TRIANGULATION AND SHAPE-COMPLEXITY

BERNARD CHAZELLE AND JANET INCERPI
Brown University

This paper describes a new method for triangulating a simple n-sided polygon. The algorithm runs in time $O(n \log s)$, with $s \leq n$. The quantity s measures the *sinuosity* of the polygon, that is, the number of times the boundary alternates between complete spirals of *opposite* orientation. The value of s is in practice a very small constant, even for extremely winding polygons. Our algorithm is the first method whose performance is linear in the number of vertices, up to within a factor that depends only on the *shape-complexity* of the polygon. Informally, this notion of *shape-complexity* measures how *entangled* a polygon is, and is thus highly independent of the number of vertices. A practical advantage of the algorithm is that it does not require sorting or the use of any balanced tree structure. Aside from the notion of sinuosity, we are also able to characterize a large class of polygons for which the algorithm can be proven to run in $O(n \log \log n)$ time. The algorithm has been implemented, tested, and empirical evidence has confirmed its theoretical claim to efficiency.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*geometrical problems and computations*; 1.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*geometric algorithms, languages, and systems*

General Terms: Algorithms, Computational Geometry

Additional Key Words and Phrases: Divide-and-conquer, shape-complexity, triangulation

1. INTRODUCTION

Triangulation problems are many and varied [8]. One common thread between them is the attempt to refine the notion of *neighborhood* among the objects under consideration. This is quite apparent in the case of polygonal triangulations, where one is asked to augment the set of boundary adjacencies of a simple polygon by expressing it as a set of pairwise disjoint triangles. The problem is simply stated as follows:

> *Decompose a simple polygon into a set of nonoverlapping triangles without adding new vertices.*

This research was supported in part by National Science Foundation grant MCS 8303925, and by the Office of Naval Research and the Defense Research Projects Agency under Contract N00014-83-K-0146 and ARPA Order No. 4786.

Authors' address: Department of Computer Science, Brown University, Providence, RI 02912

The relevance of this question is twofold. On one hand, it is the basis of many other geometrical problems (e.g., visibility, shortest-path, region-filling problems); on the other hand, it raises one of the most puzzling questions of computational geometry: Does the knowledge of a simple path between n points allow us to break the $\Omega(n \log n)$ lower bound on the time for computing any triangulation of these points [6]? The answer to this question is not yet known, but a number of $O(n \log n)$ time algorithms have already been proposed to triangulate a simple n-sided polygon [1, 2, 4]. In this paper we propose to extend our current knowledge of the problem by describing a new algorithm which, in many aspects, outperforms all known methods.

Before presenting the main features of the algorithm, let us summarize the most important previously obtained results. Linear-time triangulations of special classes of polygons were found by Toussaint and Avis [9], and Schoone and van Leeuwen [7]. In the first case, the class contained the so-called *edge visible* and *monotone separable* polygons, and in the latter, the *star-shaped* polygons. Garey et al. [2] also gave a linear algorithm for *monotone* polygons, as well as the first general $O(n \log n)$ time method for triangulating an arbitrary simple n-sided polygon. The same time bound was achieved by Chazelle [1], using a radically different method. Since both of these algorithms involve sorting n numbers, their $O(n \log n)$ upper bound reflects the actual running time of the algorithm in a fairly accurate fashion. Recently, Hertel and Mehlhorn [4] have described a sweep-line based algorithm that performs all the better as the polygon has few reflex angles. The running time of the method is $O(n + r \log r)$, where r denotes the number of reflex angles.

Theoretically, Hertel and Mehlhorn's algorithm is interesting because it *adapts* itself to the shape of the polygon. Since beating $O(n \log n)$ in the worst case, seems very difficult, the natural trend of research has been to look for algorithms that behave linearly on a large class of polygons and require on the order of $n \log n$ operations only for fairly contrived polygonal shapes. Hertel and Mehlhorn's result takes a big step in the first direction but is unfortunately barely relevant to the second concern. Indeed, the number of reflex angles does not reflect how geometrically contrived a polygon is; to see this, just add n artificial vertices to any n-sided polygon, giving their adjacent edges an infinitesimal twist so as to create n reflex angles (of $180 + \epsilon$ degrees). This transformation will make r, the number of reflex angles, proportional to the input size, without altering at all the basic *shape* of the polygon.

In this paper we take a further step to achieve a time complexity that indeed reflects the *shape-complexity* of the polygon. We describe a triangulation algorithm that runs in time $O(n \log s)$, with $s \leq n$. The quantity s measures the *sinuosity* of the polygon, that is, the number of times the boundary alternates between complete spirals of *opposite* orientation. The value of s is, in practice, a very small constant, even for extremely winding polygons. Our algorithm is the first method whose performance is linear in the number of vertices, up to within a factor that depends only on the shape-complexity of the polygon. Informally, this notion of *shape-complexity* measures how *entangled* a polygon is, and is thus highly independent of the number of vertices. A practical advantage of the algorithm is that it does not require sorting or the use of any balanced tree structure. Aside from the notion of sinuosity, we are also able to characterize a

large class of polygons for which the algorithm can be proved to run in $O(n \log\log n)$ time. The algorithm has been implemented and tested, and empirical evidence has confirmed its theoretical claim to efficiency.

The paper is organized as follows: In Section 2 we introduce our notation and give a precise description of the algorithm. We analyze its complexity in Section 3, and in Section 4 we identify a class of polygons for which the algorithm runs in $O(n \log\log n)$ time. In Section 5 we slightly modify the algorithm and introduce the notion of *sinuosity*, by means of which we express its performance. In Section 6 we report on the actual coding of the algorithm and outline the essential components of the program. Finally, we draw various conclusions in Section 7 and indicate directions for further research.

2. DECOMPOSING A POLYGON INTO TRAPEZOIDS

2.1 The Basic Ingredients

The first and main goal of the algorithm is to compute the *vertical decomposition* of the polygon P, denoted VD(P). This is the unique partition of P obtained by drawing a vertical line through each vertex of P, extending each line as long as it does not properly cross the boundary of the polygon (Figure 1). We show later that it is straightforward to derive a triangulation of P once its vertical decomposition is available. Note that for VD(P) to be well-defined, P must be a simple, yet not necessarily closed, curve. We can extend the notion of vertical decomposition to any simple, oriented polygonal line L. Before proceeding, we introduce some terminology: Let t be a vertical segment with an endpoint p on L, and let D be a disk centered at p. If the radius is chosen small enough, we can always ensure that L subdivides D into exactly two parts, provided that p is not an endpoint of L. Let C be the part with respect to which the orientation of L runs clockwise. If the intersection of t with D lies completely in C, or if p happens to be an endpoint of L, we say that p is in *right-contact* with L.

Next, we introduce the concept of *L-extension*. For any vertex p of L, extend a vertical segment from p upward until it hits L in right-contact. Stop this process if it generates an intersection with L which is not in right-contact. Repeat the same process, substituting downward for upward. The segment generated is called the *L-extension* of p. In the simplest case, the L-extension of p has one (Figure 2 [A]) or two (Figure 2 [D]) endpoints in right-contact. It may have one (Figure 2 [C]) or two endpoints at infinity. Or, it may have one or two endpoints not in right-contact (Figure 2 [B]). In this case, the segment is stretched to infinity past the endpoint, by convention. Finally, it may be reduced to a single point (e.g., leftmost point in Figure 2), if extending a segment should generate an endpoint not in right-contact. A point of L that lies on an L-extension is called a *support-vertex* if it is a vertex of L, and a *pseudovertex* otherwise. Note that an L-extension may not have a pseudovertex, but it always has a support-vertex.

The set of all L-extensions can be conveniently represented as a set of vertical trapezoids, with each vertical side coinciding with the L-extension of some vertex of L. This set defines the *vertical decomposition* of L, denoted VD(L). If L is not closed, some of the trapezoids may be unbounded and there is no partitioning of space proper, since trapezoids may actually overlap. The set of trapezoids, however, induces a partition of the line L into line segments, called VD-edges. A

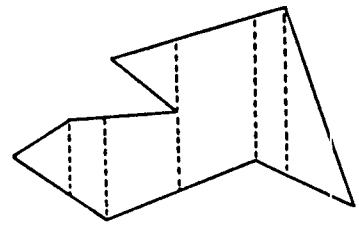


Fig. 1. The vertical decomposition of a simple polygon.

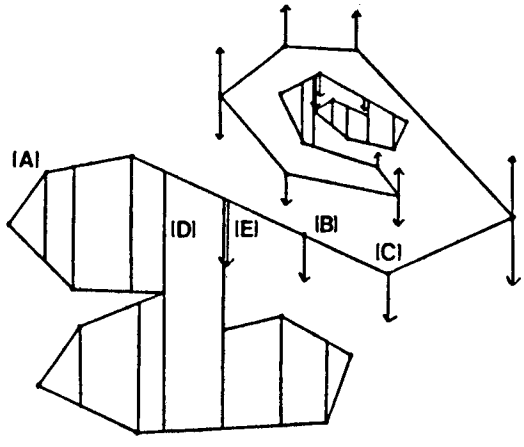


Fig. 2. The notion of L-extensions.

VD-edge is a subsegment of an edge of L that *sustains* exactly one trapezoid. Note that a bounded (respectively, unbounded) trapezoid is, in general, sustained by two (respectively, one) VD-edges. Figure 2 illustrates most of the relevant configurations of VD(L). Note that although two adjacent trapezoids often share an entire side (Figure 2 [A]), one side may be properly contained in the other (Figure 2 [D]), one of which may actually be infinite (Figure 2 [E]).

The data structure for VD(L) provides two accessing schemes. On the one hand, each trapezoid is represented by its boundary, kept as a doubly linked list, with adjacent trapezoids pointing to each other. On the other hand, the line L is represented as a doubly linked list of VD-edges, each of them pointing to its sustained trapezoid. It may sometimes happen that the vertical side of a trapezoid contains several support vertices. We resolve these singularities by introducing extra trapezoids of null width. This will allow us to assume that each trapezoid side contains exactly one support vertex.

2.2. Merging two VD-structures

The key step of the divide-and-conquer algorithm involves merging two structures VD(L_1) and VD(L_2), with $L_1 = \{v_m, \ldots, v_1\}$, $L_2 = \{w_1, \ldots, w_m\}$, and $v_1 = w_1$. These lists indicate the orientation of the polygonal lines. For consistency, we use a descending sequence for L_1 so that the merge can proceed toward higher indices for both L_1 and L_2. In the following, we say that a point $a \in L_1$ is L_1-*further* than a point $b \in L_1$, if it is encountered after b in the course of traversing L_1 from v_1 to v_m. The same applies about L_2. Let $L = \{v_m, \ldots, v_1, w_2, \ldots, w_m\}$. The goal is to compute VD(L) by proceeding concurrently from v_1 to v_m and w_1

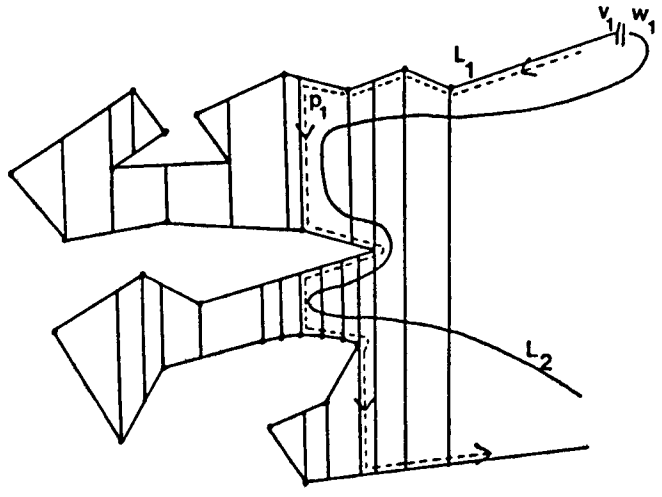


Fig. 3. Taking shortcuts.

to w_m, reconfiguring the trapezoids on-line. The main feature of this *stitching* operation is to take *shortcuts* whenever possible, that is, avoid traversing vertices whose adjacent trapezoids remain unchanged. This concept of shortcuts, extensively developed later on, is essential to the algorithm. Without it, any hope of beating $O(n \log n)$ time would vanish. The computation will be completely guided by two pointers, p_1 and p_2, running concurrently through the VD-edges of L_1 and L_2, respectively. The computation will ensure the following invariant:

INVARIANT. *At any time during the merge of L_1 and L_2, the current state of $VD(L)$ consists of the L-extension of all the points of L_1 (respectively, L_2) between v_1 and p_1 (respectively, p_2) as well as the L_1- (respectively, L_2-) extension of all the other points in L_1 (respectively, L_2).*

Figure 3 gives an example of a typical tour of p_1. For simplicity, we have represented L_2 as a curved line to give a rough indication of its interaction with L_1. The trajectory of p_1 appears as a dotted line. Figure 3 shows that p_1 will point to VD-edges of trapezoids in $\mathrm{VD}(L_1)$, moving only between adjacent trapezoids. Occasionally, p_1 will switch from one VD-edge of a trapezoid to another of the same trapezoid (recall that a trapezoid has one or two VD-edges). As we shall see shortly, VD-structures contain all the information necessary to implement the merging without too much effort. The only difficulty resides in identifying all possible situations and treating them in a unifying manner.

At the generic step, all the VD-edges of L_1 (respectively, L_2) between v_1 and p_1 (respectively, p_2) have been processed and will *never* be visited again during the merging of L_1 and L_2. If at time θ the last trapezoid t, examined in $\mathrm{VD}(L_2)$, has been entirely processed, we must consider the "next one" in $\mathrm{VD}(L_2)$. This new trapezoid is referred to as the *fresh* trapezoid at time θ. More need be said about this trapezoid to make it defined unambiguously; all we can now say is that since it is adjacent to t, it can be retrieved in constant time from $\mathrm{VD}(L_2)$. In general, if the fresh trapezoid is provided by L_2, the last trapezoid, t', examined in $\mathrm{VD}(L_1)$,

will have been only partly updated at time θ, and further processing will be in order; t' is called the *carry* trapezoid at time θ. Typically, t' is the remaining part of a clipped trapezoid of VD(L_1).

We are now ready to merge the fresh and carry trapezoids, and thus add to the construction of VD(L). We proceed by a case-analysis, which although a bit lengthy, is fairly straightforward. Before describing the algorithm, let us mention some of the difficulties we shall encounter. This preview, intended only as an illustration, will prepare the reader for the forthcoming case-analysis. One important feature, which we have to respect carefully, is the presence of exactly one support-vertex on each vertical side of each trapezoid. As we have seen earlier, the multiplicity of such vertices can be easily corrected by the introduction of null-width trapezoids. Another problem is the handling of newly appeared vertical sides with no support-vertex at all. We solve this problem by combining the two trapezoids adjacent to the side in question. Note also that since a trapezoid often has two sustaining VD-edges, its updating in the course of processing one of these edges may have the effect of upsetting the sustaining status of the other edge; hence, updating with respect to both VD-edges will be in order. This is required in order to satisfy the invariant. Note that one likely effect of this updating is to modify the breakdown of L_1 or L_2 into VD-edges. We shall illustrate this type of modification later.

By construction, the carry and fresh trapezoids lie on the same side—say, without loss of generality (wlog) on the left side—of the last trapezoid constructed. As far as the construction of the next trapezoid is concerned, the difference between a carry and a fresh trapezoid is immaterial. For this reason, we can assume wlog that p_1 stretches further to the left than p_2, that is, the x-coordinate of p_1's left endpoint does not exceed the x-coordinate of p_2's left endpoint.

The algorithm rests on a case-analysis that can be best described pictorially. Figures 4 illustrates all possible cases. Each of the six pairs of pictures depicts the construction of the next trapezoid(s), represented in hashed lines in the right-hand column. Each of the 12 pictures contains an arrow to indicate the direction from which the next construction will take place. We represent three kinds of trapezoids: *fresh*(F) and *carry*(C), as well as *next*(N), one of the trapezoids of VD(L_2) adjacent to *fresh*. We shall define *next* more precisely in each of the cases. The trapezoids are represented very schematically. Each of them has a VD-(lower or upper) edge currently under consideration; this edge is represented by a continuous line, whereas the other (upper or lower) edge is represented by a dotted line. The latter edge is either sustaining or at infinity; in either case, its position has no particular significance, unless the line is doubled up with a continuous line (Figures 4b, e). In this case, the edge is sustaining and its relative position is meaningful. The case-analysis is based on the relative position of p_2 (= ab) and bc, the VD-edge adjacent to p_2 in the direction of the construction. Figures 4 (a–c) (respectively, d–f)) assume that c (respectively, b) has the least x-coordinate among a, b, c. Here are a few remarks intended to supplement Figure 4.

(1) (Figures 4a, b) Let q be the unique support-vertex of F. From the Jordan Curve theorem, it follows that either $q = b$, or q is strictly L_2-further than b.

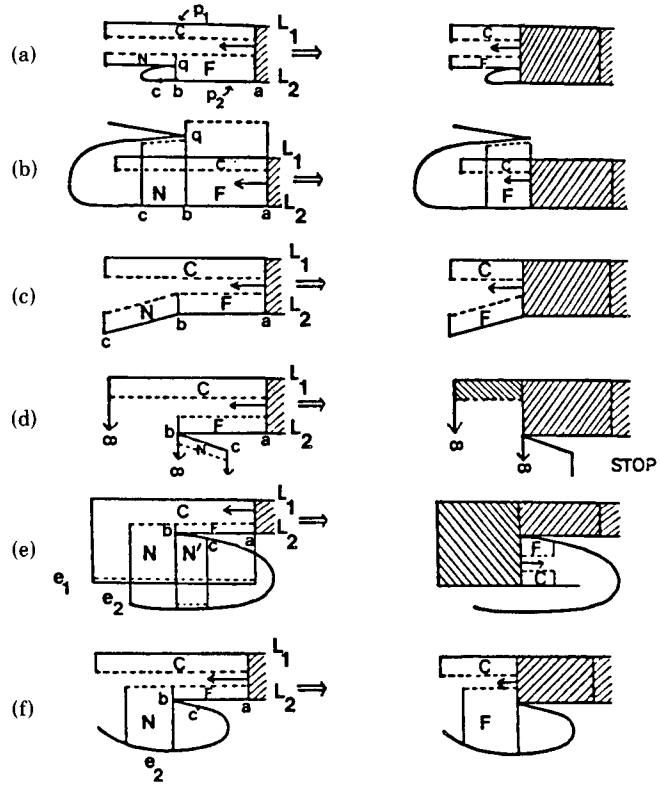


Fig. 4. The case-analysis.

Figures 4a and b, assume the latter case. In the direction of the construction, there are two candidates for the role of next trapezoid from L_2. Discriminating between Figure 4a and b settles this question. In 4a (respectively, 4b), the support-vertex lies below (respectively, above) p_1.

(2) (Figures 4c, d–f) In all these cases, $q = b$, that is, b is the support-vertex of the left side of F. Case 4c is the simplest and, as such, fairly self-explanatory. In 4d, the left sides of both C and N extend to infinity downward, and therefore, no further interaction between L_1 and L_2 is to be expected; the merging is over. If this is not the case, let e_1 (respectively, e_2) be the lower edge of C (respectively, N). Note that one of these edges may be at infinity. Draw a segment from b downward, until it first intersects either e_1 (4e) or e_2 (4f). In the former case, let N' be the trapezoid of VD(L_2) sustained by bc. We can process directly all of C and N, declare N' fresh, and keep C as the carry. Two observations are in order: first of all, although C is still a carry trapezoid, its sustaining edge that gives its value to p_1 will be switched from the upper to the lower edge. Also, it may be the case that N is not properly defined as indicated in Figure 4e. This will happen when the trapezoid of VD(L_2) sustained by bc is unbounded below. We then blithely proceed toward N', after applying the treatment depicted in Figure 4e. Finally, in case 4f, we proceed as shown in the corresponding figure, observing that the treatment is somewhat fairly similar to 4a and c.

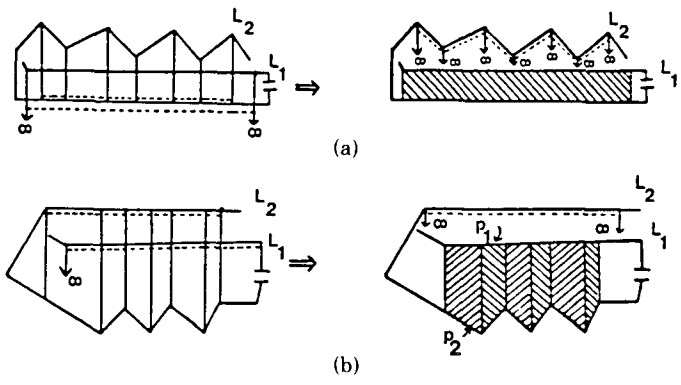


Fig. 5. Handling side-effects.

We must now deal with the various side effects mentioned earlier. When introducing a new trapezoid, we must always check whether it can be combined with the previous one so as to form a single trapezoid, in which case, the trapezoids in question should be combined. In Figure 5a, for example, VD(L_1) provides the same carry for several steps, whereas VD(L_2) keeps on supplying fresh trapezoids with no support-vertex on L_2. As a result, the newly created trapezoids will be combined, one after the other. To ensure the invariant, we must also update the remaining part of VD(L_2). In the same vein, Figure 5b illustrates the notion of *side-effect* updating: the creation of new trapezoids in VD(L) resets the lower edges of a number of trapezoids in VD(L_2) to $-\infty$, thus causing the loss of their support-vertices. As a result, we must merge all these trapezoids into a single one. Naturally, in the course of merging L_1 and L_2, we must set all the adjacencies between newly created trapezoids, as prescribed in the definition of VD(L).

The proof of correctness relies, upon the inspection, that Figure 4 considers all significant cases, and that the actions taken respect the invariant. The latter can be asserted by observing that (1) the new trapezoids are valid trapezoids of VD(L), and (2) each trapezoid of VD(L_1) or VD(L_2) that is modified by the computation leads to an updating of the adjacencies of all its sustaining edges. Note that the algorithm is conceptually quite simple, and the difficulty resides only in specifying the details correctly. For this reason, our successful implementation of the algorithm is, we believe, an indispensable part of this work, as it contains all the details left unmentioned here for the sake of clarity.

2.3. The Divide-and-Conquer Procedure

We are now ready to compute the vertical decomposition of the polygon P. Let $\{p_1, \ldots, p_n\}$ be the vertices of P in clockwise order. Recursively, compute VD(L_1) and VD(L_2), for $L_1 = \{p_1, \ldots, p_{\lfloor n/2 \rfloor}\}$ and $L_2 = \{p_{\lfloor n/2 \rfloor + 1}, \ldots, p_n\}$, and finally apply the procedure described above to merge VD(L_1) and VD(L_2). This will produce the vertical decomposition of P, which completes the description of the algorithm. Here is a simple, preliminary result on the complexity of the algorithm.

THEOREM 1. *The algorithm computes the vertical decomposition of a simple n-sided polygon in $O(n \log n)$ time and $O(n)$ space.*

PROOF. Since the L-extension of a vertex of L consists of at most three points, VD(L) requires $O(|L|)$ storage. Merging VD(L_1) and VD(L_2) takes time linear in the input size; therefore the divide-and-conquer procedure requires $O(n \log n)$ time. □

We shall see later on that the algorithm has a much better performance than indicated by Theorem 1. To seek any improvement, however, it is crucial to ensure two requirements regarding the implementation.

(1) If the algorithm is implemented recursively, we should pass pointers to the endpoints of L_1 and L_2 as arguments to the merge routine VD and not to the full-fledged lists themselves.
(2) We should compute VD(L) *in place* by modifying existing links, allocating new space only when necessary and freeing space after deletions.

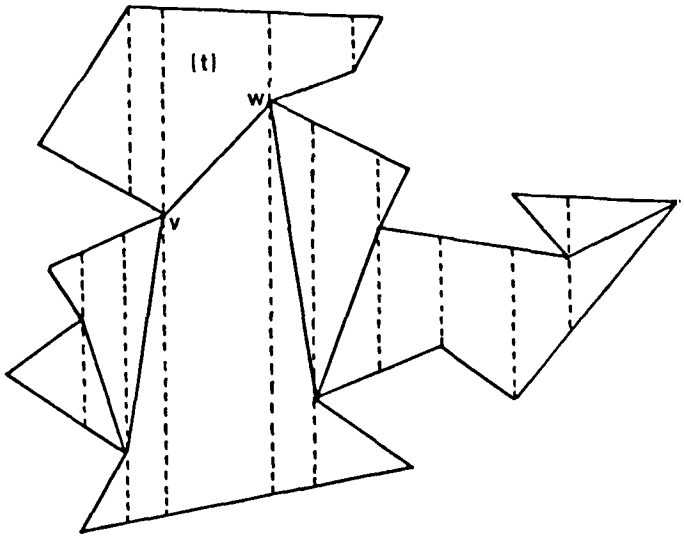
3. THE TRIANGULATION ALGORITHM

We shall show how to transform the vertical decomposition of P into a triangulation, using a variant of the algorithm of Garey et al. [2]. This algorithm involves computing a *regular decomposition* of P and applying to it a simple $O(n)$ time procedure to produce a triangulation. A regular decomposition of a polygon P is a partition of P into *monotone* polygons that does not introduce new vertices. A polygon Q is said to be *monotone* if there exists a line T such that the boundary of Q can be decomposed into two consecutive chains, each of whose orthogonal projections on T has the same vertex-list order as its originating chain.

It is quite simple to go from a vertical to a regular decomposition of P. To do so, we choose the x-axis as the reference line T and identify all the vertices of P that cause the polygon not to be monotone. These are exactly the support-vertices whose P-extension contains two pseudovertices. For each such vertex v, let t be the unique trapezoid for which the P-extension of v coincides with a vertical side, and let w be the other support-vertex of t. It is easy to show that connecting each pair (v, w) in P by a straight edge will produce a regular decomposition of P (Figure 6). We omit the proof of this elementary fact. The description of the triangulation algorithm is now complete. Its time complexity is clearly $O(n \log n)$, since transforming a vertical decomposition into a triangulation requires linear time.

4. INTRODUCING THE NOTION OF NESTING NUMBER

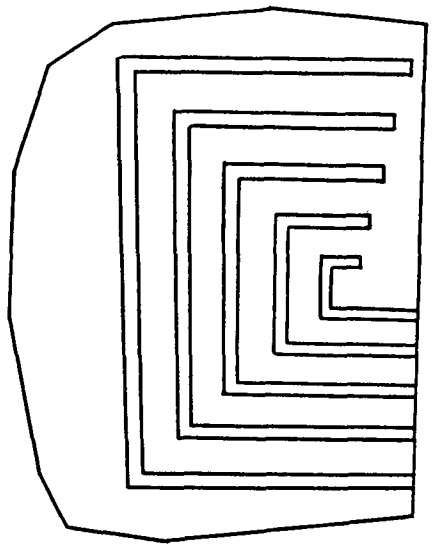
Since the complexity of the triangulation algorithm is entirely dominated by the construction of the vertical decomposition, we shall concentrate exclusively on this phase of the algorithm. A quick observation shows that the upper bound of $O(n \log n)$ on the time complexity of the algorithm is tight (Figure 7a). The shapes of the worst-case polygons are so contrived, however, that we can argue that in most practical cases, the algorithm is *extremely* efficient—actually more efficient than any other triangulation algorithm known to date. This feature is due to the fact that the algorithm is highly adaptive. Indeed, by constantly seeking shortcuts, the algorithm may often merge two large inner decompositions in constant or near-constant time. In Figure 7b, for example, all the merges take constant time, except for the last one, which requires linear time. The overall running time is therefore $O(n)$. In this section we try to characterize classes of polygons for which the running time is linear or quasi-linear.



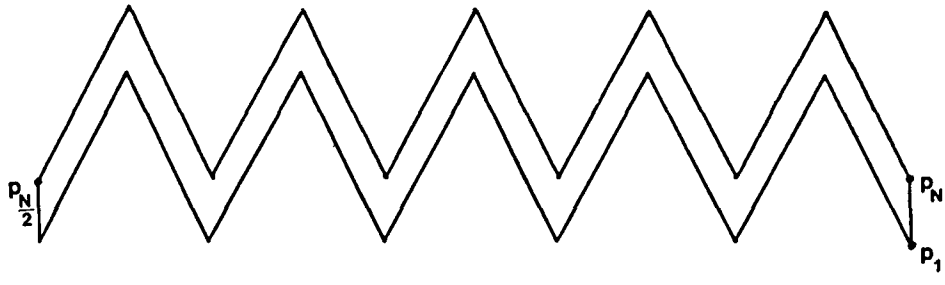
To refine the crude analysis of Theorem 1, we introduce the concept of *nesting number*. A vertical line D that intersects P has the effect of dividing the boundary of P into polygonal chains running from one point of D to another, each lying entirely on one side of D. Consider now the polygonal lines as fences and imagine that a dog placed initially on the line D wishes to run away to infinity, staying always on one side of D. Let m_1 (respectively m_2) be the minimum number of fences it will have to jump in order to escape through the left (respectively, right) side, and let $m = \max(m_1, m_2)$. We define $\nu(P)$, the *nesting number* of P, as the maximum value of m for all starting positions and all lines D. In Figure 8, for example, $\nu(P) = 3$.

Practical experience (with, say, graphic typesetting or pattern recognition, where the need for efficient triangulations arises) shows that the nesting number of a polygon is, in general, 1 and usually less than or equal to 2 or 3. Let us assume in the following that $\nu(P) < 4$; we shall show that all the n-sided polygons in this class can be triangulated in $O(n \log\log n)$ time.

It is clear from the description of the algorithm that the running time is proportional to the number of times L-extensions are updated. Pseudovertices are never updated, properly speaking; they are created and possibly destroyed. Consequently, the costs that they incur can be charged to the corresponding support-vertices. Without loss of generality, we can concentrate on the L-extensions that have a pseudovertex lower than their support-vertex along the y-axis. These extensions are called *top-extensions*, and their lower endpoints (which are pseudovertices) are called the *low-points* of the extensions. Since a low-point can never be set to infinity twice in a row, we can restrict our attention to the number of times a low-point is set to a new position on the boundary of P. These updates are called *breaks*. Let p_i be the support-vertex of some top-extension. Note that, chronologically, breaks occur closer and closer to p_i. The



(a)



(b)

Fig. 7. Bad and good cases.

vertex p_i is said to be *right-broken* (respectively, *left-broken*) if the break is caused by an edge p_jp_{j+1} with $j > i$ (respectively, $j < i$). Let $C(n)$ be an upper bound on the total number of times a low-point is updated in the course of computing VD(P). For reasons of symmetry, we immediately derive that the time necessary to compute VD(P) is $O(n + C(n))$. In the following analysis we therefore consider only left-breaks of top-extensions without further justification. We also use the term left-break with the implicit understanding that it refers to top-extensions.

Let us examine how left-breaks can occur in the course of merging the vertical decomposition of two polygonal lines L_1 and L_2. Let $L_1 = \{q_1, \ldots, q_m\}$ and $L_2 = \{q_{m+1}, \ldots, q_{2m}\}$, with $\{q_1, q_2, \ldots\} = \{p_l, p_{l+1}, \ldots\}$ for some l. For any $k > 2$, let $D(k)$ be the number of vertices in L_2 that satisfy the three following conditions: (1) they are left-broken in the course of merging L_1 and L_2 (this can happen only once per merge), (2) they were previously left-broken exactly $k - 1$ times, and

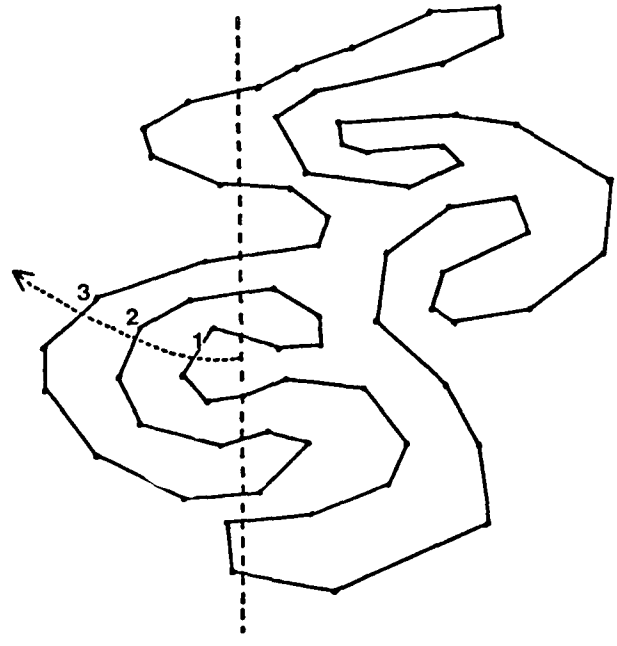


Fig. 8. The notion of nesting number: $\nu(P) = 3$.

(3) they will be left-broken later on at least once more. We prove the following result:

LEMMA 1. *For any* $k \leq 2$, *we have* $D(k) \leq m/2^{k-3}$.

PROOF. Let q_i be the vertex of L_2 that satisfies the three conditions above and has the highest index i, and let B be the low point of the new top-extension of q_i right after the merge. Let $\{q_j, \ldots, q_i, \ldots, q_{i'}\}$ be the polygonal line resulting from the merge during which q_i was left-broken for the first time. Let A be the low-point of the top-extension of q_i right after its first left-break, and let $q_\nu q_{\nu+1}$ be the supporting edge in L_2 (Figure 9a). We will show that no vertex between q_m and q_j can contribute any value to $D(k)$. To do so, we will prove the stronger result that no vertex between q_m and q_j can be actually left-broken for at least the second time, while merging L_1 and L_2, with the hope of it being left-broken later on. Assume that this is not the case, and that $q_h (m \leq h \leq j)$ is left-broken by the edge $q_u q_{u+1}$ of L_1. We distinguish between two cases:

(1) q_h lies inside the polygon enclosed by $\{q_i, A, q_{\nu+1}, q_{\nu+2}, \ldots, q_i\}$. In this case, in addition to $q_u q_{u+1}$, each of the following polygonal lines contributes one to the nesting number: (1) from A to q_i; (2) from q_h to the low-point C of the previous top-extension of q_h; (3) as a direct application of the Jordan Curve theorem, a third polygonal line between (1) and (2) must also exist and increase q_h's contribution to the nesting number. This raises the nesting number to at least four, and brings a contradiction (Figure 9a).

(2) If q_h does not lie inside the polygon enclosed by $\{q_i, A, q_{\nu+1}, q_{\nu+2}, \ldots, q_i\}$ (Figure 9b), it will be impossible to left-break q_h later on without bringing two

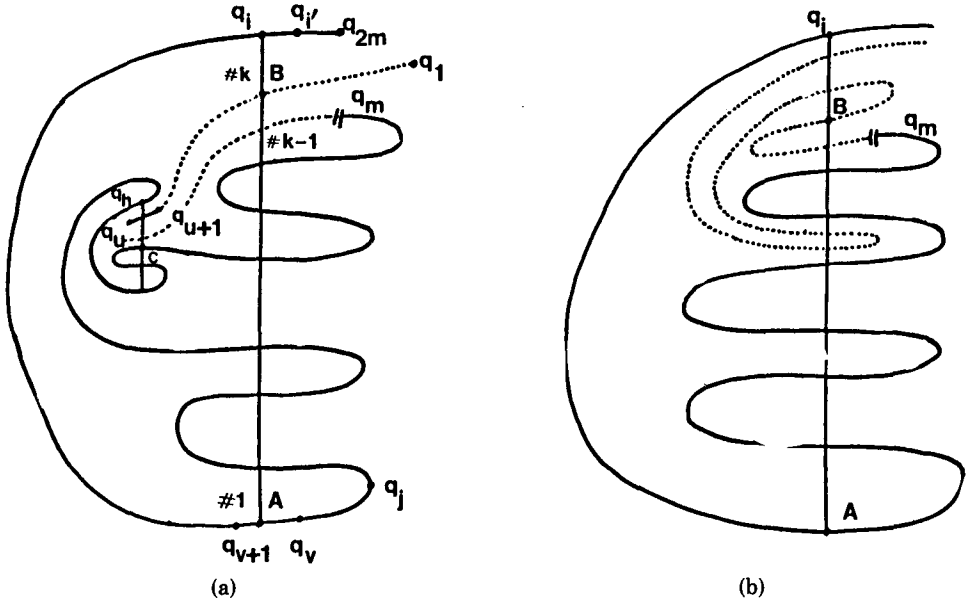


Fig. 9. Counting left-breaks.

polygonal lines across Aq_i, once to the left then to the right, thus bringing up the nesting number to a prohibitive value ≥ 4.

It follows from this result that, since by definition of q_i no vertex between q_{i+1} and q_{2m} can satisfy the three conditions stated above, $D(k)$ is dominated by the number of vertices between q_j and q_i. Let $L_1^{(l)}$ be the L_1-argument passed to the merge routine at the time q_i was left-broken for the lth time. The divide-and-conquer algorithm prescribes that $L_1^{(l)}$ contain at least twice as many vertices as $L_1^{(l-1)}$; therefore

$$i - j \leq \frac{1}{2^{k-3}} \mid L_1^{(k-1)} \mid \leq \frac{m}{2^{k-3}},$$

which completes the proof. □

THEOREM 2. *The triangulation algorithm runs in* $O(n \log\log n)$ *time for any n-sided polygon with nesting number less than 4.*

PROOF. Since converting a vertical decomposition of a polygon into a triangulation takes linear time, it suffices to show that VD(P) can be computed in $O(n \log\log n)$ time. We give a fairly intuitive counting argument. Let T denote the computation tree. T is a complete binary tree over n leaves, with the following trivial interpretation: a node v at level k represents the merge of two polygonal lines of the form $L_1 = \{q_i, \ldots, q_{i+2^{k-1}}\}$ and $L_2 = \{q_{i+2^{k-1}}, \ldots, q_{i+2^k}\}$. Each level of T thus represents a *stage* of the computation. We shall evaluate the maximum number of left-breaks by playing a pebbling game. For obvious reasons, we may restrict our attention to the subset V consisting of vertices that are left-broken at least three times over the entire computation. This will allow us to apply the result of Lemma 1. We set up the following charging policy: with each vertex of

V is associated a distinct pebble, and, at any stage of the computation, we require that if v has been left-broken k times so far, its pebble should be placed at level k in the tree (by convention, leaves are at level 1). We can thus evaluate the number of left-breaks incurred so far at any given stage by adding up the heights of all pebbles.

At any stage t, we update the tree by moving up the pebble of each vertex that is left-broken at t. Let $\mathscr{L}_k$ designate level $\#k$ in the tree. Lemma 1 shows that, at any stage up to within constant factors, any level $\mathscr{L}_k$ cannot receive more than $c = n/2^{k-3}$ pebbles. We observe that c represents the number of nodes at level $\mathscr{L}_{k-2}$, which is four times the number of nodes at level $\mathscr{L}_k$. Consequently, at the end of the computation, no level will have $4(\lceil \log_2 n \rceil + 1)$ times more pebbles than nodes, since there is a total of $h = \lceil \log_2 n \rceil + 1$ stages. It is clear that the added height of all pebbles H is maximized when all the upper levels are filled to their maximum capacity. This means that each level $\mathscr{L}_i$ should be given $4h \times \text{size}(\mathscr{L}_i)$ nodes, for $i = h, h-1, \ldots, h-k$. The limit k is determined by the fact that only n pebbles are available; therefore, $\sum_{0 \le i \le k} 2^i(4h)$, that is, $2^{k+1} \le 1 + n/4h$. We thus have

$$H = 4h \times \sum_{0 \le i \le k} 2^i(h-i) = 4h((h-k+1)2^{k+1} - h - 2);$$

hence $H = O(n \log\log n)$. This implies that $C(n) = O(n \log\log n)$, which completes the proof. □

5. PRECONDITIONING THE INPUT

As is often the case with adaptive algorithms, preprocessing the input prior to calling the main algorithm might often lead to substantial gains. One advantage of the algorithm is that it is based on polygonal lines rather than polygons. This feature allows us to break down the polygon into a (preferably small) number of polygonal lines, the vertical decompositions of which we compute separately and merge together in a final stage. We take advantage of the fact that the algorithm is particularly efficient on spiral-shaped inputs.

Let $L = \{q_1, \ldots, q_m\}$ be a simple polygonal line. Consider the motion of the straight line passing through $q_i q_{i+1}$, as i goes from 1 to $m-1$. Every time the line reaches the vertical position clockwise (respectively, counterclockwise), we increment (respectively, decrement) a *winding counter* by one (the initial value of the counter is irrelevant). We will say that L is *spiraling* (respectively *antispiraling*) if the winding counter is never decremented (respectively, incremented) twice in a row.

It is straightforward to decompose any simple polygon into spiraling and antispiraling polygonal lines. This can be done in a single linear pass, and we may omit the details. Note that we should restart a new polygonal line only when the previous line ceases to be spiraling or antispiraling (Figure 10). At that point, we start scanning a new polygonal line and continue the traversal as long as the line can be classified as either spiraling or antispiraling. In general, the line will be of both types at the start and then fall into one type. We define s, the *sinuosity* of P, as the number of polygonal lines thus obtained. Note that, depending on the starting vertex, this number may vary by one. In the following, s will refer to

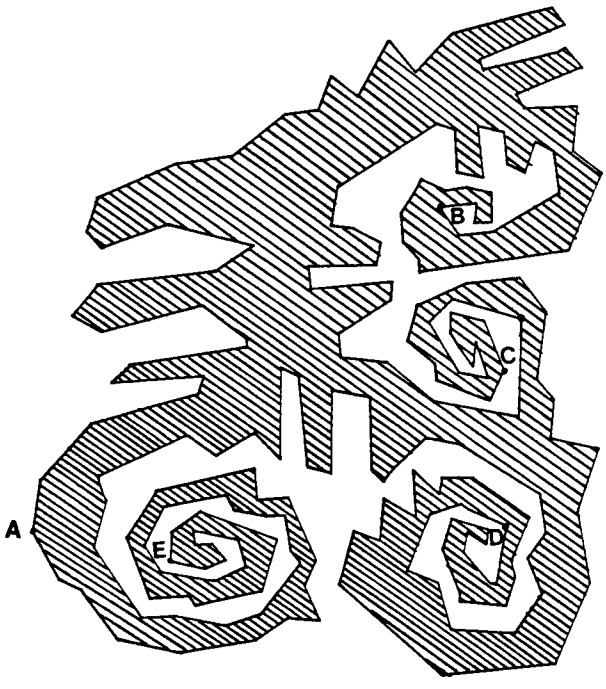


Fig. 10. Preconditioning the input.

the larger of these numbers. Empirical evidence shows that in practice s is a small constant. Even a fairly contrived polygon like the one depicted in Figure 10 has a sinuosity of only 5. We shall show successively that the vertical decomposition of a spiraling or antispiraling line can be computed in linear time, and that a simple n-sided polygon P can be triangulated in time $O(n \log s)$.

LEMMA 2. *The vertical decomposition of any spiraling or antispiraling polygonal line can be computed in linear time.*

PROOF. Without loss of generality, assume that the polygonal line, which we denote L, is spiraling. We first examine how left-breaks can occur in the course of merging the vertical decomposition of two polygonal lines L_1 and L_2. Once again, we restrict our attention to left-breaks related to top-extensions. Without loss of generality, we may assume that L_2 contains at least one vertex q that is left-broken during the merging of L_1 and L_2 by an edge e, and that this is neither the first time nor the last time that this vertex is left-broken. Let w be the value of the winding counter at q (also referred to as the *winding value* of q). Since we restrict ourselves to top-extensions, we consider only the left-breaks of vertices whose winding value has the same parity as w. Note that any edge causing a left-break must have a winding value equal to the winding value of the broken vertex minus one. Since q is left-broken by edges outside of L_1 both following and preceding e, we derive immediately that $w - 1$ and $w - 2$ are the only possible winding values in L_1, and therefore L_1 is totally free of left-breaks. This allows

us to charge to L_1 the left-breaks incurred while merging L_1 and L_2. Since L_1 and L_2 are of the same size, dividing the costs evenly between the vertices of L_1 will amount to charging each vertex a cost of at most one. Note that since L_1 is free of left-breaks, none of its vertices will have been charged in this manner up to this point, and since L is not free of left-breaks, no vertex of L_1 will ever be charged in this way hereafter. We conclude the existence of at most $O(n)$ left-breaks during the computation of VD(P). Observing that a similar reasoning applied to right-breaks would lead to the same findings, the proof is now complete. □

We are now ready to prove the main result of this section.

THEOREM 3. *It is possible to triangulate a simple n-sided polygon P in $O(n \log s)$ time and $O(n)$ space.*

PROOF. It suffices to show that VD(P) can be computed in $O(n \log s)$ time. As mentioned earlier, we will first decompose the boundary of P into $k = s$ (or perhaps $s - 1$) spiraling and/or antispiraling polygonal lines, which will take $O(n)$ time, and then compute the vertical decomposition of each of them. Since merging two vertical decompositions takes time at most proportional in the total number of vertices involved, we now face a problem similar to merging k sorted lists together. Let $w_1, \ldots, w_k$ be the sizes of the k polygonal lines that partition the boundary of P. We merge these lists according to the order given by an optimal alphabetic binary tree constructed by the Hu–Tucker algorithm. Computing the optimal tree requires $O(k \log k)$ time and $O(k)$ space, which leads to an added merge time of $O(\sum_{1 \le i \le k} w_i \log(n/w_i))$ [5]. By elementary calculus, this quantity is shown to be $O(n \log k)$ time, which completes the proof. □

6. CODING UP THE ALGORITHM

We mention a few details concerning the implementation of the algorithm. The main emphasis of this section is to demonstrate how the computation is distributed hierarchically among the various parts of the algorithm, and, in particular, what layers lie on top of the inner loop depicted in Fig. 4. We also briefly describe the basic data structures.

The algorithm has been implemented in the C language. Polygon vertices, supplied in clockwise order, are stored in a doubly linked list so as to handle both clockwise and counterclockwise traversals. Along with pointers to the next and previous vertices on the boundary, we have, associated with each vertex of the polygon, a pointer to some trapezoid, for which it is either the top-left or bottom-right corner. In general, each edge of the polygon has a few trapezoids hanging from it, and the vertex v_i has a pointer to the first trapezoid hanging from the edge v_iv_{i+1} when traversed in clockwise order. The vertical trapezoids are stored by keeping the minimum and maximum x-values, along with pointers to the polygonal edges that support the trapezoid. These pointers lead directly into the polygon's doubly linked list. Unbounded trapezoids have *null* pointers. Thus, a trapezoid is defined using two x-values and two pointers to vertices. In order to walk around VD(L), pointers to adjacent trapezoids are also needed. As men-

tioned earlier, we assume that each side of a trapezoid contains exactly one support-vertex; therefore, at most two trapezoids can lie on each side of a given trapezoid. Thus, associated with each trapezoid are four pointers, two for each side.

Initially, each edge has an infinite trapezoid associated with it. The main procedure controls the divide-and-conquer. It sets pointers to the heads of the polygonal lists L_1 and L_2. These pointers are then passed to a *merge* routine. *Merge* is responsible for maintaining the pointers p_1 and p_2. In the implementation, however, p_1 and p_2 point to actual edges of the polygon and not VD-edges. With an edge is associated a direction, clockwise or counterclockwise, depending on its supporting polygonal line, as well as a position, *above* or *below*, to indicate whether the trapezoids supported by the edge hang above or below the edge. Note that this information is needed, since it is not readily available from the trapezoids themselves. The *merge* routine operates the merging of VD(L_1) and VD(L_2). It deals with actual edges of L_1 and L_2. The routine finds the starting positions of p_1 and p_2 (which are not necessarily the first edges of L_1 and L_2). *Merge* determines shortcuts; its inner loop calls a subroutine, *mergeVD*, which carries out the actual merging of trapezoids. When *mergeVD* returns, the basic invariant is satisfied up to that point, and one of the edges, say p_1, will never be considered again. *Merge* will then find the next trapezoid and its supporting edge, and update p_1. While *merge* essentially controls the case analysis, *mergeVD* computes $VD(L)$. The routine creates new trapezoids whenever necessary, modifies links, and operates the garbage collection. It also handles side effects, that is, combining trapezoids whose side edges have no support-vertex. For this reason, *mergeVD* does not simply combine the fresh and carry trapezoids but rather combines all the trapezoids along one edge. The inner loop implements the case-analysis of Figure 4. To handle the case of side-effect updating shown in Figure 5b, the bottom of each *spurious* trapezoid is set to infinity and the trapezoids are merged when the upper edge of L_2 is passed to *mergeVD*.

The length of the program is approximately 700 lines for *merge* and 800 lines for *mergeVD*. The rest of the code includes initialization and I/O operations, and amounts to an insignificant percentage of the total length. Debugging the algorithm was greatly facilitated by using SGP, a graphics system modeled on ACM/SIGGRAPH's CORE79 and ISO's GKS, on both a Lexidata 3400 and a Ramtek 9400. We still had to devote substantial effort to the design of the data structure in order to make the implementation of the case-analysis simply manageable. One of the crucial requirements was to provide the structures with enough scope to treat special cases uniformly. With the intricate geometric setting defined by the algorithm, such an approach was not only desirable, but absolutely mandatory. Following Guibas and Stolfi's methodological approach to complex geometric implementations [3], we have attempted to define a small set of powerful primitives in order to separate the combinatorial aspects of the problem from its purely geometrical ones. This separation is embedded in the data structure by distinguishing geometric data types, which lie at the bottom of the hierarchical structure of the computation, and combinatorial types, which constitute the backbone of the algorithm.

7. CONCLUSIONS

We have presented a new triangulation algorithm for simple polygons that contains a number of attractive features, both of theoretical and practical nature. On the practical side, our algorithm has been implemented and has shown to be remarkably efficient. In graphics, a common application of polygon triangulation is region filling, whereby a given area is to be *painted.* With modern raster graphics devices, painting a vertical trapezoid can often be accomplished just as fast as painting a triangle, so we may use the vertical decomposition algorithm directly without resorting to the triangulation, which is bound to save substantial CPU time.

On the theoretical side, we have established that the algorithm runs in $O(n \log s)$ time, where n is the input size and s is the sinuosity of the polygon. The latter quantity is in general extremely small, but even in most of the cases where it is large, we believe that our algorithm, being highly adaptive, will behave linearly or quasi-linearly. We have also exhibited a large class of polygons for which the algorithm runs in time $O(n \log \log n)$. Identifying exactly the class of polygons for which the algorithm runs in linear time is a challenging, yet very worthwhile endeavor; indeed, we conjecture that this class is considerably larger than the subset exhibited in this paper. The theoretical implications of our results are, in particular, evidence that, in general, triangulating is easier than sorting, although, in the worst case, no such statement can yet be made. Deciding on this issue once and for all is undoubtedly one of the most outstanding open questions of computational geometry.

REFERENCES

1. CHAZELLE, B. A theorem on polygon-cutting with applications. In *Proceedings of the 23rd IEEE Annual Symposium on Foundations of Computer Science*, (Chicago, Nov. 3–5 1982). IEEE, New York, pp. 339–349.
2. GAREY, M. R., JOHNSON, D. S., PREPARATA, F. P. AND TARJAN, R. E. Triangulating a simple polygon. *Inf. Proc. Lett. 7*, 4 (June 1978), 175–180.
3. GUIBAS, L., AND STOLFI, J. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. In *Proceedings of the 15th ACM Annual Symposium on Theory of Computing*, (Boston, April 25–27 1983) ACM, New York, pp. 221–234.
4. HERTEL, S., AND MEHLHORN, K. Fast triangulation of simple polygon. In *Proceedings of the Conference on Foundations of Computing Theory* (Borgholm, Sweden, Aug. 21–27). Springer-Verlag, New York, 1983, 207–218.
5. KNUTH, D. E. The Art of Computer Programming: Sorting and Searching. Vol. 3, Addison-Wesley, Reading, Mass., 1973.
6. SHAMOS, M. I. Computational geometry. PhD dissertation, Dept. of Computer Science, Yale University, New Haven, Conn., 1978.
7. SCHOONE, A. A., AND VAN LEEUWEN, J. Triangulating a star-shaped polygon. Tech. Rep. RUV-CS-80-3, Univ. of Utrecht, April 1980.
8. TOUSSAINT, G. T. Pattern recognition and geometrical complexity. In *Proceedings of the 5th International Conference on Pattern Recognition* (Dec. 1980), pp. 1324–1347.
9. TOUSSAINT, G. T., AND AVIS, D. On a convex hull algorithm for polygons and its application to triangulation problems. *Pattern Recog. 15*, 1 (1982) 23–29.

Received November 1983; accepted August 1984