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.

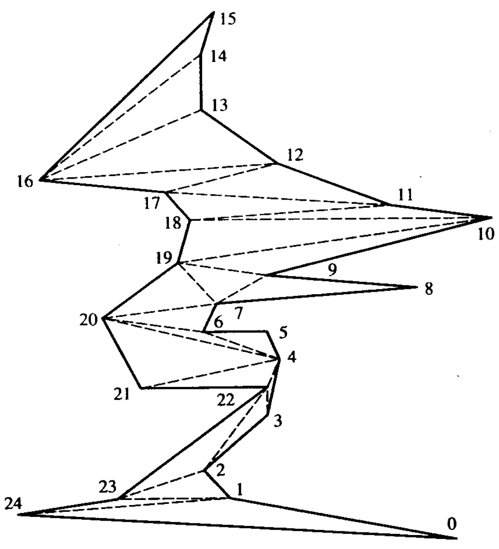

FIGURE 2.1 A polygon monotone with respect to the vertical.

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.

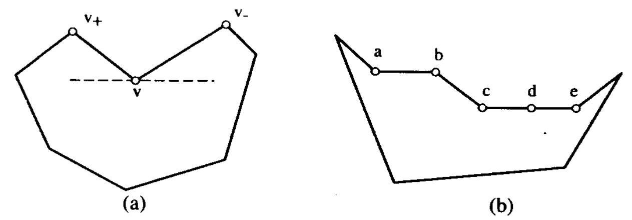

FIGURE 2.2 Interior cusps: (a) v_+ and v_- are both above v; (b) a, c, and e are interior cusps; b and d are not.

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).

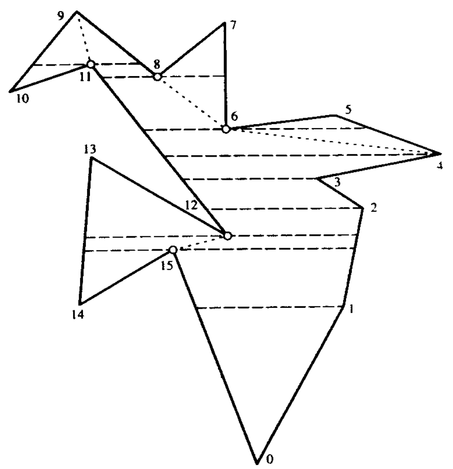

FIGURE 2.3 Trapezoidalization. Dashed lines show trapezoid partition lines; dotted diagonals resolve interior cusps (circled). The shaded polygon is one of the resulting monotone pieces.

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).

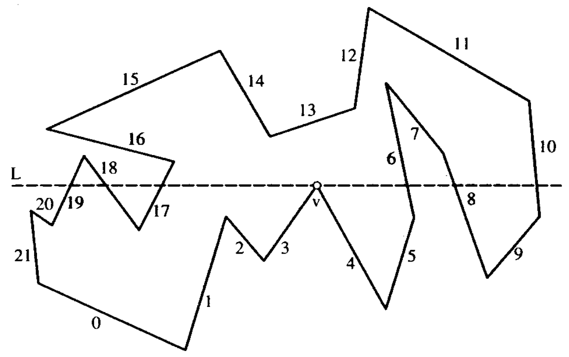

FIGURE 2.4 Plane sweep. Labels index edges.

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).

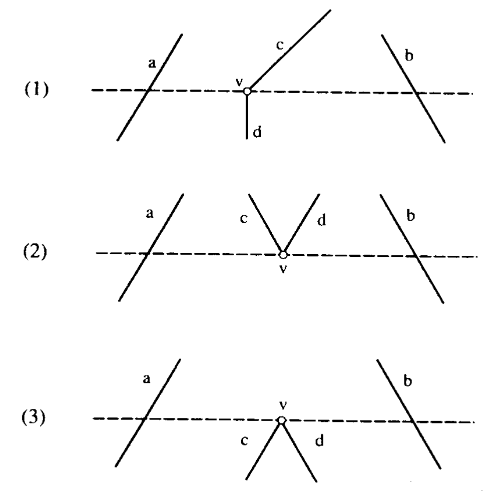

FIGURE 2.5 Sweep line events: (1) replace c by d; (2) delete c and d; (3) insert c and d.

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*.

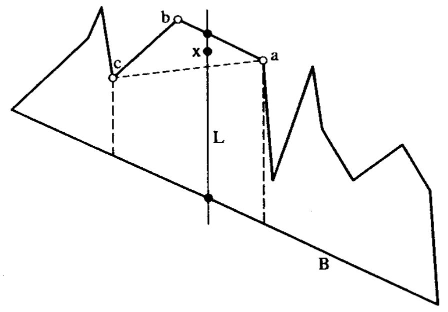

FIGURE 2.6 A monotone mountain with base B; b is an ear tip.

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 Δabc (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 Δabc. 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 Δabc. 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 $\Delta abc$.
    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.

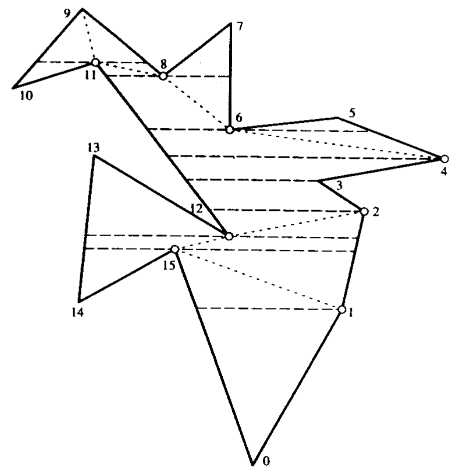

FIGURE 2.7 A partition into monotone mountains.

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.

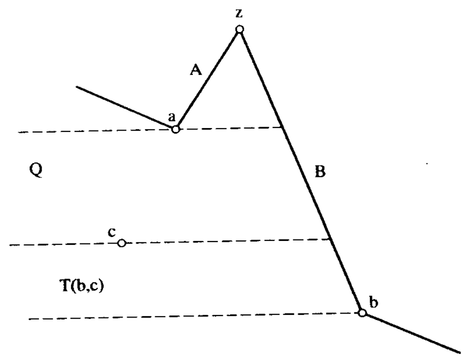

FIGURE 2.8 Proof of Lemma 2.3.2: Diagonal cb must be present.

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 ⇒ Convex quadrilaterals.* Prove or disprove: Every polygon with an even number of vertices may be partitioned by diagonals into convex quadrilaterals.
6. *Polygon ⇒ Quadrilaterals.* Prove or disprove: Every polygon with an even number of vertices may be partitioned by diagonals into quadrilaterals.
7. *Orthogonal pyramid ⇒ 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.

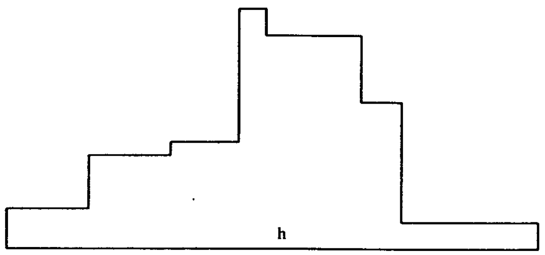

FIGURE 2.9 Orthogonal pyramid.

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 ⇒ 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

7

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.

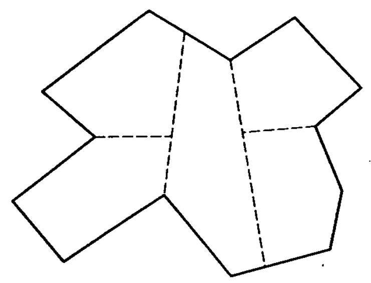
FIGURE 2.10 $r+1$ convex pieces: $r = 4$; 5 pieces.

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,* $\lceil r/2 \rceil + 1 \le \Phi \le r + 1$.

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. □

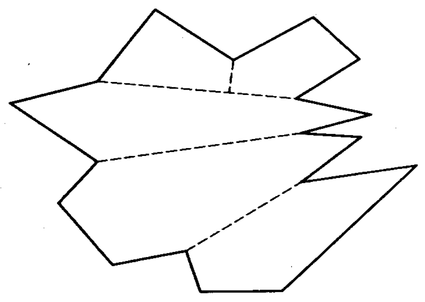
FIGURE 2.11 $\lceil r/2 \rceil + 1$ convex pieces: $r = 7$; 5 pieces.

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, $4\Phi \geq 2r + 4 > 2r + 1 \geq M$. □

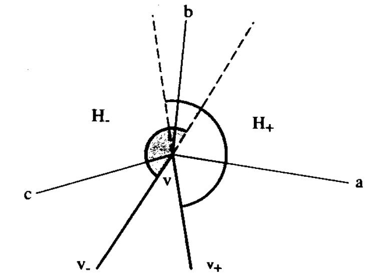

FIGURE 2.12 Essential diagonals. Diagonal a is not essential because b is also in H_+. Similarly c is not essential.

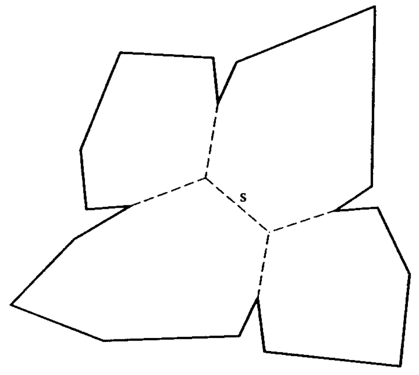

FIGURE 2.13 An optimal convex partition. Segment s does not touch ∂P.

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 $O(r^2 n \log n) = O(n^3 \log n)$ 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).