unsolved problems as far as I know at this writing. I have tried to credit authors of individual problems where appropriate. Instructors may contact me for a partial solutions manual.

Second Edition Improvements

It is a law of nature that second editions are longer than first editions, and this book is no exception: It is about fifty pages longer, with fifty new exercises, thirty new figures, and eighty additional bibliographic references. All the code from the first edition is significantly improved: All programs now produce Postscript output, all have been translated to Java, many are simpler and/or logically cleaner, most are more robust in the face of degeneracies and numerical error, and most run (sometimes) significantly faster. Both the polygon triangulation code and the Delaunay triangulation code are now $O(n^2)$.

Four new programs have been included: for computing Delaunay triangulation from the three-dimensional convex hull (Section 5.7.4), for intersecting a ray with a triangle in 3-space (Section 7.3), for deciding if a point is inside a polyhedron (Section 7.5), for computing the convolution (Minkowski sum) of a convex polygon with a general polygon (Section 8.4.4), as well as the point generation code that produced the cover image.

New sections are included on partitioning into monotone mountains (Section 2.3), randomized triangulation (Section 2.4.1), the ultimate(?) planar convex hull algorithm (Section 3.8.4), randomized convex hull in three dimensions (Section 4.5), the twin edge data structure (Section 4.4), intersection of a segment and triangle (Section 7.3), the point-in-polyhedron problem (Section 7.5), the Bentley–Ottmann algorithm for intersecting segments (Section 7.7), computing Boolean operations between two polygons (Section 7.8), randomized trapezoidal decomposition for point location (Section 7.11.4), Minkowski convolution computation (Section 8.4.4), and a list of sources for further reading (Chapter 9).

[4]Connect with `ftp cs.smith.edu` and use the name `anonymous`. Or access the files via `http://cs.smith.edu/~orourke`.

[5]The material from one paper is incorporated into Section 7.6.

Other sections are greatly improved, including those on QuickHull (Section 3.4), Graham's algorithm (Section 3.5.5), volume overflow (Section 4.3.5), Delaunay triangulation via the paraboloid transformation (Section 5.7.4), the point-in-polygon problem (Section 7.4), intersecting two segments (Section 7.2), and the implementation of convex polygon intersection (Section 7.6.1).

Acknowledgments

I have received over six hundred e-mail messages from readers of the first edition of this book, and I despair of accurately apportioning credit to their many specific contributions to this edition. I deeply appreciate the suggestions of the following people, many of whom are my professional colleagues, twenty-nine whom are my former students, but most of whom I have met only electronically: Pankaj Agarwal, Kristy Anderson, Bill Baldwin, Michael Baldwin, Pierre Beauchemin, Ed Bolson, Helen Cameron, Joanne Cannon, Roy Chien, Satyan Coorg, Glenn Davis, Adlai DePano, Matthew Diaz, Tamala Dirlam, David Dobkin, Susan Dorward, Scot Drysdale, Herbert Edelsbrunner, John Ellis, William Flis, Steve Fortune, Robert Fraczkiewicz, Reinaldo Garcia, Sharmilli Ghosh, Carole Gitlin, Jacob E. Goodman, Michael Goodrich, Horst Greiner, Suleyman Guleyupoglu, Eric Haines, Daniel Halperin, Eszter Hargittai, Paul Heckbert, Claudio Heckler, Paul Heffernan, Kevin Hemsteter, Christoph Hoffmann, Rob Hoffmann, Chun-Hsiung Huang, Knut Hunstad, Ferran Hurtado, Joan Hutchinson, Andrei Iones, Chris Johnston, Martin Jones, Amy Josefczyk, Martin Kerscher, Ed Knorr, Nick Korneenko, John Kutcher, Eugene Lee, David Lubinsky, Joe Malkevitch, Michelle Maurer, Michael McKenna, Thomas Meier, Walter Meyer, Simon Michael, Jessica Miller, Andy Mirzaian, Joseph Mitchell, Adelene Ng, Seongbin Park, Irena Pashchenko, Octavia Petrovici, Madhav Ponamgi, Ari Rappoport, Jennifer Rippel, Christopher Saunders, Catherine Schevon, Peter Schorn, Vadim Shapiro, Thomas Shermer, Paul Short, Saul Simhon, Steve Skiena, Kenneth Sloan, Stephen Smeulders, Evan Smyth, Sharon Solms, Ted Stern, Ileana Streinu, Vinita Subramanian, J.W.H. Tangelder, Yi Tao, Seth Teller, Godfried Toussaint, Christopher Van Wyk, Gert Vetger, Jim Ward, Susan Weller, Wendy Welsh, Rephael Wenger, Gerry Wiener, Bob Williamson, Stacia Wyman, Min Xu, Dianna Xu, Chee Yap, Amy Yee, Wei Yinong, Lilla Zollei, and the Faculty Advancement in Mathematics 1992 Workshop participants. My apologies for the inevitable omissions.

Lauren Cowles at Cambridge has been the ideal editor. I have received generous support from the National Science Foundation for my research in computational geometry, most recently under grant CCR-9421670.

Joseph O'Rourke
orourke@cs.smith.edu
http://cs.smith.edu/~orourke
Smith College, Massachusetts
December 23, 1997

Note on the Cover:

The cover image shows the convex hull of 5,000 points distributed on a spiral curve on the surface of a sphere. It was generated by running the `spiral.c` and `chull.c` code distributed with this book: `spiral 5000 -r1000 | chull`.

1

Polygon Triangulation

1.1. ART GALLERY THEOREMS

1.1.1. Polygons

Much of computational geometry performs its computations on geometrical objects known as polygons. Polygons are a convenient representation for many real-world objects; convenient both in that an abstract polygon is often an accurate model of real objects and in that it is easily manipulated computationally. Examples of their use include representing shapes of individual letters for automatic character recognition, of an obstacle to be avoided in a robot's environment, or of a piece of a solid object to be displayed on a graphics screen. But polygons can be rather complicated objects, and often a need arises to view them as composed of simpler pieces. This leads to the topic of this and the next chapter: partitioning polygons.

Definition of a Polygon

A *polygon* is the region of a plane bounded by a finite collection of line segments[1] forming a simple closed curve. Pinning down a precise meaning for the phrase "simple closed curve" is unfortunately a bit difficult. A topologist would say that it is the homeomorphic image of a circle,[2] meaning that it is a certain deformation of a circle. We will avoid topology for now and approach a definition in a more pedestrian manner, as follows.

Let $v_0, v_1, v_2, \ldots, v_{n-1}$ be n points in the plane. Here and throughout the book, all index arithmetic will be mod n, implying a cyclic ordering of the points, with v_0 following v_{n-1}, since $(n-1)+1 \equiv n \equiv 0 \pmod{n}$. Let $e_0 = v_0v_1, e_1 = v_1v_2, \ldots, e_i = v_iv_{i+1}, \ldots, e_{n-1} = v_{n-1}v_0$ be n segments connecting the points. Then these segments bound a polygon iff[3]

1. The intersection of each pair of segments adjacent in the cyclic ordering is the single point shared between them: $e_i \cap e_{i+1} = v_{i+1}$, for all $i = 0, \ldots, n-1$.
2. Nonadjacent segments do not intersect: $e_i \cap e_j = \emptyset$, for all $j \neq i+1$.

[1] A *line segment* ab is a closed subset of a line contained between two points a and b, which are called its *endpoints*. The subset is closed in the sense that it includes the endpoints. (Many authors use $\overline{ab}$ to indicate this segment.)

[2] A *circle* is a one-dimensional set of points. We reserve the term *disk* to mean the two-dimensional region bounded by a circle.

[3] "Iff" means "if and only if," a convenient abbreviation popularized by Halmos (1985, p. 403).

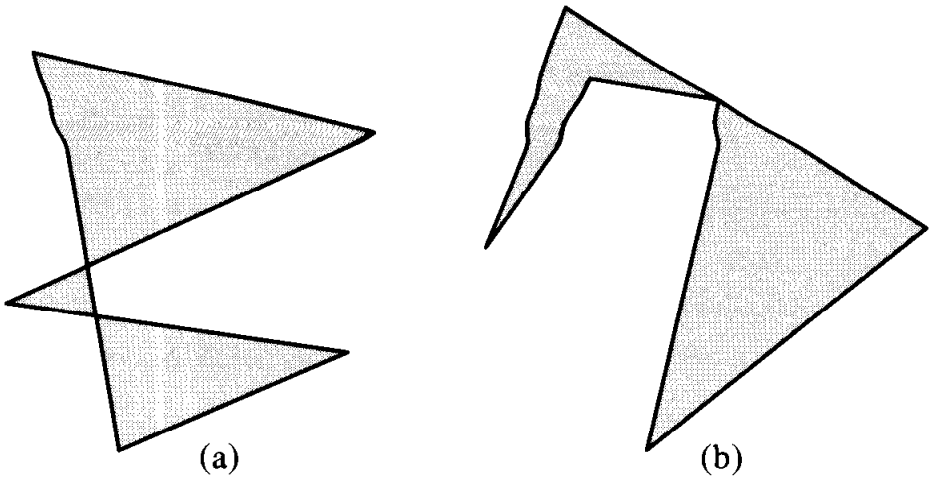


FIGURE 1.1 Nonsimple polygons.

The reason these segments define a *curve* is that they are connected end to end; the reason the curve is *closed* is that they form a cycle; the reason the closed curve is *simple* is that nonadjacent segments do not intersect.

The points v_i are called the *vertices* of the polygon, and the segments e_i are called its *edges*. Note that a polygon of n vertices has n edges.

An important theorem of topology is the Jordan Curve Theorem:

Theorem 1.1.1 (Jordan Curve Theorem). *Every simple closed plane curve divides the plane into two components.*

This strikes most as so obvious as not to require a proof; but in fact a precise proof is quite difficult.[4] We will take it as given. The two parts of the plane are called the *interior* and *exterior* of the curve. The exterior is unbounded, whereas the interior is bounded. This justifies our definition of a polygon as the region bounded by the collection of segments. Note that we define a polygon P as a closed region of the plane. Often a polygon is considered to be just the segments bounding the region, and not the region itself. We will use the notation ∂P to mean the boundary of P; this is notation borrowed from topology.[5] By our definition, $\partial P \subseteq P$.

Figure 1.1 illustrates two nonsimple polygons. For both objects in the figure, the segments satisfy condition (1) above (adjacent segments share a common point), but not condition (2): nonadjacent segments intersect. Such objects are often called polygons, with those polygons satisfying (2) called *simple polygons*. As we will have little use for nonsimple polygons in this book, we will drop the redundant modifier.

We will follow the convention of listing the vertices of a polygon in counterclockwise order, so that if you walked along the boundary visiting the vertices in that order (a *boundary traversal*), the interior of the polygon would be always to your left.

[4] See, e.g., Henle (1979, pp. 100–3). The theorem dates back to 1877.

[5] There is a sense in which the boundary of a region is like a derivative, so it makes sense to use the partial derivative symbol ∂.

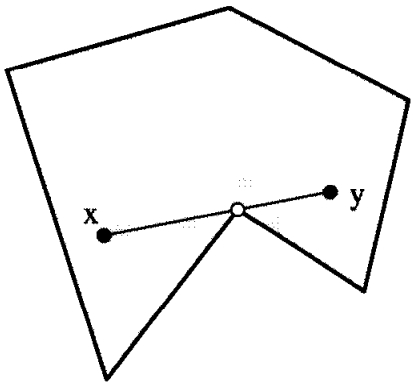


FIGURE 1.2 Grazing contact of line of sight.

1.1.2. The Art Gallery Theorem

Problem Definition

We will study a fascinating problem posed by Klee[6] that will lead us naturally into the issue of triangulation, the most important polygon partitioning. Imagine an art gallery room whose floor plan can be modeled by a polygon of n vertices. Klee asked: How many stationary guards are needed to guard the room? Each guard is considered a fixed point that can see in every direction, that is, has a 2π range of visibility.[7] Of course a guard cannot see through a wall of the room. An equivalent formulation is to ask how many point lights are needed to fully illuminate the room. We will make Klee's problem rigorous before attempting an answer.

Visibility

To make the notion of visibility precise, we say that point x can *see* point y (or y is *visible* to x) iff the closed segment xy is nowhere exterior to the polygon P: $xy \subseteq P$. Note that this definition permits the line-of-sight to have grazing contact with a vertex, as shown in Figure 1.2. An alternative, equally reasonable definition would say that a vertex can block vision; say that x has *clear visibility* to y if $xy \subseteq P$ and $xy \cap \partial P \subseteq \{x, y\}$. We will occasionally use this alternative definition in exercises (Exercises 1.1.4[2] and [3]).

A guard is a point. A set of guards is said to *cover* a polygon if every point in the polygon is visible to some guard. Guards themselves do not block each other's visibility. Note that we could require the guards to see only points of ∂P, for presumably that is where the paintings are! This is an interesting variant, explored in Exercise 1.1.4[1].

Max over Min Formulation

We have now made most of Klee's problem precise, except for the phrase "How many." Succinctly put, the problem is to find the maximum over all polygons of n vertices, of the minimum number of guards needed to cover the polygon. This max-over-min formulation is confusing to novices, but it is used quite frequently in mathematics, so we will take time to explain it carefully.

[6]Posed in 1973, as reported by Honsberger (1976). The material in this section (and more on the topic) may be found in O'Rourke (1987).

[7]We will use radians throughout to represent angles. π radians $= 180°$.

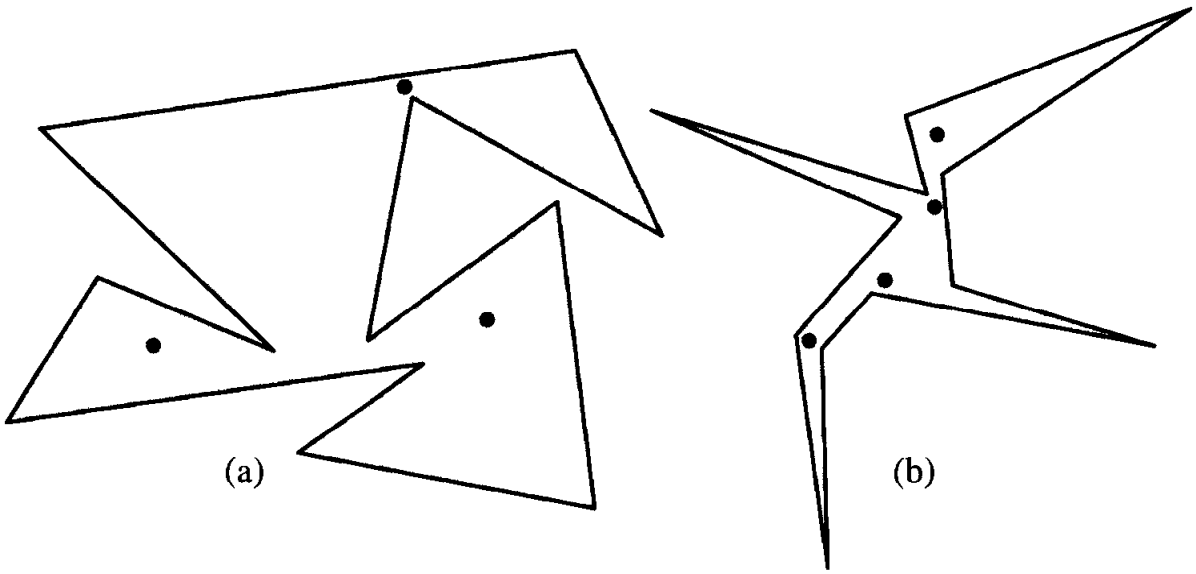


FIGURE 1.3 Two polygons of $n = 12$ vertices: (a) requires 3 guards; (b) requires 4 guards.

For any given fixed polygon, there is some minimum number of guards that are *necessary* for complete coverage. Thus in Figure 1.3(a), it is clear that three guards are needed to cover this polygon of twelve vertices, although there is considerable freedom in the location of the three guards. But is three the most that is ever needed for all possible polygons of twelve vertices? No: the polygon in Figure 1.3(b), also with twelve vertices, requires four guards. What is the largest number of guards that any polygon of twelve vertices needs? We will show eventually that four guards always *suffice* for any polygon of twelve vertices. This is what Klee's question seeks: Express as a function of n, the smallest number of guards that suffice to cover any polygon of n vertices. Sometimes this number of guards is said to be necessary and sufficient for coverage: necessary in that at least that many are needed for *some* polygons, and sufficient in that that many always suffice for *any* polygon.

We formalize the problem before exploring it further. Let $g(P)$ be the smallest number of guards needed to cover polygon P: $g(P) = \min_S |\{S : S \text{ covers } P\}|$, where S is a set of points, and $|S|$ is the cardinality[8] of S. Let P_n be a polygon of n vertices. $G(n)$ is the maximum of $g(P_n)$ over all polygons of n vertices: $G(n) = \max_{P_n} g(P_n)$. Klee's problem is to determine the function $G(n)$. It may not be immediately evident that $G(n)$ is defined for each n: It is at least conceivable that for some polygon, no finite number of guards suffice. Fortunately, $G(n)$ is finite for all n, as we will see. But whether it can be expressed as a simple formula, or must be represented by an infinite table of values, is less clear.

Empirical Exploration

Sufficiency of n. Certainly at least one guard is always necessary. In terms of our notation, this provides a lower bound on $G(n)$: $1 \leq G(n)$. It seems obvious that n guards suffice for any polygon: stationing a guard at every vertex will certainly cover the polygon. This

[8]The *cardinality* of a set is its number of elements.

provides an upper bound: $G(n) \leq n$. But it is not even so clear that n guards suffice. At the least it demands a proof. It turns out to be true, justifying intuition, but this success of intuition is tempered by the fact that the same intuition fails in three dimensions: Guards placed at every vertex of a polyhedron do not necessarily cover the polyhedron! (See Exercise 1.1.4[6].)

There are many art-gallery-like problems, and for most it is easiest to first establish a lower bound on $G(n)$ by finding generic examples showing that a large number of guards are sometimes necessary. When it seems that no amount of ingenuity can increase the number necessary, then it is time to turn to proving that that number is also sufficient. This is how we will proceed.

Necessity for Small n. For small values of n, it is possible to guess the value of $G(n)$ with a little exploration. Clearly every triangle requires just one guard, so $G(3) = 1$.

Quadrilaterals may be divided into two groups: convex quadrilaterals and quadrilaterals with a reflex vertex. Intuitively a polygon is convex if it has no dents. This important concept will be explored in detail in Chapter 3. A vertex is called *reflex*[9] if its internal angle is strictly greater than π; otherwise a vertex is called *convex.*[10] A convex quadrilateral has four convex vertices. A quadrilateral can have at most one reflex vertex, for reasons that will become apparent in Section 1.2. As Figure 1.4(a) makes evident, even quadrilaterals with a reflex vertex can be covered by a single guard placed near that vertex. Thus $G(4) = 1$.

For pentagons the situation is less clear. Certainly a convex pentagon needs just one guard, and a pentagon with one reflex vertex needs only one guard for the same reason as in a quadrilateral. A pentagon can have two reflex vertices. They may be either adjacent or separated by a convex vertex, as in Figures 1.4(c) and (d); in each case one guard suffices. Therefore $G(5) = 1$.

Hexagons may require two guards, as shown in Figure 1.4(e) and (f). A little experimentation can lead to a conviction that no more than two are ever needed, so that $G(6) = 2$.

Necessity of $\lfloor n/3 \rfloor$

At this point the reader might be able to leap to a generalization of Figure 1.4(f) for larger values of n. Figure 1.5 illustrates the design for $n = 12$; note the relation to Figure 1.4(f). This "comb" shape consists of k prongs, with each prong composed of two edges, and adjacent prongs separated by an edge. Associating each prong with the separating edge to its right, and the bottom edge with the rightmost prong, we see that a comb of k prongs has $n = 3k$ edges (and therefore vertices). Because each prong requires its own guard, we establish with this one example that $n/3 \leq G(n)$ for $n = 3k$. This is what I meant earlier by saying that a generic example can be used to establish a lower bound on $G(n)$.

[9] Often this is called *concave*, but the similarity of "concave" and "convex" invites confusion, so I will use "reflex."

[10] Some authors use "convex" to indicate what I'll call *strict convexity*, an interior angle strictly less than π.

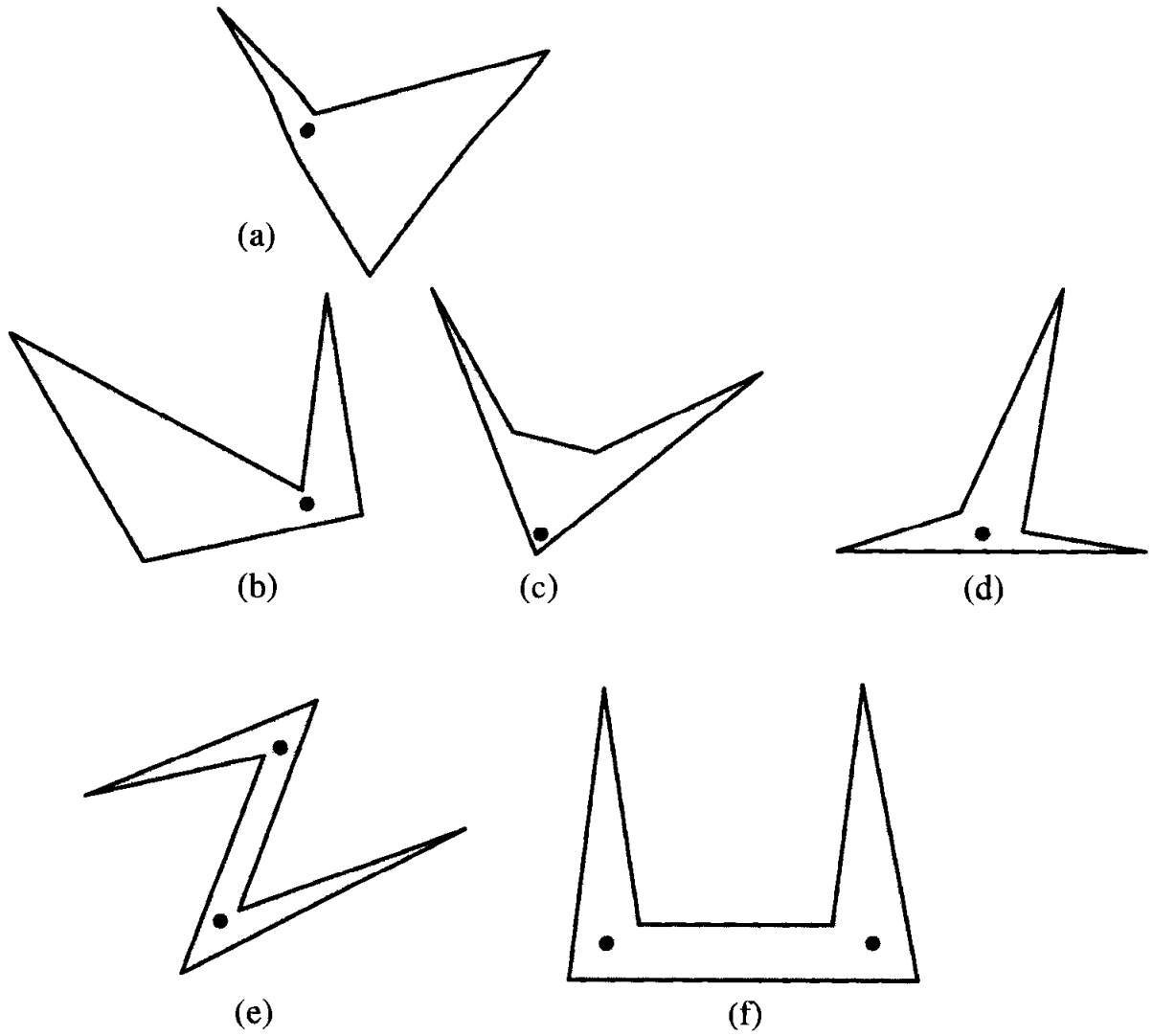


FIGURE 1.4 Polygons of $n = 4, 5, 6$ vertices.

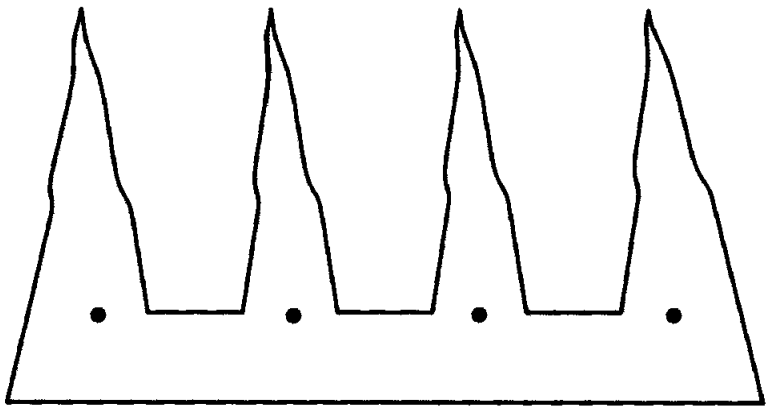


FIGURE 1.5 Chvátal's comb for $n = 12$.

Noticing that $G(3) = G(4) = G(5)$ might lead one to conjecture that $G(n) = \lfloor n/3 \rfloor$,[11] and in fact this conjecture turns out to be true. This is the usual way that such mathematical questions are answered: First the answer is conjectured after empirical exploration,

[11] $\lfloor x \rfloor$ is the *floor* of x: the largest integer less than or equal to x. The floor function has the effect of discarding the fractional portion of a positive real number.

and only then, with a definite goal in mind, is the result proven. We now turn to a proof.

1.1.3. Fisk's Proof of Sufficiency

The first proof that $G(n) = \lfloor n/3 \rfloor$ was due to Chvátal (1975). His proof is by induction: Assuming that $\lfloor n/3 \rfloor$ guards are needed for all $n < N$, he proves the same formula for $n = N$ by carefully removing part of the polygon so that its number of vertices is reduced, applying the induction hypothesis, and then reattaching the removed portion. The proof splinters into a number of cases and is quite delicate.

Three years later Fisk found a very simple proof, occupying just a single journal page (Fisk 1978). We will present Fisk's proof here.

Diagonals and Triangulation

Fisk's proof depends crucially on partitioning a polygon into triangles with diagonals. A *diagonal* of a polygon P is a line segment between two of its vertices a and b that are clearly visible to one another. Recall that this means the intersection of the closed segment ab with ∂P is exactly the set $\{a, b\}$. Another way to say this is that the open segment from a to b does not intersect ∂P; thus a diagonal cannot make grazing contact with the boundary.

Let us call two diagonals *noncrossing* if their intersection is a subset of their endpoints: They share no interior points. If we add as many noncrossing diagonals to a polygon as possible, the interior is partitioned into triangles. Such a partition is called a *triangulation* of a polygon. The diagonals may be added in arbitrary order, as long as they are legal diagonals and noncrossing. In general there are many ways to triangulate a given polygon. Figure 1.6 shows two triangulations of a polygon of $n = 14$ vertices.

We will defer a proof that every polygon can be triangulated to Section 1.2, and for now we just assume the existence of a triangulation.

Three Coloring

To prove sufficiency of $\lfloor n/3 \rfloor$ guards for *any* polygon, the proof must work for an arbitrary polygon. So assume an arbitrary polygon P of n vertices is given. The first step of Fisk's proof is to triangulate P. The second step is to "recall" that the resulting graph may be 3-colored. We need to explain what this graph is, and what 3-coloring means.

Let G be a graph associated with a triangulation, whose arcs are the edges of the polygon and the diagonals of the triangulation, and whose nodes are the vertices of the polygon. This is the graph used by Fisk. A k-*coloring* of a graph is an assignment of k colors to the nodes of the graph, such that no two nodes connected by an arc are assigned the same color. Fisk claims that every triangulation graph may be 3-colored. We will again defer a proof of this claim, but a little experimentation should make it plausible. Three-colorings of the triangulations in Figure 1.6 are shown in Figure 1.7. Starting at, say, the vertex indicated by the arrow, and coloring its triangle arbitrarily with three colors, the remainder of the coloring is completely forced: There are no other free choices. Roughly, the reason this always works is that the forced choices never double back on an earlier choice; and the reason this never happens is that the underlying figure is a polygon (with no holes, by definition).

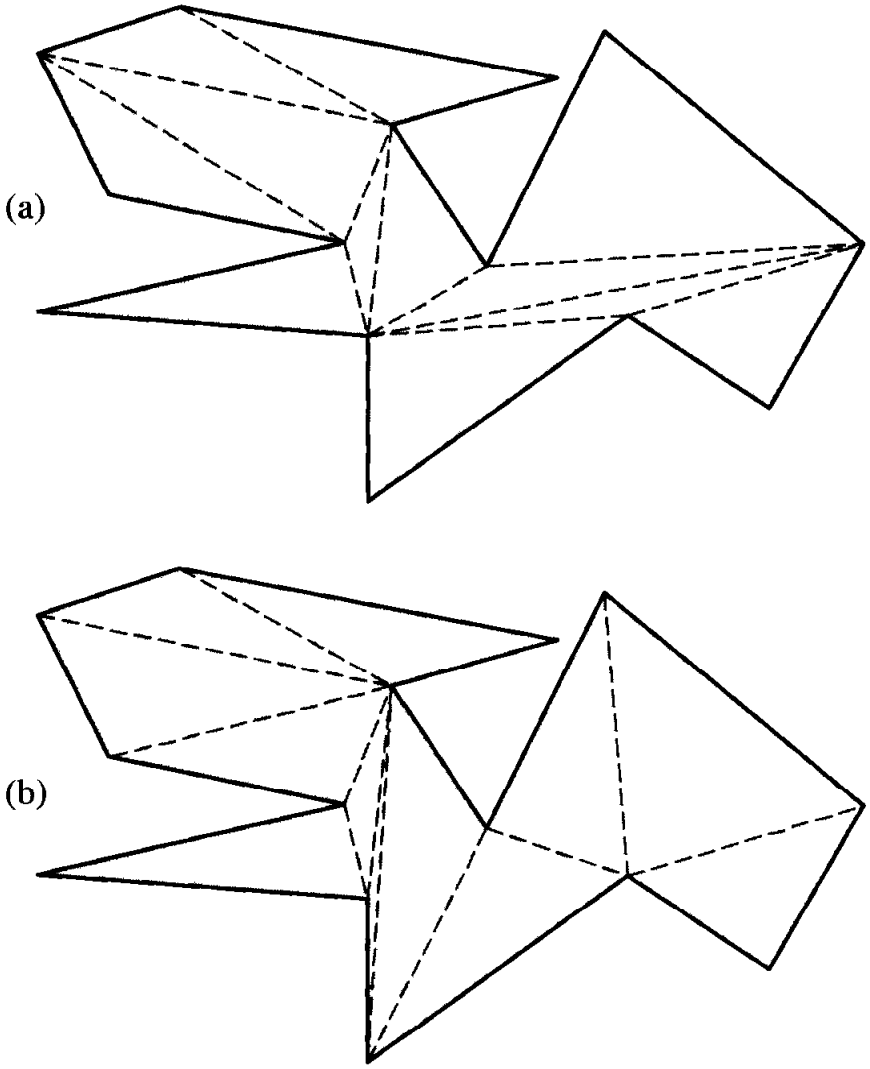


FIGURE 1.6 Two triangulations of a polygon of $n = 14$ vertices.

The third step of Fisk's proof is the observation that placing guards at all the vertices assigned one color guarantees visibility coverage of the polygon. His reasoning is as follows. Let red, green, and blue be the colors used in the 3-coloring. Each triangle must have each of the three colors at its three corners. Thus every triangle has a red node at one corner. Suppose guards are placed at every red node. Then every triangle has a guard in one corner. Clearly a triangle is covered by a guard at one of its corners. Thus every triangle is covered. Finally, the collection of triangles in a triangulation completely covers the polygon. Thus the entire polygon is covered if guards are placed at red nodes. Similarly, the entire polygon is covered if guards are placed at green nodes or at blue nodes.

The fourth and final step of Fisk's proof applies the "pigeon-hole principle": If n objects are placed into k pigeon holes, then at least one hole must contain no more than n/k objects. For if each one of the k holes contained more than n/k objects, the total number of objects would exceed n. In our case, the n objects are the n nodes of the triangulation graph, and the k holes are the 3 colors. The principle says that one color must be used no more than $n/3$ times. Since n is an integer, we can conclude that one color is used no more than $\lfloor n/3 \rfloor$ times. We now have our sufficiency proof: Just place

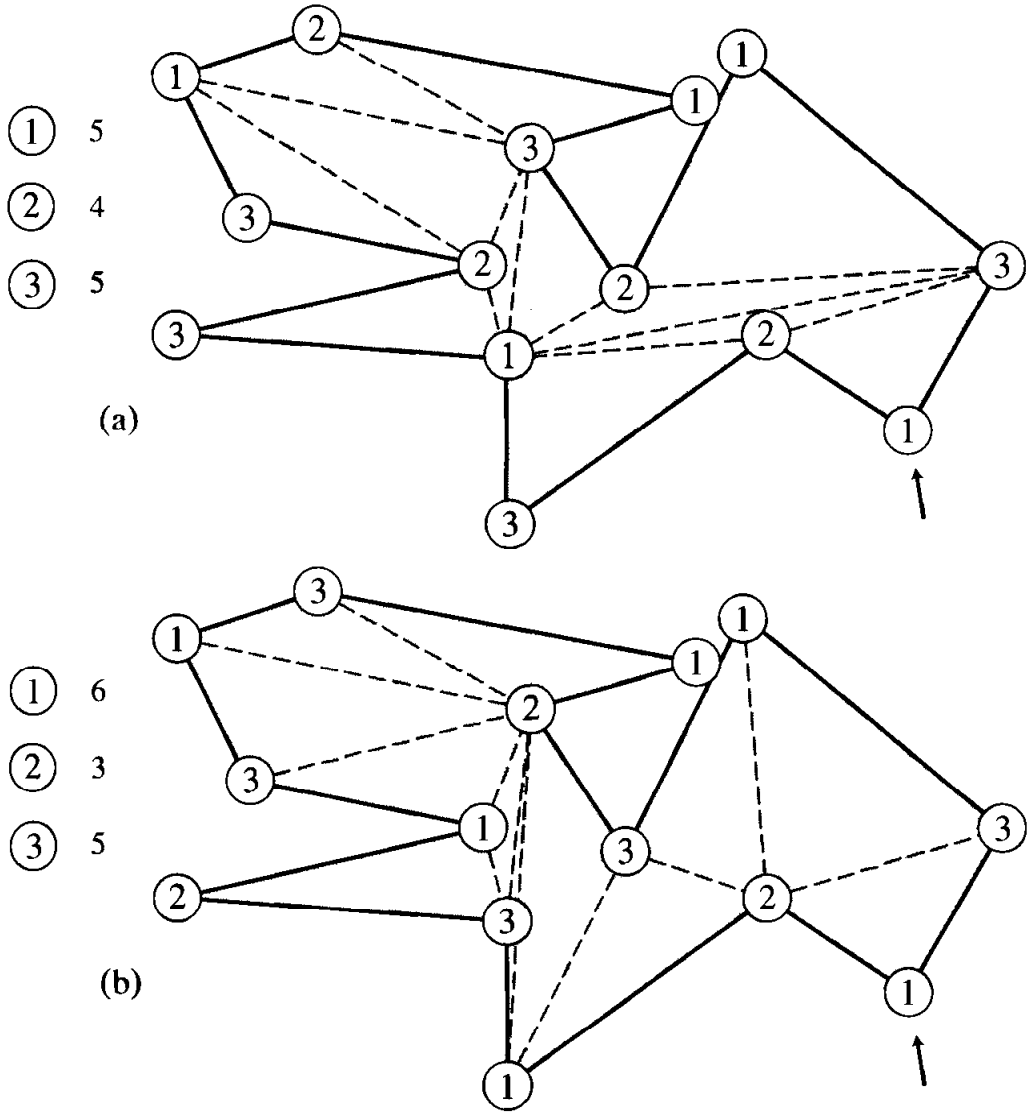


FIGURE 1.7 Two 3-colorings of a polygon of $n = 14$ vertices, based on the triangulations shown in Figure 1.6.

guards at nodes colored with the least-frequently used color in the 3-coloring. We are guaranteed that this will cover the polygon with no more than $G(n) = \lfloor n/3 \rfloor$ colors.

If you don't find this argument beautiful (or at least charming), then you will not enjoy much in this book!

In Figure 1.7, $n = 14$, so $\lfloor n/3 \rfloor = 4$. In (a) of the figure color 2 is used four times; in (b), the same color is used only three times. Note that the 3-coloring argument does not always lead to the most efficient use of guards.

1.1.4. Exercises

1. *Guarding the walls.* Construct a polygon P and a placement of guards such that the guards see every point of ∂P, but there is at least one point interior to P not seen by any guard.
2. *Clear visibility, point guards.* What is the answer to Klee's question for clear visibility (Section 1.1.2)? More specifically, let $G'(n)$ be the smallest number of *point guards* that suffice to clearly see every point in any polygon of n vertices. Point guards are guards who may stand at any point of P; these are distinguished from *vertex guards* who may be stationed only at vertices.

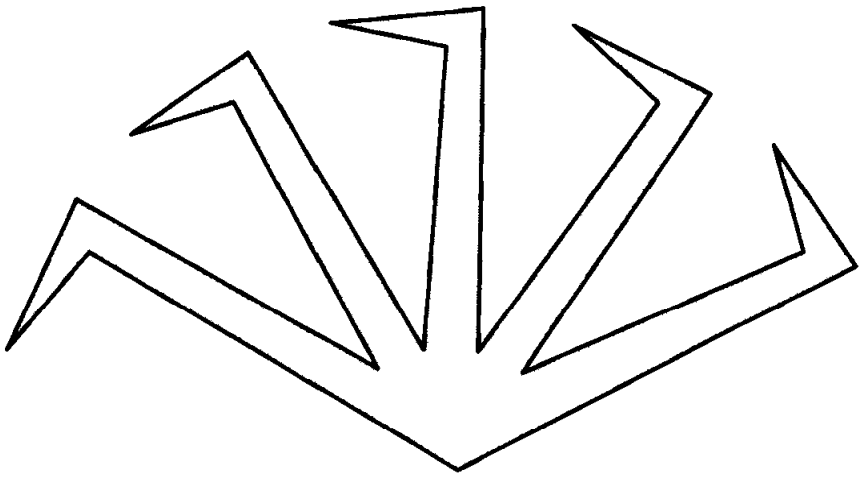


FIGURE 1.8 $\lfloor n/4 \rfloor$ edge guards are necessary (Toussaint).

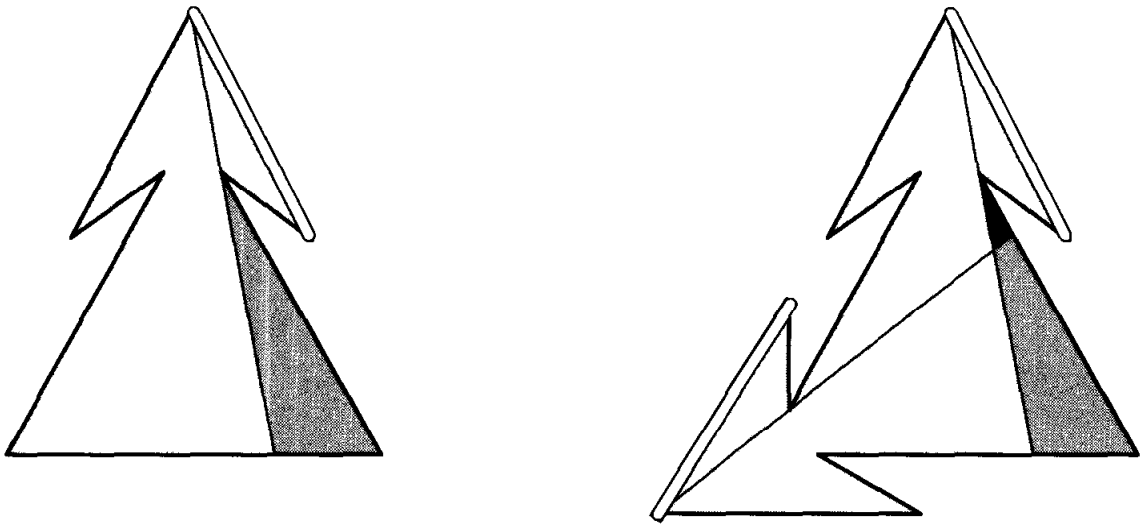


FIGURE 1.9 Two polygons that require $\lfloor (n+1)/4 \rfloor$ edge guards.

Are clearly seeing guards stronger or weaker than usual guards? What relationship between $G'(n)$ and $G(n)$ follows from their relative strength? ($G(n)$ is defined in Section 1.1.2) Does Fisk's proof establish $\lfloor n/3 \rfloor$ sufficiency for clear visibility? Try to determine $G'(n)$ exactly.

3. *Clear visibility, vertex guards* (Thomas Shermer). Answer question 2, but for *vertex* guards: guards restricted to vertices.
4. *Edge guards* [open]. An *edge guard* is a guard who may patrol one edge e of a polygon. A point $y \in P$ is covered by the guard if there is some point $x \in e$ such that x can see y. Another way to view this is to imagine a fluorescent light whose extent matches e. The portion of P that is illuminated by this light is the set of points covered by the edge guard.

   Toussaint showed that $\lfloor n/4 \rfloor$ edge guards are sometimes necessary, as demonstrated by the "half-swastika" polygon shown in Figure 1.8 (O'Rourke 1987, p. 83). He conjectured that $\lfloor n/4 \rfloor$ suffice except for a few small values of n. This odd exception is necessitated by the two "arrowhead" polygons shown in Figure 1.9, which do not seem to generalize. These examples are taken from Shermer (1992).

   Prove or disprove Toussaint's conjecture.

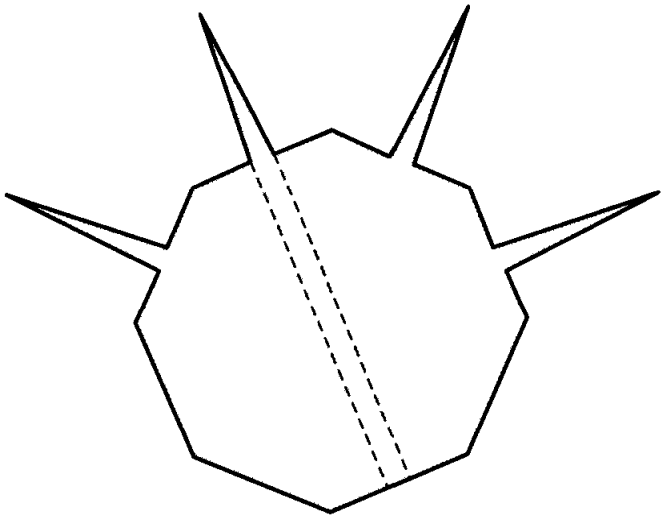


FIGURE 1.10 $\lfloor n/5 \rfloor$ edge guards are necessary (Toussaint).

5. *Edge guards in star polygons* [open]. A *star polygon* is one that can be covered by a single (point) guard. Toussaint proved that $\lfloor n/5 \rfloor$ edge guards are sometimes necessary to cover a star polygon with the example shown in Figure 1.10 (O'Rourke 1987, p. 119). The conjecture that $\lfloor n/5 \rfloor$ always suffice was shown to be false for $n = 14$ (Subramaniyam & Diwan 1991), but otherwise little is known. Prove or disprove that $n/5 + c$ suffice for some constant $c > 0$.
6. *Guards in polyhedra.* Design a polyhedron such that guards placed at every vertex fail to completely cover the interior. A polyhedron is a three-dimensional version of a polygon, composed of polygonal faces, and enclosing a volume. A precise definition is offered in Chapter 4 (Section 4.1). *Hint:* See O'Rourke (1987, Sec. 10.2.2).

1.2. TRIANGULATION: THEORY

In this section we prove that every polygon has a triangulation, and we establish some basic properties of triangulations. In later sections (1.4–1.6.5) we will discuss algorithms for constructing triangulations.

A natural reaction on being presented with the question, "Must every polygon have a triangulation?" is to respond with another question: "How could a polygon *not* have a triangulation?" Indeed it cannot not have one! But if you feel this is too obvious for a proof, consider the equivalent question in three dimensions: There the natural generalization is false! See O'Rourke (1987, p. 253–4).

1.2.1. Existence of a Diagonal

The key to proving the existence of a triangulation is proving the existence of a diagonal. Once we have that, the rest will follow easily. For the proof, we need one other even more obvious fact: Every polygon must have at least one strictly convex vertex.[12]

Lemma 1.2.1. *Every polygon must have at least one strictly convex vertex.*

[12]Recall that a (nonstrict) convex vertex could be collinear with its adjacent vertices.

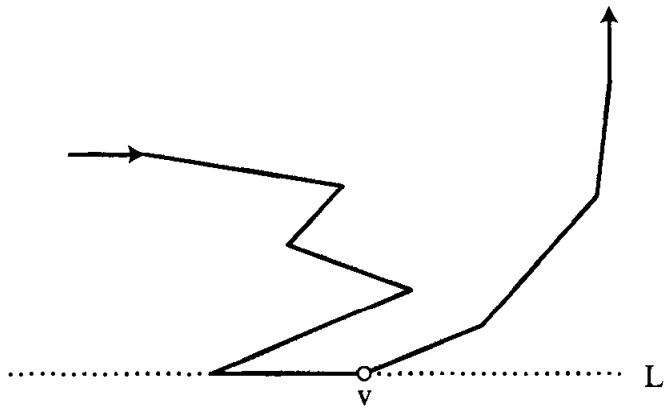


FIGURE 1.11 The rightmost lowest vertex must be strictly convex.

Proof. If the edges of a polygon are oriented so that their direction indicates a counterclockwise traversal, then a strictly convex vertex is a left turn for someone walking around the boundary, and a reflex vertex is a right turn. The interior of the polygon is always to the left of this hypothetical walker. Let L be a line through a lowest vertex v of P, lowest in having minimum y coordinate with respect to a coordinate system; if there are several lowest vertices, let v be the rightmost. The interior of P must be above L. The edge following v must lie above L. See Figure 1.11. Together these conditions imply that the walker makes a left turn at v and therefore that v is a strictly convex vertex. □

This proof can be used to construct an efficient test for the orientation of a polygon (Exercise 1.3.9[3]).

Lemma 1.2.2 (Meisters). *Every polygon of $n \geq 4$ vertices has a diagonal.*

Proof. Let v be a strictly convex vertex, whose existence is guaranteed by Lemma 1.2.1. Let a and b the vertices adjacent to v. If ab is a diagonal, we are finished. So suppose ab is not a diagonal. Then either ab is exterior to P, or it intersects ∂P. In either case, since $n > 3$, the closed triangle $\triangle avb$ contains at least one vertex of P other than a, v, b. Let x be the vertex of P in $\triangle avb$ that is closest to v, where distance is measured orthogonal to the line through ab. Thus x is the first vertex in $\triangle avb$ hit by a line L parallel to ab moving from v to ab. See Figure 1.12.

Now we claim that vx is a diagonal of P. For it is clear that the interior of $\triangle avb$ intersected with the halfplane bounded by L that includes v (the shaded region in the figure) is empty of points of ∂P. Therefore vx cannot intersect ∂P except at v and x, and so it is a diagonal. □

Theorem 1.2.3 (Triangulation). *Every polygon P of n vertices may be partitioned into triangles by the addition of (zero or more) diagonals.*

Proof. The proof is by induction. If $n = 3$, the polygon is a triangle, and the theorem holds trivially.

Let $n \geq 4$. Let $d = ab$ be a diagonal of P, as guaranteed by Lemma 1.2.2. Because d by definition only intersects ∂P at its endpoints, it partitions P into two polygons,

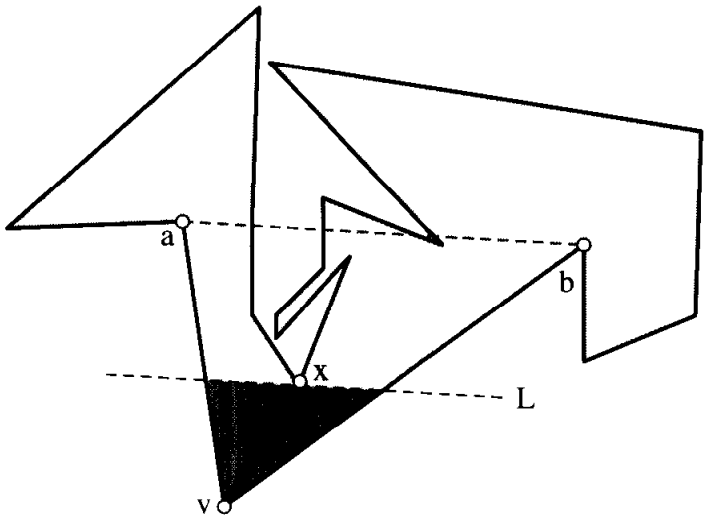


FIGURE 1.12 vx must be a diagonal.

each using d as an edge, and each of fewer than n vertices; see Figure 1.13. The reason each has fewer vertices is that no vertices are added by this process, and clearly there is at least one vertex in each part in addition to a and b. Applying the induction hypothesis to the two subpolygons completes the proof. □

1.2.2. Properties of Triangulations

Although in general there can be a large number of different ways to triangulate a given polygon (Exercise 1.2.5[4]), they all have the same number of diagonals and triangles, as is easily established by the same argument as used in Theorem 1.2.3:

Lemma 1.2.4 (Number of Diagonals). *Every triangulation of a polygon P of n vertices uses $n - 3$ diagonals and consists of $n - 2$ triangles.*

Proof. The proof is by induction. Both claims are trivially true for $n = 3$.

Let $n \geq 4$. Partition P into two polygons P_1 and P_2 with a diagonal $d = ab$. Let the two polygons have n_1 and n_2 vertices respectively. We have that $n_1 + n_2 = n + 2$, since a and b are counted in both n_1 and n_2. Applying the induction hypothesis to

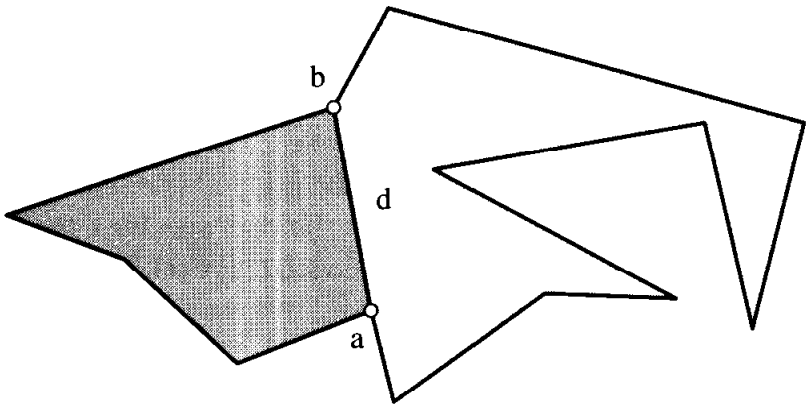


FIGURE 1.13 A diagonal partitions a polygon into two smaller polygons.

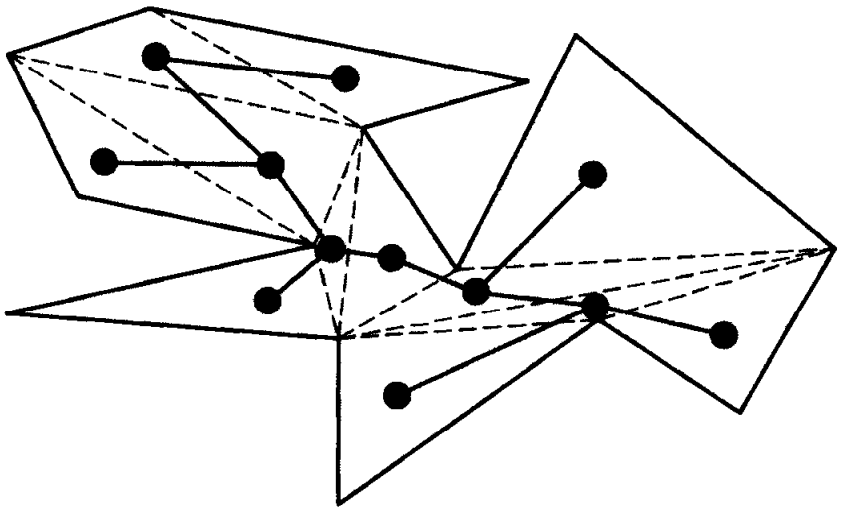


FIGURE 1.14 Triangulation dual.

the subpolygons, we see that altogether there are $(n_1 - 3) + (n_2 - 3) + 1 = n - 3$ diagonals, with the final $+1$ term counting d. And there are $(n_1 - 2) + (n_2 - 2) = n - 2$ triangles. □

Corollary 1.2.5 (Sum of Angles). *The sum of the internal angles of a polygon of n vertices is $(n - 2)\pi$.*

Proof. There are $n - 2$ triangles by Lemma 1.2.4, and each contributes π to the internal angles. □

1.2.3. Triangulation Dual

An important concept in graph theory is the "dual" of a graph. We will not need this concept in its full generality, but rather we will define specific dual graphs as the need arises. In particular, studying the triangulation dual reveals useful structure in the triangulation.

The *dual* T of a triangulation of a polygon is a graph with a node associated with each triangle and an arc between two nodes iff their triangles share a diagonal. See Figure 1.14.

Lemma 1.2.6. *The dual T of a triangulation is a tree,*[13] *with each node of degree at most three.*

Proof. That each node has degree at most three is immediate from the fact that a triangle has at most three sides to share.

Suppose T is not a tree. Then it must have a cycle C. If this cycle is drawn as a path π in the plane, connecting with straight segments the midpoints of the diagonals shared by the triangles whose nodes comprise C (to make the path specific), then it must enclose some polygon vertices: namely one endpoint of each diagonal crossed by π. But then π must also enclose points exterior to the polygon, for these enclosed vertices are on ∂P. This contradicts the simplicity of the polygon. □

[13] A *tree* is a connected graph with no cycles.

The nodes of degree one are leaves of T; nodes of degree two lie on paths of the tree; nodes of degree three are branch points. Note that T is a binary tree when rooted at any node of degree one or two! Given the ubiquity of binary trees in computer science, this correspondence between triangulation duals and binary trees is fortunate and may often be exploited (Exercise 1.2.5[7]).

Lemma 1.2.6 leads to an easy proof of Meisters's "Two Ears Theorem" (Meisters 1975), which, although simple, is quite useful. Three consecutive vertices of a polygon a, b, c form an *ear* of the polygon if ac is a diagonal; b is the ear *tip*. Two ears are *nonoverlapping* if their triangle interiors are disjoint.

Theorem 1.2.7 (Meisters's Two Ears Theorem). *Every polygon of $n \geq 4$ vertices has at least two nonoverlapping ears.*

Proof. A leaf node in a triangulation dual corresponds to an ear. A tree of two or more nodes (by Lemma 1.2.4 the tree has $(n-2) \geq 2$ nodes) must have at least two leaves. □

1.2.4. 3-Coloring Proof

This theorem in turn leads to an easy proof of the 3-colorability of triangulation graphs. The idea is to remove an ear for induction, which, because it only "interfaces" at its one diagonal, can be colored consistently.

Theorem 1.2.8 (3-coloring). *The triangulation graph of a polygon P may be 3-colored.*

Proof. The proof is by induction on the number of vertices n. Clearly a triangle can be 3-colored.

Assume therefore that $n \geq 4$. By Theorem 1.2.7, P has an ear $\triangle abc$, with ear tip b. Form a new polygon P' by cutting off the ear: That is, replace the sequence abc in ∂P with ac in $\partial P'$. P' has $n-1$ vertices: It is missing only b. Apply the induction hypothesis to 3-color P'. Now put the ear back, coloring b with the color not used at a and c. This is a 3-coloring of P. □

1.2.5. Exercises

1. *Exterior angles* [easy]. What is the sum of the exterior angles of a polygon of n vertices?
2. *Realization of triangulations.* Prove or disprove: Every binary tree is realizable as a triangulation dual of a polygon.
3. *Extreme triangulations.* Which polygons have the fewest number of distinct triangulations? Can polygons have unique triangulations? Which polygons have the largest number of distinct triangulations?
4. *Number of triangulations* [difficult]. How many distinct triangulations are there of a convex polygon of n vertices?
5. *Quad-ears.* An *orthogonal polygon* is one composed entirely of edges that meet orthogonally (e.g., horizontal and vertical edges). Define a notion of a "quad-ear" of an orthogonal polygon, a four-sided version of an ear, and answer the question of whether every orthogonal polygon has a quad-ear under your definition.

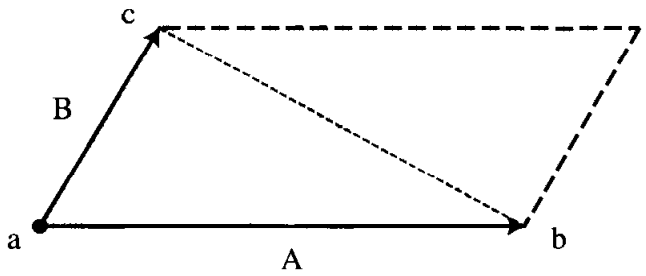


FIGURE 1.15 Cross product parallelogram.

6. *Do nonconvex polygons have mouths?* (Pierre Beauchemin). Define three consecutive vertices a, b, c of a polygon to form a *mouth* if b is reflex and the closed $\triangle abc$ does not contain any vertices other than its three corners. Prove or disprove: Every nonconvex polygon has a mouth.
7. *Tree rotations.* For those who know tree rotations used to balance binary trees:[14] Interpret tree rotations in terms of polygon triangulations.
8. *Diagonals $\Rightarrow$ triangulation.* Given a list of diagonals of a polygon forming a triangulation, with each diagonal specified by counterclockwise indices of the endpoints, design an algorithm to build the triangulation dual tree. [difficult]: Achieve $O(n)$ time at the expense of $O(n^2)$ space.

1.3. AREA OF POLYGON

In this section we will explore the question of how to compute the area of a polygon. Although this is an interesting question in its own right, our objective is to prepare the way for calculation of containment in halfplanes, the intersection between line segments, visibility relations, and ultimately to lead to a triangulation algorithm in Section 1.6.5.

1.3.1. Area of a Triangle

The area of a triangle is one half the base times the altitude. However, this formula is not directly useful if we want the area of a triangle T whose three vertices are arbitrary points a, b, c. Let us denote this area as $\mathcal{A}(T)$. The base is easy: $|a-b|$,[15] but the altitude is not so immediately available from the coordinates, unless the triangle happens to be oriented with one side parallel to one of the axes.

1.3.2. Cross Product

From linear algebra we know that the magnitude of the cross product of two vectors is the area of the parallelogram they determine: If A and B are vectors, then $|A \times B|$ is the area of the parallelogram with sides A and B, as shown in Figure 1.15. Since any triangle can be viewed as half of a parallelogram, this gives an immediate method of computing the area from coordinates. Just let $A = b - a$ and $B = c - a$. Then the area is half the length of $A \times B$. The cross product can be computed from the

[14] See, e.g., Cormen, Leiserson & Rivest (1990, pp. 265–7).

[15] $|a - b|$ is the length of the vector $a - b$, sometimes written $\|a - b\|$.

following determinant, where $\hat{i}$, $\hat{j}$, and $\hat{k}$ are unit vectors in the x, y, and z directions respectively:

$$\begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \end{vmatrix} = (A_1B_2 - A_2B_1)\hat{i} + (A_2B_0 - A_0B_2)\hat{j} + (A_0B_1 - A_1B_0)\hat{k}. \tag{1.1}$$

For two-dimensional vectors, $A_2 = B_2 = 0$, so the above calculation reduces to $(A_0B_1 - A_1B_0)\hat{k}$: The cross product is a vector normal (perpendicular) to the plane of the triangle. Thus the area is given by

$$\mathcal{A}(T) = \frac{1}{2}(A_0B_1 - A_1B_0).$$

Substitution of $A = b - a$ and $B = c - a$ yields

$$2\mathcal{A}(T) = a_0b_1 - a_1b_0 + a_1c_0 - a_0c_1 + b_0c_1 - c_0b_1 \tag{1.2}$$

$$= (b_0 - a_0)(c_1 - a_1) - (c_0 - a_0)(b_1 - a_1)\,. \tag{1.3}$$

This achieves our immediate goal: an expression for the area of the triangle as a function of the coordinates of its vertices.

1.3.3. Determinant Form

There is another way to represent the calculation of the cross product that is formally identical but generalizes more easily to higher dimensions.[16]

The expression obtained above (Equation 1.3), is the value of the 3×3 determinant of the three point coordinates, with the third coordinate replaced by 1:[17]

$$\begin{vmatrix} a_0 & a_1 & 1 \\ b_0 & b_1 & 1 \\ c_0 & c_1 & 1 \end{vmatrix} = (b_0 - a_0)(c_1 - a_1) - (c_0 - a_0)(b_1 - a_1) = 2\mathcal{A}(T). \tag{1.4}$$

This determinant is explored in Exercise 1.6.8[1]. We summarize in a lemma.

Lemma 1.3.1. *Twice the area of a triangle $T = (a, b, c)$ is given by*

$$2\mathcal{A}(T) = \begin{vmatrix} a_0 & a_1 & 1 \\ b_0 & b_1 & 1 \\ c_0 & c_1 & 1 \end{vmatrix} = (b_0 - a_0)(c_1 - a_1) - (c_0 - a_0)(b_1 - a_1). \tag{1.5}$$

[16]Note that the operation of cross product is restricted to three-dimensional vectors (or two-dimensional vectors with a zero third coordinate). It is more accurate to view the cross product as an exterior product producing, not another vector, but a "bivector." See, e.g., Koenderink (1990).

[17]One can view each row as a point in "homogenous coordinates," with the third coordinate normalized to 1.

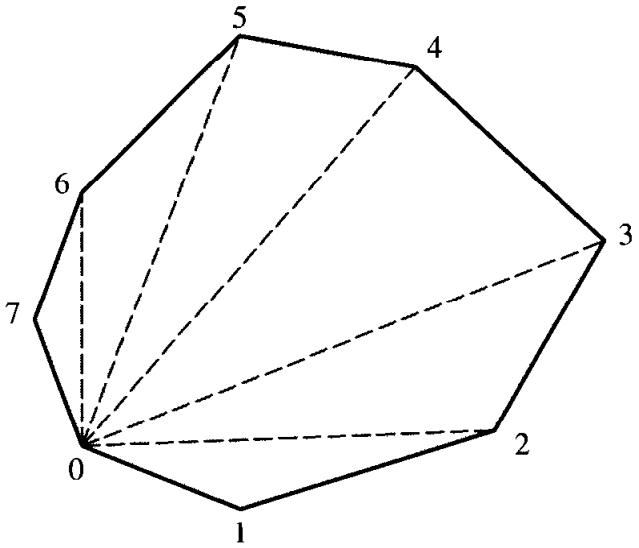


FIGURE 1.16 Triangulation of a convex polygon. The fan center is at 0.

1.3.4. Area of a Convex Polygon

Now that we have an expression for the area of a triangle, it is easy to find the area of any polygon by first triangulating it, and then summing the triangle areas. But it would be pleasing to avoid the rather complex step of triangulation, and indeed this is possible. Before turning to that issue, we consider convex polygons, where triangulation is trivial.

Every convex polygon may be triangulated as a "fan," with all diagonals incident to a common vertex; and this may be done with any vertex serving as the fan "center." See Figure 1.16. Therefore the area of a polygon with vertices $v_0, v_1, \ldots, v_{n-1}$ labeled counterclockwise can be calculated as

$$\mathcal{A}(P) = \mathcal{A}(v_0, v_1, v_2) + \mathcal{A}(v_0, v_2, v_3) + \cdots + \mathcal{A}(v_0, v_{n-2}, v_{n-1}). \tag{1.6}$$

Here v_0 is the fan center.

We will warm up to the result we will prove in Theorem 1.3.3 below by examining convex and nonconvex quadrilaterals, where the relevant relationships are obvious.

1.3.5. Area of a Convex Quadrilateral

The area of a convex quadrilateral $Q = (a, b, c, d)$ may be written in two ways, depending on the two different triangulations (see Figure 1.17):

$$\mathcal{A}(Q) = \mathcal{A}(a, b, c) + \mathcal{A}(a, c, d) = \mathcal{A}(d, a, b) + \mathcal{A}(d, b, c). \tag{1.7}$$

Writing out the expressions for the areas using Equation (1.2) for the two terms of the first triangulation, we get

$$\begin{aligned} 2\mathcal{A}(Q) = {} & a_0b_1 - a_1b_0 + a_1c_0 - a_0c_1 + b_0c_1 - c_0b_1 \\ & + a_0c_1 - a_1c_0 + a_1d_0 - a_0d_1 + c_0d_1 - d_0c_1. \end{aligned} \tag{1.8}$$

Note that the terms $a_1c_0 - a_0c_1$ appear in $\mathcal{A}(a, b, c)$ and in $\mathcal{A}(a, c, d)$ with opposite signs, and so they cancel. Thus the terms "corresponding" to the diagonal ac cancel;

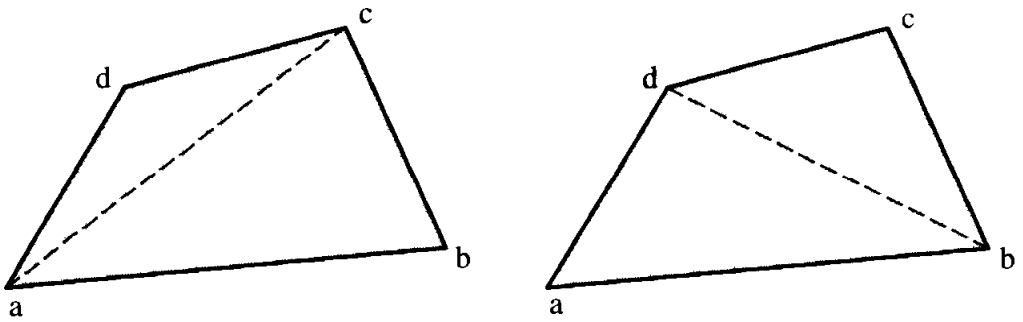


FIGURE 1.17 The two triangulations of a convex quadrilateral.

similarly the terms corresponding to the diagonal db in the second triangulation cancel. And thus we arrive at the exact same expression independent of the triangulation, as of course we must.

Generalizing, we see we get two terms per polygon edge, and none for internal diagonals. So if the coordinates of vertex v_i are x_i and y_i, twice the area of a convex polygon is given by

$$2\mathcal{A}(P) = \sum_{i=0}^{n-1} (x_i y_{i+1} - y_i x_{i+1}). \tag{1.9}$$

We will soon see that this equation holds for nonconvex polygons as well.

1.3.6. Area of a Nonconvex Quadrilateral

Now suppose we have a nonconvex quadrilateral $Q = (a, b, c, d)$ as shown in Figure 1.18. Then there is only one triangulation, using the diagonal db. But we just showed that the algebraic expression obtained is independent of the diagonal chosen, so it must be the case that the equation

$$\mathcal{A}(Q) = \mathcal{A}(a, b, c) + \mathcal{A}(a, c, d)$$

is still true, even though the diagonal ac is external to Q. This equation has an obvious interpretation: $\mathcal{A}(a, c, d)$ is negative, and it is therefore subtracted from the surrounding

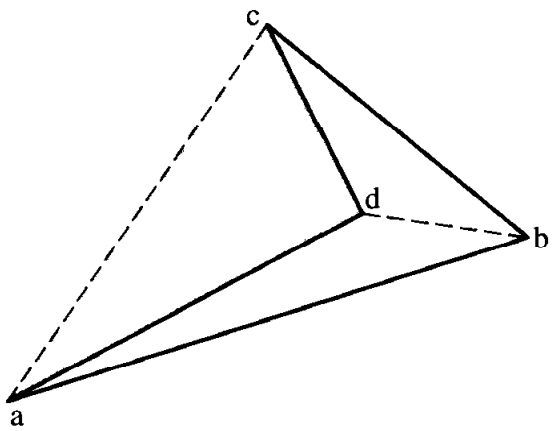


FIGURE 1.18 Triangulation of a nonconvex quadrilateral. The shaded area $\mathcal{A}(a, d, c)$ is negative.

triangle $\triangle abc$. And indeed, note that (a, c, d) is a clockwise path, so the cross product formulation shows that the area will be negative.

The phenomenon observed with a nonconvex quadrilateral is general, as we now proceed to demonstrate.

1.3.7. Area from an Arbitrary Center

We now formalize the observations in the preceding paragraphs, which we will then use to obtain the area of general nonconvex polygons.

Let us generalize the method of summing the areas of the triangles in a triangulation to summing areas based on an arbitrary, perhaps external, point p. Let $T = \triangle abc$ be a triangle, with the vertices oriented counterclockwise, and let p be any point in the plane. Then we claim that

$$\mathcal{A}(T) = \mathcal{A}(p, a, b) + \mathcal{A}(p, b, c) + \mathcal{A}(p, c, a). \tag{1.10}$$

Consider Figure 1.19. With $p = p_1$, the first term of Equation (1.10), $\mathcal{A}(p_1, a, b)$, is negative because the vertices are clockwise, whereas the remaining two terms are positive because the vertices are counterclockwise. Now note that $\mathcal{A}(p_1, a, b)$ subtracts exactly that portion of the quadrilateral (p_1, b, c, a) that lies outside T, leaving the total sum precisely $\mathcal{A}(T)$ as claimed.

Similarly, from $p = p_2$, both $\mathcal{A}(p_2, a, b)$ and $\mathcal{A}(p_2, b, c)$ are negative because the vertices are clockwise, and they remove from $\mathcal{A}(p_2, c, a)$, which is positive, precisely the amount needed to leave $\mathcal{A}(T)$.

All other positions for p in the plane not internal to T are equivalent to either p_1 or p_2 by symmetry; and of course the equation holds when p is internal, as we argued in Section 1.3.4. Therefore we have established the following lemma:

Lemma 1.3.2. *If $T = \triangle abc$ is a triangle, with vertices oriented counterclockwise, and p is any point in the plane, then*

$$\mathcal{A}(T) = \mathcal{A}(p, a, b) + \mathcal{A}(p, b, c) + \mathcal{A}(p, c, a)\ . \tag{1.11}$$

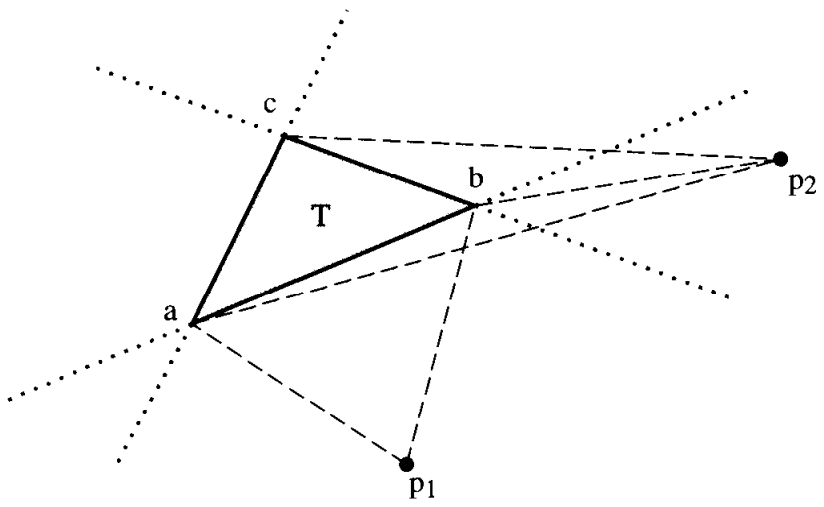


FIGURE 1.19 Area of T based on various external points p_1, p_2.

We may now generalize the preceding lemma to establish the same equation (generalized) for arbitrary polygons.

Theorem 1.3.3 (Area of Polygon).[18] *Let a polygon (convex or nonconvex) P have vertices $v_0, v_1, \ldots, v_{n-1}$ labeled counterclockwise, and let p be any point in the plane. Then*

$$\mathcal{A}(P) = \mathcal{A}(p, v_0, v_1) + \mathcal{A}(p, v_1, v_2) + \mathcal{A}(p, v_2, v_3) + \cdots + \mathcal{A}(p, v_{n-2}, v_{n-1}) + \mathcal{A}(p, v_{n-1}, v_0). \tag{1.12}$$

If $v_i = (x_i, y_i)$, this expression is equivalent to the equations

$$2\mathcal{A}(P) = \sum_{i=0}^{n-1} (x_i y_{i+1} - y_i x_{i+1}) \tag{1.13}$$

$$= \sum_{i=0}^{n-1} (x_i + x_{i+1})(y_{i+1} - y_i). \tag{1.14}$$

Proof. We prove the area sum equation by induction on the number of vertices n of P. The base case, $n = 3$, is established by Lemma 1.3.2.

Suppose then that Equation (1.12) is true for all polygons with $n - 1$ vertices, and let P be a polygon of n vertices. By Theorem 1.2.7, P has an "ear." Renumber the vertices of P so that $E = (v_{n-2}, v_{n-1}, v_0)$ is an ear. Let P_{n-1} be the polygon obtained by removing E. By the induction hypothesis,

$$\mathcal{A}(P_{n-1}) = \mathcal{A}(p, v_0, v_1) + \cdots + \mathcal{A}(p, v_{n-3}, v_{n-2}) + \mathcal{A}(p, v_{n-2}, v_0).$$

By Lemma 1.3.2,

$$\mathcal{A}(E) = \mathcal{A}(p, v_{n-2}, v_{n-1}) + \mathcal{A}(p, v_{n-1}, v_0) + \mathcal{A}(p, v_0, v_{n-2}).$$

Since $\mathcal{A}(P) = \mathcal{A}(P_{n-1}) + \mathcal{A}(E)$, we have

$$\mathcal{A}(P) = \mathcal{A}(p, v_0, v_1) + \cdots + \mathcal{A}(p, v_{n-3}, v_{n-2}) + \mathcal{A}(p, v_{n-2}, v_0) + \mathcal{A}(p, v_{n-2}, v_{n-1}) + \mathcal{A}(p, v_{n-1}, v_0) + \mathcal{A}(p, v_0, v_{n-2}).$$

But note that $\mathcal{A}(p, v_0, v_{n-2}) = -\mathcal{A}(p, v_{n-2}, v_0)$. Canceling these terms leads to the claimed equation.

Equation (1.13) is obtained by expansion of the determinants and canceling terms, as explained in Section 1.3.5. Equation (1.14) can be seen as equivalent by multiplying out and again canceling terms. □

[18] This theorem can be viewed as a discrete version of Green's theorem, which relates an integral around the boundary of a region with an integral over the interior of the region: $\int_{\partial P} \omega = \iint_P d\omega$, where ω is a "1-form" (see, e.g., Buck & Buck (1965, p. 406) or Koenderink (1990, p. 99)).

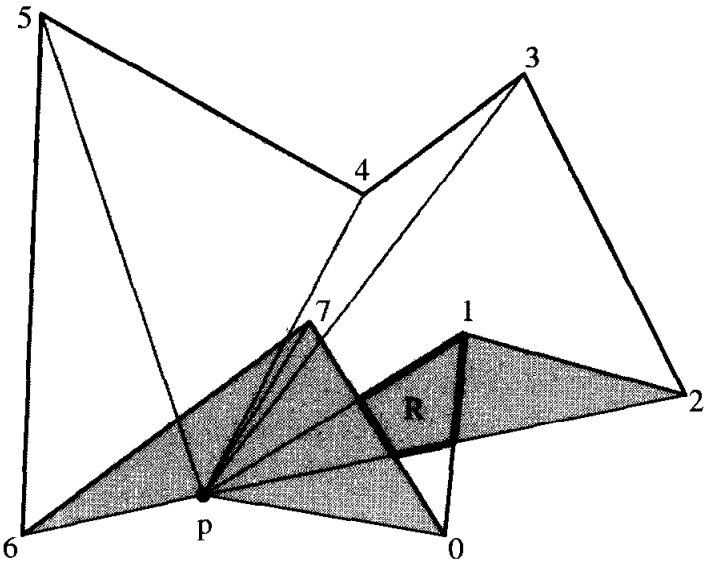


FIGURE 1.20 Computation of the area of a nonconvex polygon from point p. The darker triangles are oriented clockwise, and thus they have negative area.

Equation (1.14) can be computed with one multiplication and two additions per term, whereas Equation (1.13) uses two multiplications and one addition. The second form is therefore more efficient in most implementations.

In Figure 1.20, the triangles $\triangle p12$, $\triangle p67$, and $\triangle p70$ are oriented clockwise, and the remainder are counterclockwise. One can think of the counterclockwise triangles as attaching to each point they cover a $+1$ charge, whereas the clockwise triangles attach a -1 charge. Then the points R of $\triangle p12$ that falls inside the polygon (labeled in the figure) are given a -1 charge by this clockwise triangle; but R is also covered by two counterclockwise triangles, $\triangle p01$ and $\triangle p23$. So R has net $+1$ charge. Similarly every point inside P is assigned a net $+1$ charge, and every point outside is assigned a net 0 charge.

1.3.8. Volume in Three and Higher Dimensions

One of the benefits of the determinant formulation of the area of a triangle in Lemma 1.3.1 is that it extends directly into higher dimensions. In three dimensions, the volume of a tetrahedron T with vertices a, b, c, d is

$$6\mathcal{V}(T) = \begin{vmatrix} a_0 & a_1 & a_2 & 1 \\ b_0 & b_1 & b_2 & 1 \\ c_0 & c_1 & c_2 & 1 \\ d_0 & d_1 & d_2 & 1 \end{vmatrix} \tag{1.15}$$

$$\begin{aligned} = &-(a_2 - d_2)(b_1 - d_1)(c_0 - d_0) + (a_1 - d_1)(b_2 - d_2)(c_0 - d_0) \\ &+ (a_2 - d_2)(b_0 - d_0)(c_1 - d_1) - (a_0 - d_0)(b_2 - d_2)(c_1 - d_1) \\ &- (a_1 - d_1)(b_0 - d_0)(c_2 - d_2) + (a_0 - d_0)(b_1 - d_1)(c_2 - d_2). \end{aligned} \tag{1.16}$$

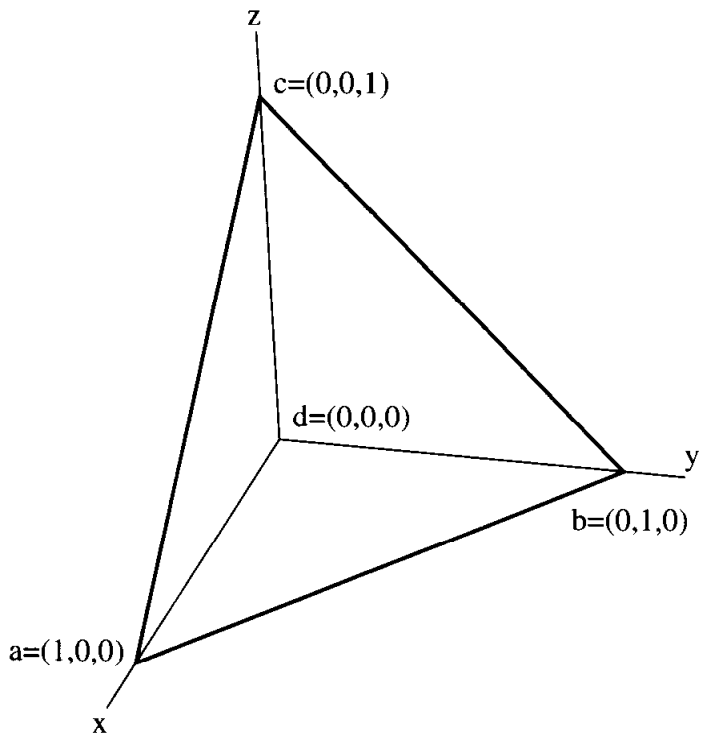


FIGURE 1.21 Tetrahedron at the origin.

This volume is signed; it is positive if (a, b, c) form a counterclockwise circuit when viewed from the side away from d, so that the face normal determined by the right-hand rule points toward the outside. For example, let $a = (1, 0, 0)$, $b = (0, 1, 0)$, $c = (0, 0, 1)$, and $d = (0, 0, 0)$. Then (a, b, c) is counterclockwise from outside; see Figure 1.21. Substitution into Equation (1.15) yields a determinant of 1, so $\mathcal{V}(T) = \frac{1}{6}$. This accords with the $\frac{1}{3}$ base area times height rule: $\frac{1}{3} \cdot \frac{1}{2} \cdot 1$. We will make use of this volume formula later to compute the "convex hull" of points in three dimensions (Chapter 4).

Remarkably, Theorem 1.3.3 generalizes directly also: The volume of a polyhedron may be computed by summing the (signed) volumes of tetrahedra formed by an arbitrary point and each triangular face of the polyhedron (Exercise 4.7[7].) Here all the faces must be oriented counterclockwise from outside.

Moreover, Equation (1.15) generalizes to higher dimensions d, yielding the volume of the d-dimensional "simplex" (the generalization of a tetrahedron to higher dimensions) times the constant $d!$.

1.3.9. Exercises

1. *Triple product.* Interpret the determinant expression (Equation (1.4)) for the area of a triangle in terms of the triple vector product.

   If A, B, and C are three-dimensional vectors, then their triple product is $A \cdot (B \times C)$. This is a scalar with value equal to the volume of the parallelepiped determined by the three vectors, determined in the same sense that two vectors determine a parallelogram. The value is the same as that of the determinant

$$A \cdot (B \times C) = \begin{vmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{vmatrix}.$$

Assuming this determinant is the parallelepiped volume, argue that Equation (1.4) is twice the area of the indicated triangle.

2. *Orientation of a polygon: from area* [easy]. Given a list of vertices of a simple polygon in boundary traversal order, how can its orientation (clockwise versus counterclockwise) be determined using Theorem 1.3.3?
3. *Orientation of a polygon.* Use the proof of Lemma 1.2.1 to design a more efficient algorithm for determining the orientation of a polygon.
4. *Volume of a cube.* Compute the volume of a unit cube (side length 1) with the analog of Equation (1.12), using one vertex as p.

1.4. IMPLEMENTATION ISSUES

The remainder of the chapter takes a rather long "digression" into implementation issues. The goal is to present code to compute a triangulation. This hinges on detecting intersection between two segments, a seemingly trivial task that often is implemented incorrectly. We will approach segment intersection using the computation of areas from Section 1.3. We start with a few representation issues.

1.4.1. Representation of a Point

Arrays versus Records

All points will be represented by arrays of the appropriate number of coordinates. It is common practice to represent a point by a record with fields named `x` and `y`, but this precludes the use of for-loops to iterate over the coordinates.[19] There may seem little need to write a for-loop to iterate over only two indices, but I find it easier to understand, and it certainly generalizes to higher dimensions more easily.

Integers versus Reals

We will represent the coordinates with integers rather than with floating-point numbers wherever possible. This will permit us to avoid the issue of floating-point round-off error and allow us to write code that is verifiably correct within a range of coordinate values. Numerical error is an important topic and will be discussed at various points throughout the book (e.g., Sections 4.3.5 and 7.2). Obviously this habit of using integers will have to be relaxed when we compute, for example, the point of intersection between two line segments. The type definitions will be isolated so that modification of the code to handle different varieties of coordinate datatypes can be made in one location.

Point Type Definition

All type identifiers will begin with lowercase `t`. All defined constants will appear entirely in uppercase. The suffixes `i` and `d` indicate `integer` and `double` types respectively. See Code 1.1. In mathematical expressions, we will write p_0 and p_1 for `p[0]` and `p[1]`.

[19] That is, precludes it in most programming languages.

```
#define X 0
#define Y 1
typedef enum {FALSE, TRUE } bool;

#define DIM 2                    /* Dimension of points */
typedef int tPointi[DIM];        /* Type integer point */
```

Code 1.1 Point type.

1.4.2. Representation of a Polygon

The main options here are whether to use an array or a list, and if the latter, whether singly or doubly linked, and whether linear or circular.

Arrays are attractive for code clarity: The structure of loops and index increments are somewhat clearer with arrays than with lists. However, insertion and deletion of points is clumsy with arrays. As the triangulation code we develop will clip off ears, we will sacrifice simplicity to gain ease of deletion. In any case, we will need to use identical structures for the convex hull code in Chapters 3 and 4, so the investment here will reward us later. With an eye toward that generality, we opt to use a doubly linked circular list to represent a polygon. The basic cell of the data structure represents a single vertex, `tVertexStructure`, whose primary data field is `tPoint`. Pointers `next` and `prev` are provided to link each vertex to its adjacent vertices. See Code 1.2. An integer index `vnum` is included for printout, and other fields (such as `bool ear`) will be added as necessary.

```
typedef struct tVertexStructure tsVertex;     /* Used only in NEW(). */
typedef tsVertex *tVertex;
struct tVertexStructure {
  int         vnum;                 /* Index */
  tPointi     v;                    /* Coordinates */
  bool        ear;                  /* TRUE iff an ear */
  tVertex     next,prev;
};
tVertex vertices = NULL;            /* "Head" of circular list. */
```

Code 1.2 Vertex structure.

At all times, a global variable `vertices` is maintained that points to some vertex cell. This will serve as the "head" of the list during iterative processing. Loops over all vertices will take the form shown in Code 1.3. Care must be exercised if the processing in the loop deletes the cell to which `vertex` points.

```
tVertex v;
v = vertices;
do {
    /* Process vertex v */
    v = v->next;
} while ( v != vertices );
```

Code 1.3 Loop to process all vertices.

We will need two basic list processing routines for vertex structures, one for allocating a new element (NEW) and another for adding a new element to the list (ADD). Looking ahead to later chapters, we write these as macros, with NEW taking the type as one parameter. This way the routines can be used for different types. (C does not permit manipulation of variables without regard to type, but macros are text based and oblivious to types). See Code 1.4. ADD first checks to see if head is non-NULL, and if so, it inserts the cell prior to head; if not, head points to the added cell, which is then the only cell in the list. The effect is that in a series of ADDS, the nth point is added prior to the 0th (the head) but after the $(n-1)$-st point.

```
#define EXIT FAILURE 1
char *malloc();

#define NEW(p, type) \
        if ((p=(type *) malloc (sizeof(type))) == NULL) {\
        printf ("NEW: Out of Memory!\n");\
        exit(EXIT FAILURE);\
    }

#define ADD( head, p ) if ( head ) {\
        p->next = head;\
        p->prev = head->prev;\
        head->prev = p;\
        p->prev->next = p;\
    }\
    else {\
        head = p;\
        head->next = head->prev = p;\
    }

#define FREE(p)    if (p) {free ((char *) p); p = NULL; }
```

Code 1.4 NEW and ADD macros. (The backslashes continue the lines so that the preprocessor does not treat those as command lines.) FREE is used in Chapters 3 and 4.

1.4.3. Code for Area

Computing the area of a polygon is now a straightforward implementation of Equations (1.12) or (1.13). The former choice, with $p = v_0$, is shown in Code 1.5.

The data structures and conventions established in the previous section are employed.

```
int     Area2( tPointi a, tPointi b, tPointi c )
{
  return
          (b[X] - a[X]) * (c[Y] - a[Y]) -
          (c[X] - a[X]) * (b[Y] - a[Y]);
}

int     AreaPoly2( void )
{
  int     sum = 0;
  tVertex p, a;

  p = vertices;    /* Fixed. */
  a = p->next;     /* Moving. */
  do {
     sum += Area2( p->v, a->v, a->next->v );
     a = a->next;
  } while ( a->next != vertices );
  return sum;
}
```

Code 1.5 `Area2` and `AreaPoly2`.

There is an interesting potential problem with `Area2`: If the coordinates are large, the multiplications of coordinates could cause integer word overflow, which is unfortunately not reported by most C implementations. For `Area2` we have followed the expression given by Equation (1.3) rather than that in (1.2), as the former both uses fewer multiplications and multiplies coordinate differences. Nevertheless, the issue remains, and we will revisit this point in Section 4.3.5. See Exercise 1.6.4[1].

1.5. SEGMENT INTERSECTION

1.5.1. Diagonals

Our goal is to develop code to triangulate a polygon. The key step will be finding a diagonal of the polygon, a direct line of sight between two vertices v_i and v_j. The segment $v_i v_j$ will not be a diagonal if it is blocked by a portion of the polygon's boundary. To be blocked, $v_i v_j$ must intersect an edge of the polygon. Note that if $v_i v_j$ only intersects an edge e at its endpoint, perhaps only a grazing contact with the boundary, it is still effectively blocked, because diagonals must have clear visibility.

The following is an immediate consequence of the definition of a diagonal (Section 1.5.1):

Lemma 1.5.1. *The segment $s = v_i v_j$ is a diagonal of P iff*

1. *for all edges e of P that are not incident to either v_i or v_j, s and e do not intersect:* $s \cap e = \emptyset$;
2. *s is internal to P in a neighborhood of v_i and v_j.*

Condition (1) of this lemma has been phrased so that the "diagonalhood" of a segment can be determined without finding the actual point of intersection between s and each e: Only a Boolean segment intersection predicate is required. Note that this would not be the case with the more direct implementation of the definition: A diagonal only intersects polygon edges at the diagonal endpoints. This phrasing would require computation of the intersection points and subsequent comparison to the endpoints. The purpose of condition (2) is to distinguish internal from external diagonals, as well as to rule out collinear overlap with an incident edge. We will revisit this condition in Section 1.6.2. We now turn our attention to developing code to check the nonintersection condition.

1.5.2. Problems with Slopes

Let $v_i v_j = ab$ and $e = cd$. A common first inclination when faced with the task of deciding whether ab and cd intersect is to find the point of intersection between the lines L_1 and L_2 containing the segments by solving the two linear equations in slope–intercept form, and then checking that the point falls on the segments. This method will clearly work, and it is not all that difficult to code. But the code is messy and error prone; it takes a surprising amount of diligence to get it exactly right. There are two special cases to handle: a vertical segment, whose containing line's slope is infinite, and parallel segments, whose containing lines do not intersect. Both cases lead to division by zero in the computations, which must be avoided by special-case code. Even beyond this, checking that the point of intersection falls on the segments can lead to numerical precision problems.

To circumvent these problems, we avoid slopes altogether.

1.5.3. `Left`

Whether two segments intersect can be decided by using a `Left` predicate, which determines whether or not a point is to the left of a directed line. How `Left` is used to decide intersection will be shown in the next section. Here we concentrate on `Left` itself.

A directed line is determined by two points given in a particular order (a, b). If a point c is to the left of the line determined by (a, b), then the triple (a, b, c) forms a counterclockwise circuit: This is what it means to be to the left of a line. See Figure 1.22.

Now the connection to signed area is finally clear: c is to the left of (a, b) iff the area of the counterclockwise triangle, $\mathcal{A}(a, b, c)$, is positive. Therefore we may implement the `Left` predicate by a single call to `Area2` (Code 1.6).

Note that `Left` could be implemented by finding the equation of the line through a and b, and substituting the coordinates of point c into the equation. This method would be straightforward but subject to the special case objections raised earlier. The area code in contrast has no special cases.

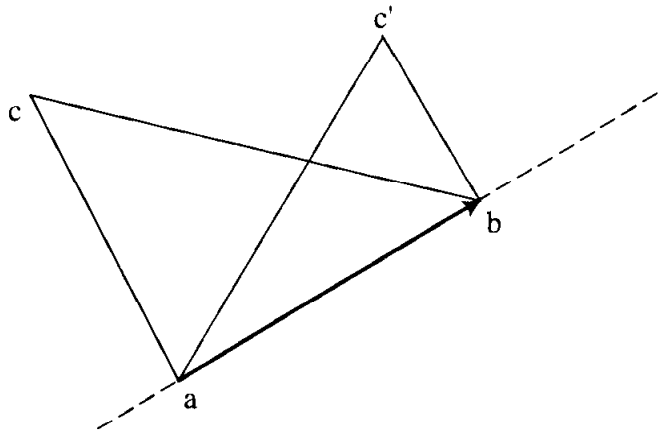


FIGURE 1.22 c is left of ab iff Δabc has positive area; $\Delta abc'$ also has positive area.

```
bool    Left( tPointi a, tPointi b, tPointi c )
{
  return Area2( a, b, c ) > 0;
}

bool    LeftOn( tPointi a, tPointi b, tPointi c )
{
  return Area2( a, b, c ) >= 0;
}

bool    Collinear( tPointi a, tPointi b, tPointi c )
{
  return Area2( a, b, c ) == 0;
}
```

Code 1.6 `Left`.

What happens when c is collinear with ab? Then the determined triangle has zero area. Thus we have the happy circumstance that the exceptional geometric situation corresponds to the exceptional numerical result. As it will sometimes be useful to distinguish collinearity, we write a separate `Collinear` predicate[20] for this, as well as `LeftOn`, giving us the equivalent of $=$, $<$, and $\leq$; see again Code 1.6.

Note that we are comparing twice the area against zero in these routines: We are not comparing the area itself. The reason is that the area might not be an integer, and we would prefer not to leave the comfortable domain of the integers.

1.5.4. Boolean Intersection

If the two segments ab and cd intersect in their interiors, then c and d are split by the line L_1 containing ab: c is to one side and d to the other. And likewise, a and b are

[20] If floating-point coordinates are demanded by a particular application, this predicate would need modification, as it depends on exact equality with zero.

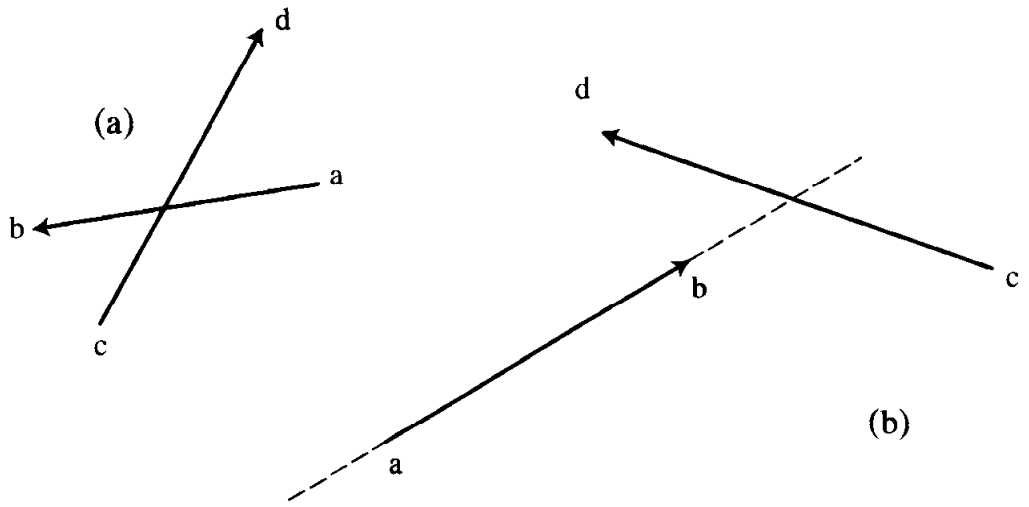


FIGURE 1.23 Two segments intersect (a) iff their endpoints are split by their determined lines; both pair of endpoints must be split (b).

split by L_2, the line containing cd. See Figure 1.23(a). Neither one of these conditions is alone sufficient to guarantee intersection, as Figure 1.23(b) shows, but it is clear that both together are sufficient. This leads to straightforward code to determine *proper* intersection, when two segments intersect at a point interior to both, if it is known that no three of the four endpoints are collinear. We can enforce this noncollinearity condition by explicit check; see Code 1.7.

```
bool  IntersectProp( tPointi a, tPointi b, tPointi c, tPointi d )
{
  /* Eliminate improper cases. */
  if (
      Collinear(a,b,c) ||
      Collinear(a,b,d) ||
      Collinear(c,d,a) ||
      Collinear(c,d,b)
      )
      return FALSE;

    return
          Xor( Left(a,b,c), Left(a,b,d) )
       && Xor( Left(c,d,a), Left(c,d,b) );
}
/* Exclusive or: T iff exactly one argument is true. */
bool     Xor( bool x, bool y )
{
  /* The arguments are negated to ensure that they are 0/1 values. */
  return !x ^ !y;
}
```

Code 1.7 `IntersectProp`.

There is unfortunate redundancy in this code, in that the four relevant triangle areas are being computed twice each. This redundancy could be removed by computing the areas and storing them in local variables, or by designing other primitives that fit the problem better. I would argue against storing the areas, as then the code would not be transparent. But it may be that the code can be designed around other primitives more naturally. It turns out that the first if-statement may be removed entirely for the purposes of triangulation, although then the routine no longer computes proper intersection (nor does it compute improper intersection). This is explored in Exercise 1.6.4[2]. I prefer to sacrifice efficiency for clarity and leave `IntersectProp` as is, for it is useful to look beyond the immediate programming task to possible other uses. In this instance, `IntersectProp` is precisely the function needed to compute clear visibility (Section 1.1.2).

One subtlety occurs here: It might be tempting to implement the exclusive-or by requiring that the products of the relevant areas be strictly negative, thus assuring that they are of opposite sign and that neither is zero:

```
Area2(a,b,c) * Area2(a,b,d) < 0
&& Area2(c,d,a) * Area2(c,d,b) < 0;
```

The weakness in this formulation is that the product of the areas might cause integer word overflow! Thus a clever coding trick to save a few lines could hide a pernicious bug. This overflow problem can be avoided by having `Area2` return $+1$, 0, or -1 rather than the true area (Sedgewick 1992, p. 350). I prefer to return the area for now, as this is useful in other contexts – for example, to compute the area of a polygon! In Chapter 4 we will revise this decision when we discuss overflow as a generic problem (Code 4.23).

Improper Intersection

Finally we must deal with the "special case" of improper intersection between the two segments, as Lemma 1.5.1 requires that the intersection be completely empty for a segment to be a diagonal. Improper intersection occurs precisely when an endpoint of one segment (say c) lies somewhere on the other (closed) segment ab. See Figure 1.24(a). This can only happen if a, b, c are collinear. But collinearity is not a sufficient condition

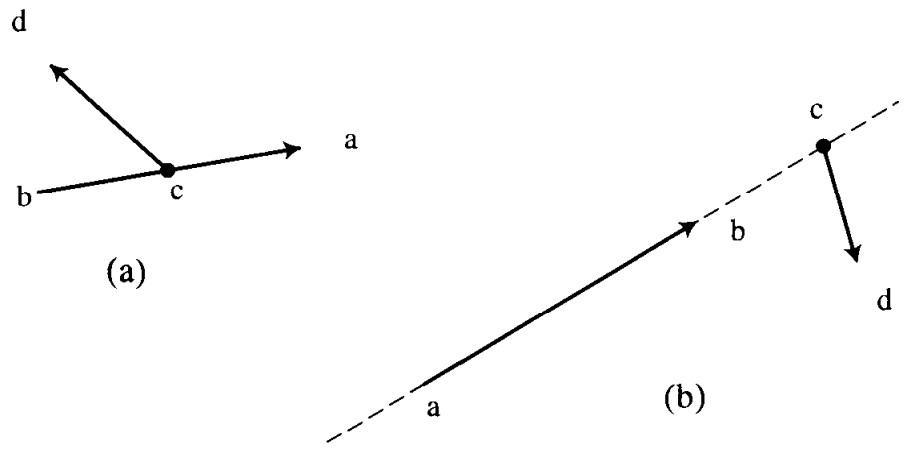


FIGURE 1.24 Improper intersection between two segments (a); collinearity is not sufficient (b).

for intersection, as Figure 1.24(b) makes clear. What we need to decide is if c is *between* a and b.

Betweenness

We would like to compute this "betweenness" predicate without resorting to slopes, which would require special-case handling. Because we will only check betweenness of c when we know it lies on the line containing ab, we may exploit this knowledge. If ab is not vertical, then c lies on ab iff the x coordinate of c falls in the interval determined by the x coordinates of a and b. If ab is vertical, then a similar check on y coordinates determines betweenness. See Code 1.8.

```
bool    Between( tPointi a, tPointi b, tPointi c )
{
   tPointi ba, ca;

   if ( ! Collinear( a, b, c ) )
      return FALSE;

  /* If ab not vertical, check betweenness on x; else on y. */
  if ( a[X] != b[X] )
     return ((a[X] <= c[X]) && (c[X] <= b[X])) ||
            ((a[X] >= c[X]) && (c[X] >= b[X]));
  else
     return ((a[Y] <= c[Y]) && (c[Y] <= b[Y])) ||
            ((a[Y] >= c[Y]) && (c[Y] >= b[Y]));
}
```

Code 1.8 `Between`.

1.5.5. Segment Intersection Code

We finally can present code for computing segment intersection. Two segments intersect iff they intersect properly or one endpoint of one segment lies between the two endpoints of the other segment. The check for improper intersection is therefore implemented by four calls to `Between`; see Code 1.9. Exercise 1.6.4[3] asks for an analysis of the inefficiencies of this routine.

1.6. TRIANGULATION: IMPLEMENTATION

1.6.1. Diagonals, Internal or External

Having developed segment intersection code, we are nearly prepared to write code for triangulating a polygon. Our first goal is to find a diagonal of the polygon.

```
bool    Intersect( tPointi a, tPointi b, tPointi c, tPointi d )
{
  if      ( IntersectProp( a, b, c, d ) )
    return  TRUE;
  else if (  Between( a, b, c )
           || Between( a, b, d )
           || Between( c, d, a )
           || Between( c, d, b )
          )
    return  TRUE;
  else    return  FALSE;
}
```

Code 1.9 `Intersect`.

Recall that Lemma 1.5.1 characterized diagonals by two conditions: nonintersection with polygon edges and being interior. If we ignore the distinction between internal and external diagonals, finding diagonals is a straightforward repeated application of `Intersect`: For every edge e of the polygon not incident to either end of the potential diagonal s, see if e intersects s. As soon as an intersection is detected, it is known that s is not a diagonal. If no such edge intersects s, then s might be a diagonal. The reason we cannot reach a positive conclusion immediately is that it is possible that one of the edges incident to an endpoint of s might be collinear with s, and this would not be detected. We will deal with this possibility shortly. The straightforward code for intersection detection is shown in Code 1.10.

```
bool   Diagonalie( tVertex a, tVertex b )
{
  tVertex c, c1;

  /* For each edge (c,c1) of P */
  c = vertices;
  do {
    c1 = c->next;
    /* Skip edges incident to a or b */
    if (     ( c != a ) && ( c1 != a )
          && ( c != b ) && ( c1 != b )
          && Intersect( a->v, b->v, c->v, c1->v )
       )
       return FALSE;
    c = c->next;
  } while ( c != vertices );
  return TRUE;
}
```

Code 1.10 `Diagonalie`.

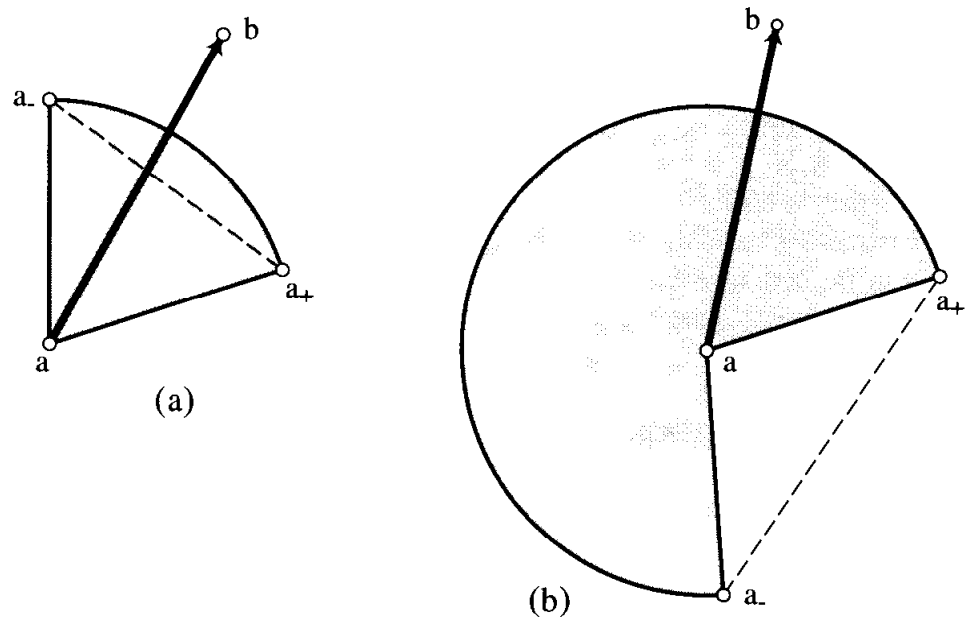


FIGURE 1.25 Diagonal $s = ab$ is in the cone determined by a_-, a, a_+: (a) convex; (b) reflex. In (b), both a_- and a_+ are right of ab.

1.6.2. InCone

Now we turn to the second condition of Lemma 1.5.1: We must distinguish the internal from the external diagonals; and we have to take care of the edges incident to the diagonal endpoints. We handle both with a Boolean procedure `InCone` that determines if one vector B lies strictly in the open cone counterclockwise between two other vectors A and C. The latter two vectors will lie along two consecutive edges of the polygon, and B lies along the diagonal. Such a procedure will suffice to determine diagonals, as will be detailed below. For the moment we concentrate on designing `InCone`.

This would be a straightforward task if the apex of the cone is a convex angle; that it may be reflex either requires a separate case or some cleverness. We will opt here for the case method, leaving the cleverness to Exercise 1.6.4[5].

The convex case is illustrated in Figure 1.25(a). The actual output produced by the code is as follows. It is clear from this figure that s is internal to P iff it is internal to the cone whose apex is a, and whose sides pass through a_- and a_+. This can be easily determined via our `Left` function: a_- must be left of ab, and a_+ must be left of ba. Both left-ofs should be strict for ab to exclude collinear overlap with the cone boundaries.

Figure 1.25(b) shows that these conditions do not suffice to characterize internal diagonals when a is reflex: a_- and a_+ could be both left of, or both right of, or one could be left and the other right of, an internal diagonal. But note that the *exterior* of a neighborhood of a is now a cone as in the convex case, for the simple reason that a reflex vertex looks like a convex vertex if interior and exterior are interchanged. So it is easiest in this case to characterize s as internal iff it is not external: It is not the case that both a_+ is left or on ab and a_- is left or on ba. Note that this time the left-ofs

must be improper, permitting collinearity, as we are rejecting diagonals that satisfy these conditions.

Finally, distinguishing between the convex and reflex cases is easily accomplished with one invocation of `Left`: a is convex iff a_- is left or on aa_+. Note that if (a_-, a, a_+) are collinear, the internal angle at a is π, which we defined as convex (Section 1.1.2).

The code in Code 1.11 implements the above ideas in a straightforward manner.

```
bool   InCone( tVertex a, tVertex b )
{
  tVertex a0,a1;     /* a0,a,a1 are consecutive vertices. */

  a1 = a->next;
  a0 = a->prev;

  /* If a is a convex vertex ... */
  if( LeftOn( a->v, a1->v, a0->v ) )
      return    Left( a->v, b->v, a0->v )
             && Left( b->v, a->v, a1->v );

  /* Else a is reflex: */
      return !(    LeftOn( a->v, b->v, a1->v )
                && LeftOn( b->v, a->v, a0->v ) );
}
```

Code 1.11 `InCone`.

Although this `InCone` test is simple, there are many opportunities to implement it incorrectly. Note that the entire function consists of five signed-area calculations, illustrating the utility of that calculation.

1.6.3. Diagonal

We now have developed code to determine if ab is a diagonal: iff `Diagonalie(a,b)`, `InCone(a,b)`, and `InCone(b,a)` are true. The `InCone` calls serve both to ensure that ab is internal and to cover the edges incident to the endpoints not examined in `Diagonalie`. There would seem to be little more to say on this topic, but in fact there is a choice of how to order the function calls. Once the question is asked, the answer is immediate: The `InCones` should be first, because they are each constant-time calculations, performed in the neighborhood of a and b without regard to the remainder of the polygon, whereas `Diagonalie` includes a loop over all n polygon edges. If either `InCone` call returns `FALSE`, the (potentially) expensive `Diagonalie` check will not be executed. See Code 1.12.

```
bool    Diagonal( tVertex a, tVertex b )
{
  return InCone( a, b ) && InCone( b, a ) && Diagonalie( a, b );
}
```

Code 1.12 `Diagonal`.

1.6.4. Exercises

1. *Integer overflow.* On a machine that restricts `int`s to $\pm 2^{31}$, how large can the coordinates of a, b, and c be to avoid integer overflow in the computation of `Area2` (Code 1.5)?
2. `IntersectProp`. Detail exactly what `IntersectProp` (Code 1.7) computes if the if-statement is deleted. Argue that after this deletion, `Intersect` (Code 1.9) still works properly.
3. *Inefficiencies in* `Intersect`. Trace out (by hand) `Intersect` (Code 1.9) and determine the largest number of calls to `Area2` (Code 1.5) it might induce. Design a new version that avoids duplicate calls.
4. *Saving intersection information.* Work out a scheme to avoid testing the same two segments for intersection twice. Analyze the time and space complexity of the new algorithm.
5. `InCone` *improvement* (Andy Mirzian). Prove that ab is in the cone at a iff at most one of these three `Lefts` are false: Left(a, a_+, b), Left(a, b, a_-), Left(a, a_-, a_+).
6. `Diagonal` *improvement.* Prove that either one of the two calls to `InCone` in `Diagonal` can be removed without changing the result.

1.6.5. Triangulation by Ear Removal

We are now prepared to develop code for finding a triangulation of a polygon. One method is to mimic the proof of the triangulation theorem (Theorem 1.2.3): Find a diagonal, cut the polygon into two pieces, and recurse on each. We will see that this method results in rather inefficient code, and we will eventually chose a method based on Meisters's two ears theorem (Theorem 1.2.7). But first we analyze the speed of the recursive method. We will use the so-called big-O notation, which we assume to be familiar to the reader.[21]

Diagonal-Based Algorithm

The method suggested by Theorem 1.2.3 is an $O(n^4)$ algorithm: There are $\binom{n}{2} = O(n^2)$ diagonal candidates, and testing each for diagonalhood costs $O(n)$. Repeating this $O(n^3)$ computation for each of the $n - 3$ diagonals yields $O(n^4)$.

We can speed this up by a factor of n by exploiting the two ears theorem: not only do we know there must be an internal diagonal, we know there must be an internal diagonal

[21] $O(f(n))$ means that a constant times $f(n)$ is an upper bound for large n; $\Omega(g(n))$ means that a constant times $g(n)$ is a lower bound for infinitely many n; $\Theta(f(n))$ means both $O(f(n))$ and $\Omega(f(n))$ hold. See, e.g., Cormen et al. (1990, Chapter 2), Albertson & Hutchinson (1988, Sec. 2.8), or Rawlins (1992, Sec. 1.4).

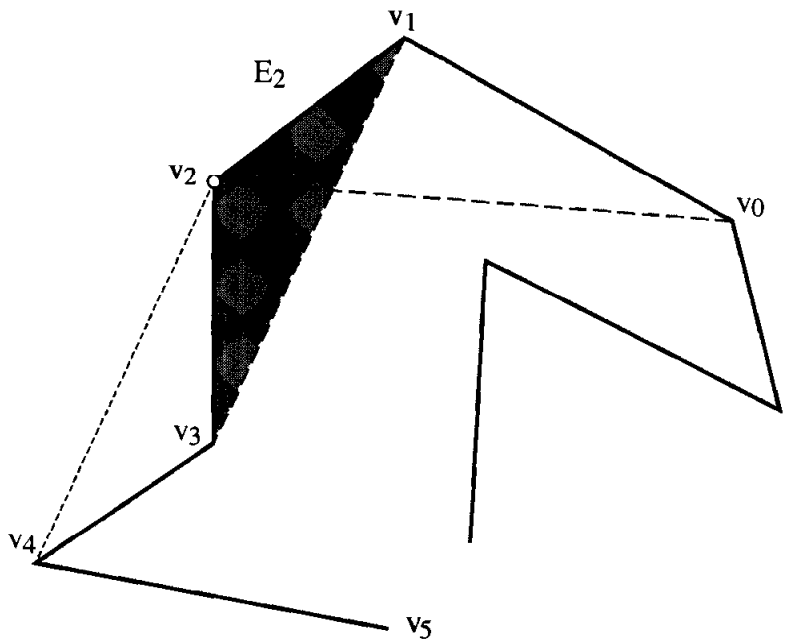


FIGURE 1.26 Clipping an ear $E_2 = \Delta(v_1, v_2, v_3)$. Here the ear status of v_1 changes from TRUE to FALSE.

that separates off an ear. There are only $O(n)$ "ear diagonal" candidates: (v_i, v_{i+2}) for $i = 0, \ldots, n-1$. This also makes the recursion simpler, as there is only one piece on which to recurse: the other is the ear, a triangle, which is of course already triangulated. Thus we can achieve a worst-case complexity of $O(n^3)$ this way.

Ear Removal

We now improve the above algorithm to $O(n^2)$. Because one call to `Diagonal` costs $O(n)$, to achieve $O(n^2)$, `Diagonal` may only be called $O(n)$ times. The key idea that permits improvement here is that removal of one ear does not change the polygon very much, and in particular, it does not change whether or not many of its vertices are potential ear tips. This suggests first determining for each vertex v_i, whether it is a potential ear tip in the sense that v_{i-1}, v_{i+1} is a diagonal. This already uses $O(n^2)$, but this expensive step need not be repeated.

Let $(v_0, v_1, v_2, v_3, v_4)$ be five consecutive vertices of P, and suppose v_2 is an ear tip and the ear $E_2 = \Delta(v_1, v_2, v_3)$ is deleted; see Figure 1.26. Which vertices' status as ear tips might change? Only v_1 and v_3. Consider v_4, for example. Whether it is an ear tip depends on whether v_3v_5 is a diagonal. The removal of E_2 leaves the endpoints of segment v_3v_5 unchanged. Certainly this removal can not block the previous line of sight between these endpoints if they could see one another. It is perhaps less clear that if they couldn't see one another, they still don't after removal of E_2. But as in the proof of Lemma 1.2.2, there are only two cases to consider. If v_3v_5 is external, then clearly removal of E_2 cannot render it internal. Otherwise $\Delta(v_3, v_4, v_5)$ must contain a vertex, and in fact a reflex vertex (x in Figure 1.12). But removal of E_2 only removes one vertex, and that is convex. Therefore, the status of v_4 is unchanged by the removal of E_2, as is that of every vertex but v_1 and v_3, whose ear diagonals are incident to the removed vertex v_2.

The implication is that, after the expensive initialization step, the ear tip status information can be updated with two calls to `Diagonal` per iteration. This leads to the pseudocode shown in Algorithm 1.1 for constructing a triangulation.

Algorithm: TRIANGULATION
Initialize the ear tip status of each vertex.
while $n > 3$ do
    Locate an ear tip v_2.
    Output diagonal v_1v_3.
    Delete v_2.
    Update the ear tip status of v_1 and v_3.

Algorithm 1.1 Triangulation algorithm.

Note that we are interpreting the task "triangulate a polygon" as "output, in arbitrary order, diagonals that form a triangulation." This is primarily for ease of presentation. Often a more structured output is required by a particular application: For example, the triangle adjacency information in the dual graph might be required. Although obtaining more structured output is no more difficult in terms of asymptotic time complexity, it often complicates the code considerably. We will not pursue these alternative triangulation outputs further.

Triangulation Code

The first task is to initialize the Boolean flag `v->ear` that is a part of the vertex structure (Code 1.2). This is accomplished by one call to `Diagonal` per vertex. See `EarInit`, Code 1.13.

```
void   EarInit( void )
{
  tVertex v0, v1, v2;    /* three consecutive vertices */

  /* Initialize v1->ear for all vertices. */
  v1 = vertices;
  do {
    v2 = v1->next;
    v0 = v1->prev;
    v1->ear = Diagonal( v0, v2 );
    v1 = v1->next;
  } while ( v1 != vertices );
}
```

Code 1.13 `EarInit`.

The main `Triangulate` code consists of a double loop. The outer loop removes one ear per iteration, halting when $n = 3$. The inner loop searches for an ear by checking

the precomputed `v2->ear` flag, where v_2 is the potential ear tip. Once an ear tip is found, the ear status of v_1 and v_3 are updated by calls to `Diagonal`, the diagonal representing the base of the ear is printed, and the ear is removed from the polygon. This removal is accomplished by rewiring the `next` and `prev` pointers for v_1 and v_3. (At this point the cell for v_2 could be freed if it is not used in a surrounding application.) Care must be exercised lest v_2 is the "head" of the vertex list, the point of access into the circular list. This head pointer, `vertices`, is moved to point to v_3 for that reason. See Code 1.14. We step through an example before analyzing the time complexity.

```
void   Triangulate( void )
{
  tVertex v0, v1, v2, v3, v4;   /* five consecutive vertices */
  int   n = nvertices;          /* number of vertices; shrinks to 3. */

  EarInit();
  /* Each step of outer loop removes one ear. */
  while ( n > 3 ) {
    /* Inner loop searches for an ear. */
    v2 = vertices;
    do {
      if (v2->ear) {
        /* Ear found. Fill variables. */
        v3 = v2->next; v4 = v3->next;
        v1 = v2->prev; v0 = v1->prev;

        /* (v1,v3) is a diagonal */
        PrintDiagonal( v1, v3 );

        /* Update earity of diagonal endpoints */
        v1->ear = Diagonal( v0, v3 );
        v3->ear = Diagonal( v1, v4 );

        /* Cut off the ear v2 */
        v1->next = v3;
        v3->prev = v1;
        vertices = v3;      /* In case the head was v2. */
        n--;
        break;     /* out of inner loop; resume outer loop */
      }/* end if ear found */
      v2 = v2->next;
    } while ( v2 != vertices );
  } /* end outer while loop */
}
```

Code 1.14 `Triangulate`.

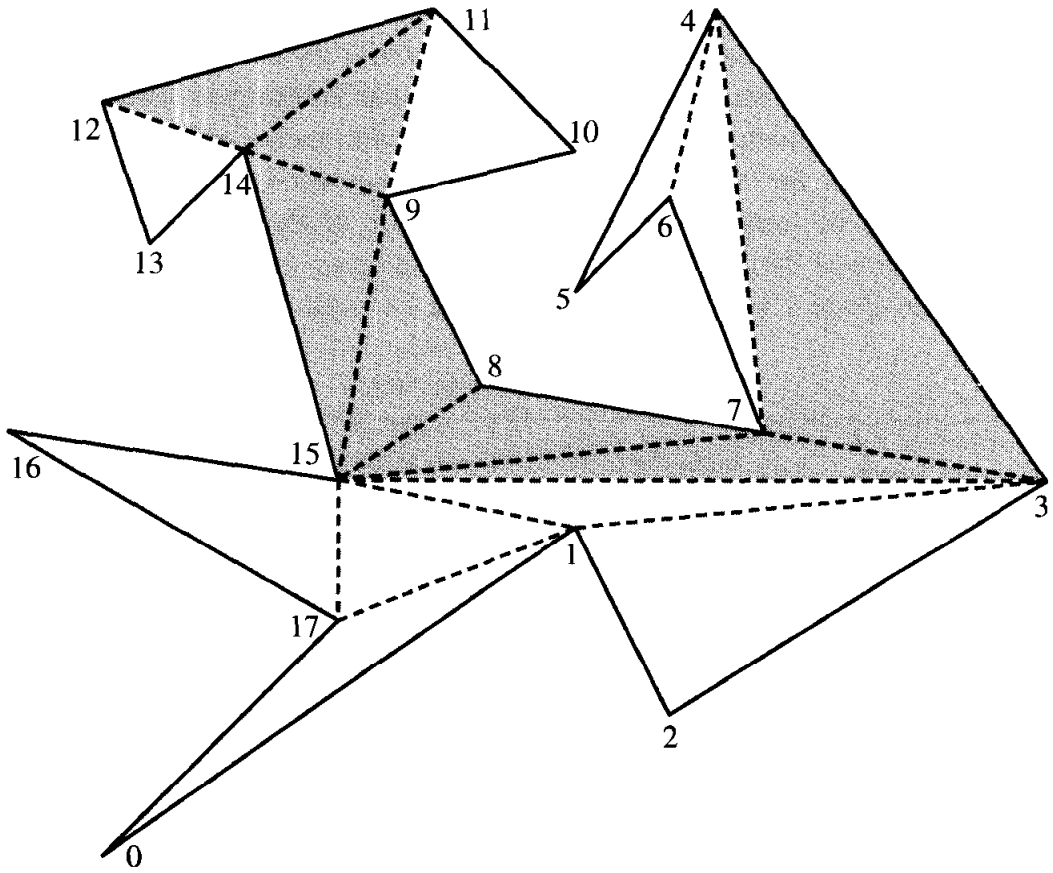


FIGURE 1.27 A polygon of 18 vertices and the triangulation produced by `Triangulate`. The dark subpolygon is the remainder after the 9th diagonal (15, 3) is output. Vertex coordinates are displayed in Table 1.1.

1.6.6. Example

Figure 1.27 shows a polygon and the triangulation produced by the simple main program (Code 1.15). The code for reading and printing is straightforward and will not be shown here.[22]

```
main()
{
  ReadVertices();
  PrintVertices();
  Triangulate();
}
```

Code 1.15 `main`.

We now walk through the output of the diagonals for this example, displayed in Table 1.2. v_0 is an ear tip, so the first diagonal output is (17, 1). v_1 is not an ear tip, so the v2 pointer moves to v_2, which is a tip, printing the diagonal (1, 3) next. Neither v_3

[22]See the Preface for how to obtain the full code.

Table 1.1. Vertex coordinates for the polygon shown in Figure 1.27.

i	(x, y)	i	(x, y)
0	(0, 0)	9	(6, 14)
1	(10, 7)	10	(10, 15)
2	(12, 3)	11	(7, 10)
3	(20, 8)	12	(0, 16)
4	(13, 17)	13	(1, 13)
5	(10, 12)	14	(3, 15)
6	(12, 14)	15	(5, 8)
7	(14, 9)	16	(−2, 9)
8	(8, 10)	17	(5, 5)

nor v_4 is an ear tip, so it is not until v_5 is reached that the next diagonal, (4, 6), is output. The segment v_3v_8 is collinear with v_7, so the next ear detected is not until v_{10}. The dark-shaded subpolygon in Figure 1.27 shows the remaining polygon after the (15, 3) diagonal (the 9th) is output. Another collinearity, v_9 with $(v_{11}v_{15})$, prevents v_9 from being an ear after the (15, 9) diagonal is cut.

1.6.7. Analysis

We now analyze the time complexity of the algorithm. `EarInit` costs $O(n^2)$, as previously mentioned. The outer loop of `Triangulate` iterates as many times as there are diagonals, $n - 3 = O(n)$. The inner search-for-an-ear loop is also $O(n)$, potentially checking every vertex. The work inside the inner loop is $O(n)$: Each of the two calls to `Diagonal` could, in the worst case, loop over the entire polygon to verify that the diagonal is not blocked. Naively we have then a time complexity of $O(n^3)$, falling short of the promised $O(n^2)$. A closer analysis will show that $O(n^2)$ is the correct bound for `Triangulate` after all.

Consider the example in Figure 1.28. After v_0 is deleted, the inner loop searches past $v_1, \ldots, v_6$ before reaching the next ear tip v_7. Then it must search past $v_8, \ldots, v_{12}$ before finding the ear tip v_{13}. This example shows that indeed the inner loop might iterate $\Omega(n)$ times before finding an ear. But notice that the two $O(n)$ `Diagonal` calls within the loop are only invoked once an ear is found – they are not called in each iteration. Thus although the superficial structure of the code suggests a complexity of $n \times n \times n = O(n^3)$, it is actually $n \times (n + n) = O(n^2)$.

Although further slight improvements are possible (Exercise 1.6.8[4]), lowering the asymptotic time complexity below quadratic requires rather different approaches, which are discussed in the next chapter.

1.6.8. Exercises

1. *Repeated intersection tests* [programming]. `Triangulate` (Code 1.14) often checks for the same segment/segment intersections. Modify the code so that you can determine how many unnecessary segment/segment intersection tests are made. Test it on Figure 1.27.

Table 1.2. The columns show the order in which the diagonals, specified as pairs of endpoint indices, are output.

Order	Diagonal Indices	Order	Diagonal Indices
1	(17, 1)	10	(3, 7)
2	(1, 3)	11	(11, 14)
3	(4, 6)	12	(15, 7)
4	(4, 7)	13	(15, 8)
5	(9, 11)	14	(15, 9)
6	(12, 14)	15	(9, 14)
7	(15, 17)		
8	(15, 1)		
9	(15, 3)		

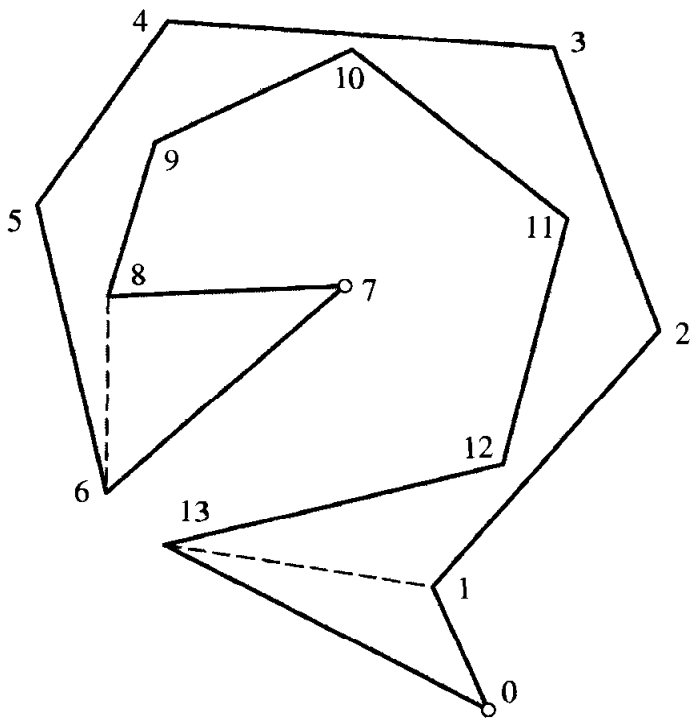


FIGURE 1.28 An example that forces the inner ear loop to search extensively for the next ear.

2. *Convex polygons* [easy]. Analyze the performance of `Triangulate` when run on a convex polygon.
3. *Spiral.* Continue the analysis of Figure 1.28: Does `Triangulate` continue to traverse the boundary in search of an ear? More specifically, if the polygon has n vertices, how many complete circulations of the boundary will the pointer `v2` execute before completion?
4. *Ear list* [programming]. The inner loop search of `Triangulate` can be avoided by linking the ear tips into their own (circular) list, linking together those vertices v for which `v->ear == TRUE` with pointers `nextear` and `prevear` in the vertex structure. Then the ear for the next iteration can be found by moving to the next ear on this list beyond the one just clipped. Implement this improvement, and see if its speedup is discernible on an example (perhaps one akin to Figure 1.28).

5. *Center of gravity.* Design an algorithm to compute the center of gravity of a polygon, assuming that it is cut from a material of uniform density. The center of gravity is a point, which can be treated as a vector. The center of gravity of a triangle is at its *centroid*, whose coordinates happen to be at the average of the coordinates of the triangle's vertices. The center of gravity $\gamma(S)$ of any set S that is the disjoint union of sets A and B is the weighted sum of the centers of gravity of the two pieces. Let $w(S) = w(A) + w(B)$ be the weight of S. Then

$$\gamma(S) = \frac{w(A)\gamma(A) + w(B)\gamma(B)}{w(S)}.$$

Here the weight of each triangle is its area under the uniform density assumption.

2

Polygon Partitioning

In this short chapter we explore other types of polygon partitions: partitions into monotone polygons (Section 2.1), into trapezoids (Section 2.2), into "monotone mountains" (Section 2.3), and into convex polygons (Section 2.5). Our primary motivation is to speed up the triangulation algorithm presented in the previous chapter, but these partitions have many applications and are of interest in their own right. One application of convex partitions is character recognition: Optically scanned characters can be represented as polygons (sometimes with polygonal holes) and partitioned into convex pieces, and the resulting structures can be matched against a database of shapes to identify the characters (Feng & Pavlidis 1975). In addition, because so many computations are easier on convex polygons (intersection with obstacles or with light rays, finding the distance to a line, determining if a point is inside), it often pays to first partition a complex shape into convex pieces.

This chapter contains no implementations (but suggests some as exercises).

2.1. MONOTONE PARTITIONING

We presented an $O(n^2)$ triangulation algorithm in Section 1.4. Further improvements will require organizing the computation more intelligently, so that each diagonal can be found in sublinear time.[1] There are now many algorithms that achieve $O(n \log n)$ time, averaging $O(\log n)$ work per diagonal.[2] The first was due to Garey, Johnson, Preparata & Tarjan (1978). Although one might expect an $O(n \log n)$ algorithm to find each diagonal by an $O(\log n)$ binary search, that is not in fact the way their algorithm works. Rather they first partition the polygon into simpler pieces, in $O(n \log n)$ time, and then triangulate the pieces in linear time. The pieces are called "monotone," a concept first introduced and exploited by Lee & Preparata (1977). It will develop that partitions into monotone polygons will have several other uses aside from triangulation, so their exploration is a worthwhile pursuit.

We will only sketch the $O(n \log n)$ algorithm based on monotone partitioning, but return in Section 2.3 to detail a closely related algorithm based on partitions into "monotone mountains."

We first define monotonicity, then show how to triangulate monotone polygons in linear time, and finally describe how to partition a polygon into monotone pieces.

[1] The technical notation for sublinear time is $o(n)$ time.

[2] Throughout the text, all logarithms are to the base 2. But since the big-O notation absorbs constants, the base of the logarithm is irrelevant when inside an $O(\ )$-expression.

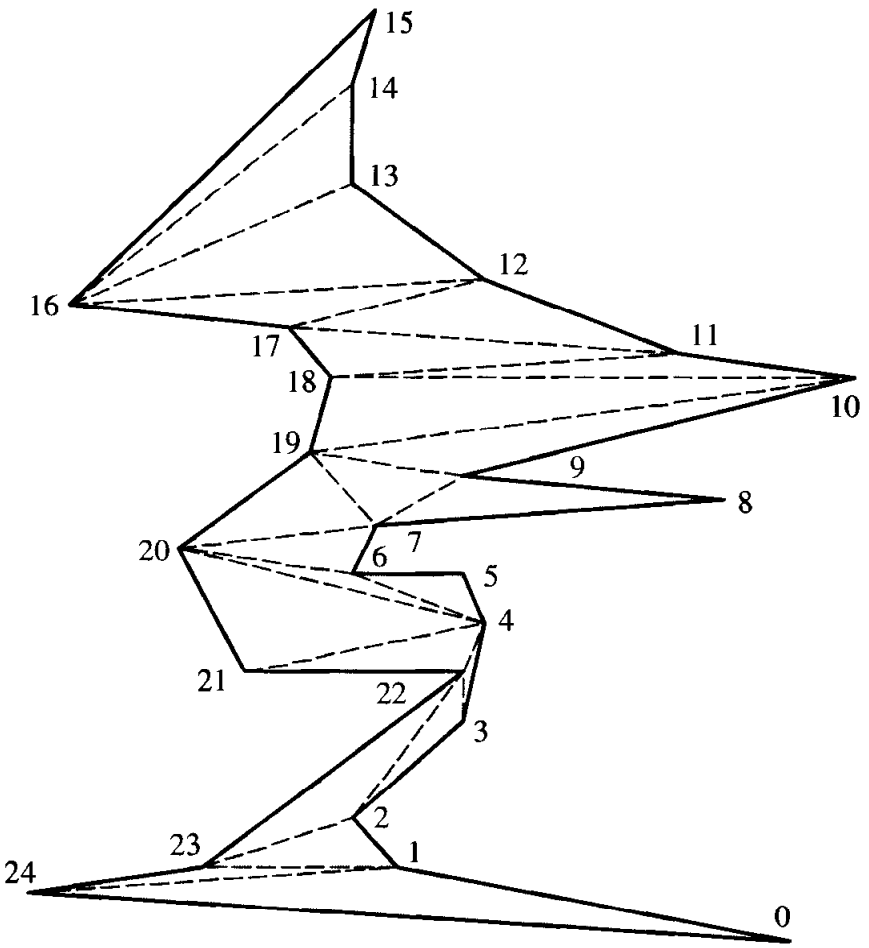


FIGURE 2.1 A polygon monotone with respect to the vertical.

2.1.1. Monotone Polygons

Monotonicity is defined with respect to a line. First we define monotonicity of polygonal chains. A polygonal chain C is *strictly monotone* with respect to L' if every line L orthogonal to L' meets C in at most one point (i.e., $L \cap C$ is either empty or a single point). A chain is *monotone* if $L \cap C$ has at most one connected component: It is either empty, a single point, or a single line segment.[3] These chains are "monotone" in the sense that a traversal of C projects to a monotone sequence on L': No reversals occur.

A polygon P is said to be *monotone* with respect to a line L if ∂P can be split into two polygonal chains A and B such that each chain is monotone with respect to L. The two chains share a vertex at either end. A polygon monotone with respect to the vertical is shown in Figure 2.1. The two monotone chains are $A = (v_0, \ldots, v_{15})$ and $B = (v_{15}, \ldots, v_{24}, v_0)$. Neither chain is strictly monotone, because edges v_5v_6 and $v_{21}v_{22}$ are horizontal. Some polygons are monotone with respect to several lines; and some polygons are not monotone with respect to any line.

[3]This definition differs from some others in the literature (e.g., from that of Preparata & Shamos (1985, p. 49)) in that here monotone chains need not be strictly monotone.

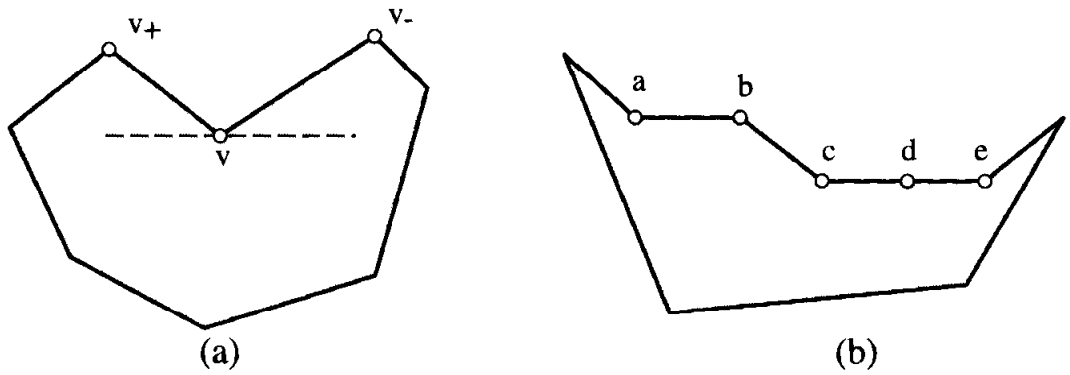


FIGURE 2.2 Interior cusps: (a) v_+ and v_- are both above v; (b) a, c, and e are interior cusps; b and d are not.

Properties of Monotone Polygons

Many algorithms that are difficult for general polygons are easy for monotone polygons, primarily because of this key property: The vertices in each chain of a monotone polygon are sorted with respect to the line of monotonicity. Let us fix the line of monotonicity to be the vertical y axis. Then the vertices can be sorted by y coordinate in linear time: Find a highest vertex, find a lowest, and partition the boundary into two chains. The vertices in each chain are sorted with respect to y. Two sorted lists of vertices can be merged in linear time to produce one list sorted by y.

There is a simple local structural feature that characterizes monotonicity. Essentially it says that a polygon is monotone if it is monotone in the neighborhood of every vertex. This can form the basis of an algorithm to partition a polygon into monotone pieces, by cutting at the local nonmonotonicities.

Define an *interior cusp* of a polygon as a reflex vertex v whose adjacent vertices v_- and v_+ are either both at or above, or both at or below, v. See Figure 2.2. Recall that a reflex vertex has internal angle strictly greater than π, so it is not possible for an interior cusp to have both adjacent vertices with the same y coordinate as v. Thus d in Figure 2.2(b) is not an interior cusp. The characterization is this simple lemma:

Lemma 2.1.1. *If a polygon P has no interior cusps, then it is monotone.*

Despite the naturalness of this lemma, a proof requires care.[4] It is perhaps not obvious that it cannot be strengthened to the claim that the lack of interior cusps implies strict monotonicity (Exercise 2.2.3[2]). We will not pause to prove this lemma, but rather continue with the high-level sketch. We will use the lemma in Section 2.2 to partition a polygon into monotone pieces.

2.1.2. Triangulating a Monotone Polygon

Because monotone polygons are so restricted, one might hope that their triangulations are similarly special – that the triangulation dual is always a path, or every diagonal connects the two monotone chains. Figure 2.1 shows that neither of these hypotheses

[4]See Lee & Preparata (1977) or O'Rourke (1994, pp. 54–5).

hold; see also Exercise 2.3.4[1]. Nevertheless, the intuition that these shapes are so special that they must be easy to triangulate is valid: Any monotone polygon (whose direction of monotonicty is given) may be triangulated in linear time.

Here is a hint of an algorithm. First sort the vertices from top to bottom (in linear time). Then cut off triangles from the top in a "greedy" fashion (this is a technical algorithms term indicating in this instance that at each step the first available triangle is removed). So the algorithm is: For each vertex v, connect v to all the vertices above it and visible via a diagonal, and remove the top portion of the polygon thereby triangulated; continue with the next vertex below v.

One can show that at any iteration, $v \in A$ is being connected to a chain of reflex vertices above it in the other chain B. For example, v_{16} is connected to (v_{14}, v_{13}, v_{12}) in the first iteration for the example in Figure 2.1. As a consequence, no visibility check is required to determine these diagonals – they can be output immediately. The algorithm can be implemented with a single stack holding the reflex chain above. Between the linear sorting and this simple data structure, $O(n)$ time overall is achieved.[5]

2.2. TRAPEZOIDALIZATION

Knowing that monotone polygons may be triangulated quickly, it becomes an interesting problem to partition a polygon into monotone pieces quickly. We do this via yet another intermediate partition, which is itself of considerable interest, and which we will use later in Section 7.11: a partition into trapezoids. This partition was introduced by Chazelle & Incerpi (1984) and Fournier & Montuno (1984) as the key to triangulation. This partition will differ from those considered previously in that we will not restrict the partitioning segments to be diagonals.

A *horizontal trapezoidalization* of a polygon is obtained by drawing a horizontal line through every vertex of the polygon. More precisely, pass through each vertex v the maximal (open) horizontal segment s such that $s \subset P$ and $s \cap \partial P = v$. Thus s represents clear lines of sight from v left and right. It may be that s is entirely to one side or the other of v; and it may be that $s = v$. An example is shown in Figure 2.3. To simplify the exposition we will only consider polygons whose vertices have unique y coordinates: No two vertices lie on a horizontal line.[6]

A *trapezoid* is a quadrilateral with two parallel edges. One can view a triangle as a degenerate trapezoid, with one of the two parallel edges of zero length. Call the vertices through which the horizontal lines are drawn *supporting vertices*.

Let P be a polygon with no two vertices on a horizontal line. Then in a horizontal trapezoidalization, every trapezoid has exactly two supporting vertices, one on its upper edge and one on its lower edge. The connection between trapezoid partitions and monotone polygons is this: If a supporting vertex is on the interior of an upper or lower

[5]For more detailed expositions, see Garey et al. (1978), O'Rourke (1994, pp. 55–9), or de Berg, van Kreveld, Overmars & Schwarzkopf (1997, pp. 55–8).

[6]Although it is not obvious, this assumption involves no true loss of generality. It suffices to sort points *lexicographically*: For two points with the same y coordinate, treat the one with smaller x coordinate as lower (Seidel 1991).

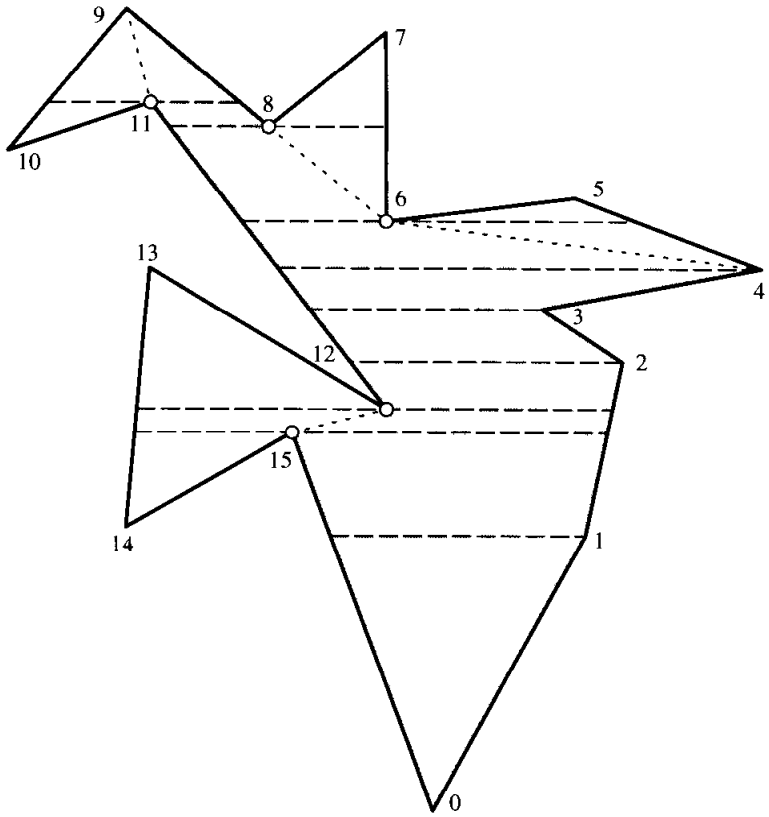


FIGURE 2.3 Trapezoidalization. Dashed lines show trapezoid partition lines; dotted diagonals resolve interior cusps (circled). The shaded polygon is one of the resulting monotone pieces.

trapezoid edge, then it is an interior cusp. If every interior supporting vertex v is connected to the opposing supporting vertex of the trapezoid v supports, downward for a "downward" cusp and upward for an "upward" cusp, then these diagonals partition P into pieces monotone with respect to the vertical. This follows from Lemma 2.1.1, since every interior cusp is removed by these diagonals. For example, the downward cusp v_6 in Figure 2.3 is resolved with the diagonal v_6v_4; the upward cusp v_{15} is resolved by connecting to v_{12} (which happens to be a downward cusp); and so on.

Now that we see that a trapezoidalization yields a monotone partition directly, we concentrate on drawing horizontal chords through every vertex of a polygon.

2.2.1. Plane Sweep

The algorithm we use to construct a trapezoidalization depends on a technique called a "plane sweep" (or "sweep line"), which is useful in many geometric algorithms (Nievergelt & Preparata 1982). The main idea is to "sweep" a line over the plane, maintaining some type of data structure along the line. The sweep stops at discrete "events" where processing occurs and the data structure is updated. For our particular problem, we sweep a horizontal line L over the polygon, stopping at each vertex. This requires sorting the vertices by y coordinate, and since the polygon is general, this requires $O(n \log n)$ time.[7]

[7] Sorting has time complexity $\Theta(n \log n)$: It can be accomplished in $O(n \log n)$ time, but no faster. See Knuth (1973).

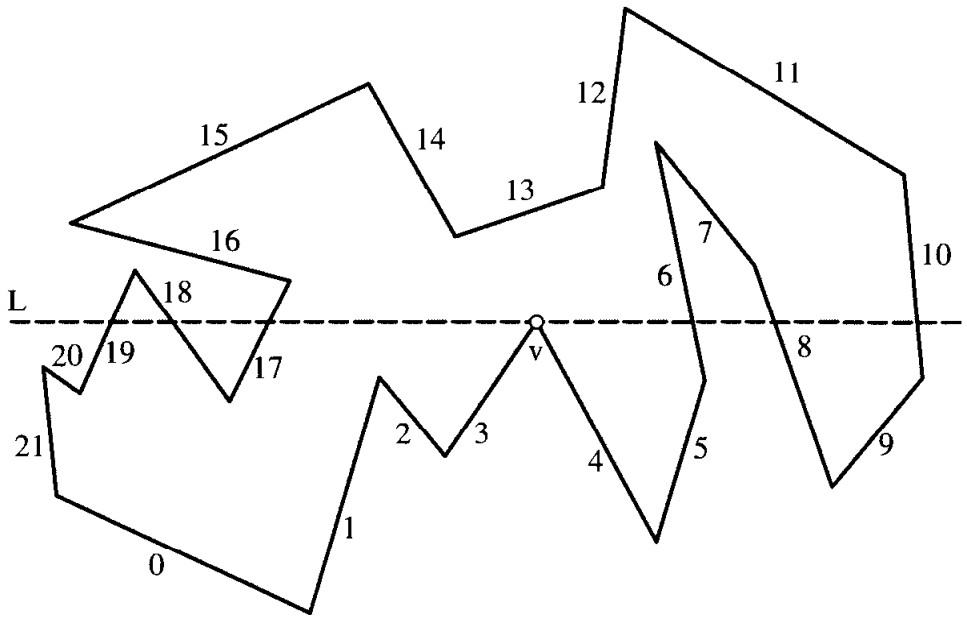


FIGURE 2.4 Plane sweep. Labels index edges.

The processing required at each event vertex v is finding the edge immediately to the left and immediately to the right of v along L. To do this efficiently, a sorted list $\mathcal{L}$ of polygon edges pierced by L is maintained at all times. For example, for the sweep line in the position shown in Figure 2.4, $\mathcal{L} = (e_{19}, e_{18}, e_{17}, e_6, e_8, e_{10})$.

Suppose this list $\mathcal{L}$ is available. How can we determine that v lies between e_{17} and e_6 in the figure? Let us assume that e_i is a pointer to an edge of the polygon, from which the coordinates of its endpoints can be retrieved easily. Suppose the vertical coordinate of v (and therefore L) is y. Knowing the endpoints of e_i, and y, we can compute the x coordinate of the intersection between L and e_i. So we can determine v's position in the list by computing the x coordinates of where L pierces each edge at height y.

This would take time proportional to the length of $\mathcal{L}$ (which is $O(n)$) if done by a naive search from left to right; but if we store the list in an efficient data structure, such as a height-balanced tree, then the search will only require $O(\log n)$ time. Since this search occurs once per each event, the total cost over the entire plane sweep is $O(n \log n)$.

It remains to show that it is possible to maintain the data structure at all times, and in time $O(n \log n)$. This is easy as long as the data structure supports $O(\log n)$-time insertions and deletions, as do, for example, height-balanced or 2-3 or red-black trees.[8] We now detail the updates at each event, assuming a downward sweeping line.

There are three possible types of event, illustrated in Figure 2.5. Let v fall between edges a and b on L, and let v be shared by edges c and d.

1. c is above L and d below. Then delete c from $\mathcal{L}$ and insert d:

$$(\ldots, a, c, b, \ldots) \Rightarrow (\ldots, a, d, b, \ldots).$$

2. Both c and d are above L. Then delete both c and d from $\mathcal{L}$:

$$(\ldots, a, c, d, b, \ldots) \Rightarrow (\ldots, a, b, \ldots).$$

[8] See, e.g., Aho, Hopcroft & Ullman (1983, pp. 169–80) or Cormen, et al. (1990, Chap. 14).

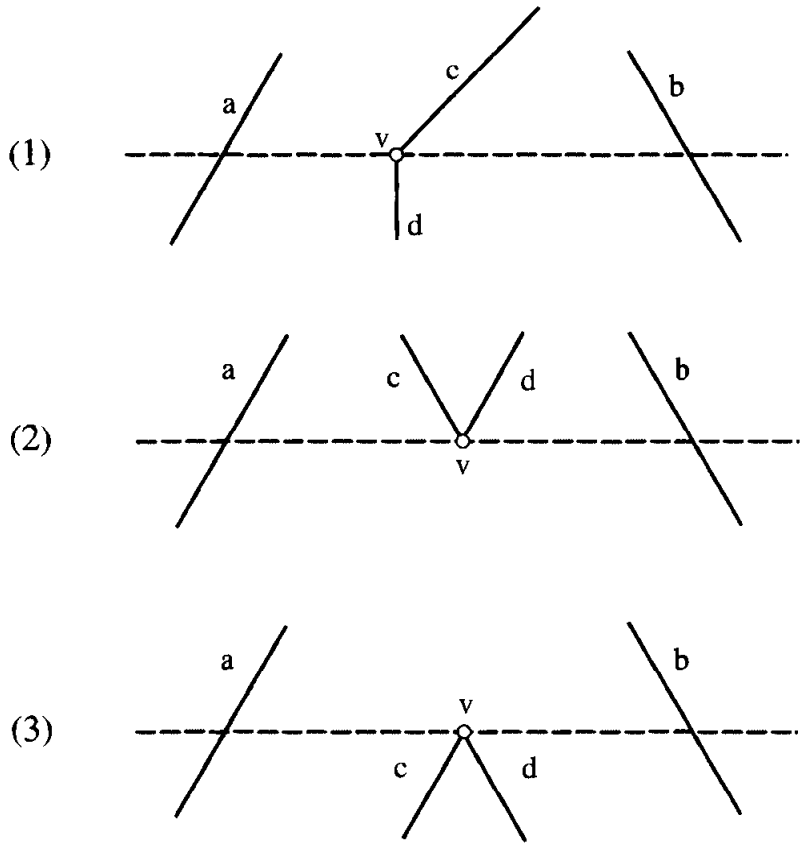


FIGURE 2.5 Sweep line events: (1) replace c by d; (2) delete c and d; (3) insert c and d.

3. Both c and d are below L. Then insert both c and d into $\mathcal{L}$:

$$(\ldots, a, b, \ldots) \Rightarrow (\ldots, a, c, d, b, \ldots).$$

Returning to Figure 2.4, we see that the list $\mathcal{L}$ of edges pierced by L is initially empty, when L is above the polygon, and then follows this sequence as it passes each event vertex:

$$(e_{12}, e_{11})$$
$$(e_{15}, e_{14}, e_{12}, e_{11})$$
$$(e_{15}, e_{14}, e_{12}, e_6, e_7, e_{11})$$
$$(e_{15}, e_{14}, e_{13}, e_6, e_7, e_{10})$$
$$(e_{16}, e_{14}, e_{13}, e_6, e_7, e_{10})$$
$$(e_{16}, e_6, e_7, e_{10})$$
$$(e_{16}, e_6, e_8, e_{10})$$
$$(e_{19}, e_{18}, e_{16}, e_6, e_8, e_{10})$$
$$(e_{19}, e_{18}, e_{17}, e_6, e_8, e_{10}) .$$

The final list corresponds to the position of L shown in the figure.

2.2.2. Triangulation in $O(n \log n)$

Leaving out the remaining (mainly data structure) details, we summarize the $O(n \log n)$ algorithm for triangulating a polygon in Algorithm 2.1.

Algorithm: POLYGON TRIANGULATION: MONOTONE PARTITION
Sort vertices by y coordinate.
Perform plane sweep to construct trapezoidalization.
Partition into monotone polygons by connecting from interior cusps.
Triangulate each monotone polygon in linear time.

Algorithm 2.1 $O(n \log n)$ polygon triangulation.

2.2.3. Exercises

1. *Monotone with respect to a unique direction.* Can a polygon be monotone with respect to precisely one direction?
2. *Interior cusps.* Construct a monotone but not strictly monotone polygon that has no interior cusps, thereby showing that Lemma 2.1.1 cannot be strengthened to the claim that the lack of interior cusps implies strict monotonicity.
3. *Several vertices on a horizontal.* Extend the trapezoid partition algorithm to polygons that may have several vertices on a horizontal line.
4. *Sweeping a polygon with holes.* Sketch an algorithm for triangulating a polygon with holes (one outer polygon P containing several polygonal holes) via plane sweep. The diagonals should partition the interior of P outside each hole. Express the complexity as a function of the total number of vertices n.

2.3. PARTITION INTO MONOTONE MOUNTAINS

A minor variation of the algorithm just described is simpler. The idea is to again start with a trapezoidalization of P, but add more than just cusp-to-cusp diagonals, partitioning P into pieces we will call monotone mountains. These shapes are even easier to triangulate.

2.3.1. Monotone Mountains

A *monotone mountain* is a monotone polygon with one of its two monotone chains a single segment, the *base*. If the direction of monotonicity is horizontal, such a polygon resembles a mountain range; see Figure 2.6.[9] Note that both endpoints of the base must be convex, as otherwise one of the chains would contain more than one segment.

The following lemma makes triangulation of such polygons easy.

Lemma 2.3.1. *Every strictly convex vertex of a monotone mountain M, with the possible exception of the base endpoints, is an ear tip.*

[9]Fournier & Montuno (1984) call these shapes *unimonotone polygons*.

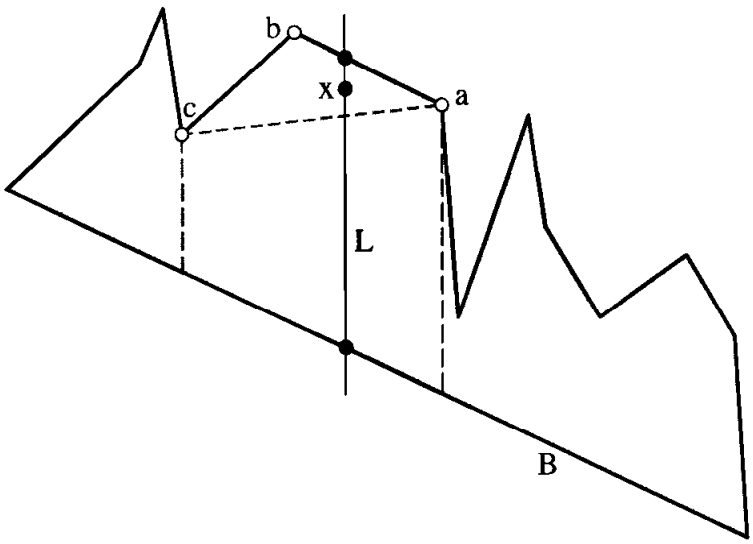


FIGURE 2.6 A monotone mountain with base B; b is an ear tip.

Proof. Let a, b, c be three consecutive vertices of M, with b a strictly convex vertex not an endpoint of the base B. Let the direction of monotonicity be horizontal. We aim to prove that ac is a diagonal, by contradiction.

Assume that ac is not a diagonal. Then by Lemma 1.5.1, either it is exterior in the neighborhood of an endpoint or it intersects ∂M.

1. Suppose first that ac is locally exterior in the neighborhood of endpoint a in Figure 2.6 (endpoint c is symmetrical and need not be considered separately). If a is not also an endpoint of B (as illustrated), then the two incident edges are left and right of a, with M below. To be exterior, ac must be locally above ab, which is inconsistent with the assumption that b is convex. If a is the right endpoint of B, then either ac is locally above ab, leading to the same contradiction, or it is locally below B. In the latter case, it could not connect to c, which must lie above B. We may conclude that ac is locally interior to M at each endpoint.
2. Assume therefore that ac intersects ∂M. This would require a reflex vertex x to be interior to $\triangle abc$ (cf. Figure 1.12). Because x is interior, it cannot lie on the chain $C = (a, b, c)$; and it cannot lie on B, which is a single segment with convex endpoints. Thus a vertical line L through x meets ∂M in at least three points: $C \cap L$, $B \cap L$, and x. This contradicts the definition of a monotone polygon. □

This lemma does not hold for monotone polygons; for example, v_3 in Figure 2.1 is convex but not an ear tip. Note that the exclusion of B's endpoints cannot be removed from the preconditions of the lemma: Neither endpoint in Figure 2.6 is an ear tip.

2.3.2. Triangulating a Monotone Mountain

Lemma 2.3.1 yields a nearly trivial linear algorithm for triangulating a monotone mountain: Find a convex vertex not on the base, clip off the associated ear, and repeat.

To ensure that this algorithm runs in linear time requires only that (a) the base be identified in linear time and (b) that the "next" convex vertex be found without a search, in constant time. The former is easy: The base endpoints are extreme along the direction of

monotonicity. If this direction is horizontal, simply search for the leftmost and rightmost vertices.[10] Once the base is identified, its endpoints can be avoided in the ear-clipping phase.

Achieving (b) is similar to the task faced in Section 1.4 when the ear tip status of a and c needed updating after clipping ear $\triangle abc$. Here instead we need to update the convexity status, which by Lemma 2.3.1 implies the ear tip status. This is easily accomplished by storing with each vertex its internal angle, and subtracting from a and c's angles appropriately upon removal of $\triangle abc$. One issue remains, however. The `Triangulate` code (Code 1.14) tolerated a linear inner-loop search for the next ear, for there we only sought $O(n^2)$ behavior. But now our goal is $O(n)$. To find the next convex vertex without a search requires following Exercise 1.6.8[4] in linking the convex vertices into a (circular) list and updating the list with each ear clip appropriately. Then as long as this list is nonempty, its "first" element can be chosen immediately for each clipping.

The algorithm is summarized in the pseudocode displayed as Algorithm 2.2.

Algorithm: TRIANGULATION OF MONOTONE MOUNTAIN
Identify the base edge.
Initialize internal angles at each nonbase vertex.
Link nonbase strictly convex vertices into a list.
while list nonempty do
    For convex vertex b, remove $\triangle abc$.
    Output diagonal ac.
    Update angles and list.

Algorithm 2.2 Linear-time triangulation of a monotone mountain.

2.3.3. Adding Diagonals to Trapezoidalization

Now we address how to convert a trapezoidalization to a partition into monotone mountains. The monotone mountains will be turned on their sides, with a vertical direction of monotonicity. We first motivate the key idea, and then we prove that it works.

Consider building a monotone mountain from trapezoids abutting on a particular base edge, for example $B = v_{11}v_{12}$ in Figure 2.7. Use the notation $T(i, j)$ to represent the trapezoid with support vertices v_i and v_j, below and above respectively. $T(12, 2)$ is based on B but must be cut by the diagonal $v_{12}v_2$ to ensure that the v_{12} endpoint is convex. $T(2, 3)$ and $T(3, 4)$ may be included in their entirety. $T(4, 6)$ must be cut by the diagonal v_4v_6 to separate off the nonmonotonicity at v_6, and similarly $T(6, 8)$ must be cut by v_6v_8. Finally, $T(8, 11)$ must be cut by v_8v_{11} to ensure convexity at v_{11}. The resulting union (shown shaded in the figure) is a monotone mountain.

Note that we have cut a trapezoid by a diagonal between its supporting vertices in exactly those cases where those vertices do not lie on the same side of the trapezoid. This suggests the following lemma:

[10] We will see in Chapter 7 (Section 7.9) that these extremes can be found in $O(\log n)$ time, but we do not need such sophistication here.

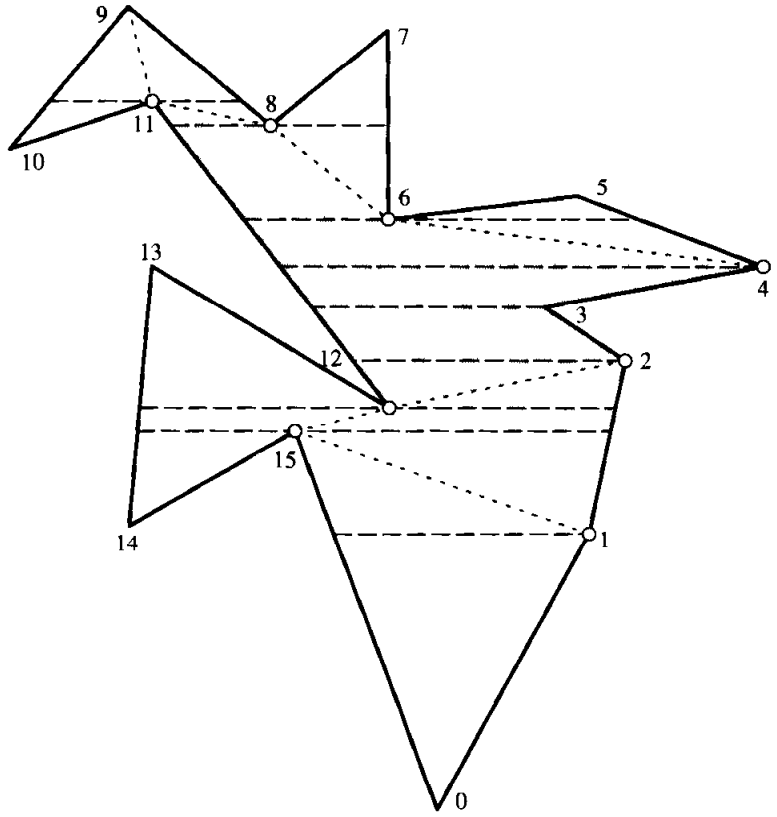


FIGURE 2.7 A partition into monotone mountains.

Lemma 2.3.2. *In a trapezoidalization of a polygon P, connecting every pair of trapezoid-supporting vertices that do not lie on the same (left/right) side of their trapezoid partitions P into monotone mountains.*

Proof. We may observe at once that the pieces must be monotone, because an interior cusp does not lie on either the left or the right side of the trapezoid it supports, so it is always the endpoint of a diagonal that resolves it. Thus Lemma 2.1.1 guarantees that the pieces of the partition are monotone. It only remains to prove that each piece has one chain that is a single segment.

Suppose to the contrary that both monotone chains A and B of one piece Q of the partition each contain at least two edges. Let z be the topmost vertex of Q, adjacent on $\partial Q = A \cup B$ to vertices $a \in A$ and $b \in B$, with b below a. See Figure 2.8. In order for B to contain more than just the edge zb, b cannot be the endpoint of a partition diagonal from above. But consider the trapezoid $T(b, c)$ supported from below by b. Its upper supporting vertex c cannot lie on zb, for c must lie at or below a (it could be that $a = c$). Thus c is not on the same side of $T(b, c)$ as b, and the diagonal cb is included in the partition, contradicting the assumption that B extends below b. □

Figure 2.7 shows the result of applying this lemma to the example used previously in Figure 2.3. Three more diagonals are needed to achieve this finer partition, resulting in eight monotone mountains compared to five monotone pieces.

We now have an outline of a second $O(n \log n)$ triangulation algorithm: After trapezoidalization, add diagonals per Lemma 2.3.2, and triangulate each piece with Algorithm 2.2.

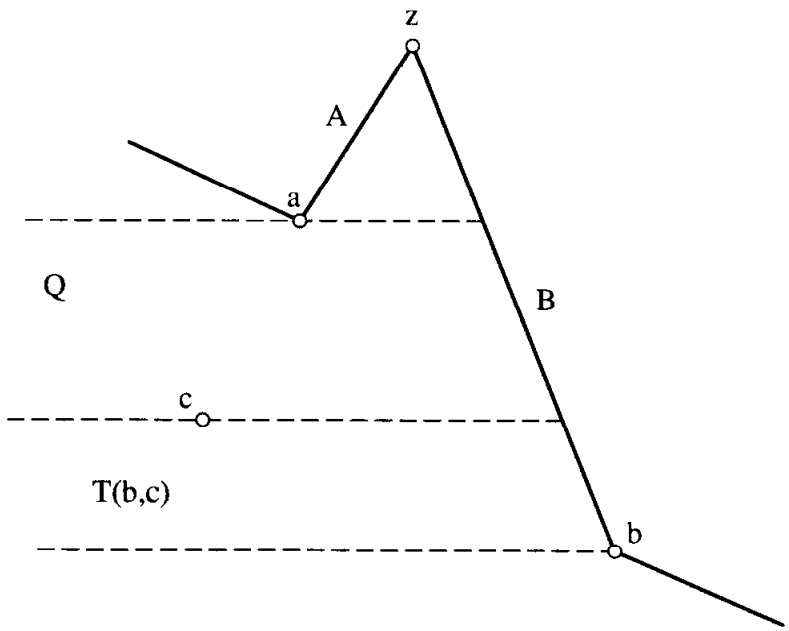


FIGURE 2.8 Proof of Lemma 2.3.2: Diagonal *cb* must be present.

We leave designing data structures to permit the efficient extraction of the monotone mountain pieces to Exercise 2.3.4[4].

2.3.4. Exercises

1. *Monotone duals.* Prove that every binary tree can be realized as the triangulation dual of a monotone mountain. (Cf. Exercise 1.2.5[2].)
2. *Random monotone mountains* [programming]. Develop code to generate "random" monotone mountains. Generating random polygons, under natural definitions of "random," is an open problem, but monotone mountains are special enough to make it easy in this case.
3. *Triangulating monotone mountains* [programming]. Implement Algorithm 2.2.
4. *Trapezoid data structure.* Design a data structure for a trapezoidalization of a polygon P augmented by a set of diagonals that permits extraction of the subpolygons of the resulting partition in time proportional to their size.
5. *Polygon $\Rightarrow$ Convex quadrilaterals.* Prove or disprove: Every polygon with an even number of vertices may be partitioned by diagonals into convex quadrilaterals.
6. *Polygon $\Rightarrow$ Quadrilaterals.* Prove or disprove: Every polygon with an even number of vertices may be partitioned by diagonals into quadrilaterals.
7. *Orthogonal pyramid $\Rightarrow$ Convex quadrilaterals.* An *orthogonal polygon* is a polygon in which each pair of adjacent edges meets orthogonally (Exercise 1.2.5[5]). Without loss of generality, one may assume that the edges alternate between horizontal and vertical.

   An *orthogonal pyramid* P is an orthogonal polygon monotone with respect to the vertical, that contains one horizontal edge h whose length is the sum of the lengths of all the other horizontal edges. Thus P is monotone with respect to both the vertical and the horizontal; in fact it is a monotone mountain with respect to the horizontal. P consists of two "staircases" connected to h, as shown in Figure 2.9.

   a. Prove that an orthogonal pyramid may be partitioned by diagonals into *convex* quadrilaterals.

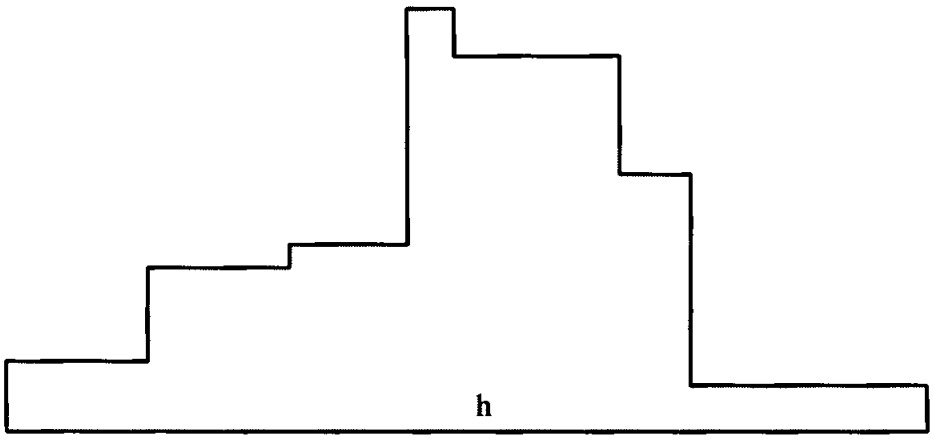


FIGURE 2.9 Orthogonal pyramid.

b. Design an algorithm for finding such a partition. Try for linear-time complexity. Describe your algorithm in pseudocode, at a high level, ignoring data structure details and manipulations.

8. *Orthogonal polygon* $\Rightarrow$ *Convex quadrilaterals.* Can every orthogonal polygon be partitioned by diagonals into convex quadrilaterals? Explore this question enough to form a conjecture.

2.4. LINEAR-TIME TRIANGULATION

Quadratic triangulation algorithms have been implicit in proofs since at least 1911(Lennes 1911).[11] The $O(n \log n)$ algorithm described in Section 2.1 was one of the early achievements of computational geometry, having been published in 1978, just three years after Shamos named the field in his thesis. Soon the question of whether $O(n \log n)$ is optimal for triangulation became the outstanding open problem in computational geometry, fueling an amazing variety of clever algorithms. Algorithms were found that succeeded in breaking the $n \log n$ barrier, but only in special cases; see Table 2.1 for a sampling. The worst case remained $O(n \log n)$.

Finally, after a decade of effort, Tarjan & Van Wyk (1988) discovered an $O(n \log \log n)$ algorithm. This breakthrough led to a flurry of activity, including two $O(n \log^* n)$ algorithms:[12] one "randomized" and one for polygons with appropriately bounded integer coordinates.

Finally, Chazelle constructed a remarkable $O(n)$ worst-case algorithm in 1991, ending a thirteen-year pursuit by the community. It would take us too far afield to describe the algorithm in detail, but I will offer a rough sketch.

The main structure computed by the algorithm is a *visibility map*, which is a generalization of a trapezoidalization to drawing horizontal chords toward both sides of each vertex in a polygonal chain. When the chain is a polygon, this amounts to extending the chords exterior as well as interior to the polygon. As Chazelle explains it, his algorithm

[11] I depend here on the historical research of Toussaint (1985*a*).

[12] $\log^* n$ is the number of times the log must be iterated to reduce n to 1 or less. Thus for $n = 2^{(2^{16})} \approx 10^{19728}$, $\log^* 2^{(2^{16})} = 5$, because $\log 2^{(2^{16})} = 2^{16}$, $\log 2^{16} = 16$, $\log 2^4 = 4$, $\log 2^2 = 2$, and $\log 2 = 1$. Note that $\log \log 2^{(2^{16})} = \log 2^{16} = 16$; for sufficiently large n, $\log^* n \ll \log \log n$.

Table 2.1. History of triangulation algorithms.

Year	Complexity	Reference
1911	$O(n^2)$	Lennes (1911)
1978	$O(n \log n)$	Garey et al. (1978)
1983	$O(n \log r)$, r reflex	Hertel & Mehlhorn (1983)
1984	$O(n \log s)$, s sinuosity	Chazelle & Incerpi (1984)
1988	$O(n + nt_0)$, t_0 int. triangs.	Toussaint (1990)
1986	$O(n \log \log n)$	Tarjan & Van Wyk (1988)
1989	$O(n \log^* n)$, randomized	Clarkson, Tarjan & Van Wyk (1989)
1990	$O(n \log^* n)$, bnded. ints.	Kirkpatrick, Klawe & Tarjan (1990)
1990	$O(n)$	Chazelle (1991)
1991	$O(n \log^* n)$, randomized	Seidel (1991)

mimics merge sort, a common technique for sorting by divide-and-conquer. The polygon of n vertices is partitioned into chains with $n/2$ vertices, and these into chains of $n/4$ vertices, and so on. The visibility map of a chain is found by merging the maps of its subchains. This leads to an $O(n \log n)$ time complexity.

Chazelle improves on this by dividing the process into two phases. In the first phase, only coarse approximations of the visibility maps are computed, coarse enough so that the merging can be accomplished in linear time. These maps are coarse in the sense that they are missing some chords. A second phase then refines the coarse map into a complete visibility map, again in linear time. A triangulation is then produced from the trapezoidalization defined by the visibility map as before. The details are formidable.

Although this closed the longstanding open problem, it remained open to find a simple, fast, practical algorithm for triangulating a polygon. Several candidates soon emerged, including Seidel's randomized $O(n \log^* n)$ algorithm, which we will sketch here (Seidel 1991).[13]

2.4.1. Randomized Triangulation

Seidel's algorithm follows the trapezoidalization → monotone mountains → triangulation path described in Section 2.3. His improvement is in building the trapezoidalization quickly. He builds the visibility map for a collection of segments into a "query structure" Q, a data structure that permits location of a point in its containing trapezoid in time proportional to the depth of the structure. We will describe this structure in detail in Chapter 7 (Section 7.11); for now an impressionistic view will suffice.

The depth of the structure could be $\Omega(n)$ for n segments, but if the structure is built incrementally by adding the segments in random order, then the *expected* cost of locating a point in Q is $O(\log n)$. This is the sense in which the algorithm is "randomized": It uses a coin flip to make decisions on which segment to add next. No assumptions are made that the segments themselves are randomly distributed in any sense. Such assumptions

[13]For another randomized algorithm with the same complexity, see Devillers (1992).

lead to algorithms that work well on "average-case" inputs but could perform poorly on unusual inputs. Randomized algorithms (sometimes called "Las Vegas" algorithms), in contrast, can be expected to work well on all inputs, but through an unluckly series of coin flips might perform poorly. Fortunately the probability of such an unlucky streak is often so minuscule as to be practically irrelevant.[14] The use of random sampling techniques in geometric algorithms has developed in the past decade into a key technique for creating algorithms that are both efficient and simple (Mulmuley & Schwarzkopf 1997). We will revisit this topic in Chapters 4 and 7 (Sections 4.5, 7.5, and 7.11.4).

Returning to Seidel's algorithm, we can construct the visibility map by inserting the segments in random order in $O(n \log n)$ time and $O(n)$ space, using the structure so far built to locate the endpoints of each new segment added. This results in another $O(n \log n)$ triangulation algorithm. But we have not yet used the fact that the segments form the edges of a simple polygon. This can be exploited by running the algorithm in $\log^* n$ phases. In phase i, a subset of the segments is added in random order, producing a query structure Q_i. Then the entire polygon is traced through Q_i, locating each vertex in a trapezoid of the current visibility map. In phase $i+1$, more segments are added, but the knowledge of where they were in Q_i helps locate their endpoints more quickly. This process is repeated until the entire visibility map is constructed, after which we fall back to earlier techniques to complete the triangulation. Analysis of the expected time for this algorithm, expected over all possible $n!$ segment insertion orders, shows it to be $O(n \log^* n)$. Moreover, the algorithm is relatively simple to implement.[15]

2.5. CONVEX PARTITIONING

A partition into triangles can be viewed as a special case of a partition into convex polygons. Because there is an optimal-time triangulation algorithm, there is an optimal-time convex partitioning algorithm. But triangulation is by no means optimal in the number of convex pieces.

There are two goals of partitions into convex pieces: (1) partition a polygon into as few convex pieces as possible and (2) do so as quickly as possible. The goals conflict of course. There are two main approaches. First, compromise on the number of pieces: Find a quick algorithm whose inefficiency in terms of the number of pieces is bounded with respect to the optimum. Second, compromise on the time complexity: Find an algorithm that produces an optimal partition, as quickly as possible. Although we will only discuss the first approach in any detail, we will mention results on the second approach.

Two types of partition of a polygon P may be distinguished: a partition by diagonals or a partition by segments. The distinction is that diagonal endpoints must be vertices, whereas segment endpoints need only lie on ∂P. Partitions by segments are in general

[14] The probability that the algorithm takes many steps can be made arbitrarily small by halting long runs and restarting with a new seed to the random number generator. See Alt, Guibas, Mehlhorn, Karp & Widgerson (1998).

[15] See Amenta (1997) for pointers to triangulation code.

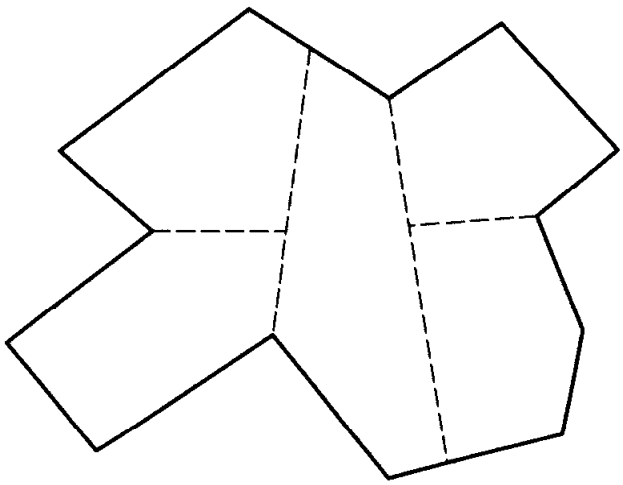


FIGURE 2.10 $r + 1$ convex pieces: $r = 4$; 5 pieces.

more complicated in that their endpoints must be computed somehow, but the freedom to look beyond the set of vertices often results in more efficient partitions.

2.5.1. Optimum Partition

To evaluate the efficiency of partitions, it is useful to have bounds on the best possible partition.

Theorem 2.5.1 (Chazelle). *Let Φ be the fewest number of convex pieces into which a polygon may be partitioned. For a polygon of r reflex vertices, $\lceil r/2 \rceil + 1 \leq \Phi \leq r + 1$.*

Proof. Drawing a segment that bisects each reflex angle removes all reflex angles and therefore results in a convex partition. The number of pieces is $r + 1$. See Figure 2.10.

All reflex angles must be resolved to produce a convex partition. At most two reflex angles can be resolved by a single partition segment. This results in $\lceil r/2 \rceil + 1$ convex pieces. See Figure 2.11. □

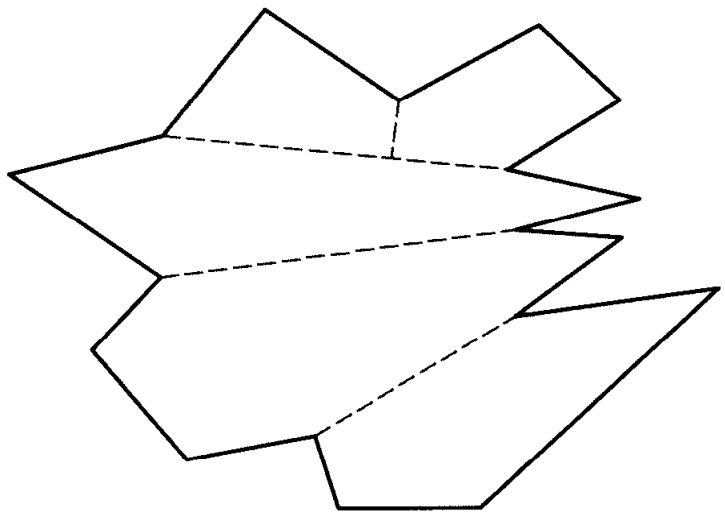


FIGURE 2.11 $\lceil r/2 \rceil + 1$ convex pieces: $r = 7$; 5 pieces.

2.5.2. Hertel and Mehlhorn Algorithm

Hertel & Mehlhorn (1983) found a very clean algorithm that partitions with diagonals quickly and has bounded "badness" in terms of the number of convex pieces.

In some convex partition of a polygon by diagonals, call a diagonal d *essential for vertex* v if removal of d creates a piece that is nonconvex at v. Clearly if d is essential it must be incident to v, and v must be reflex. A diagonal that is not essential for either of its endpoints is called *inessential.*

Hertel and Mehlhorn's algorithm is simply this: Start with a triangulation of P; remove an inessential diagonal; repeat. Clearly this algorithm results in a partition of P by diagonals into convex pieces. It can be accomplished in linear time with the use of appropriate data structures (Exercise 2.5.4[4]). So the only issue is how far from the optimum might it be.

Lemma 2.5.2. *There can be at most two diagonals essential for any reflex vertex.*

Proof. Let v be a reflex vertex and v_+ and v_- its adjacent vertices. There can be at most one essential diagonal in the halfplane H_+ to the left of vv_+; for if there were two, the one closest to vv_+ could be removed without creating a nonconvexity at v. See Figure 2.12. Similarly, there can be at most one essential diagonal in the halfplane H_- to the left of v_-v. Together these halfplanes cover the interior angle at v, and so there are at most two diagonals essential for v. □

Theorem 2.5.3. *The Hertel–Mehlhorn algorithm is never worse than four-times optimal in the number of convex pieces.*

Proof. When the algorithm stops, every diagonal is essential for some (reflex) vertex. By Lemma 2.5.2, each reflex vertex can be "responsible for" at most two essential diagonals. Thus the number of essential diagonals can be no more than $2r$, where r is the number of reflex vertices (and it can be less if some diagonals are essential for the vertices at both of its endpoints). Thus the number of convex pieces M produced by the algorithm satisfies $2r + 1 \geq M$. Since $\Phi \geq \lceil r/2 \rceil + 1$ by Lemma 2.5.1, $4\Phi \geq 2r + 4 > 2r + 1 \geq M$. □

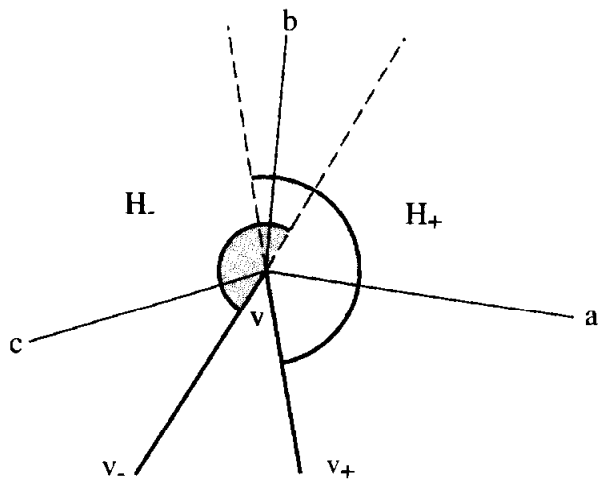


FIGURE 2.12 Essential diagonals. Diagonal a is not essential because b is also in H_+. Similarly c is not essential.

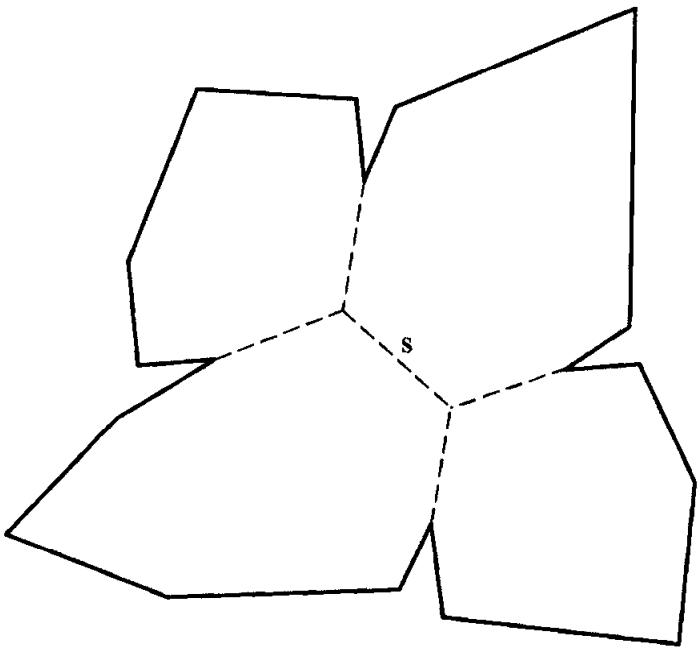


FIGURE 2.13 An optimal convex partition. Segment s does not touch ∂P.

2.5.3. Optimal Convex Partitions

As might be expected, finding a convex partition optimal in the number of pieces is much more time consuming than finding a suboptimal one. The first algorithm for finding an optimal convex partition of a polygon with diagonals was due to Greene (1983): His algorithm runs in $O(r^2n^2) = O(n^4)$ time. This was subsequently improved by Keil (1985) to $O(r^2n \log n) = O(n^3 \log n)$ time. Both employ dynamic programming, a particular algorithm technique.

If the partition may be formed with arbitrary segments, then the problem is even more difficult, as it might be necessary to employ partition segments that do not touch the polygon boundary, as shown in Figure 2.13. Nevertheless Chazelle (1980) solved this problem in his thesis with an intricate $O(n + r^3) = O(n^3)$ algorithm (see also Chazelle & Dobkin (1985)).

2.5.4. Exercises

1. *Worst case number of pieces.* Find a generic polygon that can lead to the worst case of the Hertel–Mehlhorn (H–M) algorithm: There is a triangulation and an order of inessential diagonal removal that leads to $2r$ convex pieces.
2. *Worst case with respect to optimum.* Find a generic polygon that can lead to the worst-case behavior in the H–M algorithm with respect to the optimum: H–M produces $2r$ pieces, but $\lceil r/2 \rceil + 1$ pieces are possible.
3. *Better optimality constant?* Is there any hope of improving the optimality constant of H–M below 4? Suppose the choice of diagonals was made more intelligently. Is a constant of, say, 3 possible?
4. *Implementing the Hertel–Mehlhorn algorithm* [programming]. Design a data structure that stores a subset of triangulation diagonals in a way that permits the "next" inessential diagonal to be found in constant time. Implement the H–M algorithm by altering and augmenting `Triangulate` (Code 1.14).

5. *Better approximate algorithm (diagonals)* [open]. Find a "fast" algorithm that achieves an optimality constant less than 4. By fast I mean $O(n\text{polylog}n)$, where polylog$n$ is some polynomial in $\log n$, such as $\log^3 n$.
6. *Better approximate algorithm (segments)* [open]. Find a fast approximation algorithm using segments rather than diagonals.
7. *Partition into rectangles.* Design an algorithm to partition an orthogonal polygon (Exercise 2.3.4[7]) into rectangles. Use only horizontal and vertical partition segments that are collinear with some polygon edge. Try to achieve as few pieces as possible, as quickly as possible.
8. *Cover with rectangles.* Design an algorithm to *cover* an orthogonal polygon P with rectangles whose sides are horizontal and vertical. Each rectangle should fall inside P, and their union should be exactly P. In a partition the rectangle interiors are pairwise disjoint, but in a cover they may overlap. Try to achieve as few pieces as possible, as quickly as possible.

3

Convex Hulls in Two Dimensions

The most ubiquitous structure in computational geometry is the convex hull (sometimes shortened to just "the hull"). It is useful in its own right and useful as a tool for constructing other structures in a wide variety of circumstances. Finally, it is an austerely beautiful object playing a central role in pure mathematics.

It also represents something of a success story in computational geometry. One of the first papers identifiably in the area of computational geometry concerned the computation of the convex hull, as will be discussed in Section 3.5. Since then there has been an amazing variety of research on hulls, ultimately leading to optimal algorithms for most natural problems. We will necessarily select a small thread through this work for this chapter, partially compensating with a series of exercises on related topics (Section 3.9).

Before plunging into the geometry, we briefly mention a few applications.

1. *Collision avoidance.* If the convex hull of a robot avoids collision with obstacles, then so does the robot. Since the computation of paths that avoid collision is much easier with a convex robot than with a nonconvex one, this is often used to plan paths. This will be discussed in Chapter 8 (Section 8.4).
2. *Fitting ranges with a line.* Finding a straight line that fits between a collection of data ranges maps[1] to finding the convex region common to a collection of half-planes (O'Rourke 1981).
3. *Smallest box.* The smallest area rectangle that encloses a polygon has at least one side flush with the convex hull of the polygon, and so the hull is computed at the first step of minimum rectangle algorithms (Toussaint 1983*b*). Similarly, finding the smallest three-dimensional box surrounding an object in space depends crucially on the convex hull of the object (O'Rourke 1985*a*).
4. *Shape analysis.* Shapes may be classified for the purposes of matching by their "convex deficiency trees," structures that depend for their computation on convex hulls. This will be explored in Exercise 3.2.3[2].

The importance of the topic demands not only formal definition of a convex hull, but a thorough intuitive appreciation. The convex hull of a set of points in the plane is the shape taken by a rubber band stretched around nails pounded into the plane at each point. The boundary of the convex hull of points in three dimensions is the shape taken by plastic wrap stretched tightly around the points.

We now examine a series of more formal definitions and approaches to convexity concepts. The remainder of the chapter is devoted to algorithms for constructing the hull.

[1] Maps via a duality relation to be studied in Chapter 6 (Section 6.5).

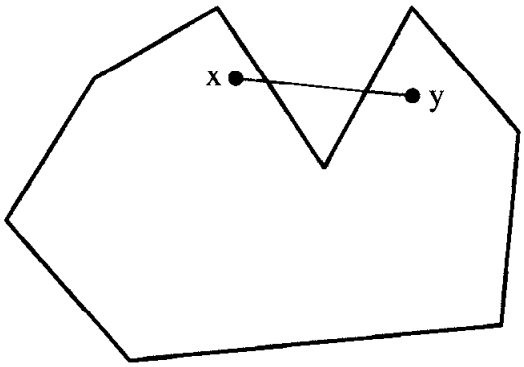


FIGURE 3.1 Any dent implies nonconvexity.

3.1. DEFINITIONS OF CONVEXITY AND CONVEX HULLS

1. A set S is *convex* if $x \in S$ and $y \in S$ implies that the segment $xy \subseteq S$. This can be taken as the primary definition of convexity. Note that this definition does not specify any particular dimension for the points, whether S is connected, bounded or unbounded, closed or open. It should be clear from Figure 3.1 that any region with a "dent" is not convex, since two points straddling the dent can be found such that the segment they determine contains points exterior to the region. Therefore in particular any polygon with a reflex vertex is not convex.
2. The *segment* xy is the set of all points of the form $\alpha x + \beta y$ with $\alpha \geq 0$, $\beta \geq 0$, and $\alpha + \beta = 1$.[2] For example the midpoint $\frac{1}{2}(x + y)$ is realized with equal weights: $\alpha = \frac{1}{2}$ and $\beta = \frac{1}{2}$; the endpoints are achieved with one weight zero. This algebraic view of a segment is quite useful both in mathematics and for computation. As an example of the latter, we will use it as the basis for finding the intersection point between two segments (Section 7.2).
3. A *convex combination* of points $x_1, \ldots, x_k$ is a sum of the form $\alpha_1 x_1 + \cdots + \alpha_k x_k$, with $\alpha_i \geq 0$ for all i and $\alpha_1 + \cdots + \alpha_k = 1$. Thus a line segment consists of all convex combinations of its endpoints, and a triangle consists of all convex combinations of its three corners. In three dimensions, a tetrahedron is the convex combinations of its four corners. Convex combinations lead to the concept of "barycentric coordinates," which we will use in Chapter 7 (Section 7.3.1).
4. The *convex hull* of a set of points S is the set of all convex combinations of points of S. In the mathematics literature, the convex hull of S is denoted by conv S. We will sometimes also use the notation $\mathcal{H}(S)$.

   Although it should be intuitively clear that the hull defined this way cannot have a dent, a proof is not immediate (Exercise 3.2.3[1]).
5. The convex hull of a set of points S in d dimensions is the set of all convex combinations of $d + 1$ (or fewer points) of S. The distinction between this and the previous definition is that here only $d + 1$ points need be used. Thus the hull of a

[2] In the expression $\alpha x + \beta y$, α and β are real numbers, while x and y are points or (equivalently) vectors.

two-dimensional set is the convex combinations of its subsets of three points, each of which, as we saw in (3) above, determine a triangle. That the $(d+1)$-points definition is equivalent to the all-points definition (4) is known as Caratheodory's Theorem (Lay 1982, p. 17).

6. The convex hull of a set of points S is the intersection of all convex sets that contain S. This definition is perhaps clearer than the previous two because it does not depend on the notion of convex combination. However, the notion of "all convex sets" is not easily grasped.
7. The convex hull of a set of points S is the intersection of all halfspaces that contain S. This is perhaps the clearest definition, equivalent (though not trivially) to all the others. A halfspace in two dimensions is a *halfplane*: It is the set of points on or to one side of a line. This notion generalizes to higher dimensions: A *halfspace* is the set of points on or to one side of a plane, and so on.

   Note that the convex hull of a set is a closed "solid" region, including all the points inside. Often the term is used more loosely in computational geometry to mean the boundary of this region, since it is the boundary we compute, and that implies the region. We will use the phrase "on the hull" to mean "on the boundary of the convex hull."
8. The convex hull of a finite set of points S in the plane is the smallest convex polygon P that encloses S, smallest in the sense that there is no other polygon P' such that $P \supset P' \supseteq S$.
9. The convex hull of a finite set of points S in the plane is the enclosing convex polygon P with smallest area.
10. The convex hull of a finite set of points S in the plane is the enclosing convex polygon P with smallest perimeter.

    The equivalence of these last two definitions (9 and 10), with smallest in terms of subset nesting (8), is intuitively but not mathematically obvious (Exercise 3.2.3[6]). But none of these three definitions of the boundary suggest an easy algorithm.
11. The convex hull of a set of points S in the plane is the union of all the triangles determined by points in S. This is just a restatement of (5) above, but in a form that hints at a method of computation.

The remainder of this chapter will concentrate on algorithms for constructing the boundary of the convex hull of a finite set of points in two dimensions. We will start with rather inefficient algorithms (Sections 3.2, 3.3, and 3.4), gradually working toward an optimal algorithm (Section 3.5), and finally examining algorithms that extend to three (and higher) dimensions (Sections 3.7 and 3.8). The only algorithm we exhibit in full detail, and for which code is provided, is Graham's (Section 3.5).

3.1.1. Extreme points

Before studying algorithms, we must first address the question of what output we desire from the algorithms, in particular, what constitutes constructing the boundary. Let us keep attention fixed to two dimensions until Chapter 4, with S a finite set of n points. Four outputs can be distinguished:

1. all the points on the hull, in arbitrary order;
2. the extreme points, in arbitrary order;
3. all the points on the hull, in boundary traversal order; and
4. the extreme points, in boundary traversal order.

The *extreme points* of a set S of points in the plane are the vertices of the convex hull at which the interior angle is strictly convex, less than π. Thus we only want to count "real" vertices as extreme: Points in the interior of a segment of the hull are not considered extreme.[3] Not only might different applications require different of the above outputs, but it is conceivable that, for example, it is easier to output hull points unordered ((1) and (2)) than to order them. We will see in Section 3.6 that in fact it is no easier (under the big-O measure).

Let us first concentrate on output (2): identifying the extreme points. First, note that the highest point of S, the one with the largest y coordinate, is extreme if it is unique, or even if there are exactly two equally highest vertices (both are then extreme). The same is of course true of the lowest points, the rightmost points, and the leftmost points. It should be clear that a point is extreme iff there exists a line through that point that otherwise does not touch the convex hull. Such "there exists" formulations, however, do not immediately suggest a method of computation.

Let us therefore look at the other side of the coin, the nonextreme points.

3.2. NAIVE ALGORITHMS FOR EXTREME POINTS

This section will be a bit of a digression, in that it will lead only to rather slow algorithms, but they will serve as useful foils for the faster algorithms to follow.

3.2.1. Nonextreme Points

Clearly, identifying the nonextreme points is enough to identify the extreme points.

Lemma 3.2.1. *A point is nonextreme iff it is inside some (closed) triangle whose vertices are points of the set and is not itself a corner of that triangle.*

Proof. The basis of this lemma is the final characterization of the hull, (11) in the list in Section 3.1. Assuming that, it is clear that if a point is interior to a triangle, it is nonextreme, and it is also evident that corners of a triangle might be extreme. A point that lies on the boundary of a triangle but not at a corner is not extreme. This accounts for all possibilities. □

Let $S = \{p_0, p_1, \ldots, p_{n-1}\}$, with all points distinct. Based on this lemma, Algorithm 3.1 is immediate. The in-triangle test can be implemented with three `LeftOns`.

[3] A more mathematical definition is that "a point x in S is extreme if there is no nondegenerate line segment in S that contains x in its relative interior" (Lay 1982, p. 42).

Algorithm: INTERIOR POINTS
for each i do
for each $j \neq i$ do
for each $k \neq i \neq j$ do
for each $l \neq i \neq j \neq k$ do
if $p_l \in \triangle(p_i, p_j, p_k)$
then p_l is nonextreme

Algorithm 3.1 Interior points.

Note that it is unnecessary to check the second clause of the lemma, that p_l not be a corner of the triangle: By our assumption that the points of S are distinct, and our exclusion of i, j, and k as indices in the l loop, this condition is guaranteed.

This algorithm clearly runs in $O(n^4)$ time because there are four nested loops, each $O(n)$: For each of the n^3 triangles, the test for extremeness costs n. It would be a challenge to find a slower algorithm!

3.2.2. Extreme Edges

It is somewhat easier to identify extreme edges, edges of the convex hull. An edge is extreme if every point of S is on or to one side of the line determined by the edge. It seems easiest to detect this by treating the edge as directed, and specifying one of the two possible directions as determining the "side." Let the left side of a directed edge be the inside. Phrased negatively, a directed edge is not extreme if there is some point that is not left of it or on it. This is the formulation we use in the pseudocode below.

Unfortunately this algorithm computes output (1) above rather than (2). For suppose xy is an extreme edge, and z lies on the interior of the segment xy. Then xz and zy will both have the property that there is no point strictly to their rights – no point that is not left of or on. But it makes sense to say that neither of these counts as an extreme edge and to demand that both endpoints of an *extreme edge*, be extreme vertices.

We opt not to check this precise condition below (since we are only sketching this algorithm in order to improve upon it); therefore Algorithm 3.2 only produces output (2) for point sets "in general position," with no three points collinear.

Algorithm: EXTREME EDGES
for each i do
for each $j \neq i$ do
for each $k \neq i \neq j$ do
if p_k is *not* left or on (p_i, p_j)
then (p_i, p_j) is not extreme

Algorithm 3.2 Extreme edges.

This algorithm clearly runs in $O(n^3)$ time because there are three nested loops, each $O(n)$: For each of the n^2 pairs of points, the test for extremeness costs n. Which vertices are extreme can be found easily now (under the general position assumption), since an extreme point is an endpoint of two extreme edges.

3.2.3. Exercises

1. *Convexity of the convex hull.* Starting from the definition of the convex hull of S as the set of all convex combinations of points from $S(4)$, prove that conv S is in fact convex, in that the segment connecting any two points is in conv $S(1)$.
2. *Extreme point implementation* [programming]. Write code to take a list of points as input and to print the extreme points in arbitrary order. Try to write the shortest, simplest code you can think of, without regard to running time. Make use of the functions in the triangulation code from Chapter 1: `ReadPoints` to read in the points (that they do not necessarily form a polygon is irrelevant), `Left` and `LeftOn` (Code 1.6), and so on.
3. *Min supporting line* (Modayur 1991). Design an algorithm to find a line L that
   a. has all the points of a given set to one side,
   b. minimizes the sum of the perpendicular distances of the points to L.
   Assume a hull algorithm is available.
4. *Affine hulls.* An *affine combination* of points $x_1, \ldots, x_k$ is a sum of the form $\alpha_1 x_1 + \cdots + \alpha_k x_k$, with $\alpha_1 + \cdots + \alpha_k = 1$. Note that this differs from the definition of a convex combination (3) in that the condition $\alpha_i \geq 0$ is dropped. In two dimensions, what is the affine hull of two points? Three points? $n > 3$ points? In three dimensions, what is the affine hull of two points? Three points? Four points? $n > 4$ points?
5. *Extreme edges.* Modify Algorithm 3.2 so that it works correctly without the general position assumption.
6. *Minimum area, convex.* Prove characterization (9) of Section 3.1: The minimum area convex polygon enclosing a set of points is the convex hull of the points.
7. *Minimum area, nonconvex* [easy]. Show by explicit example that the minimum area polygon (perhaps nonconvex) enclosing a set of points might not be the convex hull of the points.
8. *Shortest path below.* Let a set of points S and two additional points a and b be given, with a left of S and b right of S. Develop an algorithm to find the shortest path from a to b that avoids the interior of S. Assume a convex hull algorithm is available.

3.3. GIFT WRAPPING

We now move to more realistic hull algorithms. A minor variation on the Extreme Edge algorithm (Algorithm 3.2) will both accelerate it by a factor of n and at the same time output the points in the order in which they occur around the hull boundary. The idea is to use one extreme edge as an anchor for finding the next. This works because we know that the extreme edges are linked into a convex polygon. Since the most vertices this polygon can have is n, the number of extreme edges is $O(n)$. The anchored search will only explore $O(n)$ candidates, rather than the $O(n^2)$ candidates in Algorithm 3.2. This saves a factor of n and reduces the complexity to $O(n^2)$.

Now let's examine how this anchored search can be accomplished. Assume general position of the points for clarity: no three points in S are collinear, so that outputs

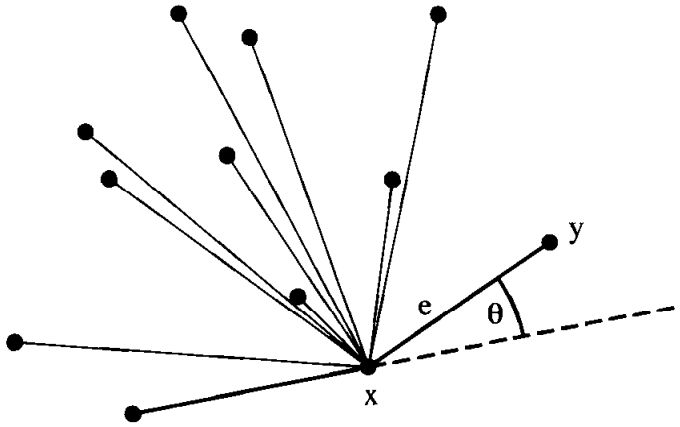


FIGURE 3.2 The next edge e makes the smallest angle θ with respect to the previous edge.

(3) and (4) are the same. Suppose the algorithm last found an extreme edge whose unlinked endpoint is x; see Figure 3.2. We know there must be another extreme edge e sharing endpoint x. Draw a directed line L from x to another point y of the set. L includes e only if all other points are to the left, or alternatively, only if there are no points to the right. But note that, if we check for each y whether all other points are to the left, we will be back to an n^3 calculation: For each x, for each y, check all other points.

The key observation to reducing the complexity is that, as can be seen from Figure 3.2, the line L that includes e also has the property of making the smallest counterclockwise angle with respect to the previous hull edge. This implies that it is not necessary to check whether all points are to the left: This can be *inferred* from the angle. So for each point y, compute that angle; call it θ. The point that yields the smallest θ must determine an extreme edge (under the general position assumption).

The reason this algorithm is called the "gift wrapping" algorithm should now be clear: One can view it as wrapping the point set with a string that bends the minimal angle from the previous hull edge until the set is hit. This algorithm was first suggested by Chand & Kapur (1970) as a method for finding hulls in arbitrary dimensions. We will see that it can be surpassed in two dimensions, but for many years it was the primary algorithm for higher dimensions. One nice feature is that it is "output-size sensitive," in that it runs faster when the hull is small: Its complexity is $O(nh)$ if the hull has h edges (Exercise 3.4.1[1]).

Again we would need to modify the algorithm to remove the general position assumption, and again we will not bother. There remains one minor detail: how to start the algorithm. We can use the lowest point of the set as the first anchor, treating the "previous" hull edge as horizontal. Pseudocode is shown in Algorithm 3.3. This algorithm runs in $O(n^2)$ time: $O(n)$ work for each hull edge.

3.4. QUICKHULL

We continue our catalog of hull algorithms with one that was suggested independently by several researchers in the late 1970s. It was dubbed the "QuickHull" algorithm by Preparata & Shamos (1985) because of its similarity to the QuickSort

Algorithm: GIFT WRAPPING
Find the lowest point (smallest y coordinate).
Let i_0 be its index, and set $i \leftarrow i_0$.
repeat
    for each $j \neq i$ do
        Compute counterclockwise angle θ from previous hull edge.
    Let k be the index of the point with the smallest θ.
    Output (p_i, p_k) as a hull edge.
    $i \leftarrow k$
until $i = i_0$

Algorithm 3.3 Gift wrapping.

algorithm (Knuth 1973).[4] The basic intuition is as simple as it is sound: For "most" sets of points, it is easy to discard many points as definitely interior to the hull, and then concentrate on those closer to the hull boundary. The first step of the QuickHull algorithm is to find two distinct extreme points; we will use the rightmost lowest and leftmost highest points x and y, which are guaranteed extreme and distinct (cf. Lemma 1.2.1 and Figure 1.11); see Figure 3.3. The full hull is composed of an "upper hull" above xy and a "lower hull" below xy. QuickHull finds these through a procedure that starts with extreme points (a, b), finds a third extreme point c strictly right of ab, discards all points inside $\triangle abc$, and operates recursively on (a, c) and (c, b).

Let S be the set of points strictly right of ab (S may be empty). The key idea is that a point $c \in S$ that is furthest away from ab must be on the hull: It is extreme in the direction orthogonal to ab. Therefore we can discard all points on or in $\triangle abc$ (except for a, b, and c themselves) and repeat the same procedure on the points A right of ac and the points B right of cb; again see Figure 3.3.

The pseudocode shown in Algorithm 3.4 assumes the procedure returns a list of points and uses '+' to represent list concatenation. The final hull is (x) + *QuickHull* (x, y, S_1)+ (y) + *QuickHull* (y, x, S_2), where S_1 and S_2 are the points strictly above and below xy respectively. The successive triangles $\triangle abc$ generated by the recursive calls are shown in Figure 3.3. We leave further details to Exercise 3.4.1[3].

Algorithm: QUICKHULL
function *QuickHull*(a, b, S)
    if $S = \emptyset$ then return ()
    else
        $c \leftarrow$ index of point with max distance from ab.
        $A \leftarrow$ points strictly right of (a, c).
        $B \leftarrow$ points strictly right of (c, b).
        return *QuickHull*$(a, c, A) + (c) +$ *QuickHull*(c, b, B)

Algorithm 3.4 QuickHull.

[4]The presentation here is based upon that in Preparata & Shamos (1985, pp. 112–14).

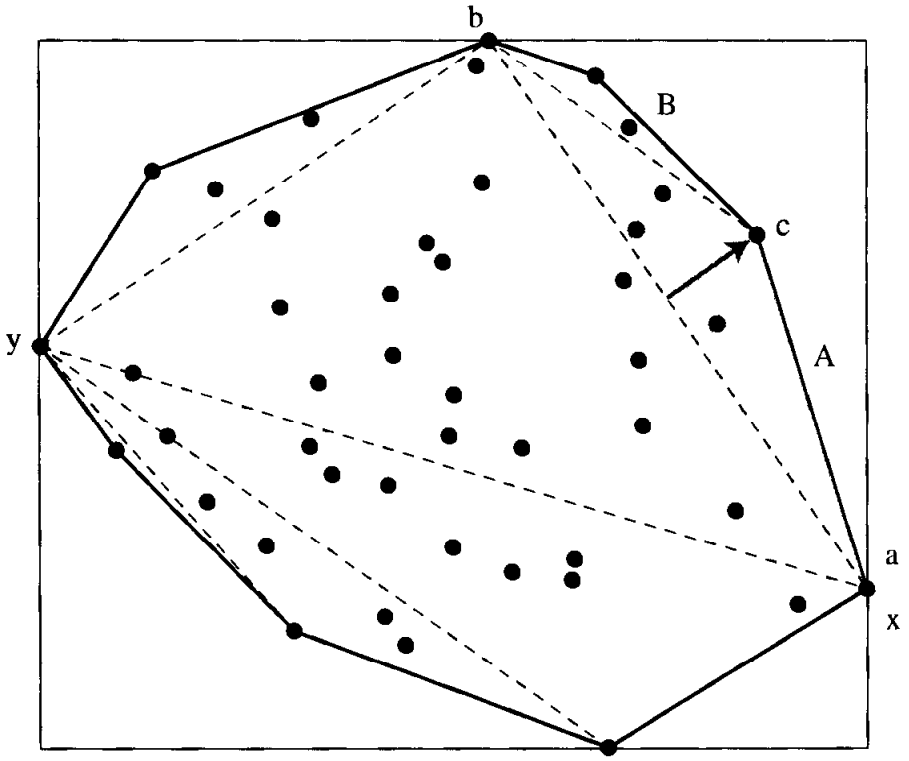


FIGURE 3.3 QuickHull discards the points in $\triangle abc$ (shaded) and recurses on A and B. Here $A = \emptyset$ and $|B| = 2$.

We turn now to an analysis of the time complexity of QuickHull. Finding the initial extremes x and y, and partitioning S into S_1 and S_2, can be accomplished in $O(n)$ time. For the recursive function, suppose $|S| = n$. Then it takes n steps to determine the extreme point c, but the cost of the recursive calls depends on the sizes of A and B. Let $|A| = \alpha$ and $|B| = \beta$ with $\alpha + \beta \leq n - 1 = O(n)$. (The sum is at most $n - 1$ because c is not included in either A or B.) If the time complexity for calling QuickHull with $|S| = n$ is $T(n)$, we can express T recursively in terms of itself: $T(n) = O(n) + T(\alpha) + T(\beta)$. It is not possible to solve this equation without expressing α and β in terms of n.

Consider the best possible case, when each division is as balanced as possible: $\alpha = \beta = n/2$ (it is safe to ignore the minor discrepancy that $\alpha + \beta$ should sum to $n - 1$ in this rough analysis). Then we have $T(n) = 2T(n/2) + O(n)$. This is a familiar recurrence relation, whose solution is $T(n) = O(n \log n)$. Therefore $T(n) = O(n \log n)$ in the "best" case, which would occur with randomly distributed point sets.

The worst case occurs when each division is as skewed as possible: $\alpha = 0$ and $\beta = n - 1$. Then we arrive at the recurrence relation $T(n) = T(n - 1) + O(n) = T(n - 1) + cn$. Repeated expansion shows this to be $O(n^2)$. Thus although QuickHull is indeed generally quick (Exercise 3.4.1[7]), it is still quadratic in the worst case.

In the next section we culminate our progression of ever-faster algorithms with a worst-case optimal $O(n \log n)$ algorithm.

3.4.1. Exercises

1. *Best case?* Find the best case for the gift wrapping algorithm (Algorithm 3.3): sets of n points such that the algorithm's asymptotic time complexity is as small as possible as a function of n. What is this time complexity?

2. *Improving gift wrapping.* During the course of gift wrapping (Algorithm 3.3), it is sometimes possible to identify points that cannot be on the convex hull and to eliminate them from the set "on the fly." Work out rules to accomplish this. What is a worst-case set of points for your improved algorithm?
3. *QuickHull details.* Provide more details for the QuickHull algorithm. In particular, specify how the point c with maximum distance from ab can be found. Also detail what strategy should be pursued in case c is not unique, in order to achieve output (4): only the extreme points. Finally, check that the algorithm works for an input consisting of n collinear points.
4. *QuickHull worst case.* Construct a generic point set that forces QuickHull to its worst-case quadratic behavior. By "generic" is meant a construction that works for arbitrarily large values of n (i.e., "general" n).
5. *Analysis of worst case.* Argue that QuickHull, like gift wrapping, has output-size sensitive complexity $O(nh)$ for a set of n points, h of which are on the hull.
6. *Implementation of QuickHull* [programming]. Implement QuickHull, and measure its performance on points uniformly distributed within a square.
7. *Average time complexity of QuickHull* (Scot Drysdale). Argue that QuickHull's average time complexity on points uniformly distributed within a square is $O(n)$. *Hint:* The area of a triangle is half the area of a surrounding parallelogram with the same base.

3.5. GRAHAM'S ALGORITHM

Perhaps the honor of the first paper published in the field of computational geometry should be accorded to Graham's algorithm for finding the hull of points in two dimensions in $O(n \log n)$ time (Graham 1972).[5] In the late 1960s an application at Bell Laboratories required the hull of $n \approx 10,000$ points, and they found the $O(n^2)$ algorithm in use too slow. Graham developed his simple algorithm in response to this need (personal comm., 1992).

3.5.1. Top Level Description

The basic idea of Graham's algorithm is simple. We will first explain it with an example, making several assumptions that will be removed later. Assume we are given a point x interior to the hull, and further assume that no three points of the given set (including x) are collinear. Now sort the points by angle, counterclockwise about x. For the example shown in Figure 3.4, the sorted points are labeled $a, b, \ldots, j$. The points are now processed in their sorted order, and the hull grown incrementally around the set. At any step, the hull will be correct for the points examined so far, but of course points encountered later will cause earlier decisions to be reevaluated.

The hull-so-far is maintained in a stack S of points. Initially the stack contains the first two points, $S = (b, a)$ in our example, with b on top. We will use the convention of listing the stack top to bottom, left to right, anticipating its implementation in Section 3.5.5. Point c is added because (a, b, c) forms a left turn at b, the previous stack top. Note that

[5] I say "perhaps" because Toussaint (1985*a*) found an earlier paper that contained many ideas that appeared in later hull algorithms. See Bass & Schubert (1967).

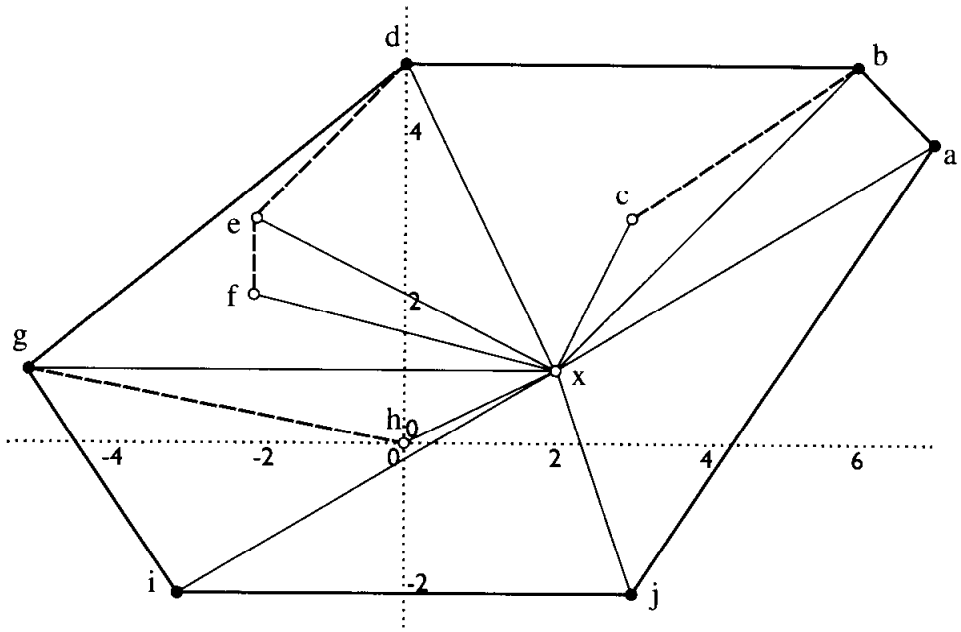


FIGURE 3.4 Example for Graham's algorithm: $x = (2, 1)$; $S = \{(7, 4), (6, 5), (3, 3), (0, 5), (-2, 3), (-2, 2), (-5, 1), (0, 0), (-3, -2), (3, -2)\}$.

$S = (c, b, a)$ is a convex chain, a condition that will be maintained throughout. Next point d is considered, but since (b, c, d) forms a right turn at the stack top c, the chain is not extended, but rather the last decision, to add c, is revoked by poping c from the stack, which then becomes $S = (b, a)$ again. Now d is added, because (a, b, d) forms a left turn at b.

Continuing in this manner, e and f are added, after which the stack is $S = (f, e, d, b, a)$. Point g causes f and then e to be deleted, since both (e, f, g) and (d, e, g) are right turns. Then g can be added, and the stack is $S = (g, d, b, a)$. And so on.

If we are as fortunate as in the considered example and our first point a is on the hull, the convex chain will close naturally, resulting in the final hull $S = (j, i, g, d, b, a)$. Note that from stack top to bottom represents a clockwise traversal, as we built it up via counterclockwise scan. If a were not on the hull, the head of the chain would start to consume the tail (so to speak), and the algorithm analysis would be more difficult. We will see that this can be avoided.

3.5.2. Pseudocode, Version A

Before proceeding to a more careful presentation, we summarize the rough algorithm in pseudocode in Algorithm 3.5. We assume stack primitives Push(p, S) and Pop(S), which push p onto the top of the stack S, and pop the top off, respectively. We use t to index the stack top and i for the angularly sorted points. Many issues remain to be examined (start and termination in particular), but at this coarse level, it should be apparent that the while loop iterates $O(n)$ times: Each stack pop permanently removes one point, so the number of backups cannot exceed n. Together with n forward steps, the loop iterates at most $2n$ times. So the algorithm runs in linear time after the sorting step, which takes $O(n \log n)$ time. We will see in Section 3.6 that this is the best that can be hoped for: Its time complexity is "worst-case optimal."

Algorithm: GRAHAM SCAN, VERSION A
Find interior point x; label it p_0.
Sort all other points angularly about x; label $p_1, \ldots, p_{n-1}$.
Stack $S = (p_2, p_1) = (p_t, p_{t-1})$; t indexes top.
$i \leftarrow 3$
while $i < n$ do
    if p_i is left of (p_{t-1}, p_t)
        then Push (p_i, S) and set $i \leftarrow i + 1$.
        else Pop(S).

Algorithm 3.5 Graham Scan, Version A.

3.5.3. Details: Boundary Conditions

A number of details have been ignored in our presentation so far. We will rectify this in two stages. First, various "boundary" conditions are examined in this section. Second, implementation issues are explored in the sections following.

Start and Stop of Loop

Even a simple loop can be difficult to start and stop properly: The algorithm so far presented might have trouble at either end.[6] We already mentioned the termination difficulties that would arise if a, the stack bottom, were not on the hull. Startup difficulties occur when b, the second point pushed on the stack, is not on the hull. For suppose that (a, b, c) is a right turn. Then b would be popped from the stack, and the stack reduced to $S = (a)$. But at least two points are needed to determine if a third forms a left turn with the stack top.

Clearly both startup and stopping problems are avoided if both a and b are on the hull. How this can be arranged will be shown in the next subsection.

Sorting Origin

We assumed that the point x, about which all others are sorted, is interior to the hull. Graham provided a careful linear algorithm for computing such an interior point.[7] However, not only is this calculation unnecessary, it may force the use of floating-point numbers even when the input coordinates are all integers. We would like to avoid all floating-point calculations to guarantee a correct answer on integer input.

A simplification is to sort with respect to a point of the set, and in particular, with respect to a point on the hull.[8] We will use the lowest point, which is clearly on the

[6] Several early published versions were in error over these difficulties. A short history is presented by Gries & Stojmenović (1987).

[7] His method may be of interest in its own right (Graham 1972): "... this can be done ... by testing 3 element subsets ... for collinearity, discarding middle points of collinear sets, and stopping when the first noncollinear set (if there is one), say x, y, and z, is found. [The point] can be chosen to be the centroid of the triangle formed by x, y, and z." It is notable that he did not assume that the given points are in general position.

[8] This useful idea occurred to several researchers independently, including Akl & Toussaint (1978) and Anderson (1978).

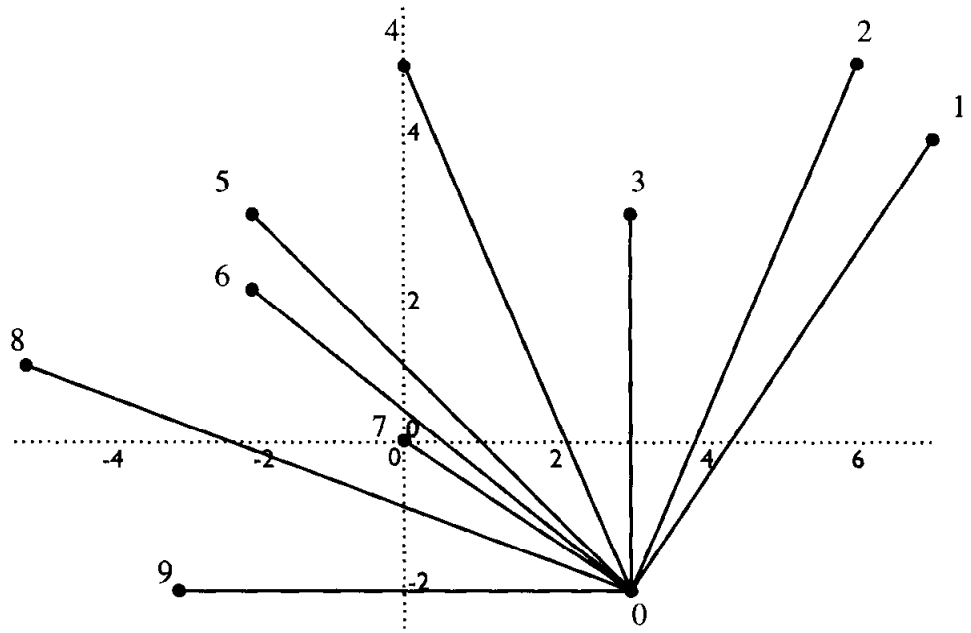


FIGURE 3.5 New sorting origin for the points in Figure 3.4.

hull. In case there are several with the same minimum y coordinate, we will use the rightmost of the lowest as the sorting origin. This is point j in Figure 3.4. Now the sorting appears as in Figure 3.5. Note all points in the figure have been relabeled with numbers; this is how they will be indexed in the implementation. We will call the points $p_0, p_1, \ldots, p_{n-1}$, with p_0 the sorting origin and p_{n-1} the most counterclockwise point.

Now we are prepared to solve the startup and termination problems discussed above. If we sort points with respect to their counterclockwise angle from the horizontal ray emanating from our sorting origin p_0, then p_1 *must* be on the hull, as it forms an extreme angle with p_0. However, it may not be an extreme point (in the sense defined in Section 3.1.9), an issue we will address below. If we initialize the stack to $S = (p_0, p_1)$, the stack will always contain at least two points, avoiding startup difficulties, and will never be consumed when the chain wraps around to p_0 again, avoiding termination difficulties.

Collinearities

The final "boundary condition" we consider is the possibility that three or more points are collinear, until now a situation conveniently assumed not to occur. This issue affects several aspects of the algorithm. First we focus on defining precisely what we seek as output.

Hull Collinearities. We insist here on the most useful output (4): the extreme vertices only, ordered around the hull. Thus if the input consists of the corners of a square, together with points sprinkled around its boundary, the output should consist of just the four corners of the square. Avoiding nonextreme hull points is easily achieved by requiring a *strict* left turn (p_{t-1}, p_t, p_i) to push p_i onto the stack, where p_t and p_{t-1} are the top two points on the stack. Then if p_t is collinear with p_{t-1} and p_i, it will be deleted.

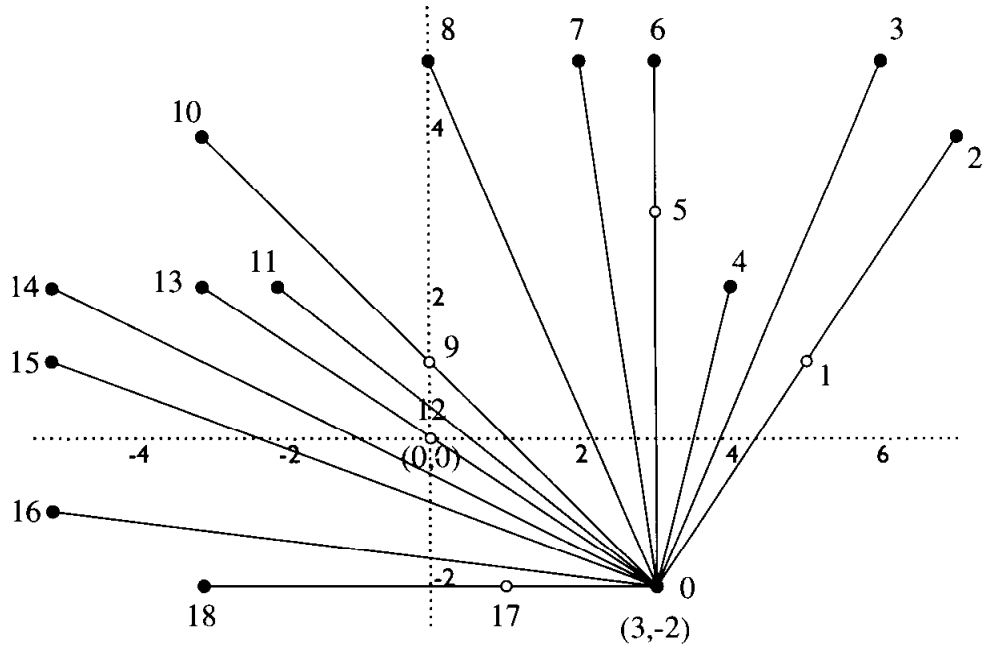


FIGURE 3.6 Sorting points with collinearities. Indices indicate sorting rank. Points to be deleted are shown as open circles.

Sorting Collinearities. Collinearities raise another issue: How should we break ties in the angular sorting if both points a and b form the same angle with p_0? One's first inclination is to assume (or hope) it does not matter, but alas the situation is more delicate. There are at least two options. First, use a consistent sorting rule, and then ensure start and stop and hull collinearities are managed appropriately. A reasonable rule is that if angle(a) = angle(b), then define $a < b$ if $|a - p_0| < |b - p_0|$: Closer points are treated as earlier in the sorting sequence. With this rule we obtain the sorting indicated by the indices in the example shown in Figure 3.6.

Note, however, that p_1 is not extreme in the figure, which makes starting with $S = (p_1, p_0)$ problematic. Although this can be circumvented by starting instead with $S = (p_{n-1}, p_0)$ (note that $p_{n-1} = p_{18}$ is extreme), we choose here a second option.[9] It is based on this simple observation: If angle(a) = angle(b) and $a < b$ according to the above sorting rule, then a is not an extreme point of the hull and may therefore be deleted. In Figure 3.6, points p_1, p_5, p_9, p_{12}, and p_{17} may be deleted for this reason.

Coincident Points. Often code that works on distinct points crashes for sets that may include multiple copies of the same point. We will see that we can treat this issue as a special case of a sorting collinearity, deleting all but one copy of each point.

3.5.4. Pseudocode, Version B

Before proceeding with implementation details, we summarize the preceding discussion with pseudocode in Algorithm 3.6 that incorporates the changes.

[9]See O'Rourke (1994, 87ff.) for the first option. I owe the idea for the second option to Chee K. Yap.

Algorithm: GRAHAM SCAN, VERSION B
Find rightmost lowest point; label it p_0.
Sort all other points angularly about p_0.
    In case of tie, delete the point closer to p_0
    (or all but one copy for multiple points).
Stack $S = (p_1, p_0) = (p_t, p_{t-1})$; t indexes top.
$i = 2$
while $i < n$ do
    if p_i is strictly left of $p_{t-1}p_t$
        then Push(p_i, S) and set $i \leftarrow i + 1$
        else Pop(S).

Algorithm 3.6 Graham Scan, Version B.

We have not discussed yet the details of loop termination. Is the condition $i < n$ correct, even when there are collinearities? Or should it be $i \leq n$, or $i < n - 1$? Note from the pseudocode that by the time the while loop is entered, the sorting collinearities have been removed. So once p_{n-1} is pushed on the stack, we are assured of being finished: p_{17} in Figure 3.6 is gone. Thus the loop stopping condition is indeed $i < n$.

3.5.5. Implementation of Graham's Algorithm

We now describe an implementation of Algorithm 3.6. We assume the input points are given with integer coordinates, and we insist upon avoiding all floating-point calculations so that a correct output can be guaranteed. We will see that this can only be guaranteed if the coordinates are not too large, but with that caveat aside, the implementation yields the correct hull.

We start with data structures, then tackle the sorting step, and finally present the code.

Data Representation

As usual, we have a choice between storing the points in an array or a list. We choose in this instance to use an array, anticipating using a sorting routine that expects its data in a contiguous section of memory. Each point will be represented by a structure paralleling that used for vertices in Chapter 1 (Code 1.2). The points are stored in a global array `P`,[10] with `P: P[0], . . . , P[n-1]` corresponding to $p_0, \ldots, p_{n-1}$. Each `P[i]` is a structure with fields for its coordinates, a unique identifying number, and a flag to mark deletion. See Code 3.1.

The stack is most naturally represented by a singly linked list of cells, each of which "contains" a point (i.e., contains a pointer to a point record). See Code 3.2. With these definitions, the stack top can be declared as `tStack top`, and the element under the top is `top->next`.

[10]The `static` declaration limits the scope to the use of functions in this file (e.g., `Compare`).

```
typedef struct tPointStructure tsPoint;
typedef tsPoint *tPoint;
struct tPointStructure {
  int     vnum;
  tPointi v;
  bool    delete;
};

#define PMAX    1000                    /* Max # of points */
typedef tsPoint tPointArray[PMAX];
static tPointArray P;
int n = 0;                              /* Actual # of points */
```

Code 3.1 Point(s) structure. (See Code 1.1 for `tPointi` and `bool` and other `defines`.

```
typedef struct tStackCell tsStack;  /* Used only in NEW() */
typedef tsStack *tStack;
struct tStackCell {
  tPoint   p;
  tStack   next;
};
```

Code 3.2 Stack structure.

We need three stack manipulation routines (Code 3.3): `Pop`, `Push`, and `PrintStack`. The stack top is always the head (leftmost) element of the list. `Pop` frees the top cell and returns a pointer to the next cell. `Push(p,t)` allocates new storage, fills it up with p, and makes it the new stack top. Note in `PrintStack` that `t->p->v` reaches the coordinates of the point: `t` is type `tStack`, `p` is type `tPoint`, `v` is type `tPointi`, the latter being the type used in Code 1.1 for the coordinates of a point.

Sorting

FindLowest. We first dispense with the easiest aspect of the sorting step: finding the rightmost lowest point in the set. The function `FindLowest` (Code 3.4) accomplishes this and swaps the point into `P[0]`. The straightforward `Swap` is not shown.

Avoiding Floats. The sorting step seems straightforward, but there are hidden pitfalls if we want to guarantee an accurate sort. First we introduce a bit of notation. Let $r_i = p_i - p_0$, the vector from p_0 to p_i. Our goal is to give a precise calculation to determine when $p_i < p_j$, where "$<$" represents the sorting relation.

```
tStack   Pop( tStack s )
{
  tStack top;

  top = s->next;
  FREE( s );
  return top;
}

tStack   Push( tPoint p, tStack top )
{
  tStack   s;

  /* Get new cell and fill it with point. */
  NEW( s, tsStack );
  s->p = p;
  s->next = top;
  return s;
}

void   PrintStack( tStack t )
{
  while (t) {
    printf("vnum=%d\tx=%d\ty=%d\n",
        t->p->vnum,t->p->v[X],t->p->v[Y]);
    t = t->next;
  }
}
```

Code 3.3 Stack routines. (See Code 1.4 for FREE and NEW.)

```
void   FindLowest( void )
{
  int i;
  int m = 0;    /* Index of lowest so far. */

  for ( i = 1; i < n; i++ )
    if ( (P[i].v[Y] < P[m].v[Y]) ||
      ((P[i].v[Y] == P[m].v[Y]) && (P[i].v[X] > P[m].v[X])) )
      m = i;
  Swap(0,m);   /* Swap P[0] and P[m] */
}
```

Code 3.4 FindLowest.

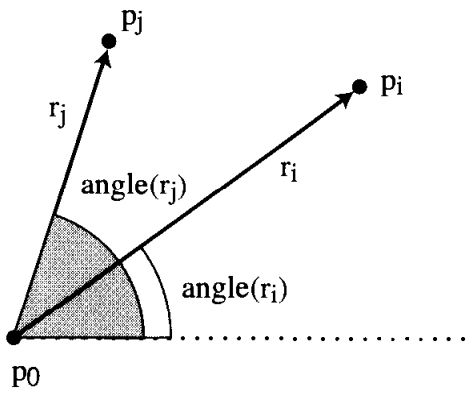


FIGURE 3.7 Notation for sorting angle.

Atan2. The obvious choice is to define $p_i < p_j$ if angle(r_i) < angle(r_j), where angle(r) is the counterclockwise angle of r from the positive x axis. See Figure 3.7. (We will discuss tie breaking later.) Since p_0 is lowest, all these angles are in the range $(0, \pi]$, which is convenient because sorting positive and negative angles can be tricky (as we will see in Section 8.4.4). C provides precisely the desired function: angle(r) = `atan2( r[Y], r[X] )`. There are at least two reasons not to use this:

- Although the conversion from `ints` to `doubles` (required by `atan2`) should be accurate, there is no guarantee that the arctangent computation is itself accurate.
- The arctangent is a complicated, expensive function – slopes are simpler and serve the same purpose.

Slopes. For r in the first quadrant (i.e., both coordinates positive), the slope `r[Y]/r[X]` can substitute for the arctangent, and in the second quadrant, we could use `(-r[X]/r[Y])`. Although invoking a simpler calculation than `atan2`, this approach suffers from several of the same weaknesses. An example should suffice to illustrate the point.

Clearly if $r_i = cr_j$, where c is some positive number, then we want to conclude that angle(r_i) = angle(r_j), and then fall into our tie-breaking code. If we are using slopes, this amounts to requiring that $a/b = (ca)/(cb)$, which is of course true mathematically. But it may come as a shock to realize that this equality is not guaranteed to hold for floating-point division in C. It depends on the machine. In particular, on a Cray Y-MP/4, the following evaluates to FALSE: `1.0/1.0 == 3.0/3.0`. The reason is that the Cray performs division by reciprocating and multiplying, and the two operations sometimes lead to small errors. Here `3.0/3.0 == 0.999999999999996`.[11]

The lesson is clear: Any floating-point calculation might be in error on some machine whose C compiler and libraries fully meet specifications. This is why we opt for integer computation whenever possible.

`Left`. The impatient reader will have realized long ago that the solution is already in hand: `Left`, the function to determine whether a point is left of a line determined by two other points, is precisely what we need to compare r_i and r_j. And we implemented this entirely with integer computations in Chapter 1. Recall that `Left` was itself a simple test on the value of `Area2`, which computes the signed area of the triangle determined

[11] I thank Dik T. Winter for this example (personal comm., 1991).

by three points. We will use this area function rather than `Left`, as it is then easier to distinguish ties.

As we mentioned before (Section 1.4.3), `Area2` itself is "flawed" in that standard C gives no error upon integer overflow, which could occur in the calculation if the input coordinates are large with respect to the machine word size. This point will be revisited in Chapter 4 (Code 4.23), but it is worth noting the dilemma faced by most existent C code[12] that performs geometric computations: If the code uses floating-point calculations, it may rely on machine-dependent features, whereas if the code uses integer calculations only, it is not guaranteed to work correctly unless the size of the integers is severely restricted. This situation is fortunately starting to change now with the availability of packages with robust computation options, such as LEDA (see Chapter 9).

We will first discuss the overall sorting method before showing how `Area2` is employed for comparisons.

`Qsort`. The standard C library includes a sorting routine `qsort` ("quicker sort") that is at least as good as most programmers could develop on their own, and it makes sense to use it. Because it is general, however, it takes a bit of effort to set it up properly. The routine requires information on the shape and location of the data and a comparison function to compare keys. It then uses the provided comparison function to sort the data in place, from smallest to largest. Its four arguments are:

1. A pointer to the base of the data, in our case `&P[1]`, the address of the first point. (Remember that `P[0]` is fixed as our lexicographic minimum.)
2. The number of elements to be sorted: `n-1`.
3. The width of an element in bytes: `sizeof( tsPoint )`.
4. The name of the two-argument comparison routine: `Compare`.

`Compare`. The routine `qsort` expects the comparison function to "return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second." Moreover, it will be called "with two arguments which are pointers to the elements being compared."[13] This latter requirement presents a slight technical problem, because we need three inputs: p_0, p_i, and p_j. One way around this difficulty is to compute a separate array filled with the r_i's. Here we choose another solution: Make `P` global, so that p_0 can be referenced inside the body of `Compare` without passing it as a parameter.

Finally we can specify the heart of `Compare` (Code 3.5): If p_j is left of the directed line through p_0p_i, then $p_i < p_j$. In other words, $p_i < p_j$ if `Area2` $(p_0, p_i, p_j) > 0$, in which case `qsort` expects a return of `-1` to indicate "less than." See Figure 3.7.

When the area is zero, we fall into a sorting collinearity, which requires action as discussed previously (Section 3.5.3). First we need to decide which point is closer to p_0. We can avoid computing the distance by noting that if p_i is closer, then $r_i = p_i - p_0$ projects to a shorter length than does $r_j = p_j - p_0$. The code computes these projections

[12]This is true of many other languages as well, including any language that uses a fixed word size to represent either integers or floating-point numbers.

[13]The quotes are from `man qsort`, the manual page on the routine.

x and y and bases further decisions on them. If p_i is closer, mark it for later deletion; if p_j is closer, mark it instead. If $x = y = 0$, then $p_i = p_j$ and we mark the one with lower index for later deletion. (Which one we choose to delete doesn't matter, but consistency is required by `qsort`.) In all cases, one member of $\{-1, 0, +1\}$ is returned according to the angular sorting rule detailed earlier.

The peculiar casting from `const void *tpi` to `pi = (tPoint)tpi` satisfies the demands of `qsort` but permits natural use in the body of `Compare`.

```
int   Compare( const void *tpi, const void *tpj )
{
  int a;              /* area */
  int x, y;           /* projs. of ri & rj in 1st quadrant */
  tPoint pi, pj;
  pi = (tPoint)tpi;
  pj = (tPoint)tpj;

  a = Area2( P[0].v, pi->v, pj->v );
  if (a > 0)
    return -1;
  else if (a < 0)
    return 1;
  else {/* Collinear with P[0] */
    x = abs( pi->v[X] - P[0].v[X] )-abs( pj->v[X] - P[0].v[X] );
    y = abs( pi->v[Y] - P[0].v[Y] )-abs( pj->v[Y] - P[0].v[Y] );

    if ( (x < 0) || (y < 0) ) {
      pi->delete = TRUE;
      return -1;
    }
    else if ( (x > 0) || (y > 0) ) {
      pj->delete = TRUE;
      return 1;
    }
    else {/* points are coincident */
      if (pi->vnum > pj->vnum)
          pj->delete = TRUE;
      else
          pi->delete = TRUE;
      return 0;
    }
    ndelete++;
  }
}
```

Code 3.5 `Compare`.

Main

Having worked out the sorting details, let us move to the top level and discuss `main` (Code 3.6). The points are read in, the rightmost lowest is swapped with `P[0]` in `FindLowest`, and `P[1], ..., P[n-1]` are sorted by angle with `qsort`. The repeated calls to `Compare` mark a number of points for deletion by setting the Boolean `delete` field. The next (easy) task is to delete those points, which we accomplish with a simple routine `Squash` (Code 3.7). This maintains two indices i and j into the points array P, copying `P[i]` on top of `P[j]` for all undeleted points i. After this the most problematic cases are gone,[14] and we can proceed with the Graham scan, with the call `top = Graham()`.

```
main()
{
  tStack   top;

  n = ReadPoints();
  FindLowest();
  qsort(
    &P[1],              /* pointer to 1st elem */
    n-1,                /* number of elems */
    sizeof( tsPoint ),  /* size of each elem */
    Compare             /* -1,0,+1 compare function */
  );
  Squash();

  top = Graham();
  PrintStack( top );
}
```

Code 3.6 `main`.

Code for the Graham Scan

The code for the Graham scan is a nearly direct translation of the while loop in the pseudocode presented earlier (Algorithm 3.6); see Code 3.8. We have the all too common situation where the majority of the coding effort is on the periphery, with relatively little needed for the heart of the geometric algorithm.

Example

The example in Figure 3.8 (a repeat of the points in Figure 3.6) was designed to be a stringent test of the code, as it includes collinearities of all types. The points after angular sorting, with marks for those to be deleted, are shown in Table 3.1. (The `vnum` indices shown accord with the labels in Figure 3.8.) After deleting five points in `Squash`, $n = 14$ points remain.

[14] Quite literally: If this code is embedding in another application, it may be necessary to avoid corrupting the original input by copying into another array.

```
void   Squash( void )
{
  int i, j;
  i = 0; j = 0;
  while ( i < n ) {
    if ( !P[i].delete ) {/* if not marked for deletion */
      Copy( i, j );        /* Copy P[i] to P[j]. */
      j++;
    }
    /* else do nothing: delete by skipping. */
    i++;
  }
  n = j;
}
```

Code 3.7 `Squash`. (The code for `Copy` is not shown.)

```
tStack   Graham()
{
  tStack top;
  int    i;
  tPoint p1, p2;   /* Top two points on stack. */

  /* Initialize stack. */
  top = NULL;
  top = Push ( &P[0], top );
  top = Push ( &P[1], top );

  /* Bottom two elements will never be removed. */
  i = 2;

  while ( i < n ) {
    p1 = top->next->p;
    p2 = top->p;
    if ( Left( p1->v , p2->v, P[i].v ) ) {
      top = Push ( &P[i], top );
      i++;
    }else
      top = Pop( top );
  }
  return top;
}
```

Code 3.8 `Graham`.

Table 3.1. Points in Figure 3.8 after sorting.

vnum	(x, y)	delete
15	(3, −2)	
13	(5, 1)	x
18	(7, 4)	
6	(6, 5)	
12	(4, 2)	
0	(3, 3)	x
1	(3, 5)	
3	(2, 5)	
16	(0, 5)	
2	(0, 1)	x
7	(−3, 4)	
4	(−2, 2)	
17	(0, 0)	x
5	(−3, 2)	
8	(−5, 2)	
14	(−5, 1)	
9	(−5, −1)	
10	(1, −2)	x
11	(−3, −2)	

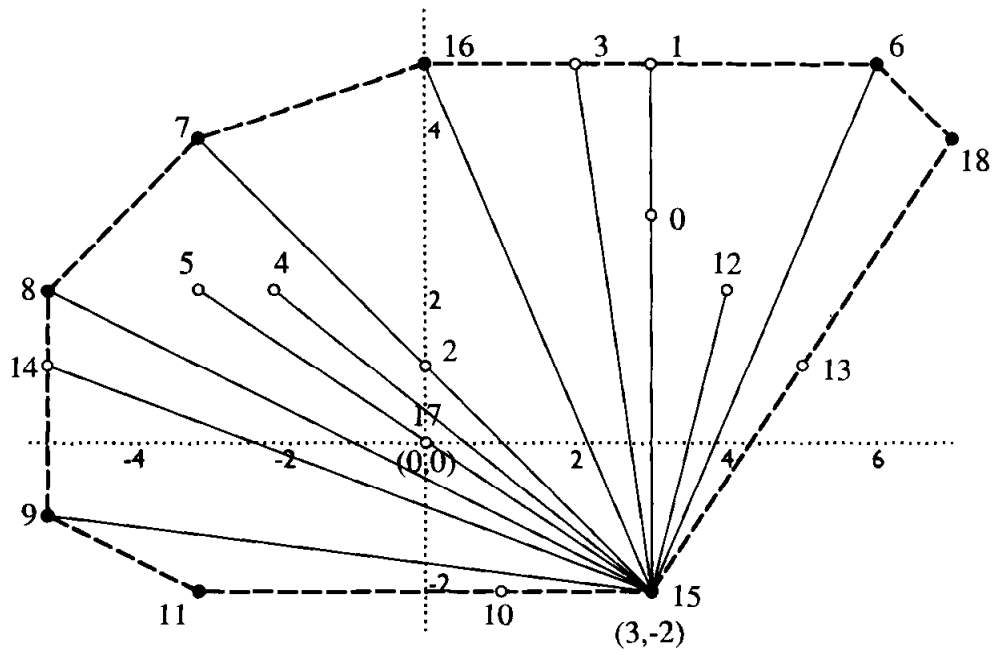


FIGURE 3.8 Graham Scan for Figure 3.6. Indices correspond to the coordinates in Table 3.1.

Below is shown the stack (point vnums only) and the value of `i` at the top of the `while` loop:

```
i = 2  :18, 15
i = 3  :6, 18, 15
i = 4  :12, 6, 18, 15
i = 4  :6, 18, 15
i = 5  :1, 6, 18, 15
i = 5  :6, 18, 15
i = 6  :3, 6, 18, 15
i = 6  :6, 18, 15
i = 7  :16, 6, 18, 15
i = 8  :7, 16, 6, 18, 15
i = 9  :4, 7, 16, 6, 18, 15
i = 9  :7, 16, 6, 18, 15
i = 10 :5, 7, 16, 6, 18, 15
i = 10 :7, 16, 6, 18, 15
i = 11 :8, 7, 16, 6, 18, 15
i = 12 :14, 8, 7, 16, 6, 18, 15
i = 12 :8, 7, 16, 6, 18, 15
i = 13 :9, 8, 7, 16, 6, 18, 15
i = 14 :11, 9, 8, 7, 16, 6, 18, 15
```

The stack is initialized to (p_{18}, p_{15}). Point p_6 and p_{12} are added to form $(p_{12}, p_6, p_{18}, p_{15})$, but then p_1 causes p_{12} to be deleted. After p_1 is pushed onto the stack, p_3 causes p_1 to be popped, because (p_6, p_1, p_3) is not a strict left turn (the three points are collinear). Then p_3 is removed by p_{16}. And so on. For this example, the total number of iterations is $19 < 2 \cdot n = 2 \cdot 14 = 28$. The final result is $(p_{11}, p_9, p_8, p_7, p_{16}, p_6, p_{18}, p_{15})$, the extreme points in clockwise order.

3.5.6. Conclusion

We summarize in a theorem.

Theorem 3.5.1. *The convex hull of n points in the plane can be found by Graham's algorithm in $O(n \log n)$ time: $O(n \log n)$ for sorting and no more than $2n$ iterations for the scan. His algorithm can be implemented entirely with integer arithmetic.*

3.5.7. Exercises

1. *All points collinear.* What will the code output if all input points are collinear?
2. *Best case.* How many iterations of the scan's while loop (in Algorithm 3.6) occur if the input points are all already on the hull?
3. *Worst case.* Construct a set of points for each n that causes the largest number of iterations of the while loop of the scan (in Algorithm 3.6).

4. *Random hulls* [programming]. Write code to generate random points in a square. Modify `graham.c` to output just the number of points read in and the number on the hull. Run this on sufficiently many random trials with large enough n to confirm the known theorem that the expected number is $O(\log n)$.[15]

3.6. LOWER BOUND

Having presented an $O(n \log n)$ algorithm to construct the convex hull, the next question is: Can we do better? It seems at least feasible that $O(n)$ might be possible. We show in this section that this is in fact *not* possible: $n \log n$ is a "lower bound" on the complexity of any algorithm that finds the hull. In technical language, the time complexity of any algorithm that constructs the convex hull is in the set $\Omega(n \log n)$;[16] this often is abbreviated by saying that hull construction has a lower bound of $\Omega(n \log n)$.

Researchers in computer science have found it fiendishly difficult to establish nontrivial lower bounds for problems. The difficulty is that the lower bound must hold for *any* conceivable algorithm, and it is hard to capture all algorithms in a proof. Nevertheless, this has been accomplished for a few key problems, notably sorting: $\Omega(n \log n)$ is a lower bound for sorting n elements.[17] Once a lower bound for one problem has been established, lower bounds for other problems can be proved via "problem reduction."

Suppose problem A is known to have some particular lower bound. One can view this as knowing that problem A is "hard" to the degree exhibited by the lower bound. So if A has a lower bound of $\Omega(n^4)$, it is a rather hard problem. Now suppose problem A can be *reduced* to problem B, in the sense that an algorithm for solving problem B can be used to solve problem A (with little additional work). A has been reduced to B in the sense that if we can solve B quickly, we can solve A quickly.[18] This often leads to a lower bound on B, since if we could solve B too quickly, the known lower bound on A would be violated.

This is how we will establish a lower bound on constructing the hull: We will show that if we had a fast algorithm for the hull, we could sort faster than $O(n \log n)$; but this is known to be impossible.

The original method of sorting using the hull is due to Shamos (1978), and is quite simple. Suppose we are given an unsorted list of numbers to be sorted, $(x_1, x_2, \ldots, x_n)$, $x_i \geq 0 \; \forall i$; this is problem A. Second, suppose we have an algorithm B that constructs the convex hull as a convex polygon (say, output (4), Section 3.1.1) of n points in $T(n)$ time. Now our task is to use B to solve A in time $T(n) + O(n)$, where the additional $O(n)$ represents the time to convert the solution of B to a solution of A.

[15]The theorem is due to Rényi-Sulanke, quoted in Preparata & Shamos (1985, Thm. 4.1). A more precise estimate is $(8/3)(\gamma + \ln n)$, where $\gamma \approx 0.57721566$ is the Euler–Mascheroni constant.

[16]In even more technical language, this set is defined for any function $f(n)$ as all those functions $g(n)$ that satisfy $\Omega(f(n)) = \{g(n) : \exists c$ such that $|g(n)| \geq cf(n)$ infinitely often$\}$.

[17]This statement is imprecise as it stands, and it would take us too far afield to make it airtight. Suffice it to say that the model of computation only permits comparisons between the input numbers, and the lower bound is on the number of such comparisons necessary.

[18]It is a frequent confusion of algorithm novices to assimilate this definition backwards, and to say mistakenly that B is reduced to A.

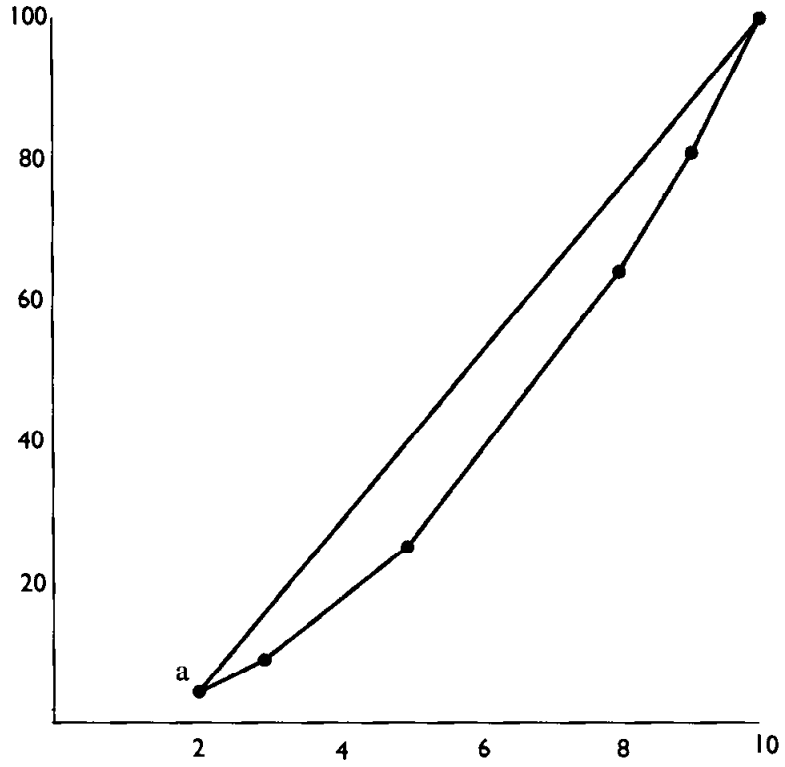


FIGURE 3.9 Parabola construction for sorting (2, 3, 5, 8, 9, 10).

Form a set of two-dimensional points (x_i, x_i^2), as shown in Figure 3.9. These points fall on a parabola. Run algorithm B to construct the hull. Clearly, every point is on the hull. Identify the lowest point a on the hull in $O(n)$ time; this corresponds to the smallest x_i. The order in which the points occur on the hull counterclockwise from a is their sorted order. Thus we can use a hull algorithm to sort.

Note that it is crucial for this reduction that algorithm B yields the vertices of the hull in order around the boundary. It is not at all clear how to perform a similar reduction from sorting to the problem of identifying the points of the hull in arbitrary order (outputs (1) or (2)). It remained an open problem for several years to determine if finding extreme points was easier, but the work of several researchers finally established that $\Omega(n \log n)$ is a lower bound on this problem also.[19]

3.7. INCREMENTAL ALGORITHM

Having arrived at an optimal $O(n \log n)$ algorithm (Section 3.5), it may seem there is no point in exploring additional algorithms. But there is motivation: extension to three (and higher) dimensions. The convex hull is if anything more useful in three dimensions than in two, and we will spend the next chapter exploring two algorithms for constructing three-dimensional hulls. The difficulty is that Graham's algorithm has no obvious extension to three dimensions: It depends crucially on angular sorting, which has no direct counterpart in three dimensions. So we now proceed to describe two further algorithms in two dimensions, each of which extends to three dimensions.

[19]See Preparata & Shamos (1985, pp. 101–3).

The first algorithm is one of the most straightforward imaginable: The incremental algorithm. Its basic plan is simple: Add the points one at a time, at each step constructing the hull of the first k points and using that hull to incorporate the next point. It turns out that "factoring" the problem this way simplifies it greatly, in that we only have to deal with one very special case: adding a single point to an existing hull.

Let our set of points be $P = \{p_0, p_1, \ldots, p_{n-1}\}$, and assume for simplicity of exposition that the points are in general position: No three are collinear. The high-level structure is shown in Algorithm 3.7.

Algorithm: INCREMENTAL ALGORITHM
Let $H_2 \leftarrow \text{conv}\{p_0, p_1, p_2\}$.
for $k = 3$ to $n - 1$ do
    $H_k \leftarrow \text{conv}\{H_{k-1} \cup p_k\}$

Algorithm 3.7 Incremental.

The first hull is the triangle $\text{conv}\{p_0, p_1, p_2\}$. Let $Q = H_{k-1}$ and $p = p_k$. The problem of computing $\text{conv}\{Q \cup p\}$ naturally falls into two cases, depending on whether $p \in Q$ or $p \notin Q$.[20]

1. $p \in Q$.
   Of course once p is determined to be in Q, it can be discarded. Note that we can discard p even if it is on the boundary of Q (assuming our goal is output (4)). Although there are several ways to decide if $p \in Q$, perhaps the most robust is to use `LeftOn` from Chapter 1 (Code 1.6): $p \in Q$ iff p is left of or on every directed edge.[21] Clearly this test takes time linear in the number of vertices of Q.
2. $p \notin Q$.
   If any `LeftOn` test returns false, then $p \notin Q$, and we have to compute $\text{conv}\{Q \cup p\}$. What makes this task relatively easy is that we need only find the two lines of tangency from p to Q, as shown in Figure 3.10, and modify the hull accordingly. Our general position assumption assures that each line of tangency between p and Q touches Q at just one point. Suppose p_i is one such point of tangency. How can we find p_i?

   Examination of Figure 3.10 shows that we can use the results of the `LeftOn` tests to determine tangency. For the lower point of tangency p_i, p is left of $p_{i-1}p_i$ but right of $p_i p_{i+1}$. For the upper point of tangency p_j, the sense is reversed: p is right of $p_{j-1}p_j$ but left of $p_j p_{j+1}$. Both cases can be captured with the exclusive-or (Xor) function: p_i is a point of tangency if two successive edges yield different `LeftOn` results (Algorithm 3.8). Thus the two points of tangency can be identified via the same series of `LeftOn` tests used to decide if $p \in Q$.

[20] We could ensure that $p \notin Q$ by presorting the p_i's by x coordinate (for example), as suggested by Edelsbrunner (1987, p. 143). See Exercise 3.7.1[3].

[21] Note that this method only works for convex Q, which is all we need. More general point-in-polygon algorithms will be discussed in Section 7.4.

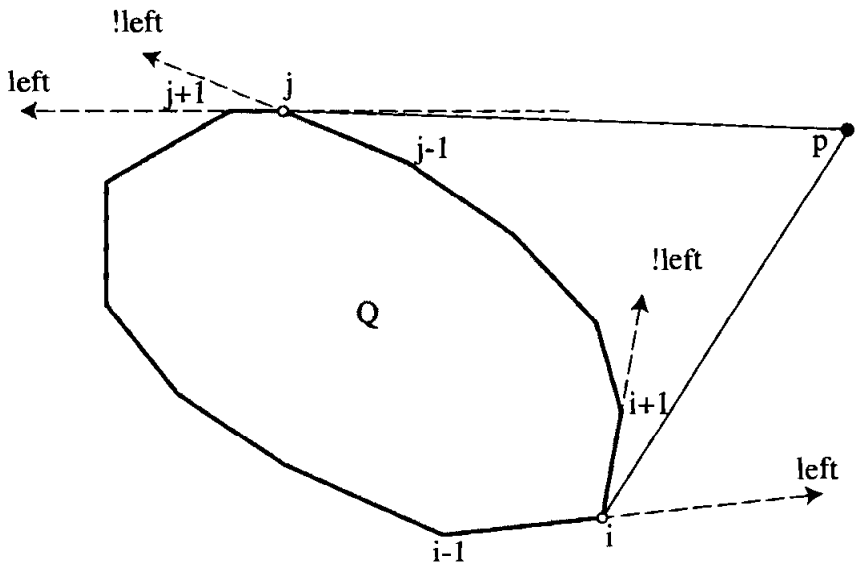


FIGURE 3.10 Tangent lines from p to Q; "left" means that p is left of the indicated directed line, and "!left" means "not left."

It only remains to form the new hull. In Figure 3.10, the new hull is

$$(p_0, p_1, \ldots, p_{i-1}, p_i, p, p_j, p_{j+1}, \ldots, p_{n-1}).$$

If the hulls are represented by linked lists, this update can be accomplished by a simple sequence of insertions and deletions.

Algorithm: TANGENT POINTS
for $i = 0$ to $n - 1$ do
    if Xor (p left or on (p_{i-1}, p_i), p left or on (p_i, p_{i+1}))
        then p_i is point of tangency

Algorithm 3.8 Tangent points.

We leave as an exercise (Exercise 3.7.1[1]) exploring how to proceed when a line of tangency from p is flush with an edge of Q.

The complexity analysis of this algorithm is simple: The work at each step is $O(n)$; more precisely, it is proportional to the number of vertices of the kth hull. In the worst case we would have $p \notin Q$ at each step, resulting in total work proportional to $3 + 4 + \cdots + n = O(n^2)$.

It turns out that with only a little more effort, the time complexity can be reduced to $O(n \log n)$ (Exercise 3.7.1[3]).

3.7.1. Exercises

1. *Degenerate tangents.* Modify the incremental algorithm as presented to output the correct hull when a tangent line from p includes an edge of Q. The "correct" hull should not have three collinear vertices.
2. *Collinearities.* Modify the incremental algorithm to work with sets of points that may include three or more collinear points, extending [1] above.

3. *Optimal incremental algorithm.* Presort the points by their x coordinate, so that $p \notin Q$ at each step. Now try to arrange the search for tangent lines in such a manner that the total work over the life of the algorithm is $O(n)$. This then provides an $O(n \log n)$ algorithm.[22]

3.8. DIVIDE AND CONQUER

The final two-dimensional hull algorithm we consider achieves optimal $O(n \log n)$ time by a method completely different than Graham's algorithm: divide and conquer. This is the only technique known to extend to three dimensions and achieve the same asymptotically optimal $O(n \log n)$ time complexity. It is therefore well worth studying, even though in two dimensions it is relatively complicated.

3.8.1. Divide-and-Conquer Recurrence Relation

"Divide and conquer" is a general paradigm for solving problems that has proved very effective in computer science. The essence is to partition the problem into two (nearly) equal halves, solve each half recursively, and create a full solution by "merging" the two half solutions. When the recursion reduces the original problem down to very small subproblems, they are usually quite easy to solve. All the work therefore lies in the merge step.

Let $T(n)$ be the time complexity of a divide-and-conquer hull algorithm for n points. If the merge step can be accomplished in linear time, then we have the recurrence relation $T(n) = 2T(n/2) + O(n)$: The two problems of half size take $2T(n/2)$ time, and the merge takes $O(n)$ time. As we mentioned before, this has solution $O(n \log n)$.

Exercises 3.8.5[1] and [2] ask for exploration of the recurrence relation when the merge step is less efficient; the conclusion is that the merge must be $O(n)$ to achieve optimality.

3.8.2. Algorithm Description

The divide-and-conquer technique was first applied to the convex hull problem by Preparata & Hong (1977), whose goal was to create an efficient algorithm for three dimensions.

To keep the explanation simple, we assume two types of nondegeneracy: No three points are collinear, and no two points lie on a vertical line. The outline of their algorithm is as follows:

1. Sort the points by x coordinate.
2. Divide the points into two sets $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ containing the left $\lceil n/2 \rceil$ points, and $\mathcal{B}$ the right $\lfloor n/2 \rfloor$ points.
3. Compute the convex hulls $A = \mathcal{H}(\mathcal{A})$ and $B = \mathcal{H}(\mathcal{B})$ recursively.
4. Merge A and B: Compute conv $\{A \cup B\}$.

[22]This idea is due to Edelsbrunner (1987, p. 144).

The sorting step (1) guarantees the sets $\mathcal{A}$ and $\mathcal{B}$ will be separated by a vertical line (by our assumption that no two points lie on a vertical), which in turn guarantees that A and B will not overlap. This simplifies the merge step. Steps 2, 3, and 4 are repeated at each level of the recursion, stopping when $n \leq 3$; if $n = 3$ the hull is a triangle by our assumption of noncollinearity.

All of the work in this divide-and-conquer algorithm resides in the merge step. For this algorithm it is tricky to merge in linear time: The most naive algorithm would only achieve $O(n^2)$, which as we mentioned is not sufficient to yield $O(n \log n)$ performance overall.

We will use a and b as indices of vertices of A and B respectively, with the vertices of each ordered counterclockwise and numbered from 0. All index arithmetic is modulo the number of vertices in the relevant polygon, so that expressions like $a+1$ and $a-1$ can be interpreted as the next and previous vertices around A's boundary, respectively. The goal is to find two tangent lines, one supporting the two hulls from below, and one supporting from above. From these tangents, conv$\{A \cup B\}$ is easily constructed in $O(n)$ time. We will only discuss finding the lower tangent; the upper tangent can be found analogously.

Let the lower tangent be $T = ab$. The difficulty is that neither endpoint of T is known ahead of time, so a search has to move on both A and B. Note that the task in the incremental algorithm was considerably easier because only one end of the tangent was unknown; the other was a single, fixed point. Now a naive search for all possible a endpoints and all possible b endpoints would result in a worst-case quadratic merge step, which is inadequate. So the search must be more delicate.

The idea of Preparata and Hong is to start T connecting the rightmost point of A to the leftmost point of B, and then to "walk" it downwards, first advancing on one hull, then on the other, alternating until the lower tangent is reached. See Figure 3.11. Pseudocode is displayed in Algorithm 3.9. Note that $a-1$ is clockwise on A, and $b+1$ is counterclockwise on B, both representing downward movements. $T = ab$ is a lower tangent at a if both $a-1$ and $a+1$ lie above T; this is equivalent to saying that both of these points are left or on ab. A similar definition holds for lower tangency to B. And again we assume for simplicity of exposition that lines are always tangent at a point, rather than along an edge.

```
Algorithm: LOWER TANGENT
a ← rightmost point of A.
b ← leftmost point of B.
while T = ab not lower tangent to both A and B do
    while T not lower tangent to A do
        a ← a − 1
    while T not lower tangent to B do
        b ← b + 1
```

Algorithm 3.9 Lower tangent.

An example is shown in Figure 3.11. Initially, $T = (4, 7)$; note that T is tangent to both A and B, but it is only a lower tangent for A. The A loop does not execute, but the

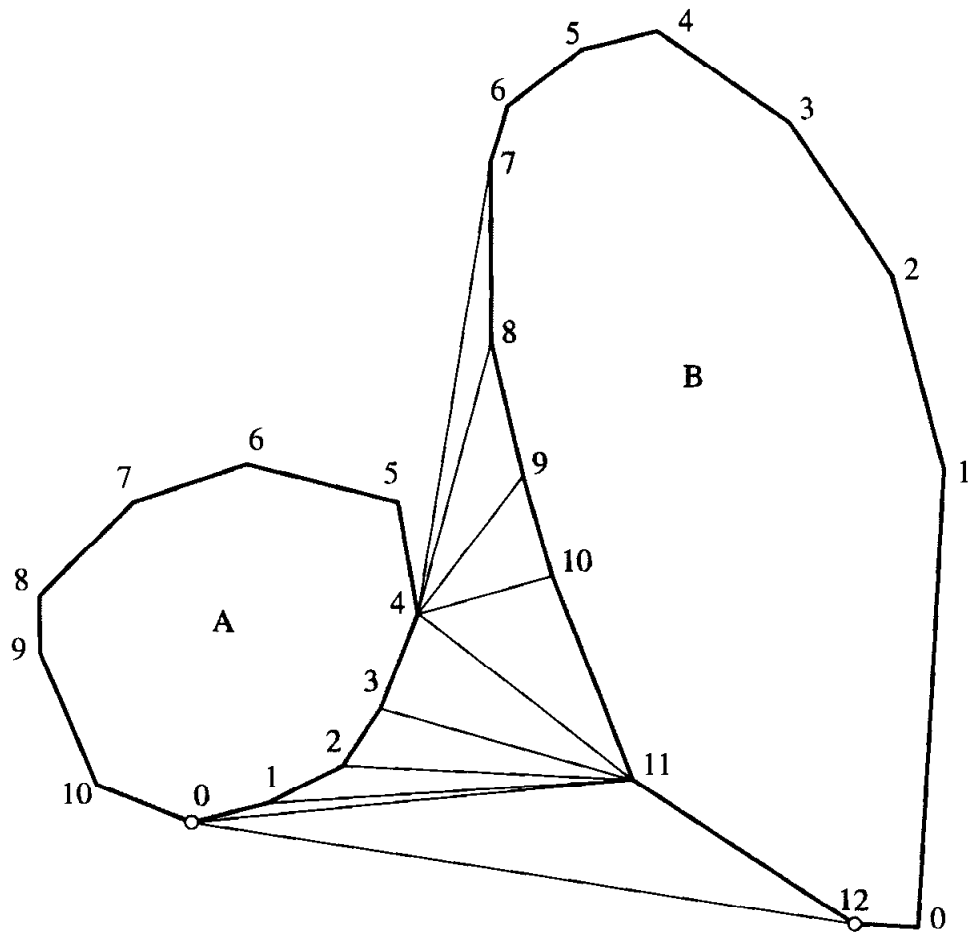


FIGURE 3.11 Finding the lower tangent: from (4, 7) to (0, 12).

B loop increments b to 11, at which time $T = (4, 11)$ is a lower tangent for B. But now T is no longer a lower tangent for A, so the A loop decrements a to 0; now $T = (0, 11)$ is a lower tangent for A. This is not a lower tangent for B, so b is incremented to 12. Now $T = (0, 12)$ is a lower tangent for both A and B, and the algorithm stops. Note that $b = 12$ is not the lowest vertex of B: 0 is slightly lower.

3.8.3. Analysis

Neither the time complexity nor the correctness of the hull-merging algorithm are evident. Certainly when the outermost **while** loop terminates, T is tangent to both A and B. But two issues remain:

1. The outer loop must terminate: It is conceivable that establishing tangency at a always breaks tangency at b and vice versa.
2. There are four mutual tangents (see Figure 3.12), and the algorithm as written should only find one, the one that supports both A and B to its left.

Lemma 3.8.1. *The lower tangent $T = ab$ touches both A and B on their lower halves: Both a and b lie on the closed chain from the leftmost vertex counterclockwise to the rightmost vertex, of A and B respectively.*

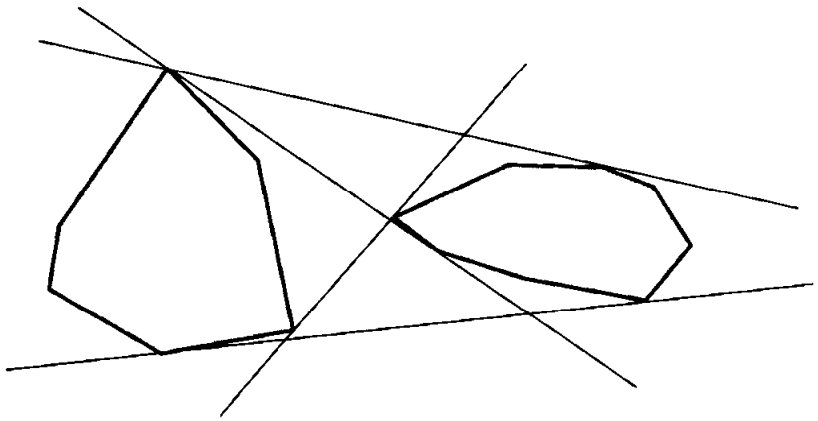


FIGURE 3.12 Four mutual tangents.

Proof. This is a consequence of the horizontal separation of A and B. Let L be a vertical line separating them. Then if B is very high above A, T approaches being parallel to L, and a approaches the rightmost vertex of A. Similarly if B is very low below A, a approaches the leftmost vertex of A. Between these extremes, a lies on the lower half of A. □

Since a starts at the rightmost vertex, and is only decremented (i.e., moved clockwise), the inner A loop could only iterate infinitely if a could pass the leftmost vertex. However, the next lemma will show this is not possible.

Lemma 3.8.2. *Throughout the life of Algorithm 3.9, ab does not meet the interior of A or of B.*

Proof. The proof is by induction. The statement is true at the start of the algorithm.

Suppose it is true after some step, and suppose that a is about to be decremented, which only happens when T is not a lower tangent to A. The new tangent $T' = (a-1, b)$ could only intersect the interior of A if b is left of $(a-1, a)$; see Figure 3.13. But b could not also be left of $(a, a+1)$, for then T would have intersected A, which we assumed

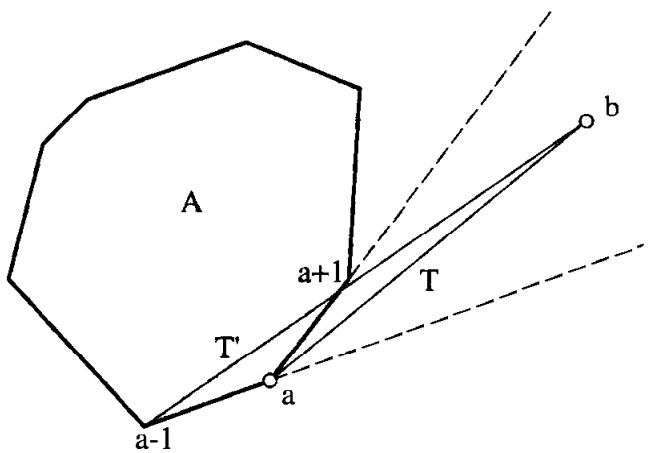


FIGURE 3.13 If T' intersects A, then T must be a lower tangent at a.

by induction was not the case. So if the next step would cause intersection, T must be tangent at a, and the next step would not be taken. □

Now because T does not meet the interior of A, and because b is right of L (the line separating A and B), a cannot advance (clockwise) beyond the leftmost vertex of A. Similar reasoning applies to B. Therefore both loops must terminate eventually, since they each move the indices in one direction only, and there is a limit to how far the indices can move. Now everything we need for correctness follows. Because the loops terminate, they must terminate with double tangency, which clearly must be the lower tangent.

The loops can only take linear time, since they only advance, never back up, and the number of steps is therefore limited by the number of vertices of A and B. Therefore the merge step is linear, which implies that the entire algorithm is $O(n \log n)$.

3.8.4. Output-Size Sensitive Optimal Algorithm

As a function of n, this asymptotic time complexity cannot be improved, in view of the lower bound (Section 3.6). But recall that the gift wrapping and QuickHull algorithms are output-size sensitive, running in time $O(nh)$ for hulls of size h. Examining the lower bound proof more closely shows it to establish $\Omega(n \log h)$. The divide-and-conquer algorithm does not match this more refined bound, which opened the possibility that it was not "the ultimate planar convex hull algorithm." Kirkpatrick & Seidel (1986) published a paper with this title (suffixed by '?'), seven years after the Preparata and Hong algorithm. They introduced a novel variation of the divide-and-conquer paradigm they called "marriage-before-conquest" that leads to an $O(n \log h)$ time complexity. The algorithm constructs the upper and lower hulls separately. Rather than conquering the subproblems after dividing, it finds an upper tangent of the two sets prior to finding their hulls recursively. Finding this "bridge" in linear time is tricky, but knowing it permits all the points vertically underneath the bridge and between its endpoints to be discarded before recursing. This partial discard is the key to achieving the lower asymptotic time complexity. However, it is questionable whether this theoretical improvement is worth the additional programming complexity in practical situations.

3.8.5. Exercises

1. *Recurrence relation with $O(n^2)$ merge.* Solve the recurrence relation $T(n) = 2T(n/2) + O(n^2)$.
2. *Recurrence relation with $O(n \log n)$ merge.* Solve the recurrence relation $T(n) = 2T(n/2) + O(n \log n)$.
3. *Degeneracies.* Remove all assumptions of nondegeneracy from the divide-and-conquer algorithm by considering the following possibilities:
   a. Several points lie on the same vertical line.
   b. A tangent line T is collinear with an edge of A and/or B.
   c. The recursion bottoms out with three collinear points.
4. *Merge without sorting* (Toussaint 1986). If the sorting step of the divide-and-conquer algorithm is skipped, the hulls to be merged might intersect. Design an algorithm that can merge two arbitrarily located hulls of n and m vertices in $O(n + m)$ time. This then provides an alternative $O(n \log n)$ divide-and-conquer algorithm.

3.9. ADDITIONAL EXERCISES

3.9.1. Polygon Hull

1. *Hull of monotone polygon.* Develop an algorithm to find the convex hull of a monotone polygon in linear time.
2. *Hull of polygon* [difficult]. Develop an algorithm to find the convex hull of an arbitrary polygon in linear time. (Note that the lower bound in Section 3.6 holds for sets of unorganized points, not for the vertices of a polygon.) This is quite tricky, and several published algorithms for this problem were later discovered to be flawed.

3.9.2. Orthogonal Polygons

1. *Orthogonally convex polygons: characterization.* The standard definition of a convex polygon is a polygon P for which the line segment connecting any two points in P lies entirely within P (Section 3.1(1)). This definition is equivalent to: The intersection of P with any (infinite) line L has at most one connected component – it is either empty, a line segment, or a point. The only truly convex orthogonal polygon is a rectangle. But we can generalize the second definition of convexity in a natural way to define "orthogonally convex" to include more than just the rectangle.

   Define an orthogonal polygon to be *orthogonally convex* if its intersection with any vertical or horizontal line has at most one connected component. Characterize the shape of orthogonal convex polygons. By "characterize the shape" I mean state and prove something like "an orthogonal polygon is orthogonally convex if and only if every two convex vertices have at least one reflex vertex between them, and there is one vertical edge whose length is the square root of the sum of the lengths of the horizontal edges." This is manifestly false but indicative of what constitutes a characterization. Of course, the definition of orthogonal convexity itself provides a characterization: "Its intersection with any vertical or horizontal line is either empty or a line segment." But there are other ways to characterize the shape, some of which will help in the next exercise.
2. *Orthogonal convex polygons: test algorithm.* Design an algorithm to test whether a given orthogonal polygon is orthogonally convex, based on your characterization above. Argue for its correctness (perhaps invoking your characterization) and analyze its time complexity as a function of n.
3. *Orthogonal convex hull.* Let P be an orthogonal polygon. Define the *orthogonal convex hull* $\overline{\mathcal{H}}(P)$ of an orthogonal polygon P as the smallest orthogonally convex polygon that encloses P. (See [1] above for a definition of orthogonally convex.) If P is already orthogonally convex, then $\overline{\mathcal{H}}(P) = P$. Otherwise, $\overline{\mathcal{H}}(P)$ encloses P. In the example in Figure 3.14, P is shown in dark lines, and $\overline{\mathcal{H}}(P)$ is shown shaded.

   Design an algorithm to compute $\overline{\mathcal{H}}(P)$. Assume as input a list of the coordinates of the vertices of P, given in counterclockwise order around the boundary of P. Produce as output a similar list for $\overline{\mathcal{H}}(P)$. Do not assume that no two edges occur on the same vertical or horizontal line; in fact most applications that produce orthogonal polygons select their vertices from an integer grid, and it is quite likely that several lie on the same grid line (as in Figure 3.14).

   Hand-execute your algorithm on at least one nontrivial example.
4. *Orthogonal star polygons: characterization.* A polygon P is *star* if there exists a point $x \in P$ that can see every point in the polygon (Exercise 1.1.4[5]). (The line of sight does not have to be vertical or horizontal; it may be at any angle.) Characterize (state and prove) the shape of *orthogonal* star polygons, star polygons composed of horizontal and vertical edges.

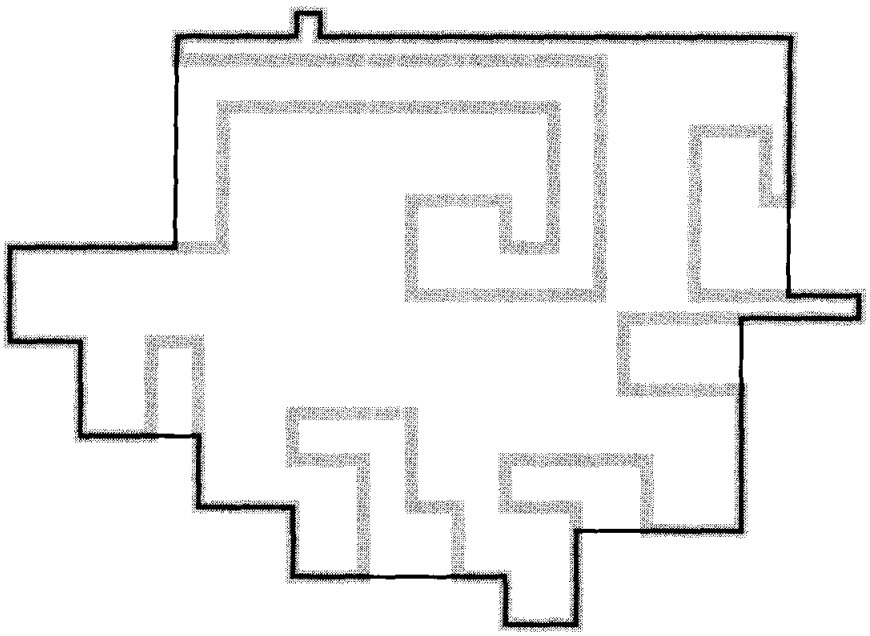


FIGURE 3.14 The orthogonal convex hull of an orthogonal polygon.

5. *Orthogonal star kernels: algorithm.* The *kernel* of a star polygon is the set of all points that can see every point in the polygon. Design an algorithm for constructing the kernel of an orthogonal star polygon. Argue for its correctness (perhaps invoking your characterization) and analyze its time complexity as a function of n.
6. *Krasnosselsky's theorem.*[23] Characterize (state and prove) the shape of an orthogonal polygon P for which the following holds: For every two points a and b on the boundary of P, there exists a point $x \in P$ that can see both a and b. (Note that x is (or may be) dependent on a and b.)

3.9.3. Miscellaneous Hull-Related

1. *Distance between convex polygons.* Let A and B be two convex polygons. Define two distance concepts as follows:

$$\delta = \min_{x \in A, y \in B} |x - y|,$$

$$\Delta = \max_{x \in A, y \in B} |x - y|,$$

where $|x - y|$ is the Euclidean distance between the points x and y. Design algorithms to compute both δ and Δ. You may assume that $A \cap B = \emptyset$.

First establish a few geometric lemmas that characterize the type of points that can achieve the minimum or maximum distance. Can the points be interior to A or B, or must they be on the boundary? If the latter, must one or both be vertices, that is, can either or both lie on the interior of an edge? The answers to these questions are not necessarily the same for both types of distance measures.

Try to achieve $O(n)$ for computing Δ and $O(\log n)$ for computing δ. The former is not that difficult, but the latter is rather tricky (Edelsbrunner 1985), depending on the fact that the median of n numbers can be found in linear time. It is somewhat surprising that δ can be computed in less than linear time.

[23] Krasnosselsky's theorem states that for any compact set S in d dimensions containing at least $d + 1$ points if each $d + 1$ point of S are visible to one point, then S is star-shaped (Lay 1982, p. 53).

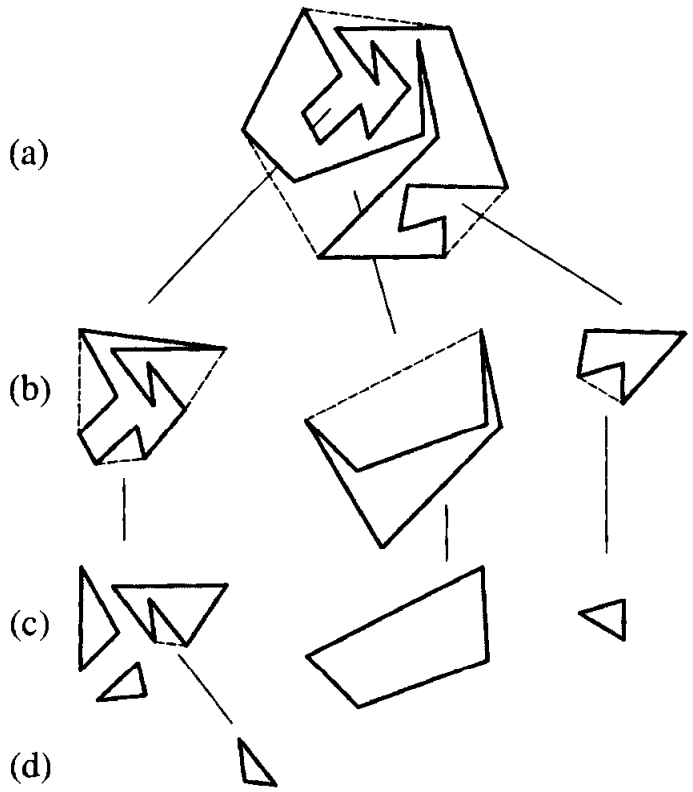


FIGURE 3.15 (a) P and $\mathcal{H}(P)$; (b) $D(P)$ and hulls; (c) deficiencies of polygons from (b); (d) deficiency of the one nonconvex piece from (c).

2. *Convex deficiency tree.* Let P be a polygon (with no holes) and $\mathcal{H}(P)$ its convex hull, where as usual both are considered closed regions in the plane. Define the *convex deficiency* of P, $D(P)$, to be the set of points $\mathcal{H}(P) \setminus P$, that is, the set difference between the hull and the polygon.[24] In general this set will have several disconnected components, which I will call *pockets*. Each of these pockets has the shape of a polygon, as shown in Figure 3.15(b). Technically these are partially "open" sets, because portions of their boundaries were subtracted away. To clean up this minor blemish, redefine D to fill in the boundaries by taking the *closure*. Let $\overline{Q}$ be the closure of Q. Then

$$D(P) = \overline{\mathcal{H}(P) \setminus P}.$$

Define the *deficiency tree* $T(P)$ for a polygon P as follows. The root of T is a node associated with P. The children of the root are nodes associated with the distinct connected components of $D(P)$. And in general, if P' is the set of points corresponding to a node of T, the children of the node correspond to the distinct connected components of $D(P')$. Define $\mathrm{P}(N)$ to be the parent of a node N. Define the depth of a node of T in the usual manner: The depth of the root is 0, and the depth of any node N is 1 greater than the depth of $\mathrm{P}(N)$.

These exercises ask you to explore the properties of the convex deficiency tree.

a. Is $T(P)$ a tree in fact? Is it always finite for a polygon P? What if P were permitted to have curved edges?

b. Let us identify a node and its associated region, so that phrases such as "the area of a node" make sense. Is the area of a node N always less than or equal to the area of $\mathrm{P}(N)$? Is the sum of the areas of all the children of N less than or equal to the area of N?

[24]This concept was introduced by Batchelor (1980). The "\" symbol is used for set subtraction.

c. Is there any relationship between the number of vertices of N and $\mathrm{P}(N)$, or between N and the sum of the number of vertices in all the children of N?

d. For a polygon P of n vertices, what is the maximum possible *breadth* of $T(P)$ as a function of n, that is, what is the largest possible degree of a node? What is the maximum possible depth of $T(P)$ as a function of n? In both cases, show worst-case examples.

e. Can you write one equation that expresses the set of points represented by the root in terms of all of its descendants? Using set unions and differences of regions? Using sums and differences of areas?

f. Do two polygons with the same deficiency tree (viewed as a combinatorial object, i.e., without consideration of the specific regions associated with each node) necessarily have similar shapes? Can you find polygons with identical trees with wildly different shapes? This is an imprecise question because "shape" is left intuitive. So reask the questions with shape replaced by "number of vertices": Must two polygons with the same deficiency tree have the same number of vertices? Is there any feature shared by the class of polygons with the same deficiency tree?

g. Can every tree be realized as the convex deficiency tree of some polygon? Given an arbitrary tree, how can you construct a representative "realizing" polygon, a polygon whose convex deficiency tree is the given tree?

h. Can the concept of deficiency tree be extended to include polygons with holes?

i. Does the concept of deficiency tree make sense for three-dimensional polyhedra? Is it a tree? Is it finite?

3. *Diameter and width.* Define the *diameter* of a set of points $\{p_0, p_1, \ldots\}$ to be the largest distance between any two points in the set: $\max_{i,j} |p_i - p_j|$.

a. Prove that the diameter of a set is achieved by two hull vertices.

b. A *line of support* to a set is a line L that touches the hull and has all points on or to one side of L. Prove that the diameter of a set is the same as the maximum distance between parallel lines of support for the set.

c. Two points a and b are called *antipodal* if they admit parallel lines of support: There are parallel lines of support through a and b. Develop an algorithm for enumerating (listing) all antipodal pairs of a set of points in two dimensions. It helps to view the lines of support as jaws of a caliper. An algorithm can be based on the idea of rotating the caliper around the set.[25]

d. Define the *width* as the minimum distance between parallel lines of support. Develop an algorithm for computing the width of a set of points in two dimensions.

4. *Onion peeling.* Start with a finite set of points $S = S_0$ in the plane, and consider the following iterative process. Let S_1 be the set $S_0 \setminus \partial\mathcal{H}(S_0)$: S minus all the points on the boundary of the hull of S. Similarly, define $S_{i+1} = S_i \setminus \partial\mathcal{H}(S_i)$. The process continues until the empty set is reached. The hulls $H_i = \partial\mathcal{H}(S_i)$ are called the *layers* of the set, and the process of peeling away the layers is called *onion peeling* for obvious reasons. See Figure 3.16. Any point on H_i is said to have *onion depth*, or just *depth*, i. Thus the points on the hull of the original set have depth 0.

a. For each n, determine the maximum number of layers for any set of n points in two dimensions.

[25]This idea is due to Toussaint (1983*b*).

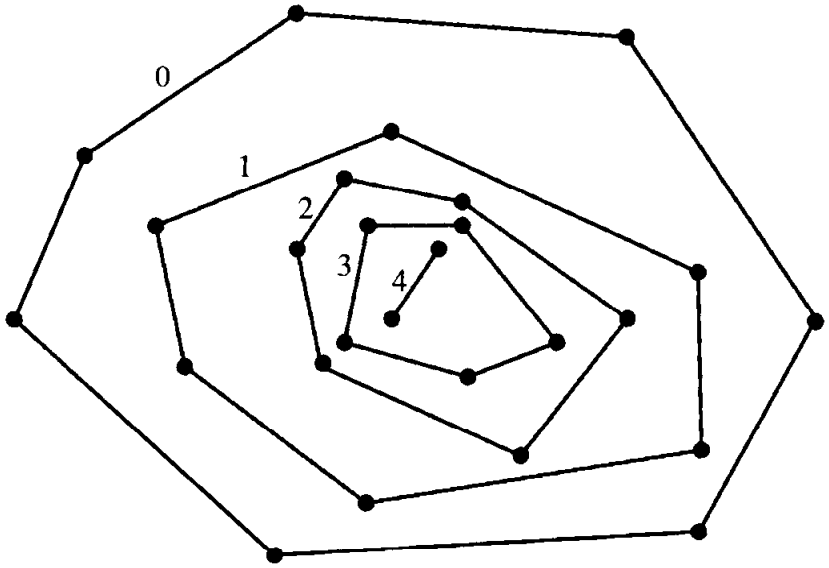


FIGURE 3.16 The onion layers of a set of points, labeled with depth numbers.

b. For a polygon P, define its *depth sequence* to be the sequence of onion depths of its vertices (a list of integers) in a counterclockwise traversal of the boundary.[26] Express some gross constraints on the form of a depth sequence.

c. Is every sequence that meets your constraints realizable by some polygon? If not, find an unrealizable sequence. If so, provide an argument.

[26]These sequences were introduced by Abellanas, García, Hernández, Hurtado & Serra (1992).

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.

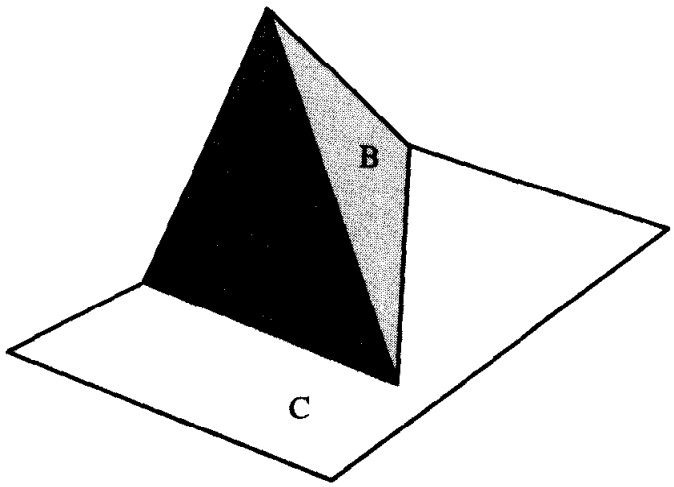


FIGURE 4.1 Faces A and B meet C improperly even though they do not penetrate C.

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).

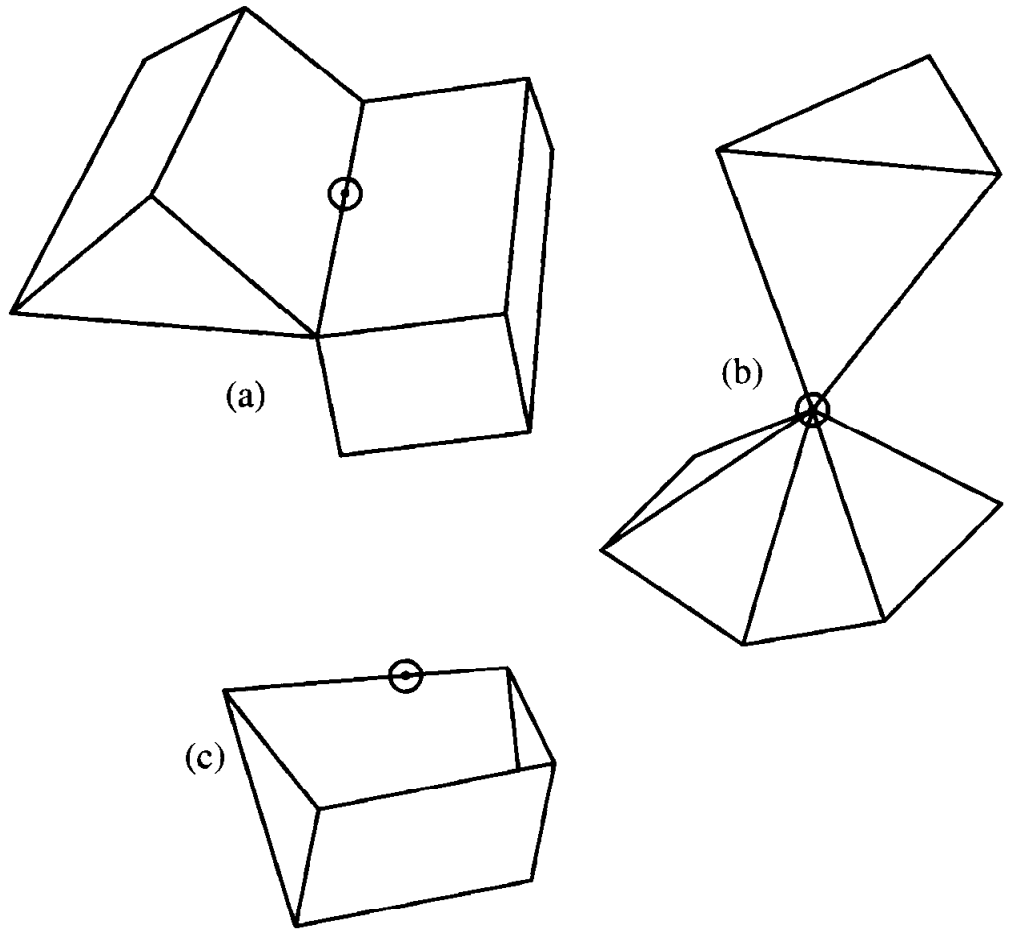


FIGURE 4.2 Three objects that are not polyhedra. In all three cases, a neighborhood of the circled point is not homeomorphic to an open disk. In (a) the point lies both on the top surface shown and on a similar surface underneath. Object (c) is not closed, so the indicated point's neighborhood is a half-disk.

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	{3, 5}	20	30	12
Icosahedron	{5, 3}	12	30	20

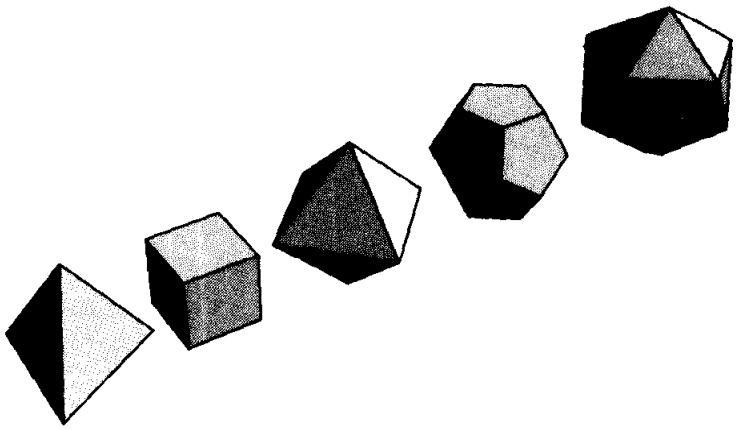


FIGURE 4.3 The five Platonic solids (left to right): tetrahedron, cube, octahedron, dodecahedron, and icosahedron.

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.

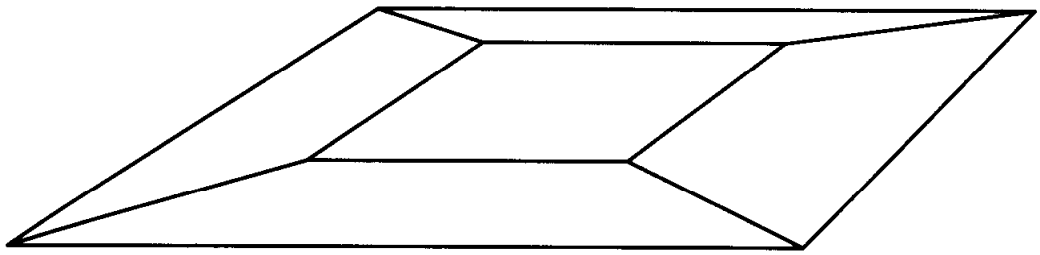


FIGURE 4.4 The 1-skeleton of a cube, obtained by flattening to a plane.

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."

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 $\text{conv}\{p, e\}$ 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

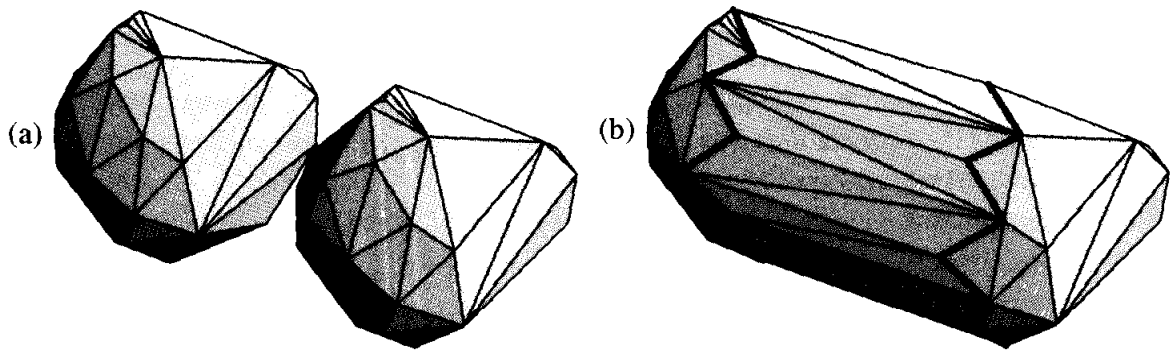


FIGURE 4.5 (a) Polytopes prior to merge. In this example, A and B are congruent, although that will not be true in general. (b) conv $\{A \cup B\}$. The dark edges show the "shadow boundary": the boundary of the newly added faces.

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.

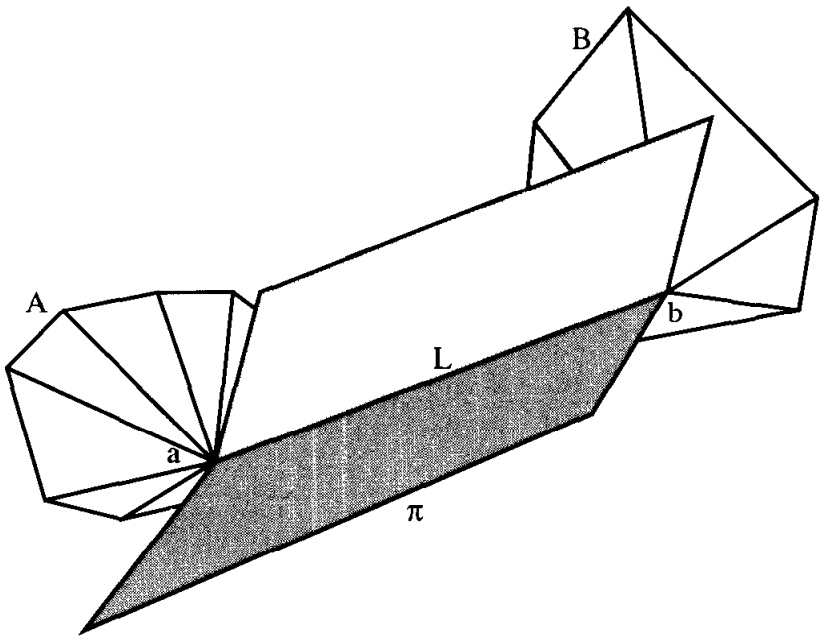


FIGURE 4.6 Plane π is creased along L and bent toward the polytopes A and B (only the faces incident to a and b are shown).

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 :e 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 $\text{conv}\{A \cup B\}$ 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)

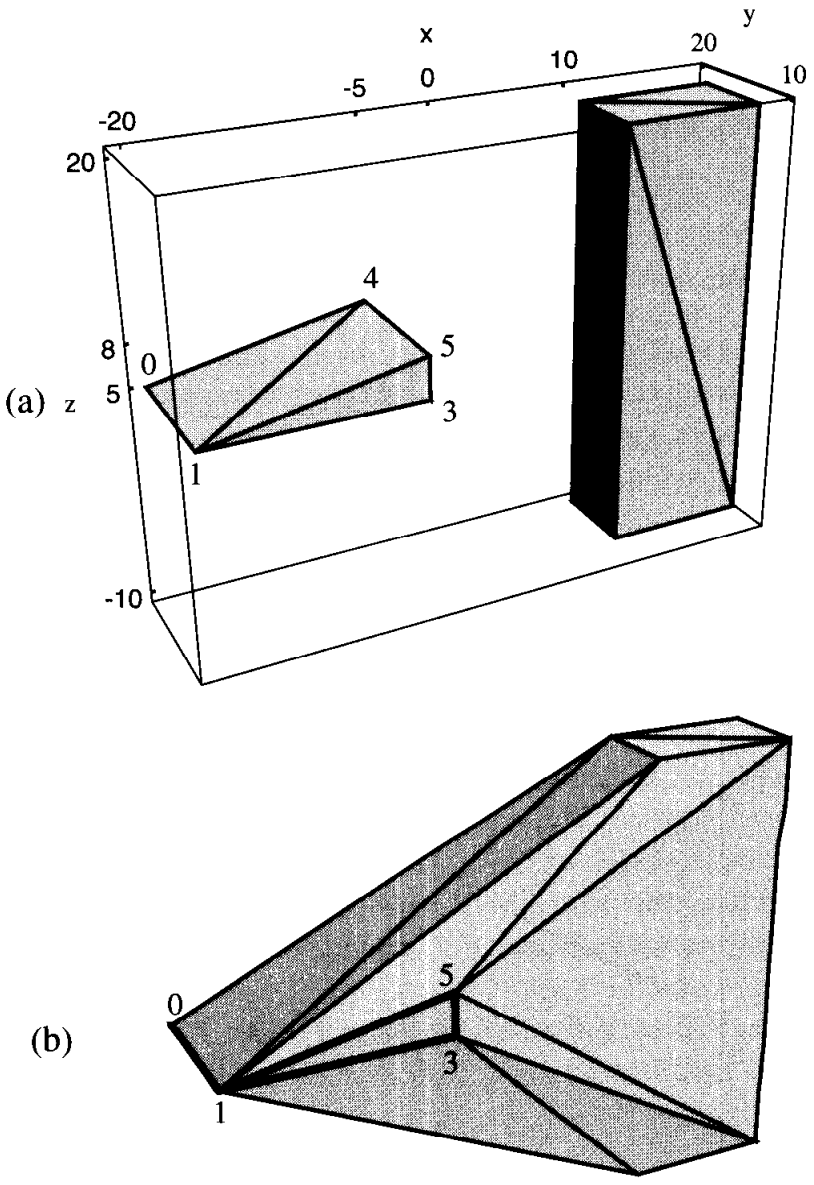


FIGURE 4.7 (a) Wedge and block prior to merging; (b) hull of wedge and block.

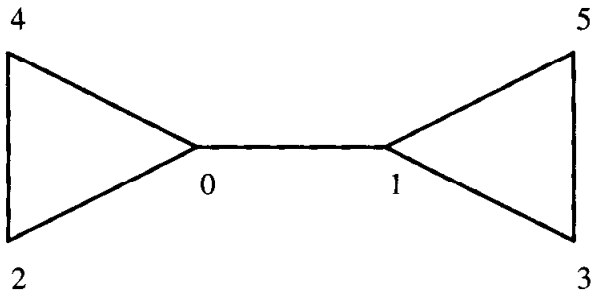


FIGURE 4.8 Topology of shadow boundary edges for Figure 4.7(b).

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 $\text{conv}\{A \cup B\}$ has the topology shown in Figure 4.9.

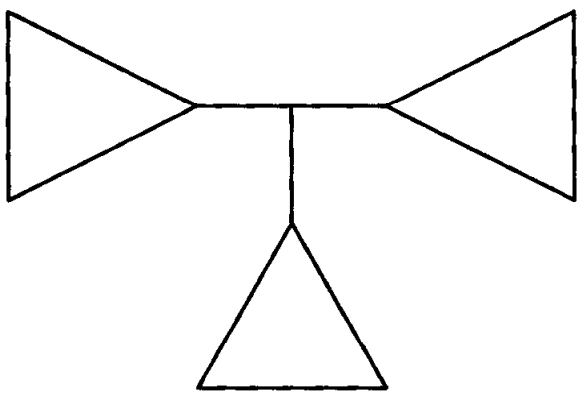


FIGURE 4.9 Can this graph be realized as a shadow boundary?

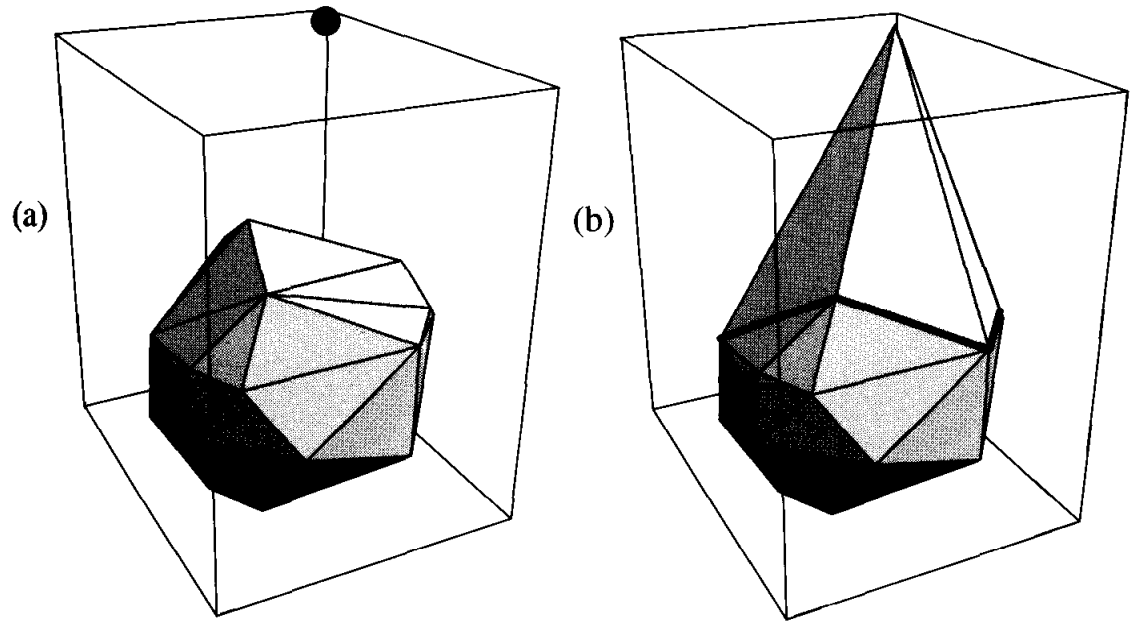


FIGURE 4.10 Viewpoint one: (a) H_{i-1} before adding point in corner; (b) after: H_i.

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)).

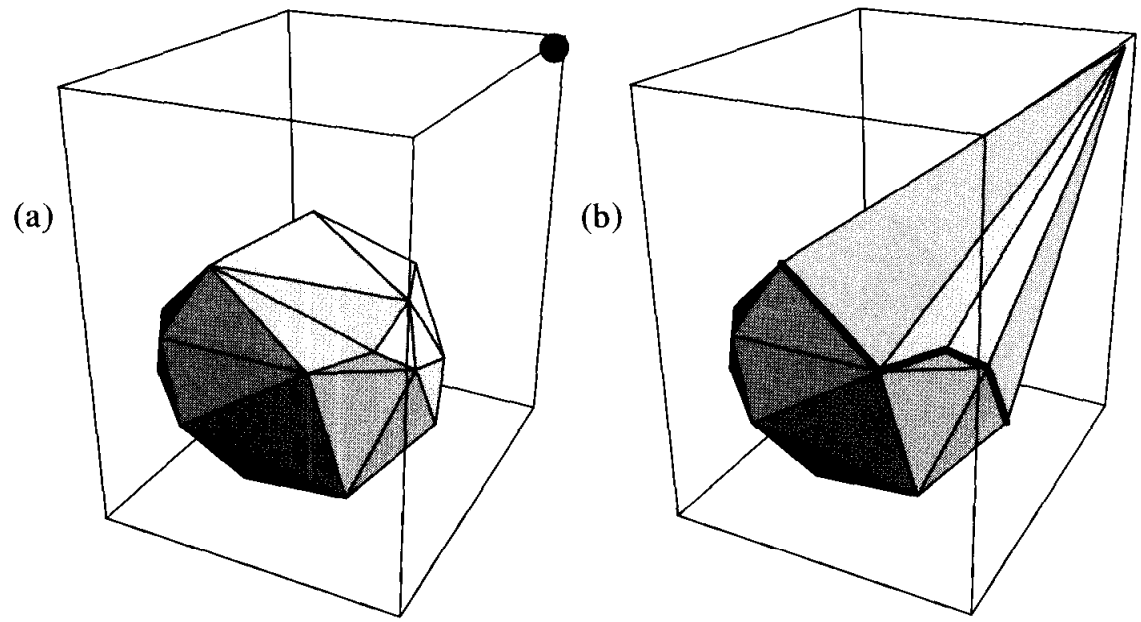


FIGURE 4.11 Viewpoint two: (a) H_{i-1} before adding point in corner; (b) after: H_i.

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: $O(n^2)$.

Algorithm: 3D INCREMENTAL ALGORITHM
Initialize H_3 to tetrahedron (p_0, p_1, p_2, p_3).
    for $i = 4, \ldots, n-1$ 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.

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).

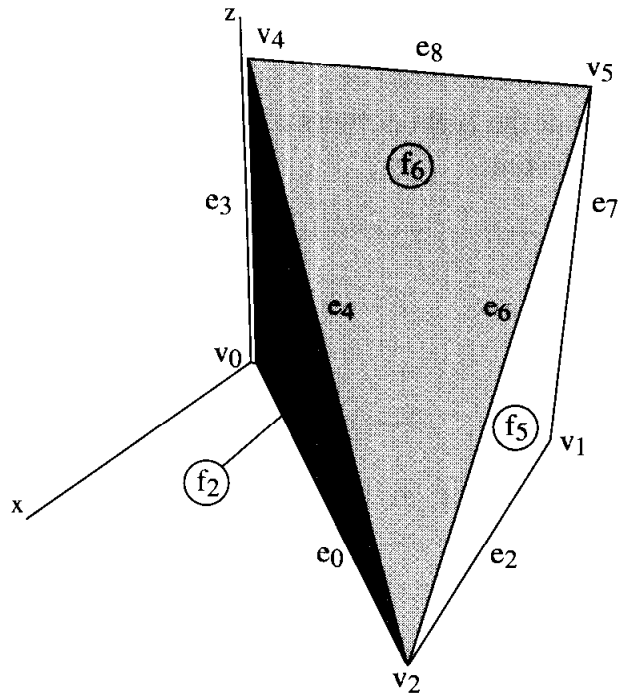


FIGURE 4.12 A view of P_5, with labels.

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.

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.

```
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.

```
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

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."

```
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.

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.

```
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

```
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].

```
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.

`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`.

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.

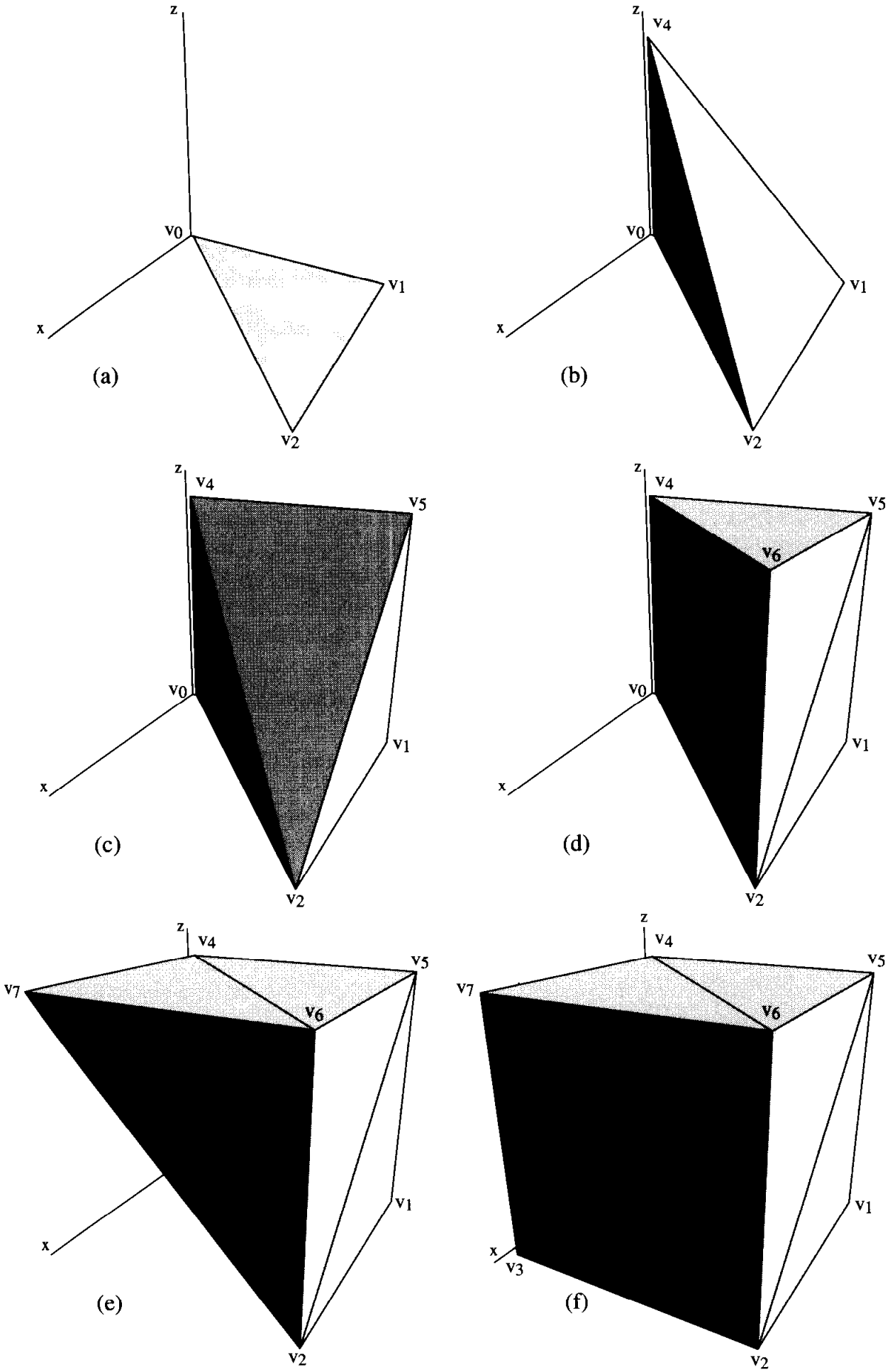


FIGURE 4.13 (a) P_2; (b) P_4; (c) P_5; (d) P_6; (e) P_7; (f) P_3.

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`.

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

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:

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`.

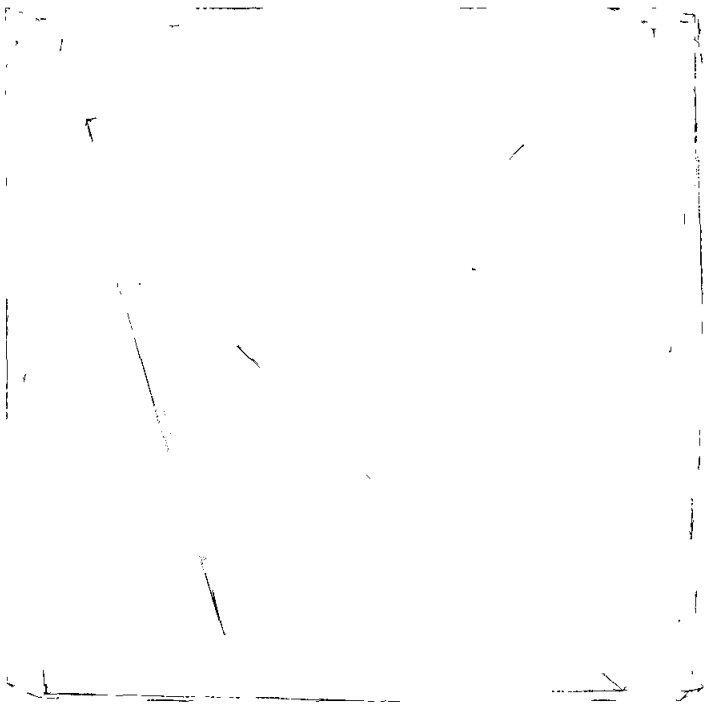


FIGURE 4.14 Hull of 10,000 points in a cube.

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

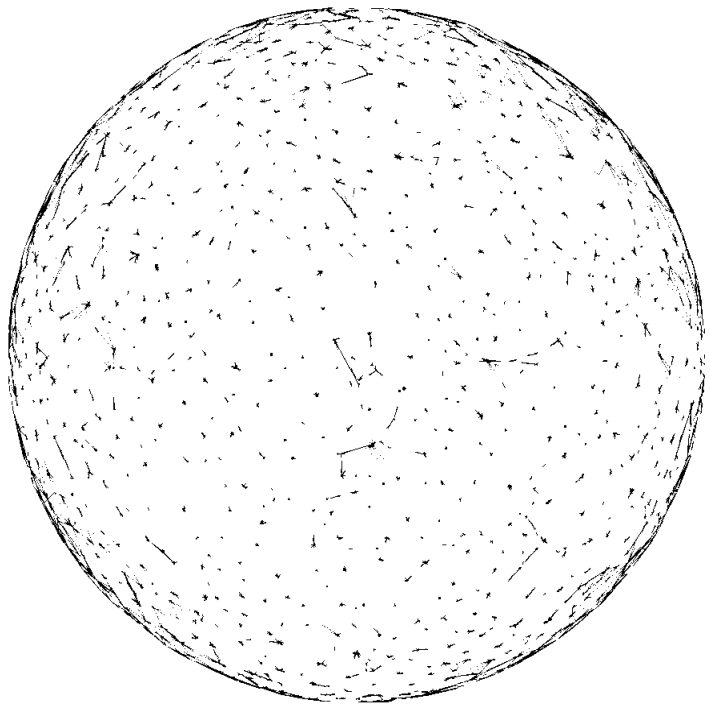


FIGURE 4.15 Hull of 10,000 points near the surface of a sphere.

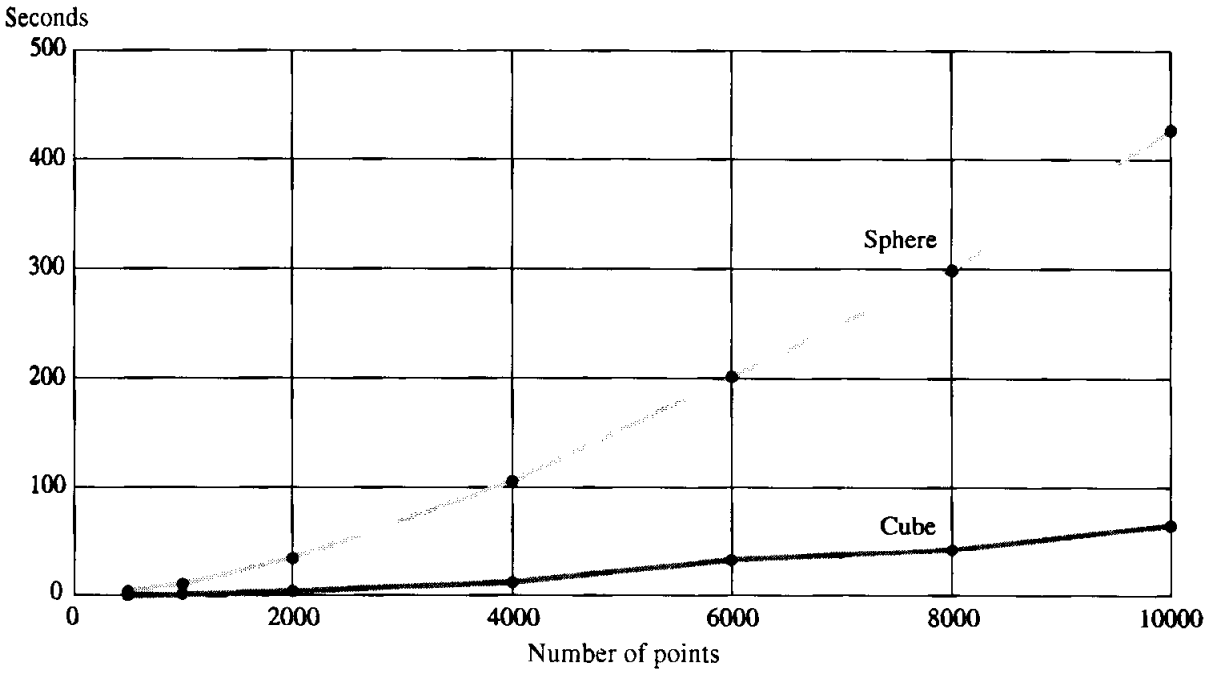


FIGURE 4.16 Runtimes for random points in a cube and near a sphere surface.

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.

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.

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 `double`s 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`

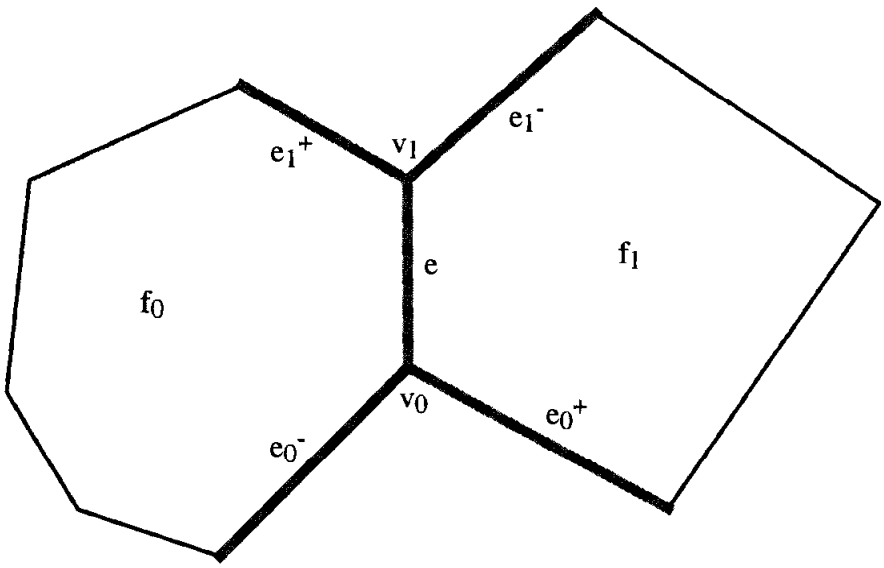


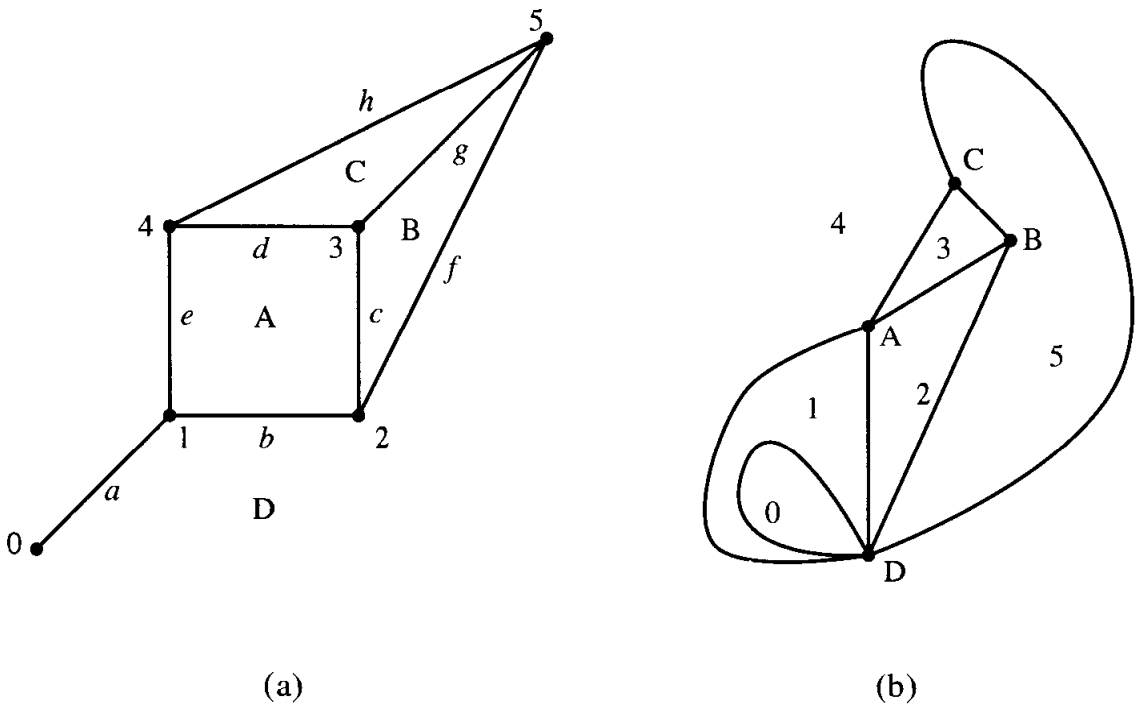
FIGURE 4.17 The winged-edge data structure.

(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.

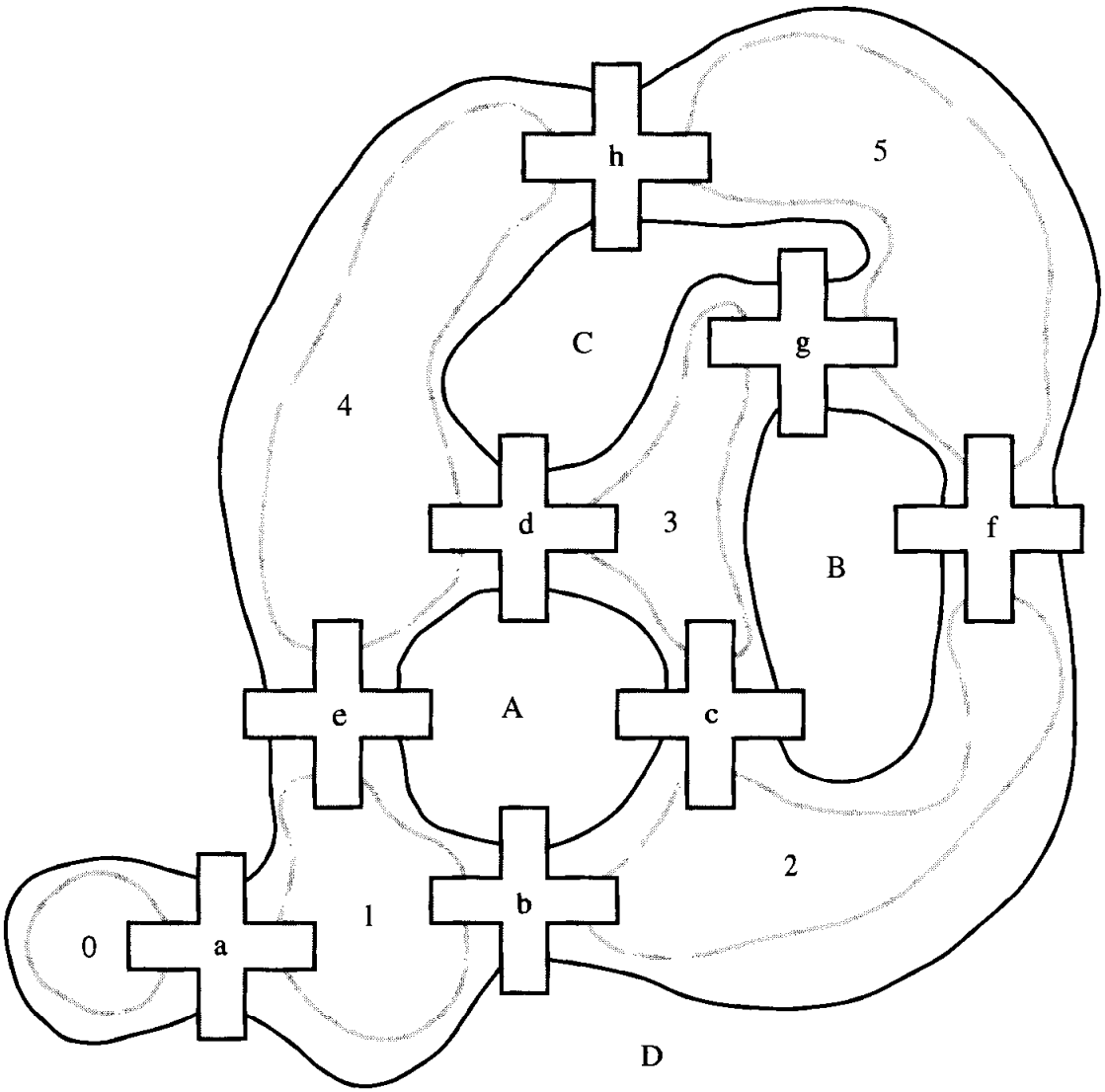


FIGURE 4.19 The quad-edge data structure for the graph in Figure 4.18. Dark cycles represent faces, and light cycles vertices.

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:

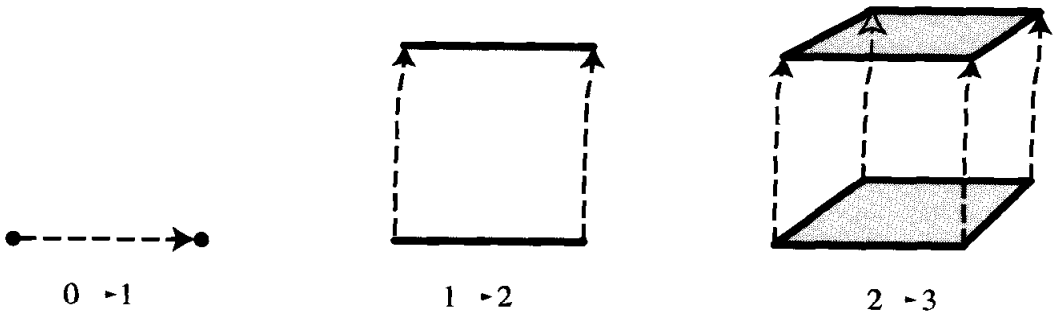


FIGURE 4.20 A cube can be viewed as a square swept through the third dimension.

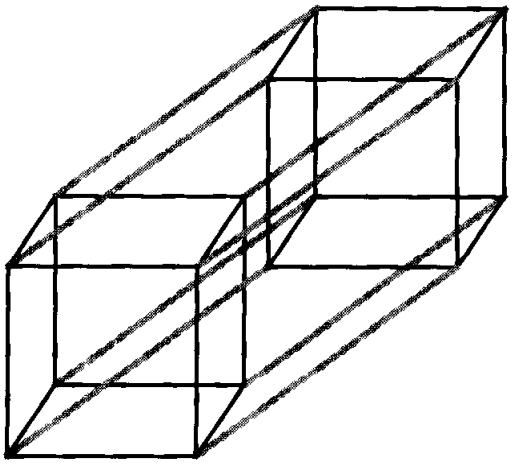


FIGURE 4.21 The edges of a hypercube. The shaded edges represent the sweep in the fourth dimension, connecting two copies of a three-dimensional cube.

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 one-dimensional cube (a point) and stretch it in a second dimension, producing a two-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.

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

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

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.

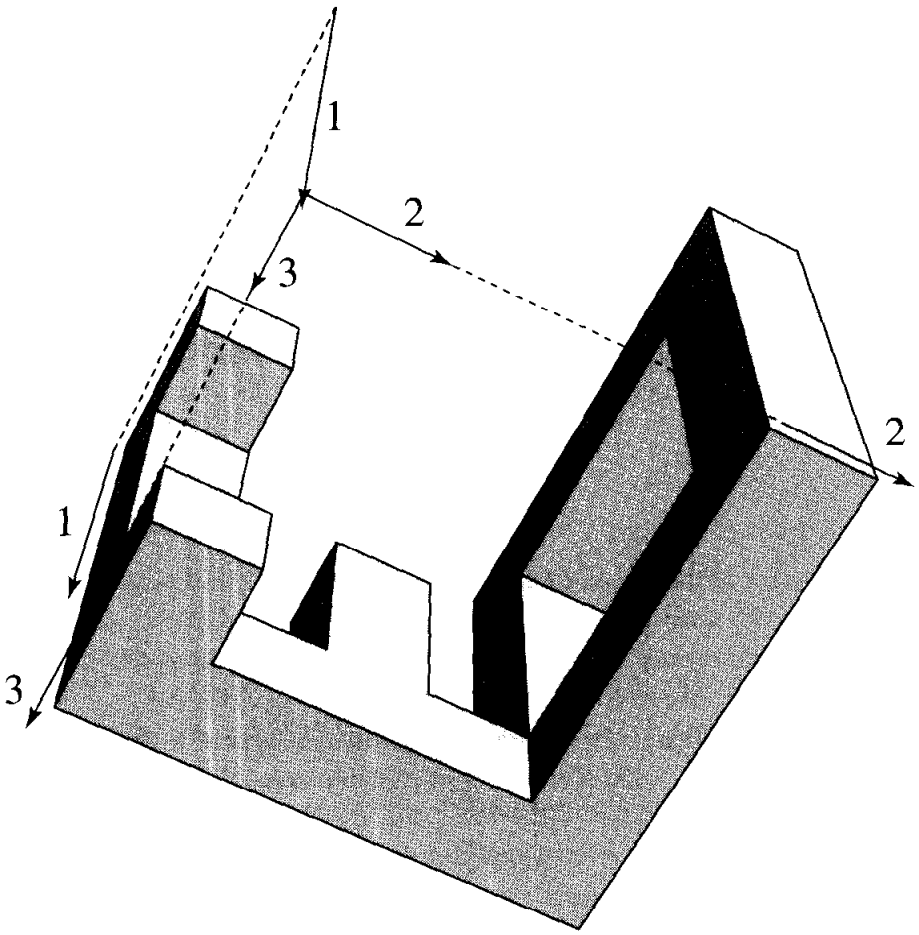


FIGURE 4.22 An orthogonal polyhedron whose shadow in each of the three labeled directions is an orthogonally polygonal letter of the alphabet.

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 ∩ 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.

5

Voronoi Diagrams

In this chapter we study the Voronoi diagram, a geometric structure second in importance only to the convex hull. In a sense a Voronoi diagram records everything one would ever want to know about proximity to a set of points (or more general objects). And often one does want to know detail about proximity: Who is closest to whom? who is furthest? and so on. The concept is more than a century old, discussed in 1850 by Dirichlet and in a 1908 paper of Voronoi.[1]

We will start with a series of examples to motivate the discussion and then plunge into the details of the rich structure of the Voronoi diagram (in Sections 5.2 and 5.3). It is necessary to become intimately familiar with these details before algorithms can be appreciated (in Section 5.4). Finally we will reveal the beautiful connection between Voronoi diagrams and convex hulls in Section 5.7. This chapter includes only two short pieces of code, to construct the dual of the Voronoi diagram (the Delaunay triangulation), in Section 5.7.4.

5.1. APPLICATIONS: PREVIEW

1. *Fire Observation Towers*
   Imagine a vast forest containing a number of fire observation towers. Each ranger is responsible for extinguishing any fire closer to her tower than to any other tower. The set of all trees for which a particular ranger is responsible constitutes the "Voronoi polygon" associated with her tower. The Voronoi diagram maps out the lines between these areas of responsibility: the spots in the forest that are equidistant from two or more towers. (A look ahead to Figure 5.5 may aid intuition.)
2. *Towers on Fire*
   Imagine now the perverse situation where all the rangers ignite their towers simultaneously, and the forest burns at a uniform rate. The fire will spread in circles centered on each tower. The points at which the fire quenches because it reaches previously consumed trees are those points equidistant from two or more towers, which are exactly the points on the Voronoi diagram.
3. *Nearest Neighbor Clustering*
   A technique frequently employed in the field of pattern recognition is to map a set of target objects into a feature space by reducing the objects to points whose coordinates are feature measurements. The example of five tailor's measurements

[1] See Aurenhammer (1991) for a history.

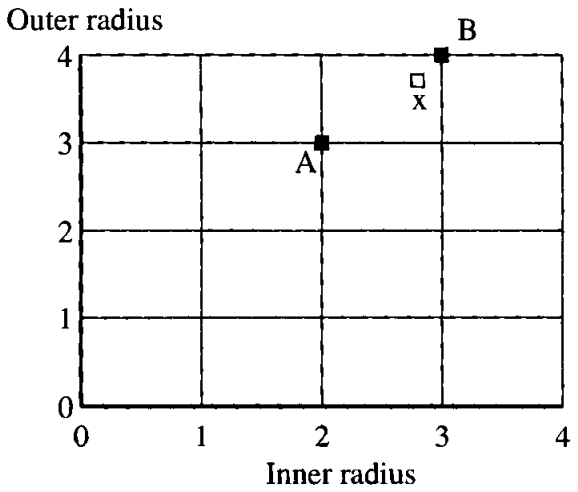


FIGURE 5.1 x is closer to B than to A.

from Section 4.6.1 can be viewed as defining such a feature space. The identity of an object of unknown affiliation then can be assigned the nearest target object in feature space.

An example will make this clearer. Suppose a parts bin includes two types of nuts A and B, A with inner and outer diameters of 2 and 3 centimeters respectively, and B with diameters 3 and 4 cm. Feature space is the positive quadrant of the two-dimensional Euclidean plane, positive because neither radius can be negative. A maps to the point (2, 3), and B to the point (3, 4).

Suppose a vision system focuses on a nut x in the bin and measures its inner and outer radii to be 2.8 and 3.7 cm. Knowing that there are measurement inaccuracies, and that only nuts of type A and B are in the bin, which type of nut is x? It is most likely to be a B nut, because its distance to B in feature space is 0.36, whereas its distance to A is 1.06. See Figure 5.1. In other words, the nearest neighbor of x is B, because x is in B's Voronoi polygon.

If there are many types of nuts, the identification task is to locate the unknown nut x in the Voronoi diagram of the target nuts. How this can be done efficiently will be discussed in Section 5.5.1.

4. *Facility Location*
   Suppose you would like to locate a new grocery store in an area with several existing, competing grocery stores. Assuming uniform population density, where should the new store be located to optimize its sales? One natural method of satisfying this vague constraint is to locate the new store as far away from the old ones as possible. Even this is a bit vague; more precisely we could choose a location whose distance to the *nearest* store is as large as possible. This is equivalent to locating the new store at the center of the largest empty circle, the largest circle whose interior contains no other stores. The distance to the nearest store is then the radius of this circle.

   We will show in Section 5.5.3 that the center of the largest empty circle must lie on the Voronoi diagram.

5. *Path Planning*
   Imagine a cluttered environment through which a robot must plan a path. In order to minimize the risk of collision, the robot would like to stay as far away from all

obstacles as possible. If we restrict the question to two dimensions, and if the robot is circular, then the robot should remain at all times on the Voronoi diagram of the obstacles. If the obstacles are points (say thin poles), then this is the conventional Voronoi diagram. If the obstacles are polygons or other shapes, then a generalized version of the point Voronoi diagram determines the appropriate path.

We will revisit this example in Chapter 8 (Section 8.5.2).

6. *Crystallography*
Assume a number of crystal seeds grow at a uniform, constant rate. What will be the appearance of the crystal when growth is no longer possible? It should be clear now that this is analogous to the forest fire, and that each seed will grow to a Voronoi polygon, with adjacent seed regions meeting along the Voronoi diagram. Voronoi diagrams have long been used to simulate crystal growth.[2]

The list of applications could go on and on, and we will see others in Section 5.5. But it is time to define the diagram formally.

5.2. DEFINITIONS AND BASIC PROPERTIES

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in the two-dimensional Euclidean plane. These are called the *sites*. Partition the plane by assigning every point in the plane to its nearest site. All those points assigned to p_i form the *Voronoi region* $V(p_i)$.[3] $V(p_i)$ consists of all the points at least as close to p_i as to any other site:

$$V(p_i) = \{x : |p_i - x| \leq |p_j - x| \; \forall j \neq i\}. \tag{5.1}$$

Note that we have defined this set to be closed. Some points do not have a unique nearest site, or *nearest neighbor*. The set of all points that have more than one nearest neighbor form the *Voronoi diagram* $\mathcal{V}(P)$ for the set of sites.

Later we will define Voronoi diagrams for sets of objects more general than points. We first look at diagrams with just a few sites before detailing their properties for larger n.

Two Sites

Consider just two sites, p_1 and p_2. Let $B(p_1, p_2) = B_{12}$ be the perpendicular bisector of the segment p_1p_2. Then every point x on B_{12} is equidistant from p_1 and p_2. This can be seen by drawing the triangle (p_1, p_2, x) as shown in Figure 5.2. By the side-angle-side theorem of Euclid,[4] $|p_1x| = |p_2x|$.

Three Sites

For three sites, it is clear that away from the triangle (p_1, p_2, p_3), the diagram contains the bisectors B_{12}, B_{23}, and B_{31}. What is not so clear is what happens in the vicinity of the

[2] See Schaudt & Drysdale (1991) for more information.

[3] This is also called a "Voronoi polygon," "Dirichlet domain," a "Thiessen polygon," or a "Wigner–Seitz region." The Voronoi region is not a polygon by our definition of "polygon," because it might be unbounded.

[4] Euclid (1956, I.4).

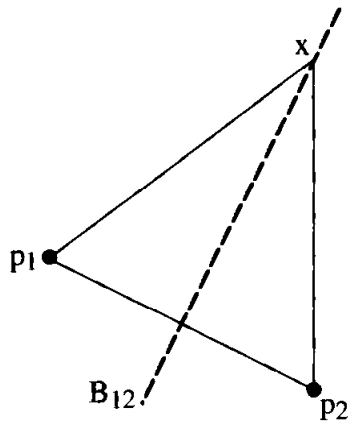


FIGURE 5.2 Two sites: $|p_1x| = |p_2x|$.

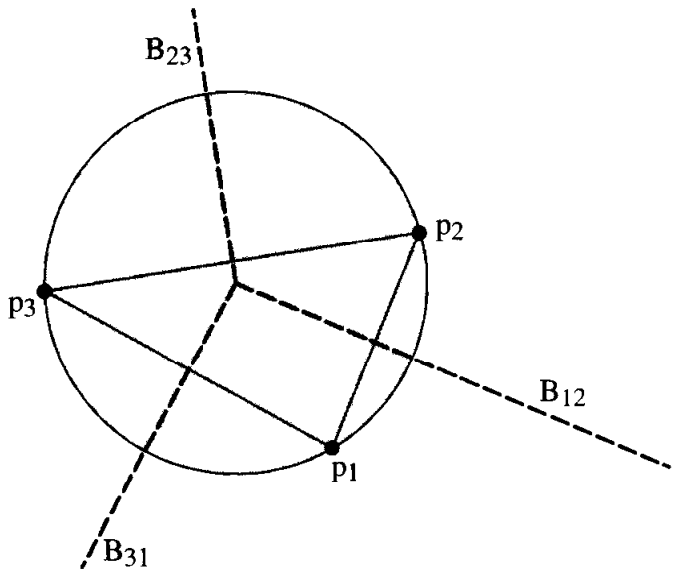


FIGURE 5.3 Three sites: bisectors meet at circumcenter.

triangle. Again from Euclid[5] the perpendicular bisectors of the three sides of a triangle all pass through one point, the circumcenter, the center of the unique circle that passes through the triangle's vertices. Thus the Voronoi diagram for three points must appear as in Figure 5.3. (However, the circumcenter of a triangle is not always inside the triangle as shown.)

5.2.1. Halfplanes

The generalization beyond three points is perhaps not yet clear, but it is certainly clear that the bisectors B_{ij} will play a role. Let $H(p_i, p_j)$ be the closed halfplane with boundary B_{ij} and containing p_i. Then $H(p_i, p_j)$ can be viewed as all the points that are closer to p_i than they are to p_j. Now recall that $V(p_i)$ is the set of all points closer to p_i than to any other site: in other words, the points closer to p_i than to p_1, and closer to p_i than to p_2,

[5]Euclid (1956, IV.5).

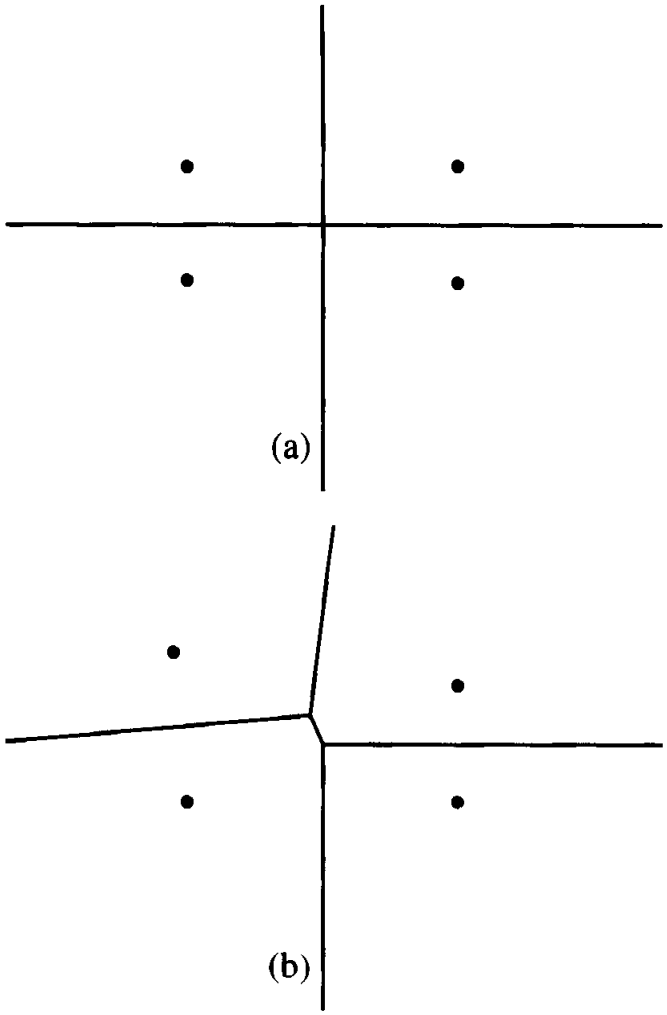


FIGURE 5.4 (a) Voronoi diagram of four cocircular points; (b) the diagram after moving the upper left point.

and closer to p_i than to p_3, and so on. This shows we can write this equation for $V(p_i)$:

$$V(p_i) = \bigcap_{i \neq j} H(p_i, p_j), \tag{5.2}$$

where the notation implies that the intersection is to be taken over all i and j such that $i \neq j$. Note that the English conjunction "and" has been translated to set intersection.

Equation (5.2) immediately gives us an important property of Voronoi diagrams: The Voronoi regions are convex, for the intersection of any number of halfplanes is a convex set. When the regions are bounded, they are convex polygons. The edges of the Voronoi regions are called *Voronoi edges*, and the vertices are called *Voronoi vertices*. Note that a point on the interior of a Voronoi edge has two nearest sites, and a Voronoi vertex has at least three nearest sites.

Four Sites

The diagram of four points forming the corners of a rectangle is shown in Figure 5.4(a).[6] Note that the Voronoi vertex is of degree four. Now suppose one site is moved slightly,

[6]This and several similar figures in this chapter were produced by the XYZ GeoBench software (Schorn 1991).

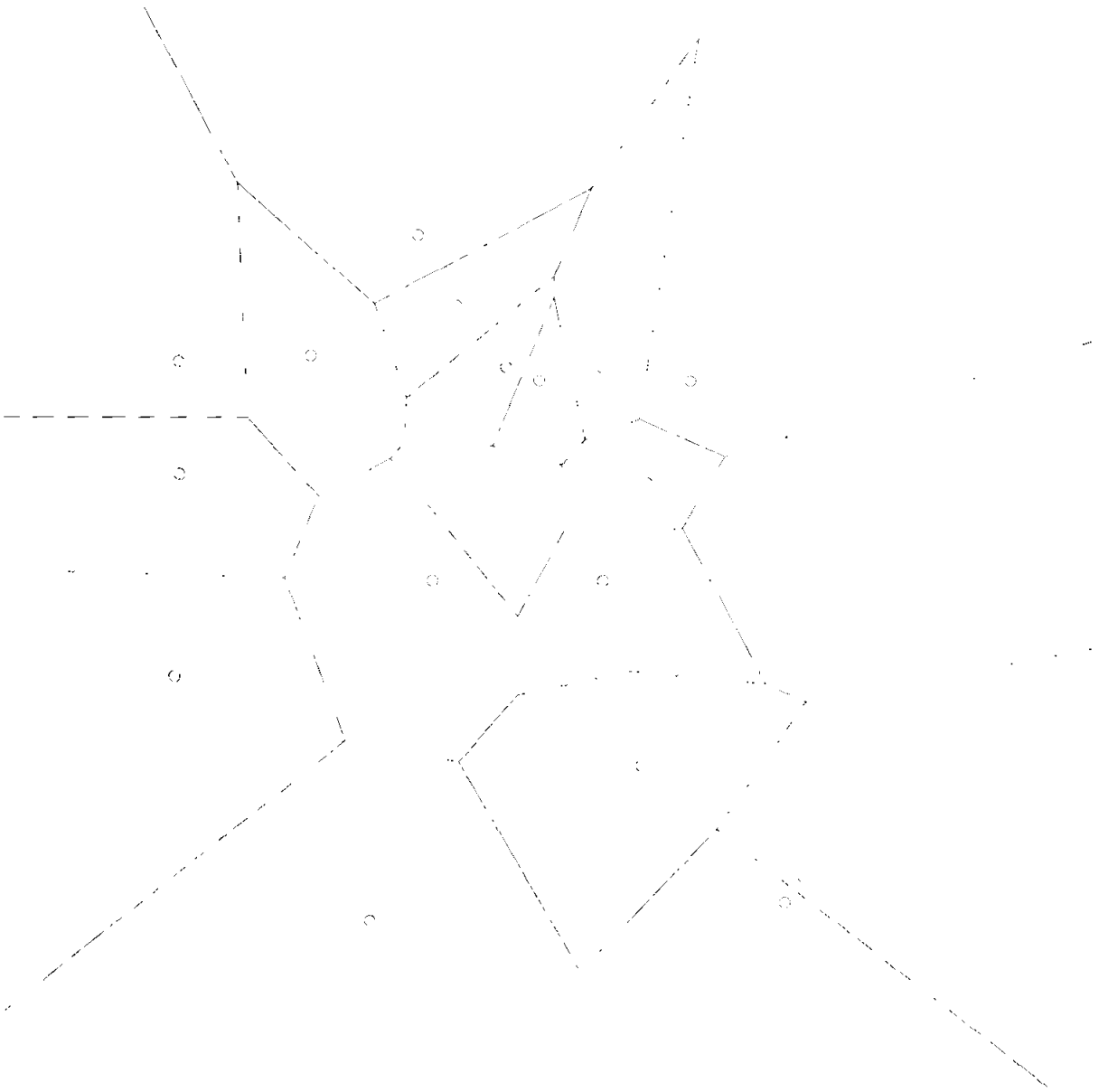


FIGURE 5.5 Voronoi diagram of $n = 20$ sites.

as in Figure 5.4(b). There is a sense in which this diagram is normal, and the one in Figure 5.4(a) is abnormal, or "degenerate." It is degenerate in that there are four cocircular points. We often will find it useful to exclude this type of degeneracy.

Many Sites

A typical diagram with many sites is shown in Figure 5.5. One Voronoi vertex is not shown in this figure: The two nearly horizontal rays leaving the diagram to the left are not quite parallel and intersect at a Voronoi vertex about 70 centimeters left of the figure.

5.2.2. Size of Diagram

Although there are exactly n Voronoi regions for n sites, the total combinatorial size of the diagram conceivably could be quadratic in n, for any particular Voronoi region can

have $\Omega(n)$ Voronoi edges (Exercise 5.3.3[4]). However, we now show that this is in fact not the case, that the total size of the diagram is $O(n)$.

Let us assume for simplicity that no four points are cocircular, and therefore every Voronoi vertex is of degree three. Construct the *dual* graph G (Section 4.4) for a Voronoi diagram $\mathcal{V}(P)$ as follows: The nodes of G are the sites of $\mathcal{V}(P)$, and two nodes are connected by an arc if their corresponding Voronoi polygons share a Voronoi edge (share a positive length edge).

Now observe that this is a planar graph: We can embed each node at its site, and all the arcs incident to the node can be angularly sorted the same as the polygon edges. Moreover, all the faces of G are triangles, corresponding to the degree-three Voronoi vertices. This claim will be made clearer in a moment (Figure 5.6).

But we previously showed that Euler's formula implies that a triangulated planar graph with n vertices has $3n - 6$ edges and $2n - 4$ faces; see Section 4.1.5, Theorem 4.1.1. Because the faces of G correspond to Voronoi vertices, and because the edges of G correspond to Voronoi edges (since each arc of G crosses a Voronoi edge), we have shown that the number of Voronoi vertices, edges, and faces are $O(n)$.

If we now remove the assumption that no four points are cocircular, the graph is still planar, but not necessarily triangulated. For example, the dual of the diagram shown in Fig 5.4(a) is a quadrilateral. But such nontriangulated graphs have fewer edges and faces, so the $O(n)$ bounds continue to hold.

One consequence of the $3n - 6$ edge bound is that the average number of edges of a Voronoi polygon can be no more than six (Exercise 5.3.3[5]).

5.3. DELAUNAY TRIANGULATIONS

In 1934 Delaunay proved that when the dual graph is drawn with straight lines, it produces a planar triangulation of the Voronoi sites P (if no four sites are cocircular), now called the *Delaunay triangulation* $\mathcal{D}(P)$. Figure 5.6 shows the Delaunay triangulation for the Voronoi diagram in Figure 5.5, and Figure 5.7 shows the Delaunay triangulation superimposed on the corresponding Voronoi diagram. Note that it is not immediately obvious that using straight lines in the dual would avoid crossings in the dual; the dual segment between two sites does not necessarily cross the Voronoi edge shared between their Voronoi regions, as is evident in Figure 5.7. We will not prove Delaunay's theorem now, but rather we will wait until we have gathered more properties of Voronoi diagrams and Delaunay triangulations, when the proof will be easy.

5.3.1. Properties of Delaunay Triangulations

Because the Delaunay triangulation and Voronoi diagram are dual structures, each contains the same "information" in some sense, but represented in a rather different form. To gain a grasp on these complex structures, it is important to have a thorough understanding of the relationships between a Delaunay triangulation and its corresponding Voronoi diagram. We list without proof several Delaunay properties and follow with a

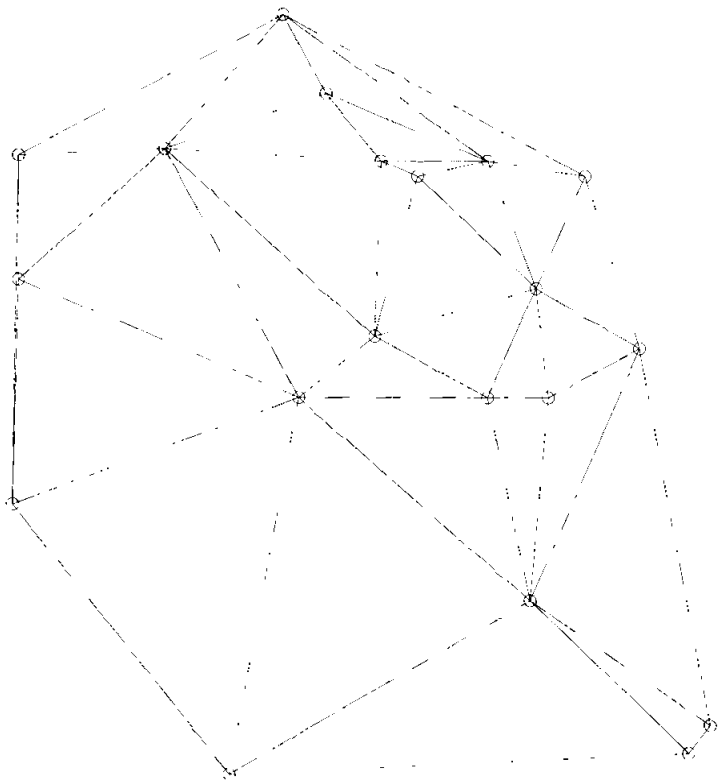


FIGURE 5.6 Delaunay triangulation for the sites in Figure 5.5.

more substantive list of Voronoi properties.[7] Only the properties **D6** and **D7** have not been mentioned before. Fix a set of sites P.

D1. $\mathcal{D}(P)$ is the straight-line dual of $\mathcal{V}(P)$. This is by definition.
D2. $\mathcal{D}(P)$ is a triangulation if no four points of P are cocircular: Every face is a triangle. This is Delaunay's theorem. The faces of $\mathcal{D}(P)$ are called *Delaunay triangles*.
D3. Each face (triangle) of $\mathcal{D}(P)$ corresponds to a vertex of $\mathcal{V}(P)$.
D4. Each edge of $\mathcal{D}(P)$ corresponds to an edge of $\mathcal{V}(P)$.
D5. Each node of $\mathcal{D}(P)$ corresponds to a region of $\mathcal{V}(P)$.
D6. The boundary of $\mathcal{D}(P)$ is the convex hull of the sites.
D7. The interior of each (triangle) face of $\mathcal{D}(P)$ contains no sites. (Compare **V5**.)

Properties **D6** and **D7** here are the most interesting; they can be verified in Figures 5.6 and 5.7.

5.3.2. Properties of Voronoi Diagrams

V1. Each Voronoi region $V(p_i)$ is convex.
V2. $V(p_i)$ is unbounded iff p_i is on the convex hull of the point set. (Compare **D6**.)

[7]Here I am following the pedagogic lead of Preparata & Shamos (1985, Section 5.5.1). Also, some notation is borrowed from Okabe, Boots & Sugihara (1992).

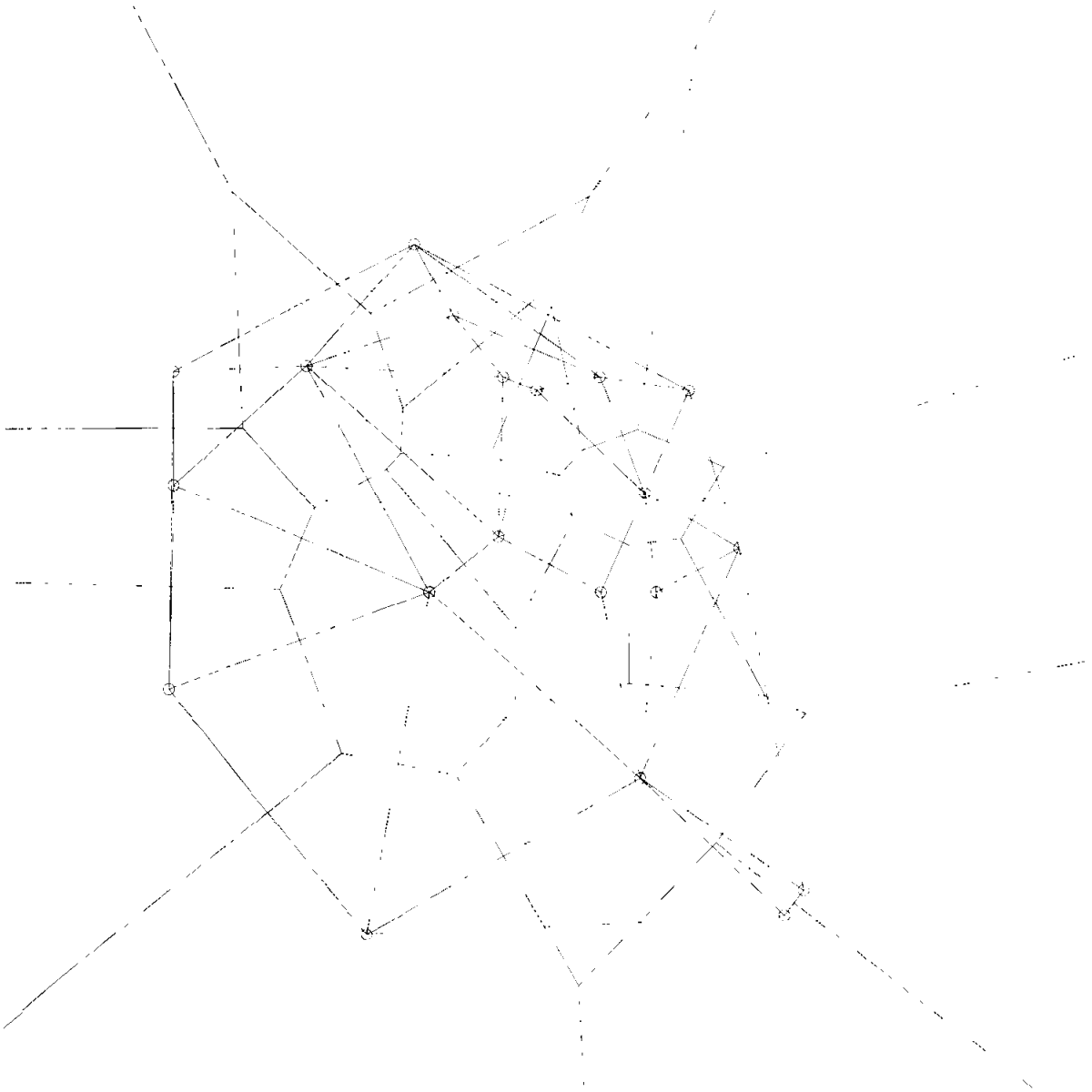


FIGURE 5.7 Delaunay triangulation and Voronoi diagram: Figures 5.5 and 5.6 together.

V3. If v is a Voronoi vertex at the junction of $V(p_1)$, $V(p_2)$, and $V(p_3)$, then v is the center of the circle $C(v)$ determined by p_1, p_2, and p_3. (This claim generalizes to Voronoi vertices of any degree.)

V4. $C(v)$ is the circumcircle for the Delaunay triangle corresponding to v.

V5. The interior of $C(v)$ contains no sites. (Compare **D7**.)

V6. If p_j is a nearest neighbor to p_i, then (p_i, p_j) is an edge of $\mathcal{D}(P)$.

V7. If there is some circle through p_i and p_j that contains no other sites, then (p_i, p_j) is an edge of $\mathcal{D}(P)$. The reverse also holds: For every Delaunay edge, there is some empty circle.

Property **V7**, the least intuitive, is an important characterization of Delaunay edges and will be used in several proofs later on. This is the only property we will prove formally.

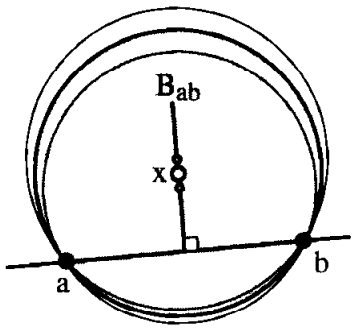


FIGURE 5.8 $C(x)$ is the shaded circle. Its center x can move along B_{ab} while remaining empty and still through a and b.

Theorem 5.3.1. *$ab \in \mathcal{D}(P)$ iff there is an empty circle through a and b: The closed disk bounded by the circle contains no sites of P other than a and b.*

Proof. One direction is easy: If ab is a Delaunay edge, then $V(a)$ and $V(b)$ share a positive-length edge $e \in \mathcal{V}(P)$. Put a circle $C(x)$ with center x on the interior of e, with radius equal to the distance to a or b. This circle is obviously empty of other sites, for if it were not, if, say, site c were on or in the circle, x would be in $V(c)$ as well, but we know that x is only in $V(a)$ and $V(b)$.

The reverse implication is more subtle. Suppose there is an empty circle $C(x)$ through a and b, with center x. We aim to prove that $ab \in \mathcal{D}(P)$. Because x is equidistant from a and b, x is in the Voronoi regions of both a and b as long as no other point interferes with "nearest-neighborliness." But none does, because the circle is empty. Therefore, $x \in V(a) \cap V(b)$ (recall we defined Voronoi regions to be closed sets). Because no points are on the boundary of $C(x)$ other than a and b (by hypothesis), there must be freedom to wiggle x a bit and maintain emptiness of $C(x)$. In particular, we can move x along B_{ab}, the bisector between a and b, and maintain emptiness while keeping the circle through a and b. See Figure 5.8. Therefore x is on a positive-length Voronoi edge (a subset of B_{ab}) shared between $V(a)$ and $V(b)$. And therefore $ab \in \mathcal{D}(P)$. □

We leave the proof of the other properties to intuition, exercises, and to Section 5.7.2.

5.3.3. Exercises

1. *Regular polygon* [easy]. Describe the Voronoi diagram and Delaunay triangulation for the vertices of a regular polygon.
2. *Unbounded regions.* Prove property **V2**: $V(p_i)$ is unbounded iff p_i is on the convex hull of the point set. Do not assume the corresponding Delaunay property **D6**, but otherwise any Delaunay or Voronoi property may be employed in the proof.
3. *Nearest neighbors.* Prove property **V6**: If p_j is a nearest neighbor to p_i, then (p_i, p_j) is an edge of $\mathcal{D}(P)$. Any Delaunay or Voronoi property may be employed in the proof.
4. *High-degree Delaunay vertex.* Design a set of n points, with n arbitrary, and with no four cocircular, such that one vertex of the Delaunay triangulation has degree $n - 1$.
5. *Average number of Voronoi polygon edges.* Prove that the number of edges in a Voronoi polygon, averaged over all Voronoi regions for any set of n points, does not exceed 6 (Preparata & Shamos 1985, p. 211).

6. *Pitteway triangulations.* A triangulation of a set of points P is called a *Pitteway triangulation* (Okabe et al. 1992, p. 90) if, for each triangle $T = (a, b, c)$, every point in T has one of a, b, or c as its nearest neighbor among the points of P.
   a. Show by example that not every Delaunay triangulation is a Pitteway triangulation.
   b. Characterize those Delaunay triangulations that are Pitteway triangulations.

5.4. ALGORITHMS

The many applications of the Voronoi diagram and its inherent beauty have spurred researchers to invent a variety of algorithms to compute it. In this section we will examine four algorithms, each rather superficially, for we will see in Section 5.7.2 that the Voronoi diagram can be computed using our convex hull code.

5.4.1. Intersection of Halfplanes

We could construct each Voronoi region separately, by intersecting $n - 1$ halfplanes according to Equation (5.2). Constructing the intersection of n halfplanes is dual to the task of constructing the convex hull of n points in two dimensions and can be accomplished with similar algorithms in $O(n \log n)$ time (Exercise 6.5.3[5]). Doing this for each site would cost $O(n^2 \log n)$.

5.4.2. Incremental Construction

Suppose the Voronoi diagram $\mathcal{V}$ for k points is already constructed, and now we would like to construct the diagram $\mathcal{V}'$ after adding one more point p. Suppose p falls inside the circles associated with several Voronoi vertices, say $C(v_1), \ldots, C(v_m)$. Then these vertices of $\mathcal{V}$ cannot be vertices of $\mathcal{V}'$, for they violate the condition that Voronoi vertex circles must be empty of sites (**V5**, Section 5.3.2). It turns out that these are the only vertices of $\mathcal{V}$ that are not carried over to $\mathcal{V}'$. It also turns out that these vertices are all localized to one area of the diagram. These vague observations can be made precise, and they form one of the cleanest algorithms for constructing the Voronoi diagram (Green & Sibson 1977). The algorithm spends $O(n)$ time per point insertion, for a total complexity of $O(n^2)$. Despite this quadratic complexity, this has been the most popular method of constructing the diagram; see Field (1986) for implementation details.

The incremental algorithm has been revitalized recently with randomization, which we will touch upon in Section 5.7.4.

5.4.3. Divide and Conquer

The Voronoi diagram can be constructed with a complex divide-and-conquer algorithm in $O(n \log n)$ time, first detailed by Shamos & Hoey (1975). It was this paper that introduced the Voronoi diagram to the computer science community. This time complexity is asymptotically optimal, but the algorithm is rather difficult to implement. However, it can be done with careful attention to data structures; see (Guibas & Stolfi 1985).

We will pass over this historically important algorithm in order to focus on some exciting recent developments.

5.4.4. Fortune's Algorithm

Until the mid-1980s, most implementations for computing the Voronoi diagram used the $O(n^2)$ incremental algorithm, accepting its slower performance to avoid the complexities of the divide-and-conquer coding. But in 1985, Fortune (1987) invented a clever plane-sweep algorithm that is as simple as the incremental algorithms but has worst-case complexity of $O(n \log n)$. We will now sketch the main idea behind this algorithm.

Plane-sweep algorithms (Section 2.2.4) pass a sweep line over the plane, leaving at any time the problem solved for the portion of the plane already swept and unsolved for the portion not yet reached. A plane-sweep algorithm for constructing the Voronoi diagram would have the diagram constructed behind the line. At first blush, this seems quite impossible, as Voronoi edges of a Voronoi region $V(p)$ would be encountered by the sweep line L *before* L encounters the site p responsible for the region. Fortune surmounted this seeming impossibility by an extraordinarily clever idea.[8]

Cones

Imagine the sites in the xy-plane of a three-dimensional coordinate system. Erect over each site p a cone whose apex is at p, and whose sides slope at 45°. If the third dimension is viewed as time, then the cone over p represents a circle expanding about p at unit velocity: After t units of time, its radius is t.

Now consider two nearby cones, over sites p_1 and p_2. They intersect in a curve in space. Recalling the expanding circles view of the Voronoi diagram, it should come as no surprise that this curve lies entirely in a vertical plane,[9] the plane orthogonal to the bisector of p_1p_2. See Figure 5.9. Thus although the intersection is curved in three dimensions, it projects to a straight line on the xy-plane.

It is but a small step from here to the claim that if the cones over all sites are opaque, and they are viewed from $z = -\infty$, what is seen is precisely the Voronoi diagram!

Cone Slicing

We are now prepared to describe Fortune's idea. His algorithm sweeps the cones with a slanted plane π, slanted at 45° to the xy-plane. The sweep line L is the intersection of π with the xy-plane. Let us assume that L is parallel to the y axis and that its x coordinate is ℓ. See Figure 5.10. Imagine that π, as well as all the cones, are opaque, and again consider the view from $z = -\infty$.

To the $x > \ell$ side of L, only π is visible from below: It cuts below the xy-plane and so obscures the sites and cones. This represents the portion of the plane yet to be swept. To the $x < \ell$ side of L, the Voronoi diagram is visible up to the intersection of π with the right (positive x) "frontier" of cones. The intersection of π with any one cone is a parabola (a basic property of conic sections), and so the intersection of π with this right frontier projects to the xy-plane (and so appears from $z = -\infty$) as a "parabolic front," a curve composed of pieces of parabolas. See Figure 5.11. Two parabolas join at a spot

[8] My exposition relies heavily on that of Guibas & Stolfi (1988), rather than on Fortune's original paper, which explained the algorithm in a rather different manner.

[9] The curve is a branch of a hyperbola, the conic section formed by intersection with a plane parallel to the axis of the cone.

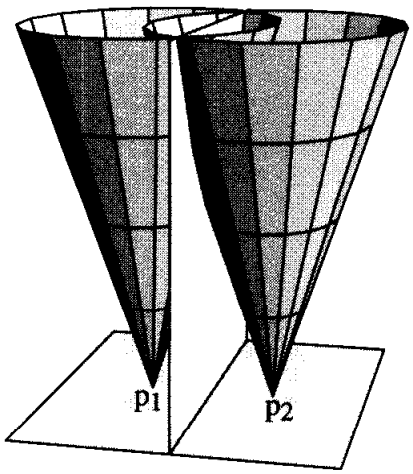


FIGURE 5.9 The curve of intersection of two cones projects to a line.

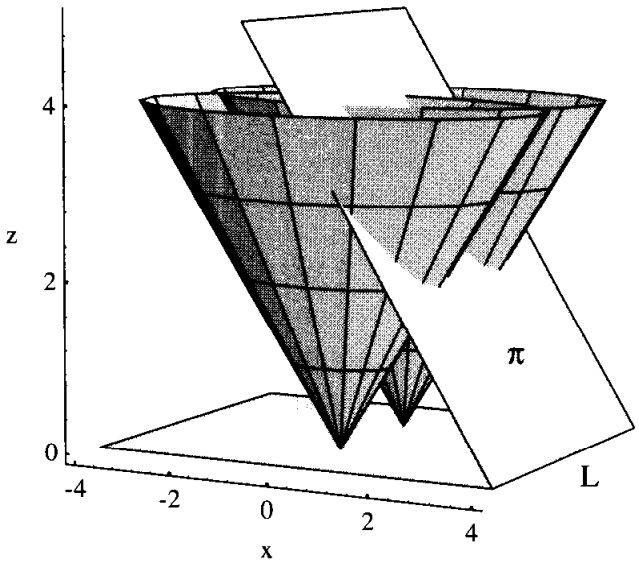


FIGURE 5.10 Cones cut by sweep plane. π and L are sweeping toward the right, $x \to \infty$.

where π meets two cones. From our discussion of the intersection of two cones above, this must be at a Voronoi edge.

Parabolic Front

Now we finally can see how Fortune solved the problem of the sweep line encountering Voronoi edges prior to the generating sites: Because his sweep plane π slopes at the same angle as the cone sides, L encounters a site p exactly when π first hits the cone for p! Hence it is not the case that the Voronoi diagram is at all times constructed to the left of L, but it is at all times constructed *underneath* π, which means that it is constructed to the left of L up to the parabolic front, which lags L a bit.

What is maintained at all times by the algorithm is the parabolic front, whose joints trace out the Voronoi diagram over time, since these kinks all lie on Voronoi edges. Although we are by no means finished with the algorithm description, we will make no attempt to detail it further here.

Finally, it should be clear that the algorithm only need store the parabolic front, which is of size $O(n)$ and is often $O(\sqrt{n})$. This is a significant advantage of Fortune's algorithm

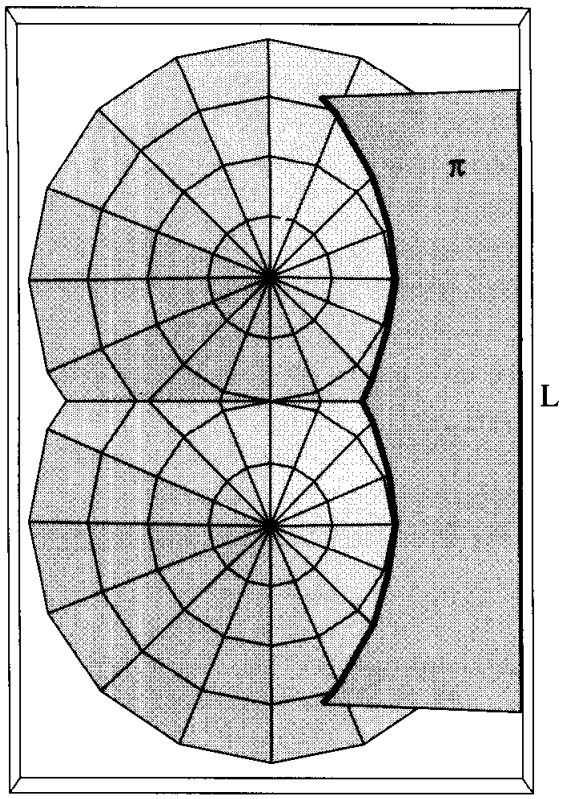


FIGURE 5.11 Figure 5.10 viewed from $z \approx -\infty$. The heavy curve is the parabolic front.

when n is large: The storage needed at any one time is often much smaller than the size of the diagram. And n is often large, perhaps 10^6 (Sugihara & Iri 1992), for diagrams based on data gathered by, for example, geographic information systems.

5.4.5. Exercises

1. $\mathcal{D}(P) \Rightarrow \mathcal{V}(P)$. Design an algorithm for computing the Voronoi diagram, given the Delaunay triangulation. Try to achieve $O(n)$ time complexity.
2. *One-dimensional Voronoi diagrams.* A one-dimensional Voronoi diagram for a set of points $P = \{p_1, \ldots, p_n\}$ on a line (say the x axis) is a set of points $\mathcal{V}(P) = \{x_1, \ldots, x_{n-1}\}$ such that x_i is the midpoint of $p_i p_{i+1}$.

   Suppose you are given a set $X = \{x_1, \ldots, x_{n-1}\}$. Design criteria that will permit you to determine whether or not X is a one-dimensional Voronoi diagram of a set of points, and if so, determine P. How fast is the implied algorithm?
3. *Dynamic Voronoi diagrams.* Imagine a set of points moving on the plane, each with a fixed velocity and direction. Let $V(t)$ be the Voronoi diagram of the points at time t. It is an unsolved problem to obtain tight bounds on the number of combinatorially distinct diagrams that can result over all time. Here I ask you to establish the best-known lower bound: $\Omega(n^2)$. In other words, find a set of n moving points such that $V(t)$ changes its combinatorial structure cn^2 times for some constant c.

   No one has been able to find an example in which there are more than n^2 changes, but the best upper bound is about $O(n^3)$ (Fu & Lee 1991), Guibas, Mitchell & Roos (1991).

4. *Arbitrary triangulation.* Design an algorithm to find an arbitrary triangulation of a point set P: a collection of diagonals incident to every point of P that partitions $\mathcal{H}(P)$ into triangles. The absence of the requirement that the triangulation be Delaunay permits considerable freedom in the design.
5. *Flipping algorithm.* Investigate the following proposed algorithm for constructing $\mathcal{D}(P)$: Start with an arbitrary triangulation of P. Then repeat the following procedure until $\mathcal{D}(P)$ is attained. Identify two adjacent triangles abc and cbd sharing diagonal bc, such that the quadrilateral $abcd$ is convex. If d is inside the circumcircle of abc, then delete cb and add ad. Will this work?

5.5. APPLICATIONS IN DETAIL

We will now discuss five applications of the Voronoi diagram, in uneven detail: nearest neighbors, "fat" triangulations, largest empty circles, minimum spanning trees, and traveling salesperson paths.

5.5.1. Nearest Neighbors

An application of the Voronoi diagram for nearest-neighbor clustering was mentioned in Section 5.1. That problem can be viewed as a *query* problem: Which is the nearest neighbor to a query point? Another version is the *all nearest neighbors* problem: Find the nearest neighbor to each point in a given set. This has a number of applications in a variety of fields, including biology, ecology, geography, and physics.[10]

Define the nearest neighbor relation among a set of points P as follows: b is a *nearest neighbor* of a iff $|a - b| \leq \min_{c \neq a} |a - c|$, where $c \in P$. We can write this relation $a \rightarrow b$: A nearest neighbor of a is b. Note that the definition is not symmetric with respect to the roles that a and b play, suggesting that the relation is not itself symmetric. And in fact this is indeed the case: If $a \rightarrow b$, it is not necessary that $b \rightarrow a$; see Figure 5.12. Also note that a point can have several equally nearest neighbors (e.g., point d in the figure).

Nearest Neighbor Queries

Given a fixed set of points P, construct the Voronoi diagram in $O(n \log n)$ time. Now for a query point q, finding a nearest neighbor of q reduces to finding in which Voronoi region(s) it falls, for the sites of those Voronoi regions are precisely its nearest neighbors. The problem of locating a point inside a partition is called *point location*. The problem has been studied heavily and will be discussed in Chapter 7 (Section 7.11). We will see there that in this instance, $O(\log n)$ time suffices for each query.

All Nearest Neighbors

Define the *Nearest Neighbor Graph (NNG)* to have a node associated with each point of P and an arc between them if one point is a nearest neighbor of the other. We have defined this to be an undirected graph, although because the relation is not symmetric,

[10]Citations in Preparata & Shamos (1985, p. 186) and Okabe et al. (1992, p. 422).

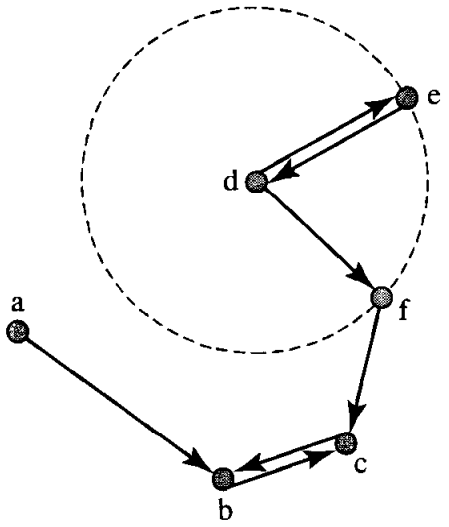


FIGURE 5.12 $a \to b$, but $b \to c$; also $d \to e$ and $d \to f$.

it could well be defined to be directed. But we will not need the directed version here.

A succinct way to capture the essence of efficient nearest neighbor algorithms is through the following lemma.

Lemma 5.5.1. $NNG \subseteq \mathcal{D}(P)$.

I leave the proof to Exercises 5.5.6[2] and [3].

A brute-force algorithm for finding the nearest neighbors for each point in a set would require $O(n^2)$ time, but the above lemma lets us search only the $O(n)$ edges of the Delaunay triangulation and therefore achieve $O(n \log n)$.

5.5.2. Triangulation Maximizing the Minimum Angle

Analyzing the structural properties of complex shapes is often accomplished by a technique called "finite element analysis." This is used, for example, by automobile manufacturers to model car bodies (Field 1986). The domain to be studied is partitioned into a *mesh* of "finite elements," and then the relevant differential equations modeling the structural dynamics are solved by discretizing over the partition. The stability of the numerical procedures used depends on the quality of the partition, and it so happens that Delaunay triangulations are especially good partitions. We will now discuss the sense in which Delaunay triangulations are good.

A *triangulation* of a point set S is the generalization of the object of which the Delaunay triangulation is a particular instance: a set of segments whose endpoints are in S, which only intersect each other at endpoints, and which partition the convex hull of S into triangles. For the purposes of finite element analysis, triangulations with "fat" triangles are best. One way to make this more precise is to avoid triangles with small angles. Thus it is natural to seek a triangulation that has the largest smallest angle, that is, to maximize the smallest angle over all triangulations. This happens to be precisely the Delaunay triangulation! In fact, a somewhat stronger statement can be made, which we now describe after introducing some notation.

Let T be a triangulation of a point set S, and let its *angle sequence* $(\alpha_1, \alpha_2, \ldots, \alpha_{3t})$, be a list of the angles of the triangles, sorted from smallest to largest, with t the number of triangles in T. The number t is a constant for each S (Exercise 5.5.6[4]). We can define a relation between two triangulations of the same point set, T and T', that attempts to capture the "fatness" of the triangles. Say that $T \geq T'$ (T is fatter than T') if the angle sequence of T is lexicographically greater than the angle sequence of T': either $\alpha_1 > \alpha_1'$, or $\alpha_1 = \alpha_1'$ and $\alpha_2 > \alpha_2'$, or $\alpha_1 = \alpha_1'$ and $\alpha_2 = \alpha_2'$ and $\alpha_3 > \alpha_3'$, and so on.

Edelsbrunner (1987, p. 302) proved this pleasing theorem:

Theorem 5.5.2. *The Delaunay triangulation $T = \mathcal{D}(P)$ is maximal with respect to the angle-fatness relation: $T \geq T'$ for any other triangulation T' of P.*

In particular this says that the Delaunay triangulation maximizes the smallest angle.

5.5.3. Largest Empty Circle

We mentioned in Section 5.1 the problem of finding the largest empty circle among a set S of sites: The center of such a circle is a good location for a new store. Another application is mentioned by Toussaint (1983a): Locate a nuclear reactor as far away from a collection of city-sites as possible. We now examine the largest empty circle problem in some detail.

The problem makes little sense unless some restriction is placed on the location of the circle center, for there are always arbitrarily large empty circles outside any finite set of points. So we phrase the problem this way:

Largest Empty Circle. *Find a largest empty circle whose center is in the (closed) convex hull of a set of n sites S, empty in that it contains no sites in its interior, and largest in that there is no other such circle with strictly larger radius.*

Let $f(p)$ be the radius of the largest empty circle centered on point p. Then we are looking for a maximum of this function over all p in the hull of S, $H = \mathcal{H}(S)$. But there are a seemingly infinite number of candidate points for these maxima. A common theme in computational geometry is to reduce an infinite candidate set to a small finite list, and then to find these efficiently. We follow this scenario in this section, starting by arguing informally that only certain points p are true candidates for maxima of f.

Centers Inside the Hull

Imagine inflating a circle from a point p in H. The radius at which this circle first bumps into and therefore includes some site of $S = \{s_1, \ldots, s_n\}$ is the value of $f(p)$. Let us temporarily assume throughout this subsection that p is strictly interior to H. If at radius $f(p)$, the circle includes just one site s_1, then it should be clear that $f(p)$ cannot be a maximum of the radius function. For if p is moved to p' along the ray $s_1 p$ (the ray from s_1 through p) away from s_1, then $f(p')$ is larger, as shown in Figure 5.13 (upper circles). Therefore p could not have been a local maximum of f, for there is a point p' in any neighborhood of p where f is larger. Note that the assumption that p is strictly interior to the hull guarantees that there is a p' as above that is also in H.

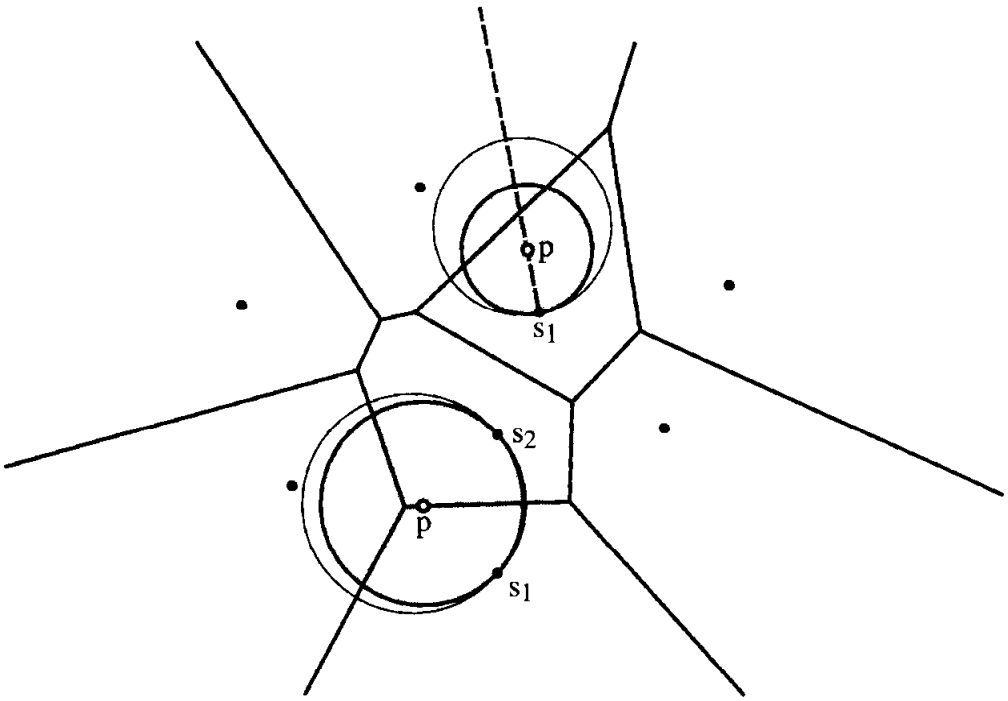


FIGURE 5.13 Center in interior, circle through one (upper) or two (lower) sites.

Now let us assume that at radius $f(p)$, the circle includes exactly two sites s_1 and s_2. Again $f(p)$ cannot be at a maximum: If p is moved to p' along the bisector of s_1s_2 (away from s_1s_2), then $f(p')$ is again larger, as shown in Figure 5.13 (lower circles). Another way to see this is via the intersection of site-centered cones, discussed in Section 5.4.4. The curve of intersection of two such cones (Figure 5.9) represents the distance from the sites for points on the bisector. Since the curve is an upward hyperbola branch, no interior point of the bisector is a local maximum: The distance increases in one direction or the other.

It is only when the circle includes three sites that $f(p)$ could be at a maximum. If the three sites "straddle" the center p, in the sense that they span more than a semicircle (as in Figure 5.3), then motion of p in any direction results in moving p closer to some site, and thus decreasing $f(p)$. We have now established this fact:

Lemma 5.5.3. *If the center p of a largest empty circle is strictly interior to the hull of the sites $\mathcal{H}(S)$, then p must be coincident with a Voronoi vertex.*

Note that it is not necessarily true that every Voronoi vertex represents a local maximum of $f(p)$ (Exercise 5.5.6[5]).

Centers on the Hull

Now let us consider circle centers p directly on the hull $H = \mathcal{H}(S)$. The reason our earlier arguments do not apply is that moving p to p' might move outside of the hull, and our problem specification restricted centers to the hull. We now argue even more informally than above that a maximal circle must include two sites.

Suppose $f(p)$ is a maximum with p on H and the circle includes just one site s_1. First, it cannot be that p is at a vertex of H, for the vertices of H are all sites themselves,

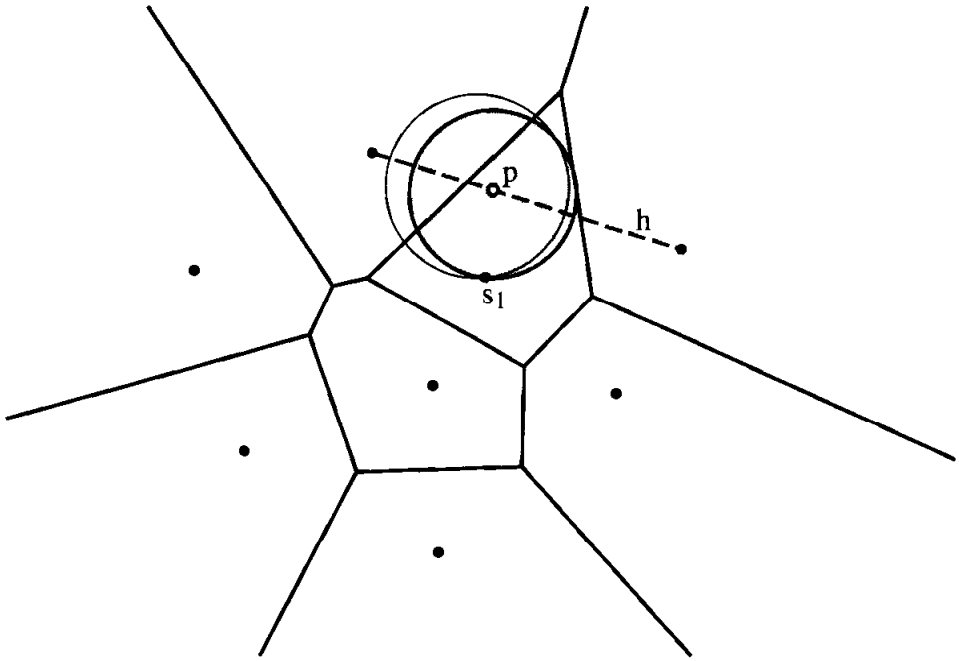


FIGURE 5.14 Center on hull edge h, circle through one site.

and this would imply that $f(p) = 0$. So p is on the interior of an edge h of H. Then moving p one way or the other along h must increase its distance from s_1, as shown in Figure 5.14. One can again see this intuitively by thinking of the cone apexed at s_1, sliced by a vertical plane (Figure 5.9).

If, however, the circle centered on p contains two sites s_1 and s_2, then it is possible that the direction along the bisector of the sites that increases distance is the direction that goes outside the hull. Thus it could well be that $f(p)$ is at a local maximum. We have shown this fact:

Lemma 5.5.4. *If the center p of a largest empty circle lies on the hull of the sites $\mathcal{H}(S)$, then p must lie on a Voronoi edge.*

Algorithm

We have now established our goal: we have found a finite set of points that are potential centers of largest empty circles: The Voronoi vertices and the intersections between Voronoi edges and the hull of the sites. This suggests the algorithm in Algorithm 5.1, due to Toussaint (1983a).[11]

Note that not every Voronoi vertex is necessarily inside the hull (Figure 5.14), which necessitates the $v \in H$ check in the algorithm. A naive implementation of this algorithm would require quadratic time in n, but locating a Voronoi vertex in H and intersecting a Voronoi edge with e can both be accomplished in $O(\log n)$ time, and these efficiencies lead to an $O(n \log n)$ algorithm overall. We leave details to Exercise 5.5.6[6].

[11]The main ideas go back to Shamos (1978).

Algorithm: LARGEST EMPTY CIRCLE
Compute the Voronoi diagram $\mathcal{V}(S)$ of the sites S.
Compute the convex hull $H = \mathcal{H}(S)$.
for each Voronoi vertex v do
    if v is inside H: $v \in H$ then
        Compute radius of circle centered on v and update max.
for each Voronoi edge e do
    Compute $p = e \cap \partial H$, the intersection of e with the hull boundary.
    Compute radius of circle centered on p and update max.
Return max.

Algorithm 5.1 Largest empty circle.

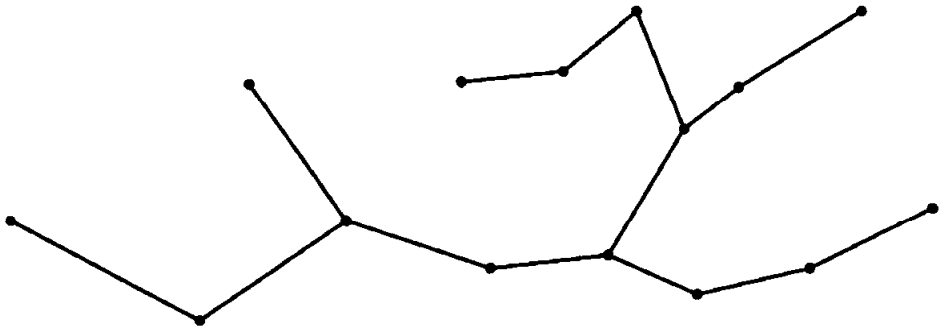


FIGURE 5.15 A Euclidean Minimum Spanning Tree.

5.5.4. Minimum Spanning Tree

A *minimum spanning tree* (MST) of a set of points is a *minimum* length tree that *spans* all the points: a shortest tree whose nodes are precisely those in the set. When the length of an edge is measured by the usual Euclidean length of the segment connecting its endpoints, the tree is often called the Euclidean minimum spanning tree, abbreviated EMST. Here we will only consider Euclidean lengths and so will drop the redundant modifier. An example is shown in Figure 5.15. MSTs have many applications. For example, many local area networks take the form of a tree spanning the host nodes. The MST is the network topology that minimizes total wire length, which usually minimizes both cost and time delays.

Kruskal's Algorithm

Here we will consider the problem of computing the MST of a set of points in the plane. Let us first look at the more general problem of computing the MST for a graph G. Although it is by no means obvious, a mindless greedy strategy finds the MST, based on the simple intuition that a shortest tree should be composed of the shortest edges. This suggests that such a tree can be built up incrementally by adding the shortest edge not yet explored, which also maintains treeness (acyclicity). This algorithm is known as Kruskal's algorithm and dates back to 1956.[12]

[12]My presentation is based on that of Albertson & Hutchinson (1988, pp. 264–8).

Let T be the tree incrementally constructed, and let the notation $T + e$ mean the tree T union the edge e. Kruskal's algorithm is shown in Algorithm 5.2. We will not stop to prove this algorithm correct but only claim that its complexity is dominated by the first sorting step. This requires $O(E \log E)$ time, where E is the number of edges in the graph.

Algorithm: KRUSKAL'S ALGORITHM
Sort all edges of G by length: $e_1, e_2, \ldots$.
Initialize T to be empty.
while T is not spanning do
    if $T + e_i$ is acyclic
        then $T \leftarrow T + e_i$
    $i \leftarrow i + 1$

Algorithm 5.2 Kruskal's algorithm.

MST $\subseteq \mathcal{D}(P)$

For the MST of points in the plane, there are $\binom{n}{2}$ edges, so the complexity of the sorting step is $O(n^2 \log n)$ if carried out on the complete graph. But recalling that the Delaunay triangulation edges record proximity information in some sense, it is reasonable to hope that only Delaunay edges ever need be used to construct an MST. And fortunately this is true, as shown by the following theorem.

Theorem 5.5.5. *A minimum spanning tree is a subset of the Delaunay triangulation: MST $\subseteq \mathcal{D}(P)$.*

Proof. We want to show that if $ab \in$ MST, then $ab \in \mathcal{D}$. Assume that $ab \in$ MST and suppose to the contrary that $ab \notin \mathcal{D}$. Then we seek to derive a contradiction by showing that the supposed MST is not minimal.

Recall that if $ab \in \mathcal{D}$, then there is an empty circle through a and b (Property **V7** and Theorem 5.3.1). So if $ab \notin \mathcal{D}$, no circle through a and b can be empty. In particular, the circle with diameter ab must have a site on or in it.

So suppose c is on or in this circle, as shown in Figure 5.16. Then $|ac| < |ab|$, and $|bc| < |ab|$; these inequalities hold even if c is on the circle, since c is distinct from a

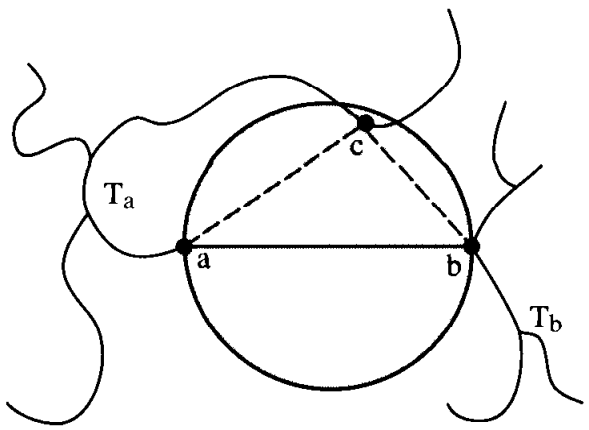


FIGURE 5.16 $T_a + bc + T_b$ is shorter than $T_a + ab + T_b$.

and b. Removal of ab will disconnect the tree into two trees, with a in one part, T_a, and b in the other, T_b. Suppose without loss of generality that c is in T_a. Remove ab and add edge bc to make a new tree, $T' = T_a + bc + T_b$. This tree is shorter, so the one using ab could not have been minimal. We have reached a contradiction by denying that ab is in $\mathcal{D}$, so it must be that $ab \in \mathcal{D}$. □

This then yields an improvement on the first step of Kruskal's algorithm: First find the Delaunay triangulation in $O(n \log n)$ time, and then sort only those $O(n)$ edges, in $O(n \log n)$ time. It turns out that the remainder of Kruskal's algorithm can be implemented to run in $O(n \log n)$, so that the total complexity for finding the MST for a set of n points in the plane is $O(n \log n)$.

5.5.5. Traveling Salesperson Problem

One of the most-studied problems in computer science is the Traveling Salesperson problem: Find the shortest closed path that visits every point in a given set. Such a path is called a *traveling salesperson path (TSP)*; imagine the points as cities that the salesperson must visit in arbitrary order before returning home. This problem has tremendous practical significance, not only for that application, but because many other problems can be reduced to it. Unfortunately, the problem has been proven to be NP-hard, a technical term that means that no polynomial algorithm is known to solve it (Garey & Johnson 1979); nor does it seem likely at this writing that one will be found. The combination of practical significance and intractability have led to a search for effective heuristics and approximation algorithms. One of the simplest approximation algorithms is based on the Delaunay triangulation, via the Minimum Spanning Tree.

The idea is rather simple-minded, but nevertheless it does a reasonable job: Find the MST for the set of points, and simply follow that out and back in the manner illustrated in Figure 5.17. It should be clear that the tour constructed this way has exactly twice the length of the MST, since each edge of the tree is traversed once in each direction.

We now obtain a bound on how bad this doubled-MST tour can be. Let M be the length of a minimum spanning tree and M_2 the length of a doubled-MST; of course $M_2 = 2M$. Let T be the length of a traveling salesperson path and T_1 the length of a TSP with one edge removed. Note that T_1 is a spanning tree.

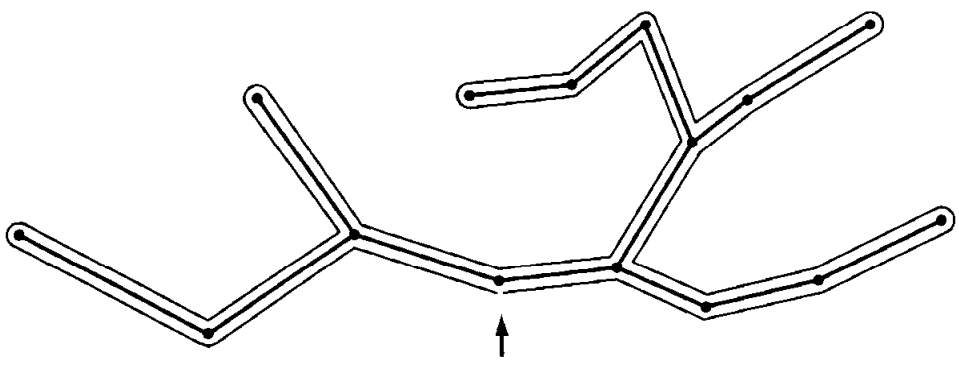


FIGURE 5.17 A tour formed by following the MST.

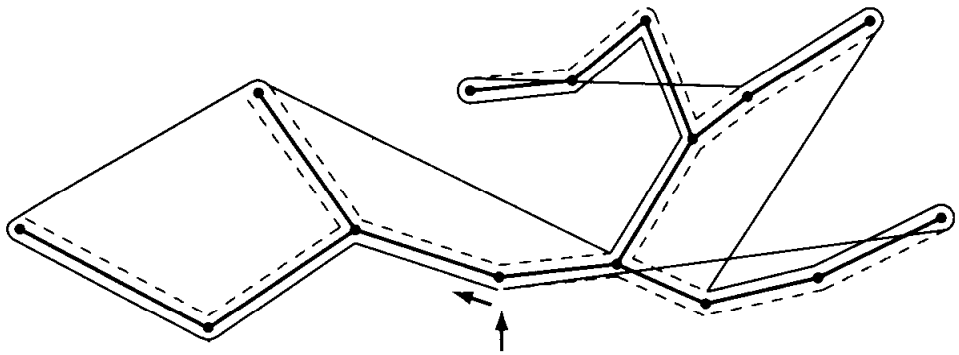


FIGURE 5.18 Shortcutting the doubled-MST tour in Figure 5.17.

The following inequalities are immediate:

$$T_1 < T,$$
$$M \leq T_1,$$
$$M < T,$$
$$M_2 < 2T.$$

This then achieves a constant upper bound on the quality of the tour: The doubled-MST is no worse than twice the optimal TSP length.

This result can be improved with various heuristics. I will sketch only the simplest such heuristic, which is based on the understandable resolve not to revisit a site twice. Traverse the doubled-MST path from the start site, with the modification that if the next site has already been visited by the path so far, skip that site and consider connecting to the next one along the doubled-MST tour. This has the effect of taking a more direct route to some sites. If we index the sites by the order in which they are visited along the doubled-MST tour, some site s_i might connect to s_j by a straight line segment in the shortcut tour, whereas in the doubled-MST tour it follows a crooked path $s_i, s_{i+1}, \ldots, s_{j-1}, s_j$. A straight path is always shorter than a crooked path (by the triangle inequality), so this heuristic can only shorten the path. An example is shown in Figure 5.18. Note that the shortened path might self-intersect.

Unfortunately this heuristic does not guarantee an improved performance, but a slight variation known as the "Christofides heuristic" does. It uses a set of segments called a "minimum Euclidean matching" as a guide to shortcutting and can guarantee a path length no more than $(3/2)T$, that is, no more than 50% longer than the optimum. More sophisticated heuristics generally find a path within a few percent of optimal (Bentley 1992), although this performance is not guaranteed as it is for the algorithm above. A recent exciting theoretical breakthrough is a "polynomial-time approximation scheme" for the TSP, about the best one can hope for an NP-complete problem. This is a method of getting within $(1+\epsilon)$ of optimal for any $\epsilon > 0$, in time $O(n^p)$, where p is proportional to $1/\epsilon$. See Arora (1996) and Mitchell (1996).

5.5.6. Exercises

1. *Degree of NNG.* What is the maximum out-degree of a node of a directed Nearest Neighbor Graph (NNG) (Section 5.5.1) of n points in two dimensions? What is the maximum

in-degree of a node? Demonstrate examples that achieve your answers, and try to prove they are maximum.

2. *NNG and $\mathcal{D}$* [easy]. Find an example that shows that NNG can be a proper subset of $\mathcal{D}(P)$.
3. *NNG $\subseteq$ D.* Prove Lemma 5.5.1: If b is a nearest neighbor of a, then $ab \in \mathcal{D}(P)$.
4. *Number of triangles in a triangulation.* Prove that the number of triangles t in any triangulation of some fixed point set S is a constant: All triangulations of S have the same t.
5. *Voronoi vertex not a local max.* Construct a set of points that has a Voronoi vertex p strictly inside the hull, such that $f(p)$ is not a local maximum, where f is the radius function defined in Section 5.5.3.
6. *Empty circle algorithm.* Detail (in pseudocode) how to implement the empty circle algorithm (Algorithm 5.1) so that its time complexity is $O(n \log n)$.
7. *Relative Neighborhood Graph.* The Relative Neighborhood Graph (RNG) of a set of points $p_1, \ldots, p_n$ is a graph whose nodes correspond to the points, and with two nodes p_i and p_j connected by an arc iff they are at least as close to each other as to any other point, that is, if

$$|p_i - p_j| \leq \max_{m \neq i,j} \{|p_i - p_m|, |p_j - p_m|\}. \tag{5.3}$$

   (See Jaromczyk & Toussaint (1992).) This equation determines a "forbidden" region within which no point p_m may lie if p_i and p_j are adjacent in the RNG, not unlike Theorem 5.3.1. This region, called $Lune(p_i, p_j)$, is the intersection of two open disks centered on p_i and p_j, both of radius $|p_i - p_j|$.

   a. Design a "brute-force" algorithm to construct the RNG. Do not worry about efficiency. What is its time complexity?

   b. Prove that $RNG \subseteq \mathcal{D}(P)$: Every edge of the RNG is also an edge of the Delaunay triangulation. (Compare with Theorem 5.5.5.)

   c. Use (b) to design a faster algorithm.

8. *Size of Delaunay triangulation in three dimensions.* We have shown that the size of the Delaunay triangulation in two dimensions is linear, $O(n)$. Show that this does not hold in three dimensions: The size of $\mathcal{D}(P)$ can be quadratic. Define $\mathcal{D}(P)$ in three dimensions exactly analogously to the two-dimensional version: It is the dual of $\mathcal{V}(P)$, which is the locus of points that do not have a unique nearest neighbor.

   Let P be a point set consisting of two parts:[13]

   a. $n/2$ points uniformly distributed around a circle in the xy-plane centered on the origin, and

   b. $n/2$ points uniformly distributed on the z axis symmetrical about the origin.

   Argue that the size of $\mathcal{D}(P)$ is $\Omega(n^2)$.

9. *Size of Relative Neighborhood Graph in three dimensions.* Exercise [7] above established that $RNG \subseteq \mathcal{D}(P)$ in two dimensions, and this relationship holds in arbitrary dimensions. It has been proved that the size of the RNG in three dimensions is $O(n^{4/3})$ (Jaromczyk & Toussaint 1992), so it is smaller than the Delaunay triangulation. But it appears that this upper bound is weak: Jaromczyk & Kowaluk (1991) conjecture that the size is $O(n)$. Confirming this conjecture is an open problem.

   Try to determine what the RNG is for the example in Exercise [8] above, which established that $\mathcal{D}(P)$ can be quadratic.

10. *MST $\subseteq$ RNG.* Prove that every edge of an MST is an edge of the RNG. (Compare with Theorem 5.5.5.)

[13] This example is from Preparata & Shamos (1985, Fig. 4.3).

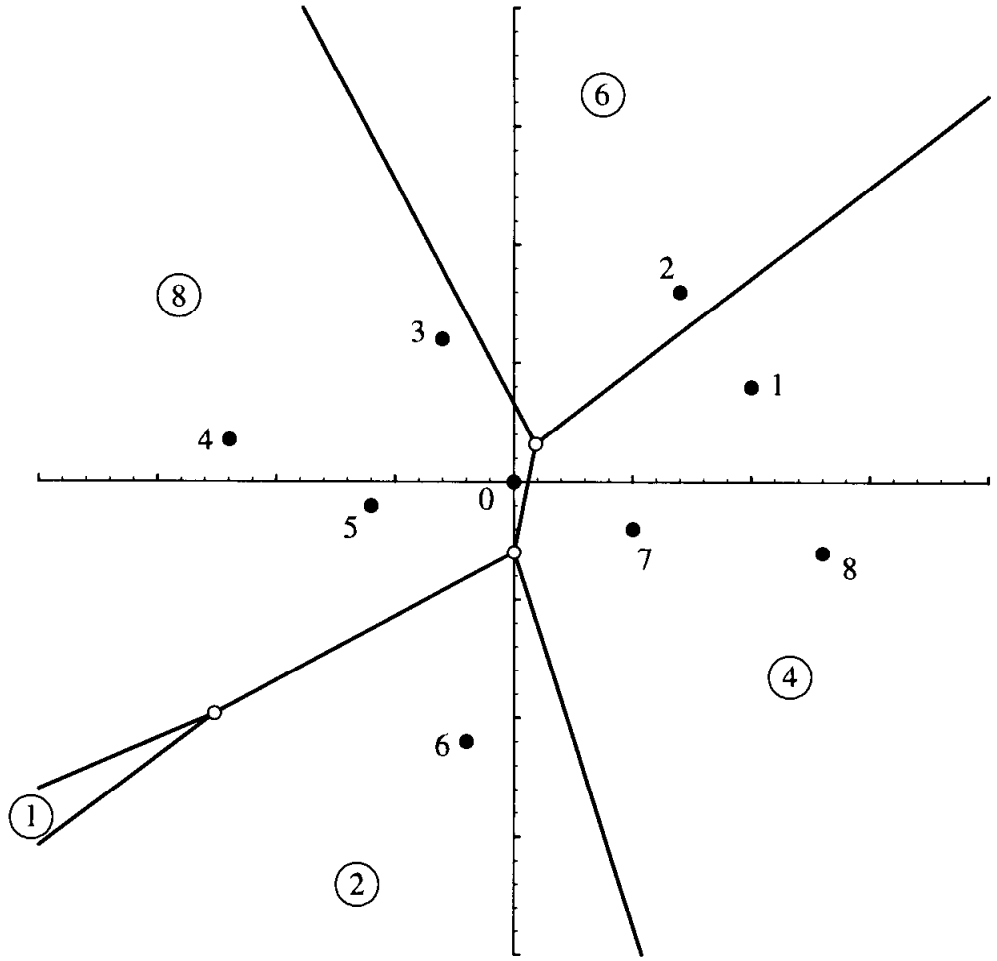


FIGURE 5.19 A furthest-point Voronoi diagram for $n = 9$ points. There are six regions, whose site indices are circled; the region for site 3 is offscreen. Sites {0, 5, 7} are not the furthest neighbor of any point in the plane.

11. *Furthest-point Voronoi diagram.* Define the furthest-point Voronoi diagram $\mathcal{F}(P)$ to associate each point of the plane to the site that is its "furthest neighbor," the site that is furthest away. Points with one furthest neighbor form a furthest-neighbor Voronoi region; points with two furthest neighbors form the edges of $\mathcal{F}(P)$. See Figure 5.19.
    a. What is $\mathcal{F}(P)$ for two sites?
    b. What is $\mathcal{F}(P)$ for three sites?
    c. Derive some structural properties of furthest-point Voronoi diagrams, similar to the Delaunay and Voronoi properties in Sections 5.3.1 and 5.3.2. Use Figure 5.19 to help form hypotheses.
12. *Minimum spanning circle.* Show how the furthest-point Voronoi diagram can be used to compute the smallest-radius circle that surrounds a given point set. Assume $\mathcal{F}(P)$ is available.

5.6. MEDIAL AXIS

The Voronoi diagram may be generalized in several directions, and some of these generalizations have considerable practical significance. In this section we touch on just one generalization, one of the simplest: allowing the set of sites to be an infinite set of points, in particular the continuous boundary of a polygon.

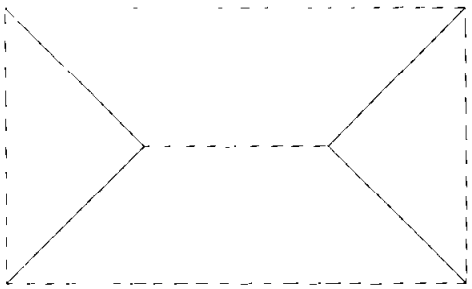


FIGURE 5.20 Medial axis of a rectangle.

In Section 5.2 we defined the Voronoi diagram as the set of points whose nearest site is not unique: These points are equidistantly closest to two or more sites. Define the *medial axis*[14] of a polygon P to be the set of points inside P that have more than one closest point among the points of ∂P. A very similar definition can be used for an arbitrary collection of points, but here we will examine only the case where the points form the boundary of a polygon.

The medial axis of a rectangle is shown in Figure 5.20. Each point on the horizontal segment inside the rectangle is equidistant from points vertically above and below it on the top and bottom sides of the rectangle. Each point on a diagonal segment is equidistant from two adjacent sides of the rectangle. And the two endpoints of the horizontal segment are equidistant from three sides of the rectangle.

A more complex example is shown in Figure 5.21, an eight-vertex convex polygon. One might guess from this example that the medial axis of a convex polygon P is a tree whose leaves are the vertices of P. This is indeed true, and is even true for nonconvex polygons. Every point of the medial axis is the center of a circle that touches the boundary in at least two points. And just as Voronoi vertices are centers of circles touching three sites, vertices of the medial axis are centers of circles touching three distinct boundary points, as shown in Figure 5.22.

Sometimes the medial axis of P is defined as the locus of centers of *maximal circles*: circles inside P that are not themselves enclosed in any other circle inside P. The process of transforming a shape into its medial axis is sometimes called the "grassfire transformation," for if one imagines the polygon P as a field of dry grass, then lighting afire the boundary of P all at once will cause the fire to burn inward at a uniform rate, and the medial axis is the set of "quench points" – where fire meets fire from another direction. The connection between this analogy and the forest fires discussed in Section 5.1 should be evident.

The medial axis was introduced by Blum (1967) for studying biological shape. He viewed it as something like a skeleton (axis) that threads down the middle (median) of a shape. This is less apparent for a convex polygon than it is for nonconvex and smooth shapes, which were Blum's main interest. One can characterize a shape to a certain extent from the structure of its medial axis, and this has led to considerable interest among researchers in pattern recognition and computer vision (Bookstein 1978). For example, Bookstein (1991, pp. 80–7) uses it to characterize the differences between normal mandible bones and deformed ones. It can be used to compute an inward *offset* of a polygon: A shrunken version of a polygon, all of whose boundaries are offset inward by

[14]This is also known as the "symmetric axis" or the "skeleton" of the polygon.

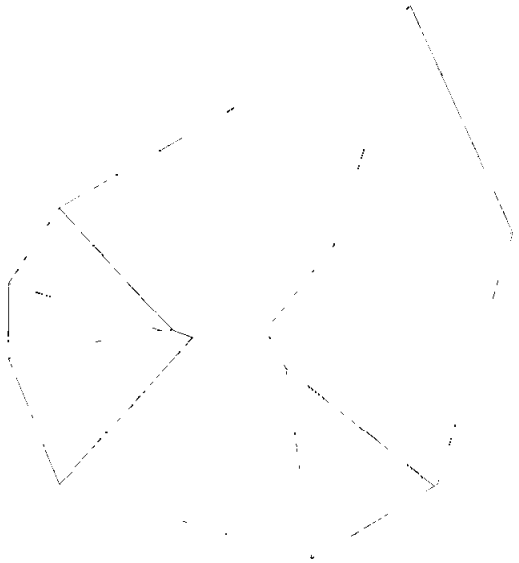


FIGURE 5.21 Medial axis of a convex polygon of eight vertices.

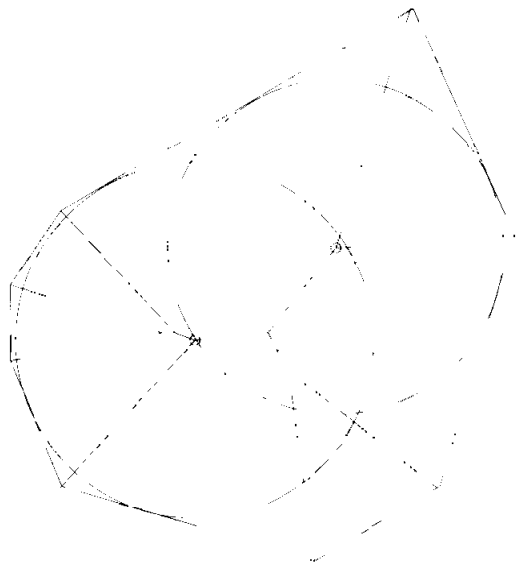


FIGURE 5.22 Circles centered on vertices touch the polygon boundary at three points.

a fixed distance. Expanded or outward offsets rely on the exterior version of the medial axis. Computing offsets is an important problem in manufacturing, where engineering tolerances lead naturally to offset shapes (Saeed, de Pennington & Dodsworth 1988).

The medial axis of a polygon of n vertices can be constructed in $O(n \log n)$ time (Lee 1982); asymptotically slower but more practical algorithms are available (Yao & Rokne 1991). For convex polygons, $O(n)$ time suffices (Aggarwal, Guibas, Saxe & Shor 1989).

5.6.1. Exercises

1. *Medial axis of a nonconvex polygon.* Show by example that the medial axis of a nonconvex polygon can contain curved segments. What can you say about the functional form of these curves?
2. *Medial axis and Voronoi diagram.* Is there any relationship between the medial axis of a convex polygon P and the Voronoi diagram of the vertices of P? Conjecture some aspects of this relationship and either prove them or construct counterexamples.

3. *Medial axis of a polytope.* Describe what the medial axis of a convex polytope must look like.
4. *Straight skeleton.* Aichholzer, Alberts, Aurenhammer & Gärtner (1995) introduced a skeleton that is similar to the medial axis, but composed of straight segments even for nonconvex polygons. Move each edge of a polygon parallel to itself inward at constant velocity, with adjacent edges shrinking and growing so that vertices travel along angle bisectors. When an edge shrinks to zero length, its neighboring edges become adjacent. When a reflex vertex bumps into an edge, the polygon is split and the shrinking process continues on the pieces.

   Work out the straight skeleton by hand for a few shapes of simply connected letters of the alphabet: T, E, X. Form some conjectures about the properties of the straight skeleton.

5.7. CONNECTION TO CONVEX HULLS

In 1986 Edelsbrunner & Seidel discovered a beautiful connection between Delaunay triangulations and convex hulls in one higher dimension.[15] I will first explain this connection between two-dimensional convex hulls and one-dimensional Delaunay triangulations (which are admittedly trivial) and then generalize to two-dimensional Delaunay triangulations and three-dimensional convex hulls. This connection will then give us an easy method for computing the Delaunay triangulation, and from that the Voronoi diagram, via three-dimensional hulls.

5.7.1. One-Dimensional Delaunay Triangulations

We start in one dimension, where the mathematics is transparent.

Let $P = \{x_1, \ldots, x_n\}$ be a set of points on the x axis. Clearly the one-dimensional Delaunay triangulation is simply the path connecting x_1 to x_2 to ... to x_n. But we will view this as a projection onto the x axis of a set of two-dimensional points with coordinates (x_i, x_i^2). These points can be viewed as the projection of the x_i's upwards to the parabola $z = x^2$. Now it is trivially true that the convex hull of these two-dimensional points project down to the one-dimensional Delaunay triangulation, as long as the "top" edge of the hull is discarded. But there is much more here than this trivial observation, which can be elucidated by considering tangents to the parabola.

The slope of the parabola $z = x^2$ at the point $x = a$ is $2a$ (because $dz/dx = 2x$). Thus the equation of the line tangent to the parabola at the point (a, a^2) is

$$\begin{aligned} z - a^2 &= 2a(x - a), \\ z &= 2ax - a^2. \end{aligned} \tag{5.4}$$

In preparation for studying the same process in three dimensions, we now investigate the intersection between this tangent and the parabola when the tangent is translated vertically by a distance r^2. When the tangent is raised by this amount, its equation becomes

$$z = 2ax - a^2 + r^2. \tag{5.5}$$

[15] Their insight was based on earlier work of Brown (1979), who was the first to establish a connection to convex hulls.

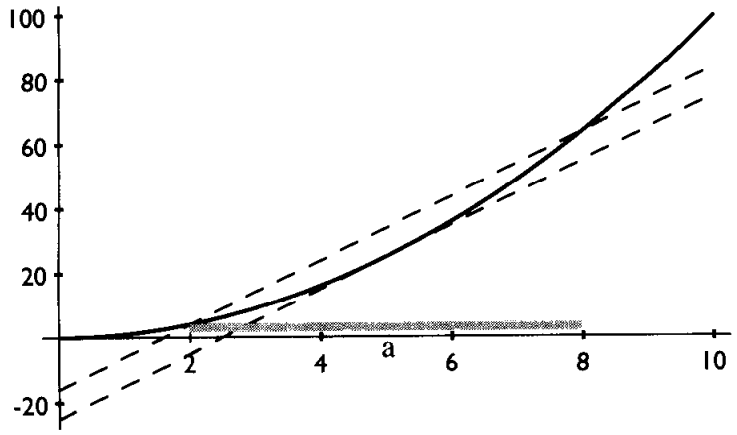


FIGURE 5.23 For $a = 5$, the tangent is $z = 10x - 25$.

Where does this line intersect the parabola? Whenever

$$\begin{aligned} z = x^2 &= 2ax - a^2 + r^2, \\ (x-a)^2 &= r^2, \\ x &= a \pm r. \end{aligned} \tag{5.6}$$

So the raised tangent intersects the parabola $\pm r$ away from a, the original point of tangency. Note that $x = a \pm r$ can be thought of as the equation of a one-dimensional circle of radius r centered on a. This is illustrated in Figure 5.23, with $a = 5$ and $r = 3$, so that the "disk" is the segment [2, 8].

5.7.2. Two-Dimensional Delaunay Triangulations

Now we repeat the same analysis in two dimensions.

The paraboloid is $z = x^2 + y^2$, see Figure 5.24. Take the given sites/points in the plane, and project them upwards until they hit the paraboloid, that is, map every point as follows:

$$(x_i, y_i) \mapsto \left(x_i, y_i, x_i^2 + y_i^2\right). \tag{5.7}$$

Take the convex hull of this set of three-dimensional points; see Figure 5.25. Now discard the "top" faces of this hull: all those faces whose outward pointing normal points upward, in the sense of having a positive dot product with the z axis vector. The result is a bottom "shell." Project this to the xy-plane. The claim is that this is the Delaunay triangulation! See Figure 5.26. We now establish this stunning connection formally.

The equation of the tangent plane above the point (a, b) is

$$z = 2ax + 2by - (a^2 + b^2). \tag{5.8}$$

(This is a direct analogy to the equation $z = 2ax - a^2$: $\partial z/\partial x = 2x$ and $\partial z/\partial y = 2y$.) Now shift this plane upwards by r^2, just as we shifted the tangent line upward in the previous subsection:

$$z = 2ax + 2by - (a^2 + b^2) + r^2. \tag{5.9}$$

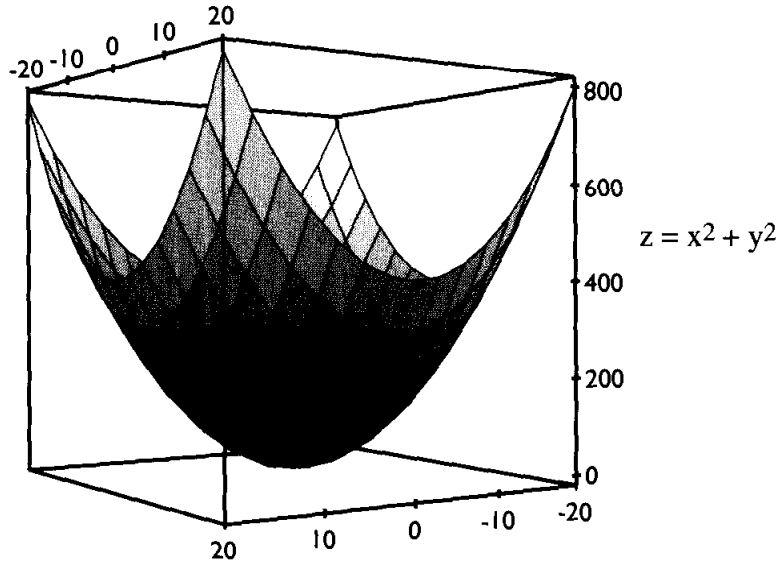


FIGURE 5.24 The paraboloid up to which the sites are projected.

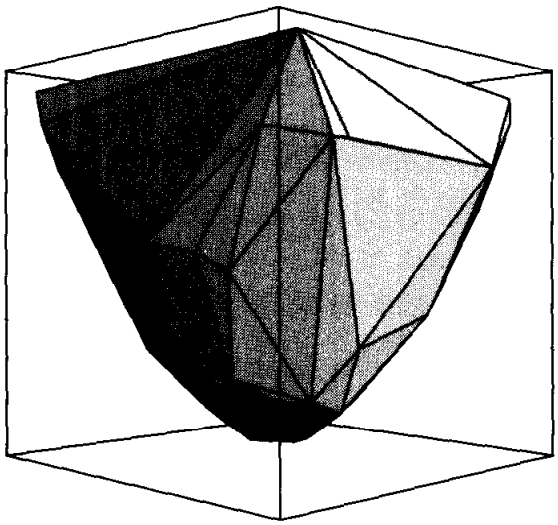


FIGURE 5.25 The convex hull of 65 points projected up to the paraboloid

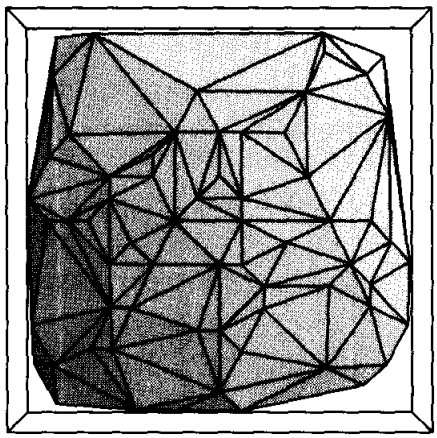


FIGURE 5.26 The paraboloid hull viewed from $z \approx -\infty$.

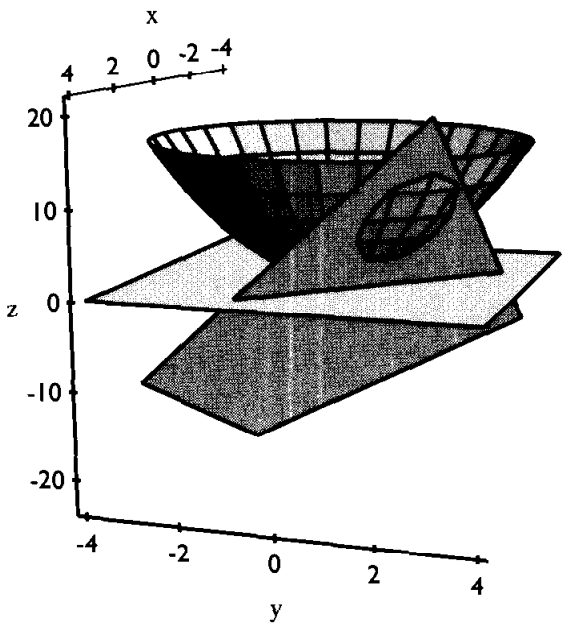


FIGURE 5.27 Plane for $(a, b) = (2, 2)$ and $r = 1$ cutting the paraboloid.

Again ask, where does this shifted plane intersect the paraboloid?

$$\begin{aligned} z = x^2 + y^2 &= 2ax + 2by - (a^2 + b^2) + r^2, \\ (x - a)^2 + (y - b)^2 &= r^2. \end{aligned} \tag{5.10}$$

The shifted plane intersects the paraboloid in a curve (an ellipse) that projects to a circle! This is illustrated in Figures 5.27 and 5.28.

Now we reverse the viewpoint to lead us to the Delaunay triangulation. Consider the plane π through three points on the paraboloid $\Delta = (p_i, p_j, p_k)$ that form a face of the convex hull in three dimensions. This plane slices through the paraboloid. If we translate π vertically downward, at some point it will cease to intersect the paraboloid. Let us say that the last point it touches is $(a, b, a^2 + b^2)$. Then we can view π as an upward shift of this tangent plane τ; call the shift amount r^2. Now it should be clear that the previous analysis applies.

Since Δ is a lower face of the hull, all of the other points of the paraboloid are above π. Since they are above π, they are more than r^2 above τ, which is r^2 below π. Therefore these points project outside of the circle of radius r in the xy-plane. Therefore the circle determined by Δ in the xy-plane is empty of all other sites. Therefore it forms a

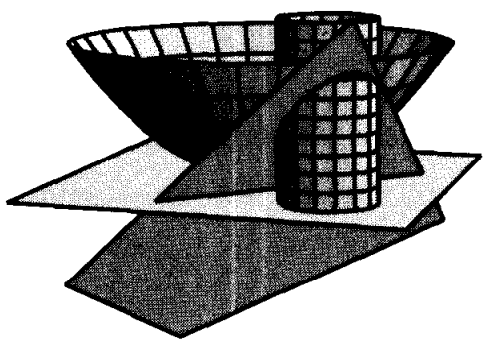


FIGURE 5.28 The curve of intersection in Figure 5.27 projects to a circle of radius 1 in the xy-plane.

Delaunay triangle. Therefore every lower triangular face of the convex hull corresponds to a Delaunay triangle. Therefore the projection of the "bottom" of the convex hull projects to the Delaunay triangulation! Again consult Figure 5.26.

Let me explain this important insight again in another way. Start with the plane τ tangent to the paraboloid above $p = (a, b)$. Its point of contact projects downward to p. Now move τ upwards. The projection of its intersection with the paraboloid is an expanding circle centered on p. When τ hits a point q on the paraboloid that is above a site, the expanding circle bumps into the site on the plane that is the projection of q. Thus the circle is empty until τ reaches π, when it passes through the three sites whose projection forms the triangle hull face Δ supported by π.

A useful corollary to the above discussion is this:[16]

Corollary 5.7.1. *Four points* (x_i, y_i), $i = 1, 2, 3, 4$, *lie on a circle iff* $(x_i, y_i, x_i^2 + y_i^2)$ *lie on a plane.*

The coplanarity of these points can be checked by seeing if the volume of the tetrahedron they determine (Equations 1.15 and 4.6) is zero.

5.7.3. Implications

Theorem 5.7.2. *The Delaunay triangulation of a set of points in two dimensions is precisely the projection to the xy-plane of the lower convex hull of the transformed points in three dimensions, transformed by mapping upwards to the paraboloid* $z = x^2 + y^2$.

Since the convex hull in three dimensions can be computed in $O(n \log n)$ time (Section 4.2.2), this implies that the Delaunay triangulation can be computed in the same time bound. Once the Delaunay triangulation is in hand, it is relatively easy to compute the Voronoi diagram (Exercise 5.7.5[2]). This leads to another $O(n \log n)$ algorithm for constructing the Voronoi diagram.

As one might hope, this relationship between Voronoi diagrams and convex hulls in one higher dimension holds in arbitrary dimensions. Thus both the Voronoi diagram and the Delaunay triangulation in three dimensions can be constructed from a convex hull in four dimensions. In fact, it may be that the most common use of 4D hull code is for constructing solid meshes of Delaunay tetrahedra. In general, the Voronoi diagram dual for a set of d-dimensional points is the projection of the "lower" hull of points in $d + 1$ dimensions.

5.7.4. Implementation of Delaunay Triangulation: $O(n^4)$ Code

Theorem 5.7.2 allows amazingly concise code to compute the Delaunay triangulation, if one is unconcerned about time complexity. In particular, if $O(n^4)$ is acceptable (and it rarely is), the Delaunay triangulation can be computed with less than thirty lines of

[16] See Pedoe (1970, p. 146) for a proof.

C code! This is presented in Code 5.1 partly as a curiosity, but also to emphasize how deep understanding of geometry can lead to clean code.

```
main()
{
  int x[NMAX],y[NMAX],z[NMAX];      /* input points xy,z=x^2+y^2 */
  int n;                            /* number of input points */
  int i, j, k, m;                   /* indices of four points */
  int xn, yn, zn;                   /* outward normal to (i,j,k) */
  int flag;                         /* t if m above of (i,j,k) */

  /* Input points and compute z = x^2 + y^2. */
  scanf("%d," &n);
  for ( i = 0; i < n; i++ ) {
    scanf("%d %d," &x[i], &y[i]);
    z[i] = x[i] * x[i] + y[i] * y[i];
  }
  /* For each triple (i,j,k) */
  for ( i = 0; i < n - 2; i++ )
  for ( j = i + 1; j < n; j++ )
  for ( k = i + 1; k < n; k++ )
  if ( j != k ) {
    /* Compute normal to triangle (i,j,k). */
    xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
    yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
    zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);

    /* Only examine faces on bottom of paraboloid: zn < 0. */
    if ( flag = (zn < 0) )
      /* For each other point m */
      for (m = 0; m < n; m++)
        /* Check if m above (i,j,k). */
        flag = flag &&
          ((x[m]-x[i])*xn +
           (y[m]-y[i])*yn +
           (z[m]-z[i])*zn <= 0);
    if (flag)
      printf("%d\t%d\t%d\n," i,j,k);
  }
}
```

Code 5.1 $O(n^4)$ Delaunay triangulation algorithm: `dt4.c`.

The $O(n^4)$ structure of the code is evident in the four nested for-loops. For each triple of points (i, j, k), the program checks to see if all other points m are on or above the plane containing i, j, and k. If so, (i, j, k) are output as a Delaunay triangle. (The similarity to the two-dimensional hull algorithm in Algorithm 3.2 should be evident.)

The above-plane test is performed by dotting the outward-pointing normal to the triangle, (x_n, y_n, z_n), with a vector from point i to point m.

Although it is interesting to see such concise code compute an object like the Delaunay triangulation, it is impractical for several reasons:

1. The code outputs all triangles inside a Delaunay face whose boundary consists of four or more cocircular points. Thus a square face results in four triangles output, representing the two triangulations of the square. Obtaining more useful output would require postprocessing.
2. There are obvious inefficiencies in the code (for example, the m-loop could break when one point is discovered below the ijk plane). These could be easily repaired at the cost of lengthening it a bit, but ...
3. The n^4 time dependence is unacceptable. For two test runs with $n = 100$ and $n = 200$ points, the computation time was 12 and 239 seconds respectively,[17] exhibiting a more than $2^4 = 16$-fold increase for twice the number of points. This indicates that $n = 1{,}000$ would take several days.

3D Hull to Delaunay Triangulation: $O(n^2)$ Code

It is an easy task to convert the quadratic code developed in Chapter 4 for constructing the 3D hull into quadratic code for constructing the Delaunay triangulation. All the complexity is in the hull code (about 1,000 lines of code) and the reasoning that went into Theorem 5.7.2. The additional modifications are relatively minor. First, the `ReadVertices` routine (Code 4.10) should read in x and y and compute $z = x^2 + y^2$. Second, after the entire hull is constructed, a procedure `LowerFaces` is called to loop over all faces and identify which are on the lower hull (Code 5.2). As in Code 5.1, this is accomplished by computing the z coordinate of a vector normal to each face f. This computation, embodied in `Normz`, is just a slight change to `Collinear` (Code 4.12). If `Normz( f ) < 0`, then the face is on the lower hull, and its projection onto the xy-plane is a Delaunay triangle.

We illustrate with an example of $n = 10$ points whose coordinates are displayed in Table 5.1. Note that the z coordinates are pumped up in magnitude by the squaring. This small example stays well within the safe range of the volume computation discussed in Section 4.3.5, but the squaring does reduce the range of applicability of this code in comparison to the 3D hull code. The Postscript output of the code is shown in Figure 5.29.

As expected, this $O(n^2)$ implementation is much faster than the $O(n^4)$ code: On the same $n = 100$ and $n = 200$ examples that the slower code used 12 and 239 seconds, the faster code used 0.2 and 1.4 seconds respectively. Moreover, the randomized speedup discussed in Section 4.5 turns this into an $O(n \log n)$ expected-time algorithm.

5.7.5. Exercises

1. *Range of* `dt2.c` [programming]. Find a point set whose coordinates are as small as possible, and for which the `dt2.c` code (Code 5.2) returns an incorrect result due to overflow of the volume calculation (cf. Exercise 4.3.6[12]).

[17] On an 80 MHz Sun workstation.

```
void   LowerFaces( void )
{
  tFace f = faces;
  int Flower = 0;      /* Total number of lower faces. */

  do {
    if ( Normz( f ) < 0 ) {
       Flower++;
       printf("lower face indices: %d, %d, %d\n,"
          f->vertex[0]->vnum,
          f->vertex[1]->vnum,
          f->vertex[2]->vnum );
    }
    f = f->next;
  }while ( f != faces );
  printf("%d lower faces identified. \n," Flower);
}

int     Normz( tFace f )
{
  tVertex a, b, c;

  a = f->vertex[0];
  b = f->vertex[1];
  c = f->vertex[2];
  return
     ( b->v[X] - a->v[X] ) * ( c->v[Y] - a->v[Y] ) -
     ( b->v[Y] - a->v[Y] ) * ( c->v[X] - a->v[X] );
}
```

Code 5.2 `Lowerfaces` and `Normz`: additions to `chull.c` to form `dt2.c`.

2. $\mathcal{D}(P) \Rightarrow \mathcal{V}(P)$ [programming]. Modify the `dt2.c` code to compute the Voronoi diagram from the Delaunay triangulation. (See Exercise 5.5.6[1].) It will be necessary to repeatedly construct circles through three given points a, b, c. The coordinates of the center $p = (p_0, p_1)$ of this circle can be computed as follows:

$$\begin{aligned} A &= b_0 - a_0, \\ B &= b_1 - a_1, \\ C &= c_0 - a_0, \\ D &= c_1 - a_1, \\ E &= A(a_0 + b_0) + B(a_1 + b_1), \\ F &= C(a_0 + c_0) + D(a_1 + c_1), \\ G &= 2(A(c_1 - b_1) - B(c_0 - b_0)), \\ p_0 &= (DE - BF)/G, \\ p_1 &= (AF - CE)/G. \end{aligned} \tag{5.11}$$

Table 5.1. Coordinates of Delaunay sites, including $z = x^2 + y^2$.

i	x	y	$x^2 + y^2$
0	31	−76	6737
1	−13	21	610
2	−63	−83	10858
3	−5	−66	4381
4	87	−94	16405
5	40	71	6641
6	23	−46	2645
7	64	−80	10496
8	0	−57	3249
9	−14	2	200

FIGURE 5.29 Delaunay triangulation of the points displayed in Table 5.1. The origin is marked with '+.'

(The somewhat awkward form of these equations reduces the number of multications to reach the final coordinates.) If $G = 0$ then the three points are collinear and no finite-radius circle through them exists. Otherwise, the radius of the circle is

$$r^2 = (a_0 - p_0)^2 + (a_1 - p_1)^2.$$

Output coordinates for all the Voronoi vertices. For each finite-length Voronoi edge, output its two endpoints (either their coordinates or an index into your Voronoi vertex list). For each unbounded Voronoi edge-ray, output its endpoint and a vector (of arbitrary length) along the ray, oriented toward infinity.

3. *Furthest-point Voronoi diagram.* Argue that the "top" of the convex hull of the transformed points is the dual of the furthest-point Voronoi diagram. See Exercise 5.5.6[11] for a definition of this diagram. The "top" faces are those whose outward normals have a positive z

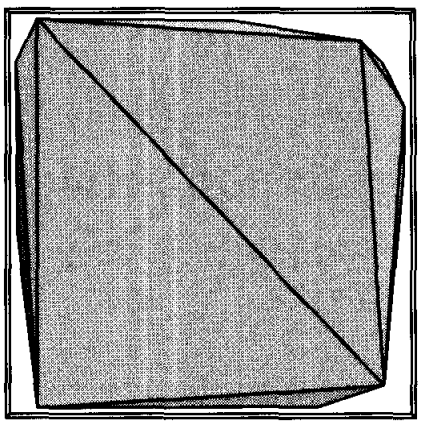


FIGURE 5.30 View of the hull in Figure 5.25 seen from $z \approx +\infty$.

component. Thus the view of the paraboloid hull from $z = +\infty$ shows the dual of $\mathcal{F}(P)$! See Figure 5.30.

4. *Circular separability.* Given two sets of planar points A and B, design an algorithm for finding (if it exists) a closed disk that encloses every point of A but excludes every point of B.

5.8. CONNECTION TO ARRANGEMENTS

We have shown how the Delauany triangulation can be derived from the paraboloid transformation and indicated that it is then easy to obtain the Voronoi diagram. It is also possible to obtain the Voronoi diagram directly from the paraboloid transformation. Although a full understanding of this will have to await the next chapter (Section 6.6), we will sketch the connection now while the relevant equations are nearby.

5.8.1. One-Dimensional Voronoi Diagrams

Consider two tangents to the parabola examined in Section 5.7.1 (Equation 5.4), one above $x = a$ and the other above $x = b$:

$$\begin{aligned} z &= 2ax - a^2, \\ z &= 2bx - b^2. \end{aligned} \tag{5.12}$$

Where do they intersect? Solving these equations simultaneously yields

$$\begin{aligned} 2ax - a^2 &= 2bx - b^2, \\ x(2a - 2b) &= a^2 - b^2, \\ x &= \frac{(a+b)(a-b)}{2(a-b)}, \\ x &= \frac{a+b}{2}. \end{aligned} \tag{5.13}$$

Therefore, the intersections of adjacent tangents projects to the one-dimensional Voronoi diagram of the point set.

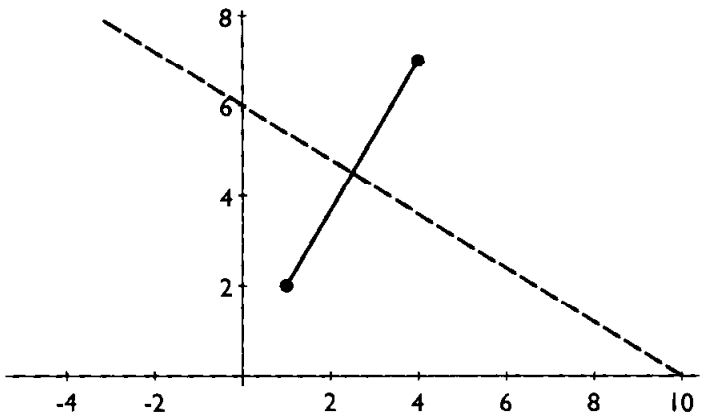


FIGURE 5.31 Bisector of (1, 2) and (4, 7) is $x(-6) + y(-10) = -60$, or $y = (-6/10)x + 6$.

5.8.2. Two-Dimensional Voronoi Diagrams

Consider two tangent planes to the paraboloid analyzed in Section (5.7.2) (Equation 5.8), one above (a, b) and the other above (c, d):

$$z = 2ax + 2by - (a^2 + b^2), \tag{5.14}$$

$$z = 2cx + 2dy - (c^2 + d^2). \tag{5.15}$$

Where do they intersect? Solving these equations simultaneously yields

$$\begin{aligned} 2ax + 2by - (a^2 + b^2) &= 2cx + 2dy - (c^2 + d^2), \\ x(2a - 2c) + y(2b - 2d) &= (a^2 - c^2) + (b^2 - d^2). \end{aligned} \tag{5.16}$$

This equation is precisely the perpendicular bisector of the segment from (a, b) to (c, d). See Figure 5.31.

If we view the (opaque) tangent planes from $z = +\infty$ (with the paraboloid transparent), then they would only be visible up to their first intersection. Their first intersection is the bisector between the sites that generate the tangent planes. The projection of these first intersections is precisely the Voronoi diagram!

So we have the remarkable situation that viewing the points projected onto the paraboloid from $z = -\infty$ one sees the Delaunay triangulation, and viewing the planes tangent to the paraboloid at those points from $z = +\infty$, one sees the Voronoi diagram.

Further Reading

Several surveys cover algorithms for constructing Voronoi diagrams: Aurenhammer (1991), Fortune (1992), and Fortune (1997). The book by Okabe et al. (1992) covers applications as well as algorithms.

6

Arrangements

6.1. INTRODUCTION

Arrangements of lines (and planes) form the third important structure used in computational geometry, as important as convex hulls and Voronoi diagrams. And as we glimpsed at the end of the previous chapter, and will see more clearly in Section 6.6, all three structures are intimately related. An arrangement of lines is shown in Figure 6.1. It is a collection of (infinite) lines "arranged" in the plane. These lines induce a partition of the plane into convex regions (called *cells*, or faces), segments or *edges* (between line crossings), and vertices (where lines meet). The example in the figure has $V = 45$ vertices, $E = 100$ edges, and $F = 56$ faces; not all of these are visible within the limited window of the figure. It is this partition that is known as the *arrangement*. It is convenient to view the faces as open sets (not including their edges) and the edges as open segments (not including their bounding vertices), so that the dissection is a true partition: Its pieces cover the plane, but the pieces are disjoint from one another, "pairwise disjoint" in the idiom preferred by mathematicians.

Arrangements may seem too abstract to have much utility, but in fact they arise in a wide variety of contexts. Here are four; more will be discussed in Section 6.7.

1. *Visibility Graphs*
   Let S be a set of n disjoint segments with no three endpoints collinear. The *endpoint visibility graph* has a node for each endpoint and an arc between endpoints x and y if the open segment (x, y) does not touch any segment in S: So x and y can see one another clearly. Usually arcs corresponding to the segments themselves are also included in the graph. This graph has application in robotics, as we will see in Chapter 8 (Section 8.2). A naive algorithm for constructing this graph has complexity $O(n^3)$, for each x and y, spend $O(n)$ time checking (x, y) against all segments. Employing arrangements leads to an $O(n^2)$ algorithm (O'Rourke 1987, pp. 211–17).
2. *Hidden Surface Removal*
   Hidden surface removal is the process of computing which surfaces in a three-dimensional scene are hidden from the viewpoint, and using this to construct a two-dimensional graphics image. The first worst-case optimal $\Theta(n^2)$ algorithm found depends on arrangements (McKenna 1987) (see Section 6.7.2).
3. *Empty Convex Polygons*
   Given a set of n points in the plane, find the *largest empty convex polygon* whose vertices are drawn from S. Here "largest" means the most vertices. This problem is inspired by an unresolved question posed by Erdős: It is unknown whether every sufficiently large set of points must contain an empty hexagon (Horton 1983).

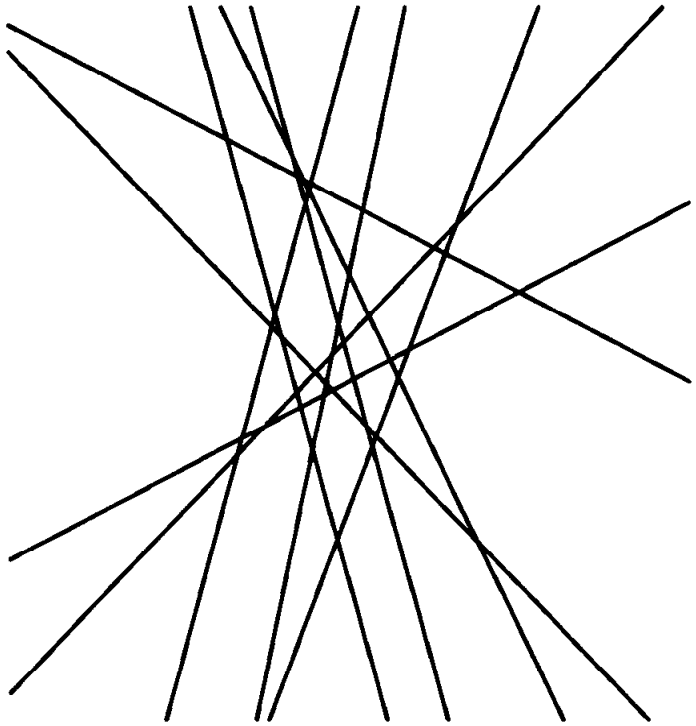


FIGURE 6.1 An arrangement of ten lines.

Using arrangements, the largest empty convex polygon can be found in $O(n^3)$ time (Edelsbrunner & Guibas 1989; Dobkin, Edelsbrunner & Overmars 1990).

4. *Ham-Sandwich Cuts*
   It is a remarkable theorem that any ham and cheese sandwich may be cut by a plane so that the two halves have precisely the same amount of bread, ham, and cheese! The two-dimensional version of this theorem states that there is always a line that simultaneously bisects two point sets. Arrangements permit finding this bisection in time linear in the size of the sets (see Section 6.7.6).

This chapter will develop the fundamentals of arrangements of lines but will not delve deeply enough to explain all four of the above applications. Rather, my goal is to sketch the essentials and leave the remainder for other sources.[1] This chapter contains no implementations, and may be the most challenging of the book in its degree of abstraction.

6.2. COMBINATORICS OF ARRANGEMENTS

An arrangement of lines is called *simple* if every pair of lines meet in exactly one point and no three lines meet in a point; this implies that no two lines are parallel. Nonsimple arrangements are "degenerate" in some sense, and often theorems and algorithms are easiest with simple arrangements.

It is a remarkable fact that all simple arrangements on n lines have exactly the same number of vertices, edges, and faces.

[1] See especially Edelsbrunner (1987), to which my presentation is heavily indebted. A recent survey is by Halperin (1997).

Theorem 6.2.1. *In a simple arrangement of n lines, the number of vertices, edges, and faces is* $V = \binom{n}{2}$, $E = n^2$, *and* $F = \binom{n}{2} + n + 1$, *respectively, and no nonsimple arrangement exceeds these quantities.*

Proof. That the number of vertices is $\binom{n}{2}$ follows directly from the fact that in a simple arrangement, each pair of lines generates exactly one vertex. The formula for E can be proven by an easy induction. Assume any simple arrangement $\mathcal{A}$ of $n-1$ lines has $(n-1)^2$ edges. Insert a new line L into $\mathcal{A}$. It splits one edge on each of the $n-1$ lines of $\mathcal{A}$ in two, and L itself is partitioned by $\mathcal{A}$ into n new edges. Thus $E = (n-1)^2 + (n-1) + n$, which simple algebra reveals to be n^2.

We can now derive F from Euler's formula (Theorem 4.1.1): $V - E + F = 2$. We cannot apply this directly, as it counts these quantities for plane graphs, and an arrangement $\mathcal{A}$ is not a plane graph under the usual interpretation. There are at least two ways to proceed here: Convert $\mathcal{A}$ into a graph by joining the lines to a new vertex, or reexamine the proof of Euler's theorem. Here we choose the latter route. Recall that we proved Euler's theorem by puncturing a face of a polytope with a point in its interior and flattening to the plane. If we instead puncture at a vertex v, we lose one vertex so that the formula is now $V - E + F = 1$, and the flattening stretches all edges incident to v to extend to infinity. This flattening could be achieved, for example, by stereographic projection from v as the north pole of a surrounding sphere, with any other point p of the skeleton mapped to the spot where the line through v and p hits the plane supporting the south pole. The result is an unbounded object topologically equivalent to an arrangement. Thus $V - E + F = 1$ holds for arrangements. Now substituting in the known values of V and E yields $F = 1 + n^2 + n(n-1)/2 = (n^2 + n + 2)/2$, which is the same as the claimed formula.

Finally, we argue informally that simple arrangements are the worst case for these combinatorial quantities. If $k > 2$ lines meet at a vertex, we can "perturb" the lines slightly to break the coincidence, as shown in Figure 6.2(a). This increases each of V, E, and F, as is evident from examining just the contributions of the shaded region in (a) of the figure. If two lines are parallel, then again perturbation, as in Figure 6.2(b), increases V by 1, E by 2, and F by 1. Thus breaking degeneracies only increases the combinatorial complexity of an arrangement, and so a nonsimple arrangement cannot be a worst case.

What hasn't been demonstrated, but what should accord with intuition, is that all the degeneracies in an arrangement can be broken simultaneously. Establishing this formally would take us too far afield.[2] □

The important consequence of this theorem for algorithm design is that arrangements in the plane are fundamentally quadratic: V, E, and F are all $\Theta(n^2)$.

The key combinatorial property of arrangements that permits efficient construction is that no one line of an arrangement pierces cells with too many edges. The reason this is key will be made clear after we make this notion of a "zone" precise. Following that we will prove the "Zone Theorem."

[2] See Edelsbrunner & Mücke (1990).

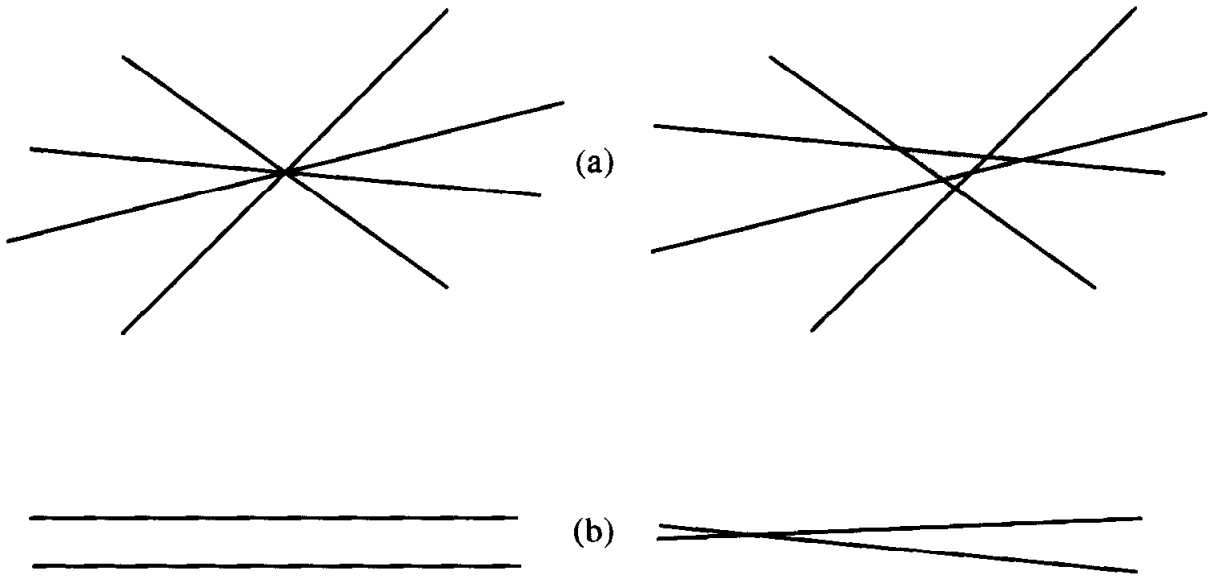


FIGURE 6.2 Perturbing the lines in a nonsimple arrangement only increases the number of vertices, edges, and faces: (a) $k > 3$ lines through a point; (b) parallel lines.

6.2.1. Zone Theorem

Fix an arrangement $\mathcal{A}$ of n lines, and let L be any other line (usually not in $\mathcal{A}$). We assume for clarity that the arrangement $\mathcal{A} \cup \{L\}$ is simple. The *zone* of L in $\mathcal{A}$, $Z_{\mathcal{A}}(L)$ (or just $Z(L)$ when the arrangement is clear from the context), is the set of cells (faces) intersected by L. For example, in Figure 6.3, $Z(h) = \{A, B, C, D, E, F\}$. The Zone Theorem bounds the total number of edges of these cells. Let $|C|$ be the number of edges bounding a cell/face C. In that figure, $|A| = 2$, $|B| = 4$, $|C| = 3$, $|D| = 4$, $|E| = 2$, and $|F| = 4$. The total number of edges of the cells in the zone $Z(L)$ we denote by $z(L)$; thus $z(h) = 19$ in Figure 6.3. Note that edges adjacent to two cells in the zone are counted twice in $z(L)$.[3] Lastly we let z_n be the maximum value of $z(L)$ over all possible lines L in all arrangements of n lines: The largest $z(L)$ could ever be as a function of n.

To look ahead to Section 6.3 quickly, we will construct an arrangement of lines incrementally, by inserting each line one after another into a growing arrangement. The complexity of this insertion will be bound above by z_n, as the edges of the zone of the inserted line will be traversed by the algorithm.

We now focus on the Zone Theorem, which claims that $z_n = O(n)$. This was first proved by Chazelle, Guibas & Lee (1985) and Edelsbrunner, O'Rourke & Seidel (1986)[4] and since then many alternative proofs have been found. Here I expand on a proof of Edelsbrunner et al. (1993). My proof is a bit long-winded, so the reader should take a deep breath.

Theorem 6.2.2. *The total number of edges in all the cells that intersect one line in an arrangement of n lines is $O(n)$: Specifically, $z_n \leq 6n$.*

[3] In other words, we are counting the half-edges in the corresponding twin-edge data structure (Section 4.4).

[4] Unfortunately the proof for dimensions ≥ 3 in this paper (and in Edelsbrunner (1987)) is incorrect, although the theorem is true. A correct proof appears in Edelsbrunner, Seidel & Sharir (1993).

Proof. We will make three assumptions to simplify the exposition: The arrangement with the new line is simple, the line h whose zone we seek is horizontal, and no line is vertical. I will not take the time to justify these assumptions, since the proof is difficult enough without dealing with "special" cases. Suffice it to say that the worst case is again achieved by simple arrangements, so it is no loss of generality to assume this for an upper bound.

Because no line is vertical, it makes sense to partition the lines of each cell of $Z(h)$ into left-bounding and right-bounding edges; we will simplify these to "left" and "right" edges. Points on a *left edge* of a cell C have interior cell points immediately to their right; thus they form the left boundary of C. Right edges are those that are not left edges. Note that, by our assumption of simplicity, no line is parallel to h, and therefore the highest and lowest vertices of bounded cells C are unique, providing a clean separation between left and right edges. In Figure 6.3, the left edges of the zone cells are highlighted with dotted lines.

Since left and right edges play a symmetrical role, we need only prove that the number of left edges contributing to z_n, call this l_n, is $\leq 3n$. In Figure 6.3, there are 9 left edges, and $n = 5$.

The proof is by induction. The basis of the induction is the obvious $l_0 \leq 0$: An empty arrangement has no left edges. Suppose it is true that $l_{n-1} \leq 3n - 3$. Let $\mathcal{A}$ be an arrangement of n lines satisfying our assumptions. The plan is to remove one line from $\mathcal{A}$, apply induction, and put it back. The line we choose to remove is the one whose intersection with h is rightmost: L_5 in Figure 6.3. (Note that, by the assumption of simplicity, no two lines are parallel, and thus every line intersects h.) Call this rightmost line r. Let $\mathcal{A}'$ be the arrangement $\mathcal{A} \setminus \{r\}$: $\mathcal{A}$ with r removed. It has $n - 1$ lines, and so the induction hypothesis holds. Now our goal is to show that inserting r back into $\mathcal{A}'$ can increase l_{n-1} by at most 3. The remainder of the proof establishes this, by showing that r introduces one new left edge and splits at most two old left edges. Here "old" refers to $\mathcal{A}'$, before reinsertion of r, and "new" refers to $\mathcal{A}$, after insertion of r.

Figure 6.4 shows $\mathcal{A}'$ corresponding to $\mathcal{A}$ in Figure 6.3. We label all the cells with primes, using the same letter for obvious correspondents. Inserting $r = L_5$ splits cell G' of $\mathcal{A}'$ into cells F and E in $\mathcal{A}$, and it clips cells A', B', C', and D' to form A, B, C, and D respectively. The total effect of this insertion is complicated: For example, the number of left edges of B' and B are the same, C has one less left edge than C', and F has one more left edge than G'. What makes the situation simpler than it might first appear is that (a) we only need an upper bound on the increase, not an exact accounting, and (b) the effect on the left edges is simpler than the changes to the right edges. This latter claim results from our choice of the rightmost line to obtain a bound on the left edges, as we will see.

Because r was chosen to have the rightmost intersection with h in $\mathcal{A}$, this intersection (call it $x = r \cap h$) must lie in the rightmost cell intersected by h in $\mathcal{A}'$ (G' in Figure 6.4). The rightmost line r will bound the rightmost cell of $\mathcal{A}$ (F in Figure 6.3) from the left, for r contains x and the ray from x to the right must be in the rightmost cell. So r will contain at least one new left edge. Now the key observation is that r does not contain any other left edges in $\mathcal{A}$ (note that it does contain several new right edges in Figure 6.3), for any line α of $\mathcal{A}$ (such as L_3 in Figure 6.3) that contains more than one left edge must be cut by a line β (such as L_4) that separates the cells α supports to the right; but then

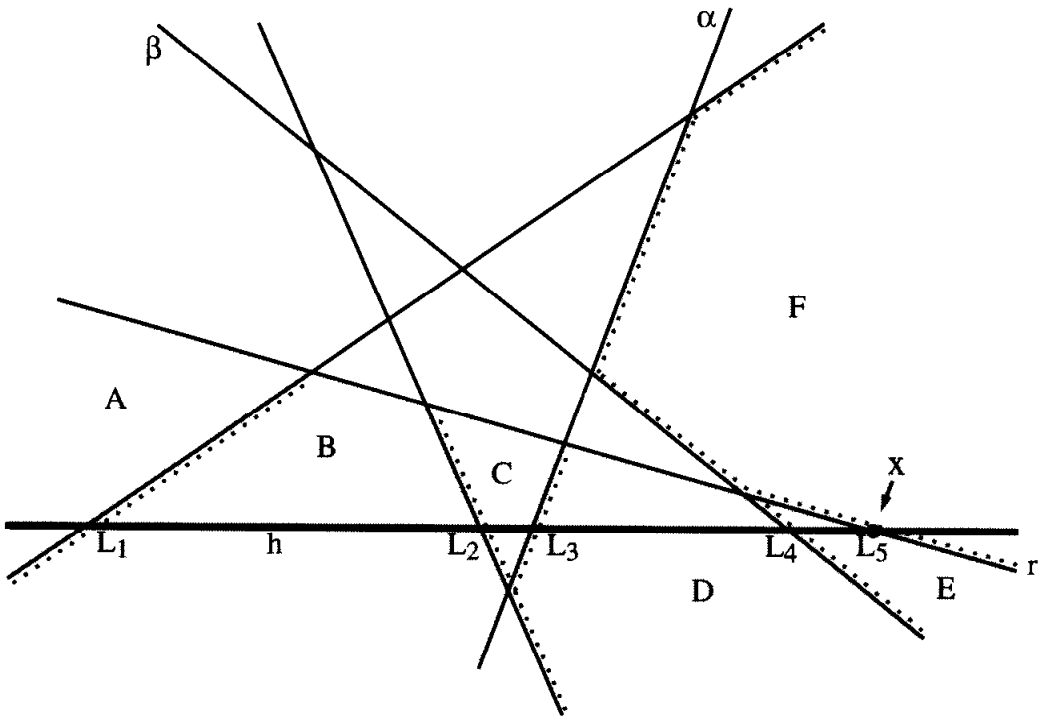


FIGURE 6.3 The zone of h is $Z(h) = \{A, B, C, D, E, F\}$; $z(h) = 2+4+3+4+2+4 = 19$. The lines of $\mathcal{A}$ are numbered $L_1, \ldots, L_5$.

β would intersect h to the right of α. Thus α could not have the rightmost intersection with h. This explains the choice of r.

Having concluded that r contains exactly one new left edge, we need only limit the number of old left edges that r splits in two. For example, the left edge of G' in Figure 6.4 that crosses h (contained in L_4) is split by $r = L_5$ in Figure 6.3 into a left edge for E and one for F. This splitting can only happen in the rightmost cell on h in $\mathcal{A}'$, for r "clips" rather than splits all other cells that it intersects. The reason is similar to that just used above: If r splits a left edge, then the two cells supported to the right by these left edges must straddle r on h, implying that one is rightmost (since r has the rightmost intersection with h); this in turn implies that the old edge split must have been part of the rightmost cell.

So we have established that r can split only edges of the rightmost zone cell. Because this cell is convex, r can cross it at most twice (r only intersects the boundary of $G' = E \cup F$ once in Figure 6.3). Therefore r can split at most two old left edges.

We now have our theorem: r adds one new left edge and can split at most two old left edges, increasing l_{n-1} by at most 3, to $l_n \leq 3n$. □

6.2.2. Exercises

1. *Biggest zone* [difficult]. Construct a generic example that achieves the largest value of z_n that you can manage. Theorem 6.2.2 guarantees that $z_n \leq 6n$, but this is not in fact achievable (Bern, Eppstein, Plassman & Yao 1991).
2. *Space partitions.* Derive formulas for the number of vertices, edges, faces, and cells of a simple arrangement of n planes in three-dimensional space.

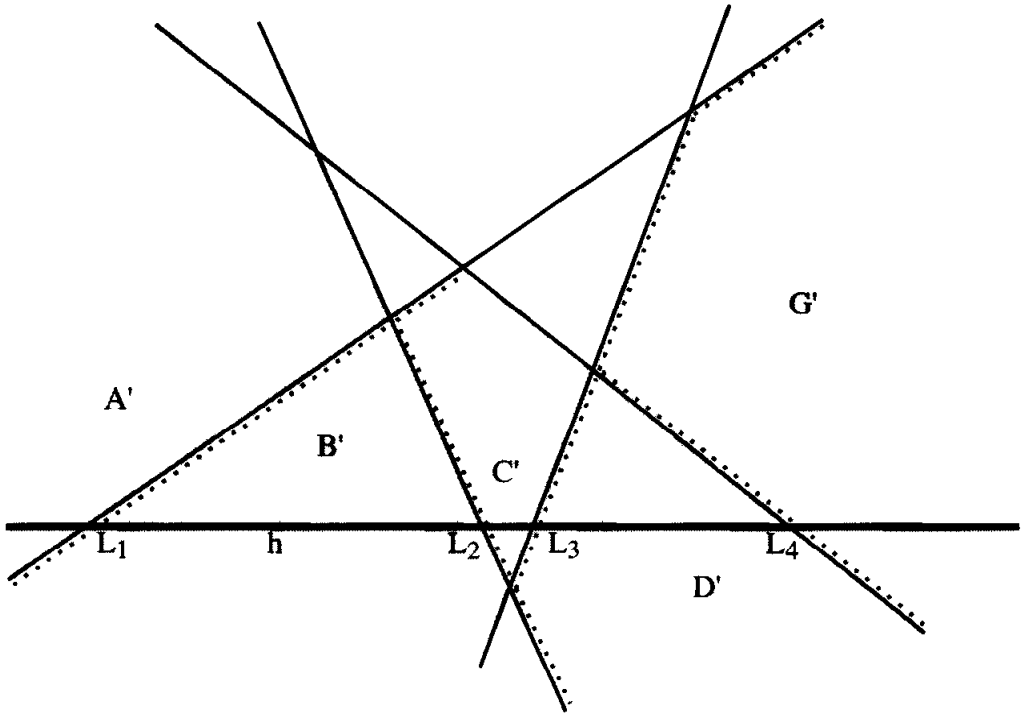


FIGURE 6.4 The arrangement $\mathcal{A}' = \mathcal{A} \setminus \{r\}$.

6.3. INCREMENTAL ALGORITHM

We now have in place the machinery to discuss an algorithm for constructing an arrangement of lines. First we must decide on the input and output. Input is easy: Any representation of the lines, such as slope and intercept, will do. Output is less clear. But I hope that after our discussion of data structures for polytope surfaces (Section 4.4) it should be evident that any of those could be used to represent an arrangement, with slight modification to account for unbounded edges. In particular, the quad-edge data structure can be used as is, since it can represent any subdivision. And the twin-edge structure's emphasis on half-edges melds well with the combinatorics of arrangements. We will not explore representation issues but just assume that the representation permits easy traversal of the edges bounding a face and movement between adjacent faces.

The incremental algorithm for constructing an arrangement (Algorithm 6.1) is pleasingly simple. At any given stage, we have an arrangement $\mathcal{A}_{i-1}$ constructed for the first

Algorithm: ARRANGEMENT CONSTRUCTION
Construct $\mathcal{A}_0$, a data structure for an empty arrangement.
for each $i = 1, \ldots, n$ do
    Insert line L_i into $\mathcal{A}_{i-1}$ as follows:
        Find an intersection point x between L_i
            and some line of $\mathcal{A}_{i-1}$.
        Walk forward from x along cells in $Z(L_i)$.
        Walk backward from x along cells in $Z(L_i)$.
    Update $\mathcal{A}_{i-1}$ to $\mathcal{A}_i$.

Algorithm 6.1 Incremental construction of an arrangement.

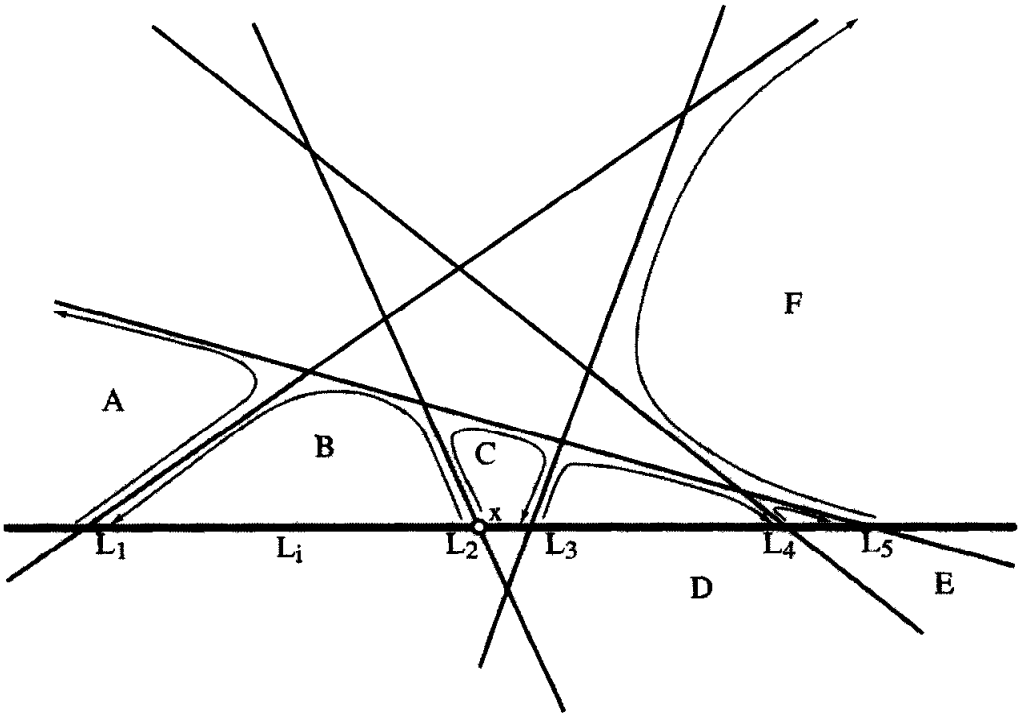


FIGURE 6.5 Inserting one line L_i into an arrangement. The curves show the path of zone traversal for discovering the vertices on L_i.

$i - 1$ lines. The task is to find all the points of intersection between $\mathcal{A}_{i-1}$ and L_i, the ith input line. First an intersection point x between L_i and any line of $\mathcal{A}_{i-1}$ is found in constant time. In Figure 6.5, $x = L_i \cap L_2$. Then we walk forward along the zone of L_i, $Z(L_i)$, traversing the edges of each face clockwise, repeating each cell traversal until an edge is again encountered that crosses L_i. So in Figure 6.5, we traverse three edges of C before meeting the intersection between L_3 and L_i; then we traverse three edges of D; and so on as illustrated. The forward march terminates when an infinite zone edge is encountered; then the process is repeated from x backwards, traversing cell edge counterclockwise (cells B and A in the figure). Each of the steps in this traversal moves between incident or adjacent objects, and so each takes constant time. The total cost of the insertion traversal is dependent on the complexity of the zone, which as we saw in Theorem 6.2.2, is $O(n)$. Note how the structure of the arrangement is used to avoid sorting. Only after all the points of intersection with L_i have been found or (more likely) during the traversal itself does the data structure for $\mathcal{A}_{i-1}$ get updated to that for $\mathcal{A}_i$. That this can be accomplished in $O(n)$ time we leave for Exercise 6.4.1[1].

It is clear then that the entire construction requires $O(n^2)$ time, a result first obtained by Chazelle et al. (1985) and Edelsbrunner et al. (1986):

Theorem 6.3.1. *An arrangement of n lines in the plane may be constructed in $\Theta(n^2)$ time and space.*

Proof. The algorithm takes $O(n^2)$ time, and as we saw in Theorem 6.2.1, the structure may be this big, so this is the best possible asymptotic bound. Storing the structure could require quadratic space in the worst case. □

6.4. THREE AND HIGHER DIMENSIONS

One of the most beautiful aspects of the theory of arrangements is that almost every feature carries through smoothly to higher dimensions. Although we will not discuss this topic in any detail, it is worth mentioning analogs of Theorems 6.2.1, 6.2.2, and 6.3.1:[5]

Theorem 6.4.1. *The number of faces of any dimension in an arrangement of hyperplanes in d dimensions is $O(n^d)$, the zone of any hyperplane has total complexity $O(n^{d-1})$, and such an arrangement can be constructed in $O(n^d)$ time and space.*

In particular, an arrangement of planes in three dimensions has complexity $O(n^3)$ and can be constructed in this time, a fact we will use in Sections 6.6 and 6.7.

6.4.1. Exercises

1. *Insertion updates.* Argue that if the arrangement is represented by the twin-edge data structure, the updates caused by insertion of one new line can be effected in $O(n)$ time.
2. *Pencil of lines, planes.*
   a. How many vertices, edges, and faces are in an arrangement formed by a *pencil* of n lines, lines all through a common point?
   b. How many vertices, edges, faces, and 3-cells are in an arrangement formed by n planes all sharing a common point?

6.5. DUALITY

It may seem odd that arrangements are so useful for problems on sets of points in the plane, as in item (4) of Section 6.1. The key to this and many applications of arrangements is an important concept known as *duality*. The basic idea is that because lines may be specified by two numbers, lines can be associated with the *point* whose coordinates are those two numbers. For example, a line specified by $y = mx + b$ can be associated with the point (m, b). Often the space of these points is called *parameter space*, as the point coordinates are the parameters of the line. Because both the primary and the parameter space are equivalent two-dimensional spaces, it is customary (albeit confusing) to treat them as a single space whose coordinates have two interpretations. Once the mapping from lines to points is determined, it can be reversed: Any point in the plane can be viewed as specifying a line when its coordinates are interpreted as, for example, slope and intercept. Together these mappings determine a duality between points and lines: Every[6] line is associated with a unique point, and every point with a unique line.

There are many different point–line duality mappings possible, depending on the conventions of the standard representation of a line. Each mapping has its advantages and disadvantages in particular contexts. We mentioned already the mapping $L : y = mx + b \Leftrightarrow p : (m, b)$, which has the advantage of tapping into our familiarity with slope and intercept. The mapping $L : ax + by = 1 \Leftrightarrow p : (a, b)$ defines what is known as *polar*

[5]For proofs see Edelsbrunner (1987) and Edelsbrunner et al. (1993).

[6]This "every" will be qualified in Lemma 6.5.2.

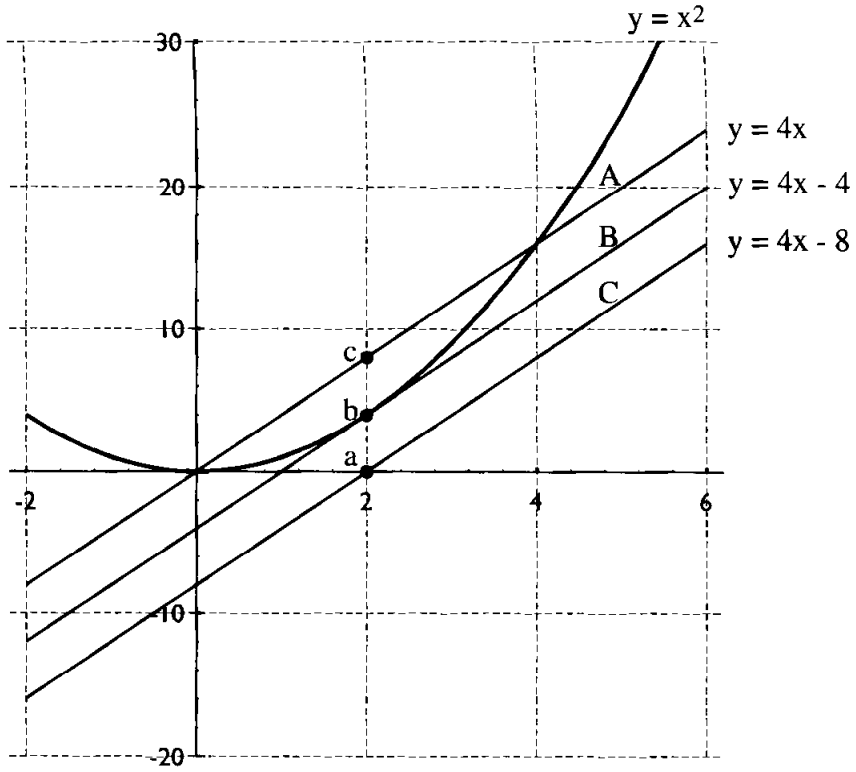


FIGURE 6.6 $\mathbb{D}(a) = A$, $\mathbb{D}(b) = B$, $\mathbb{D}(c) = C$.

duality (Coxeter & Greitzer 1967).[7] This mapping has pleasing geometric properties, some of which are explored in exercises (Exercise 6.5.3[3] and [4]). But the mapping we will use throughout this chapter is

$$L : y = 2ax - b \Leftrightarrow p : (a, b). \tag{6.1}$$

We use the symbol $\mathbb{D}$ to indicate this mapping: $\mathbb{D}(L) = p$ and $\mathbb{D}(p) = L$. Although this may seem like an odd choice for a mapping, it is often the most convenient in computational geometry, largely because of its intimate connection to the paraboloid transformation (Section 5.7.2). We now examine this connection, first informally, and then via a series of lemmas (Section 6.5.2).

6.5.1. Duality Mapping

The relationship between the point $p = (a, b)$ and the line $L : y = 2ax - b$ is not immediately evident. However, the similarity of L to Equation (5.12) in Section 5.8.1 should indicate a relationship to the parabola $y = x^2$. Recall that $y = 2ax - a^2$ is the tangent to this parabola at the point (a, a^2). Thus $\mathbb{D}(p)$ for $p = (a, b)$ with $b = a^2$ maps to this tangent. If $b < a^2$, then $\mathbb{D}(p)$ maps to a line parallel to this tangent but raised vertically by $(a^2 - b)$ (as we saw in Figure 5.7.1). If $b > a^2$, then $\mathbb{D}(p)$ maps to a parallel line shifted $(b - a^2)$ below the tangent. This is illustrated in Figure 6.6 for three points with $a = 2$ and $b \in \{0, 4, 8\}$. Here and throughout we display the points and their duals in the same space.

[7]The mapping $L : ax + by = -1 \Leftrightarrow p : (a, b)$ is often given the same name (Chazelle et al. 1985).

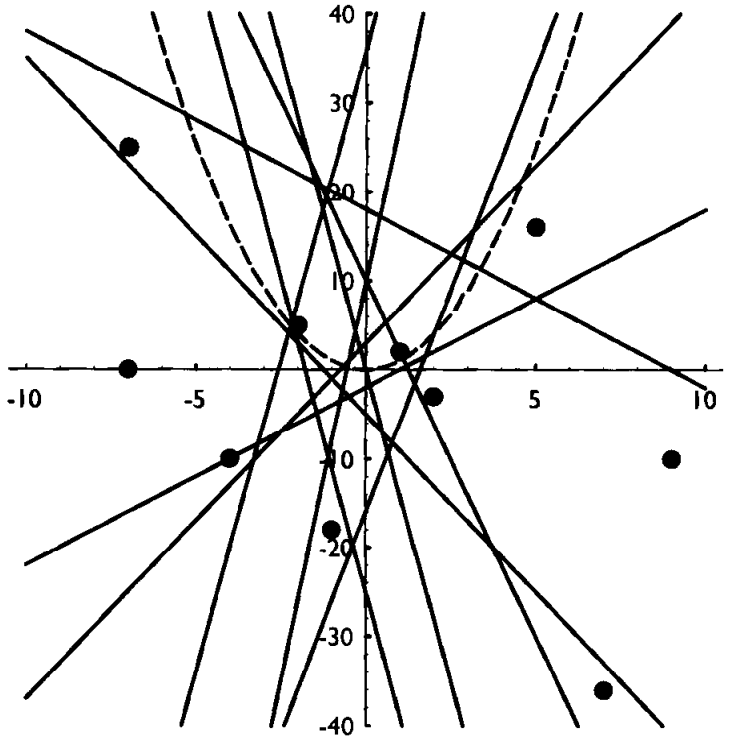


FIGURE 6.7 The duals of the lines in Figure 6.1.

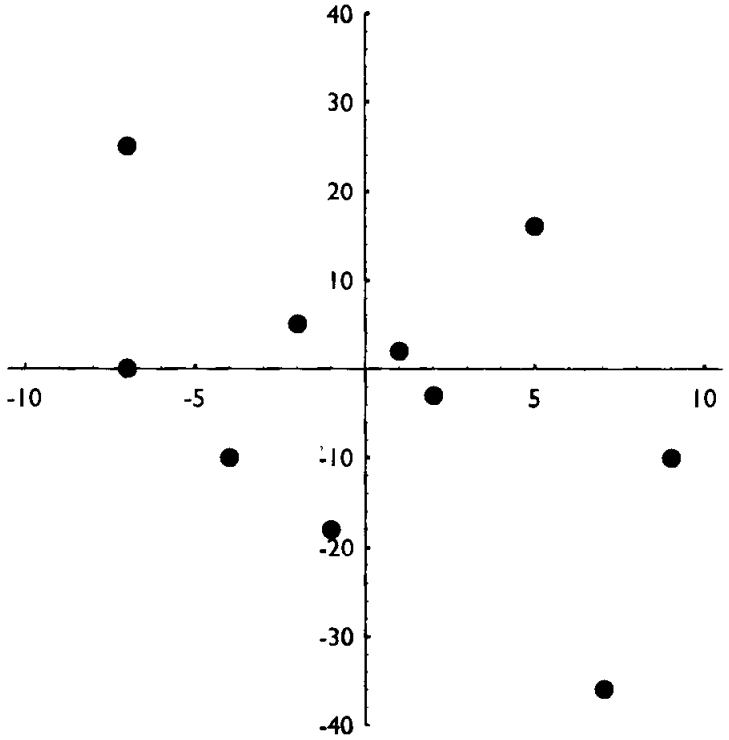


FIGURE 6.8 The points from Figure 6.7: $\{(-7, 25)\ (-7, 0), (-4, -10), (-2, 5), (-1, -18), (1, 2), (2, -3), (5, 16), (7, -36), (9, -10)\}$.

With this duality transformation, we can convert any set of points into an arrangement of lines and vice versa. One reason this is often so useful is that the relationships between points are revealed more explicitly in the dual arrangement of lines.[8]

[8]This observation is due to Edelsbrunner (1987, p. 4).

Figure 6.7 shows the construction of the points dual to the ten lines shown in Figure 6.1, and Figure 6.8 displays the points alone. This example will be employed later, in Section 6.7.6.

6.5.2. Duality Properties

In this section we develop some basic properties of the duality transform, which will then be employed in later sections.

Lemma 6.5.1. $\mathbb{D}$ *is its own inverse:* $\mathbb{D}(\mathbb{D}(x)) = x$, *where* x *is either a point or a line.*

Proof. The mapping was defined to be symmetric. □

Lemma 6.5.2. $\mathbb{D}$ *is a one-to-one correspondence between all nonvertical lines and all points in the plane.*

Proof. Vertical lines cannot be represented in the form $y = 2ax - b$, and these are the only lines that cannot be so represented. □

The special cases involving vertical lines can be skirted in any given problem by rotating the lines slightly so that none is vertical. We will simply exclude vertical lines from consideration.

Duality preserves point-line incidence:

Lemma 6.5.3. *Point* p *lies on line* L *iff point* $\mathbb{D}(L)$ *lies on line* $\mathbb{D}(p)$.

Proof. Let L be the line $y = 2ax - b$, and let $p = (c, d)$. Then since p lies on L, $d = 2ac - b$. $\mathbb{D}(L)$ is (a, b), and $\mathbb{D}(p)$ is the line $y = 2cx - d$. Substituting the coordinates of $\mathbb{D}(L)$ into $\mathbb{D}(p)$'s equation results in $b = 2ca - d$, which holds since this is just a rearrangement of $d = 2ac - b$. Therefore $\mathbb{D}(L)$ lies on $\mathbb{D}(p)$.

The reverse implication follows from Lemma 6.5.1. □

The fact that two points determine a line dualizes to two lines determining a point of intersection:

Lemma 6.5.4. *Lines* L_1 *and* L_2 *intersect at point* p *iff the line* $\mathbb{D}(p)$ *passes through the two points* $\mathbb{D}(L_1)$ *and* $\mathbb{D}(L_2)$.

Proof. This follows by applying Lemma 6.5.3 twice: Since p lies on L_1 and on L_2, both $\mathbb{D}(L_1)$ and $\mathbb{D}(L_2)$ lie on $\mathbb{D}(p)$. Again the reverse implication follows from Lemma 6.5.1. □

When vertical lines are excluded from consideration, points and lines can be related unambiguously as above, on, or below. The duality mapping can be seen to reverse vertical ordering, in the following sense:

Lemma 6.5.5. *If point p lies above line L, then line $\mathbb{D}(p)$ lies below point $\mathbb{D}(L)$; and symmetrically if p lies below L, $\mathbb{D}(p)$ lies above $\mathbb{D}(L)$.*

Proof. We only prove the first claim. So assume p lies above L. Let L be the line $y = 2ax - b$, and let $p = (c, d)$. Because p lies above L, the y coordinate of p is larger than L evaluated at $x = c$: $d > 2ac - b$. $\mathbb{D}(p)$ is the line $y = 2cx - d$, and $\mathbb{D}(L) = (a, b)$. Substituting $x = a$ into $\mathbb{D}(p)$ yields a y coordinate of $2ca - d$, which is smaller than b because $b > 2ca - d$ is just a rearrangement of $d > 2ac - b$. Thus line $\mathbb{D}(p)$ lies below point $\mathbb{D}(L)$. □

This can be seen clearly in Figure 6.6. For example, point c is above line B, and line C is below point b.

6.5.3. Exercises

1. *Collinear points* [easy]. What is the dual $\mathbb{D}$ of k collinear points?
2. *Dual of regular polygons.* Find the dual $\mathbb{D}$ of the vertices, and of the lines containing the edges, of a regular polygon centered on the origin, oriented so that no edge is vertical. *Hint:* Analyze what happens when the number of vertices $n \to \infty$ by studying the unit origin-centered circle.
3. *Polar dual properties.* Recall that the polar dual is defined by the mapping $L : ax + by = 1 \Leftrightarrow p : (a, b)$.
   a. Relate polar dual points and lines geometrically to the unit circle centered on the origin.
   b. Prove that the polar dual of a line that intersects this unit circle at points a and b is the point p that is the intersection of the tangents to the circle at a and b.
4. *Polar dual of regular polygons.* Redo Exercise [2] above under polar duality: Find the polar dual of the vertices, and of the lines containing the edges, of a regular polygon centered on the origin.
5. *Intersection of halfplanes.*
   a. Let H be a set of n halfplanes, each of which contains a portion of the negative y axis (i.e., they are all facing downwards). Let $Q = \cap H$, the intersection of all the halfplanes. Let S be $\mathbb{D}(H)$: the set of points dual to the lines bounding the halfplanes. Finally, let $P = \mathcal{H}(S)$, the convex hull of S. Explain the relationship between the structures of P and of Q.
   b. Suggest an algorithm for computing the intersection of halfplanes based on your observations.

6.6. HIGHER-ORDER VORONOI DIAGRAMS

In this section we will explore the intimate connection between arrangements and Voronoi diagrams, a connection foreshadowed in Section 5.8. We will detail the relationship only for one-dimensional Voronoi diagrams, leaving the more interesting two-dimensional case largely to analogy. The focus of the connection is on objects called "higher-order Voronoi diagrams," which we will explain after developing the requisite machinery.

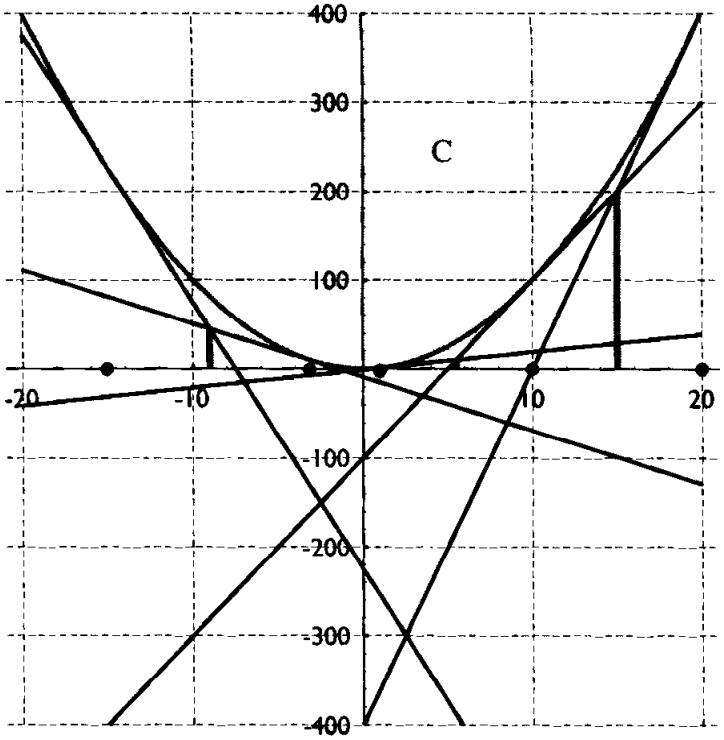


FIGURE 6.9 The parabola arrangement for $P = \{-15, -3, 1, 10, 20\}$. Cell C vertically projects to the Voronoi diagram of P, $\{-9, -1, 5\frac{1}{2}, 15\}$.

6.6.1. One-Dimensional Diagrams

Recall from Sections 5.8.1 that a set of points $P = \{x_1, \ldots, x_n\}$ on the x axis are mapped to a set of lines tangent to the parabola $y = x^2$. The points of tangency are (x_i, x_i^2), directly above x_i. The equations of the tangent lines are $T_i : y = 2x_i x - x_i^2$ (Equation 5.12). Note that this tangent is precisely $\mathbb{D}((x_i, x_i^2))$. Let us choose the indices of the points so that they are sorted: $x_i < x_{i+1}$. We showed that the x coordinate of the intersection point between two adjacent tangents is the midpoint between their generating points: The tangents for x_i and x_{i+1} intersect at $\frac{1}{2}(x_i + x_{i+1})$ (Equation 5.13). These intersections vertically project, therefore, to the one-dimensional Voronoi diagram of P, the set of midpoints for P.

Now we consider the entire arrangement of lines formed by the n parabola tangents, as illustrated in Figure 6.9 for $P = \{-15, -3, 1, 10, 20\}$. Note that the parabola is entirely contained within one cell C of this arrangement, and it is the projection of the boundary of this cell that gives the Voronoi diagram of P: at $x = \{-9, -1, 5\frac{1}{2}, 15\}$ in Figure 6.9. It will be useful to view this in another manner, as follows. Imagine dropping down the vertical line $x = b$. The first edge of C encountered maps to the Voronoi cell (a segment on the x axis) in which b lies.

We give yet another interpretation of this observation, already implicit in Section 5.7.1, before introducing the new connections. Let T be a line tangent to the parabola above $x = a$, so T is $y = 2ax - a^2$. We claim that the vertical distance d between the parabola and T above $x = b$ is the square of the distance between a and b. See Figure 6.10. This can be verified by a simple calculation: $d = b^2 - (2ab - a^2) = (b - a)^2$. The relation between this observation and the preceding one should now be clear: If, when dropping down $x = b$, T_i is encountered prior to T_j, then T_i is closer to the parabola above b than is T_j, and therefore b is closer to x_i than it is to x_j.

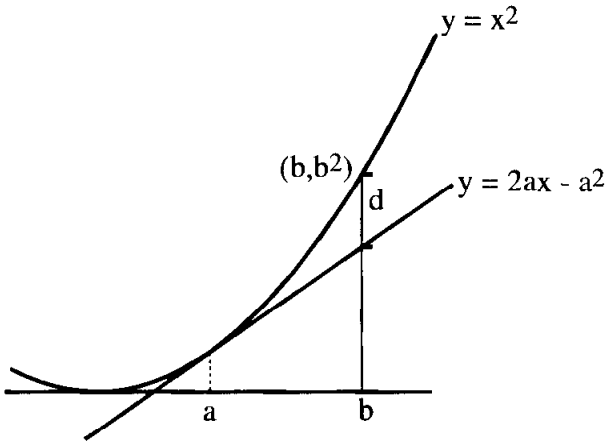


FIGURE 6.10 $d = (b-a)^2$. Here $(b-a) < 1$ so that $d < (b-a)$.

From this discussion we can conclude the following generalization:

Lemma 6.6.1. *The order in which the tangents are encountered moving down the vertical $x = b$ is the same as the order of closeness of b to the x_i's that generate the tangents.*

In other words, vertical sorting of the tangents corresponds to nearest-neighbor sorting.

Finally we come to the punchline. Define the *2nd-order Voronoi diagram* to partition the relevant space (in the case we are discussing, the x axis) into regions that have the same first two nearest neighbors. Which of these neighbors is first and which second is irrelevant for this definition. Thus if a's closest neighbor is x_i and its second closest is x_j, it is in the same 2nd-order Voronoi region as a point b whose nearest neighbor is x_j and second closest is x_i. The 2nd-order diagram is implicit in those edges of the parabola arrangement composed of points that have exactly one line strictly above them vertically (and therefore two lines at or above them, since each edge is on a line). These edges comprise what is known as the *2-level* of the arrangement. The 2-level for the arrangement from Figure 6.9 is highlighted with dashes in Figure 6.11. The projection of the vertices of the 2-level partition the x axis into cells whose points have the same first two nearest neighbors in the same order. Thus in Figure 6.11, all $x > 15$ have (20, 10) as their two nearest neighbors; all $10\frac{1}{2} < x < 15$ have (10, 20) as nearest neighbors; all $5\frac{1}{2} < x < 10\frac{1}{2}$ have (10, 1) as nearest neighbors; and so on. This partition of the line induced by the projection of the 2-level is finer than the 2nd-order Voronoi diagram, since in that diagram the order of the neighbors does not matter. So in Figure 6.11, all points $x > 10\frac{1}{2}$ have {10, 20} as their set of two nearest neighbors. We now argue that the transition points for the 2nd-order Voronoi diagram are the projections of the points of intersection between the 2-level and the 3-level of the arrangement.

Define the k-level of an arrangement as the set of edges whose points have exactly $k-1$ lines strictly above them, together with the endpoints of these edges. (Recall that arrangement edges are open segments.) We do not demand any certain number of lines above the vertices, as they might not have $k-1$. The 3-level is highlighted with dots in Figure 6.11. Let a be the projection of a vertex at the intersection between the 2-level and the 3-level. (These three vertices are circled in the figure.) Let the first three tangents met by the

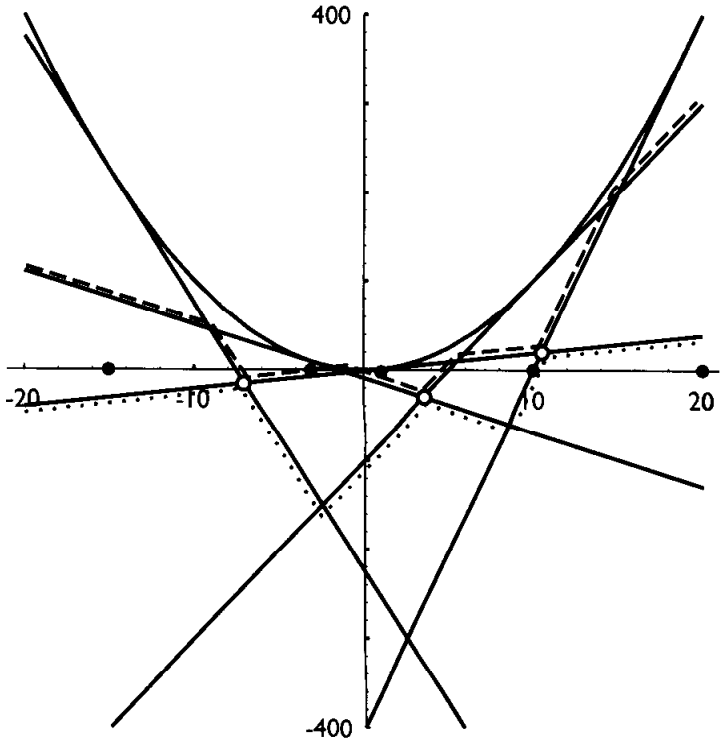


FIGURE 6.11 Dashed 2-level; dotted 3-level. Open circles indicate points of intersection between these levels. The projection of these points, $x = \{-7, 3\frac{1}{2}, 10\frac{1}{2}\}$, forms the 2nd-order Voronoi diagram.

vertical line $x = a+\epsilon$ be (A, B, C) from top to bottom, where $\epsilon > 0$ is small. B is on the 2-level and C on the 3-level at this x value. Then just to the other side of a, the line $x = a-\epsilon$ meets those tangents in the order (A, C, B), for here C is on the 2-level and B on the 3-level, with B and C intersecting at $x = a$. Therefore $x = a$ represents a change in first-two nearest neighbors from $\{A, B\}$ to $\{A, C\}$. This shows that the vertices common to the 2-level and 3-level do indeed represent 2nd-order Voronoi region transitions. It is equally clear that the other vertices of the 2-level (those not also on the 3-level) represent a switching of the order of the two nearest neighbors, without changing the set of these neighbors.

What we just argued informally holds for arbitrary k:

Theorem 6.6.2. *The points of intersection of the k- and $(k+1)$-levels in the parabola arrangement project to the kth-order Voronoi diagram* (Edelsbrunner 1987, p. 317).

Note that this theorem even "works" for $k = 1$: The points of intersection between the 1-level and the 2-level are precisely the vertices of the 1-level, which are the vertices of the parabola-containing cell, which project to the ordinary Voronoi diagram, which can be viewed as the 1st-order Voronoi diagram.

6.6.2. Two-Dimensional Diagrams

We will not derive any results in two dimensions, but as the reader should expect, all definitions and results from one dimension generalize exactly as one might hope. Given a set of points in the plane, construct an arrangement of planes tangent to the paraboloid above the points, as in Section 5.8.2. The Voronoi diagram is the projection

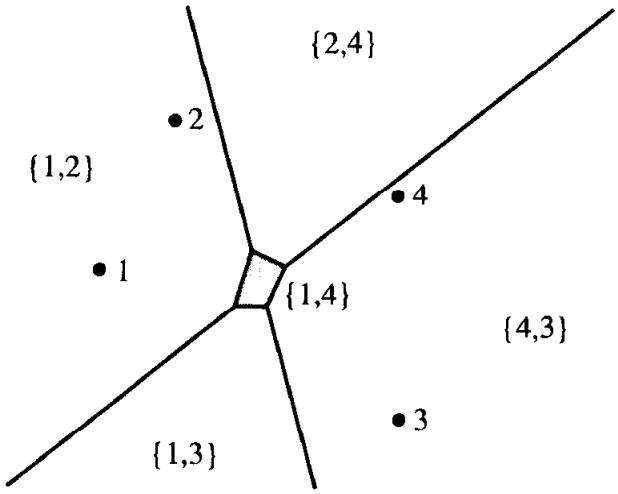


FIGURE 6.12 A 2nd-order Voronoi diagram for four points. The central shaded region's nearest neighbors are $\{1, 4\}$.

of the the 1-level, the edges and vertices of the cell containing the paraboloid. The k-level is an undulating "sheet" of faces (and the edges and vertices in their closures). The kth-order Voronoi diagram is the projection of the intersection of the k- and $(k+1)$-levels, which is a collection of edges and vertices. A simple 2nd-order Voronoi diagram is shown in Figure 6.12. Thus all the higher-order Voronoi diagrams are in a precise sense embedded in the arrangement of tangent planes.

This incidentally shows that the total complexity of all these diagrams is $O(n^3)$, since the levels are all embedded in an arrangement with complexity of $O(n^3)$ (by Theorem 6.4.1), and no face is shared between levels. And it is not difficult to construct all the kth-order Voronoi diagrams, for $k = 1, \ldots, n-1$, in time $O(n^3)$, by constructing the arrangement of planes.

6.6.3. Exercises

1. *Furthest-point Voronoi diagram.* Show that the furthest-point Voronoi diagram (Figure 5.19) is the same as the $(n-1)$st-order Voronoi diagram.
2. *kth-order Voronoi diagram in dimension* 1. How many regions are there in a kth-order Voronoi diagram in one dimension?
3. *Cells are convex.* Prove that the cells of a kth-order Voronoi diagram are convex.
4. *Bisector bounding more than one cell.* Demonstrate by example that a bisector of two points might bound two nonadjacent cells of a kth-order Voronoi diagram.
5. *k-levels.* Prove that the k-level in a simple arrangement of lines is a polygonal chain that separates the plane into two parts.

6.7. APPLICATIONS

6.7.1. k-Nearest Neighbors

In the same way that the Voronoi diagram can be used to find the nearest neighbor of a query point (Sections 5.1 and 5.5.1), the kth-order Voronoi diagram can be used to find the k-nearest neighbors of a query point. This is used for what is called the "k-nearest

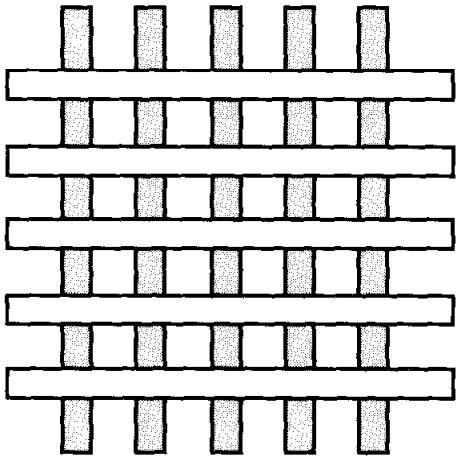


FIGURE 6.13 A grid of crossed rectangles establishes an $\Omega(n^2)$ lower bound on output complexity.

neighbors decision rule": classify an unknown into the class most heavily represented among its k nearest neighbors.[9] The k-nearest neighbors are also useful for facility location, information retrieval, and surface interpolation. See Okabe, et al. (1992) for further applications and references.

6.7.2. Hidden Surface Removal

Surely there is no geometric computation performed more frequently today than hidden surface removal, as it is the basis of all three-dimensional computer graphics, which is the basis of many television advertisements and movie special effects. The task is to take a set of flat, opaque, colored polygons in three-dimensional space and produce an image or "scene" of their appearance from a particular viewpoint. Often the polygons are linked into a surface, and the occluded portions of the surface are "hidden" and must be "removed" from the final scene.

Let n be the total number of vertices of the input polygons. One can see that the complexity of the output scene can be $\Omega(n^2)$: A grid of vertical rectangles obscuring horizontal rectangles leads to $> \frac{1}{16}n^2$ scene vertices as shown in Figure 6.13. If we demand a list of polygons as output (each square hole in the figure needs filling), no algorithm can beat quadratic time in the worst case. Many algorithms achieved $O(n^2 \log n)$, only $O(\log n)$ time slower than optimal, by including an $O(n \log n)$ sorting at some juncture (Sutherland, Sproull & Shumacker 1974); but an optimal algorithm remained elusive for years. The theory of arrangements finally led to a worst-case optimal $O(n^2)$ algorithm (due to McKenna (1987)), which I will now sketch.

First, assume the polygons do not interpenetrate in space: Their interiors are disjoint, although they may share boundary points. Second, assume the viewpoint is infinitely far from the polygons, so that all lines of sight are parallel, and we do not have to deal with the complications of perspective. Although not immediately obvious, any scene with a finite viewpoint can be transformed to one with the eye at infinity, so this is no loss of

[9]See Devijver & Kittler (1982) and Mizoguchi & Kakusho (1978).

generality.[10] Let the eye be at $(0, 0, +\infty)$, so the "viewplane" is the xy-plane, $z = 0$. It is convenient to add one large "background" polygon below all the others so that all lines of sight hit some polygon.

The first step is to project every edge of the input polygons to the xy-plane (by discarding the z coordinates of their endpoints). This is known as *orthographic projection* (in contrast to perspective projection). Next extend each edge to the line that contains it. The result is an arrangement $\mathcal{A}$ of n lines in the xy-plane, which can be constructed in $O(n^2)$ time by Theorem 6.3.1. Now the task is to decide, for each cell of $\mathcal{A}$, which polygon in space among those whose projection contains it, is highest, and therefore which is closest to the eye. Knowing this permits the cells to be "painted" appropriately according to the color of the polygon (and according to its orientation if shading is desired). Note that each cell has a unique foremost polygon.

A naive algorithm would require $O(n^3)$ time: For each of the $O(n^2)$ cells, compute the height for each of the $O(n)$ polygons. The challenge is to spend only constant time per cell.

McKenna's algorithm employs a topological sweep of the arrangement, a generalization of plane sweep (Section 2.4) introduced by Edelsbrunner & Guibas (1989). Rather than sweep a vertical line over the arrangement, we sweep a vertical "pseudoline" L, a curve that intersects each line of $\mathcal{A}$ exactly once, at which point it crosses it bottom to top. The advantage of making the sweep line "bendable" is that it is then unnecessary to spend $O(\log n)$ time in priority queue lookup to determine which vertex is the next to be swept. Rather, an unordered collection of "sweepable" vertices can be maintained: those incident to two edges adjacent among those crossed by L. In Figure 6.14, vertex v is sweepable because two edges of the cell C crossed by L are incident to it.

The data structures maintained by the algorithm, besides the fixed arrangement, include the list of *active* cells and edges crossed by L (such as the shaded cell in the figure), and for each active cell C, a list of all the polygons whose projections contain C in the xy-plane, sorted by z depth. Note that these lists are only maintained for active cells, of which there are always precisely $n + 1$ (since L crosses all n lines). Clearly these lists provide enough information to determine the foremost polygon for every cell of $\mathcal{A}$. I will not provide a detailed accounting of the algorithm actions as L sweeps over a vertex, but rather just mention one feature of the algorithm. As a vertex is swept, old cells "die" and new cells become active, but their lists of containing polygons are either the same or nearly the same. This "coherence" can be exploited to inherit enough information across a swept vertex to keep the updating cost to constant time per cell, amortized over all cells of $\mathcal{A}$. The result is a hidden surface algorithm that is $O(n^2)$ in the worst case.

This is not, however, the "best" hidden surface algorithm in practice, because it *always* takes $\Omega(n^2)$ time and space, whereas most realistic scenes have much smaller complexity. Since it is not uncommon for n to be as large as 10^6 for high-quality graphics, it is important to avoid quadratic time when possible. Algorithms whose performance is sensitive to the output scene complexity are called *output-size sensitive* hidden surface algorithms and are a topic of considerable current research (Dorward 1994).

[10] See, for example, Foley, van Dam, Feiner & Hughes (1990, Sec. 6.5.2).

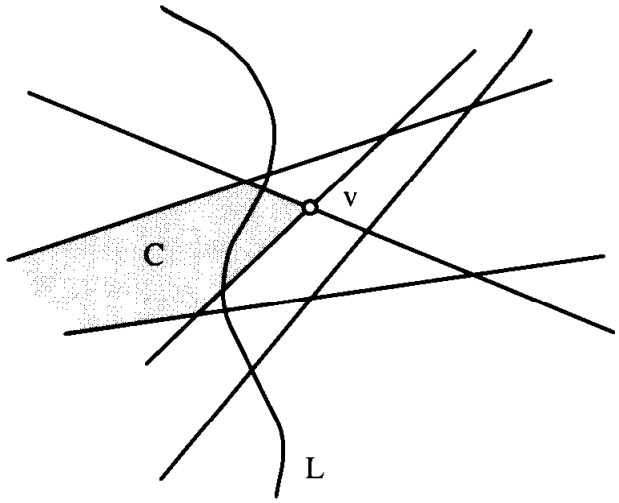


FIGURE 6.14 Vertex v is sweepable.

6.7.3. Aspect Graphs

In the late 1970s, researchers in computer vision introduced the notion of an "aspect graph" to aid image recognition (Koenderink & van Doorn 1976, 1979). The idea is to store all the "characteristic views" an object can present to a viewer, and compare these against what is actually seen. For a polyhedral object, a characteristic view is determined by combinatorial equivalence: Two viewpoints see the same *aspect* of the polyhedron if the image has the same combinatorial structure, that is, the (labeled) plane graph induced by the projection of the visible faces of the polyhedron on the viewplane is the same. The *visual space partition (VSP)* is a partition of all space exterior to an object into connected regions or cells of constant aspect. Finally, the *aspect graph* is the dual of the VSP (dual in the sense used in Sections 1.2.3 and 4.4), with a node for every region and an arc connecting regions that share a face.

Arrangements provide a clean framework for understanding VSPs (and therefore aspect graphs) for convex polyhedra, an important special case. For a polytope P, the VSP is precisely the arrangement formed by planes containing the faces of P (Plantinga & Dyer 1990). For example, consider a cube that partitions space into twenty six unbounded regions, as shown in Figure 6.15. There are six rectangular cylinders based on the cube faces, eight octants, one incident to each vertex, and twelve "wedges," one incident to each edge of the cube. Consider the view of the cube from a point p that moves from one cell A, across a face f of the arrangement, to an adjacent cell B, as illustrated in the figure. Suppose from cell A the cube face F, in whose plane the arrangement face f lies, is visible. Then when p is on f, it views F edge-on, and when p is in cell B, F is no longer visible. So f indeed represents a transition in the aspect.

From Theorem 6.4.1 we obtain immediately that the VSP of a convex polytope of n vertices has size $O(n^3)$ and can be constructed in $O(n^3)$ time. The aspect graph is then available by traversing the representation of the VSP.

The aspect graph may be defined for general, nonconvex polyhedra as well, where its combinatorial complexity shoots up to $\Theta(n^9)$ (under perspective projection)! See Gigus, Canny & Seidel (1991).

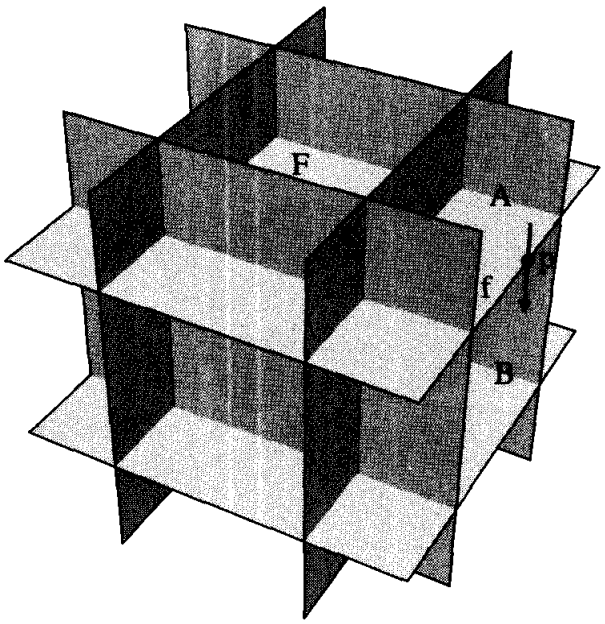


FIGURE 6.15 The arrangement of planes containing the faces of a cube. p can see F from cell A but not from cell B.

6.7.4. Smallest Polytope Shadow

Consider the problem of finding the smallest area shadow a given polytope P can cast orthogonally on a plane from a light source at infinity. This problem was first investigated by McKenna & Seidel (1985) and McKenna (1989), who gave a solution based on arrangements. I will sketch their employment of arrangements, without explaining their solution in full.

The primary insight is the same as the basis of aspect graphs: The combinatorial structure of the shadow projection changes when the viewpoint crosses a plane containing a face of P. What makes this problem different from the VSP construction is that the viewpoint/lightsource is at infinity, so the projection is orthographic rather than perspective. From a viewpoint infinitely far away, P in effect shrinks to a point, and all the face planes include that point. This intuition suggests the following approach.

Let π_f be the plane parallel to face f of P that passes through the origin. Let $\mathcal{A}$ be the arrangement of planes formed by π_f for all f of P. $\mathcal{A}$ cuts up space into unbounded cones apexed at the origin. Any vector u representing the direction of light rays falls inside some cone. The cone determines the combinatorial equivalence class of the view from infinity in the direction u and therefore the combinatorial structure of the shadow on a plane orthogonal to u. See Figure 6.16.

Although the combinatorial structure of the shadow is constant for any direction vector within one cone, the area of the shadow is not constant throughout the cone. McKenna and Seidel proved, however, that the minimum area is achieved along some edge of $\mathcal{A}$, that is, along a direction determined by the intersection of two face planes.

Although $\mathcal{A}$ is an arrangement of planes, and therefore has size $O(n^3)$ by Theorem 6.4.1, it is highly degenerate since all planes include the origin. In fact, it only has size $O(n^2)$, as the following argument shows.

Intersect $\mathcal{A}$ with a plane π parallel to the xy-plane, say $\pi : z = 1$. It should be clear that $\mathcal{A} \cap \pi = \mathcal{A}'$ is itself an arrangement of lines. Any direction vector u maps to the

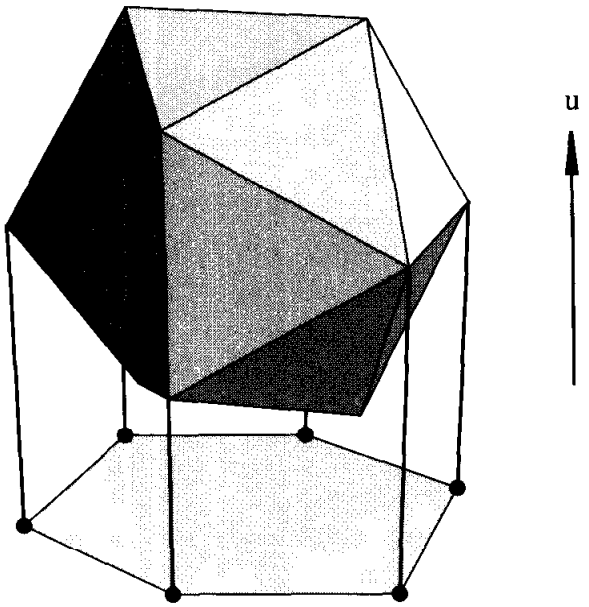


FIGURE 6.16 The shadow of a polytope from light at infinity is a convex polygon.

point on π that is the intersection of π with the line containing u. Thus all the viewpoints at infinity are in one-to-one correspondence with points in the two-dimensional arrangement $\mathcal{A}'$, which has complexity $O(n^2)$. Finally, a viewpoint that achieves minimum area corresponds to a vertex of $\mathcal{A}'$, a claim proved in McKenna & Seidel (1985).

We now have an algorithm. Construct $\mathcal{A}'$ in $O(n^2)$ time (there is no need to construct $\mathcal{A}$). For each of its $O(n^2)$ vertices, compute the area of the shadow on the plane orthogonal to the direction determined by the vertex. Return the smallest area.

What remains is to compute the shadow area from each vertex of $\mathcal{A}'$, and this is the part I will not explain. There is a clever method of avoiding recomputing the area at each vertex, which achieves constant time per vertex, thereby yielding $O(n^2)$ time overall (Exercise 6.7.5[1]).

6.7.5. Exercises

1. *Area calculation.*
   a. Let N be an *area normal*, a vector perpendicular to a face F, whose length is the area of F. Let u be the viewing direction. Show that the area of the projection of F onto a plane orthogonal to u is $N \cdot u$.
   b. Let N_i be area normals for faces F_i. Show that the area of the projection of all the F_i is $(\sum N_i) \cdot u$.
   c. Use (b) to show how to compute the area of the polytope shadow from the direction determined by each vertex of $\mathcal{A}'$.
2. *Maximum area shadow.* Find the maximum area shadow for a unit cube, projected onto a plane orthogonal to the light rays.

6.7.6. Ham-Sandwich Cuts

We will now explore the beautiful manner in which arrangements can be used to find ham-sandwich cuts for separated sets of points, as mentioned in Section 6.1. Define a *bisector* of a set of points to be a line that has at most half the points strictly to each side.

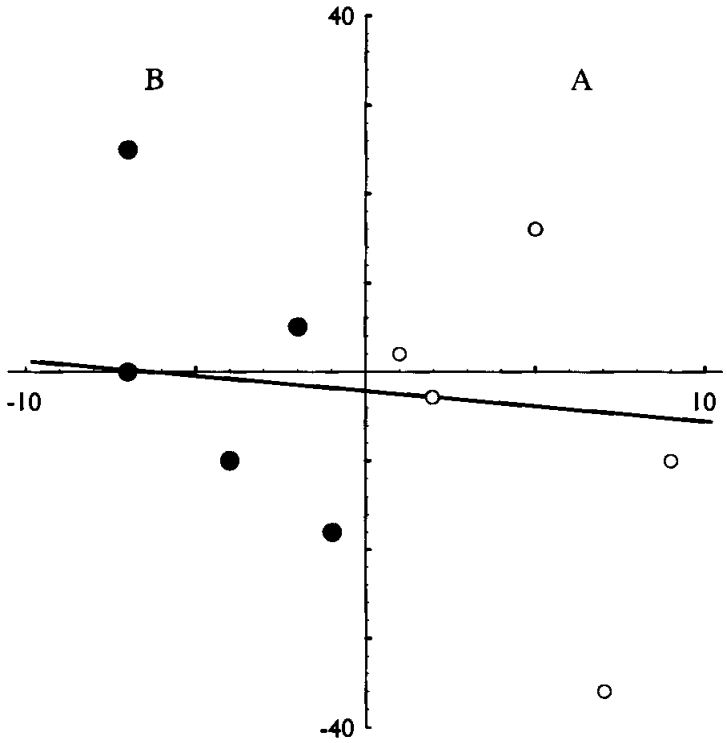


FIGURE 6.17 Two sets of five points each (from Figure 6.8): A right of the y axis, and B left. The line shown is a ham-sandwich cut: It bisects both A and B.

We will restrict our attention for simplicity to points in general position (no three on a line). In addition, we will assume our sets each have an odd number of points; thus a bisector of a set passes through (at least) one point (Exercise 6.7.7[1] asks for a removal of this restriction).

Consider first bisectors of a single set A of n points. Under our assumptions above, a set will never have only vertical bisectors, so we can safely ignore them. Dualize the points of A by the mapping $\mathbb{D}$ discussed in Section 6.5, producing an arrangement $\mathcal{A}$ of n lines. We now argue that all the bisectors of A dualize precisely to the *median level* $M_{\mathcal{A}}$ of $\mathcal{A}$. The median level is the collection of edges of $\mathcal{A}$ (and their connecting vertices) whose points have exactly $(n-1)/2$ lines strictly above them vertically (and the same number below). For by Lemma 6.5.5, a point $p \in M_{\mathcal{A}}$ dualizes to a line $\mathbb{D}(p)$ that has the same number of points below it as p has lines above it. Since p has $(n-1)/2$ lines above it by definition of the median level, $\mathbb{D}(p)$ has $(n-1)/2$ points of A below it: That is, $\mathbb{D}(p)$ bisects A. Thus $\mathbb{D}(p)$ is a bisector iff $p \in M_{\mathcal{A}}$.

Lemma 6.7.1. *The bisectors of a set of points dualize to the median level of the dual arrangement of lines.*

By this lemma, a line that is a ham-sandwich cut for A and B must dualize to a point that lies on both $M_{\mathcal{A}}$ and $M_{\mathcal{B}}$ (where $\mathcal{B}$ is the arrangement dual to B). Thus all ham-sandwich cuts can be found by intersecting the median levels of the two sets.

These two levels can intersect in a complicated way, but the situation is simpler if the two sets are separable by a line (as they often are in applications). Let A' and B' be two sets separable by a line. Then by a suitable translation and rotation, they can be transformed to sets A and B separated by the y axis (A right and B left). See Figure 6.17

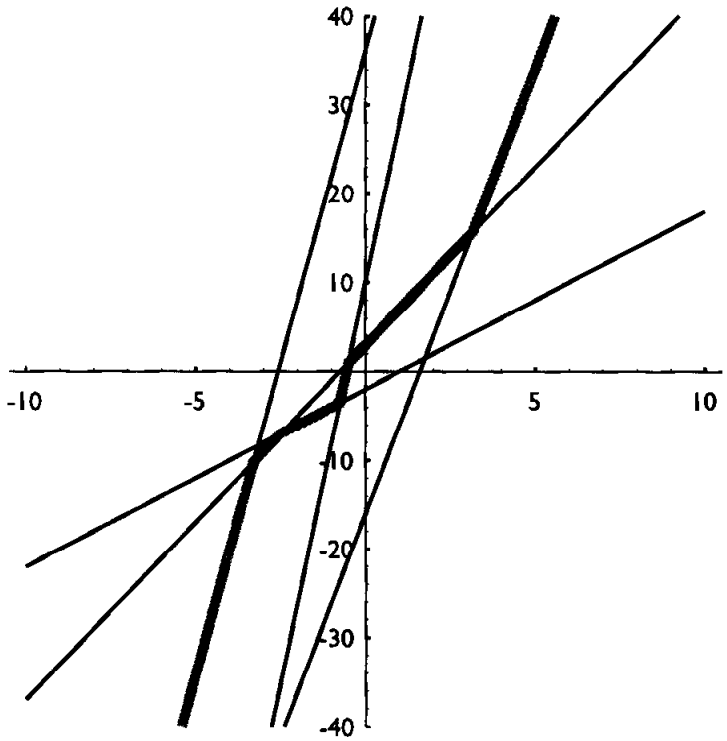


FIGURE 6.18 The duals of the points with $x > 0$ (set A in Figure 6.17) all have positive slope.

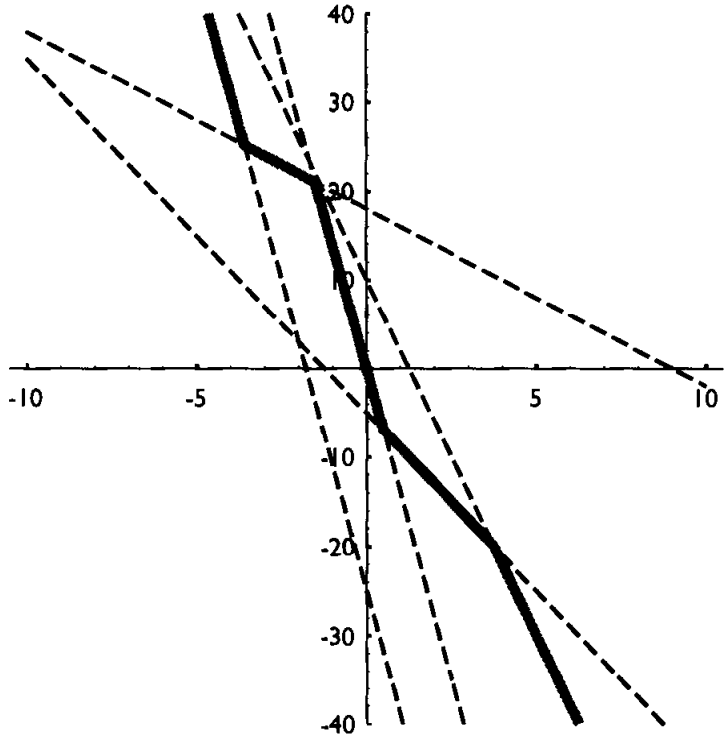


FIGURE 6.19 The duals of the points with $x < 0$ (set B in Figure 6.17) all have negative slope.

for an example. Now apply the dual mapping $\mathbb{D}$ to both. The lines in arrangement $\mathcal{A}$ all have positive slope, as shown in Figure 6.18, whereas the lines in B all have negative slope, as shown in Figure 6.19.

Because $M_{\mathcal{A}}$ is composed of subsegments of positively sloped lines, it is strictly monotonically increasing; similarly, $M_{\mathcal{B}}$ is strictly monotonically decreasing. (Both are drawn shaded in Figures 6.18 and 6.19.) Therefore they intersect in a single point: The

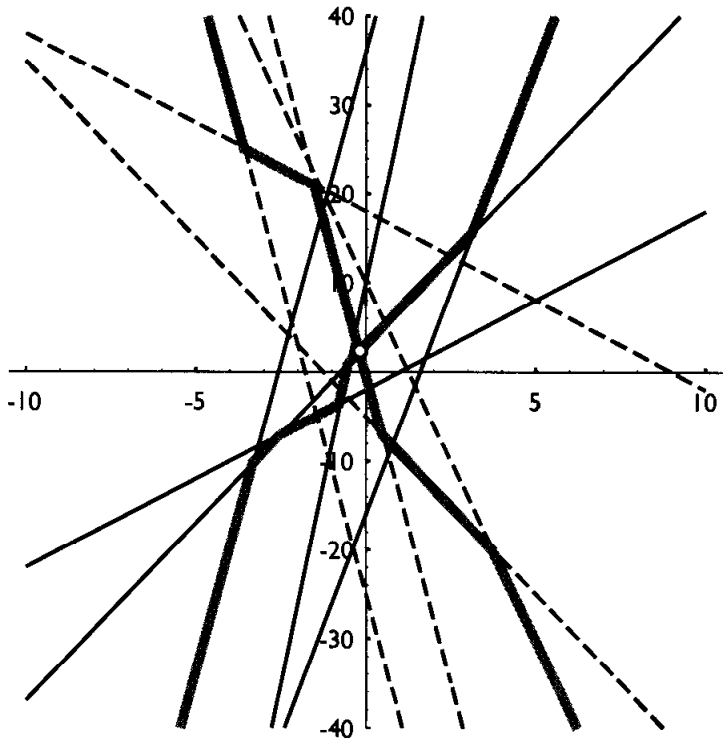


FIGURE 6.20 $\mathcal{A}$ and $\mathcal{B}$ together. The intersection of their median levels is at $(-\frac{1}{6}, 2\frac{1}{3})$.

ham-sandwich cut is unique. Figure 6.20 shows they intersect at $(-\frac{1}{6}, 2\frac{1}{3})$, and indeed the line $y = 2(-\frac{1}{6})x - \frac{7}{3}$ is the ham-sandwich cut for those two point sets shown in Figure 6.17.

It turns out that this intersection point can be found without constructing the entirety of either arrangement, in only $O(n + m)$ time for sets of n and m points! The algorithm is rather intricate, and I will not explain it here (Edelsbrunner 1987, pp. 336–45). Moreover, the same linear time complexity can be achieved for point sets that are not separated (Lo, Matoušek & Steiger 1994). This provides an optimal algorithm for an interesting matching problem first studied by Atallah (1985), which we describe next.

Red–Blue Matching

Given n red and n blue points in the plane, the task is to match them in red–blue pairs with noncrossing segments. The points might be features of an object that has translated between successive ("red" and "blue") frames of a computer vision system; the matching then recovers the translation. This is a specialized instance of the problem where all matching segments are not only noncrossing, but translates of one another. Other applications have no such length restriction.

The general red–blue matching problem can be solved by a divide-and-conquer algorithm that uses ham-sandwich cuts at each level. Consider the set of $n = 6$ red and 6 blue points shown in Figure 6.21(a). First ham-sandwich the set into two sets of three red and three blue points each (cut 1 in (b) of the figure). Each of these sets is next cut into one red point, one blue, and one red and blue on the cut (cuts labeled 2). Note the necessity to have these cuts pass through points to achieve a bisection, because the number of each color is odd (three in this case); otherwise we'd be left with an imbalance. Finally, each of the remaining sets of two points are separated again (dashed cuts). Now

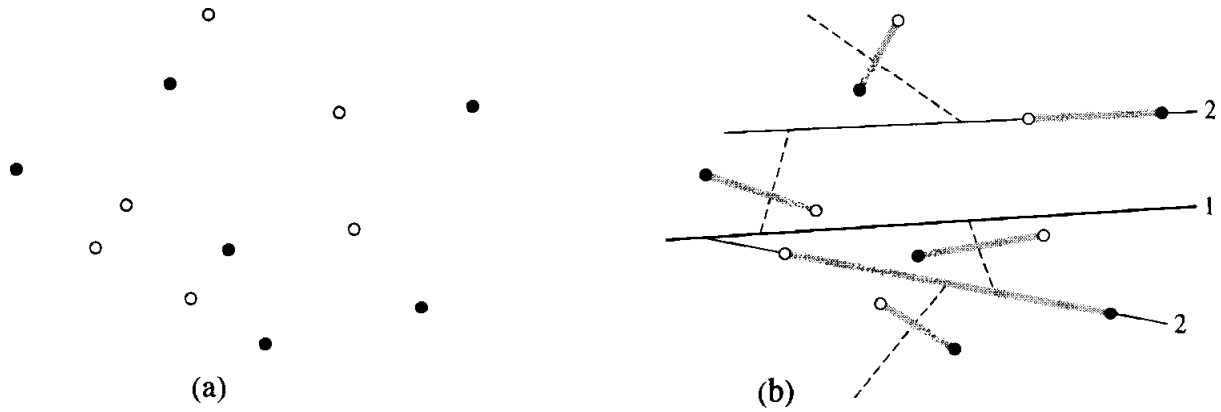


FIGURE 6.21 (a) $n = 6$ red (solid) and 6 blue (open) points, (b) Noncrossing matching segments shown in gray, after repeated ham-sandwich divisions.

it is trivial to match up the points by noncrossing segments, as shown in Figure 6.21(b). The final cuts give a direct matching of the pairs they separated, and some matches lie within cuts. It should be clear that no matching segments produced by this process cross (Exercise 6.7.7[4]).

The time complexity of this algorithm is $O(n \log n)$: linear work for finding the cuts on each of $O(\log n)$ levels. This can be shown to be optimal by reduction from sorting. See Lo et al. (1994) for details.

Higher Dimensions

Lastly we should mention that the ham-sandwich theorem generalizes to higher dimensions:

Theorem 6.7.2. *For any d sets of distinct points $P_1, \ldots, P_d$ in d dimensions, there is a hyperplane that simultaneously bisects each P_i.*

6.7.7. Exercises

1. *Even number of points* [easy]. Using the definition of a bisector, argue that the cases where A and/or B have an even number of points can be reduced to sets with an odd number of points.
2. *Size of median level.* Let A be the following set of points: Draw three rays from the origin separated by $2\pi/3 = 120°$. Place $n/3$ points equally spaced along each ray. Compute the number of edges in the median level of the arrangement formed by the duals of the points of A.
3. *Bisection program* [programming]. Write a program to find one bisector of a given set of points in the plane. Make no assumptions about the points aside from distinctness.
4. *Red–blue matching.* Prove that the match segments produced by the ham-sandwich divide-and-conquer algorithm are disjoint.

6.8. ADDITIONAL EXERCISES

1. *Centerpoints.* A point x is called a "centerpoint" of a set of n points P if every halfplane that includes x also includes a large proportion of the points of P (in a sense to be made precise momentarily). The point x is not necessarily in P. A centerpoint is "central" to P in the sense

that capturing it with a halfplane necessarily captures a large portion of P. The technical definition is that x is a *centerpoint* if no open halfplane that avoids x includes more than $\frac{2}{3}n$ points of P (Edelsbrunner 1987, p. 64).

a. Verify that every set of $n = 4$ points has a centerpoint by exploring "all" configurations of four points.

b. Interpret the claim that every finite set of points has a centerpoint in terms of levels in arrangements.

c. Suggest an algorithm for finding a centerpoint based on (b).

2. *Minimum area triangle.*

a. Prove that if points $\{a, b, c\}$ achieve a minimum area triangle among the points in a given finite point set P, then c is a closest point among $P \setminus \{a, b\}$ to the line L_{ab} containing ab, where distance is measured orthogonal to L_{ab}.

b. Interpret this relationship in the dual arrangement of lines $\mathcal{A}(P)$.

c. Use this relationship to design an algorithm for finding a minimum area triangle whose vertices are selected from a set of n points P in the plane. Try to beat the brute-force $O(n^3)$ algorithm.

3. *Voracious circle points.* Given a set of n points $P = \{p_1, \ldots, p_n\}$, define $\mu(p_i, p_j)$ as the fewest points of P contained in any closed disk that contains both p_i and p_j. Call a pair of points *voracious circle points* (Diaz 1990) if they maximize μ over all pairs of points in P. Call this maximum $M(P) = \max_{p_i, p_j \in P} \mu(p_i, p_j)$.

a. Determine $M(P)$ for all sets P of $n = 3$ points.

b. Determine $M(P)$ for all sets P of $n = 4$ points.

c. Prove that, if there is a disk D that includes p_i and p_j and k other points of P, there is a disk $D' \subseteq D$ whose boundary includes p_i and p_j, and which encloses $\leq k$ points of P.

d. Use (c) to design an algorithm to compute $\mu(p_i, p_j)$ for a fixed p_i and p_j.

e. Use (d) to design an algorithm to find a pair of voracious circle points.

4. *Four-section.* A *four-section* of a point set P is a pair of lines such that the number of points in each of the open wedges formed by these lines is no more than $\lceil n/4 \rceil$.

a. Argue that every finite point set has a four-section.

b. Design an algorithm to find a four-section.

5. *Orthogonal four-section.* Design an algorithm to find a four-section of a point set such that the two sectioning lines are orthogonal (Diaz 1990).

7

Search and Intersection

7.1. INTRODUCTION

In this (long) chapter we examine several problems that can be loosely classified as involving search or intersection (or both). This is a vast, well-developed topic, and I will make no attempt at systematic coverage.[1] The chapter starts with two constant-time computations that are generally below the level considered in the computational geometry literature: intersecting two segments (Section 7.2) and intersecting a segment with a triangle (Section 7.3). Implementations are presented for both tasks. Next we employ these algorithms for two more difficult problems: determining whether a point is in a polygon – the "point-in-polygon problem" (Section 7.4), and the "point-in-polyhedron problem" (Section 7.5). The former is a heavily studied problem; the latter has seen less scrutiny. Again implementations are presented for both. We next turn to intersecting two convex polygons (Section 7.6), again with an implementation (the last in the chapter). Intersecting a collection of segments (Section 7.7) leads to intersection of nonconvex polygons (Section 7.8).

The theoretical jewel in this chapter is an algorithm to find extreme points of a polytope in any given query direction (Section 7.10). This leads naturally to planar point location (Section 7.11), which allows us to complete the explanation of the randomized triangulation algorithm from Chapter 2 (Section 2.4.1) with a presentation of a randomized algorithm to construct a search structure for a trapezoid decomposition (Section 7.11.4).

7.2. SEGMENT–SEGMENT INTERSECTION

In Chapter 1 (Section 1.5) we spent some time developing code that detects intersection between two segments for use in triangulation (`Intersect`, Code 1.9), but we never bothered to *compute* the point of intersection. It was not needed in the triangulation algorithm, and it would have forced us to leave the comfortable world of integer coordinates. For many applications, however, the floating-point coordinates of the point of intersection are needed. We will need this to compute the intersections between two polygons in Sections 7.6 and 7.8. Fortunately, it is not too difficult to compute the intersection point (although there are potential pitfalls), and the necessary floating-point calculations are not as problematical here as they sometimes are. In this section we develop code for this task.

[1] See, e.g., de Berg et al. (1997).

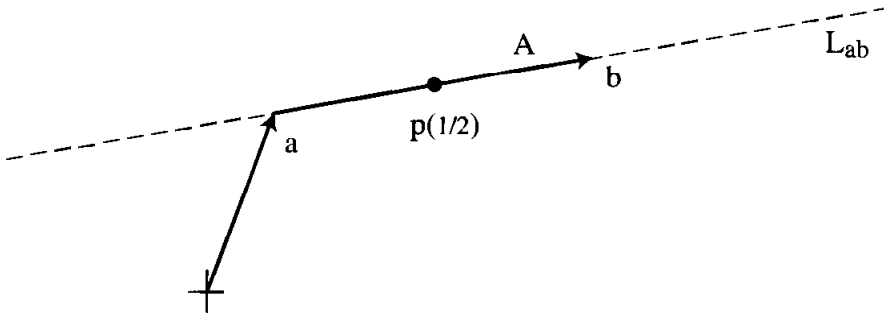


FIGURE 7.1 $p(s) = a + sA$; $p(\frac{1}{2}) = a + \frac{1}{2}A$ is shown.

Although the computation could be simplified a bit by employing the Boolean `Intersect` from Chapter 1, we opt here for an independent calculation. Let the two segments have endpoints a and b and c and d, and let L_{ab} and L_{cd} be the lines containing the two segments. A common method of computing the point of intersection is to solve slope–intercept equations for L_{ab} and L_{cd} simultaneously:[2] two equations in two unknowns (the x and y coordinates of the point of intersection). Instead we will use a parametric representation of the two segments, as the meaning of the variables seems more intuitive. We will see in Section 7.3 that the parametric approach generalizes nicely to more complex intersection computations.

Let $A = b - a$ and $C = d - c$; these vectors point along the segments. Any point on the line L_{ab} can be represented as the vector sum $p(s) = a + sA$, which takes us to a point a on L_{ab}, and then moves some distance along the line by scaling A by s. See Figure 7.1. The variable s is called the *parameter* of this equation. Consider the values obtained for $s = 0$, $s = 1$, and $s = \frac{1}{2}$: $p(0) = a$, $p(1) = a + A = a + b - a = b$, and $p(\frac{1}{2}) = (a + b)/2$. These examples demonstrate that $p(s)$ for $s \in [0, 1]$ represents all the points on the segment ab, with the value of s representing the fraction of the distance between the endpoints; in particular, the extremes of s yield the endpoints.

We can similarly represent the points on the second segment by $q(t) = c + tC$, $t \in [0, 1]$. A point of intersection between the segments is then specified by values of s and t that make $p(s)$ equal to $q(t)$: $a + sA = c + tC$. This vector equation also comprises two equations in two unknowns: the x and y equations, both with s and t as unknowns. With our usual convention of subscripts 0 and 1 indicating x and y coordinates, its solution is

$$s = [a_0(d_1 - c_1) + c_0(a_1 - d_1) + d_0(c_1 - a_1)]/D, \tag{7.1}$$

$$t = [a_0(c_1 - b_1) + b_0(a_1 - c_1) + c_0(b_1 - a_1)]/D, \tag{7.2}$$

$$D = a_0(d_1 - c_1) + b_0(c_1 - d_1) + d_0(b_1 - a_1) + c_0(a_1 - b_1) \tag{7.3}$$

Division by zero is a possibility in these equations. The denominator D happens to be zero iff the two lines are parallel, a claim left to Exercise 7.3.2[1]. Some parallel segments involve intersection, and some do not, as we detailed in Chapter 1 (Section 1.5.4). Temporarily, we will treat parallel segments as nonintersecting. The above equations lead to the rough code shown in Code 7.1. We will first describe this code, then criticize it, and finally revise it.

[2]E.g., see Berger (1986, pp. 332–5).

```
#define  X 0
#define  Y 1
typedef  enum {FALSE, TRUE }bool;
#define DIM 2                    /* Dimension of points */
typedef  int tPointi[DIM];       /* Type integer point */
typedef  double tPointd[DIM];    /* Type double point */

bool  SegSegInt( tPointi a, tPointi b, tPointi c, tPointi d,
                 tPointd p )
{

   double s, t;            /* Parameters of the parametric eqns. */
   double num, denom;      /* Numerator and denominator of eqns. */

   denom = a[X] * ( d[Y] - c[Y] ) +
           b[X] * ( c[Y] - d[Y] ) +
           d[X] * ( b[Y] - a[Y] ) +
           c[X] * ( a[Y] - b[Y] );

   /* If denom is zero, then segments are parallel. */
   if (denom == 0.0)
      return FALSE;

   num = a[X] * ( d[Y] - c[Y] ) +
         c[X] * ( a[Y] - d[Y] ) +
         d[X] * ( c[Y] - a[Y] );
   s = num / denom;

   num = -( a[X] * ( c[Y] - b[Y] ) +
            b[X] * ( a[Y] - c[Y] ) +
            c[X] * ( b[Y] - a[Y] ) );
   t = num / denom;

   p[X] = a[X] + s * ( b[X] - a[X] );
   p[Y] = a[Y] + s * ( b[Y] - a[Y] );

   if       ( (0.0 <= s) && (s <= 1.0) &&
              (0.0 <= t) && (t <= 1.0) )
         return TRUE;
   else  return FALSE;
}
```

Code 7.1 Segment–segment intersection code: rough attempt.

The code takes the four integer-coordinate endpoints as input and produces two types of output: It returns a Boolean indicating whether or not the segments intersect, and it returns in the point p the double coordinates of the point of intersection; note that p is of type `double tPointd`. The computations of the numerators and denominators parallel Equations (7.1)–(7.3) exactly, and the test for intersection is $0 \leq s \leq 1$ and $0 \leq t \leq 1$.

There are at least three weaknesses to this code:

1. The code does not handle parallel segments. Most applications will need to know whether the segments overlap or not.
2. Many applications need to distinguish proper from improper intersections, just as we did for triangulation in Chapter 1. It would be useful to distinguish these in the output.
3. Although floating-point variables are used, the multiplications are still performed with integer arithmetic before the results are converted to `doubles`. Here is a simple example of how this code can fail due to overflow. Let the four endpoints be

$$a = (-r, -r),$$
$$b = (+r, +r),$$
$$c = (+r, -r),$$
$$d = (-r, +r).$$

The segments form an 'X' shape intersecting at $p = (0, 0)$. Calculation shows that the numerators from Equations (7.1) and (7.2) are both $-4r^2$. For $r = 10^5$, this is -4×10^{10}, which exceeds what can be represented in 32 bits. In this case my machine returns $p = (-267702.8, -267702.8)$ as the point of intersection!

We now address each of these three problems. First, we change the function from `bool` to `char` and have it return a "code" that indicates the type of intersection found. Applications that need to base decisions on whether or not the intersection is proper can use this code. Although the exact codes used should depend on the application, the following capture most needs:

'e': The segments collinearly overlap, sharing a point; 'e' stands for 'edge.'
'v': An endpoint of one segment is on the other segment, but 'e' doesn't hold; 'v' stands for 'vertex.'
'1': The segments intersect properly (i.e., they share a point and neither 'v' nor 'e' holds); '1' stands for `TRUE`.
'0': The segments do not intersect (i.e., they share no points); '0' stands for `FALSE`.

Note that the case where two collinear segments share just one point, an endpoint of each, is classified as 'e' in this scheme, although 'v' might be more appropriate in some contexts.

Second, we increase the range of applicability of the code by forcing the multiplications to floating-point by casting with `(double)`. This leads us to the code shown in Code 7.2. Before moving to the parallel segment case, let us point out a few features

```
char  SegSegInt( tPointi a, tPointi b, tPointi c, tPointi d,
                 tPointd p )
{
   double s, t;          /* The two parameters of the parametric eqns. */
   double num, denom;    /* Numerator and denominator of equations. */
   char code = '?';      /* Return char characterizing intersection. */

  denom = a[X] * (double)( d[Y] - c[Y] ) +
          b[X] * (double)( c[Y] - d[Y] ) +
          d[X] * (double)( b[Y] - a[Y] ) +
          c[X] * (double)( a[Y] - b[Y] );

  /* If denom is zero, then segments are parallel: handle separately. */
  if (denom == 0.0)
     return ParallelInt(a, b, c, d, p);

  num = a[X] * (double)( d[Y] - c[Y] ) +
        c[X] * (double)( a[Y] - d[Y] ) +
        d[X] * (double)( c[Y] - a[Y] );
  if ( (num == 0.0) || (num == denom) ) code = 'v';
  s = num / denom;

  num = -( a[X] * (double)( c[Y] - b[Y] ) +
           b[X] * (double)( a[Y] - c[Y] ) +
           c[X] * (double)( b[Y] - a[Y] ) );
  if ( (num == 0.0) || (num == denom) ) code = 'v';
  t = num / denom;

  if      ( (0.0 < s) && (s < 1.0) &&
            (0.0 < t) && (t < 1.0) )
    code = '1';
  else if ( (0.0 > s) || (s > 1.0) ||
            (0.0 > t) || (t > 1.0) )
    code = '0';

  p[X] = a[X] + s * ( b[X] - a[X] );
  p[Y] = a[Y] + s * ( b[Y] - a[Y] );

  return code;
}
```

Code 7.2 `SegSegInt`.

```
char  ParallelInt( tPointi a, tPointi b, tPointi c, tPointi d,
                   tPointd p )
{
  if ( !Collinear( a, b, c) )
      return '0';

  if ( Between( a, b, c ) ) {
      Assigndi( p, c ); return 'e';
  }
  if ( Between( a, b, d ) ) {
      Assigndi( p, d ); return 'e';
  }
  if ( Between( c, d, a ) ) {
      Assigndi( p, a ); return 'e';
  }
  if ( Between( c, d, b ) ) {
      Assigndi( p, b ); return 'e';
  }
  return '0';
}

void Assigndi( tPointd p, tPointi a )
{
  int i;
  for ( i = 0; i < DIM; i++ )
      p[i] = a[i];
}

bool    Between( tPointi a, tPointi b, tPointi c )
{
   tPointi      ba, ca;

   /* If ab not vertical, check betweenness on x; else on y. */
   if ( a[X] != b[X] )
     return ((a[X] <= c[X]) && (c[X] <= b[X])) ||
            ((a[X] >= c[X]) && (c[X] >= b[X]));
   else
     return ((a[Y] <= c[Y]) && (c[Y] <= b[Y])) ||
            ((a[Y] >= c[Y]) && (c[Y] >= b[Y]));
}
```

Code 7.3 `ParallelInt`.

of this code. Checking the 'v' case is done with `num` rather than with s and t after division; this skirts possible floating-point inaccuracy in the division. The check for proper intersection is $0 < s < 1$ and $0 < t < 1$; the reverse inequalities yield no intersection.

With the computations forced to `doubles`, the range is greatly extended. I could only make it fail for coordinates each over a billion: $r = 1234567809 \approx 10^9$ in the previous overflow example. It is not surprising that it fails here, as $-4r^2$ is now over 10^{18}, which requires 60 bits, exceeding the accuracy of `double` mantissas on most machines.

Finally we come to parallel segments, handled by a separate procedure `ParallelInt`. Collinear overlap was dealt with in Chapter 1 with the function `Between` (Code 1.8), which is exactly what we need here: The segments overlap iff an endpoint of one lies between the endpoints of the other. There is one small simplification. In the triangulation code, we had `Between` check collinearity, but here we can make one check: If c is not collinear with ab, then the parallel segments ab and cd do not intersect. The straightforward code is shown in Code 7.3. Note that an endpoint is returned as the point of intersection p. It is conceivable that some application might prefer to have the midpoint of overlap returned; in Section 7.6 we will need the entire segment of overlap.

It should be clear that minor modification of this intersection code can find ray–segment, ray–ray, ray–line, or line–ray intersection, by altering the acceptable s and t ranges. For example, accepting any nonnegative s corresponding to a positive stretch of the first segment yields ray–segment intersection.

7.3. SEGMENT–TRIANGLE INTERSECTION

We now turn to the more difficult, but still ultimately straightforward, computation of the point of intersection between a segment and a triangle in three dimensions. We will use this code in Section 7.5 to detect whether a point is in a polyhedron, but it has many other uses. In fact this is one of the most prevalent geometric computations performed today, because it is a key step in "ray tracing" used in computer graphics: finding the intersection between a light ray and a collection of polygons in space.

We will again use a parametric representation to derive the equations. Throughout we will let $T = \triangle abc$ be the triangle and qr the segment, where q is viewed as the originating ("query") endpoint in case qr represents a ray and r is the "ray" endpoint. We will assume throughout that $r \neq q$, so the input segment has nonzero length.

7.3.1. Segment–Plane Intersection

The first step is to determine if qr intersects the plane π containing T. We will pursue this halfway goal throughout this subsection before turing to determining if the point of intersection lies in the triangle.

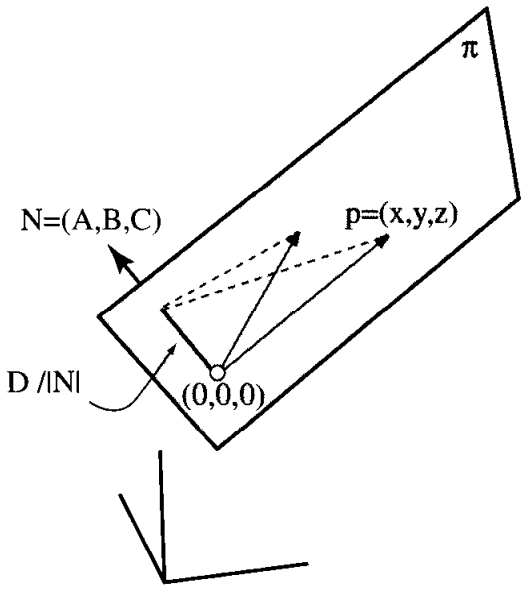


FIGURE 7.2 The dot product of a point on the plane with N is a constant, D when $|N| = 1$.

Recall that just as all points on a line must satisfy a linear equation in x and y, so too must all the points on a plane satisfy an equation

$$\pi : Ax + By + Cz = D. \tag{7.4}$$

We will represent the plane by these four coefficients.[3] It will help to view the first three coefficients as a vector (A, B, C), for then the plane equation can be viewed as a dot product:

$$\pi : (x, y, z) \cdot (A, B, C) = D. \tag{7.5}$$

The geometric meaning of this equation is that every point (x, y, z) on the plane projects to the same length onto (A, B, C). From this and Figure 7.2 it should be clear that $N = (A, B, C)$ is a vector normal (i.e., perpendicular) to the plane. If this vector is unit length,[4] then D is the distance from the origin to π.

Just as in two dimensions, any point $p(t)$ on the segment can be represented by moving out to the q endpoint, and then adding a scaled version of a vector along the segment: $p(t) = q + t(r - q)$. Let us temporarily assume that $q = (0, 0, 0)$ so that $p(t) = tr$; this will make the calculations more transparent. Now we are seeking a value of the parameter t that will stretch r out to the plane. Because every point on the plane must satisfy Equation (7.5), we must have

$$\begin{aligned} p(t) \cdot (A, B, C) &= D, \\ tr \cdot N &= D, \\ t(r \cdot N) &= D, \\ t &= \frac{D}{r \cdot N}. \end{aligned} \tag{7.6}$$

[3]One could "normalize" the equation by dividing by D; then only three coefficients are needed. This, however, requires dealing with $D = 0$ separately.

[4]Unit vectors are said to be "normalized," which can cause confusion with "normal" used in the sense of perpendicular.

For example, consider the plane containing the three points $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. Its plane equation is $x + y + z = 1$, so $N = (1, 1, 1)$. Let $r = (1, 2, 3)$. Equation (7.6) yields

$$t = \frac{1}{(1, 2, 3) \cdot (1, 1, 1)} = \frac{1}{6}.$$

And indeed $tr = \frac{1}{6}(1, 2, 3) = (\frac{1}{6}, \frac{1}{3}, \frac{1}{2})$ lies on the plane because $\frac{1}{6} + \frac{1}{3} + \frac{1}{2} = 1$.

Generalizing Equation (7.6) for arbitrary q is not difficult:

$$\begin{aligned} [q + t(r - q)] \cdot N &= D, \\ q \cdot N + t(r - q) \cdot N &= D, \\ t &= \frac{D - q \cdot N}{(r - q) \cdot N}. \end{aligned} \tag{7.7}$$

The reader may wonder why this more complex situation yields an equation in only one unknown, whereas the simpler segment–segment intersection led to two equations in two unknown parameters (Equations (7.1) and (7.2)). The reason is that we have not pinpointed the intersection with respect to the triangle yet; that will involve other unknowns. We could have followed this same strategy in two dimensions, first intersecting a segment with a line, but the situation there was simple enough to permit jumping right to simultaneous determination of the parameters.

One more step remains before we develop code: obtaining the plane coefficients. Usually, and in our case, we start with three points determining a triangle in space, not with the coefficients. The coefficients may be found using the observation above that three of them define a vector normal to the triangle. We can use the cross product from Chapter 1 (Equation (1.1)) to determine N, just as it was used in Chapter 5 to find the lower hull (Section 5.7.4). With N in hand, we can find the fourth coefficient D by substituting any point on the plane, for example, one of the triangle vertices, into Equation (7.5).

Segment–Plane Implementation

We now proceed to code. Looking ahead to how we will employ this code, we use the following simple data structure: Input points will be of type `tPointi`, stored in an array `tPointi Vertices[PMAX]`. Triangles are represented as three integer indices into this array. With some possibility of confusion, we will use type `tPointi` for these three indices. Eventually we will have a collection of triangles, which are stored in an array `tPointi Faces[PMAX]`. Thus `Face[i]` is a particular triangle, and if `T` is a triangle, `Vertices[T[j]]`, $j = 0, 1, 2$ are its three vertices. The data will be read in with straightforward read routines we will not show. See Code 7.4.

We partition the work into two procedures. The first, `PlaneCoeff`, computes N and D as just detailed. We choose to return the coefficients in two pieces, as they are employed in Equation (7.7) separately. For reasons that will become apparent later, we also record and return the index m of the largest component of N. See Code 7.5. The

```
#define X 0
#define Y 1
#define Z 2
#define DIM 3                          /* Dimension of points */
typedef int    tPointi[DIM];           /* Type integer point */
typedef double tPointd[DIM];           /* Type double point */
#define PMAX 10000                     /* Max # of pts */
tPointi Vertices[PMAX];                /* All the points */
tPointi Faces[PMAX];                   /* Each triangle face is 3 indices */

main()
{
   int V, F;

   V = ReadVertices();
   F = ReadFaces();
   /* Processing */
}
```

Code 7.4 Type definitions for triangles in space.

```
int     PlaneCoeff( tPointi T, tPointd N, double *D )
{
  int i;
  double t;                    /* Temp storage */
  double biggest = 0.0;        /* Largest component of normal vector. */
  int m = 0;                   /* Index of largest component. */

  NormalVec( Vertices[T[0]], Vertices[T[1]], Vertices[T[2]], N );
  *D = Dot( Vertices[T[0]], N );

  /* Find the largest component of N. */
  for ( i = 0; i < DIM; i++ ) {
    t = fabs( N[i] );
    if ( t > biggest ) {
      biggest = t;
      m = i;
    }
  }
  return m;
}
```

Code 7.5 `PlaneCoeff`.

```
void    NormalVec( tPointi a, tPointi b, tPointi c, tPointd N )
{
  N[X]=( c[Z] - a[Z] ) * ( b[Y] - a[Y] ) -
       ( b[Z] - a[Z] ) * ( c[Y] - a[Y] );
  N[Y]=( b[Z] - a[Z] ) * ( c[X] - a[X] ) -
       ( b[X] - a[X] ) * ( c[Z] - a[Z] );
  N[Z]=( b[X] - a[X] ) * ( c[Y] - a[Y] ) -
       ( b[Y] - a[Y] ) * ( c[X] - a[X] );
}

double  Dot( tPointi a, tPointd b )
{
  int i;
  double sum = 0.0;

  for( i = 0; i < DIM; i++ )
     sum += a[i] * b[i];
  return sum;
}

void    SubVec( tPointi a, tPointi b, tPointi c )
{
  int i;

  /* a - b => c. */
  for( i = 0; i < DIM; i++ )
     c[i] = a[i] - b[i];
}
```

Code 7.6 Vector utility functions.

code for `NormalVec` (Code 7.6) follows Code 4.12, returning $N = (b - a) \times (c - a)$. Note that we represent N with `double`s even though its coordinates are integers, for the familiar reason: to stave off overflow.

We will follow the convention established in Section 7.2 in having the intersection procedure return a code to classify the intersection:

'p': The segment lies wholly within the plane.
'q': The (first) q endpoint is on the plane (but not 'p').
'r': The (second) r endpoint is on the plane (but not 'p').
'0': The segment lies strictly to one side or the other of the plane.
'1': The segment intersects the plane, and none of {p, q, r} hold.

We now discuss how to determine when the code 'p' applies. When the denominator of Equation (7.7) is zero, then qr is parallel to the plane π. This can perhaps best be seen in the simpler version, Equation (7.6), where it is clear that the denominator is zero iff r is orthogonal to N (i.e., if r is parallel to the plane π to which N is orthogonal). It is also clear from that equation that if, in addition, the numerator D is zero, then

```
char SegPlaneInt( tPointi T, tPointi q, tPointi r, tPointd p,
                  int *m)
{
  tPointd N; double D;
  tPointi rq;
  double num, denom, t;
  int i;

  *m = PlaneCoeff( T, N, &D );
  num = D - Dot( q, N );
  SubVec( r, q, rq );
  denom = Dot( rq, N );

  if ( denom == 0.0 ) {   /* Segment is parallel to plane. */
     if ( num == 0.0 )    /* q is on plane. */
        return 'p';
     else
        return '0';
  }
  else
     t = num / denom;

  for( i = 0; i < DIM; i++ )
     p[i] = q[i] + t * ( r[i] - q[i] );

  if ( (0.0 < t) && (t < 1.0) )
       return '1';
  else if ( num == 0.0 )       /* t == 0 */
       return 'q';
  else if ( num == denom )     /* t == 1 */
       return 'r';
  else return '0';
}
```

Code 7.7 `SegPlaneInt`.

r lies in the plane. Generalizing to qr, we see in Equation (7.7) that the numerator is zero whenever $q \cdot N = D$, which is precisely the plane equation ((7.5)) with q substituted. So the numerator is zero iff q lies on π. Thus the code 'p' should be returned iff both the numerator and denominator are zero. The codes 'q' and 'r' are determined by $t = 0$ and $t = 1$ respectively, which may be tested on the numerator and denominator to avoid reliance on the floating-point division.

See Code 7.7.

Segment–Triangle Classification

Now that we have the point p of intersection between the segment qr and the plane π containing the triangle T, it only remains to classify the relationship between p and T: Is it inside or out, on the boundary, at a vertex? Although this may seem a simple task,

there are some subtleties. We first describe an elegant mathematical approach that we will ultimately choose not to implement.

Barycentric Coordinates. The *barycenter* of an object is its center of gravity.[5] The *barycentric coordinates* of a point p with respect to a triangle $T = \triangle abc$ (in two or three or any dimensions) are the unique real numbers (α, β, γ) that sum to 1 such that

$$\alpha a + \beta b + \gamma c = p. \tag{7.8}$$

From the discussion of convex combinations and affine combinations in Chapter 3 (Section 3.1 and Exercise 3.2.3[4]), it should be clear that Equation (7.8) describes a point on the plane π. The point is in T iff each of the three barycentric coordinates is in $[0, 1]$. The coordinates can be viewed as masses placed at the vertices whose center of gravity is p. For example, let $a = (0, 0)$, $b = (1, 0)$, and $c = (3, 2)$. The barycentric coordinates $(\alpha, \beta, \gamma) = (\frac{1}{2}, 0, \frac{1}{2})$ specify the point $p = (0, 0)/2 + 0(1, 0) + (3, 2)/2 = (\frac{3}{2}, 1)$, the midpoint of the ac edge.

This example illustrates that all the classes we might want to distinguish are encoded in the barycentric coordinates: p is on an edge interior iff exactly one coordinate is zero, p coincides with a vertex iff one coordinate is one, and of course the inside/outside distinction is determined by whether the coordinates are in $[0, 1]$.

The barycentric coordinates can be calculated by solving the Equation (7.8) together with $\alpha + \beta + \gamma = 1$. This gives four equations in three unknowns for three-dimensional triangles. Because the triangle lies in plane, we have redundant information, and the problem can be reduced to solving three equations in three unknowns.

Although it is quite possible to perform this computation, we choose another tack, partly to connect with techniques we used in Chapter 4, and partly because the computation slides into needing considerable precision. Let us make a crude estimate of this precision, assuming no attempt at optimizing. If our input coordinates use L bits of precision, then the normal vector N uses $2L$, and $q \cdot N$ consumes $3L$. Thus the numerator and, similarly, the denominator of Equation (7.7) are each $3L$, so t needs potentially $6L$ bits. Next t is multiplied by $r - q$, raising the count to $7L$ for p. And we have not even started solving the barycentric coordinate equations. We conclude that it will be delicate to classify p based on the floating-point representation of p. Nevertheless, we will in any case need to classify p when it is an endpoint of the query segment (which has precision only L), and we proceed to this task next.

Projection to Two Dimensions. The situation is that we have a point p known to lie on the plane π containing triangle T, and we would like to classify p's relationship to T. Because p lies in π, the problem is fundamentally two dimensional, not three dimensional. However, it would take a bit of work to translate and rotate π so that it coincides with, say, the xy-plane. But two observations allow us to solve the problem

[5] "Bary" means "heavy" in Greek.

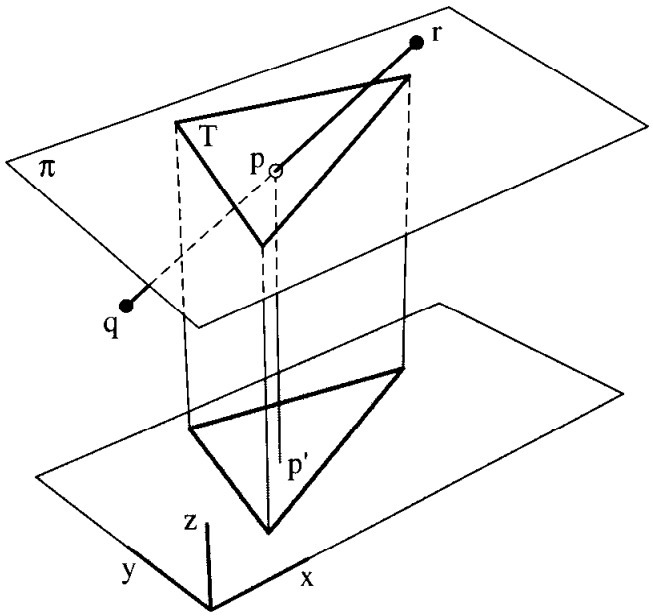


FIGURE 7.3 $p \in T$ iff $p' \in T'$.

in two dimensions without this realignment of the plane. First, p is in T iff it is in a projection of T, say to the xy-plane. This is evident from Figure 7.3. Let p' and T' be the projections of p and T respectively. The complete classification of p with respect to T can be made with these projections: p is in the interior of an edge of T iff p' is in the interior of an edge of T', and so on. But there is a worry: What if π is vertical, when the claim just made fails? This can be avoided by a second observation: Projecting out the coordinate corresponding to the largest component of the vector N normal to π guarantees nondegeneracy. Thus a nearly horizontal plane has a large z component, and projection to the xy-plane is called for. A vertical plane's N will have zero z component and so will be projected to either the xz- or yz-plane, depending on which one is closer to being parallel to π. This is why Code 7.5 computed the index m of the largest component.

We are now prepared to write a procedure `InTri3D` that classifies a point p on a triangle T using the following classification scheme:

'V': p coincides with a Vertex of T.
'E': p is in the relative interior of an Edge of T.
'F': p is in the relative interior of a Face of T.
'0': p does not intersect T.

The top-level code, shown in Code 7.8, does very little: It projects out coordinate m, and passes p' and T' to a procedure `InTri2D` that operates on the two-dimensional projection. Note that we fill up the x and y coordinate slots of `pp` and `Tp` regardless of the coordinate of projection.

Now that the problem is in the xy-plane, it is easy to solve. We can classify p' by computing signed areas as in Chapter 1. The only complication is that we do not know the orientation of the three vertices. But because there are only three, the given order

```
char    InTri3D( tPointi T, int m, tPointi p )
{
  int i;              /* Index for X,Y,Z */
  int j;              /* Index for X,Y */
  int k;              /* Index for triangle vertex */
  tPointi pp;         /* projected p */
  tPointi Tp[3];      /* projected T: three new vertices */

  /* Project out coordinate m in both p and the triangular face */
  j = 0;
  for ( i = 0; i < DIM; i++ ) {
    if ( i != m ) {     /* skip largest coordinate */
      pp[j] = p[i];
      for ( k = 0; k < 3; k++ )
        Tp[k][j] = Vertices[T[k]][i];
      j++;
    }
  }
  return( InTri2D( Tp, pp ) );
}
```

Code 7.8 `InTri3D`.

must be either counterclockwise or clockwise. The code must handle both orientations. `InTri2D` first computes the three areas determined by p' with each of the three edges of T'. The classification is based on these areas. See Figure 7.4 (and compare with Figure 1.19). If all three are positive, or all three negative, p' is strictly interior to T'. If two are zero, then p' lies on the lines containing two edges (i.e., at a vertex). If all three are zero, then p' must be collinear with all three edges, which can only happen when T' lies in a line. This case should never occur, so we exit with an error message. That leaves only the case when a single area is zero and the other two are nonzero. Only when the other two have the same sign does p' lie on the interior of an edge of T'. This leads to the code shown in Code 7.9.

Segment in Plane. It should be clear now that the case where the segment qr lies in the plane π can be handled by the same projection method: Project to two dimensions, check if either q' or r' lies in T' (in which case the corresponding endpoint may be returned as p), and if not, check if $q'r'$ intersects each edge of T', using `SegSegInt` (Code 7.2). As we have all these pieces of code assembled, we will not pursue this further, but rather leave the implementation details to Exercise 7.3.2[4].

Classification by Volumes. We are finally prepared to tackle the "usual" case, where qr crosses plane π, and therefore q is on one side and r is on the other. We can classify how qr meets T in a manner similar to how we classified p' in `InTri2D`, except now we compute volumes rather than areas. In particular, we compute the signed volumes of

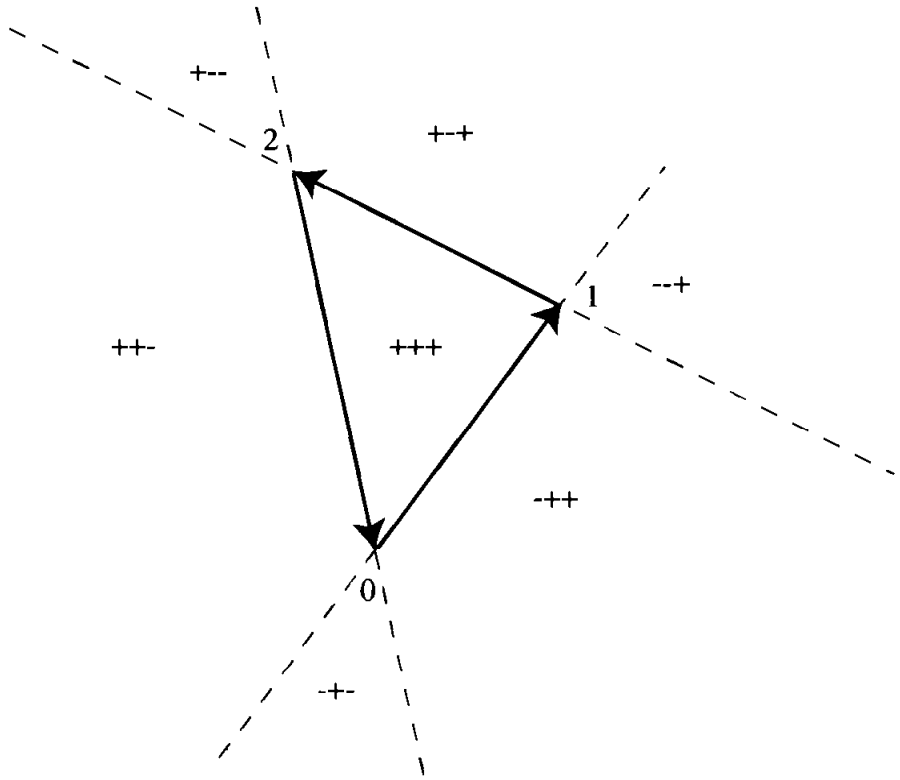


FIGURE 7.4 Assuming the edges of T' are counterclockwise, the sign pattern of the areas determined by p' and each edge are as shown. The boundary line between each + and − has "sign" 0.

the three tetrahedra determined by qr and each edge of T.[6] Let $T = (v_0, v_1, v_2)$. Then the volumes we use are $V_i = \text{Volume}(q, v_i, v_{i+1}, r)$. As with the two-dimensional case, we can only assume the vertices are ordered counterclockwise or clockwise, but this is enough information. We will employ this classification scheme:

'v': The open segment includes a vertex of T.
'e': The open segment includes a point in the relative interior of an edge of T.
'f': The open segment includes a point in the relative interior of a face of T.
'0': The open segment does not intersect triangle T.

If all three V_i are positive, or all three negative, then qr goes through a point strictly interior to T; see Figure 7.5(f). If two of the V_i are of opposite sign, then qr misses T. If one is zero, then qr passes through a point interior to some edge. For example, in Figure 7.5(e), $V_1 = 0$. If two are zero, qr passes through a vertex. In Figure 7.5(v), $V_1 = V_2 = 0$. If all three V_i are zero this implies that qr lies in the plane of T, a situation handled earlier. All these conditions are easily seen to be necessary and sufficient for the corresponding characterization. The straightforward implementation of these rules is embodied in the procedure `SegTriCross`, Code 7.10. `VolumeSign` is the same

[6] An elegant formulation of the same computation can be based on "Plücker coordinates" (Erickson 1997).

```
char    InTri2D( tPointi Tp[3], tPointi pp )
{
  int area0, area1, area2;

  area0 = AreaSign( pp, Tp[0], Tp[1] );
  area1 = AreaSign( pp, Tp[1], Tp[2] );
  area2 = AreaSign( pp, Tp[2], Tp[0] );

  if ( ( area0 == 0 ) && ( area1 > 0 ) && ( area2 > 0 )||
       ( area1 == 0 ) && ( area0 > 0 ) && ( area2 > 0 )||
       ( area2 == 0 ) && ( area0 > 0 ) && ( area1 > 0 ))
    return 'E';

  if ( ( area0 == 0 ) && ( area1 < 0 ) && ( area2 > 0 )||
       ( area1 == 0 ) && ( area0 < 0 ) && ( area2 > 0 )||
       ( area2 == 0 ) && ( area0 < 0 ) && ( area1 > 0 ))
    return 'E';

  if ( ( area0 > 0 ) && ( area1 > 0 ) && ( area2 > 0 )||
       ( area1 < 0 ) && ( area1 < 0 ) && ( area2 < 0 ))
    return 'F';

  if ( ( area0 == 0 ) && ( area1 == 0 ) && ( area2 == 0 ) )
    fprintf( stderr, "Error in InTriD\n" ),
      exit (EXIT_FAILURE);

  if ( ( area0 == 0 ) && ( area1 == 0 ) ||
       ( area0 == 0 ) && ( area2 == 0 ) ||
       ( area1 == 0 ) && ( area2 == 0 ) )
    return 'V';

  else
    return '0';
}
```

Code 7.9 `InTri2D`. See Code 4.23 for `AreaSign`.

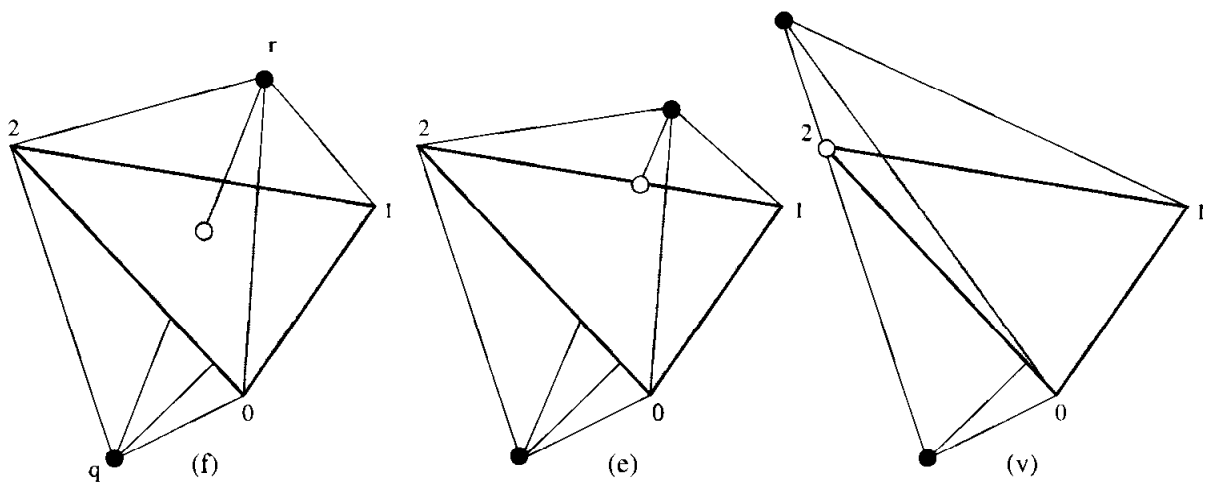


FIGURE 7.5 The segment qr intersects T in the face (f), on an edge (e), or through a vertex (v).

```
char SegTriCross( tPointi T, tPointi q, tPointi r )
{
  int vol0, vol1, vol2;

  vol0 = VolumeSign( q, Vertices[ T[0] ], Vertices[ T[1] ], r );
  vol1 = VolumeSign( q, Vertices[ T[1] ], Vertices[ T[2] ], r );
  vol2 = VolumeSign( q, Vertices[ T[2] ], Vertices[ T[0] ], r );

  /* Same sign: segment intersects interior of triangle. */
  if ( ( ( vol0 > 0 ) && ( vol1 > 0 ) && ( vol2 > 0 ) ) ||
       ( ( vol0 < 0 ) && ( vol1 < 0 ) && ( vol2 < 0 ) ) )
    return 'f';

  /* Opposite sign: no intersection between segment and triangle. */
  if ( ( ( vol0 > 0 ) || ( vol1 > 0 ) || ( vol2 > 0 ) ) &&
       ( ( vol0 < 0 ) || ( vol1 < 0 ) || ( vol2 < 0 ) ) )
    return '0';

  else if ( ( vol0 == 0 ) && ( vol1 == 0 ) && ( vol2 == 0 ) )
    fprintf( stderr, "Error 1 in SegTriCross\n" ),
    exit (EXIT_FAILURE);

  /* Two zeros: segment intersects vertex. */
  else if ( ( ( vol0 == 0 ) && ( vol1 == 0 ) ) ||
            ( ( vol0 == 0 ) && ( vol2 == 0 ) ) ||
            ( ( vol1 == 0 ) && ( vol2 == 0 ) ) )
    return 'v';

  /* One zero: segment intersects edge. */
  else if ( ( vol0 == 0 ) || ( vol1 == 0 ) || ( vol2 == 0 ) )
    return 'e';

  else
    fprintf( stderr, "Error 2 in SegTriCross\n" ),
    exit (EXIT_FAILURE);
}
```

Code 7.10 `SegTriCross`. See Code 4.16 for `VolumeSign`.

as Code 4.16 used in Chapter 4, with accomodation for the slightly different input data structures.

This completes our development of code to intersect a segment with a triangle. The simple top-level procedure is shown in Code 7.11. With `InPlane` unimplemented and simply returning 'p', the code returns a character in {`0`, `p`, `V`, `E`, `F`, `v`, `e`, `f`}, with the following mutually exclusive meanings:

'0': The closed segment does not intersect T.

'p': The segment lies wholly within the plane of T. All the remaining categories assume that 'p' does not hold.

'V': An endpoint of the segment coincides with a Vertex of T.
'E': An endpoint of the segment is in the relative interior of an Edge of T.
'F': An endpoint of the segment is in the relative interior of a Face of T.
'v': The open segment includes a vertex of T.
'e': The open segment includes a point in the relative interior of an edge of T.
'f': The open segment includes a point in the relative interior of a face of T.

The return codes may be viewed as a refinement on the usual Boolean 0/1, expanding 1 into seven types of degenerate intersection. As mentioned earlier, it is easy to modify this to permit intersection of a ray or line with a triangle, by permitting the range of the parameter t to vary outside of [0, 1]. We will delay illustrating the use of this code until Section 7.5.

```
char    SegTriInt( tPointi T, tPointi q, tPointi r, tPointd p )
{
  int code;
  int m;

  code = SegPlaneInt( T, q, r, p, &m );

  if      ( code == 'q')
      return InTri3D( T, m, q );
  else if ( code == 'r')
      return InTri3D( T, m, r );
  else if ( code == 'p' )
      return InPlane( T, m, q, r, p );
  else
      return SegTriCross( T, q, r );
}
```

Code 7.11 `SegTriInt`.

7.3.2. Exercises

1. *Denominator zero.* Prove that the denominator in the segment–segment intersection equations (Equations (7.1)–(7.3)) is zero iff the segments are parallel.
2. *Ray–segment intersection* [programming]. Modify the `SegSegInt` code to `RaySegInt`, interpreting a as a ray origin and b as a point on the ray, so that it returns a character code indicating the variety of possible intersections between the ray and the segment cd.
3. *Barycentric coordinates.* Let p be a point in the triangle $T = (v_1, v_2, v_3)$ with barycentric coordinates (t_1, t_2, t_3). Join p to the three vertices, partitioning T into three triangles. Call them T_1, T_2, T_3, with T_i not incident to v_i. Prove that the areas of these triangles T_i are proportional to the barycentric coordinates t_i of p (Coxeter 1961, p. 217).
4. *Segment in plane* [programming]. Extend the `SegTriInt` code to handle the case where qr lies entirely in the plane of T, by implementing an appropriate procedure `InPlane`.

7.4. POINT IN POLYGON

Every time a mouse is clicked inside a shape on a workstation screen, an instance of the point-in-polygon problem is solved: Given a fixed polygon P and a query point q, is $q \in P$? Although the hardware of a particular machine may permit solutions that avoid geometry, we consider the problem here from the computational geometry viewpoint.

If P is convex, the obvious method is to perform a `LeftOn` test (Code 1.6) for each edge of the polygon. Indeed, we used precisely this technique in the two-dimensional incremental hull algorithm in Chapter 3 (Section 3.7). This can be improved to $O(\log n)$, but we leave this to Exercise 7.4.3[1].

The more interesting case is when P is nonconvex. Two rather different methods for solving this problem have become popular: counting ray crossings and computing "winding" numbers.[7] Both are $O(n)$, but one is significantly faster than the other. These algorithms are the topics of the next two subsections.

7.4.1. Winding Number

We start with a mathematically pleasing method that, alas, has been shown to be greatly inferior in practice. It is based on the notion of the "winding number" of a polygon. Imagine you are standing at point q. While watching a point p completely traverse ∂P counterclockwise, pivot so that you always face toward p. If $q \in P$, you would turn a full circle, 2π radians, whereas if $q \notin P$, your total angular turn would be exactly zero (with the usual convention: counterclockwise turns are positive, and clockwise turns negative). This is easy to see if P is convex, and I hope at least intuitively believable when P is arbitrary: After all, you return to your starting orientation, so the total turn must be a whole number of revolutions. See Figure 7.6. We will not pause to prove this claim. The *winding number*[8] of q with respect to P is the number of revolutions ∂P makes about q: the total signed angular turn (call it ω) divided by 2π. We will leave details of the computation to Exercise 7.4.3[8].

Although the winding-number algorithm is appealing, its dependence on floating-point computations, and trigonometric computations in particular, makes it significantly slower (on standard hardware) than the ray-crossing algorithm which we discuss next: An implementation comparison showed it to be more than twenty times slower (Haines 1994)! This incidentally demonstrates the danger of thoughtlessly absorbing constants in the big-O notation.

7.4.2. Ray Crossings

Draw a ray R from q in an arbitrary direction (say, in the $+x$ direction), and count the number of intersections of R with ∂P. The point q is in or out of P if the number of crossings is odd or even, respectively. For example, suppose there are two crossings, as with point q_2 in Figure 7.7, and imagine traveling backwards along R from infinity to

[7] See Haines (1994) for other methods.

[8] See Chinn & Steenrod (1966, pp. 84–6); the winding number is also used in Foley et al. (1990, p. 965).

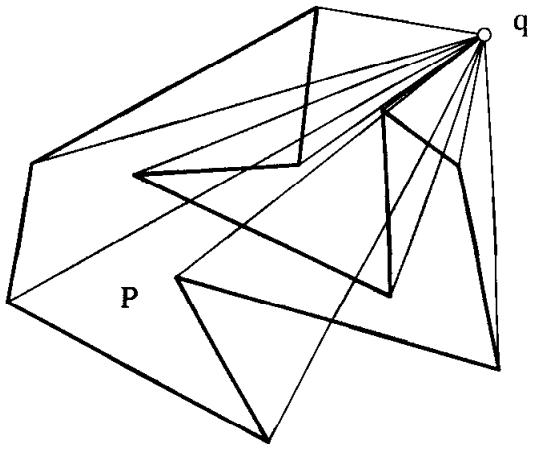


FIGURE 7.6 Exterior points (such as point q) have winding number 0: a total angular turn of 0.

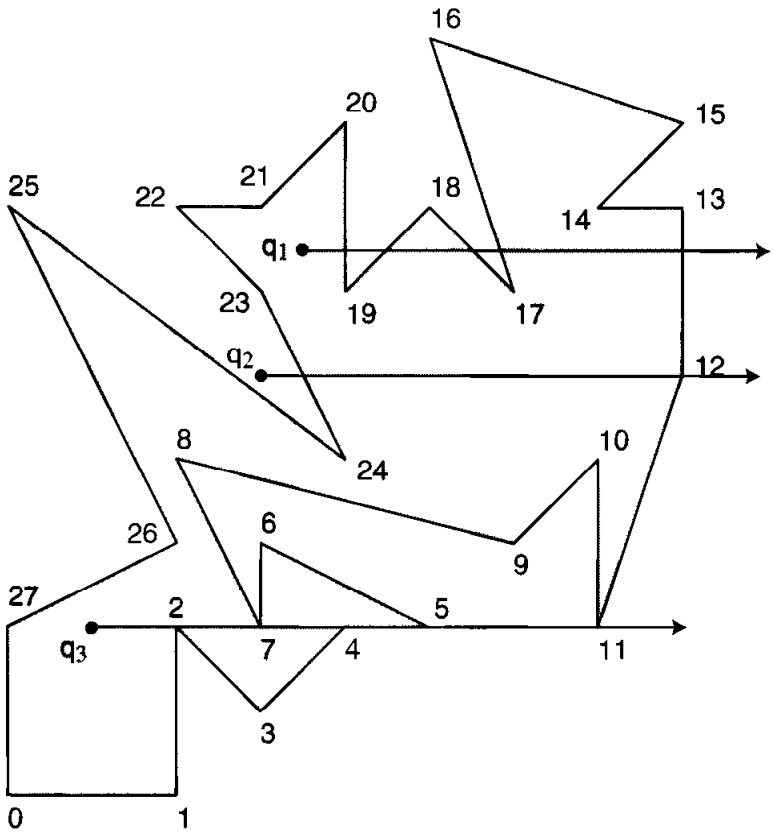


FIGURE 7.7 q_1's ray has five crossings and is inside; q_2's has two crossings and is outside; q_3's rays has five-crossings and is inside.

q_2. The first crossing penetrates to the interior of P; the second moves to the exterior. So $q_2 \notin P$. Similar reasoning shows that q_1 in the figure, whose ray has five crossings, must be inside P.

Despite the simplicity of this idea, implementation is fragile due to the necessity of handling special-case intersections of R with ∂P, as illustrated with point q_3 in Figure 7.7: The ray may hit a vertex or be collinear with an edge. There is also the possibility that q lies directly on ∂P, in which case we would like to conclude that $q \in P$ (because P is closed). Note that even the traditional assumption that no three polygon vertices are collinear will not exclude all these "degenerate" cases.

Fix R to be horizontal to the right. One method of eliminating most of the difficulties is to require that for an edge e to count as a *crossing* of R, one of e's endpoints must be strictly above R, and the other endpoint on or below. Informally, e is considered to include its lower endpoint but exclude its upper endpoint.[9] Applying this convention for q_3 of the figure, edges (1, 2) and (2, 3) are not crossing (neither edge has an endpoint strictly above), (6, 7) and (7, 8) do count as crossing (v_7 is on or below), (3, 4) and (4, 5) do not cross, and (5, 6), (10, 11), and (11, 12) all cross. The total of five crossings implies that $q_3 \in P$. Note that no edge collinear with the ray counts as crossing, as it has no point strictly above.

Before revealing what this convention leaves unresolved, we turn to simple code for a function `InPoly0` (Code 7.12) implementing the idea.[10] The code first translates the entire polygon so that q becomes the origin and R coincides with the positive x axis. This step is unnecessary (and wasteful) but makes the code more transparent.[11] In a loop over all edges $e = (i - 1, i)$, it checks whether e "straddles" the x axis according to the definition above. If e straddles, then the x coordinate of the intersection of e with $y = 0$ is computed via a straightforward formula obtained by solving for x in the equation

$$y - y_{i-1} = (x - x_{i-1})(y_i - y_{i-1})/(x_i - x_{i-1}) \tag{7.9}$$

and setting $y = 0$; here (x_{i-1}, y_{i-1}) and (x_i, y_i) are the endpoints of e. Note that `x` is `double` in the code; this dependence on a floating-point calculation can be eliminated (Exercise 7.4.3[7]), but we will leave it to keep attention focused on the algorithm. A crossing is counted whenever the intersection is strictly to the right of the origin. The code returns the character 'i' or 'o' to indicate "in" or "out" respectively.

There is a flaw to this code (aside from the floating-point calculation): Although it returns the correct answer for any point strictly interior to P, it does not handle the points on ∂P consistently. If q_3 is moved horizontally to v_4 in Figure 7.7, `InPoly0` returns `i`, but if q_3 is placed at v_5, it returns `o`. The behavior of this code for points on ∂P is complex, as shown in Figure 7.8. Let us analyze why v_{27} is considered inside. Neither edge (26, 27) nor (27, 0) counts as crossing, because of the strict inequality in the statement `if (x > 0)`. Otherwise v_{27}'s ray has the same five crossing as q_3's, and so $v_{27} \in P$. However, note that v_{22}, a superficially similar vertex, is deemed exterior. How the code treats edges is a bit easier to characterize: Left and horizontal bottom edges are in; right and horizontal top edges are out.

Although this hodgepodge treatment of points on the polygon boundary is dissatisfying from a purist's point of view, for some applications, notably GIS (Geographic Information Systems), this (or similar) behavior is preferred, because it has the property that in a partition of a region into many polygons, every point will be "in" exactly one polygon.[12] This is not obvious, but we'll take it as fact (Exercise 7.4.3[4]). Other

[9] This rule is followed, e.g., in the polygon-filling algorithm of Foley, van Dam, Feiner, Huges & Phillips (1993, Sec. 3.5).

[10] This code is functionally equivalent to many others, e.g., that in FAQ (1997) and Haines (1994).

[11] Note that as written the code will overwrite the polygon coordinates with the shifted coordinates, a side effect rarely desired. Exercise 7.4.3[6] asks for the simple modifications that avoid this.

[12] I owe this point to Haines (1997).

```
char InPoly0( tPointi q, tPolygoni P, int n )
{
  int    i, i1;          /* point index; i1 = i-1 mod n */
  int    d;              /* dimension index */
  double x;              /* x intersection of e with ray */
  int    Rcross = 0;     /* number of right edge/ray crossings */

  /* Shift so that q is the origin. Note this destroys the polygon.
     This is done for pedagogical clarity. */
  for( i = 0; i < n; i++ ) {
     for( d = 0; d < DIM; d++ )
        P[i][d] = P[i][d] - q[d];
  }

  /* For each edge e = (i-1,i), see if crosses ray. */
  for( i = 0; i < n; i++ ) {
     i1 = ( i + n - 1 ) % n;

    /* If e "straddles" the x axis... */
    if( ( ( P[i] [Y] > 0 ) && ( P[i1][Y] <= 0 ) ) ||
        ( ( P[i1][Y] > 0 ) && ( P[i] [Y] <= 0 ) ) ) {

      /* ... compute intersection with x axis. */
      x = (P[i][X] * P[i1][Y] - P[i1][X] * P[i][Y])
          / (double)(P[i1][Y] - P[i][Y]);

      /* e crosses ray if strictly positive intersection. */
      if (x > 0) Rcross++;
    }
  } /* end for */

  /* q inside iff an odd number of crossings. */
  if( (Rcross % 2) == 1 ) return 'i';
  else return 'o';
}
```

Code 7.12 `InPoly0`. Typedefs in Code 7.4 and 1.2.

applications demand a consistent treatment of the boundary points, or a distinction between strictly interior, strictly exterior, and on the boundary.

Although `InPoly0` could be modified with ad hoc tests to check if the query point lies on each edge of the boundary, the characterization of the treatment of edges mentioned above can be exploited to achieve relatively simple code that completely determines the relationship of q with P. The idea is this: If we reflect P in the origin to form a new polygon P', all left and bottom edges become top and right edges respectively, and vice versa. Consequently, points on edges can be distinguished by `InPoly0` giving different

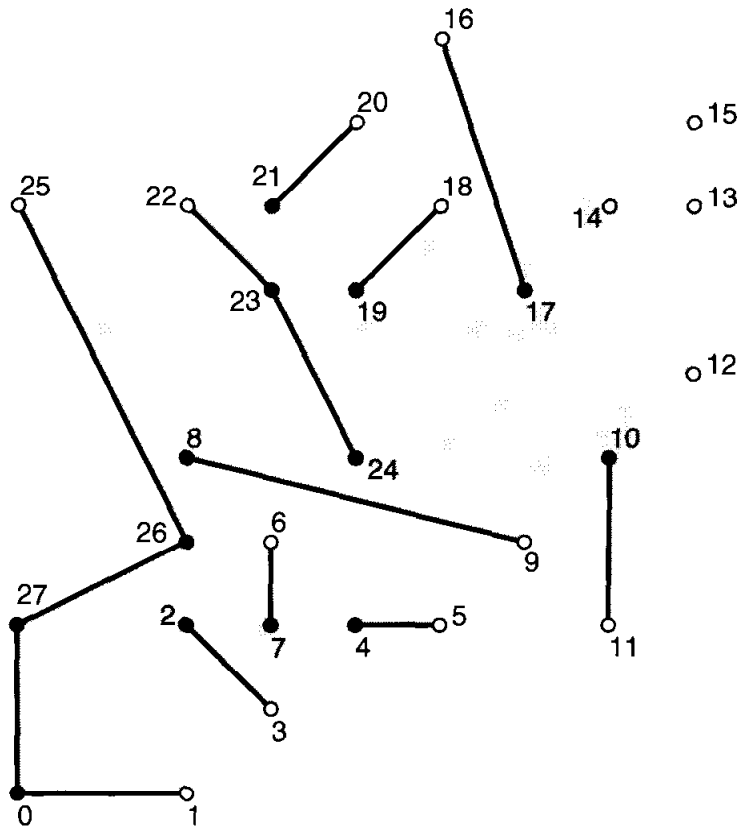


FIGURE 7.8 According to `InPoly0`, dark edges and solid vertices are `i`; other edges and open circles are `o`.

results for the same q in P and in P'. There is no need to actually perform the reflection: Using a leftward ray, with a "straddles" test biased below rather than the above bias used with rightward rays, achieves the same effect.

This method handles points on the relative interior of the edges but does not work for vertices (e.g., both of v_{16}'s rays have zero crossings, due to the `x > 0` test). An explicit easy check for q being a vertex handles this case. This leads to `InPoly1`, Code 7.13, which returns one of four `chars`: {`i`, `o`, `e`, `v`}, representing these mutually exclusive cases:

'i': q is strictly interior.
'o': q is strictly exterior (outside).
'e': q is on an edge, but not an endpoint.
'v': q is a vertex.

A few comments on this code need to be made. The straddles test from `InPoly0`,

```
( ( P[i] [Y] > 0 ) && ( P[i1][Y] <= 0 ) ) ||
( ( P[i1][Y] > 0 ) && ( P[i] [Y] <= 0 ) )
```

has been replaced in `InPoly1` with an assignment of this expression to the Boolean variable `Rstrad`:

```
( P[i][Y] > 0 ) != ( P[i1][Y] > 0 )
```

```
char InPoly1( tPointi q, tPolygoni P, int n )
{
  /* Declarations of i, i1, d, x same as in InPoly0; DELETED here. */
  int     Rcross = 0;       /* number of right edge/ray crossings */
  int     Lcross = 0;       /* number of left edge/ray crossings */
  bool    Rstrad, Lstrad;   /* flags indicating the edge strads the x axis. */

  /* Shift so that q is the origin, same as in InPoly0; DELETED here. */

  /* For each edge e = (i-1,i), see if crosses rays. */
  for( i = 0; i < n; i++ ) {
     /* First check if q = (0, 0) is a vertex. */
     if ( P[i][X]==0 && P[i][Y]==0 ) return 'v';
     i1 = ( i + n - 1 ) % n;

     /* Check if e straddles x axis, with bias above/below. */
     Rstrad = ( P[i][Y] > 0 ) != ( P[i1][Y] > 0 );
     Lstrad = ( P[i][Y] < 0 ) != ( P[i1][Y] < 0 );

     if( Rstrad || Lstrad ) {
        /* Compute intersection of e with x axis. */
        x = (P[i][X] * P[i1][Y] - P[i1][X] * P[i][Y])
        / (double)(P[i1][Y] - P[i][Y]);

        if (Rstrad && x > 0) Rcross++;
        if (Lstrad && x < 0) Lcross++;

     } /* end straddle computation */
  } /* end for */

  /* q on an edge if L/Rcross counts are not the same parity. */
  if( ( Rcross % 2 ) != (Lcross % 2 ) ) return 'e';

  /* q inside iff an odd number of crossings. */
  if( (Rcross % 2) == 1 ) return 'i';
  else return 'o';
}
```

Code 7.13 `InPoly1`. (Some portions shared with `InPoly0` are deleted above.)

Although it may not be obvious, these two expressions are logically equivalent: `Rstrad` is `TRUE` iff one endpoint of e is strictly above the x axis and the other is not (i.e., the other is on or below). This more concise form makes it easier to see the proper definition for `Lstrad`: Just reverse the inequalities to bias below. Now the computation of x is needed whenever either of these straddle variables is `TRUE`, which only excludes edges passing through $q = (0, 0)$ (and incidentally protects against division by 0). Finally, the

key determination of when to return `e` reduces to seeing if the two ray-crossing counts have different parity.

7.4.3. Exercises

1. *Point in convex polygon.* Design an algorithm to determine in $O(\log n)$ time if a point is inside a convex polygon.
2. *Worst ray crossings.* Could the ray crossing algorithm be made to run in $O(\log n)$ time for arbitrary query points?
3. *Division vs. multiplication* [programming]. On some machines (e.g., PCs), floating-point division can be as much as twenty times slower than multiplication. Modify the `InPoly1` code to avoid the division in Equation (7.9), and time it on examples to see if there is a significant difference on your machine.
4. *Tessellation by polygons.* Argue that in a partition of a region of the plane into polygons, `InPoly0` classifies a point q "in" at most one polygon.
5. *Speed-up* [programming]. Speed up `InPoly0` (Code 7.12) by avoiding the computation of x whenever the straddling segment is on the negative side of the ray.
6. *Avoid translation* [programming; easy]. Develop a new version of `InPoly1` (Code 7.13) that avoids the unnecessary translation of P and q.
7. *Integer ray crossing* [programming].
   a. Modify `InPoly0` (Code 7.12) to avoid the sole floating-point calculation. Use `AreaSign` (Code 4.23).
   b. Use the `AreaSign` results to decide if q lies on an edge, thereby achieving the functionality of `InPoly1` without shooting rays in both directions.
8. *Winding number* [programming]. Implement the winding number algorithm. The basic routine required is the angle subtended by a polygon edge e_i from the point q. The angle θ_i can be found from the cross product: $v_i \times v_{i+1} = |v_i||v_{i+1}| \sin\theta_i$. Recall that `Area2( q, P[i], P[i+1] )` from Chapter 1 (Code 1.5) is the magnitude of this cross product. The lengths of the vectors must then be divided out, the arcsine computed, and the angles summed over all i.

   Develop code for computing this angle sum. Pay particular attention to the range of angles returned by the `asin` library function, remembering that all counterclockwise turns must be positive angles, and clockwise turns negative angles. Decide what should be done when $|v_i| = 0$ or $|v_{i+1}| = 0$.

7.5. POINT IN POLYHEDRON

Determining whether a point is inside a polyhedron has many applications, including collision detection: Determining if a moving point (e.g., the tip of a tool) has penetrated an object in its environment is an instance. As in two dimensions, the problem is easy if the polyhedron is convex; in fact, the convex hull code in Chapter 4 solves this problem as the first step in `AddOne` (Code 4.15). The nonconvex case admits the same two solutions as in two dimensions: the generalization of the winding number computation and counting ray crossings.

Solid Angles

It is perhaps surprising that the winding number idea works in three dimensions as well as in two. It depends on a notion of signed *solid angle*, a measure of the fraction of a sphere surface consumed by a cone apexed at a point. It is measured in "steradians," which assigns 4π to the full-sphere angle. The solid angle of a tetrahedron with apex q and base T is the surface area of the unit sphere S falling within the tetrahedron when q is placed at the center of S, and the faces incident to q are extended (if necessary) to cut through S. The sign of the angle depends on the orientation of T. If the solid angles formed by q and every face of a polyhedron P are summed, the result is 4π if $q \in P$ and zero of $q \notin P$. This provides an elegant algorithm for point in polyhedron, which, alas, suffers the same pragmatic flaws as its two-dimensional counterpart: It is subject to numerical errors, and it is slow. A timing comparison between the ray-crossing code to be presented below and an implementation of the solid angle approach (Carvalho & Cavalcanti 1995) showed the latter to be twenty-five times slower. Their code is, however, much shorter.

Ray Crossings

The logic behind the ray-crossing algorithm in three dimensions is identical to that for the two-dimensional version: q is inside iff a ray from q to infinity crosses the boundary of P an odd number of times. A ray to infinity can be effectively simulated by a long segment qr, long enough so that r is definitely outside P. As we have worked out segment–triangle intersection in Section 7.3, it would seem easy to count ray crossings. The problematic aspect of this approach is to develop a scheme to count crossings accurately in the presence of the wide variety of possible degeneracies that could occur: qr could lie in a face of P, could hit a vertex, could collinearly overlap with an edge, hit an edge transversely, etc. It seems a proper accounting could be made, but I am not aware of any attempt in this direction. I leave this as an open problem (Exercise 7.5.2[1]).

Here we proceed on the basis of two observations. First, the crossings of a ray without degeneracies, which, for each face f of P, either misses f entirely or passes through a point in its relative interior, are easily counted. Second, "most" rays are nondegenerate in this sense, so a random ray is likely to be nondegenerate. Our plan is then to generate a random ray and check for degeneracies. If there are none, the crossing count answers the query. If a degeneracy is found, the ray is discarded and another random one chosen. Degeneracies can be detected with the `SegTriInt` code developed in Section 7.3 (Code 7.11). This leads to the pseudocode shown in Algorithm 7.1.

We now discuss the generation of the random ray. Let D be the the length of the diagonal of the smallest coordinate-aligned[13] "bounding box" B surrounding P. D is easily computed from the maximum and minimum coordinates of the vertices of P. Let $R = \lceil D \rceil + 1$. If a query point q is outside B, then it is outside P. For any query point inside B, a ray of length R from q must reach outside B (because D is the largest separation between any two points within B) and therefore outside P. We will use this value of R to guarantee that our query ray/segment qr reaches strictly outside P.

Generation of random rays of length R can be viewed as generating random points on the surface of a sphere of radius R. This is a well-studied problem, which we will not explore here.[14] The code `sphere.c` distributed with this book, and used to

[13] A coordinate-aligned object is often called *isothetic*.

[14] See O'Rourke (1997), Shoemake (1992), Arvo (1991), and Knuth (1969, p. 116, 485).

produce the 10,000 points in Figure 4.15, implements one method of generating such points. We will employ that as part of our point-in-polyhedron code without detailing it further.

Algorithm: POINT IN POLYHEDRON
Compute bounding radius R.
loop forever
    r_0 = random ray of length R.
    $r = q + r_0$.
    crossings = 0.
    for each triangle T of polyhedron P do
        `SegTriInt`(T, q, r).
        if degenerate intersection
            then Go back to loop.
            else Increment *crossings* appropriately.
    if *crossings* odd
        then q is inside P.
        else q is outside P.
    Exit.

Algorithm 7.1 Point in Polyhedron.

We make one last point before proceeding to code for the entire algorithm. The for-loop of Algorithm 7.1 calls `SegTriInt` for each face of P. Not only does the ray miss most faces of P, it misses them by a wide margin. This is a situation that calls for a quick miss-test, one that does not do all the (considerable) computation of `SegTriInt`. We will include in our implementation a very simple bounding-box test, as follows: As each face is read in (by `ReadFaces`, a simple routine not shown), a bounding box is computed and stored as two minimum and maximum corner points `tPointi Box[PMAX][2]`. Before committing to the full intersection test with face f, we first see if the query ray qr lies entirely to one side of one of the six faces of the box bounding f with a call to `BoxTest(f,q,r)` (Code 7.14). This returns '0' when nonintersection is guaranteed for this reason, and '?' otherwise. My testing shows that this simple rejection handles more than half of the intersection checks, well worth the slight overhead on the remaining half.

We now present `InPolyhedron`, code for testing if a query point q is in a polyhedron. We design it to return a code as follows:

'V': q coincides with a Vertex of P.
'E': q is in the relative interior of an Edge of P.
'F': q is in the relative interior of a Face of P.
'i': q is strictly interior to P.
'o': q is strictly exterior (outside) to P.

The codes {`V`, `E`, `F`} are inherited from the same codes returned by `SegTriInt`: Because we have ensured that r is strictly outside P, if qr has an endpoint on a vertex, edge, or face of P, then it must be the q endpoint. Thus we distinguish the on-boundary cases for

q without further effort. The codes {i, o} are distinguished by the parity of the crossings counter.

```
char BoxTest ( int n, tPointi a, tPointi b )
{
  int i;  /* Coordinate index */
  int w;

  for ( i=0; i < DIM; i++ ) {
      w = Box[ n ][0][i];  /* min: lower left */
      if ( (a[i] < w) && (b[i] < w) ) return '0';
      w = Box[ n ][1][i];  /* max: upper right */
      if ( (a[i] > w) && (b[i] > w) ) return '0';
  }
  return '?';
}
```

Code 7.14 `BoxTest`.

The overall structure of the main procedure `InPolyhedron` is shown in Code 7.15. First, query points outside the bounding box for P lead to an immediate return after the `InBox( q, bmin, bmax )` test; the simple code for `InBox` is not shown. Then a near-infinite loop adds a random ray to q to get r. (An upper limit is placed on the number of repetitions just as a matter of programming practice.) Next, qr is tested against every face of P, with a for-loop whose body is displayed separately (Code 7.16). If the for-loop runs to completion, then we are certain that the ray is generic, and the parity of `crossings` determines the result.

The for-loop (Code 7.16) first uses `BoxTest` hoping for a quick conclusion that the ray misses f, as discussed above. Otherwise `SegTriInt` is called and its return code used for subsequent decisions. Only if the code is 'f' (qr intersects the face in its relative interior) is the crossings counter incremented. The three codes {p, v, e} all indicate degeneracies: The ray lies in the plane of f or passes through a vertex or edge of f. We do not need further distinctions within the in-plane case 'p': Even if the ray misses f entirely, we are still safe in rejecting this as a degenerate case and awaiting a "better" ray. For all these degeneracies, the for-loop is abandoned, with control returning back to the infinite while-loop for another random ray. The codes {V, E, F} allow immediate exit, as discussed before.

Example: Cube. A simple example is shown in Figure 7.9. Here $q = (5, 5, 5)$ is at the center of a $10 \times 10 \times 10$ cube of 12 triangular faces. With $D = \sqrt{300}$, the ray radius is $R = 19$. The call to `RandomRay` results (in one particular trial) to $r = (23, 6, 11)$, which is well outside of P. The test against each of the 12 faces leads to 8 decided by `BoxTest` and 4 calls to `SegTriInt`, only one of which returns '1'. Thus there is exactly one ray crossing, and q is determined to be inside.

```
char InPolyhedron( int F, tPointi q,
     tPointi bmin, tPointi bmax, int radius )
{
   tPointi r;   /* Ray endpoint. */
   tPointd p;   /* Intersection point; not used. */
   int f, k = 0, crossings = 0;
   char code = '?';

   /* If query point is outside bounding box, finished. */
   if ( !InBox( q, bmin, bmax ) )
      return 'o';

   LOOP:
   while( k++ < F ) {
      crossings = 0;

      RandomRay( r, radius );
      AddVec( q, r, r );

      for ( f = 0; f < F; f++ ) { /* Begin check each face */
            /* Intersect ray with face f and increment crossings: see BELOW. */
      } /* End check each face */

      /* No degeneracies encountered: ray is generic, so finished. */
      break;

   } /* End while loop */

   /* q strictly interior to polyhedron iff an odd number of crossings. */
   if( ( crossings % 2 ) == 1 )
      return   'i';
   else return 'o';
}
```

Code 7.15 `InPolyhedron`. (`AddVec` is similar to `SubVec` in Code 7.6.)

Example: Nonconvex Polyhedron. A more stringent test is provided by the polyhedron P of $V = 400$ vertices and $F = 796$ (triangle) faces shown in Figure 7.10. Performance of the code was tested by generating random query points within the bounding box of P.[15] Out of 1,000,000 random rays generated, 8,121 (0.8%) were degenerate and caused the while-loop to try again. In 110 cases (0.01%) the loop again failed, and in only one instance of the one million trials did the loop generate three random rays before finding a generic one. Although the polyhedron can hardly be said to be "typical" (whatever that might mean), I do not expect performance to be significantly worse than this 99% "hit rate."

[15]The code `cube.c` that produced Figure 4.14 was used to generate the query points.

```
for ( f = 0; f < F; f++ ) {   /* Begin check each face */
    if ( BoxTest( f, q, r ) == '0' )
         code = '0';
    else code = SegTriInt( Faces[f], q, r, p );

  /* If ray is degenerate, then goto outer while to generate another. */
  if ( code == 'p' || code == 'v' || code == 'e' ) {
     printf("Degenerate ray\n");
     goto LOOP;
  }

  /* If ray hits face at interior point, increment crossings. */
  else if ( code == 'f' ) {
  crossings++;
  printf( "crossings = %d\n", crossings );
  }

  /* If query endpoint q sits on a V/E/F, return that code. */
  else if ( code == 'V' || code == 'E' || code == 'F' )
     return( code );

  /* If ray misses triangle, do nothing. */
  else if ( code == '0' )
     ;

  else
  fprintf( stderr, "Error, exit(EXIT_FAILURE)\n" ),
  exit (1);

  /* End check each face */
  }
```

Code 7.16 For-loop of `InPolyhedron`.

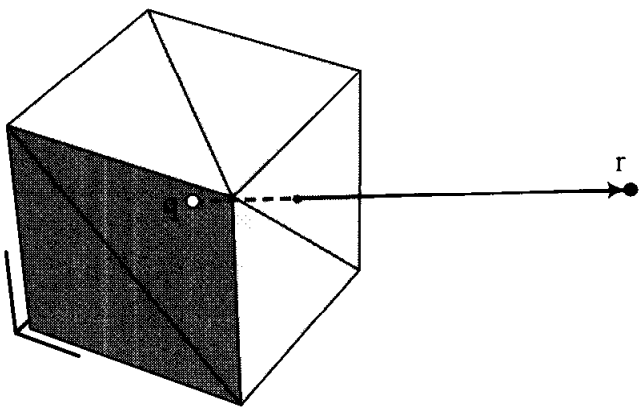


FIGURE 7.9 The ray qr intersects $\triangle(10, 10, 10), (10, 0, 10), (10, 10, 0)$ in its interior.

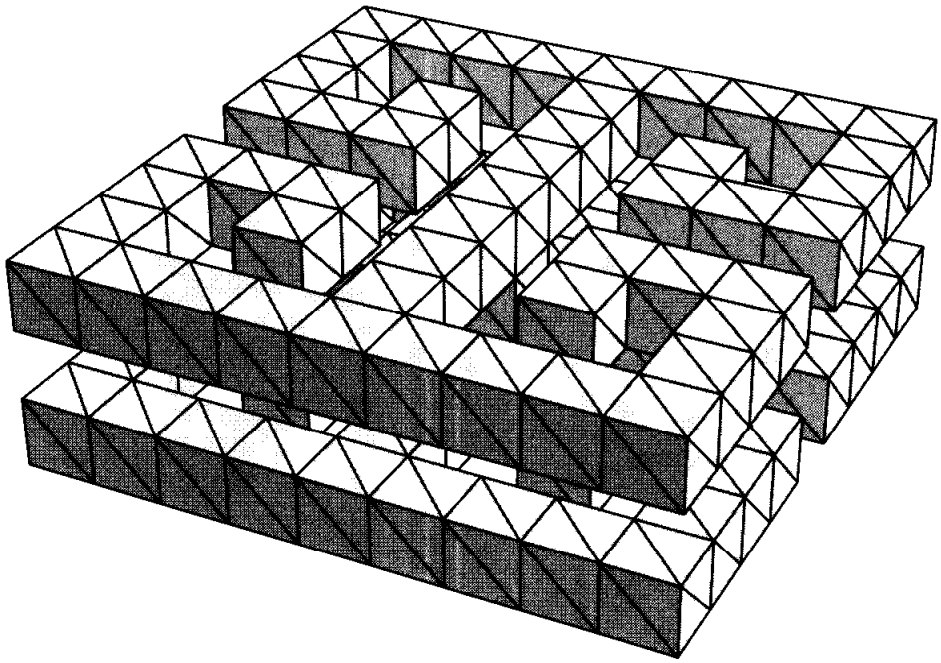


FIGURE 7.10 A polyhedron of $(V, E, F) = (400, 1194, 796)$ vertices, edges, and faces. The top and bottom "layers" are identical, connected by a single cubical channel in the middle layer. The polyhedron is symmetric about all three coordinate axis planes through the center of gravity.

7.5.1. Analysis

Algorithm 7.1 runs in expected time $O(\rho n)$, where ρ is the expected number of iterations before the while-loop finds a generic ray. Although we have just seen that for one combinatorially dense sample polyhedron, $\rho \approx 1.01$, it would be reassuring to prove a theoretical bound. I have not performed an exact analysis (Exercise 7.5.2[2]) but will offer an argument to show that $\rho = 1 + \epsilon$ can be achieved for any $\epsilon > 0$.

We start with two simplifying observations. First, it is easier to analyze random rays whose integer-coordinate tips fall on the surface of a surrounding cube C rather than a surrounding sphere. This is no loss of generality, as we could alter the implementation to follow this less aesthetically pleasing strategy (or choose a sphere large enough to include rays to all the cube surface points). Let each edge of the cube have length L.

Second, we need only concern ourselves with a degeneracy between q and an edge e of P. If q lies in the plane of a face, then there are rays qr that have a q–e degeneracy with edges of the face; and if a ray from q passes through a vertex, it passes through each (closed) incident edge. So let us just concentrate on one edge e of the polyhedron.

If e is close enough to q, then it "projects" to a segment that cuts completely across a face of the bounding cube C, as illustrated in Figure 7.11. In the worst case, the segment renders L integer points on that face of C unusable as ray tips, in the sense that they lead to degenerate rays.[16] If P has E edges, then at most EL points of a face of C can be rendered unusable. In effect, the edges of P produce an arrangement of lines on the

[16]"Renders" is particularly apropos here, because a line is rendered on a raster display by turning on L pixels. See Foley et al. (1990, Sec. 3.2).

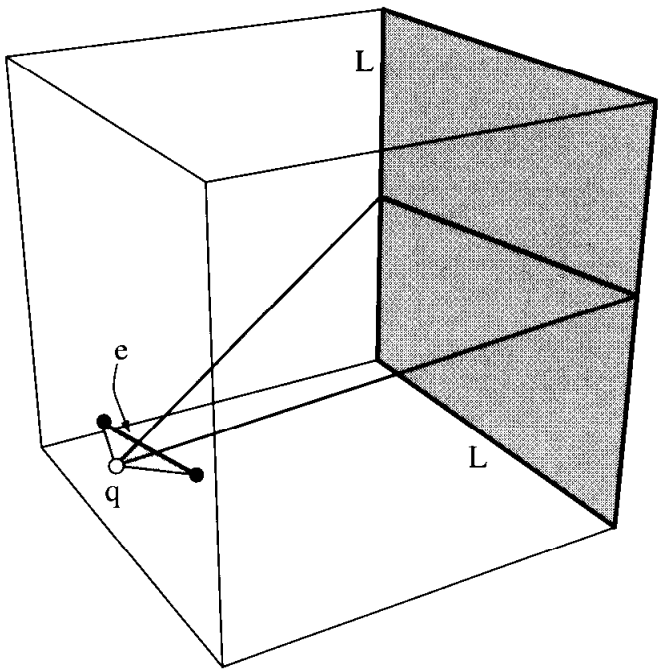


FIGURE 7.11 Edge e "kills" a line of points on the face of the surrounding cube in the sense that any r on that line makes a ray qr degenerate with (pass through) e.

cube face; only rays that miss the arrangement are safely generic. But there are many such, because this cube face contains L^2 integer points. Thus the probability of hitting a degeneracy is at most $EL/L^2 = E/L$. Because E is a constant (1,194 in Figure 7.10), choosing L large enough guarantees any $\epsilon = E/L$ desired.

There is pragmatic pressure in the other direction, however: The larger L is, the smaller the safe range of vertex coordinates before the onset of overflow problems. In practice I have found it best to choose R (which corresponds here to $L/2$) as small as possible, just 1 more than the box diagonal D.

7.5.2. Exercises

1. *Count degenerate crossings* [open]. Work out a scheme that counts ray crossings for any ray, taking into account all possible types of degeneracy. The parity of the count should determine if the point is in or out of the polyhedron. Test by fixing the qr ray in `InPolyhedron` (Code 7.15) to be parallel to the x axis.
2. *Sphere analysis* [open]. Compute a bound on the probability that an integer-coordinate ray tip on the surface of a "digital sphere" will degenerately pass through a vertex, edge, or face of the enclosed polyhedron P. Express your answer as a function of the sphere radius R, the diagonal D of a box surrounding P, and the combinatorial complexity of P (V, E, and F).

7.6. INTERSECTION OF CONVEX POLYGONS

The intersection of two arbitrary polygons of n and m vertices can have quadratic complexity, $\Omega(nm)$: the intersection of the polygons in Figure 7.12 is 25 squares. But the intersection of two convex polygons has only linear complexity, $O(n+m)$. Intersection of convex polygons is a key component of a number of algorithms, including determining whether two sets of points are separable by a line and for solving two-variable

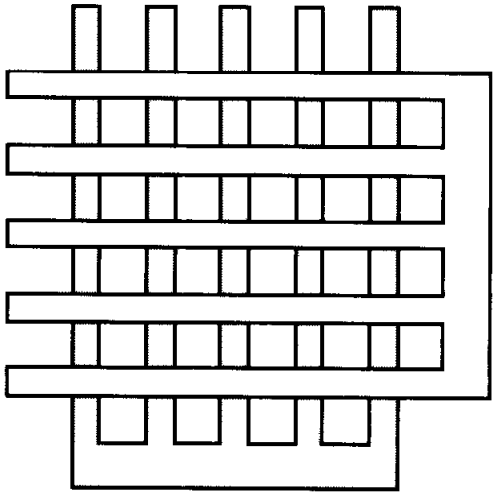


FIGURE 7.12 The intersection of two polygons can have quadratic complexity.

linear programming problems (Shamos 1978). The first linear algorithm was found by Shamos (1978), and since then a variety of different algorithms have been developed, all achieving $O(n + m)$ time complexity. This section describes one that I developed with three undergraduates, an amalgamation of their solutions to a homework assignment (O'Rourke, Chien, Olson & Naddor 1982). I feel it is the simplest algorithm available, but this is hardly an objective opinion.

The basic idea of the algorithm is straightforward, but the translation of the idea into code is somewhat delicate (as is often the case). Assume the boundaries of the two polygons P and Q are oriented counterclockwise as usual, and let A and B be directed edges on each. The algorithm has A and B "chasing" one another, adjusting their speeds so that they meet at every crossing of ∂P and ∂Q. The basic structure is as shown in Algorithm 7.2. A "movie" of the algorithm in action is shown in Figure 7.13.[17] The edges A and B are shown as vectors in the figure. The key clearly lies in the rules for advancing A and B, to which we now turn.

Algorithm: INTERSECTION OF CONVEX POLYGONS
Choose A and B arbitrarily.
repeat
    if A intersects B then
        Check for termination.
        Update an *inside* flag.
    Advance either A or B,
        depending on geometric conditions.
until both A and B cycle their polygons
Handle $P \cap Q = \emptyset$ and $P \subset Q$ and $P \supset Q$ cases.

Algorithm 7.2 Intersection of convex polygons.

[17]This figure was inspired by the animation of this algorithm provided in XYZ GeoBench.

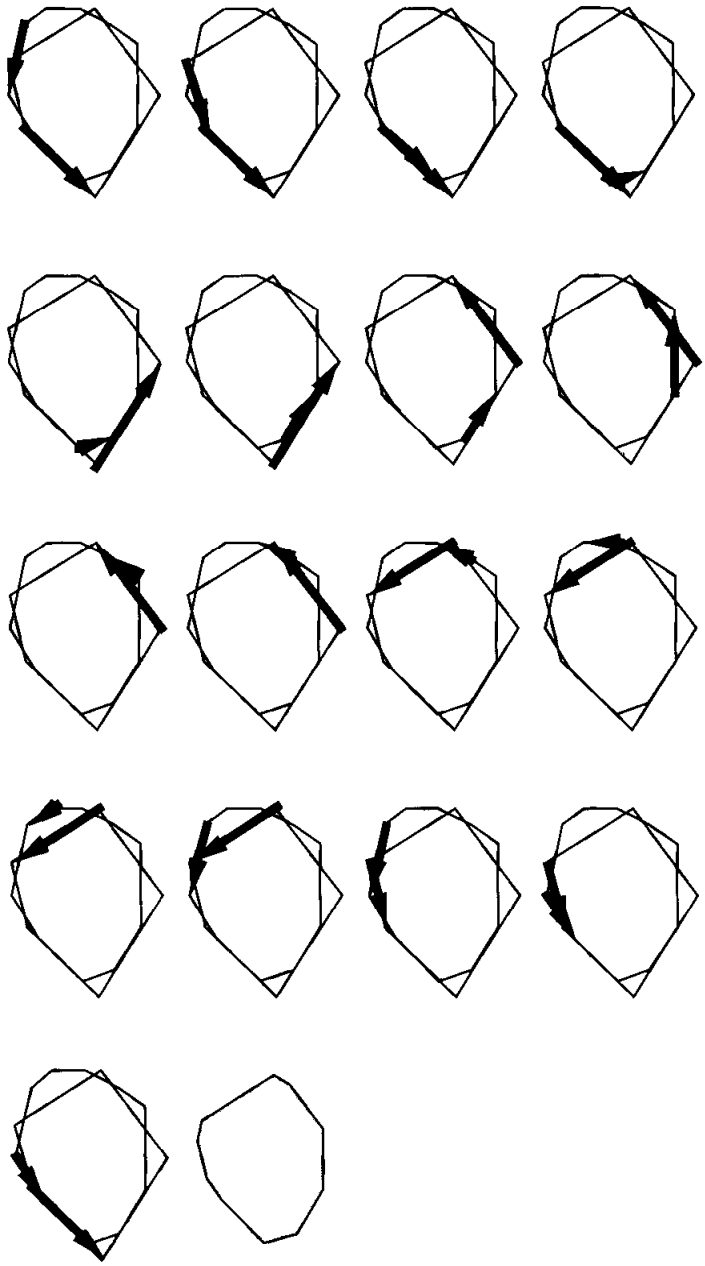


FIGURE 7.13 Snapshots of polygon intersection algorithm, sequenced left to right, top to bottom. This example is explored in more detail in Section 7.6.1.

Let a be the index of the head of A, and b the head of B. If B "aims toward" the line containing A, but does not cross it (as do all the solid vectors in Figure 7.14), then we want to advance B in order to "close in" on a possible intersection with A. This is the essence of the advance rules. The situations in the figure can be captured as follows: Let $H(A)$ be the open halfplane to the left of A. I will use the notation "$A \times B > 0$" to mean that the z coordinate of the cross product is > 0 (recall that this means that the shortest turn of A into B is counterclockwise):

if $A \times B > 0$ and $b \notin H(A)$, or
if $A \times B < 0$ and $b \in H(A)$,
then advance B.

(Let us ignore for the moment collinearities of $P[a]$ with B or $P[b]$ with A.) A similar

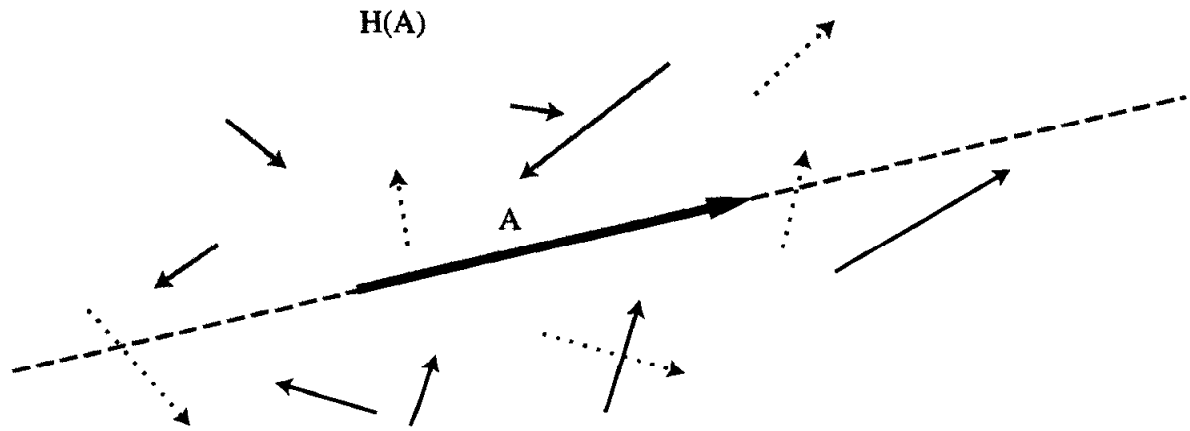


FIGURE 7.14 All the solid B vectors "aim" toward A; none of the dotted vectors do.

rule applies with the roles of A and B reversed (recall that $B \times A = -A \times B$):

if $A \times B < 0$ and $a \notin H(B)$, or
if $A \times B > 0$ and $a \in H(B)$,
then advance A.

If both vectors aim toward each other, either may be advanced. When neither A nor B aim toward the other, we advance whichever is outside the halfplane of the other or either one if they are both outside. It takes some thought to realize that if both $a \in H(B)$ and $b \in H(A)$, then one must aim toward the other; so the above rules cover all cases. The cases may be organized in the following table:

$A \times B$	$a \in H(B)$	$b \in H(A)$	**Advance Rule**
>0	T	T	A
>0	T	F	A or B
>0	F	T	A
>0	F	F	B
<0	T	T	B
<0	T	F	B
<0	F	T	A or B
<0	F	F	A

These rules are realized by the following condensation, which exploits the freedom in the entries that are arbitrary.

$A \times B$	**Halfplane Condition**	**Advance Rule**
>0	$b \in H(A)$	A
>0	$b \notin H(A)$	B
<0	$a \in H(B)$	B
<0	$a \notin H(B)$	A

These are the advance rules we use below.

7.6.1. Implementation

The core of the implementation is a do-while loop that implements the advance rules in the above table. Although the translation of the table is straightforward, I have not found a concise way to handle all the peripheral issues surrounding this core, which threaten to overwhelm the heart of the algorithm. We will discuss the generic cases before turning to special cases.

The polygon vertices are stored in two arrays P and Q, indexed by a and b, with n and m vertices, respectively. The three main geometric variables on which decisions are based are all computed with `AreaSign` (the sign-version of `Area2`): $A \times B$ is provided by `AreaSign( O, A, B )`, where `O` is the origin; $a \in H(B)$ is `AreaSign( Q[b1], Q[b], P[a] )`, with $b_1 = (b-1) \bmod n$; $b \in H(A)$ is a similar expression.

Code 7.17 shows the local variables, initialization, and overall structure. The central do-while is shown in Code 7.18, for generic cases only (corresponding to the above table). The special case Code 7.21 will be discussed later.

In Code 7.17, the variable `inflag` is an enumerated type that keeps track of which polygon is currently "inside"; it takes on one of the three values `{Pin, Qin, Unknown}`. Before the first intersection, its value is `Unknown`. After a crossing is detected between A and B, `inflag` remembers which one is locally inside just beyond the intersection point. This flag is used to output appropriate polygon vertices as the edge vectors are advanced. If `inflag` remains `Unknown` throughout a cycling of the counters, we know ∂P and ∂Q do not (properly) cross, and either they do not intersect, or they intersect only at a point, or one contains the other.

Termination of the loop is conceptually simple but tricky in practice. One could await the return of the first output point (Exercise 7.6.2[1]), but the version shown bases termination on the edge vector indices: When both a and b have cycled around their polygons, we are finished. In some cases of degenerate intersection, one of the indices does not cycle; so the while-statement also terminates when either has cycled twice after the first intersection, when the counters `aa` and `ba` (the suffix `a` stands for "advances") are reset.

Aside from the initialization details, the basic operations within the loop (Code 7.18) are: Intersect the A and B edge vectors with `SegSegInt`, print the intersection point p and toggle the `inflag` if they do intersect by a call to `InOut`, and finally, advance either a or b according to the advance rules, and perhaps print a vertex, by a call to `Advance`.

An intersection is considered to have occurred when `SegSegInt` returns a code of either '1' or 'v'; the code 'e,' indicating collinear overlap, will not toggle the flag, nor produce any output, except in a special case considered later. `InOut` (Code 7.19) prints the point of intersection and then bases its decision on how to set `inflag` according to which edge vector's head is inside the other vector's half plane. If the situation is not determined, the flag is not toggled.

The `Advance` routine (Code 7.20) advances the counters (`a` by return, `aa` by side effect) and prints out the vertex just passed if it was `inside`. Note that when `inflag`

```
void    ConvexIntersect( tPolygoni P, tPolygoni Q,
                         int n, int m )
                           /* P has n vertices, Q has m vertices. */
{
  int      a, b;           /* indices on P and Q (resp.) */
  int      a1, b1;         /* a-1, b-1 (resp.) */
  tPointi  A, B;           /* directed edges on P and Q (resp.) */
  int      cross;          /* sign of z-component of A x B */
  int      bHA, aHB;       /* b in H(A); a in H(b). */
  tPointi  Origin = {0,0}; /* (0, 0) */
  tPointd  p;              /* double point of intersection */
  tPointd  q;              /* second point of intersection (for 'e') */
  tInFlag  inflag;         /* {Pin, Qin, Unknown}: which inside */
  int      aa, ba;         /* # advs. on a & b indices (after 1st inter.) */
  bool     FirstPoint;     /* Is this first point? (used to initialize). */
  tPointd  p0;             /* The first point. */
  int      code;           /* SegSegInt return code. */

  /* Initialize variables. */
  a = 0; b = 0; aa = 0; ba = 0;
  inflag = Unknown; FirstPoint = TRUE;

  do {

      /* BODY of do-while: see part (b) below. */

  /* Quit when both adv. indices have cycled, or one has cycled twice. */
  } while ( ((aa < n) || (ba < m)) && (aa < 2*n) && (ba < 2*m) );

  if ( !FirstPoint )   /* If at least one point output, close up. */
      LineTo( p0 );

  /* Deal with remaining special cases: not implemented. */
  if ( inflag == Unknown)
      printf("The boundaries of P and Q do not cross.\n");
}
```

Code 7.17 `ConvexIntersect`, part (a): top-level structure.

is set to, for example, `Pin`, it is not known that a is actually inside; only that just beyond the point of intersection, A is inside (except in a special case). But by the time `Advance` increments a, it is known that a is truly in the intersection.

Before proceeding to a discussion of special cases, we show the output of the code on a relatively generic example, shown in Figure 7.15. The code produces the following

```
do {
   /* Computations of key variables. */
   a1 = (a + n - 1) % n;
   b1 = (b + m - 1) % m;

   SubVec( P[a], P[a1], A );
   SubVec( Q[b], Q[b1], B );

   cross = AreaSign( Origin, A, B );
   aHB   = AreaSign( Q[b1], Q[b], P[a] );
   bHA   = AreaSign( P[a1], P[a], Q[b] );

   /* If A & B intersect, update inflag. */
   code = SegSegInt( P[a1], P[a], Q[b1], Q[b], p );
   if ( code == '1' || code == 'v' ) {
      if ( inflag == Unknown && FirstPoint ) {
         aa = ba = 0;
         FirstPoint = FALSE;
         p0[X] = p[X]; p0[Y] = p[Y];
         MoveTo_d( p0 );
      }
      inflag = InOut( p, inflag, aHB, bHA );
   }
   /*-----Advance rules-----*/
   /* SPECIAL CASES: see part (c) below */
   /* Generic cases. */
   else if ( cross >= 0 ) {
       if ( bHA > 0)
          a = Advance( a, &aa, n, inflag == Pin, P[a] );
       else
          b = Advance( b, &ba, m, inflag == Qin, Q[b] );
   }
   else /* if( cross < 0 ) */ {
       if ( aHB > 0)
          b = Advance( b, &ba, m, inflag == Qin, Q[b] );
       else
          a = Advance( a, &aa, n, inflag == Pin, P[a] );
   }
} while ( ((aa < n) || (ba < m)) && (aa < 2*n) && (ba < 2*m) );
```

Code 7.18 `ConvexIntersect`, part (b): **do-while** loop. `MoveTo_d` prints a Postscript "moveto" command for `double` coordinates.

```
tInFlag InOut( tPointd tp, tInFlag inflag, int aHB, int bHA )
{
  LineTo_d( p );

  /* Update inflag. */
  if       ( aHB > 0)
       return Pin;
  else if ( bHA > 0)
       return Qin;
  else     /* Keep status quo. */
       return inflag;
}
```

Code 7.19 `InOut`. `LineTo_d` prints a Postscript "lineto" command.

```
int    Advance( int a, int *aa, int n, bool inside, tPointi v )
{
  if ( inside )
     LineTo_i( v );
  (*aa)++;
  return (a+1) % n;
}
```

Code 7.20 `Advance`. `LineTo_i` prints a "lineto" command for `int`s.

Postscript output for these polygons:

```
 5.00     8.00  moveto
 5.00     8.00  lineto
13.00     0.00  lineto
19        2     lineto
24       10     lineto
24.00    21.33  lineto
17.80    29.60  lineto
14.67    31.17  lineto
 1.62    23.01  lineto
 0.72    19.12  lineto
 2.50    12.00  lineto
 5.00     8.00  lineto
```

The "movie" in Figure 7.13 shows the progression of A and B for this example. The example is not entirely generic, in that edge $(1, 2)$ of P collinearly overlaps with $(4, 0)$ of Q.

The alternation between integers and reals in the output reflects whether the point is a vertex (and printed in `Advance`) or a computed intersection point (and printed in `InOut`). The repeats of the point $(5, 8)$ at the beginning and end of the list are an artifact of Postscript initialization and closure and could be removed easily. Below we will see more pernicious duplication of points.

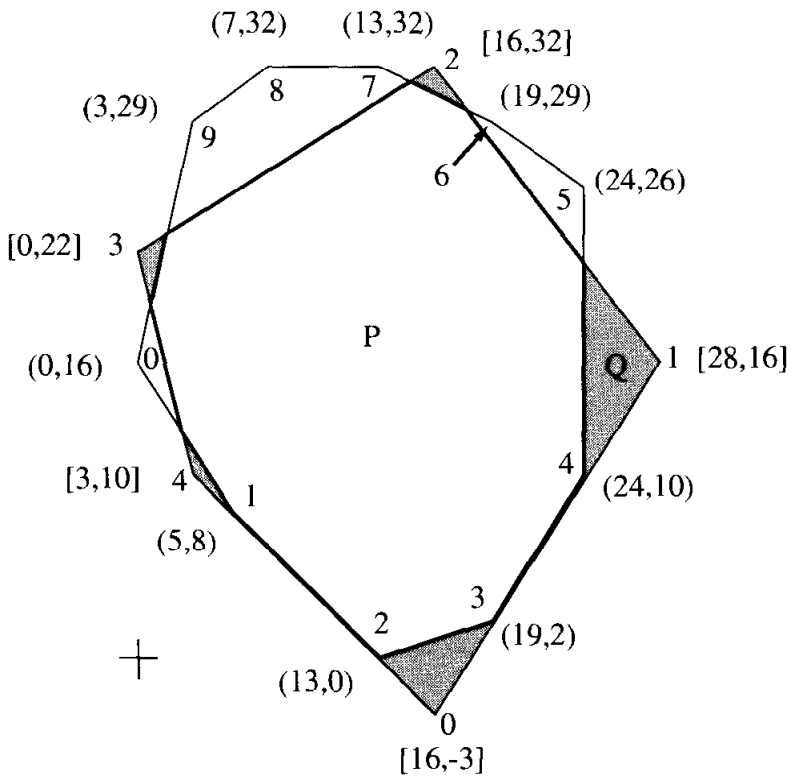


FIGURE 7.15 P is in front; Q is behind. P's vertex coordinates are in parentheses; Q's in brackets.

Special Cases

We finally turn to special cases, of which there seem to be a bewildering variety. See Figure 7.16 for a sampling of test examples.

The special cases all hinge on the special cases of the three geometric variables: $A \times B = 0$; when a lies on the line containing B; and when b lies on the line containing A. All three are indicated by returns of 0 from `AreaSign`. Several combinations are handled appropriately by the "generic cases" advance rules (Code 7.18), but three are isolated prior to that in Code 7.21:

1. If A and B overlap (return code 'e') and are oppositely oriented ($A \cdot B < 0$), then their overlap is the total intersection, for P and Q must lie on opposite sides of the line containing A and B. How the segment of intersection is found will be described in a moment.
2. If A and B are parallel ($A \times B = 0$) and a is strictly right of B and b is strictly right of A, then we may conclude $P \cap Q = \emptyset$.
3. If A and B are collinear (and case 1 above does not hold), then we advance one pointer, but we ensure that no point is output during the advance by arranging for the parameter `inside` to be `FALSE`.

Recall from Section 7.2 that our `SegSegInt` code returned only one point of intersection p even when the return code 'e' indicated overlap. Now that we need the actual overlap, we modify `SegSegInt` to return a second point q, computed in `ParallelInt`, such that pq is the segment of intersection. The modification to `ParallelInt` is shown in Code 7.22. Six possible cases of overlap are methodically identified, and p or q set appropriately: $cd \subseteq ab$; $ab \subseteq cd$; $c \in ab$ (two cases); and $d \in ab$ (two cases).

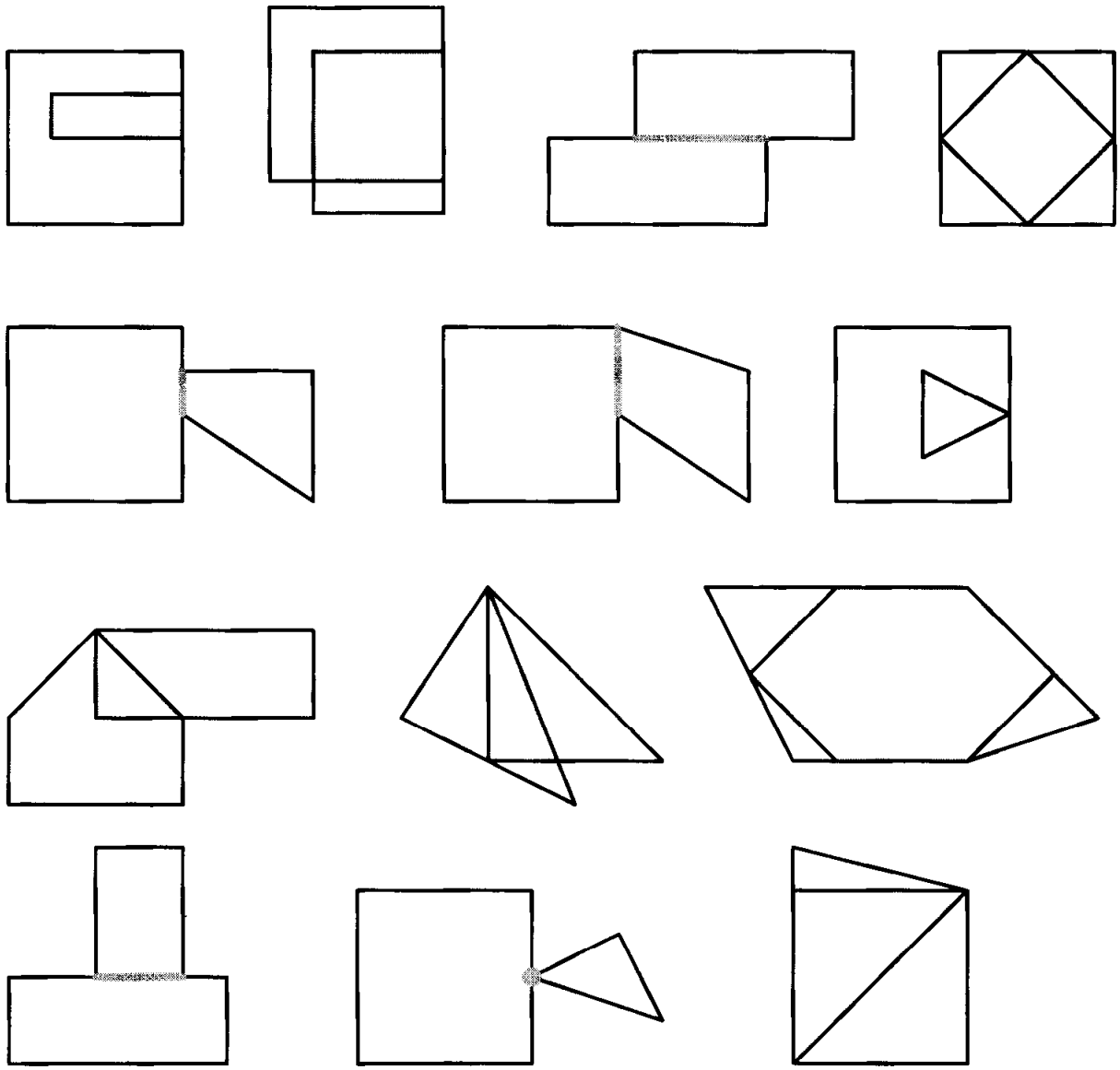


FIGURE 7.16 Output of the `ConvexIntersect` code on a variety of "degenerate" intersections. The intersection is shown shaded in each case.

We conclude with a litany of the weaknesses of the presented code and suggest improvements:

1. When the loop finishes with the `inflag` still `Unknown`, it could be that $P \subset Q$ or $Q \subset P$ or $P \cap Q = \emptyset$, or even that $P \cap Q = v$ where v is a vertex (the latter because `InOut` sometimes maintains the status quo). Handling these cases requires further code. None are all that difficult, but it would be preferable if they could be distinguished "automatically."
2. The loop termination, using two counters, is clumsy. Checking for a repeat of the first point is equally clumsy.
3. Although not evident from the example in Figure 7.15, degeneracies can cause points to be output more than once, sometimes both as vertices and as intersection points. For example, output from two nested squares sharing a corner included

```
/*....Advance rules (continued from part (a)) */

/* Special case: A & B overlap and oppositely oriented. */
if ( ( code == 'e' ) && (Dot( A, B ) < 0) )
      PrintSharedSeg( p, q ), exit(EXIT_SUCCESS);

/* Special case: A & B parallel and separated. */
if ( (cross == 0) && ( aHB < 0) && ( bHA < 0 ) )
      printf("%%P and Q are disjoint.\n"), exit(EXIT_SUCCESS);

/* Special case: A & B collinear. */
else if ( (cross == 0) && ( aHB == 0) && ( bHA == 0 ) ) {
        /* Advance but do not output point. */
        if ( inflag == Pin )
            b = Advance( b, &ba, m, inflag == Qin, Q[b] );
        else
            a = Advance( a, &aa, n, inflag == Pin, P[a] );
  }

/* Generic cases (continued in part (b) above)..... */
```

Code 7.21 `ConvexIntersect`, part (c): special cases.

this sequence:

```
0.00  50.00  lineto
0     50     lineto
0.00  50.00  lineto
```

Such duplicate points could wreak havoc with another program that expects all polygon vertices to be distinct. Although it is not difficult to suppress output of duplicates (Exercise 7.6.2[2]), this raises the integer versus float problem directly, for it will be necessary to decide, for example, if the integer 50 is equal to the floating-point number 50.00 where this latter number is computed with `doubles` in `SegSegInt`. An inexact calculation might result in identical points being considered distinct; use of a "fuzz" factor will inevitably enable acceptance of some distinct points as equal.

4. The biggest weakness is the need to handle many cases specially, often with delicate logic. It would be more satisfying to have the generic code specialize naturally. I leave this as an open problem (Exercise 7.6.2[4]).

7.6.2. Exercises

1. *Loop termination* (Peter Schorn) [programming]. Modify the code so that loop termination depends on cycling past the first output vertex, rather than on the loop counters `aa` and `ba`. Test your code on the two triangles (3, 4), (6, 4), (4, 7) and (4, 7), (2, 5), (6, 2).

```
char   ParallelInt( tPointi a, tPointi b, tPointi c, tPointi d,
                    tPointd p, tPointd q )
{
  if ( !Collinear( a, b, c) )
     return '0';

  if ( Between( a, b, c ) && Between( a, b, d ) ) {
     Assigndi( p, c );
     Assigndi( q, d );
     return 'e';
  }
  if ( Between( c, d, a ) && Between( c, d, b ) ) {
     Assigndi( p, a );
     Assigndi( q, b );
     return 'e';
  }
  if ( Between( a, b, c ) && Between( c, d, b ) ) {
     Assigndi( p, c );
     Assigndi( q, b );
     return 'e';
  }
  if ( Between( a, b, c ) && Between( c, d, a ) ) {
     Assigndi( p, c );
     Assigndi( q, a );
     return 'e';
  }
  if ( Between( a, b, d ) && Between( c, d, b ) ) {
     Assigndi( p, d );
     Assigndi( q, b );
     return 'e';
  }
  if ( Between( a, b, d ) && Between( c, d, a ) ) {
     Assigndi( p, d );
     Assigndi( q, a );
     return 'e';
  }
  return '0';
}
```

Code 7.22 `ParallelInt`; See Code 7.3 for `Assigndi`.

2. *Duplicate points* [programming]. Modify the code to suppress the output of duplicate points. Do not concern yourself with the float versus integer problem discussed above, but rather just compare `int`s and `double`s with ==, which will force a conversion to `double`s.

   Once you have your code working, try to find a case that will break it. This may be quite difficult, and even impossible on some machines.

3. *Rationals.*
   a. Discuss (but do not implement) how the point of intersection could be represented as an exact rational, a ratio of two integers.

b. Design a Boolean function that determines if two such rationals are equal. Note that, e.g., $2/6 = 127131/381393$.

c. [programming] Implement the rational equal function.

4. *Clean code* [open]. Design a set of advance rules that handle the special cases more cleanly than does the presented code. Ideally all the cases in Figure 7.16, as well as $P \subset Q$, $Q \subset P$, and $P \cap Q = \emptyset$, would be handled naturally.

7.7. INTERSECTION OF SEGMENTS

Although we have seen that $\Omega(n^2)$ is a lower bound on intersecting two polygons of n edges each, in many applications the worst case is rare. This suggests a goal of developing an output-size sensitive algorithm, one whose complexity depends on k, the size of (the number of vertices in) the output.[18] It turns out that the hard part of this task is a more general problem: finding the intersections among a collection of n segments in the plane. This is more general in that no assumption is made that the segments connect to form polygons. We will now pursue the segment intersection problem, presenting an elegant algorithm due to Bentley & Ottmann (1979), one of the first output-size sensitive algorithms in computational geometry, and return in Section 7.8 to the problem of intersecting polygons.

A brute-force intersection algorithm takes $\Omega(n^2)$ time: Check each segment against every other (using, e.g., `SegSegInt` from Section 7.2). To achieve output sensitivity, we want to compute intersections between only those pairs of segments that actually intersect. This goal sounds circular, but even this formulation carries a hint of a solution, for segments that intersect are close to one another – not throughout their length, but certainly at their point of intersection! If we could somehow "travel down" the lengths of a pair of segments until they become close to one another before deciding to intersect, we could achieve output sensitivity. Plane sweep provides the needed "travel down" mechanism.

Recall from Section 2.4 that a trapezoidalization could be computed efficiently by sweeping a horizontal line L over a collection of polygon edges, and only searching for chord intersections in the local neighborhood of a vertex hit by L. It is just this sort of local focus we need for the segment intersection problem.

Imagine sweeping the line L over a collection of segments $S = \{s_0, s_1, \ldots, s_{n-1}\}$. Let $x = s_i \cap s_j$ be an intersection point between two segments. Just before L reaches x, L pierces both s_i and s_j, and they are adjacent along L: No other segment is between them on L. Thus, at some time prior to every intersection "event" (when L crosses an intersection point), the intersecting segments are adjacent on L. This gives us the sought-for locality: Computing intersections between segments adjacent on L suffices to capture all intersection points. Some of these adjacent segments do not in fact intersect, but we will see that the "wasted effort" is small.

Let us make several simplifying assumptions to keep focused on the main idea: Assume no segment is horizontal and no three segments pass through one point. The plan is to sweep L over the segments, stopping at events of three types, when

[18] See Sections 3.3, 4.6.4, and 6.7.2 for other output-size sensitive algorithms.

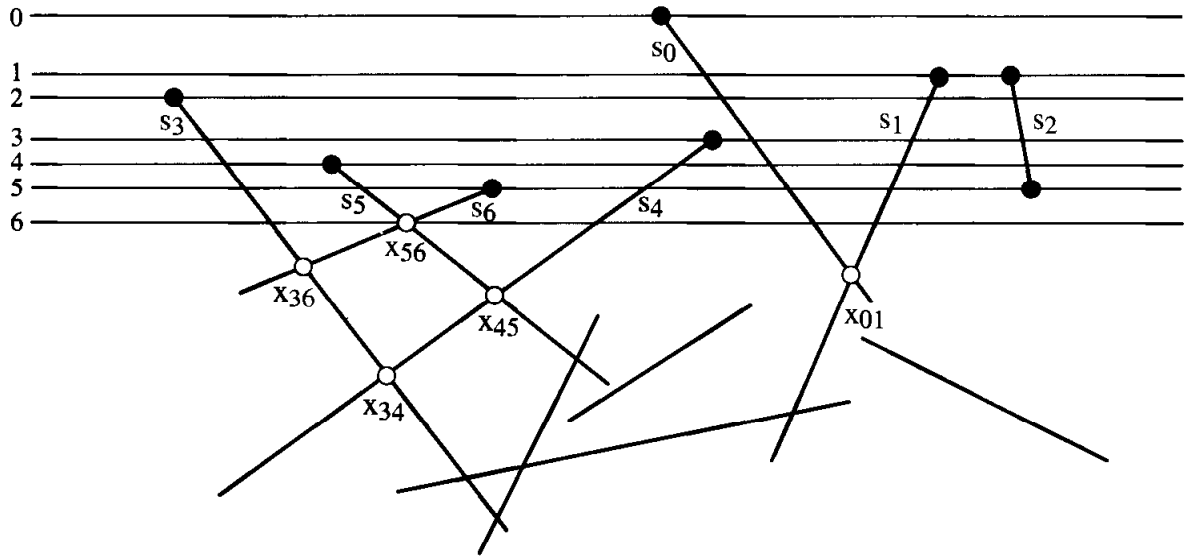


FIGURE 7.17 Bentley–Ottmann sweepline algorithm. Endpoint events are shown as solid circles; intersection points x_{ij} computed by position 6 of L are shown as open circles.

1. the top endpoint of a segment is hit,
2. the bottom endpoint of a segment is passed, or
3. an intersection point between two segments is reached.

All three of these events cause the list $\mathcal{L}$ of segments pierced by L to change: A segment is inserted, deleted, or two adjacent segments switch places, respectively. With each change, intersections between newly adjacent segments must be computed.

Although segments must become adjacent in $\mathcal{L}$ prior to their point of intersection x, it is not guaranteed that x is the next intersection event when it is computed. Rather, the intersection events must be placed in a queue Q sorted by height, along with the segment endpoints. An illustration should make the algorithm clear. Consider the set of segments $S = \{s_0, s_1, \ldots\}$ shown in Figure 7.17. Let a_i be the upper endpoint of segment s_i, and b_i its lower endpoint. Then the event queue is initialized to $Q = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, b_2, \ldots)$, all the segment endpoints sorted top to bottom. When L reaches a_1 and a_2 (position 1), s_0 and s_1 become newly adjacent, and their intersection point x_{01} is added to the queue after b_2. s_1 and s_2 are also newly adjacent but do not intersect. Note that the higher intersection point x_{56} has not yet been constructed. At position 2, L hits a_3; the newly adjacent segments s_3 and s_0 do not intersect. At this point $\mathcal{L} = (s_3, s_0, s_1, s_2)$. At position 3, L hits a_4, and intersection point x_{34} is added to Q at its appropriate location. Note that three intersection events "between" s_3 and s_4 will be encountered before x_{34} is reached. By the time L reaches the first intersection event at position 6, all the endpoints above have been processed, and $\mathcal{L} = (s_3, s_5, s_6, s_4, s_0, s_1)$. This event causes s_5 and s_6 to switch places in $\mathcal{L}$, introducing new adjacencies that result in x_{36} and x_{45} being added to Q. Q now contains all the circled intersection points shown in the figure.

The algorithm needs to maintain two dynamic data structures: one for $\mathcal{L}$ and one for Q. Both must support fast insertions and deletions in order to achieve an overall

low time complexity. We will not pursue the data structure details[19] but only claim that balanced binary trees suffice to permit both $\mathcal{L}$ and Q to be stored in space proportional to their number of elements m, with all needed operations performable in $O(\log m)$ time. We now argue that such structures lead to a time complexity for intersecting n segments of $O((n+k)\log n)$, where k is the number of intersection points between the segments. We will continue to assume that no three segments meet in one point.

The total number of events is $2n + k = O(n + k)$: the $2n$ segment endpoints and the k intersection points. Thus the length of Q is never more than this. Because each event is inserted once and deleted once from Q, the total cost of maintaining Q is $O((n + k)\log(n + k))$. Because $k = O(n^2)$, $O(\log(n + k)) = O(\log n + 2\log n) = O(\log n)$. Thus maintaining Q costs $O((n + k)\log n)$.

The total cost of maintaining $\mathcal{L}$ is $O(n \log n)$: n segments inserted and deleted at $O(\log n)$ each. It only remains to bound the number of intersection computations (each of which can be performed in constant time, by a call to `SegSegInt`, Code 7.2). Recall the earlier worry about "wasted effort." However, the number of intersection calls is at most twice the number of events, because each event results in at most two new segment adjacencies: an inserted segment with its new neighbors, two new neighbors when the segment between is deleted, and new left and right neighbors created by a switch at an intersection event. Thus the total number of intersection calls is $O(n + k)$.

The overall time complexity of the algorithm is therefore $O((n+k)\log n)$, sensitive to the output size k. We have seen that the space requirements are $O(n + k)$ because this is how long Q can grow. It turns out that this can be reduced to $O(n)$ (Exercise 7.8.1[2]). Moreover, both of these desirable complexities can be achieved without any of our simplifying assumptions (Exercise 7.8.1[1]).

These results were achieved by 1981 (Bentley & Ottmann 1979; Brown 1981), but more than a decade of further work was needed to reach an optimal algorithm in both time and space:

Theorem 7.7.1. *The intersection of n segments in the plane may be constructed in $O(n \log n + k)$ time* (Chazelle & Edelsbrunner 1992) *and $O(n)$ space* (Balaban 1995), *where k is the number of intersection points between the segments.*

Here k is not multiplied by $\log n$ as in the original Bentley–Ottmann algorithm. The practical difference may be slight, but closing the theoretical gap required the development of new techniques.

7.8. INTERSECTION OF NONCONVEX POLYGONS

It is not difficult to alter the Bently–Ottmann sweepline algorithm to compute the intersection of two polygons. Let the two polygons be A and B, with vertices labeled a_i and b_j respectively. The main idea is similar to that used by scan-line algorithms for filling (painting) a polygonal region on a graphics screen[20] and is related to our ray-crossing

[19] See de Berg, van Kreveld, Overmars & Schwarzkopf (1997, Sec. 2.1), Preparata & Shamos (1985, Sec. 7.2.3), or Mehlhorn (1984, Sec. VIII. 4.1).

[20] See, e.g., Foley, van Dam, Feiner, Hughes & Phillips (1993, Sec. 3.5).

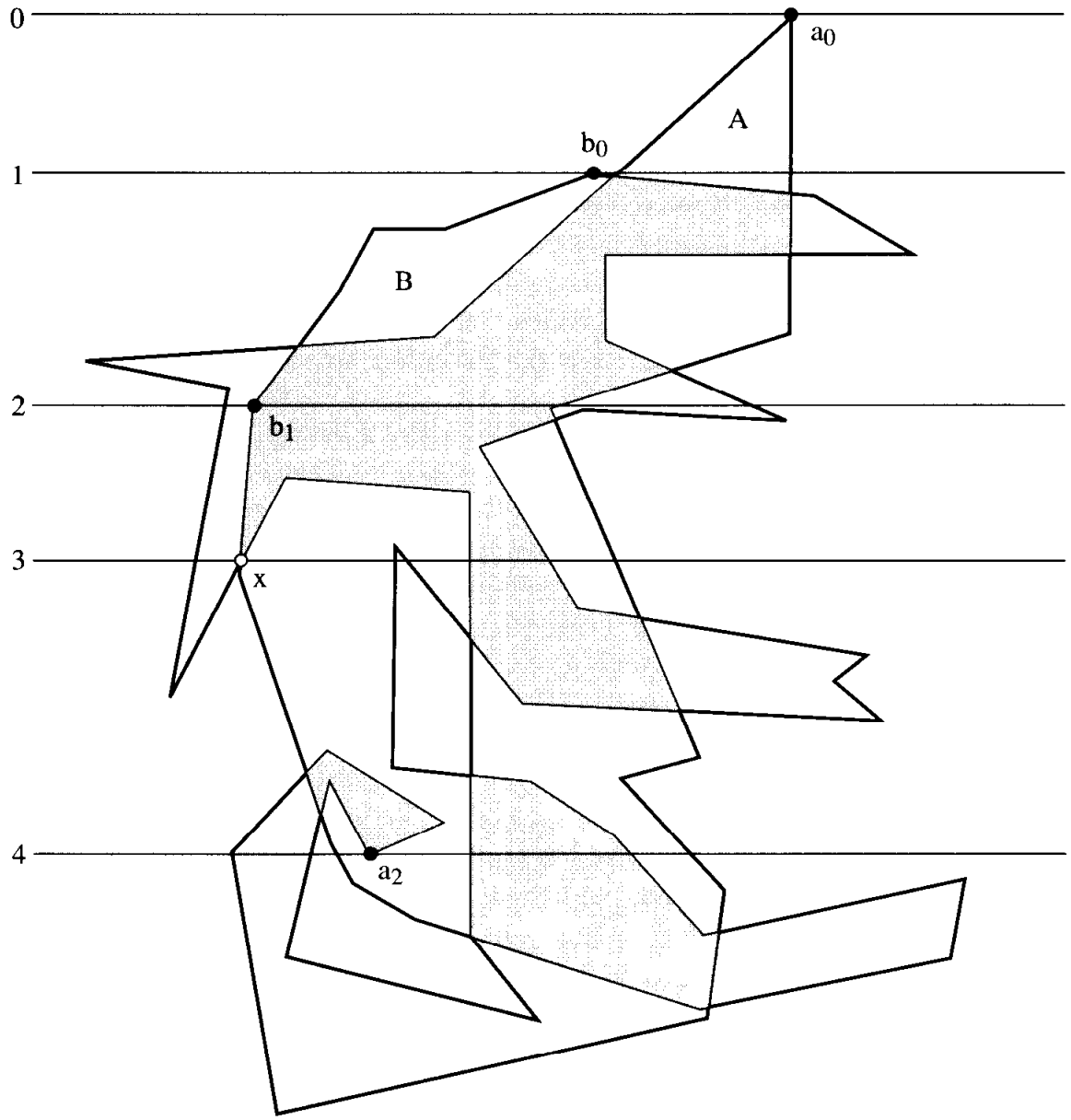


FIGURE 7.18 Intersection of two polygons: $A \cap B$ is shaded darkest.

analysis in Section 7.4. One maintains along the length of the sweep line L a "status" indicator, which has the following value:

$\emptyset$: exterior to both polygons;
A: inside A, but outside B;
B: inside B, but outside A; or
AB: inside both A and B.

The status is recorded for the span between each two adjacent segments pierced by L; clearly it is constant throughout each span.

Consider the example shown in Figure 7.18. When L is at position 2 (event b_1), the left-to-right status list is $(\emptyset, A, AB, B, \emptyset)$. This information can be easily stored in the same data structure representing L. We will not delve into the data structure details, but rather sketch how the status information can be updated in the same sweep that processes the segment intersection events, using the example in Figure 7.18.

At position 0, when L hits a_0, the fact that both A-edges are below a_0 indicates that we are inserting an A-span. At position 1, a B-span is inserted. Just slightly below b_0, an intersection event opens up an AB-span, easily recognized as such because the intersecting segments each bound A and B from opposite sides, with A and B below. At position 3, intersection event x, the opposite occurs: The intersecting segments each bound A and B above them. Thus an AB-span disappears, replaced by an $\emptyset$-span between the switched segments. At a_2 (position 4), the inverse of the a_0 situation is encountered: The A-edges are above, and an A-span is engulfed by the surrounding B-spans. Although we have not provided precise rules (Exercise 7.8.1[5]), it should be clear that the span status information may be maintained by the sweepline algorithm without altering the asymptotic time or space complexity.

Although this enables us to "paint" the intersection $A \cap B$ on a raster display, there is a further step or two to obtain lists of edges for each "polygonal" piece of $A \cap B$. The reason for the scare quotes around "polygonal" is that the intersection may include pieces that are degerate polygons: segments, points, etc. – what are sometimes collectively called "hair." Whether this is desired as part of the output depends on the application. This issue aside, there is still further work. For example, at position 3 in Figure 7.18, an AB-span disappears at x, but the polygonal piece that disappears locally at x continues on to lower sweepline positions elsewhere. Two AB-spans may merge, revealing that what appeared to be two separate pieces above L are actually joined below.

This aspect of the algorithm may be handled by growing polygonal chain boundaries for the pieces of the intersection as the sweepline progresses and then joining these pieces at certain events. Thus position 3 in the figure is an event that initiates joining a left-bounding AB-chain with a right-bounding AB-chain. Keeping track of the number of "dangling endpoints" of a chain permits detection of when a complete piece of the output has been passed: For example, at position 4 of the figure, a_2 closes up the chain and an entire piece can be printed, whereas at position 3, the chain joined at x remains open at its rightmost piercing with L. Again we will not present details.

Finally, it is easy to see that we could just have easily computed $A \cup B$, or $A \setminus B$, or $B \setminus A$ – the status indicator is all we need to distinguish these. Thus all "Boolean operations" between polygons may be constructed with variants of the Bentley–Ottmann sweepline algorithm, in the same time complexity. These Boolean operations are the heart of many CAD/CAM software systems, which, for example, construct complex parts for numerically controlled machining by subtracting one shape from another, joining shapes, slicing away part of a shape, etc., all of which are Boolean operations.

Theorem 7.8.1. *The intersection, union, or difference of two polygons with a total of n vertices, whose edges intersect in k points, may be constructed in $O(n \log n + k)$ time and $O(n)$ space.*

7.8.1. Exercises

1. *Handling degeneracies.*
   (a) [easy] Show that horizontal segments can be accommodated within the presented algorithm without increasing time or space complexity.

(b) [more difficult] Show that permitting many segments to pass through one intersection point does not lead to greater time or space complexity.

2. *Reducing the space requirements.* Show that the following strategy of Pach & Sharir (1991) (see also de Berg et al. (1997, p. 29)) reduces the space requirements of the Bentley–Ottmann algorithm to $O(n)$ without increasing its time complexity. Delete an intersection point, such as x_{34} in Figure 7.17, from Q whenever its generating segments cease being adjacent because another segment (s_5 in this example) is encountered between them. The same intersection point is recomputed and reinserted into Q later.
3. *Intersection of segments: implementation* [programming]. Using `SegSegInt` (Code 7.2) as a subroutine, implement the Bentley–Ottmann algorithm with naive data structures.
4. *Degenerate intersections.*
   (a) [easy] Show by example that $A \cap B$ can be a path of segments (i.e., a polygonal chain).
   (b) Prove that no connected component of the intersection of two simple polygons can be topologically equivalent to the union of three segments forming a "Y"-shape.
5. *Span status rules.* Detail the rules for updating the span status information (Section 7.8) for the various events that could occur during a sweep of two polygons.
6. *Polygon simplicity* [easy]. Prove that the Bentley–Ottmann algorithm may be used to detect whether a given list of n points form a simple polygon in time $O(n \log n)$.

7.9. EXTREME POINT OF CONVEX POLYGON

It is frequently necessary to find a boundary point of a convex polygon extreme in a certain direction. For example, the smallest box enclosing a polygon, where the box sides are aligned with the coordinate axes, can be constructed from extreme points in the four compass directions. Although often this computation is performed by a simple $O(n)$ scan of all vertices, it is not surprising that a minor variant of binary search will accomplish the same goal in $O(\log n)$ time. In this section we will sketch such a search algorithm to find a highest point and then generalize to an extreme in a particular, arbitrary direction.

Let the n polygon vertices be $P[0], \ldots, P[n-1]$, labeled counterclockwise. Suppose at some point of the search we know a highest vertex is counterclockwise between indices a and b. We will represent the collection of these indices, our search interval, by $[a, b]$. So if $a < b$, one of $P[a], P[a+1], \ldots, P[b-1], P[b]$ is a highest vertex. For this initial sketch, we will not worry about wraparound through index 0, nor will we be concerned with the possibility that two vertices are equally highest, although both of these issues complicate implementations.

The main idea is to use the directed edges of the polygon to decide how to halve the search interval. Let c be an index strictly between a and b. If the edge A after $P[a]$ points upward, then a is on the right chain of P. If in addition the edge C after $P[c]$ points downward, then c is on the left chain, and we have the situation illustrated in Figure 7.19(a): The highest point is between. In this case we may shorten the original search interval $[a, b]$ to $[a, c]$. A similar shortening occurs if A points downward and C upward ((b) of the figure), or if A and C both point upward ((c) and (d) of the figure), or if A and C both point downward (not shown). This halving process is repeated until the edge after c points down and the edge before it points up,

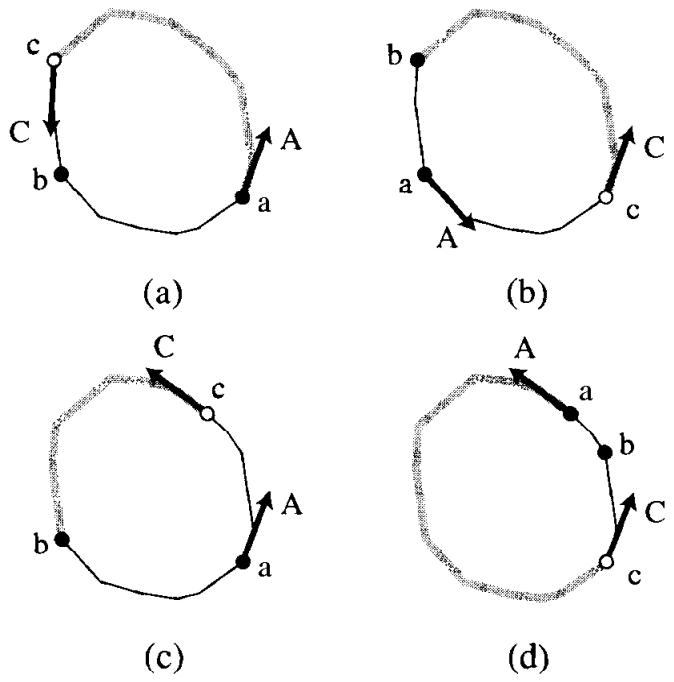


FIGURE 7.19 Four cases for finding a highest point. The $[a, b]$ interval is shortened to the shaded chain in each case.

indicating that c is highest. The pseudocode in Algorithm 7.3 shows the details of the decisions.

Algorithm: HIGHEST POINT OF CONVEX POLYGON
Initialize a and b.
repeat forever
    $c \leftarrow$ index midway from a to b.
    if $P[c]$ is locally highest then return c
    if A points up and C points down
        then $[a, b] \leftarrow [a, c]$
    else if A points down and C points up
        then $[a, b] \leftarrow [c, b]$
    else if A points up and C points up
        if $P[a]$ is above $P[c]$
            then $[a, b] \leftarrow [a, c]$
            else $[a, b] \leftarrow [c, b]$
    else if A points down and C points down
        if $P[a]$ is below $P[c]$
            then $[a, b] \leftarrow [a, c]$
            else $[a, b] \leftarrow [c, b]$

Algorithm 7.3 Highest point of convex polygon.

Three points require further clarification:

1. How is the midway index c computed?
2. How can the loop termination be implemented?
3. How does the possibility that two vertices are equally highest affect the algorithm?

Let us tackle the first problem: how to find an index midway between a and b. If $a < b$, then $(a + b)/2$ is midway. Note that if $b = a + 1$, then $(a + b)/2 = a$, due to truncation. If $a \geq b$, the interval $[a, b]$ includes 0, and the formula $(a + b)/2$ no longer works. For example, let $n = 10$, $a = 7$, and $b = 3$. Here $[7, 3] = (7, 8, 9, 0, 1, 2, 3)$, and the midpoint is 0. This can be computed by shifting b by n so that it is again larger than a, and taking the result $\bmod n$: $((a + b + n)/2) \bmod n$. In our example, $((7 + 3 + 10)/2) \bmod 10 = 0$. Note that if $a \geq b$ and $b = (a + 1) \bmod n$, then again the computation yields a, which is the same behavior as when $a < b$: When a and b are adjacent, the midpoint is a.

This gives us a midway function that could be implemented as shown in Code 7.23. Note what this function yields for the midpoint of $[a, a]$, which should represent the entire boundary of P: $(a + n/2) \bmod n$, halfway around from a, exactly as desired.

Loop termination is easy if there is a uniquely highest vertex: Then the vertices adjacent to c are both strictly lower. Capturing the situation where a horizontal edge is

```
int     Midway( int a, int b, int n )
{
  if (a < b) return ( a + b ) / 2;
  else       return ( ( a + b + n ) / 2 ) % n;
}
```

Code 7.23 `Midway`.

highest (or several collinear horizontal edges if the input permits this) is not much more difficult: Neither vertex adjacent to c is higher.

Unfortunately, we cannot be guaranteed that c will ever hit an extreme vertex, due to the truncation in `Midway` when $b = a + 1$. This always truncates clockwise, which can block a last needed counterclockwise step. Termination can be ensured by capturing the $c = a$ case specially. We leave a full implementation to Exercise 7.9.2[2] and turn now to a generalization and an application.

It is easy to alter the algorithm to find an extreme in an arbitrary direction u: Each test that a vector V points downward is replaced by the test $u \cdot V < 0$, each test that $P[a]$ is above $P[c]$ is replaced by the test that $u \cdot (P[a] - P[c]) > 0$, and so on. This permits using the extreme-finding algorithm for more than just bounding box calculations.

7.9.1. Stabbing a Convex Polygon

The problem of finding the intersection of a geometric object with a line is often called the "stabbing" problem. Here we show how the extreme-finding algorithm can be used to stab a convex polygon in $O(\log n)$ time.

Let P be the polygon and L the given line, and let u be a vector orthogonal to L. Find two vertices of P extreme in the $+u$ and $-u$ directions; call these a and b. See Figure 7.20. If both a and b are to one side of L, $L \cap P = \emptyset$. Otherwise a and b split ∂P into two chains whose intersections with L can be found by straightforward binary search: Chain $P[a, b]$ will yield intersection point x in the figure, and $P[b, a]$ will yield y.

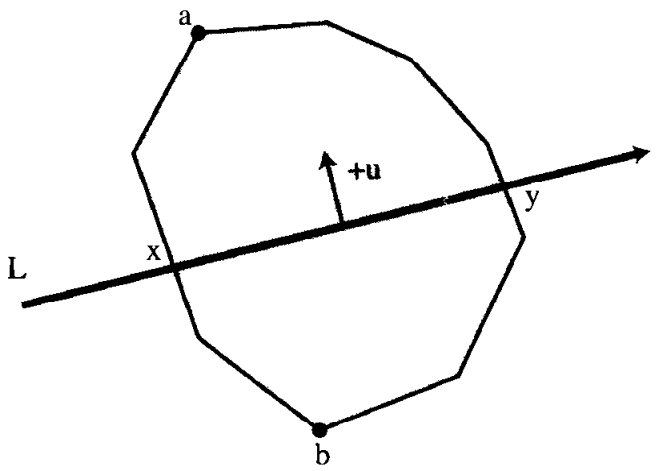


FIGURE 7.20 Stabbing a convex polygon.

7.9.2. Exercises

1. *Collinear points.* Suppose the input polygon contains three or more consecutive collinear vertices. Does this present a problem for Algorithm 7.3?
2. *Implement extremes algorithm* [programming]. Implement Algorithm 7.3, generalized to arbitrary directions u. Test on examples that have an extreme edge.
3. *Line–polygon distance.* Design an algorithm to determine the distance between an arbitrary polygon P of n vertices and a query line L. Define the distance to be

$$\min_{x,y}\{|x - y| \ : \ x \in P \, , \ y \in L\},$$

where x and y are points. Try to achieve $O(\log n)$ per query after preprocessing.

7.10. EXTREMAL POLYTOPE QUERIES

The problem of finding an extreme point of a polytope is much more difficult than the two-dimensional version covered in the previous section. There is no direct counterpart to the one-dimensional search we used on the boundary chain of the convex polygon: The two-dimensional surface of a polytope provides too much freedom in the search direction. Nevertheless, Kirkpatrick (1983) invented a breathtakingly beautiful search structure that permits the problem to be solved in $O(\log n)$ query time, asymptotically the same as in two dimensions (although we will see that the constant of proportionality is larger).

7.10.1. Sketch of Idea

The key idea is to form a sequence of simpler and simpler polytopes nested within the original given polytope P.[21] The innermost polytope is a tetrahedron or triangle, and there are $O(\log n)$ polytopes altogether. Construction of the hierarchy of polytopes can

[21]This sequence is often called the Dobkin–Kirkpatrick hierarchy; see Dobkin & Kirkpatrick (1990).

be done in $O(n)$ time, and storing all of them only uses $O(n)$ space. Once they are constructed, extremal queries can be answered in $O(\log n)$ time. Note that although this matches the time complexity for finding extreme points of convex polygons, the polygons did not require preprocessing (although even to read such a polygon into memory requires $O(n)$ time, which can be considered a crude form of preprocessing).

An extremal query is answered by first finding the extreme for the innermost polytope, and using that to work outwards through the hierarchy toward P. Let the sequence of nested polytopes be $P = P_0, P_1, P_2, \ldots, P_k$, where P_k is the innermost. And let a_i be the extreme point for polytope P_i. We first find the extreme point a_k of P_k by comparing its three or four vertices. Knowing a_k (and some other information) will give us a small set of candidate vertices of P_{k-1} to check for extremality. This yields a_{k-1}, and from that we find a_{k-2}, and so on. It will turn out that the work to move from one polytope to the next in the hierarchy is constant. Because $k = O(\log n)$, the total time to find a_0 is also $O(\log n)$. We now proceed to detail the search structure and the algorithm.

7.10.2. Independent Sets

Recall that the edges and vertices of a polytope form a planar graph (Section 4.1.4); Figure 7.21(a) shows the graph for an icosahedron, Figure 7.22, an example we will use to illustrate ideas throughout this section. Kirkpatrick's key idea depends on the graph theory notion of an "independent set." A set of nodes I of a graph G is called an *independent set* if no two nodes in I are adjacent in G. Thus they are "spread out" in a sense. Such an independent set is marked in Figure 7.21(a). This set of three nodes is in fact a *maximum independent set* for this graph, in that no four nodes form an independent set. It is important for Kirpatrick's scheme that planar graphs have "large" independent sets composed entirely of vertices of "small" degree (i.e., a small number of adjacent nodes); these vague qualifiers will be made precise later.

The construction of P_1, the first polytope nested inside $P = P_0$, proceeds as follows. An independent set of vertices for P_0 is found as in Figure 7.21(a). These vertices, and all their incident edges, are deleted from the graph. The result is shown in Figure 7.21(b). Because the vertices are independent, each deletion produces one new face in the graph. In Figure 7.21(b), each deletion produces a pentagon (which looks like a quadrilateral because two edges are collinear in the drawing). Next, these faces are triangulated; see Fig 7.21(c). In our case we can triangulate them arbitrarily; more on this is discussed in Section 7.10.4. The geometric equivalent to this operation on polytopes is to delete the vertices in the independent set and take the convex hull of the remaining vertices. This produces polytope P_1, which is clearly nested inside P_0, since it is the hull of a subset of P_0's vertices. Figure 7.23 shows P_1 corresponding to the graph in Figure 7.21(c). Note that the pentagons (two of which are visible in the figure) are comprised of three coplanar triangles. In general the vertices adjacent to a deleted independent vertex will not be coplanar; they are in this instance because of the symmetry of the icosahedron. It is the coplanarity and convexity of the face that permitted us to triangulate it arbitrarily. In general we would have to take the hull of the vertices around the boundary of the new face to construct the triangulation.

Now the process is repeated to construct P_2. A set of independent vertices of P_1 are identified, as marked in Figure 7.21(c). These are deleted, producing the graph shown

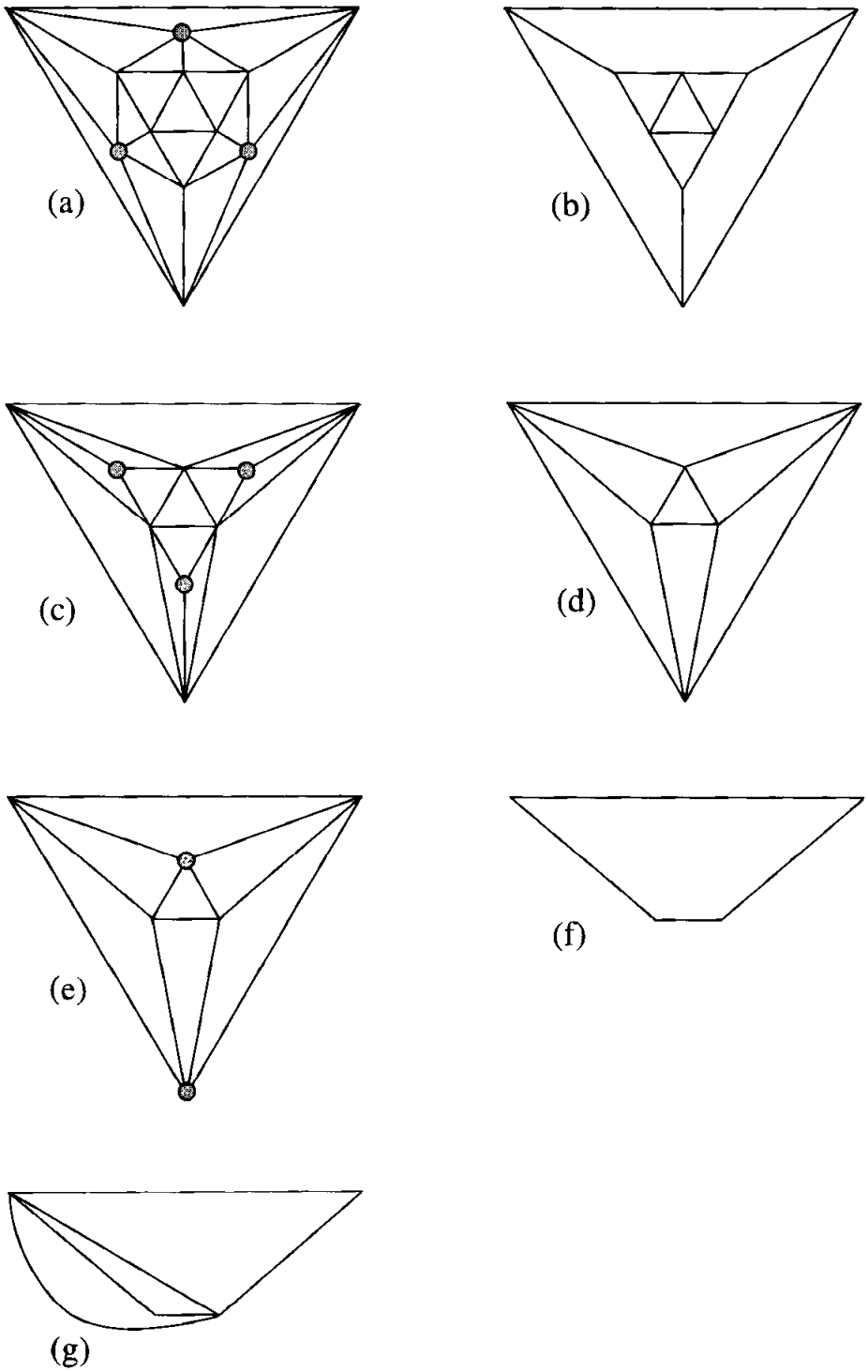


FIGURE 7.21 The graph of the vertices and edges of an icosahedron. Marked nodes form independent sets. (a) Original graph of P_0; (b) after deletion of independent set; (c) after retriangulation: the graph of P_1; (d) after deletion; (e) after retriangulation (same as (d)): the graph of P_2; (f) after deletion; (g) after retriangulation: the graph of P_3.

in Figure 7.21(d). It so happens that, this time, the deletion produces only triangle faces, so no further triangulation is needed. The reader may recognize Figure 7.21(d) as the Schlegel diagram of an octahedron, and indeed the corresponding polytope P_2 is a (nonregular) octahedron, as shown in Figure 7.24.

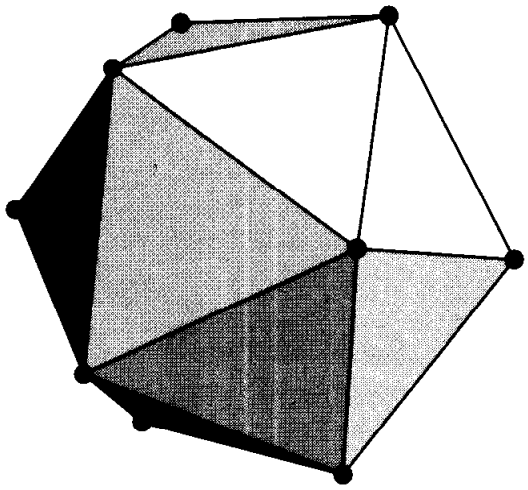


FIGURE 7.22 Icosahedron, P_0 (Figure 7.21(a)).

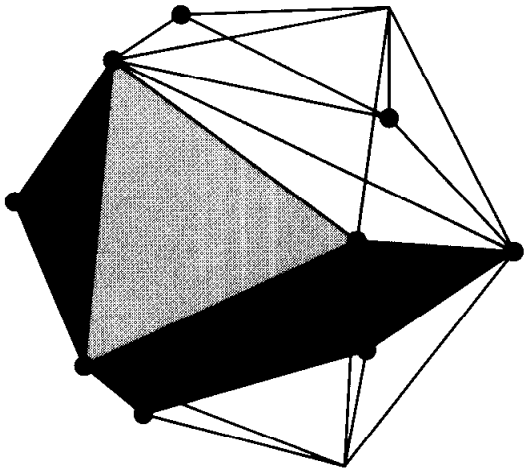


FIGURE 7.23 P_1: 9 vertices, 14 faces (Figure 7.21(c)).

The process is repeated one more time. An independent set of size two is identified in Figure 7.21(e). Deletion produces the graph in Figure 7.21(f). Triangulation of the two quadrilateral faces (one of which is exterior) produces Figure 7.21(g), which is the graph of a tetrahedron. Figure 7.25 displays this tetrahedron, which, again because of the symmetry of the icosahedron, consists of four coplanar points.

7.10.3. Independent Sets: Properties and Algorithm

To achieve a nested polytope hierarchy with the right properties, the independent sets cannot be chosen arbitrarily. Fortunately it is easy to obtain the appropriate properties, as Kirkpatrick showed for arbitrary planar graphs (Kirkpatrick 1983). The arguments are slightly easier for polytope graphs; here I follow the presentation of Edelsbrunner (1987).

In order to achieve only $O(\log n)$ polytopes, it suffices to delete a constant fraction of the vertices at each step. For suppose we can find an independent set of cn vertices on any polytope of n vertices, for $c < 1$. Then at each step, we reduce the vertices by a factor of $(1 - c)$, so after k steps, we will have $n(1 - c)^k$ vertices. This quantity reaches

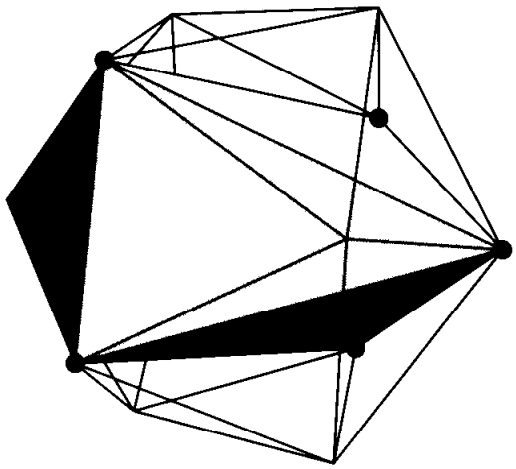


FIGURE 7.24 P_2: an octahedron (Figure 7.21(e)).

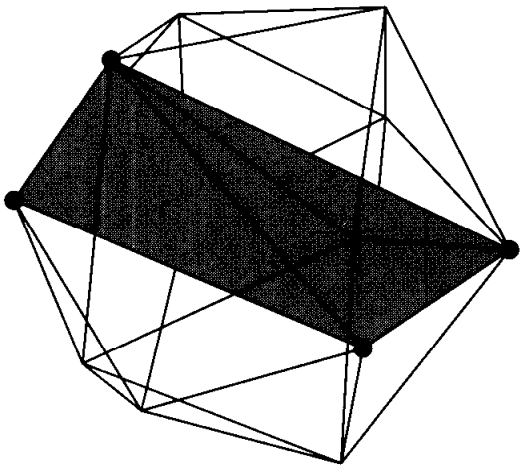


FIGURE 7.25 P_3: a flat tetrahedron (Figure 7.21(g)).

4 when k is a particular value:

$$n(1-c)^k = 4,$$

$$\log n + k\log(1-c) = 2,$$

$$k = \frac{\log n}{-\log(1-c)} - \frac{2}{-\log(1-c)} \tag{7.10}$$

Since $(1-c) < 1$, $-\log(1-c) > 0$, and the right-hand side of Equation 7.10 is a positive constant times $\log n$, minus another constant; so it is $O(\log n)$. For example, for $n = 2^{20} \approx 10^6$ and $c = 1/10$, $k = 118$.

Thus our goal is to show that every polytope graph has an independent set of size cn for some $c < 1$.

The most natural method of finding an independent set is iterate the following "greedy" procedure: Choose a vertex of lowest degree that is not adjacent to any other vertices previously chosen. The intuition is that low degree vertices "kill" as few other vertex candidates as possible. Although this simple-minded algorithm will not necessarily find a maximum independent set, it turns out to be sufficient for our purposes. We can even loosen it up a bit to choose any vertex whose degree is not too high: This avoids a search for a vertex of lowest degree. In particular, we use Algorithm 7.10.1. It is clear that

this algorithm produces an independent set and runs in $O(n)$ time on a planar graph of n nodes. What is not so clear is that it produces a "large" independent set. This is established in the following theorem of Edelsbrunner (1987, Theorem 9.8).

Algorithm: INDEPENDENT SET
Input: a graph G.
Output: an independent set I.
$I \leftarrow \emptyset$
Mark all nodes of G of degree ≥ 9.
while some nodes remain unmarked do
    Choose an unmarked node v.
    Mark v and all the neighbors of v.
    $I \leftarrow I \cup \{v\}$.

Algorithm 7.4 Independent set.

Theorem 7.10.1. *An independent set I of a polytope graph G of n vertices produced by Algorithm 7.10.1 has size at least $n/18$.*

In terms of our previous notation, the theorem claims the constant $c = 1/18$ is achieved by Algorithm 7.10.1.

Proof. The key to the proof is Euler's formula, $V - E + F = 2$. We established in Chapter 4 (Section 4.1.5, Equation (4.4)) that this formula implies that the number of edges of a polytope graph is bounded above by $3V - 6$: $E \leq 3n - 6$. We now use this to obtain an upper bound on the sum Σ of the degrees of all the nodes of G. This sum double counts every edge of G, since each edge has two endpoints. Thus $\Sigma \leq 6n - 12$.

This bound on the sum of degrees immediately implies that there must be numerous nodes with small degrees. For if all nodes had high degree, the sum of their degrees would exceed this bound. Quantitatively, there must be at least $n/2$ vertices of degree ≤ 8. For suppose the contrary: There are more than $n/2$ nodes of degree ≥ 9. The sum of the degrees of just these nodes is $\geq 9n/2$. The other nodes must each have degree ≥ 3. Let us assume that n is even, to simplify the calculations. The smallest value of Σ would occur when only half the nodes have high degree and the other half have the lowest degree possible. Therefore

$$\Sigma \geq 9n/2 + 3n/2 = 6n. \tag{7.11}$$

This contradicts the upper bound of $6n - 12$ we established above. For n odd, a similar contradiction is obtained (Exercise 7.10.6[2]). Therefore we have established that at least half the nodes of G have degree ≤ 8 and so are candidates for the independent set constructed by Algorithm 7.10.1. It remains to show that the algorithm selects a "large" number of these candidates.

Every time the algorithm chooses a node v, it marks v and all of v's neighbors. The worst that could happen is that (a) all of these nodes it marks were previously unmarked and (b) v has the highest degree possible, 8. Let m be the number of unmarked nodes

of G of degree ≤ 8. An example may make the relationships clearer. Suppose $m = 90$. A node v is chosen, and in the worst case, 8 unmarked nodes are marked. This reduces m by 9, to 81. Again a node is chosen among these 81, and again in the worst case, m is reduced by 9. It should be clear that at least 1/9-th of m nodes will be added to the independent set I; so with $m = 90$, $|I| \geq 10$.

Now since we showed above that $m \geq n/2$, it follows that $|I| \geq n/18$. And thus we have established that Algorithm 7.10.1 always produces an independent set at least 1/18-th the size of the original graph. □

With $c = 1/18$, the number of nested polytopes is (by Equation 7.10) less than $12.13 \log n$. This constant of proportionality leaves much to be desired, but always choosing the unmarked node of smallest degree improves c to 1/7 (Edelsbrunner 1987, Problem 9.9(d)) and the log constant to 4.50.

7.10.4. Construction of Nested Polytope Hierarchy

We now detail the construction of the hierarchy. In the pseudocode shown in Algorithm 7.5, $N(v)$ is the set of neighbors of v: all the vertices adjacent to v.

Algorithm: NESTED POLYTOPE HIERARCHY
Input: a polytope P.
Output: an $O(\log n)$ hierarchy of nested polytopes, $P = P_0, P_1, \ldots, P_k$
$i \leftarrow 0$; $P_0 \leftarrow P$.
while $|P_i| > 4$ do
    Apply Algorithm 7.10.1 to identify an independent set I of P_i.
    Initialize P_{i+1} to P_i.
    for each vertex $v \in I$ do
        Delete v from P_{i+1}.
        Retriangulate the hole by constructing the hull of $N(v)$.
        Link each new face of P_{i+1} to v.
    Link unchanged faces of P_{i+1} to P_i.

Algorithm 7.5 Nested polytope hierarchy.

Space Requirements

We have already established that the polytope hierarchy has height $O(\log n)$. At first it might seem that the time and space required to construct the hierarchy would be $O(n \log n)$, linear per level, but in fact the total is linear because of the constant fractional reduction between levels of the hierarchy. In particular, with $c = 1/18$, each polytope has at most 17/18-ths as many vertices as its "parent." So the total size is no more than

$$n\,[(17/18) + (17/18)^2 + (17/18)^3 + \cdots].$$

Although the sum of powers of $(1 - c)$ has only k terms, it is easier to obtain an upper bound by letting it run to infinity. Then it is the familiar geometric series, with sum

$$\frac{1}{1-(1-c)} = \frac{1}{c} = 18.$$

Therefore the total storage required is at most $18n = O(n)$. And similarly the construction time is $O(n)$, although this needs some argument, not provided here.

Retriangulating Holes

We mentioned earlier that when a vertex v is deleted from P_i, the resulting hole must be triangulated appropriately to produce P_{i+1}. Let $N(v)$ be the neighbors of v. In general they will not be coplanar, and so an arbitrary triangulation will not suffice. We need to compute the convex hull of $N(v)$ and use the "outer faces" of this hull to provide the triangulation. In practice we might recompute the entire hull at each step to construct P_{i+1} from P_i, but this would lead to $O(n \log n)$ time complexity. But observe that $|N(v)| \leq 8$, because v had degree ≤ 8. This means that each hole can be patched with triangles in constant time. And the total number of hole patches necessary for the entire hierarchy construction is no more than a constant times the number of vertices deleted, which is $O(n)$.

Linking Polytopes

It is necessary to connect the polytopes in adjacent positions of the hierarchy with data structure links to aid the search. Because the vertices removed at each step of the hierarchy construction form an independent set, the relationships among vertices, edges, and faces of two adjacent polytopes are unambiguous. We will not go into details here (see Edelsbrunner (1987, pp. 199–200)) but rather will just assume that any reasonable link we need is available.

7.10.5. Extreme Point Algorithm

Now we apply the hierarchy to answer extreme point queries. We will explain the algorithm as if we are seeking the highest point of the polytope, a vertex with largest z coordinate, but the process works for an extreme in any direction u in an analogous fashion. The algorithm was first detailed by Edelsbrunner & Maurer (1985); see also Edelsbrunner (1987, Section 9.5.3).

Let a_i be a highest point of polytope P_i. To keep the presentation simple, we will assume that a_i is unique for each i. The essence of the algorithm is to find the highest point a_k of P_k, the innermost polytope, by brute-force search, and then use a_k to help find a_{k-1}, use this to find a_{k-2}, and so on until a_0 is found, which is the highest point of $P_0 = P$, the original polytope. This process can be viewed as raising a plane π orthogonal to the z axis from a_k, to a_{k-1}, and so on to a_0. Because the polytopes are nested, this plane only moves upwards. An example is shown in Figure 7.26. Here the innermost polytope, a triangle, is not shown.

The key to the algorithm is the relationship between a_{i+1} and a_i. We condense this relationship into two lemmas, Lemmas 7.10.2 and 7.10.3 below. The first is perhaps easiest to see if we imagine π moving downwards, from a_i to a_{i+1}.

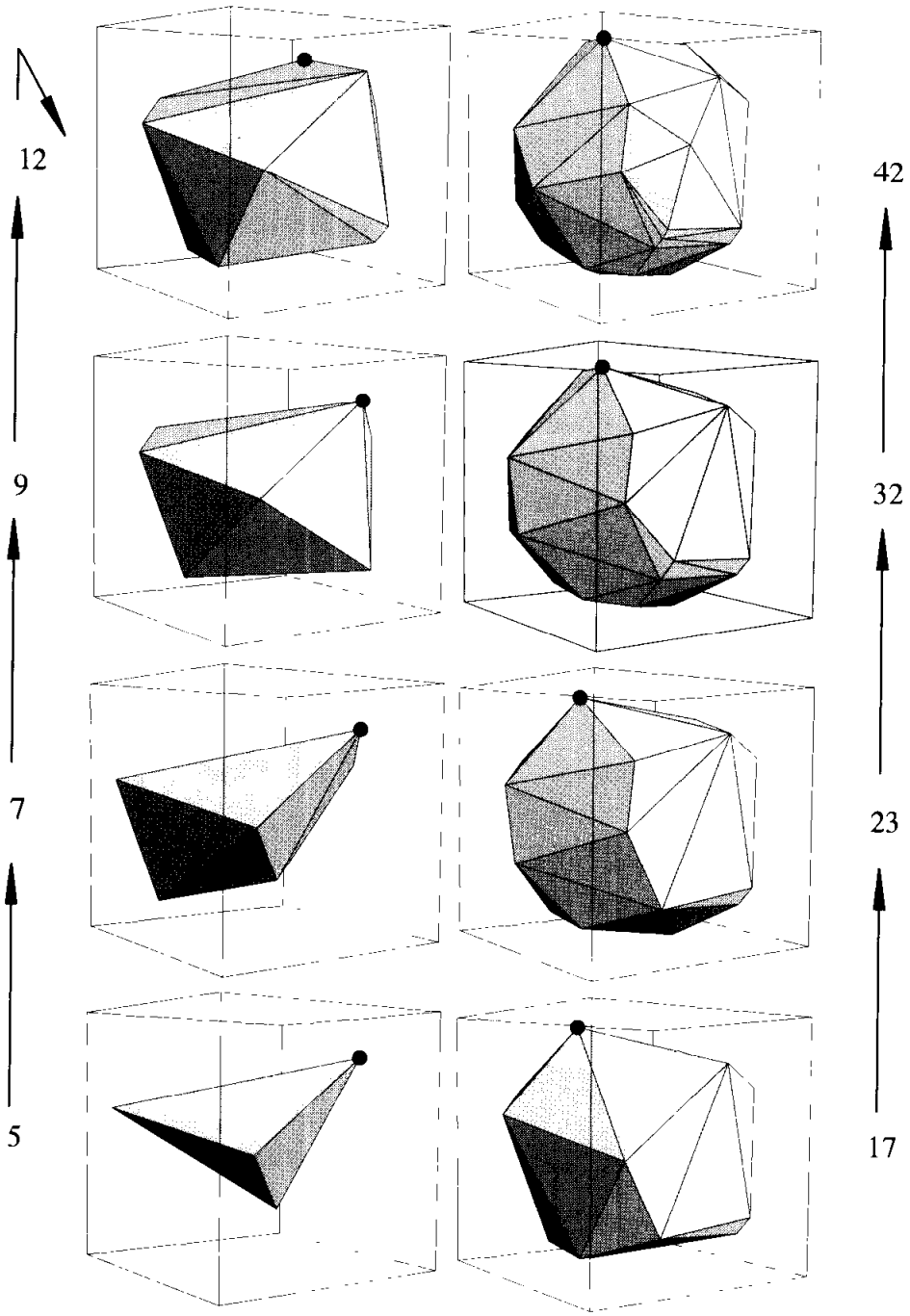


FIGURE 7.26 Polytope hierarchy with highest vertices marked, and number of vertices noted to the side. Box dimensions are ± 100. The highest vertex has z coordinate 63, 63, 63, 77, 92, 92, 94, 94, moving up the hierarchy.

Lemma 7.10.2. *Let a_i and a_{i+1} be uniquely highest vertices of P_i and P_{i+1}. Then either $a_i = a_{i+1}$ or a_{i+1} is the highest among the vertices adjacent to a_i.*

Proof. We consider two cases. First, suppose that a_i is a vertex of both P_i and P_{i+1}. Because $P_i \supset P_{i+1}$, no vertex of P_{i+1} can be higher than the highest of P_i, and therefore the highest vertex a_{i+1} of P_{i+1} must in this case be a_i.

Second, suppose a_i is one of the vertices deleted in the construction of P_{i+1}. Let b_{i+1} be the highest vertex of P_{i+1} among those adjacent to a_i in P_i. The claim of the lemma is that b_{i+1} is the highest vertex of P_{i+1}.

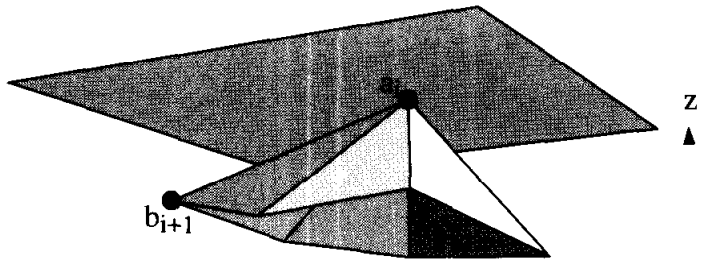


FIGURE 7.27 Highest point a_i of P_i.

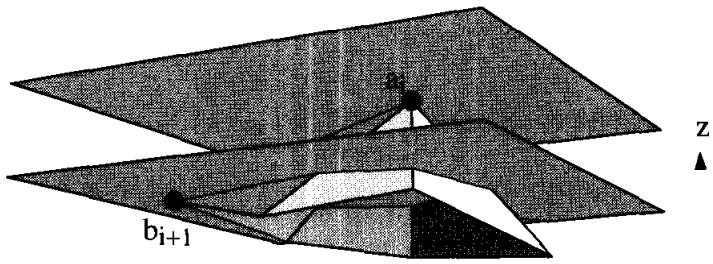


FIGURE 7.28 Highest point $b_{i+1} = a_{i+1}$ of P_{i+1}.

Consider the cone of faces incident to a_i in P_i; see Figure 7.27. Call the infinite extension of this cone C. By the convexity of P_i, $C \supset P_i$, and by nesting, $C \supset P_i \supset P_{i+1}$. Thus $a_{i+1} \in C$. But now no vertex of P_{i+1} can be located in the umbrella-shaped region under a_i, $U = C - P_{i+1}$. So a_{i+1} must lie in the other part of C, $C - U$, which is necessarily below the height of b_{i+1}, as Figure 7.28 makes clear. Therefore $b_{i+1} = a_{i+1}$. □

An immediate corollary of this lemma is that a_i is either identical to a_{i+1}, or is adjacent to it. It might seem this gives us the "hook" we need to move from a_{i+1} to a_i, but in fact this is not enough, because we have no bound on the number of vertices adjacent to a_{i+1}; so if we search them all for a_i the algorithm will work correctly, but it will have time complexity $O(n)$ rather than the $O(\log n)$ we desire. We need a more specific hook from a_{i+1} to a_i.

Extreme Edges

If one projects P_{i+1} onto a plane orthogonal to π, say the xz-plane, then π becomes a line and P_{i+1} becomes a convex polygon P'_{i+1}, as shown in Figure 7.29. Let primes denote objects projected to the xz-plane. Define L_{i+1} and R_{i+1} as the two edges of P_{i+1} that project to the two edges of P'_{i+1} incident to a'_{i+1}, as illustrated.

Now define the "umbrella parents," or just *parents*, of an edge e of P_{i+1} to be the vertices of P_i from which it derives, in the following sense: If e is an edge of P_{i+1} but not of P_i, then it sits "under" some vertex v of P_i whose umbrella of incident faces was deleted to produce P_{i+1}; this v is the (sole) parent of e. (This is most evident in Figure 7.23, where the two diagonals of the upper pentagonal face sit under a vertex of degree five.) If e is an edge of both P_{i+1} and P_i, then its parents are the two vertices of

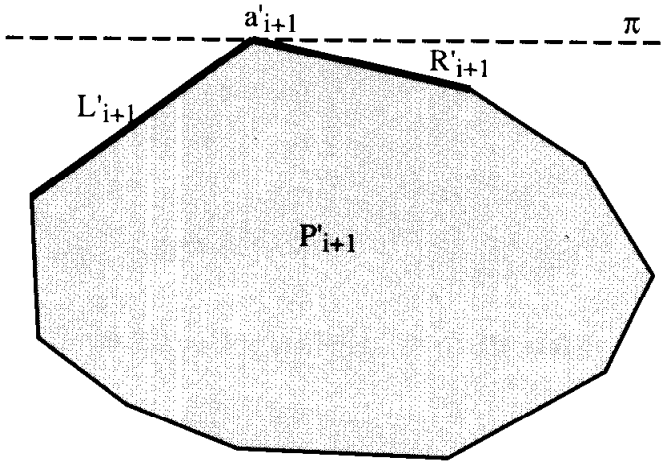


FIGURE 7.29 Definition of extreme edges L_{i+1} and R_{i+1}; The z axis is vertical.

P_i at the tips of the two triangle faces adjacent to e (which may or may not be vertices of P_{i+1}).

The key lemma is:

Lemma 7.10.3. *Let a_i and a_{i+1} be uniquely highest vertices of P_i and P_{i+1}. Then either $a_i = a_{i+1}$ or a_i is the highest among the parents of the extreme edges L_{i+1} and R_{i+1}.*

Before discussing why this might be true, let us explore its consequences. If we have both the extreme vertex a_{i+1} of P_{i+1}, and the extreme edges L_{i+1} and R_{i+1}, we can find the extreme vertex a_i of P_i by examining the (at most) five candidates provided by the lemma. If we can then find the new extreme edges L_i and R_i of P_i in constant time, we have achieved one full step up the hierarchy in constant time, which will result in $O(\log n)$ overall.

How can the extreme edges be computed once a_i is known? There are two cases to consider:

1. $a_i \neq a_{i+1}$. Here, surprisingly, we can use a brute-force search for the extreme edges. The reason is that such a search will have time complexity dependent on the degree of a_i, which, when we first encounter it in the hierarchy of polytopes, is an independent vertex chosen for deletion in the hierarchy construction and therefore has degree ≤ 8.
2. $a_i = a_{i+1}$. Here a brute-force search is not appropriate, because if we move through a series of levels of the hierarchy without the extreme vertex changing, its degree can grow larger and larger by "accretion" of edges: We are only guaranteed a degree ≤ 8 upon first encounter. This accretion is evident in the three innermost polytopes of the hierarchy in Figure 7.26. Fortunately, the new extreme edges are "close" to the old: L_i is either L_{i+1} or is adjacent to a parent of L_{i+1}; and similarly for R_i. I will not justify this claim (Exercise 7.10.6[3]).

So in both cases the new extreme edges can be found in constant time. Now we justify Lemma 7.10.3.

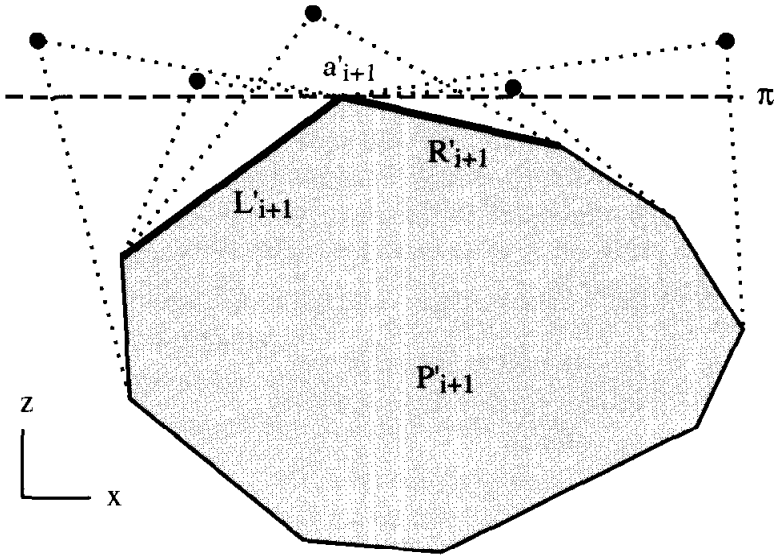


FIGURE 7.30 Potential locations for a_i' and tangents shown dotted.

Proof. Suppose $a_i \neq a_{i+1}$. Then a_i is above π. Of course the projection of P_i onto the xz-plane is a convex polygon P_i' that encloses the projection P_{i+1}'. (Recall that primes indicate projected objects.) If possible locations for a_i' are considered, as in Figure 7.30, it becomes clear that a_i must "sit over" one or both of the extreme edges L_{i+1} and R_{i+1}. This is the intuition behind the lemma.

Why are the dotted connections from a_i' in the figure reasonable possibilities? First, recall that all the vertices adjacent to a_i in P_i are also vertices of P_{i+1}. So the edges in the projection emanate from a_i' and terminate in P_{i+1}' below π. Second, $P_i \supset \text{conv}\{a_i \cup P_{i+1}\}$, so the two tangents through a_i' supporting P_{i+1}' are in P_i'. Third, there can be no edges from a_i' "outside of" these tangents, because such edges could not terminate in P_{i+1}'. Thus the boundary of P_i' includes the a_i' tangents, and Figure 7.30 is an accurate depiction.

Since it is clear that the a_i' tangents must encompass at least one of the extreme edge of P_{i+1}' and that a_i is a parent of this edge, we have established the lemma. □

We can summarize the algorithm in the pseudocode shown in Algorithm 7.6. From Lemmas 7.10.2 and 7.10.3, and from Exercise 7.10.6[3], this algorithm will work correctly, so the only issue remaining is its time complexity. But we have ensured that the work done at each level of the algorithm is constant. This then establishes the query time of the algorithm: $O(\log n)$ levels of the hierarchy are searched, and the work at each level is a constant. Modulo the details we have ignored, we have established the following theorem:

Theorem 7.10.4. *After $O(n)$ time and space preprocessing, polytope extreme-point queries can be answered in $O(\log n)$ time each.*

One important application of this theorem arises in collision detection: detecting whether two convex polyhedra of n and m vertices intersect. Representing each polytope

Algorithm: EXTREME POINT OF A POLYTOPE
Input: a polytope P, and a direction vector u.
Output: the vertex a of P extreme in the u direction.
Construct the hierarchy of nested polytopes, $P = P_0, P_1, \ldots, P_k$,
    by running Algorithm 7.5.
$a_k \leftarrow$ the vertex of P_k extreme in the u direction.
Compute L_k and R_k.
for $i = k-1, k-2, \ldots, 1, 0$ do
    $a_i \leftarrow$ the extreme vertex among a_{i+1}
        and the parents of L_{i+1} and R_{i+1}.
    if $a_i \neq a_{i+1}$ then
        for all edges incident to a_i do
            Save extreme edges L_i and R_i.
    else ($a_i = a_{i+1}$) Compute L_i from L_{i+1} etc.

Algorithm 7.6 Extreme point of a polytope.

in a hierarchy, and using Exercise 7.10.6[8], it is possible to compute efficiently the separation between the polytopes at a common level of the hierarchy from the separation between the polytopes at the level below. This leads to an $O(\log n \log m)$ intersection detection algorithm (Dobkin & Kirkpatrick 1983).

7.10.6. Exercises

1. *Innermost polytope* [easy]. Why cannot the innermost polytope of the hierarchy have ≥ 5 vertices?
2. *n odd.* In the proof of Theorem 7.10.1, we only covered the n even case. Follow the argument for n odd, and show the conclusion still holds.
3. $a_i = a_{i+1}$. Argue that if $a_i = a_{i+1}$ in Algorithm 7.6, L_i is either L_{i+1} or it is adjacent to a parent of L_{i+1}. Show how these facts permit L_{i+1} to be found in constant time in this case.
4. *Nested polygon hierarchy.* Develop a method of constructing a hierarchy of $O(\log n)$ convex polygons nested inside a given convex polygon of n vertices. Use this to design an extreme-point algorithm that achieves $O(\log n)$ query time.
5. *The constant c* [easy]. Compute the average constant c for the example in Figure 7.26, and using this, calculate k from Equation 7.10.
6. *Implementation of independent set algorithm* [programming]. Write a program to find an independent set in a given graph using Algorithm 7.10.1.
7. *Nested polytope implementation* [programming]. Use the convex hull code from Chapter 4 and the independent set code from the previous exercise to find a polytope nested inside the hull of n points. Test it on randomly generated hulls, and compute the average fractional size of the independent sets. Compare this against the $c = 1/18$ constant established in Theorem 7.10.1, and try to explain any difference.
8. *Plane–polyhedron distance.* Design an algorithm to determine the distance between an arbitrary polyhedron P of n vertices and a query plane π. Define the distance to be

$$\min_{x,y}\{|x-y| : x \in P,\ y \in \pi\},$$

where x and y are points. Try to achieve $O(\log n)$ per query after preprocessing. Compare with Exercise 7.9.2[3].

9. *Finger probing a polytope* (Skiena 1992). Develop an algorithm for "probing" a polytope P that contains the origin, with a directed line L through the origin. Each probe is to return the first face of P hit by L moving in from infinity. Try to achieve $O(\log n)$ query time, by dualizing P and L with the polar dual discussed in Chapter 6 (Exercise 6.5.3[3]).
10. *Circumscribed hierarchies.*
    a. Define an $O(\log n)$ hierarchy of polygons surrounding a convex polygon, with properties similar to the inscribed hierarchy.
    b. Define an $O(\log n)$ hierarchy of polytopes surrounding a given polytope.
    c. Suggest applications for these circumscribed hierarchies.

7.11. PLANAR POINT LOCATION

7.11.1. Applications

One of the most fundamental of geometric searches involves locating a point in a subdivision of the plane, known as the *planar point location* problem. We have already encountered needs for this search for constructing a trapezoidalization (Section 2.4.1), for searching Voronoi diagrams to find nearest neighbors (Section 5.5.1), and for searching the kth-order Voronoi diagram to find the k-nearest neighbors (Section 6.6).

Another common application of planar point location is determining if a query point is inside a given polytope. Although we already mentioned a method for solving the point-in-polytope problem in Section 7.5, that method is "single-shot" and not efficient for the situation where the polytope is fixed and must be repeatedly queried for different points. The connection between this problem, and planar point location can be seen via the same type of three-to-two dimensions projection used in Section 7.3.1.[22] Suppose the given polytope P sits on the xy-plane, and let P^+ be the set of all faces of P whose outward normal has a nonnegative upward component (i.e., whose z component is ≥ 0). These are the faces visible from $z = +\infty$. Let P^- be the set of all the other faces, whose normals point down. Project P^- onto the $z = 0$ plane, and project P^+ onto the $z = h$ plane, where h is the height of P. This results in two subdivisions on these two planes; call them S^+ and S^-. Now, given any query point q, project it up and down and locate it in both subdivisions. Suppose it projects into face f^+ of S^+. Then this selects out a vertical "prism" as shown in Figure 7.31. It is then easy to decide if q is above or below f in this prism. If it is above f, $q \notin P$. If below, then the process is repeated on the lower subdivision. $q \in P$ iff it is below the face of P provided by the search in S^+ and above the face of P provided by the search in S^-.

7.11.2. Independent Set Algorithm

The reader may have realized already that Kirkpatrick's search structure, presented in Section 7.10, provides a solution to the planar point location problem. Indeed this was his original motivation (Kirkpatrick 1983). The only complication is that a general planar

[22] Suggested in Edelsbrunner (1987, Ex. 11.5).

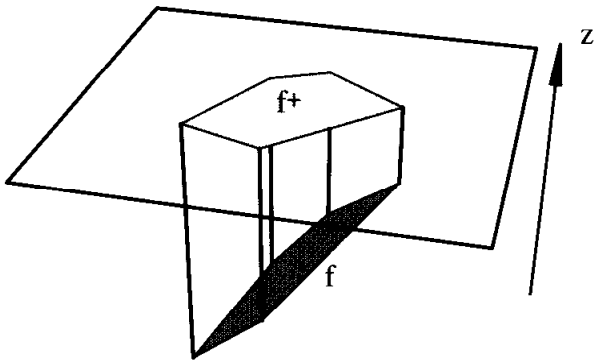


FIGURE 7.31 Face f of the polytope projects up to face f^+ of the upper planar subdivision. Cf. Figure 7.3.

subdivision may have general polygonal faces, which need to be triangulated by a polygon triangulation algorithm. After that step, we can proceed as with the polytope hierarchy, except that each hole produced by a vertex deletion should be retriangulated by a polygon triangulation algorithm. As with the polytope case, however, this retriangulation only takes constant time per hole, since the holes have at most eight vertices.

Theorem 7.11.1. *A polygonal planar subdivision of n vertices can be preprocessed in $O(n)$ time and space so that point location queries can be answered in $O(\log n)$ time.*

7.11.3. Monotone Subdivisions

Although Kirkpatrick's search structure in some sense settles the planar point location problem, it was neither the first algorithm to achieve those bounds nor the most recent. An early algorithm by Dobkin & Lipton (1976) uses quadratic space, but is very simple, and achieves a much better query constant: Queries can be performed with $2 \log n$ comparisons. Lipton & Tarjan (1980) were the first to achieve $O(n)$ preprocessing with $O(\log n)$ query time, but their algorithm is impractically complex. Kirkpatrick's algorithm is elegant and ideal for polytope problems, but its high query constant make it unattractive for general planar subdivision search.

One popular method of performing planar point location depends on *monotone subdivisions*. A subdivision is monotone if every face is monotone, say with respect to the horizontal. A face is *monotone* if it meets every vertical line in a connected set: either a point or a segment (see Section 2.1). Many commonly encountered subdivisions are monotone: triangulations and any convex subdivision such as a Voronoi diagram or a kth-order Voronoi diagram or an arrangement of lines. Those subdivisions that are not monotone can be further partitioned (by, e.g., triangulating each face) to produce a monotone subdivision. The utility of these subdivisions was recognized by Lee & Preparata (1977), and they have been studied intensively ever since.

We will now sketch roughly some of the main ideas behind monotone subdivision search. Define a *separator* in a monotone subdivision as a connected collection of edges of the subdivision that meet every vertical line exactly once. These are monotone chains that separate the subdivision into two parts, above and below.

The main idea is to find a collection of separators that partition the subdivision into "horizontal" strips. Then a double binary search is performed: a vertical search on these

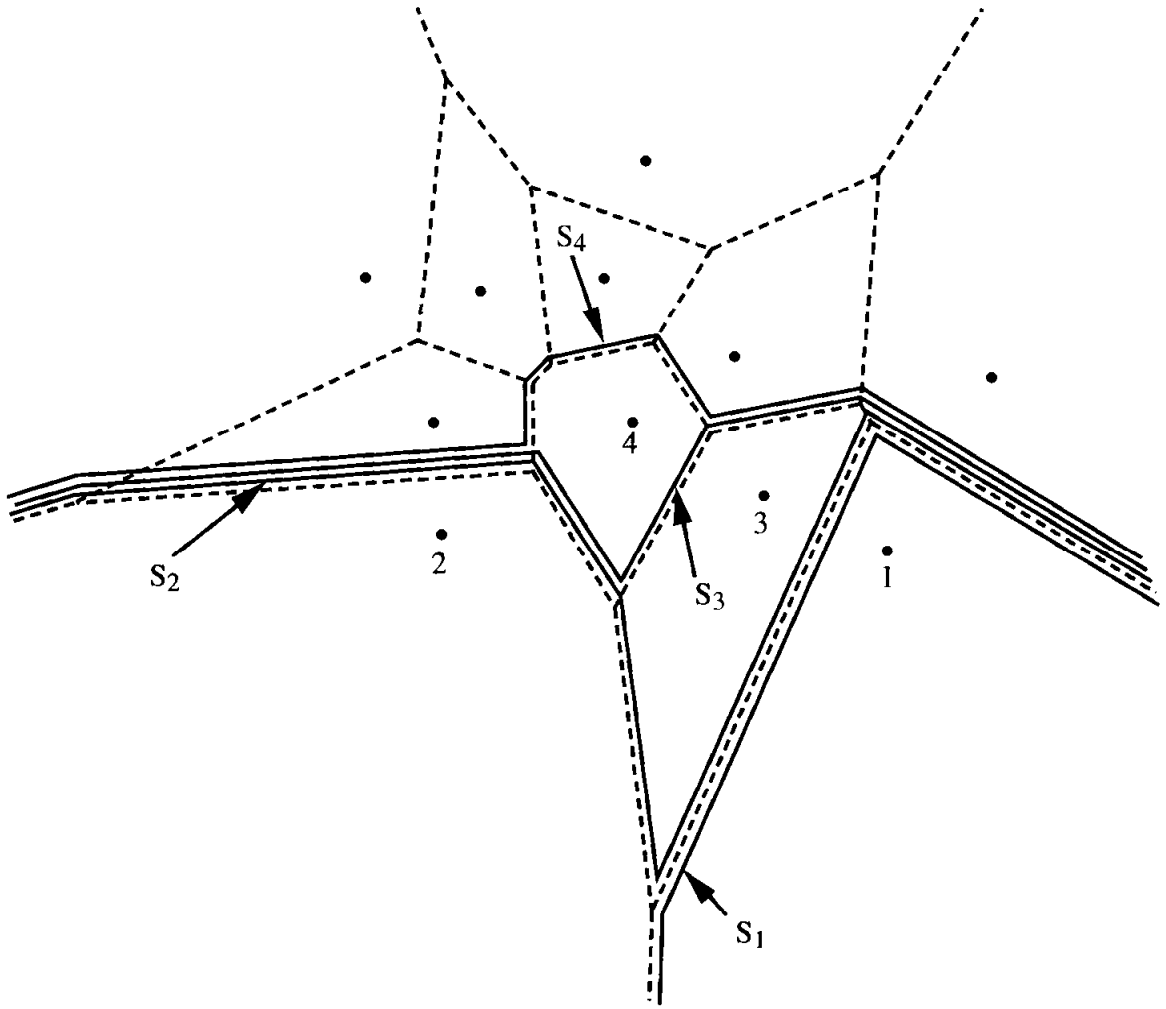


FIGURE 7.32 Separators in a monotone subdivision: $S_1 < S_2 < S_3 < S_4$.

strips to locate the query point between two separators and a horizontal search to locate it within one strip.

An example is shown in Figure 7.32. The subdivision is a Voronoi diagram, which is of course monotone. Four separators are shown. S_1 is the lowest, having only the Voronoi cell C_1 for point 1 below it. S_2 is the next highest, having C_2 and C_1 below it. Note that S_2 is above S_1 throughout their lengths. Similarly S_3 is above C_3, and S_4 is above C_4. This process could be continued, finding a collection of separators $S_1, S_2, \ldots, S_m$ that can be considered sorted vertically, with each pair of adjacent separators having one cell of the subdivision sandwiched between them.

Consider the problem of deciding whether a query point q is above or below some particular separator S_i. This can be accomplished via a horizontal binary search on the x coordinates of the vertices of S_i and the x coordinate of q, because S_i is monotone with respect to the x axis. Once the projection of q on the x axis is located between two endpoints of an edge e of S_i, it can be tested for above or below e. Since any S_i has $O(n)$ edges, the query "Is q above or below S_i?" can be answered in $O(\log n)$ time.

Now this query can be used repeatedly to perform a binary search on the collection of separators. First ask if q is above or below $S_{m/2}$. If it is below, query its relation to $S_{m/4}$; if above, query $S_{3m/4}$; and so on. This binary search will take $O(\log m)$ steps, each of which costs $O(\log n)$. Since $m = O(n)$, the total query time is $O(\log^2 n)$.

Of course this is asymptotically worse than what is achieved in Theorem 7.11.1. Moreover, it could require quadratic space to store the separators, due to the high degree of shared edges, as is evident in Figure 7.32. However, the algorithm is attractively simple, and it can be improved in both query time and space requirements to achieve the same asymptotic complexities as claimed in Theorem 7.11.1. These improvements are by no means straightforward and awaited the inventions of topological sorting and fractional cascading (Chazelle & Guibas 1986*a*; Chazelle & Guibas 1986*b*) among other ideas. See Edelsbrunner (1987, Chapter 11) for a thorough presentation.

7.11.4. Randomized Trapezoidal Decomposition

Randomized algorithms present a relatively recent, attractive alternative for point location. Here we present one such algorithm due to Seidel (1991), foreshadowed in Chapter 2. In Section 2.4.1, we used trapezoidalization to triangulate a polygon. The same general technique applies to more general objects than polygons. In particular, it works for collections of *noncrossing* segments: no two segments share a point interior to either, but they may share endpoints. Note that the edges of a polygon satisfy this definition. Let $S = \{s_1, \ldots, s_n\}$ be a collection of noncrossing segments. The goal is to extend horizontal chords left and right from each segment endpoint, partitioning the plane into "trapezoids." These are faces of two horizontal sides each, one of which may be degenerate, of zero length. Faces may be unbounded (although sometimes it is convenient to surround S with a large axis-aligned rectangle to ensure that all trapezoids are bounded). See Figure 7.33. There are $O(n)$ trapezoids; Exercise 7.11.5[1] asks for a proof that $3n + 1$ is a tight upper bound.

To simplify the details, we will assume that no two endpoints lie on a horizontal line (see Section 2.2). This limits the neighboring relations:

Lemma 7.11.2. *Each trapezoid has at most two trapezoids neighboring above, and two below, where neighboring trapezoids share a nonzero-length portion of a horizontal chord.*

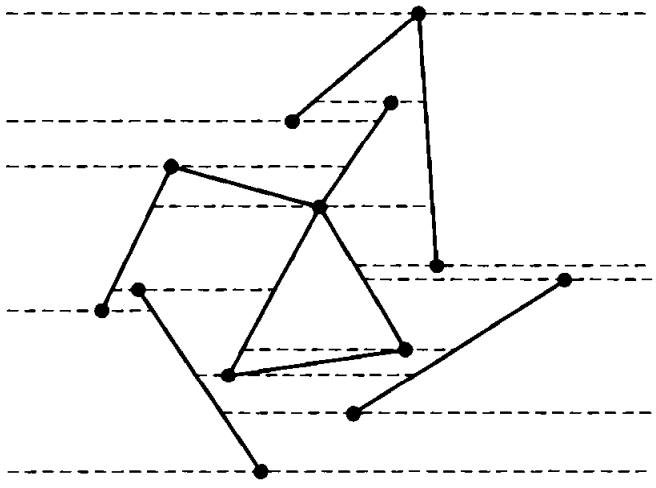


FIGURE 7.33 The trapezoid decomposition induced by $n = 10$ segments.

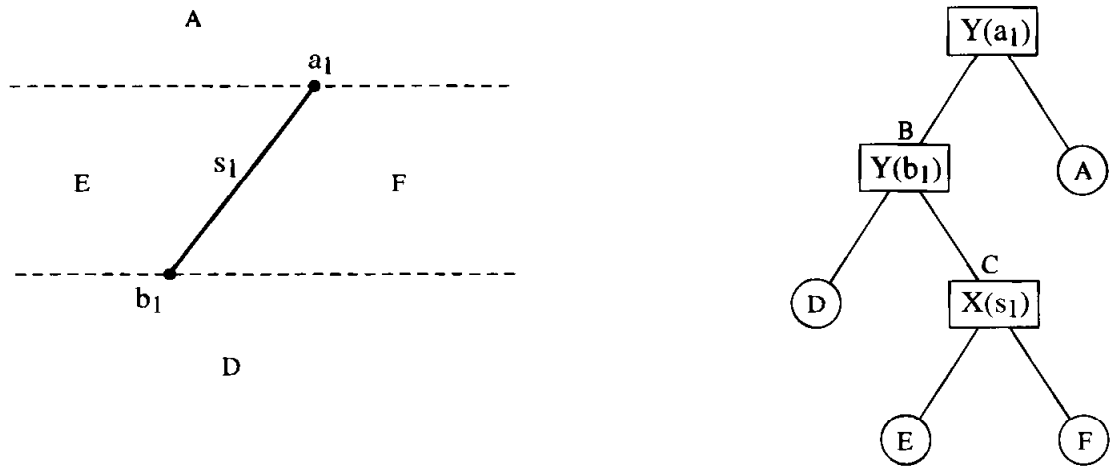


FIGURE 7.34 Search tree after inserting $s_1 = a_1b_1$.

Proof. Suppose a trapezoid had, for example, three neighbors above. The three sections of horizontal chord that form its top side cannot derive from a single endpoint, because each endpoint generates at most two: a left and right chord. Thus this upper side must contain at least two vertices, violating the assumption that no two endpoints lie on a horizontal line. □

This lemma allows us to represent the trapezoid locations with a binary search tree, binary because there are at most two neighbors. This clever search structure was developed in the late 1970s and was explicitly used by Preparata (1981). The version detailed here follows Seidel (1991), with the trapezoid decomposition algorithm growing out of work of Mulmuley (1990).

The search tree has three types of nodes:

1. internal X nodes, which branch left or right of a segment s_i;
2. internal Y nodes, which branch above or below a segment endpoint; and
3. leaf trapezoid nodes.

The search tree is constructed incrementally. Let $s = ab$ be a new segment to be added to an existing structure. The update of the structure can be partitioned into several steps:

1. Add endpoints a and b. For each endpoint, find the trapezoid that contains it by searching in the tree. Split this trapezoid with a Y node.
2. Add segment s.
   (a) "Thread" s through the partition, identifying each trapezoid cut by s.
   (b) On each side of s, merge trapezoids whose left and right bounding segments are the same.
   (c) Create X nodes for all trapezoids separated by s.

We now run through the construction of the search tree for three particular segments $\{s_1, s_2, s_3\}$. Although the details are somewhat tedious, patience will be rewarded by better appreciation of the beauty of the resulting data structure. Throughout the figures (7.34–7.39), below arcs are drawn left of above arcs, and left arcs drawn left of right arcs.

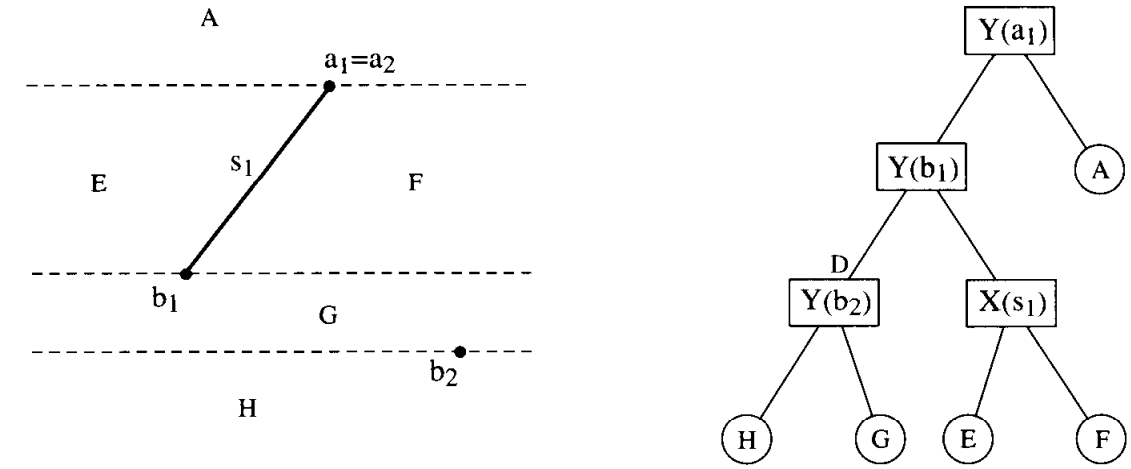


FIGURE 7.35 Search tree after inserting a_2 and b_2.

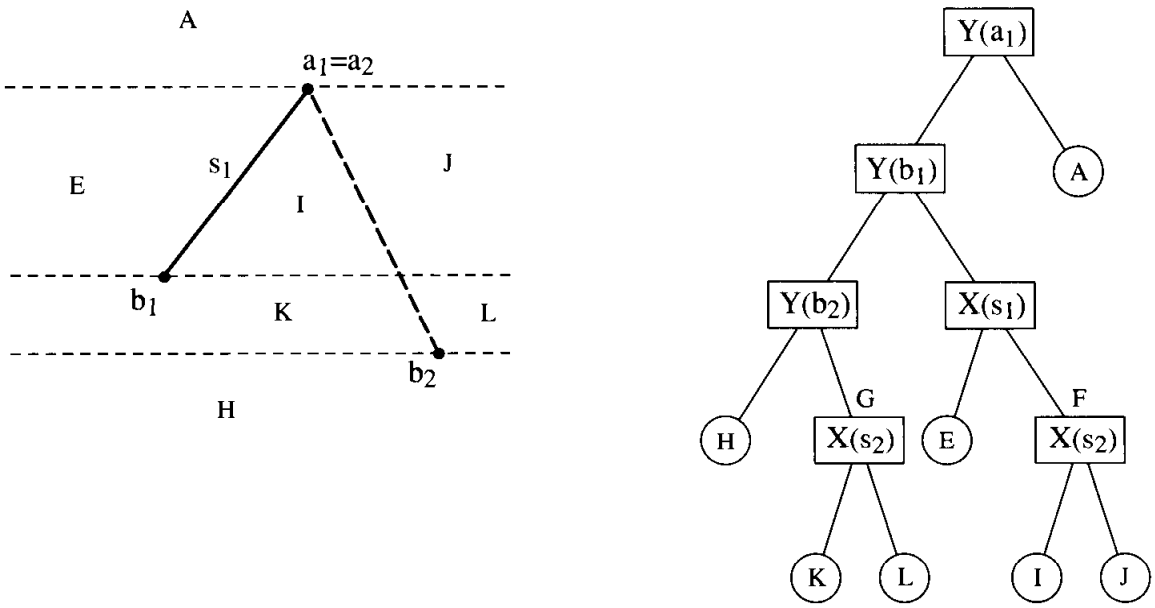


FIGURE 7.36 Search tree after threading s_2.

1. Add a_1. Split the original region (the whole plane) into A and B, above and below respectively. Create a $Y(a_1)$ node as their parent.
2. Add b_1. Locate $b_1 \in B$. Split B into C and D with a $Y(b_1)$ parent node.
3. Thread s_1. Split C into E and F, left and right respectively, with an $X(s_1)$ parent node. The search tree is now as shown in Figure 7.34.
4. Add $a_2 = a_1$. No change to the structure occurs.
5. Add b_2. Locate $b_2 \in D$. Split D into G and H with a $Y(b_2)$ node. See Figure 7.35.
6. Thread s_2. Split F into I and J, and split G into K and L, both with separate $X(s_2)$ parents. See Figure 7.36.
7. Merge along s_2. Merge J and L one region M, because they share the same left and right bounding segments (s_2 and $+\infty$). Rewire tree accordingly. See Figure 7.37.
8. Add a_3. Locate $a_3 \in I$, and split into N and O with a $Y(a_3)$ node.
9. Add b_3. Locate $b_3 \in K$, and split into P and Q with a $Y(b_3)$ node. See Figure 7.38.
10. Thread s_3. Split O into R and S, and split P into T and U, both with $X(s_3)$ parents.
11. Merge along s_3. Merge S and U into one trapezoid V. See Figure 7.39.

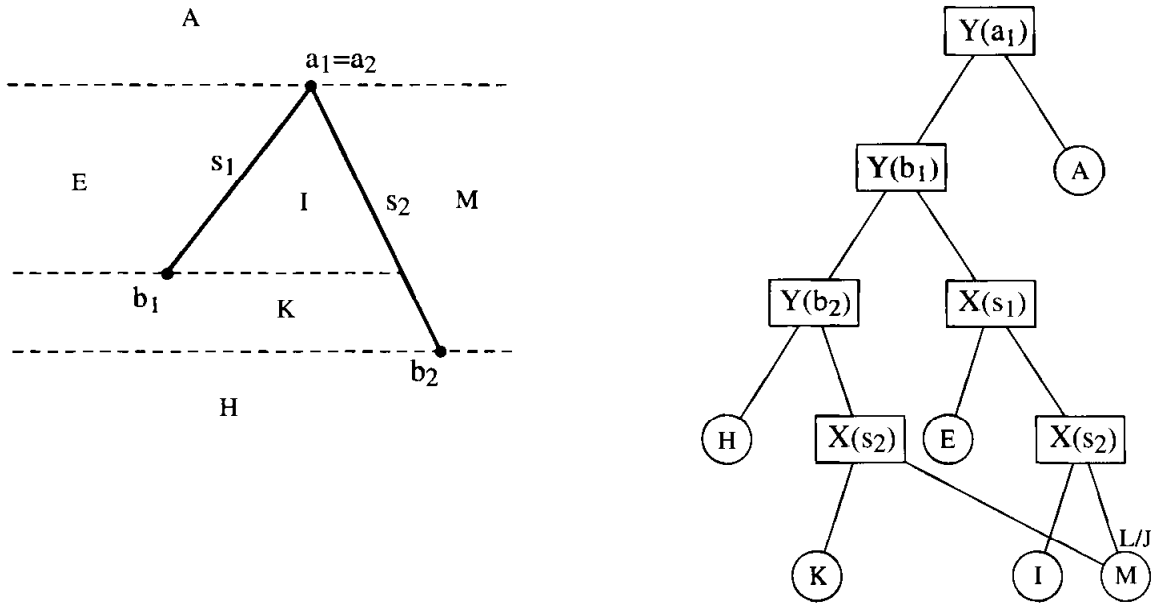


FIGURE 7.37 Search tree after merging regions along s_2.

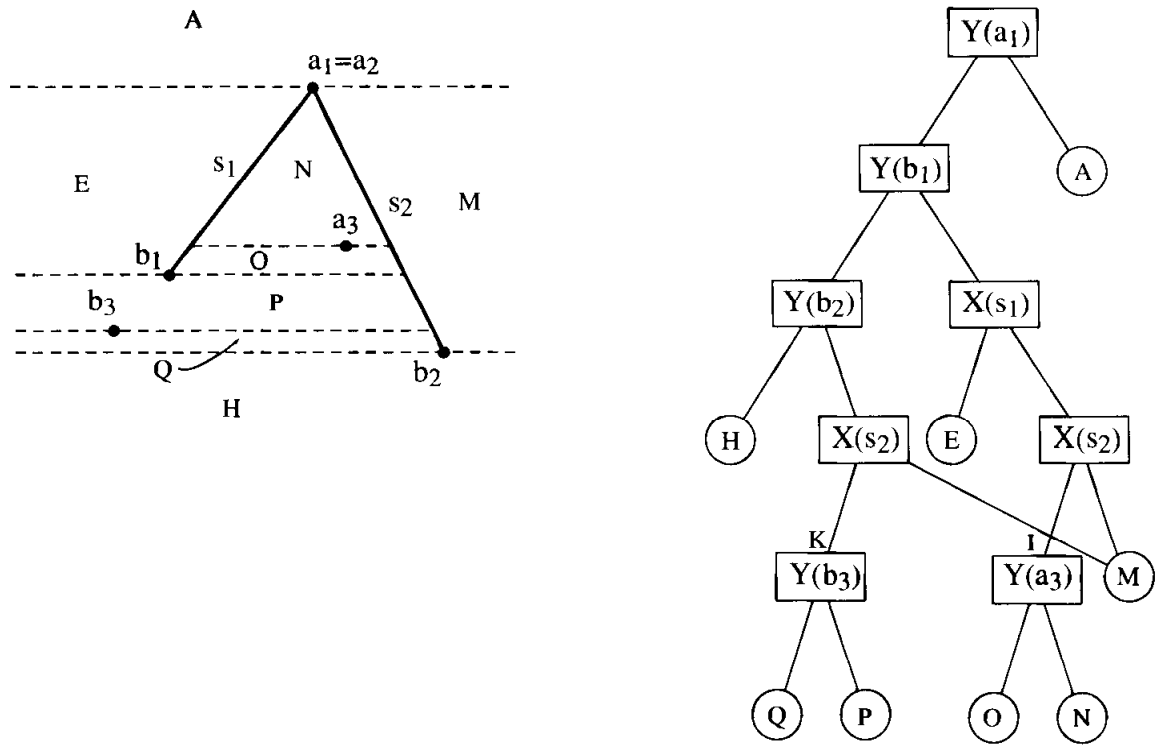


FIGURE 7.38 Search tree after inserting a_3 and b_3.

Let us use the final search tree in Figure 7.39 to locate q in the shaded trapezoid V. Point q is below a_1 and b_1 but above b_2, so from the root the path bends: left, left, right. q is left of s_2, so the left branch is taken at the $X(s_2)$ node. q is above b_3, and finally it is right of s_3. The search path leading to V is highlighted in the figure. Note that not all trapezoids have a unique paths from the root. If q were in V but above b_1, then V would be reached by another route.

It is clear from our description that the search structure obtained is dependent on the order in which the segments are inserted. A "bad" order could result in a thin tree of

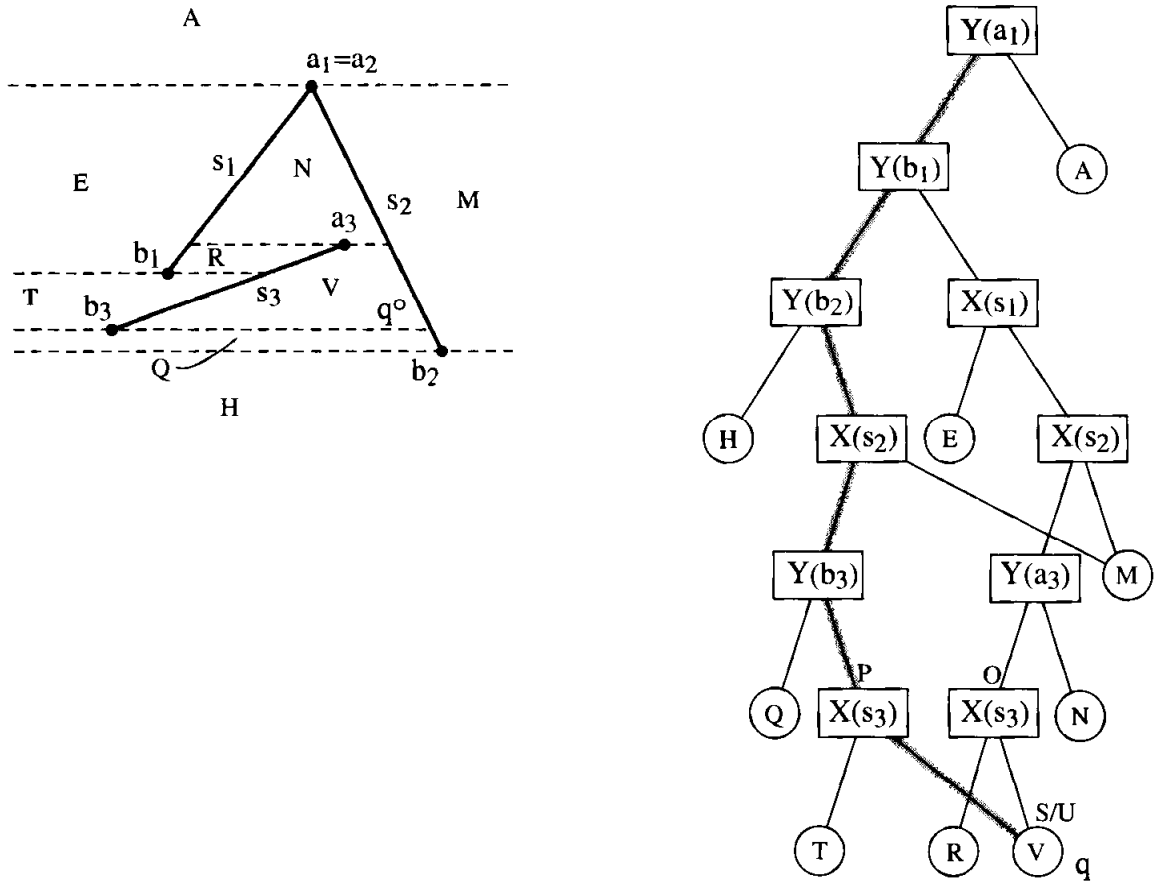


FIGURE 7.39 Search tree after inserting s_3. The search path for $q \in V$ is highlighted.

height $\Omega(n)$; a "good" order will yield a bushy tree of height $O(\log n)$. And the query time is proportional to this height. As mentioned in Section 2.4.1, if the segments are added in random order (i.e., each of the $n!$ orders is equally likely), then it can be proven that the expected height is $O(\log n)$; moreover, the expected time to build the entire structure is $O(n \log n)$. Although it is conceivable that these expectations are weak in the sense that they are broad averages masking nasty performance, in fact it can be further proved that the probability that the tree height significantly exceeds $O(\log n)$ is small.[23] Thus this clean and practical algorithm achieves expected $O(\log n)$ query time with expected $O(n \log n)$ preprocessing.

Planar point location remains an active area of research. Not only does there remain room for improvement on the basic problem discussed here, but two important related problems are very much in flux at this writing: "dynamic" planar point location, where the subdivision changes, for example, by insertions or deletions of Voronoi sites, and point location in subdivisions of three-dimensional and higher spaces.

7.11.5. Exercises

1. *Number of trapezoids.* Prove that the number of trapezoids produced by the trapezoidal decomposition algorithm is at most $3n + 1$ for n segments (de Berg et al. 1997, Lem. 6.2). Include in your count a trapezoid above all segments and one below; or equivalently, surround the segments by a large rectangle and count all trapezoids in the rectangle.

[23] See, e.g., de Berg et al. (1997, Lem. 6.7).

2. *Detection of intersection of convex polygons.* Develop an algorithm for reporting whether or not two convex polygons of n and m vertices intersect. Try to achieve $O(\log(n + m))$ time (Chazelle & Dobkin 1987).
3. *Interval trees.* Preprocess a set of n intervals I (on a line) with integer endpoints so that they can be efficiently queried. Consider three types of queries (the preprocessing need not be the same for all three):
   a. Is x in some interval in I?
   b. Within how many intervals of I does x lie?
   c. Does the interval $[a, b]$ intersect any interval of I?
4. *Length of union of intervals.* Design an algorithm to find the total length covered by the union of n intervals.
5. *Empty circle queries* (Michael Goodrich). Given a set S of n points in the plane, sketch a good method for constructing an efficient data structure to quickly answer empty circle queries. An *empty circle query* for a query point q asks for the largest circle that has q as its center and does not contain any point of S in its interior.
6. *Cops and robbers* (Michael Goodrich). Suppose you are given two sets of n points in the plane, P and R. The points in P represent "police officers" and the points in R represent "robbers." A point q in the plane is *safe* if it is inside the triangle formed by three points in P. A point q in the plane is *robbed* if it is not safe and is inside the triangle formed by three points in R. A point q is *suspect* if it is neither safe nor robbed. Describe an efficient data structure to determine, for any query point q, whether q is safe, robbed, or suspect.

8

Motion Planning

8.1. INTRODUCTION

The burgeoning field of robotics has inspired the exploration of a collection of problems in computational geometry loosely called "motion planning" problems, or more specifically "algorithmic motion planning" problems. As usual, details are abstracted away from a real-life application to produce mathematically "cleaner" versions of the problem. If the abstraction is performed intelligently, the theoretical explorations have practical import. This happily has been the case with motion planning, which applies not only to traditional robotics, but also to planning tool paths for numerically controlled machines, to wire routing on chips, to planning paths in geographic information systems (GIS), and to virtual navigation in computer graphics.

8.1.1. Problem Specification

The primary paradigm we examine in this chapter assumes a fixed environment of impenetrable obstacles, usually polygons and polyhedra in two and three dimensions respectively. Within this environment is the "robot" R, a movable object[1] with some prespecified geometric characteristics: It may be a point, a line segment, a convex polygon, a hinged object, etc. The robot is at some initial position s (*s*tart), and the task is to plan motions that will move it to some specified final position t (*t*erminus), such that throughout the motion, collision between the robot and all obstacles is avoided. A *collision* occurs when a point of the robot coincides with an interior point of an obstacle. Note that sliding contact with the boundary of the obstacles does not constitute a collision. A collision-avoiding path is called a *free path.*[2] Often there are restrictions on the type of motions permitted.

Within this general class of problems, we consider three specific questions, each asking for more information than the preceding:

1. Decision question: Does there exist a free path for R from s to t?
2. Path construction: Find a free path for R from s to t.
3. Shortest path: Find the shortest free path for R from s to t.

A solution to (3) solves (2), which in turn solves (1). So the decision question is the easiest, although in practice it is usually answered by finding some path, thereby solving

[1] There is no assumption of autonomous capability.

[2] Most authors follow Schwartz & Sharir (1983*a*) in calling this a *semifree path*, using "free path" to imply no boundary contact (e.g., Latombe (1991, p. 10)). We will not require the distinction.

(2).[3] We will see, however, an example in Section 8.6 where the decision question is much simpler than actually finding a path.

The sense in which a path is "shortest" is not always clear. If the robot is a disk, it is clear. But suppose the robot is a line segment that is permitted to rotate (Section 8.5). Then several definitions of "shortest" are conceivable, and once one is selected, it is extremely difficult to find a shortest path.

8.1.2. Outline

We will only consider shortest paths for the simplest problem instance: when the robot is a point (Section 8.2). We will then examine two of the better-understood motion planning problems: translating a convex polygon (including a partial implementation) (Section 8.4) and moving a "ladder" (a line segment) (Section 8.5). Next we study moving a hinged robot "arm" (Section 8.6). This section includes code to position an n-link arm to reach a specified hand location. Lastly we look at the fascinating problem of separating interlocking puzzle pieces (Section 8.7).

8.2. SHORTEST PATHS

In this section we examine the problem of finding a shortest path between two given points s and t amidst a collection of disjoint polygonal obstacles with a total of n vertices. An example is shown in Figure 8.1: Path A is shorter than path B and is in fact the unique shortest path connecting s and t and avoiding the interior of all obstacles. We assume that s and t are not interior to any polygon, which, together with the assumption of obstacle disjointness, implies that there is always some path; so the problem is to find the best one.

8.2.1. Visibility Graphs

A usual first step in optimizing over an infinite set of possibilities (and there are an infinite number of paths between s and t) is to reduce the set to a finite list of bona fide candidates. This is achieved in this instance by the observation that a shortest path is composed of segments whose endpoints are either s, t, or vertices of the polygons. Since s and t "act like" polygon vertices in this sense, it simplifies the discussion if we treat s and t as point polygons of one vertex each. Then the observation can be strengthened to this statement:

Lemma 8.2.1. *A shortest path is a subpath of the visibility graph of the vertices of the obstacle polygons.*

Visibility graphs were mentioned in Chapter 6 (Section 6.1). A *visibility graph* of a set of polygons is a graph whose nodes correspond to vertices of the polygons, and whose arcs correspond to vertices x and y that can "see" one another, in the sense that

[3] Problem (2) is sometimes called the "Find-Path" problem (Brooks 1983).

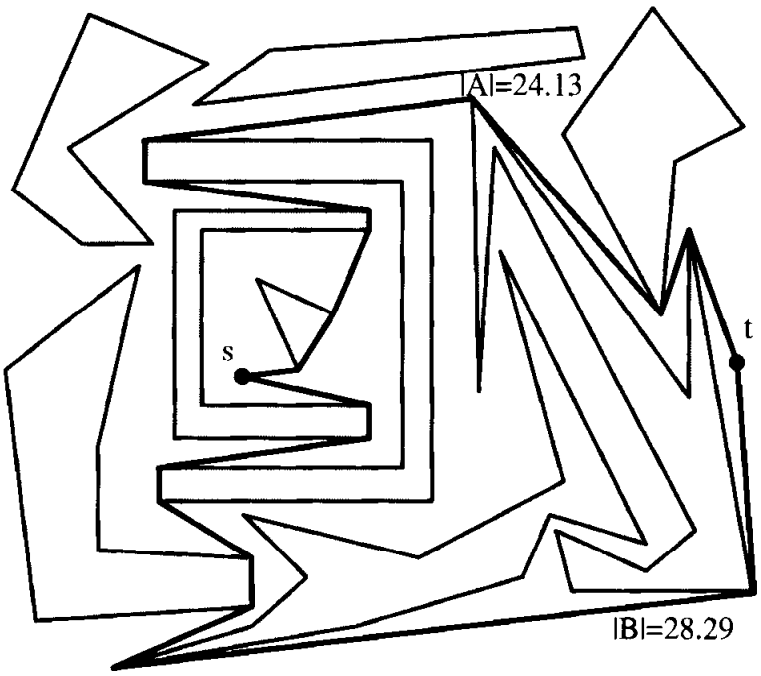


FIGURE 8.1 A is the shortest path from s to t.

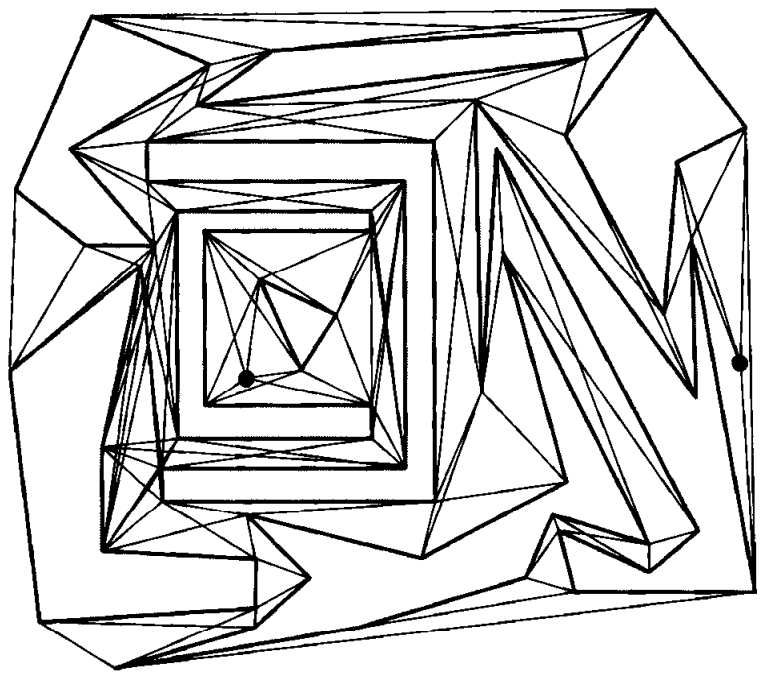


FIGURE 8.2 Visibility graph for the example in Figure 8.1.

the segment xy does not meet the interior of any polygon. Note that xy may intersect the boundaries of the polygons, so that, for example, the edges of the polygons are in the visibility graph. This is the same notion of visibility used in Chapter 1 (Section 1.1.2.2), except now operating exterior to the polygons. The visibility graph for the polygons in Figure 8.1 is shown in Figure 8.2.

Lemma 8.2.1 can be justified in three steps:

1. The path is polygonal. Suppose to the contrary that the path contains a curved section C. It cannot be the case that all of C lies along polygonal boundaries, as

these boundaries are not curved. Then there must be a convex subsection of C that does not touch any polygon and that can be shortcut by a straight segment, contradicting the assumption that the path is shortest.

2. The turning points of the path are at polygon vertices: Any turn in "free space" can be shortcut.
3. The segments of the path are visibility edges. This follows from the definition of visibility and the definition of what constitutes a free path.

Since the visibility graph is finite, this lemma establishes that there are only a finite number of candidate paths from s to t to search. However, the number of paths in the graph might be exponential in n, and we will need further analysis before we have a practical algorithm. First we briefly consider constructing the visibility graph.

8.2.2. Constructing the Visibility Graph

Constructing the visibility graph for a set of polygons is a fascinating problem with many applications, and it has been studied extensively. It would be a distraction from our main focus on motion planning to explain this, however, so we will make just a few remarks.

Finding visibility graph edges is nearly identical to finding polygon diagonals; the only differences are inessential: We have several polygons instead of just one, and consider exterior instead of interior visibility. An $O(n^3)$ algorithm is immediate: For each vertex x and for each vertex y, check xy against every edge. The graph can have a quadratic number of edges, so $\Omega(n^2)$ is a lower bound on any algorithm. We mentioned in Section 6.1 that use of arrangements leads to an optimal $O(n^2)$ algorithm (O'Rourke 1987, Chapter 8). After a long pursuit, an output-size sensitive algorithm was found by Ghosh & Mount (1991): $O(n \log n + E)$ for a graph with E edges. Of course $E = O(n^2)$, but often (as in Figure 8.2), E is much smaller than $\binom{n}{2}$.

8.2.3. Dijkstra's Algorithm

Assuming we have the visibility graph constructed and stored in some convenient data structure, the next question, and one on which we will concentrate, is how to find a shortest path in this graph. This is an instance of a problem studied in graph theory: finding a shortest path in a weighted graph. In our case, the "weights" on the edges of the graph are the lengths of those edges, the Euclidean distances between the endpoints.

A gem of an algorithm for this problem was found by Dijkstra (1959). I will start explaining the key idea on a small example before turning to implementation details.

Spreading Paint

Consider the portion of a visibility graph shown in Figure 8.3; not all visibility edges are included to reduce clutter. Imagine pouring paint on the source node s, and suppose the visibility edges are thin pipes of the same diameter, so that the paint spreads evenly along all visibility edges at a uniform rate, one unit of length per unit of time. The first vertex of the visibility graph to be hit by paint is a, shown in Figure 8.3(a): Its distance of 1.93 to s is the shortest length of all visibility edges incident to s. After just another 0.12 time units, vertex b is hit (part (b) of the figure), and the paint has crept a bit further along all other paths. At time 3.33, vertex c is hit; at time 4.61, vertex d is reached; and so on.

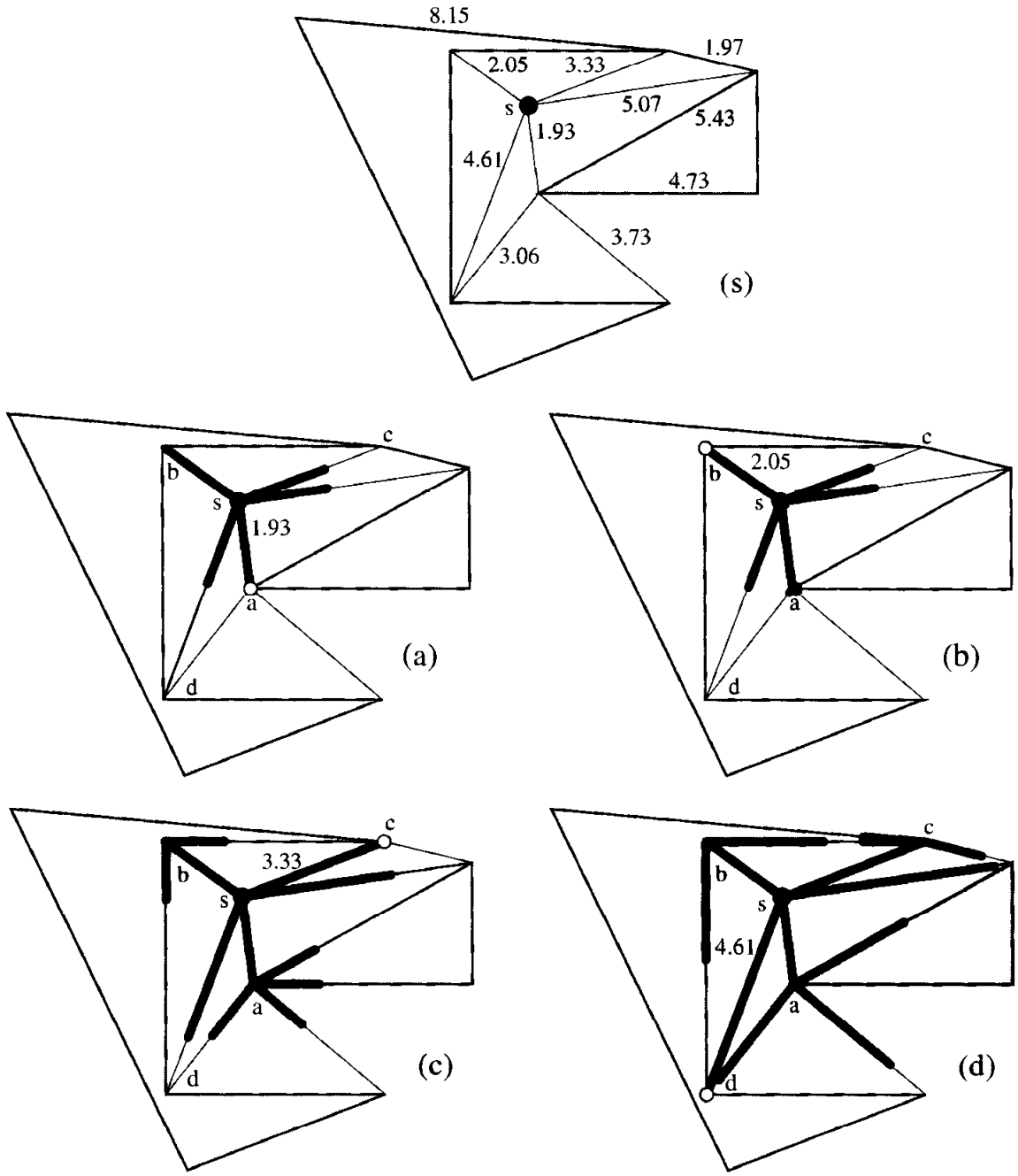


FIGURE 8.3 Visibility graph at the start s, and after vertices a, b, c, and d are reached respectively.

The idea of Dijkstra's algorithm is to simulate this paint-spreading process. Then when the destination t is reached by the paint, the simulated time gives the length of the shortest path. By storing information at each node indicating from which direction paint first reached it, it is possible to trace backwards and find the complete shortest path to any node. This is roughly equivalent to tagging each paint molecule with its path so far, so that when the first molecule reaches t, its path is known.

Algorithm

Let G be the visibility graph. Dijkstra's algorithm avoids a continuous simulation of the paint creeping down each visibility edge, recognizing that discrete steps suffice. The algorithm maintains a tree $T \subset G$ rooted at s that spans all those nodes so far reached by

paint: the discrete paint frontier. At each step, the edges incident to every node of T are examined, and one edge is added to T that (a) reaches a node x outside of T: $x \in G \setminus T$, and (b) such that the distance to x from s is shortest among all nodes satisfying (a). The point of (b) is to ensure that x is the next node to be reached by paint.

Consider the step of the algorithm after Figure 8.3(b). Nodes s, a, and b have been reached by paint, so $T = \{sa, sb\}$. Now all edges incident to these three nodes are examined, and their lengths added to the shortest distance to the nodes. So the edge ad has length 3.06, yielding a distance $1.93 + 3.06 = 4.99$ from s. Edge sc has length 3.33, yielding a distance $0 + 3.33 = 3.33$ from s. One can see that sc is the appropriate edge to add to T.

We can state Dijkstra's algorithm succinctly (see Algorithm 8.1).

Algorithm: DIJKSTRA'S ALGORITHM
$T \leftarrow \{s\}$
while $t \notin T$ do
    Find an edge $e \in G \setminus T$ that augments T to reach a node x
        whose distance from s is minimum
    $T \leftarrow T + \{e\}$

Algorithm 8.1 Dijkstra's shortest path algorithm.

Time Complexity

Analyzing the time complexity of Dijkstra's algorithm is an interesting exercise in the analysis of algorithms,[4] but tangent to our interests here. So we will just sketch some issues and leave a full analysis as an exercise.

The while loop cannot execute more than n times, as each edge added to T reaches a new node, and there are n nodes total. But the number of candidate edges to examine for each step of the loop is potentially $O(n^2)$, since a visibility graph can have a quadratic number of edges. This gives a crude bound of $O(n^3)$. One can see, however, that it is wasteful to examine these edges afresh at each iteration, and in fact the algorithm can be implemented to run in $O(n^2)$ time (Exercise 8.2.4[2]). Together with the $O(n^2)$ construction of the visibility graph, this gives the following theorem.

Theorem 8.2.2. *A shortest path for a point moving among polygon obstacles with a total of n vertices can be found in $O(n^2)$ time and space.*

8.2.4. Exercises

1. *Strengthen visibility graph lemma?* Can Lemma 8.2.1 be strengthened to say that a shortest path never includes a reflex vertex of a polygon?
2. *Complexity of Dijkstra's algorithm.* Show that Dijkstra's algorithm can be implemented to run in $O(n^2)$ time.

[4] See, e.g., Albertson & Hutchinson (1988, pp. 390–4) or Chartrand & Oellermann (1993, p. 106).

3. *Disk obstacles.* Design and analyze an algorithm for finding a shortest path for a point amidst n disjoint disk obstacles.
4. *Visibility graph* [programming]. Write a program to construct the visibility graph for a set of polygons. Just implement the brute-force $O(n^3)$ algorithm. Use as much of the triangulation code from Chapter 1 as possible.
5. *Number of shortest paths.* For polygon obstacles with a total of n vertices (not counting s and t), what is the largest number of equal-length shortest paths possible?
6. *Unit disks.*
   a. Suppose that all obstacles are unit disks, disjoint as usual. Must a shortest path between s and t be monotonic with respect to the segment st?
   b. [open] Design a subquadratic algorithm for finding a shortest path in the presence of unit disks.
7. *Shortest path in a polygon* (Guibas, Hershberger, Leven, Sharir & Tarjan 1987). Design an algorithm to find a shortest path between two points inside a polygon. Try to beat $O(n^2)$.

8.3. MOVING A DISK

We now commence our study of motion planning algorithms where the goal is to find any path (if one exists), rather than a shortest path. We proceed in three increasingly complex stages: moving a disk, translating a convex polygon (Section 8.4), and moving a segment with rotation (Section 8.5). Only in this latter section will we consider a variety of approaches.

Suppose the robot R is a disk centered on s, and the goal is to move it so that it becomes centered on t, never penetrating an obstacle during its motion. As before, the obstacles are disjoint polygons. The path shown in Figure 8.4(a) is not a free path, as the robot is too wide to fit through the indicated channel. There is a useful way to view the problem that makes it obvious that this path is not free; and this view extends nicely to more complicated situations. Let r be a reference point on the moving disk R, say its center. Then r cannot get too close to any particular polygon P – in fact r cannot move closer than the disk radius ρ to P. This suggests that we consider an "expanded" obstacle P^+ for the purposes of moving the point r, expanded by ρ. Effectively we shrink the robot to a point, and grow the obstacles by ρ, thereby reducing the problem to moving a point among obstacles.

What does the enlarged P^+ look like for any given P? Its boundary can be obtained by tracing out what happens to r when the disk is moved around the boundary of P, tracking ∂P. This is illustrated in Figure 8.4(b). It should be clear that if r stays outside of P^+, R will not intersect P; and if r is inside P^+ then R must intersect P. Figure 8.4(b) now makes it quite clear that the path in (a) is not possible, as the grown obstacles overlap in the channel.

Our description of P^+ is somewhat vague. We can be more precise by using the notion of the "Minkowski sum" of two point sets.

8.3.1. Minkowski Sum

Let A and B be two sets of points in the plane. If we establish a coordinate system, then the points can be viewed as vectors in that coordinate system. Define the *sum* of A

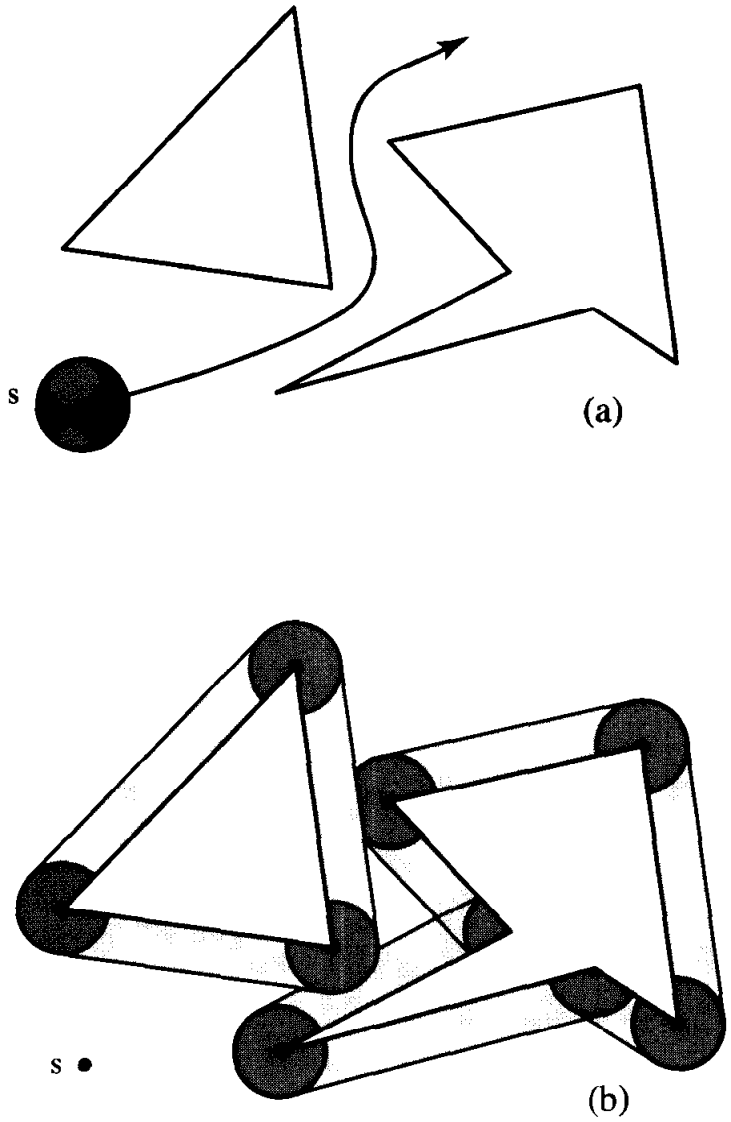


FIGURE 8.4 Enlarging a polygon by a disk: (a) a nonfree path; (b) expanded obstacles.

and B in the most natural manner possible: $A \oplus B = \{x + y \mid x \in A,\ y \in B\}$, where $x + y$ is the vector sum of the two points. This is known as the *Minkowski sum* of A and B.[5]

It will be a little easier to grasp the meaning of this abstract idea by considering the Minkowski sum of a point x and a set B: $x \oplus B = \{x + y \mid y \in B\}$. This is just a copy of B translated by the vector x, for each point y of B is moved by x. So $A \oplus B = \bigcup_{x \in A}(x \oplus B)$ is the union of copies of B, one for each $x \in A$. Now suppose A is a polygon P and B is a disk R centered on the origin. Then $P \oplus R$ can be viewed as many copies of R, translated by x for all $x \in P$. Since R is centered on the origin, $x \oplus R$ will be centered on x. So $P \oplus R$ amounts to placing a copy of R centered on top of every point of P. Now it should be clear that $P \oplus R$ is precisely the expanded region P^+.

Let us examine the expansion of the triangle obstacle in Figure 8.4(b). A copy of R is placed at each vertex of the triangle when x is a vertex, and when x lies on an edge, the

[5]It is also known as the *pointwise sum* of A and B.

tangents between these vertex disks are generated. For x interior to the triangle, $x \oplus R$ lies inside P^+

8.3.2. Conceptual Algorithm

We return now to the problem of moving a disk among polygonal obstacles. We will sketch a conceptual algorithm but will not provide details.

First, grow every obstacle by the disk R by constructing the Minkowski sum with R. We have not described how the grown obstacles can be constructed algorithmically; this issue we defer to the next section. Second, form the union of the grown obstacles. Again, we will not discuss how this can be done; it is not trivial. If t, the destination, is in a different "component" of the plane than is s, then there is no free path from s to t. If they are in the same component, then there is a path, and the shortest path can be found by modifying the visibility graph to include the appropriate arcs of the circles. We will not describe this algorithm any further, but it can be accomplished all in $O(n^2 \log n)$ time (Chew 1985).

8.4. TRANSLATING A CONVEX POLYGON

When the robot is a convex polygon, we come to a serious complication: Rotation of the robot might be necessary to move from one location to another. We will defer consideration of rotation until Section 8.5. Here we restrict the motion to translations only.

The task is still more complicated than moving a disk, but fortunately the idea of growing obstacles by Minkowski sums still works. We explain the basic idea with an example before discussing algorithmic details.

8.4.1. Minkowski Sum Example

Let the robot R be a square, and choose the reference point r to be its lower left corner. Consider a polygon P such as the pentagon shown in Figure 8.5. As R moves around ∂P, r traces out the boundary of P^+, an expanded obstacle that defines the region of the plane where r cannot penetrate. Note that the situation is somewhat different from that with a disk, since we chose the reference point at a corner of R: P grows by a different amount along each edge. For example, along edge e_0, P and P^+ match, since it is possible for r to touch e_0. And the offset of P^+ from edges e_1 and e_4 of P is different: the width of R horizontally from e_1, and the height of R vertically from e_4. Also note what happens near the reflex vertex: If we trace the complete lengths of the two edges incident to it, r traces out a self-crossing path, whose "outside" nevertheless precisely represents the physical limits of R's approach to P.

8.4.2. Minkowski Addition

With R a disk, we argued that $P^+ = P \oplus R$. But that clearly will not work in Figure 8.5: For example, $P \oplus R$ will "stick out" beyond edge e_0 of P, but P^+ does not. The

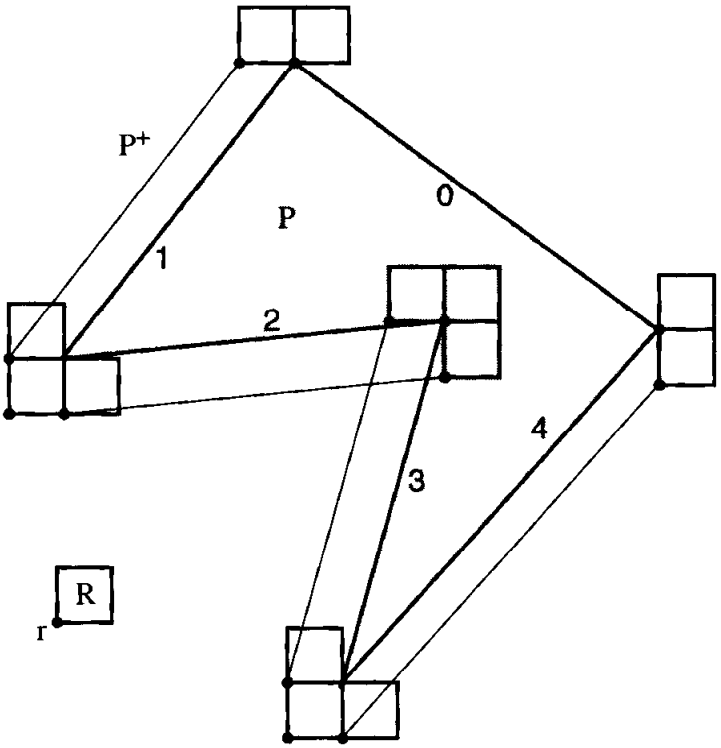


FIGURE 8.5 Growing P by R produces P^+. Critical placements of R near the vertices of P are shown.

appropriate computation is rather to take the Minkowski sum of P with a reflection of R through the reference point r. Since r is the origin for the purposes of the Minkowski sum formulation, this reflected version of R is simply $-R$, where every point of R is negated. The intuitive reason for the need for reflection is that each point $p \in R$ (say the upper right corner of R) has the effect of holding r away from ∂P by $-p$. Now we can see that P^+ in Figure 8.5 is $P \oplus -R$. Note that because a disk is centrally symmetric about its center, $R = -R$, and so this new formulation is consistent with our presentation in the previous section. Because the term "Minkowski subtraction" is sometimes used for another concept (Guggenheimer 1977), we will continue to call it Minkowski addition.

We formalize our discussion with a slightly more general claim.[6]

Theorem 8.4.1. *Let R be a region (the robot) and $r \in R$ be a reference point. Let P be an obstacle. Then the region $P^+ = P \oplus -R$ is the set of points forbidden to r in the sense that:*

1. *If R is translated so that r is strictly interior to P^+, then R penetrates P.*
2. *If R is translated so that r lies on ∂P^+, then ∂R touches ∂P.*
3. *If R is translated so that r is strictly exterior to P^+, then $R \cap P = \emptyset$.*

This is slightly more general in that neither R nor P need be convex, nor must they even be polygons. But we continue to assume in this section that both are polygons and that R is convex. For convenience we'll take r to be the origin.

[6]This seems to have been used first by Jarvis (1983).

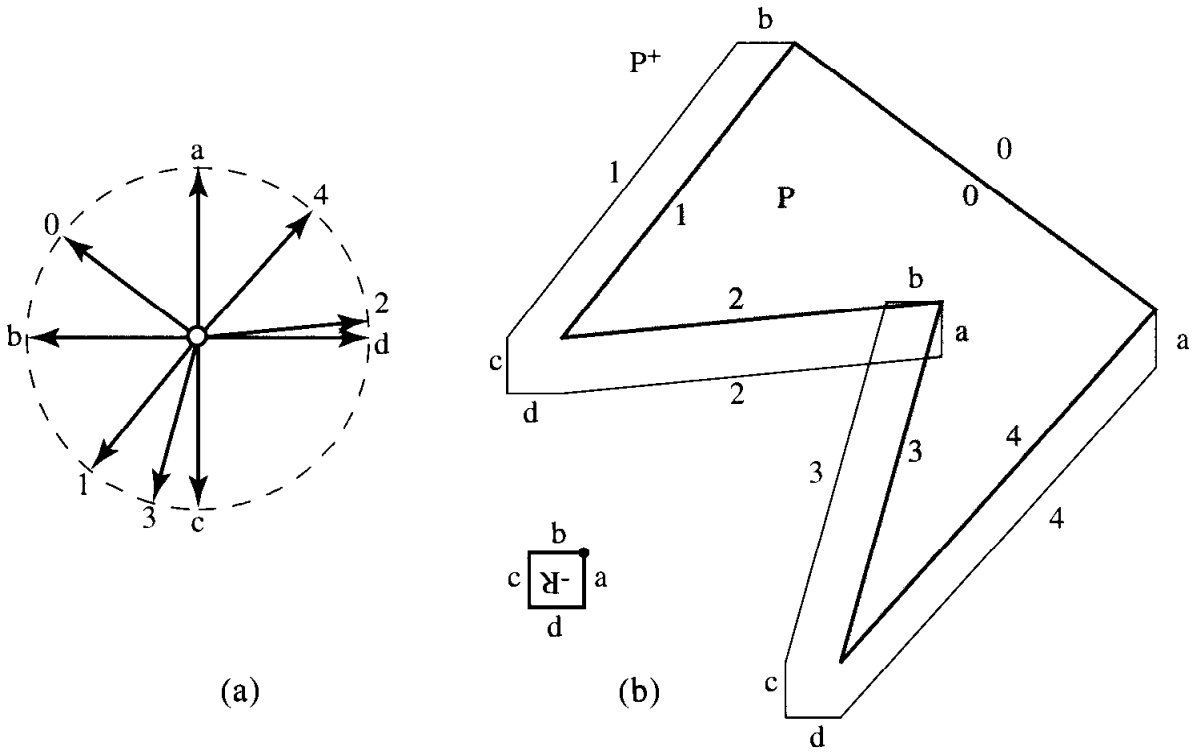


FIGURE 8.6 (a) Star diagram of the edge vectors; (b) convolution edges labeled with P or $-R$ labels.

8.4.3. Constructing the Minkowski Sum

We now sketch a method for constructing the Minkowski sum of two polygons, part of which we implement in Section 8.4.4 below. In order to keep focused on motion planning, rather than on the fascinating problems it engenders, full details are not provided, but rather these are relegated to the exercises.

We continue to use the example started in Figure 8.5. Figure 8.6(b) shows $P^+ = P \oplus -R$, with edges of P labeled $0, \ldots, 4$, edges of $-R$ labeled $a, \ldots, d$, both according to counterclockwise traversals, and edges of P^+ labeled among $\{0, 1, 2, 3, 4, a, b, c, d\}$ according to which edge of P or $-R$ "generates" it. Thus when R scrapes along edge 2 of P, the reference point r traces out a parallel edge of P^+ we label 2; and when R scrapes along c vertically at the vertex at the intersection of edges 1 and 2 of P, we label that edge of P^+ c also.

Note that we have labeled the entire self-intersecting polygonal path τ that "bounds" P^+, including edges in the vicinity of the reflex vertex that are inside P^+. It is easiest to approach P^+ by first constructing τ, which is sometimes called the *convolution* of P and $-R$ (Guibas, Ramshaw & Stolfi 1983).

The pattern of labels of the edges of τ can be neatly understood from a "star" diagram of the edge vectors of P and $-R$, shown in Figure 8.6(a). If we think of P as large and R as small, as in our running example, then roughly speaking τ has edges corresponding to those of P, interspersed with some edges of $-R$. Indeed one can see that the sequence of labels for τ, $(0, b, 1, c, d, 2, a, b, 3, c, d, 4, a)$, includes $(0, 1, 2, 3, 4)$ as a subsequence. The star diagram gives a mechanism for predicting the interspersed labels of $-R$.

Consider each edge a vector directed according to a counterclockwise traversal, and move them all to a common origin, as shown (normalized to unit length) in Figure 8.6(a).

Call this arrangement of edge vectors the *star diagram*. Now starting with 0, circle around the star counterclockwise. Between the indices i and $i+1$ of P edges, write down all the indices of $-R$ encountered. Thus between 0 and 1, b is encountered, yielding the subsequence $(0, b, 1)$. Between 1 and 2, c and d are encountered, yielding the subsequence $(1, c, d, 2)$. Continuing in this manner we generate the entire sequence for τ by the time 0 is reached again. We will discuss an implementation in a moment.

To obtain the Minkowski sum P^+ from τ, there remains work to find the self-intersections of the convolution, another interesting problem we will not explore (Guibas et al. (1983); Ramkumar (1996)). Although the details are not clear, it at least should be clear that a determinant procedure exists for constructing P^+ from P and R. We jump now to a statement of the complexity of computing $P \oplus -R$ under a variety of convexity assumptions:

Theorem 8.4.2. *If P has n vertices, and R has a fixed (constant) number of vertices, then the Minkowski sum $P \oplus -R$ can be constructed in these time complexities:*

R	P	Size of Sum	Time Complexity
convex	convex	$O(n)$	$O(n)$
convex	nonconvex	$O(n)$	$O(n^2 \log n)$
nonconvex	nonconvex	$O(n^2)$	$O(n^2 \log n)$

These results were obtained by Guibas et al. (1983), Toussaint (1983*b*), Sharir (1987), and Kaul, O'Connor & Srinivasan (1991). Note that we view the size of the robot to be some fixed constant number of vertices and only report complexities with respect to n, the number of vertices of the obstacle polygon. Exercises 8.4.6[4]–[6] explore complexities as a function of the number of robot vertices.

8.4.4. Implementation of Minkowski Convolution

Although constructing the convolution tracing is only halfway to constructing its outer boundary, the Minkowski sum, the star diagram approach is elegant enough to warrant an implementation. Most of the effort required is in constructing the star diagram; with that in hand, tracing the convolution is easily accomplished by repeated addition of the corresponding edge vectors. Constructing the star diagram is essentially angular sorting of vectors, a task we already faced in Section 3.5.5 with Graham's convex hull algorithm. We will follow most of the implementation conventions we established for that algorithm. In particular, we will store the edge vectors in an array of structures and then sort with `qsort`. These structures and the top-level main procedure are shown in Code 8.1. The point structure `tPoint` differs from that used in Section 3.5.5 only in having a Boolean field `primary` which indicates whether the point belongs to the primary (nonconvex) polygon P (`TRUE`) or the secondary (convex) polygon R (`FALSE`). This field is set by `ReadPoints` (not shown), which also numbers the points in two separate series in the field `vnum`. Thus we have enough information to uncover a point's identity after `qsort` shuffles them all together in the array `P[]`. The number of vertices of the primary polygon is n, in the secondary is s, and in total $m = n + s$. The secondary polygon R is reflected as it is read in, so that only $-R$ is stored and used henceforth.

```
typedef struct tPointStructure tsPoint;
typedef tsPoint *tPoint;
struct tPointStructure {
    int         vnum;
    tPointi     v;
    bool        primary;
};

#define PMAX  1000                    /* Max # of points */
typedef tsPoint tPointArray[PMAX];
static tPointArray P;

int m;   /* Total number of points in both polygons */
int n;   /* Number of points in primary polygon */
int s;   /* Number of points in secondary polygon */

main()
{
    tPointi p0 = {0,0};
    int j0;                          /* index of start point */

    j0 = ReadPoints( p0 );
    Vectorize();
    qsort(
       &P[0],                        /* pointer to 1st elem */
       m,                            /* number of elems */
       sizeof( tsPoint ),            /* size of each elem */
       Compare                       /* -1,0,+1 compare function */
   );
   Convolve( j0, p0 );
}
```

Code 8.1 Data structures and main program for Minkowski convolution. Cf. Code 3.6.

Although we read in polygon vertices, all computations are performed on the edge vectors. So the first substantive step of the main procedure is to compute these edge vectors with a call to `Vectorize` (Code 8.2). There is no need to normalize to unit length as in Figure 8.6(a); in fact the vectors are more useful as is. We choose to store these vectors in the same array `P[]` that held the points. Aside from a little care needed to avoid using a newly overwritten vector instead of the original point (the reason for the temporary storage `last`), this computation is straightforward.

The call to `qsort` parallels its invocation in Code 3.6 and will not be discussed further. As with Graham's algorithm, coding the comparison function `Compare` is the delicate task. We cannot follow our previous model (Code 3.5) exactly, as there all vectors fell in the upper halfplane with respect to the origin. Here the edge vectors spray in all directions. We choose the following angular sorting convention. A vector aiming along the x axis toward the left (e.g., $(-1, 0)$) is at the minimum possible angle, $-\pi$

counterclockwise from the positive x axis (the usual convention for measuring angles). Angles increase with the counterclockwise angle from there, through the halfplane below the x axis, through the positive x axis, and finally through the halfplane above. This convention matches our earlier one but extends over the full 2π range. The left-of computation is still the key to deciding which vector precedes which. In the case of ties, we again choose the shorter vector as earlier in the sequence.

```
void Vectorize( void )
{
    int i;
    tPointi last;   /* Holds the last vector difference. */

    SubVec( P[0].v, P[n-1].v, last );
    for( i = 0; i < n-1; i++ )
       SubVec( P[i+1].v, P[i].v, P[i].v );
    P[n-1].v[X] = last[X];
    P[n-1].v[Y] = last[Y];

    SubVec( P[n].v, P[n+s-1].v, last );
    for( i = 0; i < s-1; i++ )
       SubVec( P[n+i+1].v, P[n+i].v, P[n+i].v );
    P[n+s-1].v[X] = last[X];
    P[n+s-1].v[Y] = last[Y];
}
```

Code 8.2 `Vectorize`. See Code 7.6 for `SubVec`.

Although the left-of computation is the heart of `Compare` (Code 8.3 and 8.4), most of the code is devoted to managing the special cases.[7] Vectors in the lower halfplane are before those in the upper halfplane; this can be decided by comparison of `v[Y]` for each vector. Vectors on the x axis require special handling. Otherwise we fall into the left-of calculation and tie resolution, which is handled the same as we used in the Chapter 3 version.

After the call to `qsort`, the edge vectors, with their identifying tags, sit in `P[]` in angularly sorted order. The remaining task is to cycle around the star diagram in the manner described earlier, adding up the edge vectors. One tricky issue is where to start – what should be the initial point p_0? This in turn depends on where we start processing the angles. We choose to start with the minimum angle, corresponding to a vector $(-1, 0)$, even though it is likely there is no such vector among the polygon edge vectors. (An alternative is to start at some specific edge vector of the primary polygon.) Although this is by no means self-evident, this choice implies we should start at the upper-rightmost point of the primary polygon P, shifted by the upper-rightmost point of

[7] I am not confident I found the optimal structure for this code.

the secondary polygon $-R$. We will not justify this claim. This start point p_0 is computed in `ReadPoints` and passed to the convolution procedure `Convolve`, which "moveto"s this point as its first action. The output of points is shown as Postscript commands in Code 8.5.

`Convolve` then cycles around the star diagram by incrementing an index `i` over `P[i]`, wrapping around as necessary with the statement `i = (i+1)%m`. This is the outer do-while-loop in the code. At each iteration it is seeking the next edge vector of the primary polygon, indexed by j. This index j is initialized to j_0, the index of the primary edge vector based on p_0. During its search for primary vector j, it outputs

```
int Compare( const void *tpi, const void *tpj )
{
  int a;              /* AreaSign result */
  int x, y;           /* projections in 1st quadrant */
  tPoint pi, pj;      /* Recasted points */
  tPointi Origin = {0,0};
  pi = (tPoint)tpi;
  pj = (tPoint)tpj;

  /* A vector in the open upper halfplane is after
     a vector in the closed lower halfplane. */
  if      ( ( pi->v[Y] > 0 ) && (pj->v[Y] <= 0 ) )
      return 1;
  else if ( ( pi->v[Y] <= 0 ) && (pj->v[Y] > 0 ) )
      return -1;

  /* A vector on the x axis and one in the lower
     halfplane are handled by the Left computation below. */

  /* Both vectors on the x axis require special handling. */
  else if (   ( pi->v[Y] == 0 ) && (pj->v[Y] == 0 ) ) {
     if       ( ( pi->v[X] < 0 ) && ( pj->v[X] > 0 ) )
        return -1;
     if       ( ( pi->v[X] > 0 ) && ( pj->v[X] < 0 ) )
        return 1;
     else if ( abs(pi->v[X]) < abs(pj->v[X]) )
        return -1;
     else if ( abs(pi->v[X]) > abs(pj->v[X]) )
        return 1;
     else
        return 0;
  }
  /* Continued ... */
```

Code 8.3 `Compare`, Part a. Cf. Code 3.5.

```
/* ... Continued */

/* Otherwise, both in open upper halfplane, or
   both in closed lower halfplane, but not both on x axis. */
else {
   a = AreaSign( Origin, pi->v, pj->v );
   if      (a > 0)
       return -1;
   else if (a < 0)
      return 1;
   else { /* Begin collinear */
      x = abs( pi->v[X] ) - abs( pj->v[X] );
      y = abs( pi->v[Y] ) - abs( pj->v[Y] );
      if     ( (x < 0) || (y < 0) )
         return -1;
      else if( (x > 0) || (y > 0) )
         return 1;
      else /* points are coincident */
         return 0;
   } /* End collinear */
  }
}
```

Code 8.4 `Compare`, Part b.

every secondary vector it encounters (this corresponds to the interspersed labels of $-R$ in Figure 8.6(a)). This is accomplished in the inner while-loop. When this inner loop ends, the next j has been found, and a primary edge vector is output and j incremented. The process repeats until j wraps around back to j_0 again.

An example of the code's output is shown in Figure 8.7. The corresponding star diagram is displayed in Figure 8.8. The angularly minimum vector is $(-20, 0)$, corresponding to the horizontal base of R, reflected in $-R$ to pointing leftward. The first step taken from p_0 (the uppermost point of P, circled in the figure) is this leftward horizontal "lineto." The index i then cycles through all $m = n + s = 22 + 8 = 30$ points of `P[]` ten times before returning back to j_0. The ten cycles are evident in the ten loops of the final figure. This figure makes manifest the further considerable effort needed to move from the convolution to the Minkowski sum, the outer boundary of Figure 8.7.

8.4.5. Conceptual Motion Planning Algorithm

We now return to planning a motion for a convex polygon R. Again we will only provide a rough sketch of an algorithm. Let the obstacles be $P_1, P_2, \ldots, P_m$ with a total of n vertices. The algorithm consists of four steps:

1. Grow all obstacles: $P_i^+ = P_i \oplus -R$.
2. Form their union $P^+ = \bigcup_i P_i^+$.

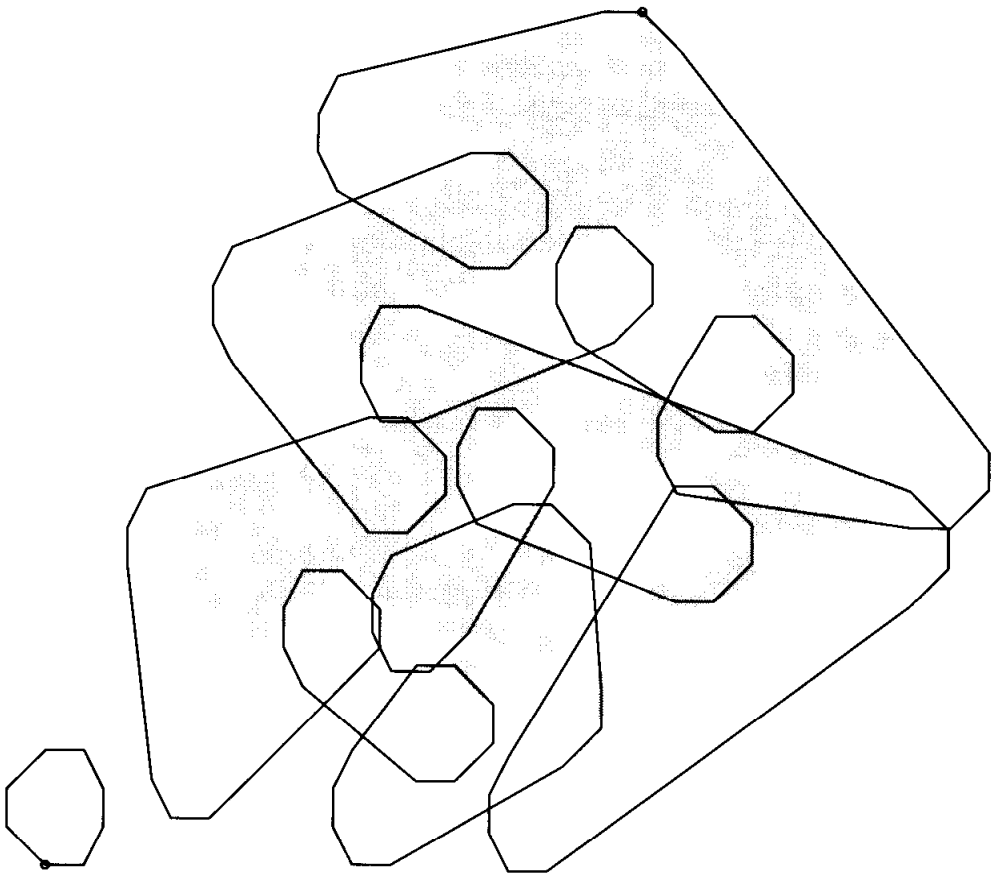


FIGURE 8.7 The convolution of a 22-vertex nonconvex polygon P (shaded) with a con octagon R, shown unreflected at the lower left with its reference point circled.

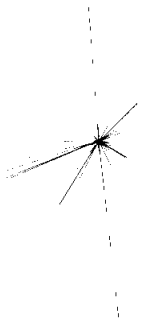


FIGURE 8.8 The edge vectors for P (longer) and $-R$ (shorter) from Figure 8.7.

```
void  Convolve( int j0, tPointi p )
{
  int i;   /* Index into sorted edge vectors P */
  int j;   /* Primary polygon index */

  MoveTo_i( p );

  i = 0;    /* Start at angle -pi, rightward vector. */
  j = j0;   /* Start searching for j0. */
  do {

      /* Advance around secondary edges until next j reached. */
      while ( !(P[i].primary && P[i].vnum == j) ) {
          if ( !P[i].primary ) {
              AddVec( p, P[i].v, p );
              LineTo_i( p );
          }
          i = (i+1)%m;
      }

      /* Advance one primary edge. */
      AddVec( p, P[i].v, p );
      LineTo_i( p );
      j = (j+1)%n;

  } while ( j != j0);
}
```

Code 8.5 `Convolve`. Subsidiary Postscript output commands are not shown.

3. Find the component containing s and t.
4. Find a path between s and t in that component.

Figure 8.9 illustrates the process with a quadrilateral robot R and eight obstacles (dark shading).

Note that, in this example, the free space has three connected components, containing the points a, b, and c. From the initial robot position shown, a is reachable, but neither b nor c is. Thus deciding whether or not a goal position t is reachable from s is reduced to determining whether s are t are in the same connected component of the free space. And planning a path for the robot reduces to finding a path for the reference point within this component.

All four of the above steps present interesting algorithmic issues, none of which we will explore here. In light of the intricacy of the whole scheme, it is a testimony to the many researchers who worked on aspects of this problem that Kedem & Sharir (1990) were able to achieve the remarkable complexity of $O(n \log n)$:

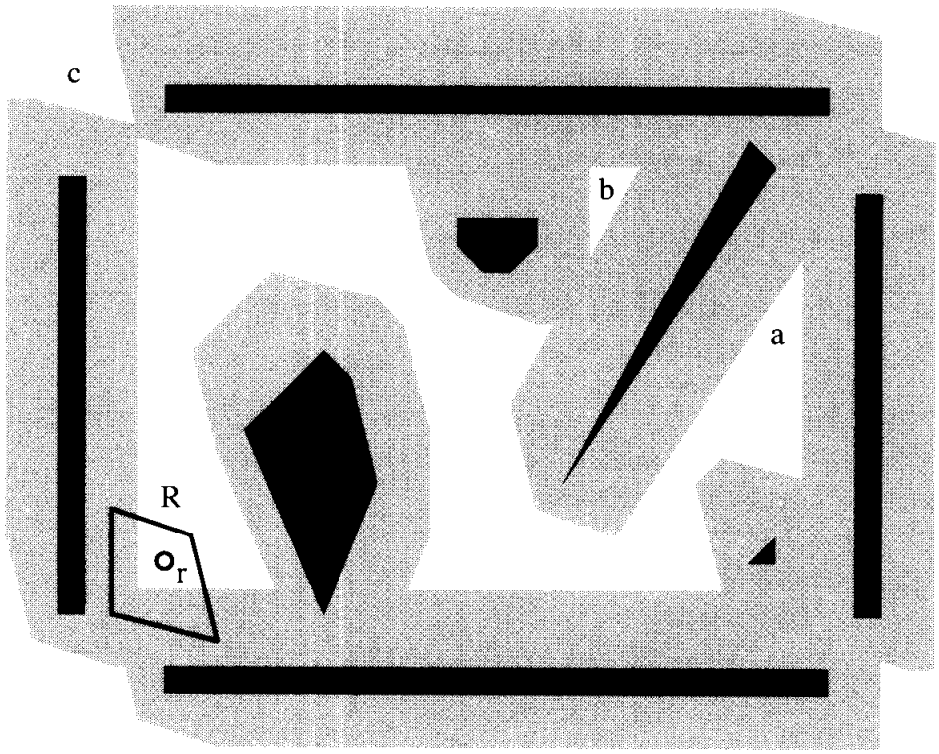


FIGURE 8.9 Robot R with reference point and origin r shown in lower left corner. Dark polygons are obstacles. The grown obstacles, each the Minkowski sum with $-R$, are shown lightly shaded.

Theorem 8.4.3. *Finding a series of translations that move a convex polygonal robot between given start and termination positions, avoiding polygonal obstacles with a total of n vertices, can be accomplished in $O(n \log n)$ time. More precisely, if the robot has k vertices, the complexity is $O(kn \log(kn))$.*

Even more remarkably, roughly the same complexity has been established for moving a convex polytope robot in 3-space, avoiding polyhedral obstacles (Aronov & Sharir 1997).

8.4.6. Exercises

1. *Sum of square and triangle* [easy]. What is the largest number of edges the Minkowski sum of a square and a triangle can have?
2. *Convolution cycles.* Develop a way to predict the number of times `Convolve` (Code 8.5) will cycle around the star diagram, based on the structure of P. Test out your theory on Figures 8.6(b) and 8.7.
3. *Star diagram: nonconvex.* Explore the convolution for nonconvex R. Does the star diagram approach still find τ?
4. *Convex–convex* [easy]. Prove that the Minkowski sum of two convex polygons R and P of k and n vertices, respectively, can have $\Omega(k + n)$ vertices.
5. *Convex–nonconvex.* Prove that the Minkowski sum of a convex polygon R and a (perhaps nonconvex) polygon P of k and n vertices, respectively, can have $\Omega(kn)$ vertices.

6. *Nonconvex–nonconvex.* Prove that the Minkowski sum of two (perhaps nonconvex) polygons R and P of k and n vertices, respectively, can have $\Omega(k^2n^2)$ vertices.
7. *Union of convex regions* [difficult]. What is the largest possible number of vertices of the union of m convex polygons with a total of n vertices? Express your answer as a function of n and m.
   a. First guess an upper bound, supported with examples.
   b. Prove your bound (Kedem, Livne, Pach & Sharir 1986).

8.5. MOVING A LADDER

The most complicated rigid-robot motion planning problem we consider is moving a line segment robot among polygonal obstacles. The segment is often called a ladder (or sometimes a rod). What makes this complicated is that, in contrast to the previous sections, here we allow rotation.

Rotation gives the ladder three "degrees of freedom" in its motion: two of translation (e.g., horizontal and vertical) and one of rotation. This means that it is not possible to transform a general instance of this problem to that of a point moving in two dimensions (as we did in the previous two sections), since such a point has two degrees of freedom. However, it is possible to reduce any ladder problem to a motion planning problem of a point robot moving in a three-dimensional obstacle space. The beautiful idea that permits this transformation was first annunciated by Lozano-Pérez & Wesley (1979). We will explain it with the simple example shown in Figure 8.10(a). The ladder L is initially horizontal; we would like it to follow the path indicated in (a). But it can only do so by rotating. Figure 8.10(b) grows the obstacles by the Minkowski sum of the objects with the horizontal ladder, as described in the previous section. That C grows to overlap A and B clearly shows that L cannot pass through the channel without rotating. Part (c) of the figure shows the grown obstacles when the ladder is rotated by 30°, and in part (d) the ladder is rotated 60°. In (d) the vertical channel between A and C has opened up, while the horizontal channel between A and B has closed. These figures show that the ladder can follow a path indicated in (a), by moving between A and B with little or no rotation, and then by rotating counterclockwise 60° or more before moving vertically.

Now imagine stacking all the grown obstacles for all possible rotation values θ, all in parallel planes, as depicted in Figure 8.11. Each point (x, y, θ) in this space represents a position of the reference point of the ladder; the plane on which it lies represents the rotation θ. Thus we have achieved what was claimed above: We have transformed the problem of moving a ladder in two dimensions to an equivalent problem of moving a point in three dimensions. This three-dimensional space is known as the *configuration space* for the robot/ladder.

Although perhaps not evident from Figure 8.11, the obstacles are not polyhedral. In each θ plane they are polygonal, but they twist along the θ direction, producing complex shapes. The space in which the reference point is free to move is called the *free space*. Were you standing at the start position s in this space, you would see a cavernous chamber with twisted walls. There is a path for the ladder iff t is in the same connected component of the free space as is s.

It should be evident that there is nothing special about a ladder: We could as well obtain a configuration space for an arbitrary polygonal robot among polygonal obstacles,

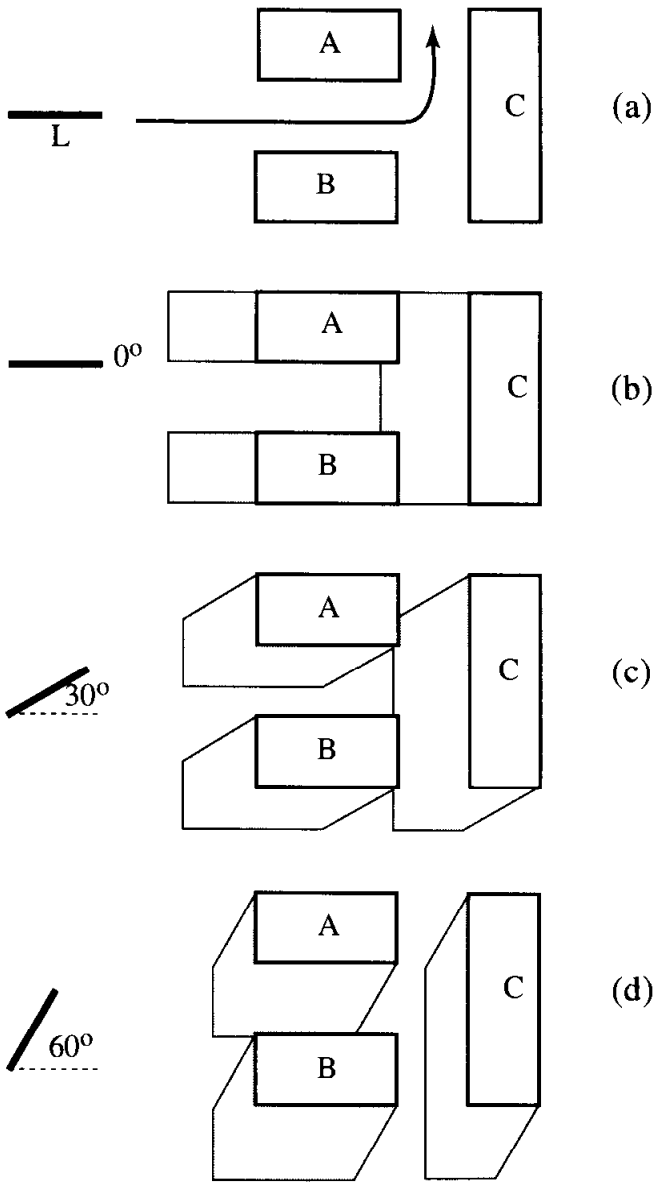


FIGURE 8.10 Rotating the ladder permits it to pass between A and B and move up, following the path indicated in (a). (b)–(d) show the grown obstacles for $\theta = 0°, 30°, 60°$, with the ladder reference its left endpoint

by the same technique. Indeed, the basic idea extends to three-dimensional robots and obstacles, and even to jointed robots.

Constructing a representation for a configuration space, and then finding a path inside of it, is a challenging task. But its importance has fostered intense research, and these configuration spaces are indeed constructed, sometimes in as high as six-dimensional space, and then navigated to construct paths for the corresponding robot. Lozano-Pérez (1983) contains a general discussion, and Brost (1991) contains some stunning images of various configuration spaces.

We will now sketch two different methods for solving motion planning problems by finding a free path through configuration space. Although both methods are quite general, we will only discuss the case of moving a ladder.

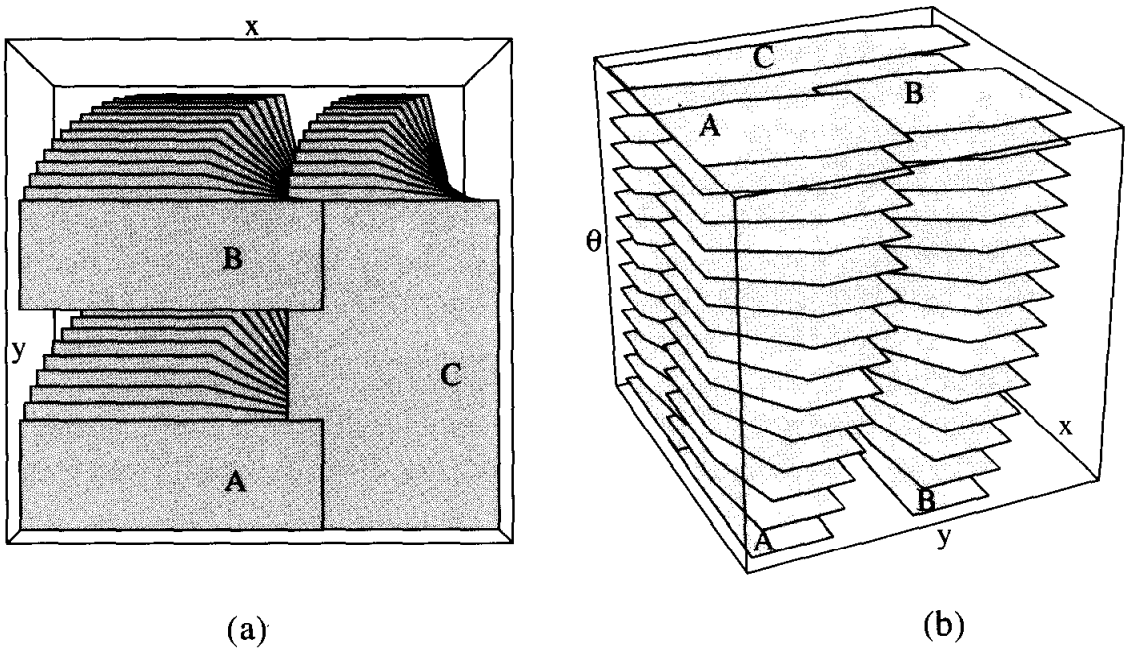


FIGURE 8.11 (a) View from underneath the stacks of grown obstacles, with the bottom corresponding to $\theta = 0$, Figure 8.10(b). (b) Front view of same stacks; θ varies from 0° at the bottom of the box to 75° on top.

8.5.1. Cell Decomposition

The first general method invented for solving motion planning problems was the *cell decomposition* method, developed in a remarkable series of five papers by Schwartz and Sharir, the "piano movers" papers.[8] These papers established that a wide variety of motion planning problems can be solved with polynomial-time algorithms,[9] with the exact complexity depending on the details of the problem. Here I will outline roughly their technique applied to the ladder problem, as detailed by Leven & Sharir (1987).

Definition of a Cell

The essence of the cell decomposition approach is to partition the unruly configuration space into a finite number of well-behaved "cells," and to determine a path in the space by finding a path between cells.

Consider the environment shown in Figure 8.12. It consists of two triangle obstacles and a bounding (open) polygonal wall. The ladder L is horizontal, with its reference point to the left, at the arrowhead. For the moment we only look at a single orientation of L, horizontal as shown. A cell is a connected region within the free space of the appropriate configuration space. Because we have fixed the ladder's orientation, the configuration space is just the plane, and the free space is what remains after growing the obstacles by the Minkowski sum with the ladder. To get a precise definition of a cell, we assign

[8] Schwartz & Sharir (1983*a*), Schwartz & Sharir (1983*b*), Schwartz & Sharir (1983*c*), Sharir & Ariel-Sheffi (1984), Schwartz & Sharir (1984). All five are collected in Hopcroft, Schwartz & Sharir (1987).

[9] A *polynomial-time algorithm* is one whose time complexity is $O(n^k)$ for some constant k.

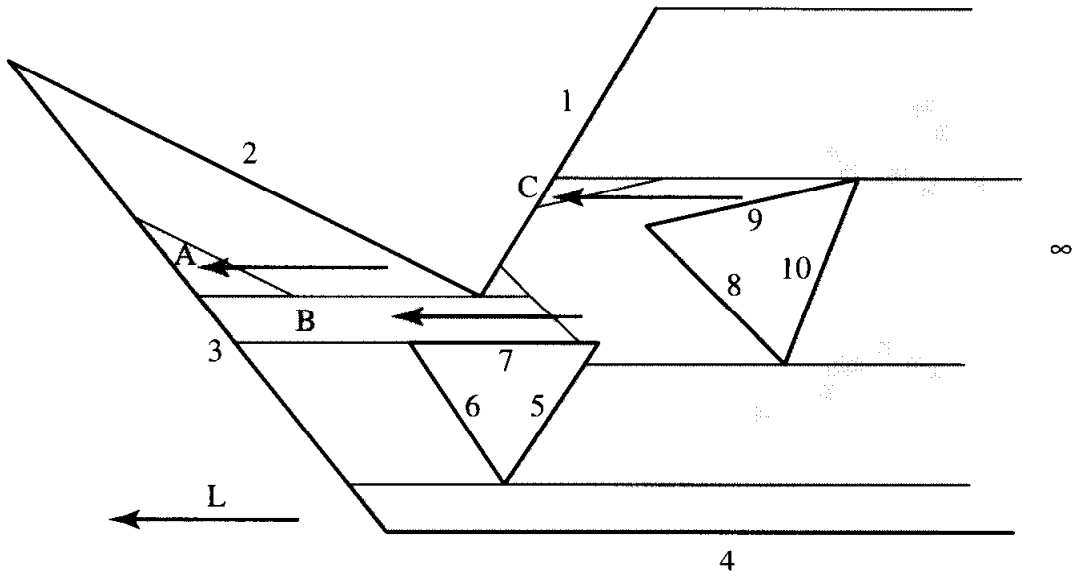


FIGURE 8.12 Cell decomposition, with ladder horizontal. Integer labels index edges.

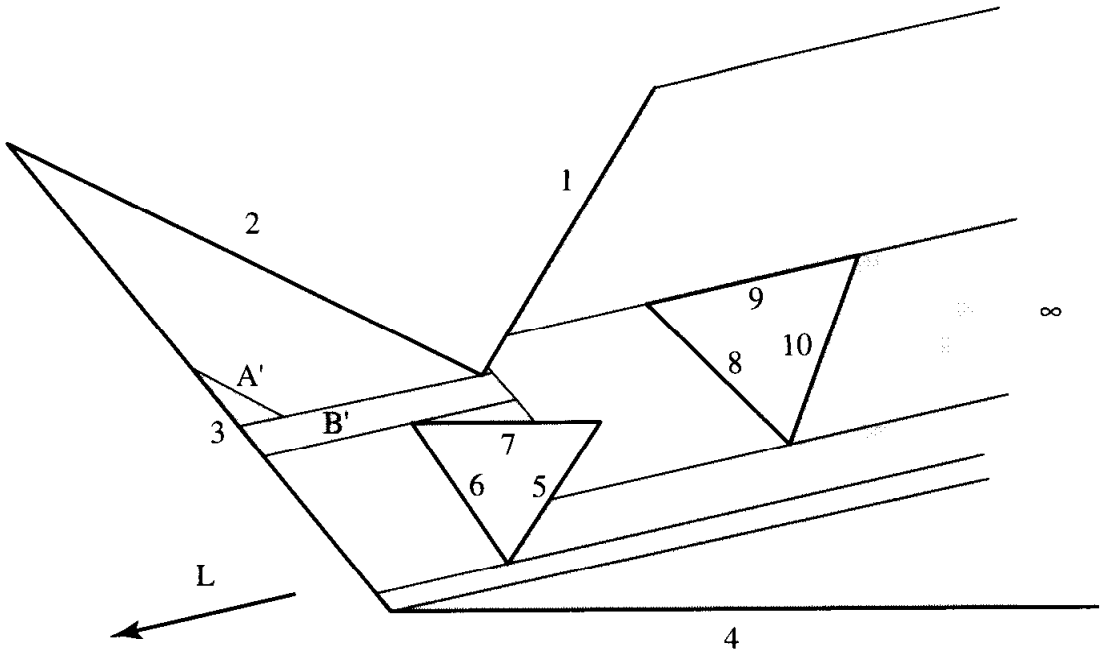


FIGURE 8.13 Cell decomposition, with ladder tilted.

labels to every obstacle edge,[10] as is done in the figure; we use the label ∞ to represent a "surrounding" edge infinitely far to the right. Suppose we place the ladder's reference point at a point x not on the same horizontal as any vertex. Then moving L horizontally forward (leftwards) will cause it to bump into some (one) obstacle edge eventually, as will moving it backward (rightwards). Label the point x with this pair of edge labels. A *cell* is a collection of free points all with the same forward/backward label pairs.

In Figure 8.12, cell A has labels (3, 2); cell B has labels (3, 8); cell C has labels (1, 9); and no cell has labels (3, 6), because there are no free points between those two edges.

[10] Also, labels should be assigned to the vertices, but we will ignore this minor complication here. See Leven & Sharir (1987).

Connectivity Graph G_θ

In the cell decomposition approach, the cell structure is represented with a graph, the *connectivity graph* G_θ. The subscript indicates that this graph captures the structure for a particular orientation of the ladder θ. The nodes of G_θ are the cells, and two nodes are connected by an arc if the cells touch, or more precisely, if their boundaries share a nonzero-length segment. So G_θ is something like the duals considered in Chapter 1 (Section 1.2.3) and Chapter 4 (Section 4.4). G_0 corresponding to the cells in Figure 8.12 is shown in Figure 8.14(a). Note that G_0 is disconnected: There is no path in G_0 between cells A and C.

The importance of this graph is that motion planning within a cell is trivial, so that a path in the graph can be easily converted into a path for the ladder. Moreover, the ladder can only move from one cell to another if there exists a path in the graph between these cells.

Critical Orientations

Now we incorporate rotation in a manner similar to the plane-stacking idea used previously. If we rotate the ladder slightly, the connectivity graph for the obstacles in Figure 8.12 normally will not change: All the cells will change shape, but they will remain and will maintain their cell adjacencies. But when the rotation exceeds some *critical orientation* θ^*, the combinatorial structure of G_{θ^*} will be different from that of G_0.

The $\theta = 0$ orientation shown in Figure 8.12 is critical, because there are obstacle edges parallel to L. Thus a slight rotation of L counterclockwise about its reference point will create a $(7, 8)$ cell and a $(4, \infty)$ cell. Further rotation to the orientation of obstacle edge e_9 causes cell C to disappear, as no points any longer have label $(1, 9)$. The cell decomposition at this orientation is shown in Figure 8.13; its corresponding connectivity graph is shown in Figure 8.14(b).

As the reader might guess, critical orientations all involve the alignment of the ladder with either edges of obstacles or two obstacle vertices. Thus there are at most $O(n^2)$ critical orientations.

Connectivity Graph G

Now the idea is to form one grand connectivity graph G that incorporates the information in all the G_θ graphs. We extend the definition of a cell to represent regions of the three-dimensional configuration space, all of whose points have the same forward/backward label pairs. This amounts to stacking the cells for fixed orientations on top of one another in the θ direction. Thus the points in cell A in Figure 8.12 are in the same three-dimensional cell as the points in cell A' in Figure 8.13. Each distinct three-dimensional cell is a node of G, and again two nodes are connected by an arc if their cells touch, which now means that they share a nonzero-area boundary section.

The graph G may be constructed by building G_0, initializing $G \leftarrow G_0$, and then moving through all critical orientations in sorted order, modifying G_θ along the way, and incorporating the changes into G. We will not present any details (see Leven & Sharir (1987)), but the reader should see that construction of G is possible.

Again motion planning within a single cell represented by a node of G is not difficult, and moving between touching cells is also not difficult. For example, one could move

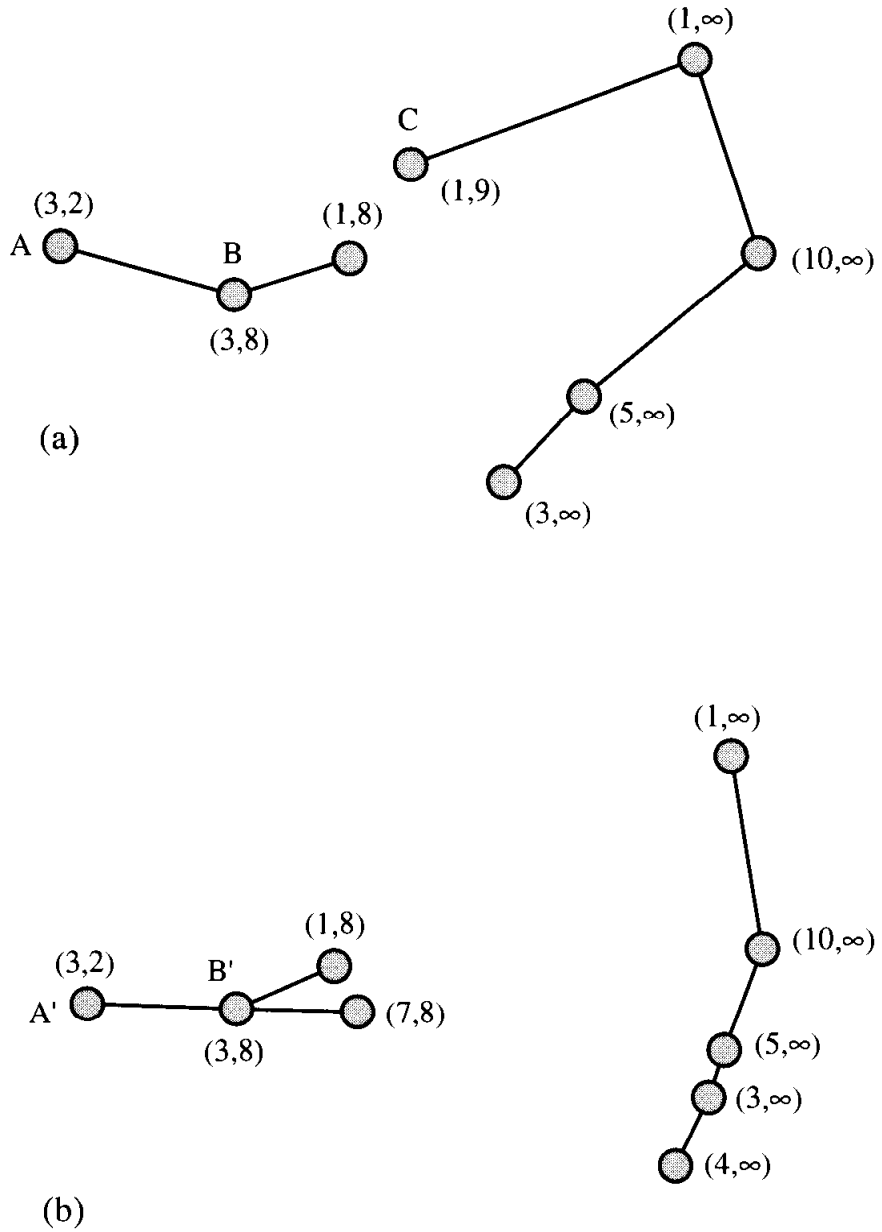


FIGURE 8.14 Connectivity graphs for cell decompositions: (a) G_0, Figure 8.12; (b) Figure 8.13.

from the interior of a cell to its boundary, and then move along the boundary to the portion shared with an adjacent cell. So the problem of motion planning is reduced to a graph problem: finding a path between the node corresponding to the cell containing s to the node corresponding to the cell containing t. If there is no such path in G, then there is no path for the ladder, and if there is a path in G, it can be used as a guide for planning the motion of the ladder.

8.5.2. Retraction

A rather different but no less general technique for solving motion planning problems is the *retraction method*, due to Ó'Dúnlaing & Yap (1985). Here we sketch the idea applied to moving a ladder; details are in Ó'Dúnlaing, Sharir & Yap (1986) and Ó'Dúnlaing, Sharir & Yap (1987).

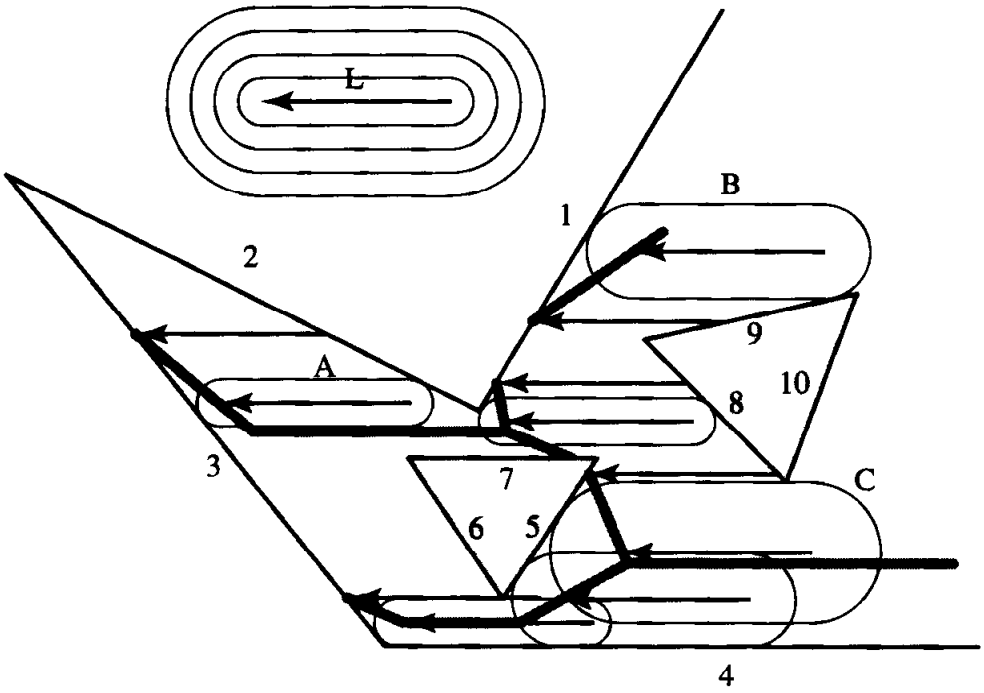


FIGURE 8.15 Voronoi diagram (partial) for a ladder. A, B, and C mark particular positions of the ladder.

Voronoi Diagram

The essence of retraction is to construct a "Voronoi diagram" for the ladder, and then "retract" from s and t to this diagram, and perform path planning within the "network" of the diagram. First we explain what a Voronoi diagram means in this context.

Recall from Chapter 5 (Section 5.2) that points on the edges of a Voronoi diagram are equidistant from at least two sites (vertices of the diagram are equidistant from at least three sites). For a fixed orientation of the ladder L, we define the Voronoi diagram of the obstacles with respect to L to be the set of free points x such that, when the ladder's reference point is placed at x, L is equidistant from at least two obstacle points.

First we must define distance to L. The distance of a point p to L is the minimum length of any line segment from p to a point on L. Just as the points at distance r from a point form a circle, the points a distance r from L form a "racetrack": an oval formed by two edges parallel to L connected by half circles. Nested racetracks are shown in Figure 8.15 surrounding L. The same figure shows the Voronoi diagram as shaded lines, together with several sample ladder positions. For example, in position A, L is equidistant from e_3 and e_2; in position B it is equidistant from e_1 and e_9; in position C it is equidistant from e_5 and the vertex common to e_8 and e_{10}.[11] We can see that the diagram is disconnected: there is no path from A to B.

This diagram has the pleasant property that moving L so that its reference point stays on diagram edges prudently places L as far from nearby obstacles as possible, for all these positions are equidistant from two or more obstacle points. This is a very useful feature for a robot who is trying to avoid collisions with obstacles.

[11] Note that the "equidistance edge labels" are not necessarily the same as the cell labels from the previous section. Thus position B in Figure 8.15 has equidistance labels $(1, 9)$ but cell labels $(1, \infty)$.

From here the strategy should sound familiar. We imagine stacking Voronoi diagrams for each value of θ orthogonal to the θ axis, thereby forming a Voronoi diagram for all of configuration space. One can see that the diagram consists of twisted "sheets," formed by the stacking of edges, and "ribs" where two sheets meet, formed by the stacking of vertices.

Retraction

Again the problem will be reduced to a graph search, but in a rather different manner than in the cell decomposition approach. The ribs between Voronoi sheets form a network of curves on the Voronoi diagram in configuration space. These curves form a graph N in the natural manner: Interpret each curve as an arc, and the point where two or more curves meet as a node.

The final step involves two *retractions* of the start and termination points s and t: The first retraction maps these points to the Voronoi surface, and the second maps from there to the network. Let s' and t' be these retracted points on the network. Then there is a path for the ladder from s to t iff there is a path in the network from s' to t', which can be determined by searching the graph N. The resulting "high-clearance" path is appropriate in many contexts, for example, controlling the motion of a cutting tool.

8.5.3. Complexity

So far we have said little about the complexity of planning motion for a ladder. Rather than attempt to analyze the complexity of the above incompletely specified algorithms, we will sketch a short history of the complexities obtained for moving a ladder in two and in three dimensions, which illustrates if nothing else the doggedness of the community's pursuit.

The two-dimensional problem, which we have been discussing in this section, has received considerable attention, serving as something of a test bed for algorithmic ideas. The first solution was obtained in the first piano movers paper, and subsequently a variety of improvements were made (not all of which are reflected in the asymptotic time complexity). I showed that there are configurations of obstacles that force any solution path to have a quadratic number of distinct "moves," establishing a lower bound on any algorithm that prints them out. This lower bound was finally reached in 1990. The time complexities are shown in the table below.

Authors	Time Complexity
(Schwartz & Sharir 1983*a*)	$O(n^5)$
(Ó'Dúnlaing et al. 1987)	$O(n^2 \log n \log^* n)$
(Leven & Sharir 1987)	$O(n^2 \log n)$
(Sifrony & Sharir 1987)	$O(n^2 \log n)$
(Vegter 1990)	$O(n^2)$
(O'Rourke 1985*b*)	$\Omega(n^2)$

We have not discussed the problem of moving a ladder in three dimensions among polyhedral obstacles. It is of course much more complicated, leading to a five-dimensional

configuration space. Again the first algorithm was achieved by cell decomposition, with a formidable time complexity. The fastest algorithm to date employs Canny's "roadmap" algorithm, which is another general technique for solving motion planning problems, a generalization of retraction. In this instance, there remains a gap with the best lower bound.

Authors	Time Complexity
(Schwartz & Sharir 1984)	$O(n^{11})$
(Ke & O'Rourke 1987)	$O(n^6 \log n)$
(Canny 1987)	$O(n^5 \log n)$
(Ke & O'Rourke 1988)	$\Omega(n^4)$

The strongest general result on motion planning is due to Canny (1987):

Theorem 8.5.1. *Any motion planning problem in which the robot has d degrees of motion freedom can be solved in $O(n^d \log n)$ time.*

Although this is the best general result, specific problem instances have asymptotically faster algorithms. For example, a ladder in two dimensions has three degrees of freedom, and Canny's result implies there is an $O(n^3 \log n)$ algorithm; but we have already seen that this has been improved by a factor of $n \log n$.

8.5.4. Exercises

1. *Shape of cells.* For moving a ladder in two dimensions via the cell decomposition approach (Section 8.5.1), prove or disprove:
   a. Every cell represented by a node of G_θ is convex.
   b. Every cell represented by a node of G is convex.
2. *Ladder Voronoi diagrams.* Prove or disprove that the ladder Voronoi diagram for a fixed orientation of the ladder consists of straight segments only.
3. $\Omega(n^2)$ *connected components.* Construct an example that establishes that the connectivity graph G for moving a ladder can have $\Omega(n^2)$ distinct components.
4. *Worst chair through a doorway* [open]. Let a *doorway* be a vertical line, say coinciding with the y axis, with the open segment from $y = 0$ to $y = 1$ removed. A polygon is said to *fit through the doorway* if there is a continuous motion that moves it from the left of the doorway to the right without the interior of the polygon ever intersecting the rays above and below the doorway: from $y = 1$ upwards and from $y = 0$ downwards. In this problem you are to concoct a class of polygons of n vertices that all fit through the doorway but require a large number of distinct moves to pass through the doorway. For this to make sense, we need to define what constitutes a "move."

   Fix a reference point r in the polygon. Then any continuous motion of the polygon can be viewed as a continuous translation of and rotation about r, continuous with respect to a time parameter t. Thus a motion of the polygon can be represented by three functions, $x(t)$, $y(t)$, and $\theta(t)$, specifying the translation of r and the rotation about r.

   Imagine plotting these functions with respect to time; see Figure 8.16. We consider a *move* to encompass a maximal interval of time during which all three functions are monotonic, either increasing or decreasing. The task is to find polygons that require the most moves under this

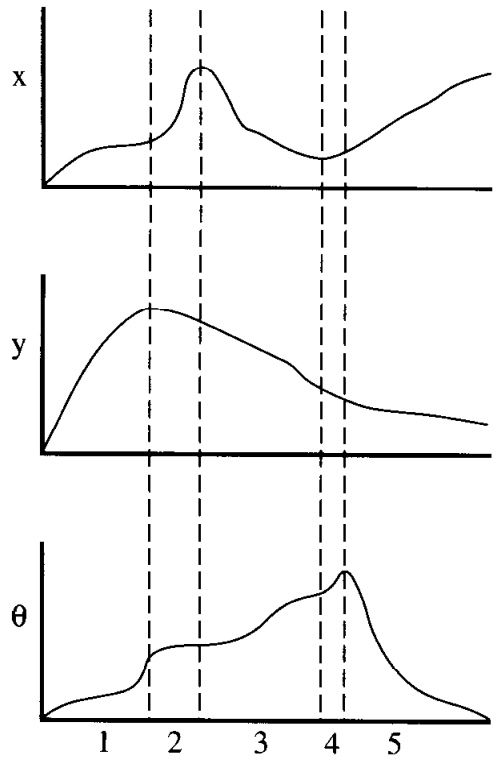


FIGURE 8.16 Motion functions of time. This motion counts as five "moves."

definition, with respect to the number of polygon vertices. These are the shapes that are the most difficult in some sense to move through a doorway.

The "worst" polygons I know of require $n/2$ moves (Jones & O'Rourke 1990). But the only upper bound is $O(n^2)$ (Yap 1987). Either find a generic example that forces more than a linear number of moves or prove a smaller upper bound. Even improving on the fraction $\frac{1}{2}$ would be interesting.

5. *Shortest ladder path.* There are many different possible measures of the length of a ladder path, the movement of a segment (with rotation). Here we explore three. Let the segment/ladder be unit length, and let there be no obstacles. For the segment in some initial position, say $[(0, 0), (1, 0)]$, the task is to find the length of a shortest path to turn the segment around to $[(1, 0), (0, 0)]$, returning it to its original position but oriented backwards. Find this length under three measures:
   a. The length of the ladder path is the length of the path the midpoint of the segment follows during the motion.
   b. The length of the ladder path is the length of the path followed by one endpoint of the segment.
   c. The length of the ladder path is measured by the sum of the lengths of the paths followed by the two endpoints of the segment (Icking, Rote, Welzl & Yap 1993).

8.6. ROBOT ARM MOTION

Problem Definition

A subfield of motion planning of considerable practical interest is planning the motion of an anchored "robot arm." In this section we will examine a particularly simple instance, the planar multilink arm. This is a chain of fixed-length segments, the *links*

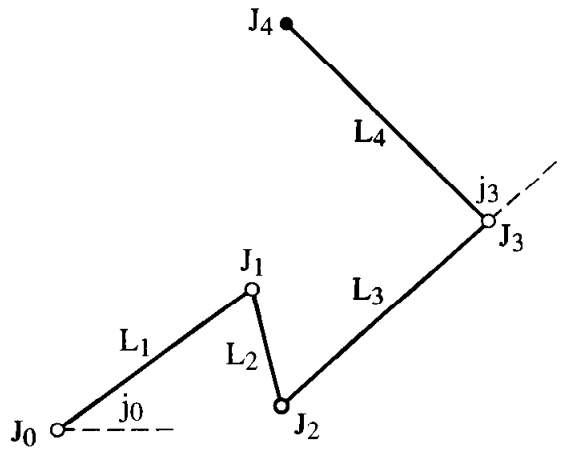


FIGURE 8.17 Notation for a multilink arm.

L_i, $i = 1, \ldots, n$, connected at *joints* J_i, $i = 0, \ldots, n$. Joint J_0 is anchored to the origin, sometimes called the "shoulder" of the arm. J_i for $0 < i < n$ is the joint between L_i and L_{i+1}. J_n is the tip of L_n, sometimes called the "hand." See Figure 8.17.

We will need notation for various quantities associated with a given robot arm. We let ℓ_i be the length of link L_i, and j_i be the angle at joint J_i, measured counterclockwise between L_i and L_{i+1}, treated as vectors from J_{i-1} to J_i and from J_i to J_{i+1} respectively. The angle j_0 is measured from the positive x axis; j_n is undefined. An *arm* A is specified by its list of link lengths: $(\ell_1, \ldots, \ell_n)$.

We will explore a particularly simple version of the general problem, simple in two respects:

1. We place no restriction on the joint angles. In particular, the arm may self-intersect.
2. We consider the plane to be empty: There are no obstacles.

With these restrictions, we study the *reachability* problem: Given n link lengths ℓ_i defining an arm A, and a point in the plane p, determine if A can reach p, and if so, find a set of joint angles that establish $J_n = p$. It will turn out that deciding whether p is reachable (the decision question) is easy, but computing joint angles that realize a solution is more challenging.

History

The algorithmic issues for robot arm motion were first explored in a paper by Hopcroft, Joseph & Whitesides (1985). They established that the problem with no obstacles (the one studied in this section) is easy, the problem with arbitrary obstacles is hard (the technical term is "NP-hard"), but the problem with the arm confined inside a circle is tractable (polynomial). Since then a number of other researchers have improved on their circle-confined algorithm or obtained similar algorithms for different obstacle environments.[12] See Whitesides (1991) for a general discussion.

[12] See Kantabutra & Kosaraju (1986), Kantabutra (1992), Kutcher (1992), and Suzuki & Yamashita (1996).

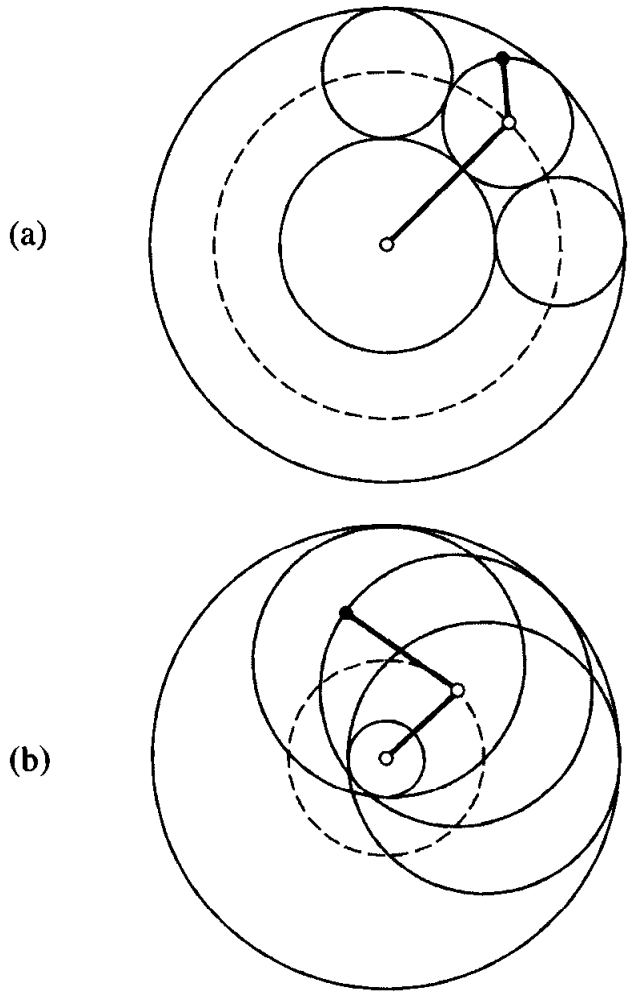


FIGURE 8.18 Reachable region for a 2-link arm: (a) $\ell_1 > \ell_2$; (b) $\ell_1 < \ell_2$.

8.6.1. Reachability: Decision

What is the set of points reachable by a multilink arm? The answer is surprisingly easy: It is always an origin-centered annulus, the closed set of points between two concentric circles. We establish this in Lemma 8.6.1 below and then proceed to determine in Theorem 8.6.3 the inner and outer radii r_i and r_o of the annulus as a function of the link lengths.

Reachability Region

The region reachable by a 1-link arm is a circle centered on the origin, which is an annulus with equal inner and outer radii.

Let $A = (\ell_1, \ell_2)$ be a 2-link arm. If $\ell_1 \geq \ell_2$, then the reachability region is clearly an annulus with outer radius $r_o = \ell_1 + \ell_2$ and inner radius $r_i = \ell_1 - \ell_2$. See Figure 8.18(a). If $\ell_1 = \ell_2$, $r_i = 0$ and the annulus is a disk of radius r_o.

When $\ell_1 < \ell_2$, the situation is perhaps not so clear. But, as Figure 8.18(b) shows, the result is again an annulus with $r_o = \ell_1 + \ell_2$, but with $r_i = \ell_2 - \ell_1$ (or, as it will sometimes be convenient to write it, $r_i = |\ell_1 - \ell_2|$).

It is revealing to view the 2-link reachability region as the Minkowski sum of two circles (see Section 8.3.1): On each point on the circle C_1 of radius ℓ_1, center a circle of radius ℓ_2. Thus the sum of two origin-centered circles is an origin-centered annulus.

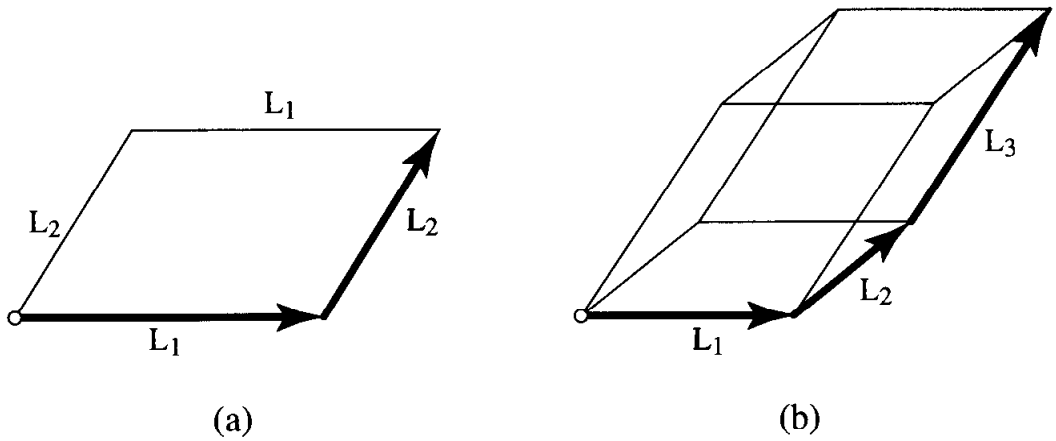


FIGURE 8.19 Parallelograms for (a) two links and (b) three links show that the order of the links does not affect reachability.

Moreover, it should now be clear that the sum of an annulus and a circle, both origin-centered, is again an origin-centered annulus. Thus we have:

Lemma 8.6.1. *The reachability region for an n-link arm is an annulus centered on the origin (shoulder).*

Annulus Radii

Although it is clear that the outer radius of the annulus in Lemma 8.6.1 is obtained by stretching all the links out straight, $r_o = \sum_{i=1}^{n} \ell_i$, the inner radius is not so obvious. We now turn to computing r_i.

Whether or not $r_i > 0$ depends on the relation between the length of the longest link and the lengths of the other links. In particular, $r_i > 0$ iff the longest link is longer than all the other link lengths combined. This is perhaps easiest to see if the longest link is the first link in the arm. We will now show how to view matters this way without loss of generality.

Lemma 8.6.2. *The region of reachability for an arm is independent of the order in which the links are arranged.*

Proof. This follows from the commutativity of vector addition. Consider, for example, a configuration of a particular 2-link arm, as shown in Figure 8.19(a). Following the other two sides of the parallelogram clearly reaches the same endpoint.[13] The same holds true for a 3-link arm, as shown in (b) of the figure, and indeed for an n-link arm. □

We therefore concentrate, without loss of generality, on arms whose first link L_1 is longest. For these arms, it should be clear from Figure 8.20 that $r_i = \ell_1 - \sum_{i=2}^{n} \ell_i$, as long as this sum is positive, and $r_i = 0$ otherwise. We can now summarize our findings, first stated by Hopcroft et al. (1985).[14]

[13]This proof idea is from Dettmers, Doraiswamy, Gorini & Toy (1992).

[14]They offer no proof in their paper. They remark, incidentally, that the theorem clearly holds in three dimensions as well.

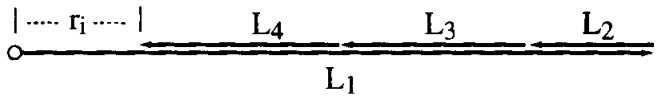


FIGURE 8.20 $r_i = \ell_1 - (\ell_2 + \ell_3 + \ell_4)$.

Theorem 8.6.3. *The reachability region for an n-link arm is an origin-centered annulus with outer radius* $r_o = \sum_{i=1}^{n} \ell_i$ *and inner radius* $r_i = 0$ *if the longest link length* ℓ_M *is less than or equal to half the total length of the links, and* $r_i = \ell_M - \sum_{i \neq M} \ell_i$ *otherwise.*

It is an immediate corollary of this theorem that we can decide reachability in $O(n)$ time: Find ℓ_M and compute r_o and r_i; then p is reachable iff $r_i \leq |p| \leq r_o$. The theorem, however, gives no hint how to find a configuration that reaches a given point. We now turn to this question.

8.6.2. Reachability: Construction

At first blush, it is not evident how to find a configuration for an n-link arm to reach a point within its reachability region. In some sense there are too many solutions, and methods that attempt to explore methodically all potential solutions can become mired in exponentially many possibilities. For example, trying to delimit the angle ranges at each joint within which solutions lie quickly fractures into an exponential number of ranges.

Fortunately, much more efficient algorithms can be achieved by exploiting the weak requirement that just some one solution is desired. We examine the 2- and 3-link problems before jumping to the n-link case.

2-Link Reachability

Determining the shoulder angle j_0 for a 1-link arm to reach a point on its circle is trivial. Solving a 2-link problem is not much more difficult. Let p be the point to be reached. Simply intersect the circle C_1 of radius ℓ_1 centered on the origin (J_0) with the circle C_2 of radius ℓ_2 centered on p. In general there will be two solutions, but there could be zero, one, two, or an infinite number, depending on how the circles intersect, as shown in Figure 8.21. We will discuss implementing this intersection computation in Section 8.6.3

3-Link Reachability

Our general approach will be to reduce multilink problems to 2-link problems. Let $A_3 = (\ell_1, \ell_2, \ell_3)$. We know from Lemma 8.6.1 that the reachability region for $A_2 = (\ell_1, \ell_2)$ is an annulus; call it R. Note that all points of the boundary ∂R of R represent configurations of A_2 that are extreme in that either the arms are aligned or antialigned: $j_1 = 0$ or $j_1 = \pi$. In these positions, A_2 acts like a single link of length $\ell_1 + \ell_2$ or $|\ell_1 - \ell_2|$ respectively.

Now examine how the circle C of radius ℓ_3, centered on $p = J_3$, intersects R. Our goal is to reduce 3-link solutions to alignments of two links, so that they may be viewed as 2-link solutions. We distinguish two cases, depending on whether or not $\partial R \cap C = \emptyset$.

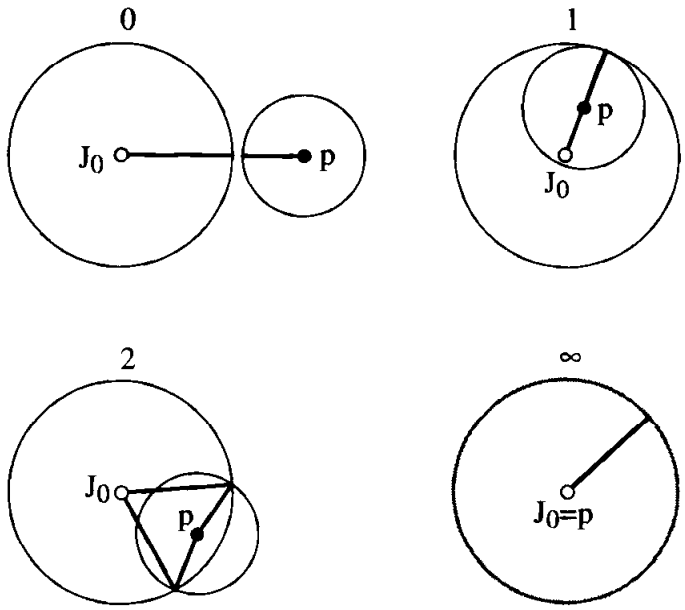


FIGURE 8.21 2-Link reachability: number of solutions shown.

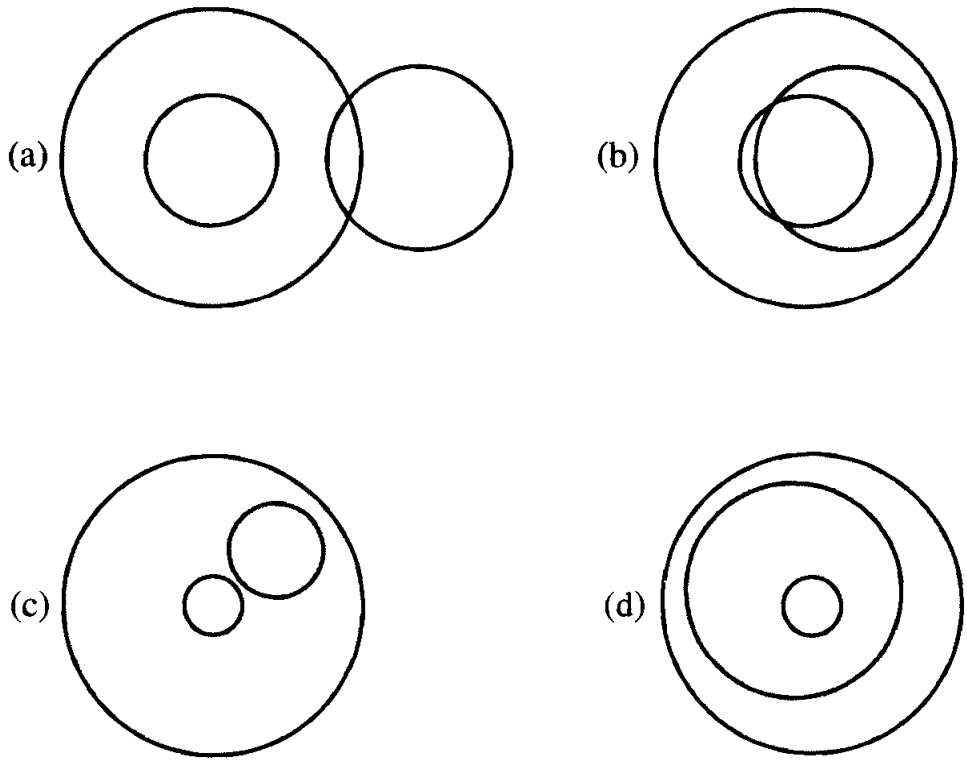


FIGURE 8.22 3-Link reachability. The shaded annulus is R; the other circle is C.

1. Case 1: $\partial R \cap C \neq \emptyset$ (Figure 8.22(a,b)).
   In this case, the problem can be reduced to a 2-link problem by aligning (a) or antialigning (b) L_1 and L_2. Of course there are in general infinitely many other solutions, but we restrict ourselves to seeking just one. It will be convenient to avoid antialignment of links, so we analyze Figure 8.22(b) a bit closer.

   Let $\partial R = I \cup O$, where I is the inner and O the outer boundary of the annulus. If $O \cap C = \emptyset$ and $I \cap C \neq \emptyset$ as in Figure 8.22(b), we can choose a circle C_2 of

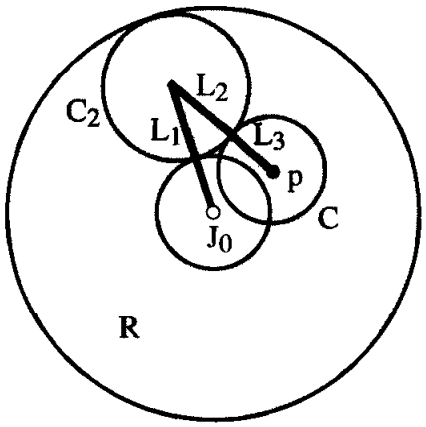


FIGURE 8.23 Aligning links L_2 and L_3 when $C \cap I \neq \emptyset$.

radius ℓ_2 tangent to C, which permits reaching p by alignment of L_2 and L_3 rather than antialignment of L_1 and L_2. See Figure 8.23.

2. Case 2: $\partial R \cap C = \emptyset$.
   Two further cases can be distinguished here, depending on whether or not C encloses the origin J_0.
   (a) C does not enclose J_0 (Figure 8.22(c)).
   We claim that again it is possible to find a solution with two links aligned. Let C_2 be a circle of radius ℓ_2 in the annulus R and tangent to C. Then L_2 and L_3 can be aligned (in a manner similar to Figure 8.23), which again reduces the problem to two links.
   (b) C does enclose J_0 (Figure 8.22(d)).
   Here there is no solution in which two links align (or antialign), dashing hopes that every 3-link problem can be solved by such alignments. Nonetheless, there is another feature of this situation that makes it easy to solve: There is a solution for every value of j_0!
   To see this, choose j_0 arbitrarily, and draw a circle C_2 centered on J_1. Because C is in the annulus R and encloses the origin, it must enclose I, the inner boundary of R. Since C_2 connects the inner to the outer boundary of R, it must cross C somewhere. That crossing provides a solution for an arbitrary j_0.
   Thus we can reduce this case to 2 links after all: Choose j_0 arbitrarily, say $j_0 = 0$, and then solve the resulting 2-link problem.

We summarize in a lemma:

Lemma 8.6.4. *Every 3-link problem may be solved by one of the following 2-link problems:*

(1) $(\ell_1 + \ell_2, \ell_3)$.
(2) $(\ell_1, \ell_2 + \ell_3)$.
(3) $j_0 = 0$ *and* (ℓ_2, ℓ_3).

Proof. Figure 8.22(a) corresponds to (1), Figure 8.22(b) (and Figure 8.23) and Figure 8.22(c) correspond to (2), and Figure 8.22(d) corresponds to (3). □

n-Link Reachability

Linear Algorithm for n-Link Reachability. Reexamine Figure 8.22, but now imagining the annulus R representing $n-1$ links of an n-link arm A, with the circle C of radius ℓ_n, centered on p. Since we are assuming A can reach the target point, we know $R \cap C$ is nonempty. Indeed the possibilities for intersection are just those illustrated in Figure 8.22. This suggests the following recursive procedure[15] for determining a configuration for an n-link arm to reach a given reachable point p:

1. Case 1: $\partial R \cap C \neq \emptyset$ (Figure 8.22(a,b)).
   Choose one of the (in general) two points of intersection t.
2. Case 2: $R \supseteq C$ (Figure 8.22(c,d)).
   Choose any point t on C, say the point furthest from J_0.

In either case, recursively find a configuration for $A_{n-1} = (\ell_1, \ldots, \ell_{n-1})$ to reach t. Append the last link L_n to this solution to connect t to p (recall C is centered on p). The base of the recursion can be our previously outlined solution to the 3-link problem.

Because the cases in Figure 8.22 are exhaustive, this procedure is guaranteed to find a solution (if one exists). That it requires only $O(n)$ time follows from the fact that reducing n by 1 is accomplished in constant time, by intersecting C with O and with I, where $\partial R = I \cup O$.

This then achieves our goal: Given a point p to reach, and a list of link lengths specifying the arm, first determine if p is reachable with Theorem 8.6.3, and if it is, find a configuration via this recursive procedure.

Two Kinks. Although it is not possible to improve on the asymptotic time complexity of $O(n)$, for it takes that long just to sum the link lengths, there is in fact a significant conceptual simplification possible. One hint is provided by the simplicity of the solution obtained in Case 1 of the above algorithm: The first $n-1$ links are straightened out if $p \in O$, and they are "kinked" only at the joints on either end of the longest link if $p \in I$. This latter claim follows from the formula for r_i: All links "oppose" L_M (the longest link) to reach a point on the inner annulus radius. Thus the arm need not have many kinks in Case 1. And in Case 2 p could lie anywhere on C, suggesting that this freedom might be exploited to avoid kinks.

In fact, it is a remarkable theorem that if an n-link arm can reach a point, it can do so with only two kinked joints![16] Moreover, which two joints can be easily determined. The implication of this is that any n-link problem can be directly reduced to one 3-link problem! We now proceed to prove this.

Theorem 8.6.5 (Two Kinks). *If an n-link arm A can reach a point, it can reach it with at most two joints "kinked": Only two joints among $J_1, \ldots, J_{n-1}$ have nonzero angles. The two joints may be chosen to be those at either end of the "median link": The link L_m such that $\sum_{i=1}^{m-1} \ell_i$ is less than or equal to half the total length of the links, but $\sum_{i=1}^{m} \ell_i$ is more than half.*

[15] Suggested by Carl Lee.

[16] This result is implicit in the work of Kutcher (1992, pp. 191–3) and that of Lenhart & Whitesides (1992). Both works prove stronger results.

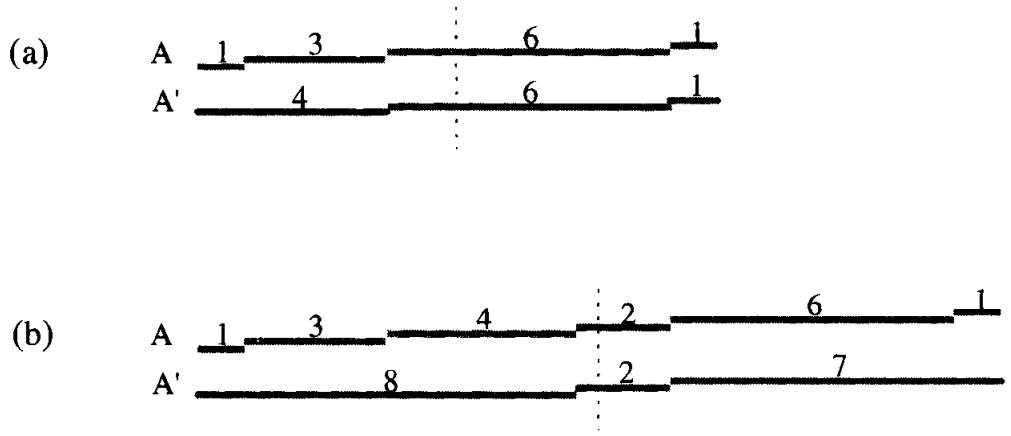


FIGURE 8.24 2-Kinks theorem, with links shown staggered for clarity: (a) $\ell/2 = 5\frac{1}{2}$, $r_i = 1 > 0$, longest link is the median link; (b) $\ell/2 = 8\frac{1}{2}$, $r_i = 0$, median link is not the longest link.

Proof. The strategy of the proof is to modify the arm A by "freezing" all but the two indicated joints, and showing that the resulting new arm A' has the same reachability region. A joint is "frozen" by fixing its angle to 0. Note that since r_o depends only on the sum of the link lengths (Theorem 8.6.3), such freezing leaves r_o fixed. So the onus of the proof is to show that r_i is also unaltered.

Let ℓ be the total length of the links. We partition the work into two cases, depending on whether or not $r_i = 0$.

1. Case $r_i > 0$ (Figure 8.24(a)).
   Recall from Theorem 8.6.3 that r_i is nonzero only when the longest link L_M exceeds the length of the remaining links. Then it must be that $\ell_M > \ell/2$. Therefore $L_M = L_m$ regardless of where it appears in the sequence of links: Because L_M is so long, it covers the midpoint of the lengths under any shift in the sequence.

   Now because $L_m = L_M$ and $\ell_M > \sum_{i \neq M} \ell_i$, if we freeze all joints except those at the endpoints of L_M to form a new arm A', we do not change the fact that L_M is the longest link. (In Figure 8.24(a), the longest link length is 6 in both A and A'.) Since r_i depends only on ℓ and ℓ_M by Theorem 8.6.3 again, A' has the same reachability region as A.
2. Case $r_i = 0$ (Figure 8.24(b)).
   In this case we know from Theorem 8.6.3 that the longest link L_M is $\leq \ell/2$, since $\ell_M \leq \sum_{i \neq M} \ell_i$. Let L_m be the median link, and freeze all joints before and after L_m, forming arm A'. This might change which link is longest; in Figure 8.24(b), the longest link length is 6 in A and 8 in A'. But note that the new longest link L'_M cannot exceed $\ell/2$ in length: Since L_m straddles the midpoint of the lengths, both what precedes it and what follows it must be $\leq \ell/2$. Since r_i is only nonzero when the longest link exceeds $\ell/2$, we are assured that r_i is still zero. Therefore the reachability region of A' is the same as that of A. □

Algorithm. The two-kinks theorem gives us an alternative $O(n)$ algorithm, where the only part that depends on n is summing the lengths of the n links: After that the algorithm is constant-time. So if we count the number of circle-intersection tests performed, the recursive algorithm requires $O(n)$ of these, whereas the two-kinks algorithm only needs $O(1)$. For after identifying L_m, the problem is reduced to a single 3-link problem,

which is reduced by Lemma 8.6.4 to three 2-link problems, each of which performs one circle-intersection test.

8.6.3. Implementation of Link Configuration

The implementation of the just-described algorithm is relatively straightforward, although intersecting two circles requires some care. We first describe the top-level procedures before plunging into the circle-intersection detail.

The link lengths are stored in an integer array. Throughout the code we stick to integers until we are forced to use `doubles`, as of course we will be (for circle intersection). This isolates problems that might arise from floating-point calculations. The main routine and data structures are as shown in Code 8.6. After reading the link lengths with `ReadLinks`, `main` enters a loop that reads in a target and solves the reachability problem for that target with a call to `Solven`. This routine initiates a cascade of function calls, each reducing the problem to a simpler problem: `Solven` → `Solve3` → `Solve2` → `TwoCircles` → `TwoCirclesOa` → `TwoCirclesOb` → `TwoCirclesOO`. The three `Solvex` routines are Boolean functions, returning `TRUE` iff the target is reachable. The four `TwoCircles` routines compute the number of circle intersections and one point of intersection `p`. This point is passed back up as `J` to `Solve3`, which prints out the solution. We now describe each of the main functions.

```
/* Global variables. */
int     linklen[NLINKS];     /* link lengths */
int     nlinks;              /* number of links */
tPointi target;              /* target point */
main()
{
  tPointi origin = {0,0};

  nlinks = ReadLinks();
  while (TRUE) { /* loop broken by EOF in ReadTarget */
     ReadTarget( target );
     MoveTo_i( origin );
     if ( !Solven( nlinks ) )
        printf("Solven: no solutions!\ n");
     LineTo_i( target );
  }
}
```

Code 8.6 `main`.

The `Solven` procedure (Code 8.7) identifies the median link and calls `Solve3` with the joints fore and aft of it frozen. Throughout the code, `L1`, `L2`, ... are used to represent the lengths $\ell_1, \ell_2, \ldots$, to avoid the awkward typography of "`l1`." `Solve3` (Code 8.8) follows Lemma 8.6.4, calling `Solve2` as many as three times. Only the last

call results in two kinked joints. `Solve2` (Code 8.9) simply arranges the arguments for `TwoCircles`, which intersects two circles.

```
bool  Solven( int nlinks )
{
  int i;
  int m;              /* index of median link */
  int L1, L2, L3;     /* length of links between kinks */
  int totlength;      /* total length of all links */
  int halflength;     /* floor of half of total */

  /* Compute total and half length. */
  totlength = 0;
  for ( i = 0; i < nlinks; i++ )
          totlength += linklen[i];
  halflength = totlength / 2;

  /* Find median link. */
  L1 = 0;
  for ( m = 0; m < nlinks; m++ ) {
    if ( (L1 + linklen[m]) > halflength)
      break;
    L1 += linklen[m];
  }

  L2 = linklen[m];
  L3 = totlength - L1 - L2;
  if ( Solve3( L1, L2, L3, target ) )
       return TRUE;
  else return FALSE;
}
```

Code 8.7 `Solven`.

Intersection of Two Circles

Two circles can clearly be intersected in constant time, so the only issues are practical. We develop code general enough to be used in other applications.

Let the two circles C_1 and C_2 have centers $c_i = (a_i, b_i)$ and radii r_i, $i = 1, 2$. Because the equation of a circle is a quadratic equation, on the basis of general algebraic principles,[17] we can expect there to be no more than four intersections. But in fact there can be no more than two intersections because of the special form of the equations. Of course there can also be zero, one, or an infinite number of intersections, as previously shown in Figure 8.21. The first task is to distinguish these cases; the second is to solve the generic two-intersection case.

[17] Bezout's Theorem: The number of proper intersections between two plane curves of algebraic degree m and n is at most mn.

```
bool  Solve3( int L1, int L2, int L3, tPointi target )
{
   tPointd Jk;          /* coords of kinked joint returned by Solve2 */
   tPointi J1;          /* Joint1 on x axis */
   tPointi Ttarget;     /* translated target */

   if ( Solve2( L1 + L2, L3, target, Jk ) ) {
       LineTo_d( Jk );
       return TRUE;
   }
   else if ( Solve2( L1, L2 + L3, target, Jk ) ) {
       LineTo_d( Jk );
       return TRUE;
   }
   else {       /* pin J0 to 0. */
       /* Shift so J1 is origin. */
       J1[X] = L1; J1[Y] = 0;
       SubVec( target, J1, Ttarget );
       if ( Solve2( L2, L3, Ttarget, Jk ) ) {
          /* Shift solution back to origin. */
          Jk[X] + = L1;
          LineTo_i( J1 );
          LineTo_d( Jk );
          return TRUE;
       }
       else
          return FALSE;
   }
}
```

Code 8.8 `Solve3`.

```
bool    Solve2( int L1, int L2, tPointi target, tPointd J )
{
   tPointi c1 = {0,0};     /* center of circle 1 */
   int nsoln;              /* # of solns: 0,1,2,3(infinite) */

   nsoln = TwoCircles( c1, L1, target, L2, J );
   return nsoln != 0;
}
```

Code 8.9 `Solve2`.

It will simplify matters considerably to arrange the circles conveniently with respect to the coordinate system. It will be no loss of generality to assume that $c_1 = (0, 0)$ and $c_2 = (a_2, 0)$. The sole function of `TwoCircles` (Code 8.10) is to ensure half of this by translating so that $c_1 = (0, 0)$ and calling `TwoCircles0a`.

```
/* TwoCircles finds an intersection point between two circles.
   General routine: no assumptions. Returns # of intersections; point in p. */
int  TwoCircles( tPointi c1, int r1, tPointi c2, int r2,
                 tPointd p)
{
  tPointi c;
  tPointd q;
  int nsoln = -1;

  /* Translate so that c1 = {0,0}. */
  SubVec( c2, c1, c );
  nsoln = TwoCircles0a( r1, c, r2, q );
  /* Translate back. */
  p[X] = q[X] + c1[X];
  p[Y] = q[Y] + c1[Y];
  return nsoln;
}
```

Code 8.10 `TwoCircles`. `SubVec` is in Code 7.6.

`TwoCircles0a` (Code 8.11) handles all the special cases. Continuing our resolve to stick with integers until forced to floating-point numbers, we detect all special cases prior to floating-point division. This is possible because we assumed the target point has integer coordinates. Computing $(r_1 + r_2)^2$ and $(r_1 - r_2)^2$ and comparing against the square of the distance to c_2 permits detection of the zero-, one-, and infinite-intersections cases. We convert to `doubles` to gain precision and thereby protect against integer overflow in the squaring, as we did earlier in Section 7.2, but the comparisons remain between integers. (Without this protection, radii of 10^5 lead to overflow.) In the one-intersection cases, we know the point of intersection is at distance r_1 from the origin, a fraction of the way to c_2. For example, if $r_1 = 10$, $r_2 = 15$, and $c_2 = (-3, -4)$, then $(r_1 - r_2)^2 = 25 = |c_2|^2$, and the fraction $f = \frac{10}{-5} = -2$ is used to compute the intersection point $p = f \cdot c_2 = (6, 8)$.

If no special case holds, then `TwoCircles0a` calls `TwoCircles0b` (Code 8.12). This routine ensures the second half of convenient arrangement within a coordinate system by placing c_2 on the x axis. It rotates c_2 so that it lies on the x axis, calls `TwoCircles00` to solve the problem in this rotated coordinate system, and rotates back. Rotation is performed by the standard method, well-known in graphics:[18] multiplying the point q by the rotation matrix R:

$$\begin{bmatrix} p_0 \\ p_1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} q_0 \\ q_1 \end{bmatrix}. \tag{8.1}$$

Note that $\sin\theta$ and $\cos\theta$ can be computed by simple ratios; no calls to trigonometric library functions are needed.

[18] See, e.g., Rogers & Adams (1990, Sec. 2–9).

```
/* TwoCircles0a assumes that the first circle is centered on the origin.
   Returns # of intersections: 0, 1, 2, 3 (inf); point in p. */
int  TwoCircles0a( int r1, tPointi c2, int r2, tPointd p )
{
    double dc2;                 /* dist to center 2 squared */
    double rplus2, rminus2;     /* (r1 +/- r2)^2 */
    double f;                   /* fraction along c2 for nsoln = 1 */

    /* Handle special cases. */
    dc2 = Length2( c2 );
    rplus2 = (r1 + r2) * (r1 + r2);
    rminus2 = (r1 - r2) * (r1 - r2);

    /* No solution if c2 out of reach + or -. */
    if ( ( dc2 > rplus2 ) || ( dc2 < rminus2 ) )
       return 0;

    /* One solution if c2 just reached. */
    /* Then solution is r1-of-the-way (f) to c2. */
    if ( dc2 == rplus2 ) {
        f = r1 / (double)(r1 + r2);
        p[X] = f * c2[X]; p[Y] = f * c2[Y];
        return 1;
    }
    if ( dc2 == rminus2 ) {
       if ( rminus2 == 0 ) { /* Circles coincide. */
           p[X] = r1; p[Y] = 0;
           return 3;
       }
       f = r1 / (double)(r1 - r2);
       p[X] = f * c2[X]; p[Y] = f * c2[Y];
       return 1;
    }
    /* Two intersections. */
    return TwoCircles0b( r1, c2, r2, p );
}
```

Code 8.11 `TwoCircles0a; Length2` not shown.

Finally, `TwoCircles00` (Code 8.13) performs the generic two-intersections computation. The task is to solve these two equations simultaneously:

$$x^2 + y^2 = r_1^2,$$
$$(x - a_2)^2 + y^2 = r_2^2.$$

Solving the first equation for y^2 and substituting into the second yields $(x - a_2)^2 + r_1^2 -$

$x^2 = r_2^2$, which can be solved for x:

$$x = \frac{1}{2}\left(a_2 + \frac{r_1^2 - r_2^2}{a_2}\right)$$

```
/* TwoCircles0b also assumes that the 1st circle is origin-centered. */
int  TwoCircles0b( int r1, tPointi c2, int r2, tPointd p )
{
   double a2;              /* center of 2nd circle when rotated to x axis */
   tPointd q;              /* one solution when c2 on x axis */
   double cost, sint;      /* sine and cosine of angle of c2 */

   /* Rotate c2 to a2 on x axis. */
   a2 = sqrt( Length2( c2 ) );
   cost = c2[X] / a2;
   sint = c2[Y] / a2;

   TwoCircles00( r1, a2, r2, q );

   /* Rotate back */
   p[X] = cost * q[X] + -sint * q[Y];
   p[Y] = sint * q[X] +  cost * q[Y];

   return 2;
}
```

Code 8.12 `TwoCircles0b`.

```
/* TwoCircles00 assumes circle centers are (0,0) and (a2,0). */
void  TwoCircles00( int r1, double a2, int r2, tPointd p )
{
    double r1sq, r2sq;
    r1sq = r1*r1;
    r2sq = r2*r2;

    /* Return only positive-y soln in p. */
    p[X] = ( a2 + ( r1sq - r2sq ) / a2 ) / 2;
    p[Y] = sqrt( r1sq - p[X]*p[X] );
}
```

Code 8.13 `TwoCircles00`.

Note that $a_2 \neq 0$ because we have already eliminated the no-solutions and infinite-solutions cases. From x we solve for y by substituting back into one of the circle

equations. The two solutions have the same x coordinate, and with one y coordinate the negative of the other. This is the advantage of working in a convenient coordinate system. The remaining utility routines necessary to make working code are straightforward and not shown.

It is unfortunately typical of computational geometry code that a large portion of the effort is spent dispensing of the special cases. In this case, the actual circle intersection is performed in two lines of code, but these are preceded by many other lines arranging that those two lines work correctly.

Example

Consider a 4-link arm, with link lengths 100, 10, 40, and 90 respectively. We first examine one particular target in detail and then look at the output for a series of targets. Start with the goal to reach back to the shoulder as target, (0, 0). `Solven` computes the total length to be 240 and identifies the third link as the median link. It then calls `Solve3(110,40,90)`. This in turn calls `Solve2(150,90)` and `Solve2(110,130)`, both of which fail: the former because the hand can't reach back to (0, 0), the latter because the hand necessarily overshoots (0, 0). We then fall into the third case of the 3-link lemma (Lemma 8.6.4): The first joint is fixed at $j_0 = 0$ and `Solve2(40,90)` is called, this time trying to reach $(-110, 0)$ (because the first two links are frozen at 0° and length $\ell_1 = 100 + 10$). This succeeds, finding the intersection point $p = (25.45, 30.86)$ in `TwoCircles00`, which, after reversing the transformations, is returned as $p = (-25.45, -30.86)$ from `Solve2`, and finally $p = (84.55, -30.86)$ from `Solve3`. It is this point that is printed as the coordinate of J_3. The corresponding arm configuration is shown in Figure 8.25.

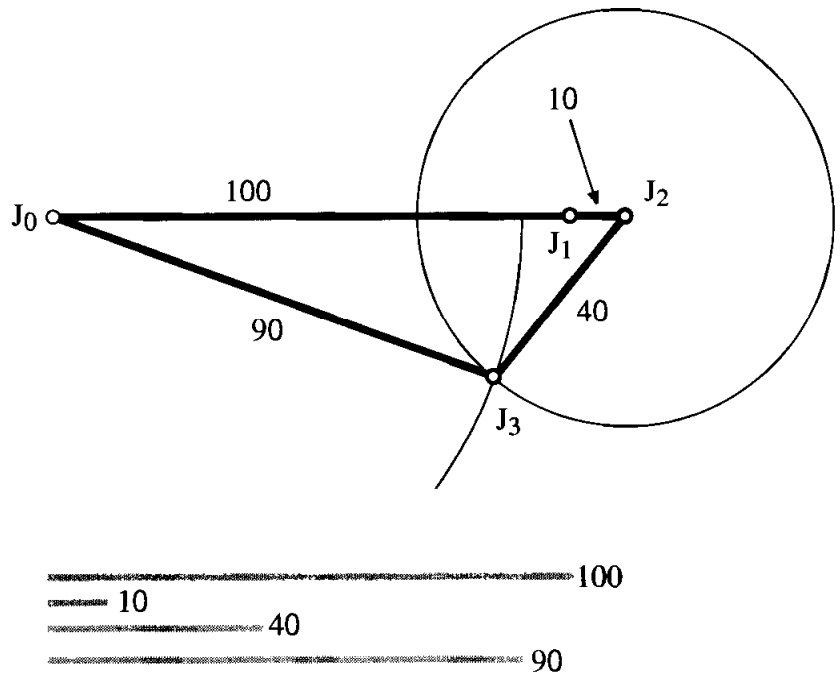


FIGURE 8.25 A 4-link example: $A = (100, 10, 40, 90)$. $J_0 = (0, 0)$; $J_1 = (100, 0)$; $J_2 = (110, 0)$; $J_3 = (84.55, -30.86)$; and $J_4 = J_0$.

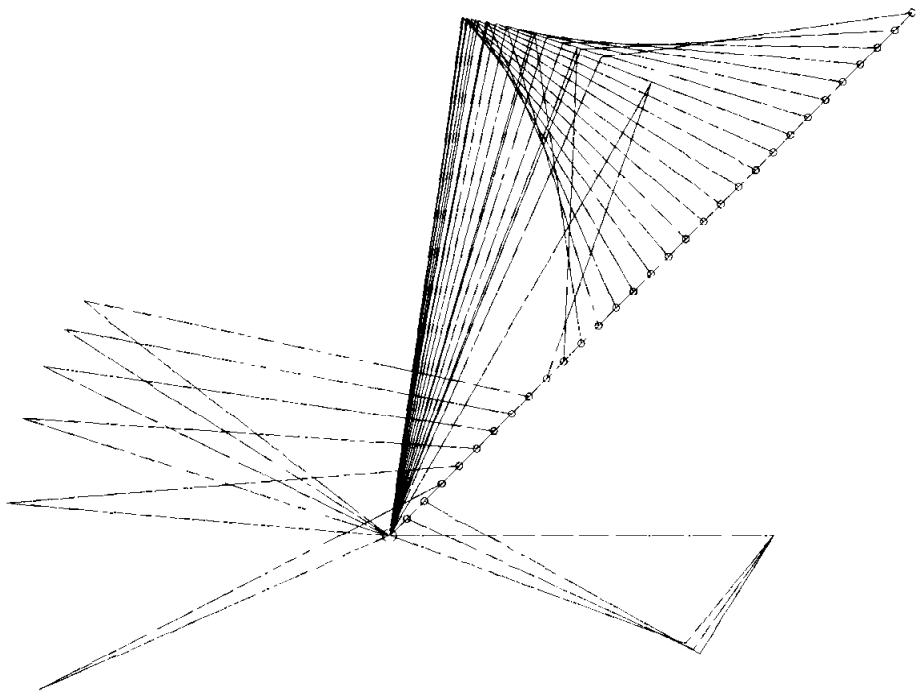


FIGURE 8.26 Arm configurations for targets (circled) along the line $y = x$ from (0, 0) to (150, 150).

Now we examine the behavior of the code on the same four links, but with a series of targets, $(5k, 5k)$ for $k = 0, 1, \ldots, 30$. The output of the code is shown in Figure 8.26. For $k = 0$ and target (0, 0), we obtain the solution displayed in Figure 8.25 (to a different scale). At $k = 3$, the target (15, 15) is reachable via `Solve2(110,130)` (the second case of Lemma 8.6.4) for the first time, resulting in J_2 "jumping" to $(-100.67, -44.33)$. At $k = 9$, the target (45, 45) is reachable via `Solve2(150,90)` for the first time, causing another discontinuity in the configuration. For the remaining larger values of k, the target is reachable with this same first option of Lemma 8.6.4. One can see from this example that finding a sequence of smoothly changing configurations that track a moving target would be an interesting problem (Exercises 8.6.4[6]).

8.6.4. Exercises

1. *Turning a polygon inside out.* Imagine a polygon whose edges are rigid links, and whose vertices are joints. "To turn a polygon inside out is to convert it by a continuous motion in the plane to the polygon that is the mirror image (with respect to some arbitrary line in the plane) of the original one" (Lenhart & Whitesides 1991). Here the intent is to permit intermediate figures to be self-crossing polygons. Can this be done for every polygon? If so prove it. If not, find conditions that guarantee configuration inversion.
2. *Division by zero* [programming]. Establish under what conditions division by zero can occur in `TwoCircles0a` (Code 8.11), `TwoCircles0b` (Code 8.12), or `TwoCircles00` (Code 8.13). Do there exist inputs that will force the code to realize these conditions? Test your conclusions on the code.
3. *Reachability region with pole(s).* Decide what is the reachability region of a 2-link arm if there are impenetrable obstacles in the plane, for example poles through which the arm may not pass.

In particular, consider the following obstacles:

a. A single point.

b. Two points.

c. One disk.

4. *Line tracking.* Define a continuous motion of an arm to be *line tracking* if the hand moves along a straight line (Whitesides 1991).

   a. Can a 2-link arm track a line? Can it track every line?

   b. Can a 3-link arm track a line? Can it track every line?

5. *Joint constraints.* Suppose that each joint is only free to move within a certain angular range, $\pm\theta_i$ for j_i.

   a. What is the reachability region of a joint-constrained 2-link arm?

   b. What is the reachability region of a joint-constrained 3-link arm?

6. *Smooth tracking.* Let a target point $p(t)$ move smoothly as a function of time t.

   a. Define what it should mean for an arm configuration to track $p(t)$ "smoothly."

   b. Can you find an example where the arm can reach $p(t)$ for all t, but there is no series of reaching configurations that satisfy your definition of smooth tracking?

8.7. SEPARABILITY

A number of applications in robotics (especially mechanical assembly), in circuit layout, and in graphics have led to research on a variety of "separability" problems, where objects are to be separated from one another without collision. A typical instance of this problem models the situation faced by movers emptying a house of its contents. Given a collection of disjoint polygons in the plane, may each be moved "to infinity" without disturbing the others? Of course the motion must be a continuous motion in the plane. Collision avoidance is the same concept as used in the robot motion planning problems: Two polygons collide if they share an interior point. By moving "to infinity" is meant moving arbitrarily far away. Often constraints are placed on the types of movement permitted (e.g., translation only). As we will see, it is also important to specify whether only one polygon can move at a time, or can several move simultaneously. In this section we dip into this area just enough to suggest its richness.[19]

8.7.1. Varieties of Separability

Not all collections of polygons are separable, even with no restriction on allowable motions: Figure 8.27(a) shows two interlocked polygons that are inseparable (without lifting one into the third dimension!). Some sets of polygons are separable if rotation is permitted, but inseparable via translation only, as are the pair in Figure 8.27(b).

If only one polygon may be moved at a time, during which time all others stay fixed, then it may be that a set of polygons are separable, but only with a huge number of motions. Figure 8.28[20] shows an instance where the configuration is separable by moving A and B alternately to the right, freeing Q to move upwards and right. But the

[19] See Toussaint (1985*b*) for a survey.

[20] Based on Figure 3.1 of Chazelle, Ottmann, Soisalon-Soininen & Wood (1984).

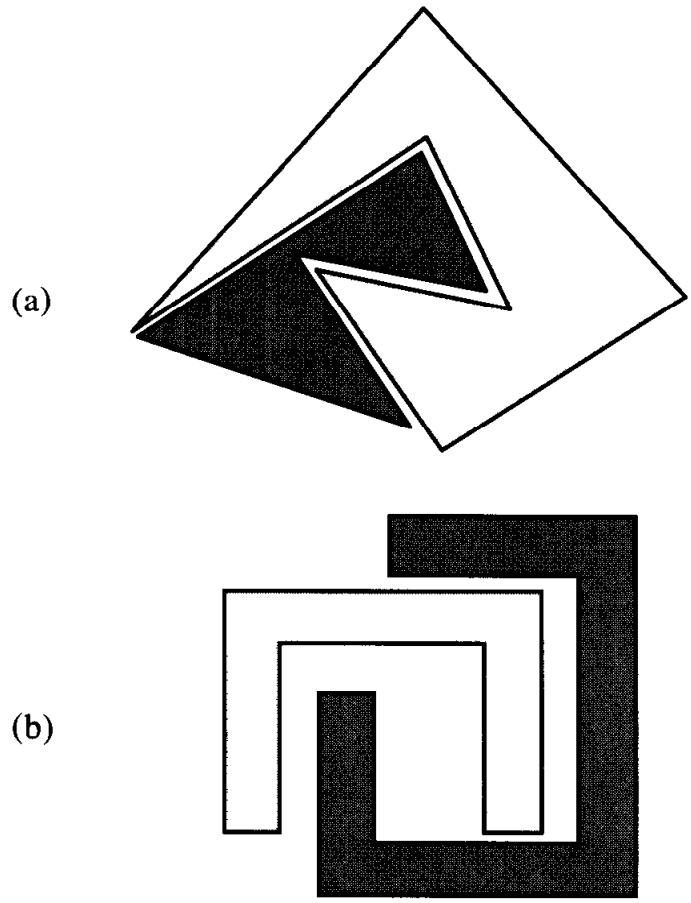


FIGURE 8.27 (a) Inseparable polygons; (b) inseparable by translations, but separable using rotations.

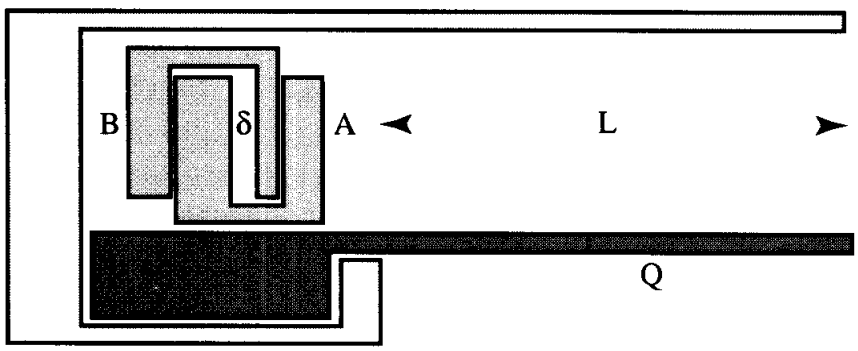


FIGURE 8.28 The number of one-at-a-time moves needed to separate this collection is proportional to L/δ.

number of moves to get A and B out depends on the gap δ with respect to the length L: The number of moves is at least L/δ, which can be made arbitrarily large independent of n, the number of vertices. This example hardly seems to demonstrate that the problem is truly difficult, however: It is easy to separate this collection of polygons if two may be moved simultaneously; and even when only one is moved at a time, no polygon has to move a "large" total distance (large with respect to, e.g., the diameter of the hull of the original configuration). Nevertheless we will see that the separability problem is indeed "hard" in these senses.

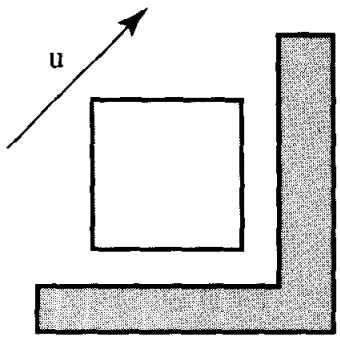


FIGURE 8.29 Polygons inseparable along u.

8.7.2. Separability by Translation

The earliest, and still perhaps the prettiest, result on separability was obtained by Guibas & Yao (1983). They proved that a collection of convex polygons can be separated under the following motion conditions:

1. Translation: All motions are translations.
2. Unidirectional: All translations are in the same (arbitrary) direction.
3. Moved once: Each polygon is moved only once.
4. One-at-a-time: Only one polygon is moved at a time.

These are severe restrictions, and many otherwise separable polygons are inseparable under them: For example, the pair of polygons in Figure 8.29 are inseparable along the direction u. However, convex polygons (or curved convex shapes, for that matter) are separable under these conditions, and along any direction. If the reader finds this intuitively obvious, it may prove a jolt to learn that convex objects in three dimensions are not always separable under these conditions (Exercise 8.7.5[1]).

Application

The work of Guibas & Yao (1983) was motivated by the then-new technology of windows on workstations. Some of these workstations have a hardware instruction that copies a block of screen memory from one location to another. Shifting several windows with this instruction without overwriting memory can be solved by moving each according to a separability ordering.

Separating Segments

We start with a special case, which we will soon see suffices to encompass the general case: separating a set of disjoint segments. Let the direction in which the segments are to be separated be the positive x direction; we can choose this without any loss of generality. It should be clear that if we can identify one segment in any collection that can be moved horizontally rightward without colliding with any other, then the segments are separable along that direction. For after we move that one to infinity, we have a smaller instance of the same problem, and we can identify another that can be moved, and so on.

Imagine illuminating the segments from $x = +\infty$, as depicted in Figure 8.30. Our question becomes: Must there always be one segment completely illuminated?

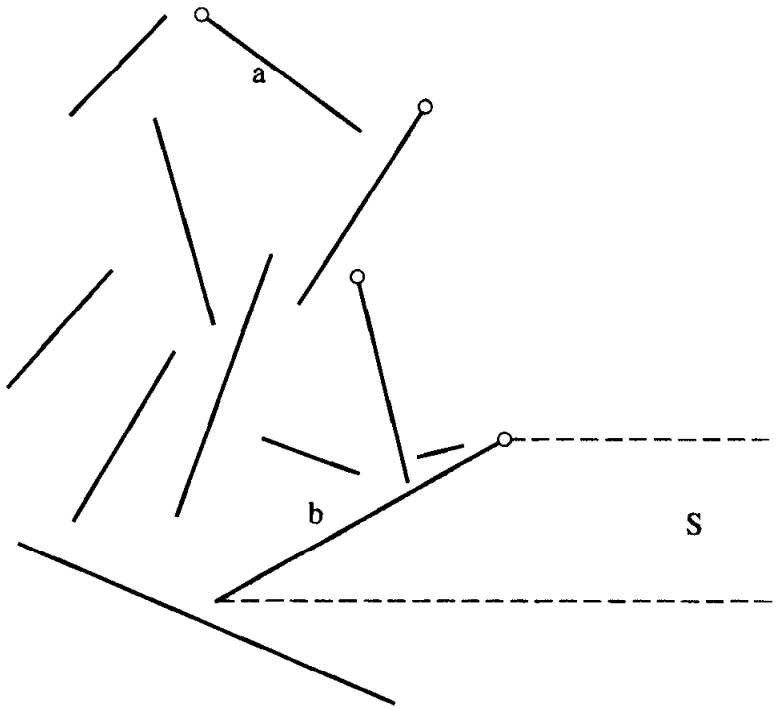


FIGURE 8.30 One segment b is always illuminated from $x = +\infty$.

Lemma 8.7.1. *In any collection of disjoint line segments, there is always at least one that is completely illuminated from* $x = +\infty$.

Proof. We first examine the subset U of segments whose upper endpoint is illuminated, that is, a horizontal rightward ray from their upper endpoint does not hit any segment. Certainly U is not empty: Consider the segments whose upper endpoint is highest. If there is just one, then it is in U. If there are several tied for highest, then the one with the rightmost upper endpoint is in U (segment a in Figure 8.30).

As the figure shows, this rightmost highest segment is not necessarily completely illuminated: a is blocked from below. But our claim is that the segment b in U with the lowest upper endpoint is completely illuminated. Let S be the infinite strip to the right of b. Because the upper endpoint of b is visible from $x = +\infty$, if any portion of S is blocked by a segment c, the upper endpoint of c must lie in S. Then the highest upper endpoint of all the segments blocking S must be illuminated, contradicting our assumption that b has the lowest illuminated upper endpoint. □

Separating Convex Polygons

The problem for convex polygons is now solved by the simple observation that the region swept by the right boundary of a convex shape C moving horizontally is a subset of the region swept by a line segment s between the leftmost highest and lowest points of C (see Figure 8.31). Therefore a schedule for separating such vertically spanning segments for a set of convex shapes will suffice to separate the shapes themselves.

Complexity

Computing a separating order for a set of convex shapes is similar to sorting them along the direction of separation, so it should not be surprising that it can be accomplished in $O(n \log n)$ time. We will not prove this result of Guibas & Yao (1983).

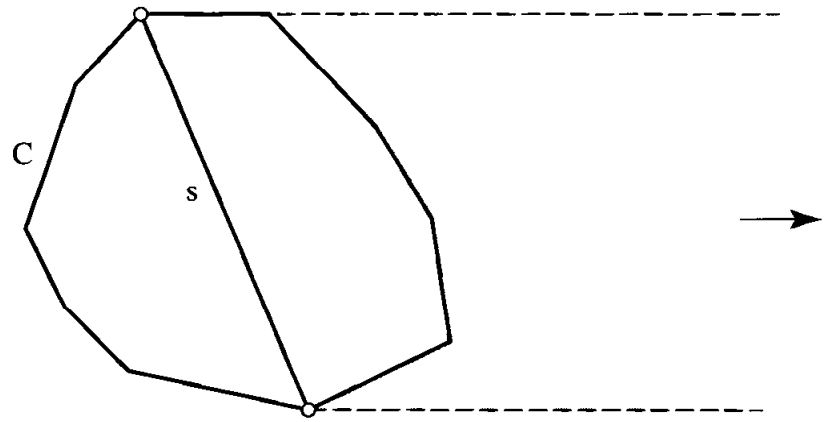


FIGURE 8.31 The region swept by C is a subset of the region swept by s.

Theorem 8.7.2. *Any set of n convex shapes in the plane may be separated via translations all parallel to any given fixed direction, with each shape moving once only. An ordering for moving them can be computed in* $O(n \log n)$ *time.*

8.7.3. Reduction from Partition

Having considered an "easy" instance of separability, we now demonstrate in this and the next subsection that general separability problems are "hard" in some sense. Proving hardness can be done by proving a lower bound on the problem, as we did in Chapter 3 (Section 3.9) by reduction from a known hard problem. Recall that the idea is to show that, if we could solve our problem B quickly, then we could solve some problem A quickly, where A is known to be difficult. This then establishes that B is at least as difficult as A: A has been *reduced to* B.

The separation problem B we examine allows only translation and movement of polygons one-at-a-time. But each translation can be in a different direction, and each polygon can be moved several times. The known difficult problem A is the *partition* problem: Given a collection S of integers, decide whether or not it may be partitioned into two parts whose sums are equal. For example, if $S = \{1, 3, 3, 5, 6\}$, the answer to the partition question is YES since $1 + 3 + 5 = 3 + 6$, but for the set $\{1, 3, 3, 5, 10\}$, the answer is NO. Although this may not seem like a very difficult problem, no one has been able to think of a way to solve it that is significantly better than examining every possible partition of S. Because there are 2^n possible partitions for a set of n elements, this is a very slow algorithm: It requires time exponential in n and so is effectively useless for e.g., $n \geq 100$. Moreover, the partition problem has been shown to be "NP-complete," which means that it is among a large class of apparently intractable problems.[21]

Given any instance of the partition problem, we construct a separability problem that can be solved iff the partition problem can. The construction is illustrated in Figure 8.32 for the set $\{1, 3, 3, 5, 6\}$.[22] It consists of blocks of height 1 and widths corresponding to each element of S. Let Σ be the sum of all the numbers in S. The piece Q in the figure

[21] As of this writing, however, it has not been proven that even the NP-complete problems are truly hard. This is the famous $P = NP$ question. See Garey & Johnson (1979).

[22] Based on Figure 4.1 of Chazelle et al. (1984).

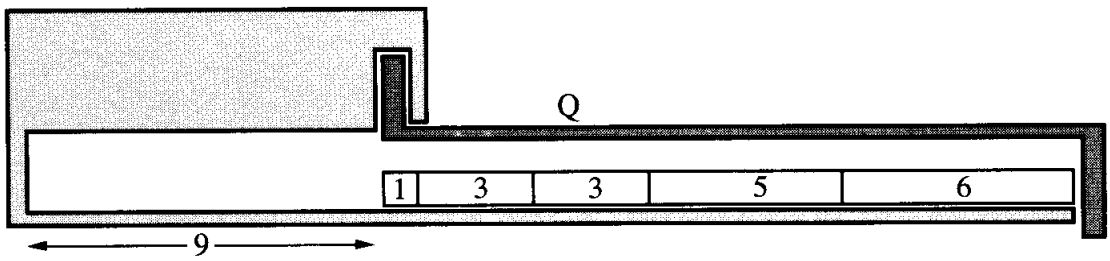


FIGURE 8.32 Q can be moved to infinity iff the partition problem can be solved.

can be moved down and right iff the blocks can be packed to the left within the $(\Sigma/2) \times 2$ rectangle of empty "storage" space. And this can be done iff S can be partitioned into two equal parts.

This proves that this version of the separation is at least as difficult as the partition problem—in the technical argot, separation is "NP-hard."

8.7.4. Mimicking the Towers of Hanoi

Although we have shown that separation is hard, note that separating the partition configuration does not require many moves: It may take considerable off-line thought, but the actual moves, once known, are easily accomplished. Separation can be effected by moving each block just once. We conclude with an example whose solution is not difficult to find, but which requires some pieces to be moved an exponential number of times. Again we restrict motions to be translations, and allow polygons to be moved more than once, but always one-at-a-time.

It is based on the well-known "Towers of Hanoi" puzzle. In this puzzle, disks of various radii are stacked on one of three pegs, sorted with largest on bottom and smallest on top; see Figure 8.33. The task is to move the disks one by one from peg A to peg B, using peg C whenever convenient, such that at all times not more than one disk is "in the air" (not on a peg) and no disk is ever placed on top of one of smaller radius. This "sorted at all times" condition forces many moves: $2^n - 1$ moves are required to move n disks from A to B (Rawlins 1992, p. 14–26).

A clever separation instance that mimics the Towers of Hanoi puzzle was suggested by Chazelle et al. (1984). The disks are simulated by n U-shaped polygons each of height h and thickness 1, which may nest snugly inside one another to form a stack $n + h$ tall; see Figure 8.34(a). Any stack not sorted by size must be at least $n + 2h$ tall, as

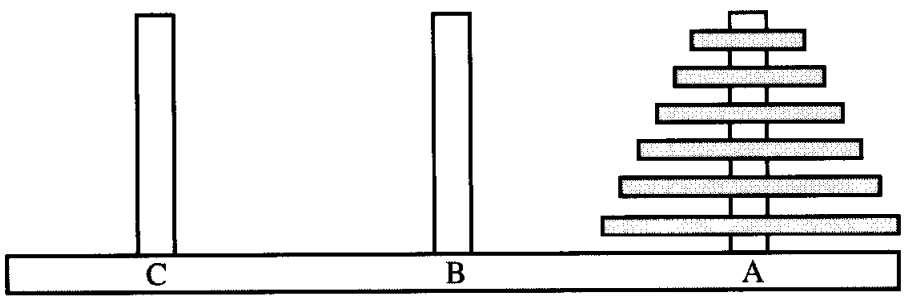


FIGURE 8.33 Towers of Hanoi, side view. A total of 63 moves are needed to transfer the stack to peg C.

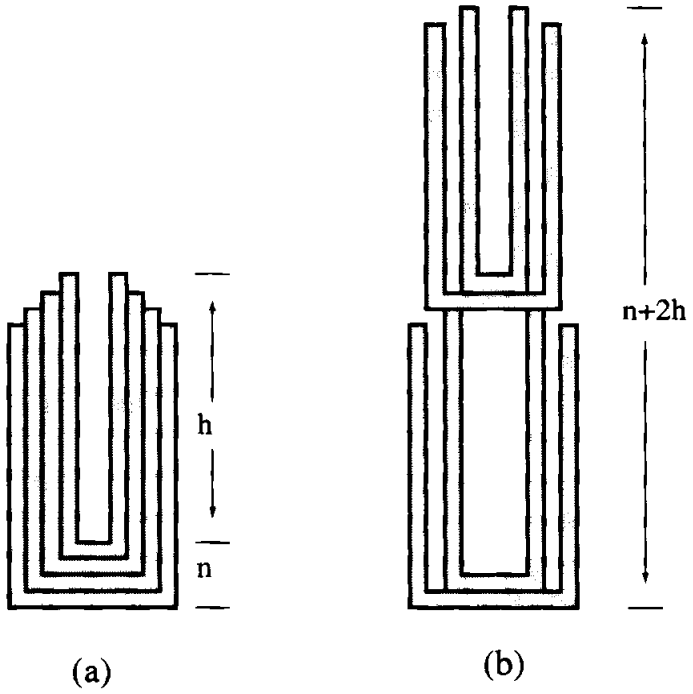


FIGURE 8.34 Stack of U-shaped polygons: (a) nested, (b) unsorted.

shown in (b) of the figure. By choosing h to be much larger than n, we can ensure that a sorted stack is much more compact than an unsorted stack.

The separation puzzle is shown in Figure 8.35.[23] The three rectangular "wells" labeled A, B, and C correspond to the three pegs. The polygon Q can be slid rightwards and down only when A is empty. A can only be emptied by moving the n Us into wells B and C. And because of the inefficiency of unsorted stacking, this can only be done by nearly mimicking the Tower of Hanoi moves, nearly in that it is possible to violate sorting once per column, but not more. It still requires an exponential number of moves for each U before Q can be separated.

8.7.5. Exercises

1. *Separating in three dimensions.* Find a set of convex polyhedra in three dimensions that cannot be separated à la Guibas & Yao (1983) (Section 8.7.2) in some direction.
2. *Separating spheres.* Prove or disprove that a collection of disjoint spheres in three dimensions may be separated one-at-a-time by translations parallel to any given direction.
3. *Nondisjoint segments.* Can Lemma 8.7.1 be extended to nondisjoint segments of this type: a collection of segments whose interiors are disjoint, but which may touch with the endpoint of one lying on another? The interior of a segment is the segment without its endpoints. A special case here is the edges of a polygon.
4. *Lower bound.* Show that $\Omega(n \log n)$ is a lower bound on computing the separating order for a disjoint set of line segments.
5. *Partition.* Strengthen the partition reduction to the case in which each piece is permitted just a single translation.

[23] Based on Figure 4.2 of Chazelle et al. (1984).

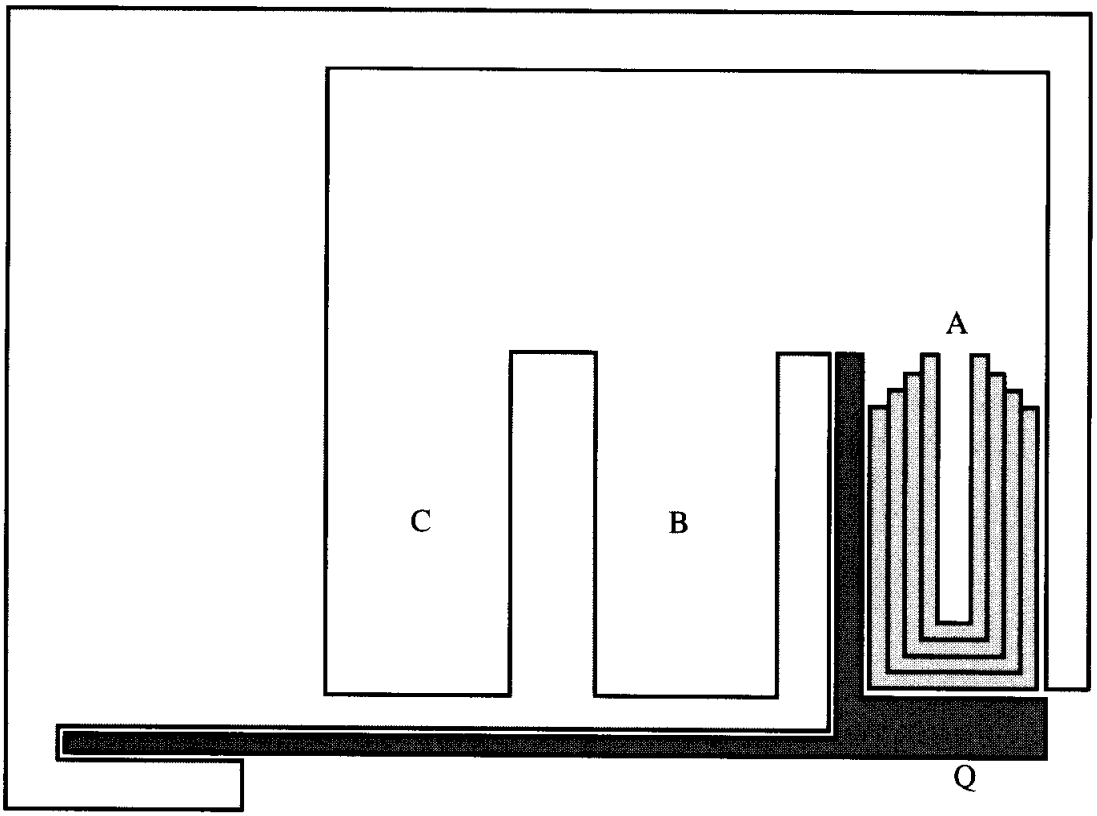


FIGURE 8.35 A separation puzzle based on the Towers of Hanoi.

6. *Hanoi improvements.*
   a. Exactly how many moves does it take to separate the configuration of polygons in Figure 8.35? Define a move as any continuous translation of one piece (not necessarily along a straight line).
   b. Prove that the puzzle in Figure 8.35 (for general n) requires an exponential number of moves, by proving an exponential lower bound on the number of moves required.
   c. Modify the structure of the puzzle so that the moves more closely mimic the Tower of Hanoi moves, requiring at least $2^n - 1$ moves of the n Us to clear A.
   d. Can the puzzle be modified so that it still requires an exponential number of moves even when any number of polygons may be moved simultaneously?
7. *Star polygons* (Toussaint 1985*b*). Recall from Chapter 1 (Exercise 1.1.4[5]) that a *star polygon* is one visible from a point in its interior.
   a. Does there always exist a single translation in some direction that will separate two star polygons? If not, provide a counterexample. If so, provide a proof.
   b. Answer the questions in (a) for three star polygons.
8. *Monotone polygons* (Toussaint 1985*b*). Recall from Chapter 2 (Section 2.1) that a *strictly monotone polygon* is one whose boundary meets every line parallel to some direction u in at most two points.
   a. Show that two strictly monotone polygons, monotone perhaps with respect to different directions, are always separable by a single translation.
   b. Design an algorithm for finding a direction that separates them.
   c. Do your results change if the polygons are monotone, but not strictly monotone, that is, if the polygon boundaries meet every line parallel to some direction in at most two connected sets (where these sets can now be line segments)?

9

Sources

This book has only scratched the surface of a large and evolving topic. This chapter lists various sources for those seeking further information. Although the lists may seem overwhelming, three sources may suffice for most purposes:

1. the *Handbook of Discrete and Computational Geometry* for short surveys,
2. the *Computational Geometry Community Bibliography* for bibliographic information, and
3. the *Directory of Computational Geometry Software* for software.

Each of these and many other sources are mentioned below.

9.1. BIBLIOGRAPHIES AND FAQs

Because computational geometry is a relatively young field, much of its literature is only available in primary sources: conference proceeding papers and journal articles. Fortunately, the community has developed a nearly comprehensive bibliography, freely available via `ftp`, complete with searching software. At this writing the bibliography contains 10,000 entries; fewer than 500 are books. I describe the *Computational Geometry Community Bibliography* in O'Rourke (1993). Its URL is `ftp://ftp.cs.usask.ca/pub/geometry/`.

Other bibliographies that include papers in computational geometry are available, most notably the *ACM SIGGRAPH Online Bibliography* at `ftp://siggraph.org/publications/bibliography/`.

Each Usenet newsgroup maintains a "FAQ," a file of answers to "frequently asked questions." There is no newsgroup specifically devoted to computational geometry, but a good portion of the traffic in `comp.graphics.algorithms` concerns geometric algorithms. Their FAQ is available from `ftp://rtfm.mit.edu/pub/faqs/graphics/algorithms-faq/`. It contains pointers to other relevant FAQs.

9.2. TEXTBOOKS

As the field matures, more textbooks are available, listed below. The three published in the 1980s remain useful:

Mehlhorn (1984): *Multi-Dimensional Searching and Computational Geometry.*
Preparata & Shamos (1985): *Computational Geometry: An Introduction.*
Edelsbrunner (1987): *Algorithms in Combinatorial Geometry.*

Mehlhorn's is especially clear on search data structures, and Preparata and Shamos's classic text, which greatly influenced the field, remains unsurpassed on many of the topics it covers. Edelsbrunner's text remains the best source on arrangements and computational geometry in arbitrary dimensions.

Two texts have emphasized particular aspects of computational geometry: randomized algorithms and parallel algorithms. These are:

Mulmuley (1994): *Computational Geometry: An Introduction through Randomized Algorithms.*
Akl & Lyons (1993): *Parallel Computational Geometry.*

The most recent textbook published is especially strong on data structures:

de Berg et al. (1997): *Computational Geometry: Algorithms and Applications.*

Aside from these, there are a number of texts on algorithms that include sections on computational geometry:

Cormen et al. (1990): *Introduction to Algorithms.*
Sedgewick (1992): *Algorithms in C++.*
Rawlins (1992): *Compared to What? An Introduction to the Analysis of Algorithms.*

Textbook in other fields, most notably computer graphics, also cover aspects of computational geometry. The *Graphics Gems* series below is especially noteworthy, as it contains working C code.

Laszlo (1996): *Computational Geometry and Computer Graphics in C++.*
Rogers & Adams (1990): *Mathematical Elements for Computer Graphics.*
Foley et al. (1993): *Introduction to Computer Graphics.*
Hill (1990): *Computer Graphics.*
Glassner (I), Arvo (II), Kirk (III), Heckbert (IV) & Paeth (V) (1990–1995): *Graphics Gems: I–V.*
Mortenson (1990): *Computer Graphics Handbook: Geometry and Mathematics.*
Samet (1990): *The Design and Analysis of Spatial Data Structures.*
Farin (1993): *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide.*
Faugeras (1993): *Three-Dimensional Computer Vision: A Geometric Viewpoint.*
Kanatani (1993): *Geometric Computation for Machine Vision.*

9.3. BOOK COLLECTIONS

Two collections of surveys are now available, with a third on the way:

Du & Hwang (1992): *Computing in Euclidean Geometry* (World-Scientific).
Goodman & O'Rourke (1997): *Handbook of Discrete and Computational Geometry* (CRC Press LLC).
Sack & Urrutia (1998): *Handbook on Computational Geometry* (North-Holland).

The CRC Handbook contains fifty-two chapters covering a broad range of topics. The North-Holland *Handbook* has a narrower focus but its surveys are more in-depth.

Motion planning (Chapter 8) papers are collected in several volumes:

Hopcroft et al. (1987): *Planning, Geometry, and Complexity of Robot Motion.*
Schwartz & Yap (1987): *Advances in Robotics I: Algorithmic and Geometric Aspects of Robotics.*
Goldberg, Halperin, Latombe & Wilson (1995): *Algorithmic Foundations of Robotics.*

The following two collections pay special attention to shape and pattern recognition:

Toussaint (1985c): *Computational Geometry.*
Toussaint (1988): *Computational Morphology.*

The rich connection between discrete geometry and computational geometry is evident in several collections:

Goodman, Pollack & Steiger (1991): *Discrete and Computational Geometry: Papers from the DIMACS Special Year.*
Pach (1993): *New Trends in Discrete and Computational Geometry.*
Chazelle, Goodman & Pollack (1998): *Advances in Discrete and Computational Geometry.*

9.4. MONOGRAPHS

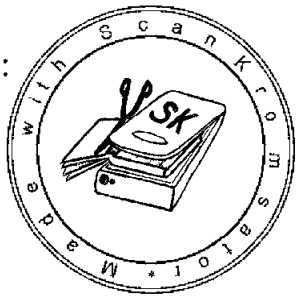
There are many monographs devoted to more specialized topics, either directly in computational geometry (such as my own book on art gallery theorems) or on related topics (such as Stolfi's book on projective geometry). A sampling follows:

Chvátal (1983): *Linear Programming.*
O'Rourke (1987): *Art Gallery Theorems and Algorithms.*
Latombe (1991): *Robot Motion Planning.*
Stolfi (1991): *Oriented Projective Geometry: A Framework for Geometric Computations.*
Okabe et al. (1992): *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.*
Sharir & Agarwal (1995): *Davenport–Schinzel Sequences and Their Geometric Applications.*
Pach & Agarwal (1995): *Combinatorial Geometry.*

9.5. JOURNALS

Two journals are devoted exclusively to computational geometry, and one to discrete and computational geometry:

Computational Geometry: Theory and Applications (Elsevier).
Computational Geometry & Applications (World Scientific).
Discrete & Computational Geometry (Springer-Verlag).



Other journals regularly publish papers on computational geometry:

SIAM Journal on Computing.
Information Processing Letters.
Journal of Algorithms.
Algorithmica.
Journal of the ACM.

About half of the 4,000 journal articles in the community bibliography are drawn from those listed above.

9.6. CONFERENCE PROCEEDINGS

Three annual conferences specialize in computational geometry. The ACM Symposium started in 1984, the Canadian Conference in 1989, and the Applied Workshop in 1996:

Proceedings of the ACM Symposium on Computational Geometry.
Proceedings of the Canadian Conference on Computational Geometry.
Proceedings of the ACM Workshop on Applied Computational Geometry.

Other conference proceedings regularly include papers in computational geometry:

Proceedings of the ACM SIGGRAPH Conference.
Proceedings of the Graph Drawing Conference.
Proceedings of the ACM-SIAM Symposium on Discrete Algorithms.
Proceedings of the IEEE Symposium on the Foundations of Computer Science.
Proceedings of the ACM Symposium on the Theory of Computing.
Proceedings of the Symposium on Theoretical Aspects of Computer Science.
Proceedings of the Workshop on Algorithms and Data Structures.

9.7. SOFTWARE

The best sources for software links are the *Directory of Computational Geometry Software* and the *Stonybrook Algorithms Repository*. The former is described in Amenta (1997); its URL is `http://www.geom.umn.edu/software/cglist/`. The latter is described in Skiena (1998); its URL is `http://www.cs.sunysb.edu/algorith/`. The code from this book is accessible through either site, as well as my own site: `http://cs.smith.edu/~orourke`.

Two pieces of software deserve special mention. `Qhull` is high-quality, robust, user-friendly code for computing the convex hull in any dimension, recently extended to construction of Voronoi diagrams via the paraboloid transformation of Section 5.7. `LEDA` (Mehlhorn & Näher 1998) is a full C++ library of computational geometry software, including an extensive class library and robust primitives.

Enjoy!