Janardan

4 Approximation and Decomposition of Shapes

Bernard Chazelle
Princeton University

ABSTRACT

This paper reviews some of the techniques developed recently for approximating or decomposing geometric shapes in two and three dimensions. The relevance of these techniques to robotics is compelling. As is well-known in the area of computer graphics, replacing complex objects by approximations or decompositions into simple components can greatly facilitate parts-manipulation (e.g. detection of interference, hidden-line elimination, shadowing). For this reason, as well as for the theoretical challenges raised by these problems, considerable work has been devoted on this subject in recent years. This paper attempts to survey some of the major advances in the area of approximation and decomposition of shapes. Methods of particular interest for their practical significance or their theoretical import are reviewed in detail.

1. INTRODUCTION

Planning collision-free motion and recognizing shapes are central tasks in robotics and automated assembly systems, in particular. In both cases, the process is complicated by the necessity to operate in real-time and the occasional presence of fairly complex objects. To cope with the complexity of the geometric problems at hand, two lines of attack can be contemplated: one involves simplifying the shapes by computing approximations of them; the other calls for rewriting the objects as combinations of simple parts. Both of these approaches have met with considerable success in computer graphics and automated design, and their relevance to robotics as a whole is evident. It is the aim of this paper to present a brief survey of the various methods and techniques known today in this area.

2. APPROXIMATION OF SHAPES

The goal is to replace a complex shape S by a simpler figure F that captures morphological features of S. Since F will often be used to approximate the clearance of S amidst obstacles, it will be assumed to enclose S, so as to provide a conservative approximation. We review some of the most common approximation schemes previously devised.

2.1. Convex Hulls

The *convex hull* of a set S of n points is the intersection of all convex sets containing S. We omit the proof that the convex hull of S, denoted C, is a convex polygon whose vertices (the *extreme* points) are points of S. A simple characterization of a vertex of the convex hull states that a point is extreme if and only if it does not lie strictly inside any triangle formed by any three points. This leads to a trivial $O(n^4)$ time algorithm. For each such triangle, eliminate each point lying inside. The remaining points will be extreme. A number of more efficient algorithms have been (only recently) discovered. We propose here to review some of the most important, practical, and/or original methods. The honor of discovering the first *optimal* convex hull algorithm goes to R. Graham (1972).

Graham Scan: The method involves sorting the points in a preliminary stage and then retrieving the convex hull in linear time via a procedure traditionally known as a *Graham scan*. Let $\{p_1, \ldots, p_n\}$ be the points of S sorted angularly clockwise around p_1, the point of S with largest y-coordinate (take the one with largest x-coordinate to break ties, if necessary). Computing the list of p_i's can be done in $O(n\log n)$ time.

Before proceeding any further, we should make an important observation concerning the computation of angles in particular and the design of geometric algorithms in general. Although it is very helpful and intuitive to think in terms of angles in the design part of a geometric algorithm, computation of angles and of their natural functions (such as trigonometric functions) tends to be computationally expensive. There are usually ways around this difficulty, however. Most often, we find all that is needed is a primitive operation to determine the relative order of two vectors. This operation can be implemented with only a few multiplies and subtracts. Figure 4.1 illustrates this concept. Let $\theta = \angle(ab, ac)$ be the angle from ab to ac, measured in counterclockwise order. We may assume $-\pi < \theta \le \pi$. As a matter of terminology, we will often refer to the expression "b, a, c *form a right (resp. left) turn*" to mean that $\theta > 0$ (resp. <0). If p_x, p_y denote respectively the x and y coordinates of point p, we easily show that

$$\theta > 0 \leftrightarrow (b_x - a_x)(c_y - a_y) - (c_x - a_x)(b_y - a_y) > 0.$$

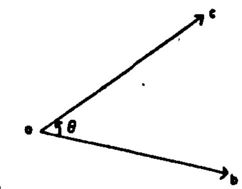


FIG. 4.1.

From a programming perspective, this equivalence allows us to compare, merge, and sort angles without having to compute them explicitly. For example, let a, $a_1, \ldots, a_p$ be $p + 1$ distinct points in E^2. To sort the points $a_1, \ldots, a_p$ angularly around a (clockwise, starting at a_1), one can simply use one's favorite sorting routine and re-implement the primitives $<, =, >$ used by the algorithm. Partition the original set of points into $L = \{a_i \mid \angle(aa_i, aa_1) > 0\}$, $M = \{a_i \mid \angle(aa_i, aa_1) = 0\}$ and $H = \{a_i \mid \angle(aa_i, aa_1) < 0\}$, and concatenate L, M, and H, after having sorted the sets separately. To do so, any comparison of the form $A_i < A_j$ in the algorithm should be replaced by $\angle(aa_j, aa_i) > 0$ and implemented with the formula given above. What we have here is an instance of *abstract data types*, a notion of crucial importance in geometric algorithm design. It is often possible and advantageous to specify geometric algorithms solely in terms of generic operations and treat the implementation of types separately. The reason is that many of these algorithms are extensions of well-known combinatorial algorithms couched in geometric terms. Identifying and separating their geometric and combinatorial components often simplifies the entire process of design, analysis, and implementation. With this observation made, we can now return to the convex hull problem.

Let px designate the horizontal ray emanating from p in the right direction and assume that $\angle(p_1p_2, p_1x)$ has the smallest value of any $\angle(p_1p_i, p_1x)$. Imagine that a rubber band is attached to p_1 at one end, and that the other end is taken to $p_2, p_3, \ldots, p_n$, in turn (Fig. 4.2). At the end of this process, the rubber band will be shaped exactly as the convex hull of S. The Graham scan is essentially a computer simulation of this process. Conceptually, it is easiest to describe this process as a stack manipulation algorithm. Recall that a stack is an *abstract data type* (i.e., a set and some operations defined on it), which behaves much like a pile of trays in a dining-hall. The only operations allowed are: $PUSH(e)$ (add e to the

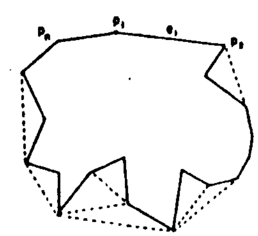


FIG. 4.2.

top); *POP* (remove top of stack); the top of the stack is designated *TOP*. The algorithm is based on the notion of *left* and *right turns*. At any stage, the stack contains the edges which should be on the convex hull were the algorithm to terminate at that instant. If the next vertex encountered witnesses a right turn, a new edge is added to the stack, otherwise, edges are popped off until the right turn condition is satisfied. The top of the stack is an edge *TOP*, whose endpoints are denoted t_1 and t_2, in clockwise order around the boundary.

```
begin
PUSH (p1,p2)
for i = 3, . . . ,n
    begin
        while (t1, t2, pi) turns left
            begin POP end
        PUSH (t2, pi)
    end
end
```

In an actual implementation, it might be more convenient (although conceptually more complicated) to simulate the stack with an array where each record holds a vertex and not an edge. The problem with representing edges is that information is essentially duplicated. See Sedgewick (1983, pp. 329) for an example of working code. A simple examination of the pseudo-code shows that the algorithm runs in linear time (the stack is *PUSHed* at most once per point in S).

A Lower Bound: We easily see why Graham's $O(n\log n)$ time algorithm is optimal. We can sort n numbers $a_1, \ldots, a_n$ by computing the convex hull of the n points $\{(a_1, a_1^2), \ldots, (a_n, a_n^2)\}$ and reading off the vertices of the hull in clockwise order. Of course, this reduction assumes that by "computing the convex hull" we mean determining the order of the extreme points around the boundary. What if we only require the extreme points without reference to their order? Using a fairly technical argument, which we will not develop here, Yao has proven that even in this relaxed instance the convex hull problem requires $\Omega(n\log n)$ operations (Yao, 1981). Yao's model of computation is limited to decision trees with quadratic polynomial evaluations at the nodes. This assumption is actually quite realistic since all known methods seem to fall squarely in this model of computation. At any rate, calling upon deep results of algebraic geometry, Ben-Or has generalized Yao's lower bound to any decision tree with fixed-degree polynomial evaluation at the nodes (Ben-Or, 1983).

Gift-Wrapping: One drawback of the Graham scan is that it has the same complexity regardless of the size of the output. For example, if the convex hull happens to have only a small number of vertices, it is conceivable that a faster algorithm could be used. The *gift-wrapping* method, also known as the *Jarvis march* (Jarvis, 1973), provides a (partial) response to this concern. Once again, the algorithm is the simulation of a very simple physical process. Let p_1 be as usual the point of S with largest y-coordinate. Attach a long rope to p_1 and wrap it around S one step at a time. A *step* is the discovery of a new contact with the rope (i.e., a new extreme point). A simple angle "calculation" (see remark above concerning angles) allows us to implement every step in linear time. This leads to an $O(nh)$ time algorithm, where h is the number of extreme points. How will this method fare against the Graham scan on the average? Let $h^*(n)$ be the expected number of extreme points in a set of n points chosen uniformly and independently inside a bounded region R. It has been shown by Renyi and Sulanke (1963) that if R is a square or any convex k-gon for a fixed k, $h^*(n) = O(\log n)$, and that if R is a circle, $h^*(n) = O(n^{1/3})$ (Shamos, 1978). This shows that for the case of a uniform distribution in a k-gon the Jarvis march will asymptotically match the Graham scan in the average case.

Divide-and-Conquer: A very good algorithm in the sense of expected behavior was developed by Bentley and Shamos (1978). Assume that the points of S are stored in an array $A[1, n]$. Recursively, the algorithm computes the convex hull P of $A[1, \lfloor n/2 \rfloor]$ and the convex hull Q of $A[\lfloor n/2 \rfloor + 1, n]$; then in time proportional to the added size of P and Q it computes the convex hull of S. The expected running time of the algorithm $T(n)$ follows the recurrence relation, $T(n) = 2T(n/2) + O(h^*(n))$. It follows that $T(n) = O(n)$ in the case of a uniform distribution inside a k-gon or a circle. Two important facts to observe:

1. The recursive calls involve passing indices or pointers and *not* the entire arrays. The latter solution would always entail an $O(n\log n)$ run time;
2. The uniformity of the distribution is preserved through recursive calls since the subsets handled are defined independently of the value of their elements.

To complete the demonstration of the efficiency of Bentley and Shamos' method, we must describe a linear method for "merging" two convex polygons. Let P and Q be two convex polygons with respectively p and q vertices. We describe a method for computing the convex hull of $P \cup Q$ in $O(p + q)$ operations. Let $\{v_1, \ldots, v_p\}$ and $\{w_1, \ldots, w_q\}$ be a clockwise vertex-list of P and Q, respectively. Since P is convex the sequence of vertices $\{v_1, \ldots, v_p\}$ is angularly sorted around v_1. Is this also true of Q? If v_1 happens to lie inside Q, it is certainly so. Otherwise, the sequence of angles is *bimodal*, i.e. unimodal up to a circular permutation. By checking each vertex of Q, compute the two *segments of support* v_1w_i and v_1w_j: these segments are such that v_1, w_i, w_{i-1} and v_1, w_i, w_{i+1} form right turns, and v_1, w_j, w_{j-1} and v_1, w_j, w_{j+1} form left turns (Fig. 4.3) — arithmetic on indices is done mod q. Note that these two conditions will be satisfied if and only if $v_1 \notin Q$, so whether $v_1 \in Q$ or $v_1 \notin Q$ need not be decided beforehand. Every vertex from w_{j+1} to w_{i-1} clockwise (if any) can be discarded and Q can be redefined as $\{w_i, w_{i+1}, \ldots, w_j\}$. In all cases, Q now forms a monotone sequence of angles with respect to v_1, so we can merge the vertices of P and Q together in linear time, and then apply the Graham scan on the resulting

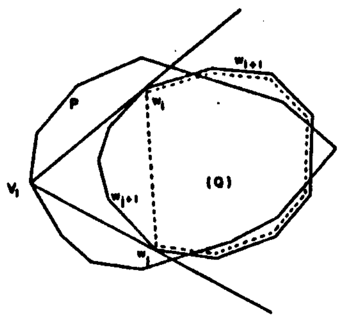


FIG. 4.3.

list of vertices. This will provide us with the convex hull of $P \cup Q$ in $O(p + q)$ time, which completes the description of Bentley and Shamos' algorithm. Despite the linear expected running time of the algorithm, one will notice the similarity between this method and *mergesort* (Aho, Hopcroft, & Ullman, 1974; Knuth, 1973; Sedgewick, 1983).

Divide-and-Conquer with Sorting: If we apply the Bentley-Shamos algorithm after having sorted the points of S by x-coordinate, the merge part can be easily implemented without resorting to a Graham scan (Preparata & Hong, 1977) (of course, by doing so we forsake any hope of breaking the $O(n\log n)$ barrier, as it is well known that sorting takes on the order of $n\log n$ operations, even on the average). Let P and Q be, as above, the two convex polygons to be *merged*. We assume that P lies totally to the left of Q. The convex hull of $P \cup Q$ is obtained by computing the *upper* and *lower bridges* of P and Q, i.e., the two unique segments joining a vertex of P and a vertex of Q with both polygons lying entirely on the same side of the infinite lines passing through the bridges (Fig. 4.4). Whether a segment is a bridge can be checked locally in constant time since P and Q are convex. For example, the upper bridge $v_k w_l$ is such that all four turns (v_k, w_l, w_{l-1}), (v_k, w_l, w_{l+1}), (v_{k-1}, v_k, w_l), (v_{k+1}, v_k, w_l) are right. To compute the bridges of P and Q, pick the leftmost vertex of P, say, v_1, and compute the segments of support, $v_1 w_i$ and $v_1 w_j$, as described earlier. Then proceed to *roll* the line U (resp. L) passing through $v_1 w_i$ (resp. $v_1 w_j$) clockwise (resp. counterclockwise) around Q, until the line contains the upper (resp. lower) bridge (Fig. 4.5).

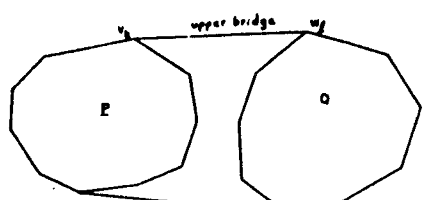


FIG. 4.4.

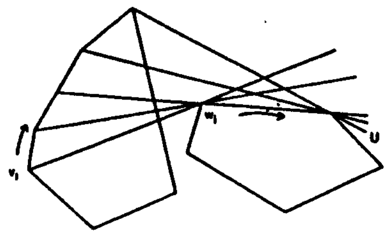


FIG. 4.5.

Rolling the line U passing through $v_1 w_i$ can be done by maintaining the two current contact-points, v_a and w_b, and deciding whether v_{a+1} or w_{b+1} should be the next one. Once again, this procedure can be understood as a two-way merge (i.e., merge of two sorted lists), with the primitives $<, =, >$ re-implemented as *left turn, no turn, right turn* conditions. The only major difference with merging is in the way the algorithm terminates. Rather than scanning the two lists through the end, the algorithm must keep on checking whether the bridge has been found, and stop as soon as it has. Unlike the previous one, the running time of this method is inherently $O(n\log n)$ because of the preliminary sorting. Why then did we bother mentioning it at all? The truth is that the algorithm serves as the ideal stepping-stone for the next method, the first one known to be optimal in both input and output size.

Marriage-Before-Conquest: Aside from the fact that the method we last described entails a sort on n numbers, a simple look at Fig. 4.4 reveals its main shortcoming. Once the bridges have been found, many edges of P and Q are bound to become irrelevant, therefore all the work done to obtain these edges will turn out to be useless. It would be nice in some sense to be able to run the algorithm *backwards*. We would first compute the bridges and then discard all the points lying directly below or above them, at which stage we would simply iterate recursively on the two sets of remaining points. The advantage of such a scheme would be to guarantee that every bridge computation is effective; in other words, no bridge computed would ever be thrown away. The bottleneck caused by the preliminary sorting can in turn be alleviated by resorting to a linear-time median algorithm (Aho et al., 1974; Knuth, 1973). This will allow us to partition the input set into two roughly equal-sized sets at a linear cost. This approach was taken by Kirkpatrick and Seidel and led to the first optimal algorithm in the sense of input *and* output size. More specifically, if h denotes as usual the number of extreme points in S, Kirkpatrick and Seidel's method runs in time $O(n\log h)$. Optimality follows from an information-theoretic argument given in Kirkpatrick and Seidel (1983).

We briefly review the main facets of the algorithm, referring the reader to Kirkpatrick and Seidel (1983) for further details. To begin with, we may restrict ourselves to the *upper hull*, i.e., the chain of the convex hull that runs clockwise from the leftmost to the rightmost point. The lower hull will be computed in a similar fashion, from which the convex hull will follow directly. Let $\{p_1, \ldots, p_n\}$ be the points of S in arbitrary order, with $x(p_i)$ denoting the x-coordinate of p_i. Let a be a real number and P (resp. Q) be the convex hull of $\{p \in S \mid x(p) \leq a\}$ (resp. $\{p \in S \mid x(p) > a\}$). The expression "*upper bridge over a*" refers to the upper bridge of P and Q. Let's assume for the time being that we know how to compute the *upper bridge over a* in a linear number of steps. The upper hull of S is computed by calling $HULL(1,n,S)$.

```
HULL(k,l,S)
Find median a of x-coordinates in S
Find upper bridge p_i p_j of S over a, with x(p_i) ≤ x(p_j)
S_1 ← {p ∈ S | x(p) ≤ x(p_i)}
S_2 ← {p ∈ S | x(p) ≥ x(p_j)}

"note that p_i ∈ S_1 and p_j ∈ S_2"

if i = k
    then print (i)
    else HULL(k, i,S_1)
if j = l
    then print (j)
    else HULL(j, l, S_2)
```

It is not difficult to see that in the worst case, one bridge computation will require $O(n)$ steps, two will require $O(n/2)$ steps each . . . , and more generally 2^i bridge computations will require $O(n/2^i)$ steps each. In other words, there is some integer k such that $\Sigma_{i<k} 2^i \gtrsim h$ and the running time is $O(\Sigma_{i<k} 2^i(n/2^i))$. This leads to the optimal $O(n\log h)$ time complexity announced earlier. To complete the demonstration, we need to describe a linear time algorithm for computing the upper bridge of P and Q. In a linear number of operations, we wish to be able to discard at least a fixed fraction of the points of S. Let α be an arbitrary slope $\in (-\infty, +\infty)$ and let L_α be the line of the form $Y = \alpha X + \beta$, with largest β such that there exists at least one index i with $p_i \in L_\alpha$. Assume that $x(p_i) < a$. The key observation is that for any pair (p_u, p_v) such that $x(p_u) < x(p_v)$ one may discard the point p_u provided that the slope of $p_u p_v$ exceeds α (Fig. 4.6). Indeed, because of the relative positions of p_i and p_v, the point p_u cannot be an endpoint of the upper bridge. Symmetrically, if we have $x(p_i) > a$, we will be able to discard p_v every time slope$(p_u p_v) < \alpha$. In order to balance the odds, we will pair up (p_1, p_2), (p_3, p_4), etc. and compute the median slope α of the $\lfloor n/2 \rfloor$ segments $p_{2i-1}p_{2i}$. Let $p_i \in L_\alpha$; once we have identified whether $x(p_i) < a$ or not, as many as $\lfloor n/4 \rfloor$ points will immediately fall out of contention. Iterating on this process will eventually produce the upper bridge. The time $T(n)$ taken by the algorithm satisfies the recurrence relation $T(n) = T(n - \lfloor n/4 \rfloor) + O(n)$, which gives $T(n) = O(n)$. Our exposition has left a number of special cases conveniently hidden (e.g.

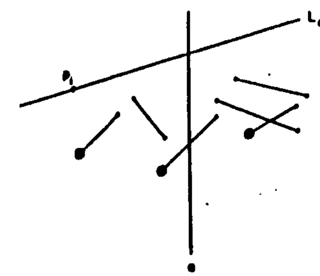


FIG. 4.6.

segments with infinite slopes, several segments with slope α, several points with x-coordinate a, etc.). All these difficulties can be easily handled with extra care, however, so we simply refer the reader to Kirkpatrick and Seidel (1983) for details.

Quickhull: Eddy and Floyd independently developed a convex hull algorithm, which although quadratic in the worst case, behaves remarkably well in practice (Eddy, 1977). This is not without resembling the case of *quicksort*, a sorting method which fares better than almost any other in practice albeit vulnerable to extremely poor behavior in the worst case (Aho et al., 1974; Knuth, 1973; Sedgewick, 1983). For the reader familiar with *quicksort* the resemblance will shortly appear much deeper than this, which is the reason why we christened the algorithm *quickhull*. Let A and B denote respectively the highest and lowest point in S. Compute the two points C, D of S on the leftmost and rightmost lines parallel to AB. Eliminate the points inside the quadrilateral $ABCD$. Sweep lines parallel to AC, CB, BD, DA, in turn, *outward* from the quadrilateral, and eliminate the points from inside the four triangles thus created (Fig. 4.7). Each of these triangles is formed by the segment from which the sweep starts and the last point swept. The algorithm iterates on this process. Under some reasonable assumptions concerning the distribution of the points in S, it is possible to show that the algorithm will run in linear expected time. In practice, this algorithm could be used as a preprocessor to Graham's algorithm, for example. The idea is that after a small number of passes, only very few points should be left, so a sort

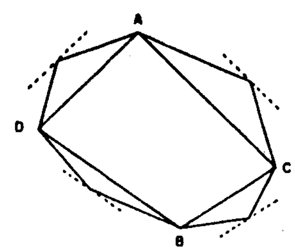


FIG. 4.7.

is unlikely to be time-consuming. Similar to *quicksort*, quickhull is a prime candidate for a simple, practical, and efficient implementation.

Approximation Method: On the practical side, it is worthwhile to mention a very fast convex hull algorithm due to Bentley, Faust, and Preparata (1982). The algorithm produces an approximation of the convex hull, that is to say, it might fail to report the *exact* convex hull, but can always be guaranteed to produce a convex hull arbitrarily *close* to the exact one (for some realistic measure). This might often be acceptable in situations where only a rough approximation of the morphology of the point-set is desired.

Higher Dimensions: In three dimensions, the situation is (ironically) much "clearer" than in E^2 because of the paucity of efficient algorithms. Preparata and Hong (1977) have devised an optimal $O(n\log n)$ time method which constitutes an extension of the divide-and-conquer method with sorting given for E^2. For higher dimensions ($d > 3$), algorithms are given in Chand and Kapur (1970), and Seidel (1981). Chand and Kapur (1970) extend the gift-wrapping method to arbitrary dimensions, while Seidel (1981) gives a general algorithm which is optimal when d is even. The running time of his algorithm is $O(n\log n + n^{\lfloor \frac{d+1}{2} \rfloor})$.

Dynamic Convex Hulls and Convex Layers: Preparata (1979) has described an optimal algorithm for inserting new points into a two-dimensional convex hull. If the convex hull has n vertices, any new point can be added to it in $O(\log n)$ time. One drawback of this method is that points which either fail to be on the convex hull, or stop being on the hull as a result of an insertion, are lost, thereby making deletions impossible. Overmars and van Leeuwen (1981) have shown that by sacrificing a little in time it is possible to accommodate deletions (see chapter by Dobkin and Souvaine in this volume). Their algorithm handles any update of this nature (insertion or deletion) in $O(\log^2 n)$ time. Chazelle has shown, on the other hand, that if only deletions from the convex hull are allowed, then optimal (amortized) time can be achieved (Chazelle, 1985). This, in particular, allows for the computation of the *convex layers* of an n-point set in $O(n\log n)$ time. These are the various polygons obtained by computing the convex hull of the point set, and then removing all the vertices from the set and iterating on this process until all the points are gone. This collection of polygons turns out to be very useful in statistical analysis (Shamos, 1978). It also allows for efficient range searching, as has been shown in Chazelle, Guibas, and Lee, 1983.

Convex Hull of Polygon: An important class of convex hull applications involves a set of points joined together by a simple curve. Recall that a curve is simple if it is not self-intersecting. Let P be a simple polygon with n vertices. This exist several algorithms for computing the convex hull of P in $O(n)$ time (Bhattacharya & El Gindy, 1984; Graham & Yao, 1983; Guibas, Ramshaw, & Stolfi, 1983; Lee, 1983; McCallum & Avis, 1979). Most of these algorithms can be viewed as generalizations of the Graham scan. A pointer scans the boundary of the polygon, while one or two stacks keep track of the "current" convex hull.

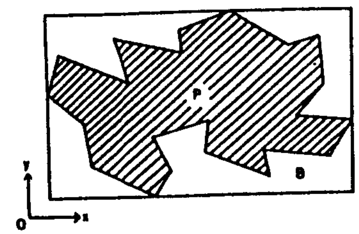


FIG. 4.8.

2.2. Rotating Calipers

A simple, albeit rough, method for approximating the shape of an object is to place it in a *bounding box*. In Fig. 4.8, object P has been tightly enclosed by a rectangle B whose sides are parallel to the axes. The term *axes* refers here to the two directions of an orthogonal system of coordinates (Ox, Oy). This method is very popular in computer graphics (Newman & Sproull, 1979), because it greatly simplifies the task of removing hidden parts from a scene, testing intersection between several objects, etc. In raster graphics, the horizontal and vertical directions are privileged, so it is natural to orient bounding boxes along them. Assume that P is represented as a polygon with vertices $p_1, \ldots, p_n$ in clockwise order. We easily compute B in $O(n)$ time by determining the maximum and minimum x and y coordinates among the p_i's.

Sometimes, insisting on a specific orientation of the bounding box can cause considerable degradation in the quality of the approximation. Figure 4.9 suggests that the proper measure of the *quality* of the bounding box should be the area that it occupies, regardless of its orientation. By rotating the box B counterclockwise by approximately 45 degrees, one reaches another enclosing box of much higher quality by this measure. The question that leaps to mind is then:

What is the complexity of computing the smallest-area rectangle enclosing the polygon P?

If P is highly symmetric, it is easy to see that its smallest-area enclosing rectangle may not be necessarily unique. For this reason, we limit our search to *any* such rectangle, which we generically denote $\mathcal{A}(P)$. Toussaint (1983) has

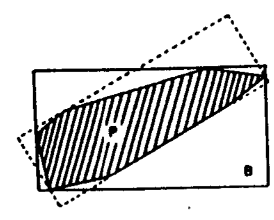


FIG. 4.9.

shown that finding $\mathcal{A}(P)$ can be done in $O(n)$ time. His method relies on a number of geometric observations. To begin with, since $\mathcal{A}(P)$ contains the convex hull C of P, we can advantageously drop P from consideration and restrict our investigation to C. As has been mentioned earlier, C can be computed in $O(n)$ time. The next crucial fact is a theorem by Freeman and Shapira (see Toussaint, 1983) which states that $\mathcal{A}(P)$ always has one side containing an edge of C. This suggests an $O(n^2)$ time algorithm based on the previous bounding box method. Make the x-axis parallel to each edge of C in turn, and for each orientation obtained solve the *bounding box* problem in $O(n)$ time. Toussaint observed that there is no need to repeat each computation from scratch for every position of the x axis. Instead, one should try to *batch* computations together by trying to derive the next answer directly from the previous one.

This intuition is materialized in the notion of *rotating calipers*. Let $c_1, \ldots, c_p$ ($p \leq n$) be the vertices of C given in clockwise order. Consider the bounding box *anchored* at $c_{i-1}c_i$, i.e., the smallest-area enclosing rectangle with one side containing $c_{i-1}c_i$. This rectangle is formed by four infinite straight lines L_i, L_j, L_k, L_l (Fig. 4.10). Let c_i, c_j, c_k, c_l be the vertices in contact with the boundary of the rectangle, with $c_i \in L_i$, $c_j \in L_j$, $c_k \in L_k$, $c_l \in L_l$ (whenever a whole edge is in contact with the rectangle, we choose its *last* vertex clockwise as its representative). We look at L_i, L_j, L_k, L_l as calipers rotating clockwise around C in such a way that they preserve the integrity of the rectangle they form. Because of Freeman and Shapira's result, we may restrict our attention to the positions of the calipers for which one line contains an edge of C. To compute the next position from the current one, it suffices to compare the four angles, $\alpha_i, \alpha_j, \alpha_k, \alpha_l$ formed respectively by L_i and c_i, c_{i+1}, L_j and c_j, c_{j+1}, L_k and c_k, c_{k+1}, and L_l and c_l, c_{l+1}. The smallest of them (α_k in Fig. 4.10) determines the next position of the calipers. The rotation will continue until the calipers are back to their initial position. Because of the obvious symmetry of the calipers, the rotation can be limited to 90 degrees.

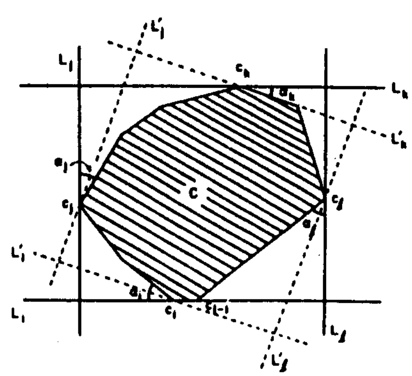


FIG. 4.10.

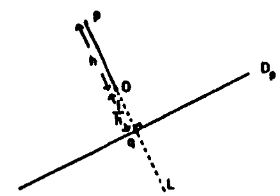


FIG. 4.11.

The algorithm is implemented by maintaining four pointers to keep track of c_i, c_j, c_k, and c_l. It strongly resembles the traditional method for merging four sorted lists into one, the so-called 4-way merge (Knuth, 1973). In particular, its complexity is clearly $O(n)$ in both time and space. The similarity with merging can be better seen if we map the polygon C into a different space. We resort to a technique of very common use in computational geometry: a *dual transform*.

The idea is (in this case) to put points and lines in one-to-one correspondence so as to be able to rephrase problems on lines as problems on points and vice-versa. Point $p = (a,b) \neq (0,0)$ is (bijectively) mapped to the line $D_p : aX + bY + 1 = 0$, and conversely D_p is mapped to p. Intuitively, this mapping is effected by considering the line L passing through both p and the origin O, and determining the point q of L at a distance $1/|Op|$ from p, with O on the segment pq (Fig. 4.11). The line D_p is obtained by taking the normal to L passing through q. It is easy to see that the transformation D preserves incidence relations. For this reason, it is tempting to consider each of the lines bounding C and map them via the dual transform D. Let L_i be the line containing the edge c_ic_{i+1} and let $v_i = D_{L_i}$ denote its dual point ($1 \leq i \leq p$; index arithmetic taken mod p). Assume that the origin has been chosen to be *inside* the polygon C. It is easy to see that the sequence $\{L_1, \ldots, L_p\}$ is mapped into a sequence of points $\{v_1, \ldots, v_p\}$, which form a clockwise vertex-list of a convex polygon V.

A mechanical analogy of this correspondence is to roll a line (a caliper) around C clockwise and follow the motion of its dual point. The trajectory of this point will be precisely the convex polygon V. Define a *cross* to be a set of two lines passing through O normal to each other. A cross intersects V in four points, called *cross-points*. Our original set of four calipers is mapped into a set of four cross-points. The analogy is now complete: rotating the calipers corresponds to moving the cross clockwise. If L is a list of numbers, let $L + x$ be the list obtained by adding the number x to each element in L, and let θ_i be the slope of Ov_i ($1 \leq i \leq p$). Let L be the sorted list of angles $\{\theta_1, \ldots, \theta_p\}$. The 4-way merge of lists to which we evasively referred as the backbone of the rotating algorithm is now taking shape. A moment's reflection shows that the algorithm for computing the smallest-area rectangle all but boils down to merging together the four lists L, $L + 90$, $L + 180$, and $L + 270$.

This example and the little excursion into dual space which followed tell us something. First, a fairly specific geometric problem is solved in a very simple way by use of a general, unifying technique: the rotating calipers. Second, this technique is nothing but a geometric instantiation of a fundamental programming concept: k-way merging (Knuth, 1973). Put in this light, it is no surprise that the rotating calipers should find many other applications in computational geometry. For example, it is possible to find the *diameter* of a set of n points in E^2 in $O(n\log n)$ time, i.e., the largest distance between any two points (Shamos & Hoey, 1975; Toussaint, 1983). We can do so by first computing the convex hull of the point-set in $O(n\log n)$ time, and then rotating two parallel calipers around the hull in $O(n)$ steps. The two points realizing the maximum distance must appear as contact-points of the calipers in some position. Of more direct relevance to the subject of shape-approximation, we pose the following problem.

What is the complexity of computing the smallest-area triangle enclosing a convex polygon?

Note that solving this problem will also allow us to compute the smallest-area triangle enclosing an arbitrary set of points. We do so by taking the convex hull of the points as a preprocessing step, and thus reduce the problem to the one posed. Klee and Laskowski (1985) have proposed an $O(n\log^2 n)$ time algorithm for computing the smallest-area triangle of a convex polygon with n vertices (or any of them if there are several). Their algorithm was recently improved to optimal $O(n)$ time in O'Rourke, Aggarwal, Maddila, and Baldwin (1984). To quote O'Rourke et al., "the strength of their (Klee and Laskowski's) paper lies in establishing an elegant geometric characterization of these (local) minima, which permits them to avoid brute-force optimization." The strength of O'Rourke et al.'s paper, on the other hand, is to prove an *interspersing lemma*, which enables them to use rotating calipers as a guiding hand. Three calipers are used to represent tentative enclosing triangles: the only conceptual difference with the rectangular case is that (1) angles between calipers are not fixed; (2) the three calipers do not have a uniform behavior. Nevertheless, the algorithm decides on the basis of a constant-time test which caliper should be moved next and by how much. Since calipers move in the same direction, the running time is trivially linear. Once again, we can interpret the basic method as a k-way merge ($k = 3$), although the non-uniformity of the ranking criterion makes the analogy slightly less compelling.

2.3. Circle and Ellipse Enclosure

We leave aside polygonal shapes temporarily and turn our attention to smoother curves like circles and ellipses. Conic sections or for that matter any polynomial curves of reasonably small degree have the advantage of being space-effective: just a few coefficients are necessary to represent them. Also being *smoother* than polygons, these curves will often provide finer approximations than, say, triangles, rectangles, or more generally polygons with a small number of vertices. To begin with, we ask the following question:

What is the complexity of computing the smallest-area circle enclosing a set of points?

2.3.1. Circle Enclosure. The case of circles is particularly interesting from an algorithmic standpoint. The problem can be traced as far back as Sylvester (1857) and has gone through numerous developments (see Shamos, 1978; Megiddo, 1983 for a short bibliography). Until recently the best solution known was due to Shamos and Hoey (1975) and was based on a geometric construction, called the *Voronoi diagram*. Its complexity of $O(n\log n)$ was subsequently improved to optimal $O(n)$ by Megiddo (1983) in a seminal paper on linear programming in $\Re^3$. The two methods are radically different and deserve separate treatments.

Shamos and Hoey's method: This is the simpler of the two. It rests crucially on a geometric construction, called the *farthest-point Voronoi diagram*. Let $S = \{p_1, \ldots, p_n\}$ be a set of n points in the Euclidean plane. Partition the plane into regions with common *farthest neighbors*. More precisely, for any point p, let $f(p)$ be the index i such that for each $j \neq i$, the distance $|pp_i|$ exceeds the distance $|pp_j|$. Whenever more than one index i can be found, $f(p)$ stands for the set of such indices. The function f partitions E^2 into faces, edges, and vertices. Let V_i denote the face associated with p_i. V_i is the intersection of each half-plane that is delimited by the bisector of (p_i, p_j) $(1 \le j \neq i \le n)$ and contains p_j. For this reason, either V_i is empty or it is a convex polygon (in the latter case, it is easy to show that V_i is unbounded). Furthermore, V_i is not empty if and only if p_i is a vertex of the convex hull of S. This leads to the fact that the faces V_i can be ordered cyclically around the boundary of the convex hull (Fig. 4.12). Since the set of edges forms a free tree (connected acyclic graph) it immediately follows that the subdivision contains at most $F - 2$ vertices, where F is the number of faces. Since $F \le n$, we conclude that the farthest-point Voronoi diagram of n points in the plane has at most $n - 2$ vertices. Shamos and Hoey (1975) have

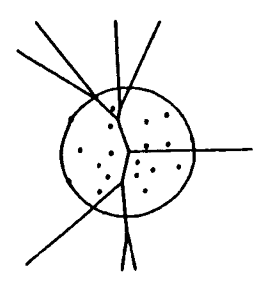


FIG. 4.12.

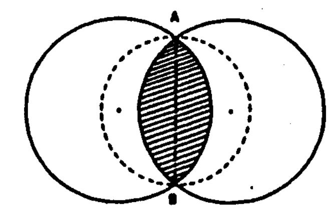


FIG. 4.13.

described how to compute this diagram in $O(n\log n)$ time, using a divide-and-conquer strategy.

We next show how to compute the smallest circle using this diagram. Let C be the smallest-area enclosing circle of S. To see that C is uniquely defined, assume the existence of two such circles C and C', and let A, B be their two intersection-points (note that the two circles *must* intersect). Every point of S lies in the shaded area in Fig. 4.13, therefore the circle of diameter AB contains S and is smaller than C and C', a contradiction. Let p_i, p_j be the pair of points in S whose distance to each other is maximum. This distance, called the diameter of S, can be obtained in $O(n)$ time using the farthest-point Voronoi diagram. To do so, determine the farthest neighbor of each point in S. Once the diameter is available, check whether every point of S lies inside the circle with diameter p_ip_j. If yes, this circle is clearly C; otherwise, C passes through at least three points p_u, p_v, p_w of S. Obviously the center of C is then the vertex at the intersection of V_u, V_v, V_w. Trying out all vertices of the farthest-point diagram and keeping the one whose associated circle has maximum radius will give us C. To conclude, we have described an $O(n\log n)$ time method for computing the smallest-area circle enclosing n points.

Megiddo's Method: Recently, Megiddo proved the surprising result that C can actually be computed in only a linear number of operations. His method is based on a general technique used primarily in the context of linear programming. It is this technique which inspired Kirkpatrick and Seidel to design their ingenious marriage-before-conquest convex hull algorithm (see section 2.1). Since the entire algorithm can be explained reasonably simply in a self-contained manner, we cannot resist the temptation of trying. As the reader will undoubtedly realize, the method contains a treasure of algorithmic insights. In the following, the pair (a_i, b_i) will denote the coordinates of p_i $(1 \leq i \leq n)$. To begin with, we solve a simpler problem. Let L be an arbitrary line; a circle is said to be *centered* at L if its center lies on L. The question we pose is the following:

I. *What is the smallest-area enclosing circle centered at L?*

The same argument used above shows that this circle, denoted C_L, is uniquely defined. For the sake of convenience, we think of L as the x-axis, so we can represent the center of C_L by its x-coordinate, x^*. Megiddo's approach is similar

in spirit to the well-known linear-time median algorithm (Aho et al., 1974; Knuth, 1973). The idea is to identify and *discard* points that are irrelevant to the problem. A point p is irrelevant if it can be determined that C_L is the smallest-area circle enclosing $S - \{p\}$ centered at L. We will show a method for discarding at least a fraction α of the input set in linear time. This will allow us to zoom in on the solution in time $O(n + (1-\alpha)n + (1-\alpha)^2n + \ldots) = O(n)$. To do so, we need to describe a number of primitive operations.

1. To begin with, it would be nice to be able to determine quickly whether $x^* \leq x$ for an arbitrary value x. We can answer this question in linear time by computing the maximum distance d from $(x, 0)$ to any point in S. Let I be the set of points that realize this distance, i.e. $I = \{p_i \mid (a_i - x)^2 + b^2 = d^2\}$. If every point in I is to the left (right) of x, so must be x^*. More specifically, we have the following implications: if for each $p_i \in I$, $a_i > x$ then $x^* > x$; if for each $p_i \in I$, $a_i < x$ then $x^* < x$. Otherwise, we have $x = x^*$.

2. Secondly, given two points p_i, p_j $(a_i < a_j)$, we would like to decide whether p_i or p_j can be discarded for good. Let z be the x-coordinate of the intersection of L with the bisector of (p_i, p_j) (we leave it to the reader to see what to do should this intersection be undefined). If it is known that $z < x^*$ (resp. $x^* < z$), then clearly p_j (resp. p_i) can be discarded.

Next we will see that with these two primitives it is possible to discard a fraction of the points in $O(n)$ operations. For each even value of i $(2 \leq i \leq n)$, compute z_i, the x-coordinate of the intersection of L with the bisector of the pair (p_{i-1}, p_i). If $a_{i-1} = a_i$, it is legitimate to discard whichever of p_i or p_{i-1} is closer to L; we then leave z_i undefined. Next, compute the median element z_h of the list of z_i's thus obtained in $O(m)$ steps, where m is the number of remaining points. Using Primitive 1, determine in $O(m)$ time whether x^* lies strictly to the left or strictly to the right of z_h. If neither is the case, we have $x^* = z_h$ and we are finished. Otherwise, Primitive 2 allows us to immediately discard at least one point for each pair z_i falling on one side of z_h. This leaves us with roughly $\leq 3m/4 \leq 3n/4$ points, which leads to an $O(n)$ running time for computing x^*. Next we ask a slightly different type of question.

II. *Given a line L, does the center of the smallest-area enclosing circle lie on L: if yes, where? If not, on which side of L does it lie?*

Using our solution to Problem I, it suffices to determine the sign of y_C, the y-coordinate of the center of C. Without invoking convexity results explicitly, we will show by a simple geometric argument how to determine the sign of y_C. Let r be the radius of C_L and let J be the set of points of S at a distance r from $(x^*, 0)$, i.e. $J = \{p_i \mid (a_i - x^*)^2 + b_i^2 = r^2\}$. If J has a unique element, $J = \{p_i\}$, then $a_i = x^*$ and y_C has the sign of b_i (always assuming that L is the x-axis). If J has at least two elements, one of them must be to the left of x^* and the other to its right, otherwise it would be possible to "improve" x^*. Let p_i and p_j be such that $p_i, p_j \in J$, $a_i \le x^* \le a_j$, and let D_i (resp. D_j) be the disk centered at p_i (resp. p_j) of radius r. The center of χ of C must lie in the intersection $D_i \cap D_j$. It is immediate to see that $K = D_i \cap D_j$ is free of any point of L besides $(x^*, 0)$, therefore K lies totally on one side of L. Which one it lies on can be determined in constant time once p_i and p_j are available. This directly solves our problem since χ lies in K.

In some instances, it is possible to determine χ immediately. To identify these cases, we will compute $K^* = \cap p_i \in J D_i$ instead of simply K. To do so, we consider each $p_i \in J$ in turn (in any order) and update the current intersection. Since all the disks have a common point, the intersection either is a single point or consists of two circular arcs, so it can be computed in constant time per disk, i.e. in $O(n)$ time. If K^* happens to be reduced to a single point then $\chi = (x^*, 0)$ and we are finished. Otherwise, K^* lies entirely on one side of L, since we have (in particular) $K^* \subseteq K$. Next we solve the last subproblem of our series, from which we will be able to conclude directly.

III. *Eliminate a fraction of the points in S on the grounds that they are irrelevant to the definition of the smallest-area circle enclosing S.*

Using our algorithm for problem II, we can carry out a binary search over the Euclidean plane and thus zoom arbitrarily close towards the center χ. Unfortunately, this will not indicate the points contributing the center, however close we get to χ. Following a pattern by now familiar, we carry out the binary search not over E^2 but over a discrete set, namely S itself. To realize this objective, we must supplement our set of primitives with the following one: given a convex polygon (not necessarily bounded) in which χ is known to lie, and given the bisector of the pair (p_i, p_j), discard either p_i or p_j. This can be easily done as long as the bisector does *not* intersect the convex polygon. Indeed, in that case, one of the points, say p_i, lies on the same side of the bisector as every point in the polygon. This implies in particular that χ is closer to p_i than to p_j, and so p_i can be discarded. We are now in a position to present the algorithm in its entirety.

To begin with, pair up the points of S in an arbitrary fashion, e.g. (p_1, p_2), (p_3, p_4), . . . , and let L_1 be the bisector of (p_1, p_2), L_2 the bisector of (p_3, p_4), etc. Consider the slopes (in $[-\infty, +\infty)$) of the $\lfloor n/2 \rfloor$ bisectors $L_1, \ldots, L_{\lfloor n/2 \rfloor}$, and in $O(n)$ time, compute the median slope, α. Next, pair up each bisector L_i with slope$(L_i) \le \alpha$ along with a distinct bisector L_j with slope $(L_j) \ge \alpha$. This gives us $\lfloor n/4 \rfloor$ pairs of the form (L_i, L_j). For convenience, let's re-orient the x-axis along the direction α; a line with slope α becomes a line with slope 0. For each pair (L_i, L_j), define the quantity y_{ij} as follows: if L_i and L_j have distinct slopes, let (x_{ij}, y_{ij}) be their intersection. Otherwise, let y_{ij} be the mean y-coordinate of L_i and L_j, i.e., the y-coordinate of the midpoint between $\{X = 0\} \cap L_i$ and $\{X = 0\} \cap L_j$. By convention, we let $y_{ij} = +\infty$ if both L_i and L_j are parallel to the

y-axis. At any rate, compute the median y of the $\lfloor n/4 \rfloor$ values of the form y_{ij} in $O(n)$ time. Using our solution to Problem II, we can decide in linear time whether χ lies on the line $Y = y$ (in which case we are finished), or lies above or below. Assume wlog that χ lies strictly below.

Next, consider all the $\lfloor n/8 \rfloor$ pairs (L_i, L_j) such that $y_{ij} \ge y$. If L_i and L_j are parallel, the intersection with $X = 0$ of one of them, say L_i, must lie strictly above $(0, y)$. Since L_i is then parallel to the x-axis, one of the two points defining the line can be discarded, namely, the one lying below L_i. Let n_1 be the number of points dropped this way and n_2 the number of remaining pairs ($n_1 + n_2 = \lfloor n/8 \rfloor$). Consider now the set of remaining pairs (with $y_{ij} \ge y$) and compute the median x of the x_{ij}'s. Test on which side of the line $X = x$ the center χ must lie. Once again if χ lies on the line, we are able to solve the problem directly. Assume wlog that χ lies to strictly to the left of the line (Fig. 4.14). For each of the $\lfloor n_2/2 \rfloor$ pairs (L_i, L_j) to the right of $X = x$, one of the four points involved can be dropped from further consideration. Indeed, for each pair (L_i, L_j), one of the lines, say L_i, is oriented in such a way that it is possible to identify with certainty on which side of L_i the center lies. To see this, L_i can always be chosen with negative slope (by construction, either L_i or L_j satisfies this property). Let p_k and p_l be the two points of which L_i is the bisector, with p_k (resp. p_l) below (resp. above) L_i. Since χ must lie in the quadrant delineated in Fig. 4.14, whatever the exact location it will always be closer to p_k than p_l, therefore p_k can be discarded. This leads to the dismissal to $n_1 + \lfloor n_2/2 \rfloor \ge \lfloor n/16 \rfloor$ points from S, hence the linear running time of the overall algorithm.

Megiddo's method, however appealing, cannot be recommended for practical applications. For one thing, its reliance on linear-time median computation may critically hamper its performance when applied to problems of modest size. But since the algorithm does not need the exact median but any value *close enough*, computing the median out of a small sample of the input might perform just as well, and give a "somewhat" simpler algorithm.

2.3.2. Ellipse Enclosure. After dealing with circles, the natural question is: how about ellipses? An ellipse is defined by five parameters whereas a circle

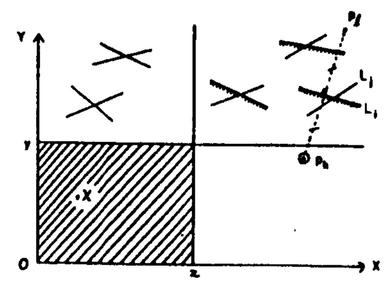


FIG. 4.14.

needs only three. These two additional degrees of freedom should in general allow for wrappings that fit more tightly around the set of points (Post, 1981; Silverman & Titterington, 1980).

What is the complexity of computing the smallest-area ellipse enclosing a set of points?

As is well-known, any ellipse in the plane centered at $u_0 = (x_0, y_0)$ can be analytically represented by an equation of the form $(u - u_0)^t A(u - u_0) = 1$, where A is a 2×2 positive-definite matrix. Since A is symmetric, five numbers suffice to define any ellipse and, as one should expect, at most five points uniquely define an ellipse. A simple algorithm consists of computing all candidate ellipses enclosing S. To so, we consider all ellipses passing through five points, and all smallest-area ellipses passing through four and three points respectively. In each case, we check whether the candidate ellipse contains every point of S, and throw it away if it does not. Elementary analysis shows that this algorithm will be extremely time-consuming, as it will require $O(n^6)$ operations.

Post (1981) has shown that it is possible to reduce the computation time to $O(n^2)$. The key to Post's method is to be able to throw away at least one point after a linear number of operations. The algorithm starts off with a spanning ellipse, checks whether it is the smallest, and if not, throws away at least one point and then shrinks the ellipse. Iteration takes over at this point after making sure that the new ellipse does enclose all the points of S. Additional work is necessary if this condition is not satisfied. The quadratic performance of the algorithm follows from the linearity of each pass.

2.4. Other Enclosure Problems

A large number of enclosure problems have been studied lately. We review some of the main results.

1. Given a set S of n points in the plane, consider all k-gons which can be formed with the points in S. Boyce, Dobkin, Drysdale, and Guibas (1982) have given an algorithm for finding the maximum perimeter triangle in $O(n\log n)$ time and the largest perimeter or area k-gon in $O(kn\log n + n\log^2 n)$ time for any k.
2. Given a simple n-gon P, find the largest-area convex polygon contained in P. This *potato-peeling* problem, as it is often called, has a number of variants, one of which calls for the maximum-perimeter enclosed polygon. Chang and Yap (1984) have proposed polynomial algorithms for all these problems.
3. Given a convex n-gon P, find a minimum-area convex k-gon enclosing P. Besides the aforementioned work of Klee and Laskowski (1985) and O'Rourke at al. (1984) for the case $k = 3$, a number of interesting results have been obtained. Using dynamic programming, Chang and Yap (1984) have described an $O(n^3\log k)$ time algorithm for the general problem. This bound has been improved to $O(n^2\log n\log k)$ (See Aggarwal, Chang, & Yap, 1985). De Pano and Aggarwal (1984) have considered special cases of this problem: they give an $O(n\log k)$ (resp. $O(nk^2)$) time algorithm for the case where the enclosing polygon is equi-angular (resp. regular). See also O'Rourke (1984) for extensions of the smallest enclosing box in three dimensions.

Some problems in location theory are very similar in spirit to the type of questions we've been asking so far. *Given a set of points and disks in the plane, is it possible to arrange the disks so as to cover all the points?* Megiddo and Supowit (1984) have shown that the problem is NP-hard. However, the problem: *Given a set of n points in the plane, what is the smallest disk that covers at least k points?* can be solved in $O(k^2 n\log n)$ time, as demonstrated by Lee (1982). Also, the question: *Given n points in the plane, what is the largest number of points that can be covered by a single disk?* has been shown to be solvable in $O(n^2)$ time in Chazelle and Lee (1986).

A useful primitive to have in applications involving enclosure problems checks whether a polygon can fit into another. *Given two polygons P and Q, determine whether Q can contain P, if rotations and translations are allowed.* Chazelle (1983b) has given a general $O(p^3q^3(p+q))$ time algorithm for deciding on this question, where p (resp. q) denotes the number of vertices in P (resp. Q). If Q is convex, the complexity can be reduced to $O(pq^2)$. The algorithm essentially involves finding "all" possible locations where P could fit. The difficulty is to establish an upper bound on the number of locations. Despite its high complexity, the algorithm is optimal with respect to that measure, if p is a constant. This still leaves open the question of whether the decision problem can be solved more efficiently.

A few results are known in the special case where only translations are allowed. Letting $n = p + q$, the problem can be solved in $O(n)$ time if both polygons are convex (Chazelle (1983b). For the case where either the inside polygon is convex or both polygons are rectilinearly convex, Baker, Fortune, and Mahaney (1984) have given an $O(n^2\log n)$ time algorithm.

3. DECOMPOSITION OF SHAPES

3.1. Triangulations

Let P be a simple *n-gon*, ie., a simple polygon with n vertices. A *triangulation* of P is any partition of the polygon into disjoint triangles, all of whose vertices are vertices of the polygon. It is often useful to have a triangulation of P. For one thing, triangles are easy to handle, so a partition of this nature simplifies tasks such as testing for intersection, inclusion, etc. The idea is to perform computations iteratively on each part; for example, one can use this "decomposition" approach to detect interference or collision between a number of moving objects (see Ahuja, Chien, Yen, & Birdwell (1980) for example). Another benefit to gain from a triangulation is valuable information concerning topological properties of the polygon (Chazelle, 1982).

Triangulation problems are many and varied. In numerical analysis, for instance, a triangulation is often used to evaluate a function of two variables by interpolation or to integrate a function by the finite-element method. In these applications, the shape of the triangulation can be of great importance. In the context of the finite-element method, for example, it is likely that long, skinny triangles should be avoided because of the errors they tend to generate (see, for example, Baker, Gross, & Rafferty, 1985). We will not concern ourselves with such matters, however, since their relevance is relatively small if a triangulation is viewed solely as a tool for facilitating geometric computations. This is most often the case in robotics, computer graphics, computer animation, CAD/CAM, etc. As a starter, we pose the following problem:

What is the complexity of computing a triangulation of a simple n-gon?

Unfortunately, despite the large amount of work devoted to this problem, a definite answer still eludes us. There have been solid advances on the subject, however, both on the practical and theoretical sides. Garey, Johnson, Preparata, and Tarjan proposed the first $O(n\log n)$ time algorithm for triangulating P (Garey et al., 1978). Their algorithm works in two phases: to begin with, a *regularization* procedure is applied to P, which in $O(n\log n)$ time produces a partition of the polygon into *L-monotone* pieces, for some line L. A polygon is said to be L-monotone if any normal to L intersects the polygon in no more than one segment. With this decomposition in hand, Garey et al.'s algorithm completes the triangulation by cutting each monotone piece into triangles. This second (and last) pass takes linear time.

Chazelle (1982) has established a *polygon-cutting theorem*, which is useful for triangulation as well as other problems (e.g., visibility, internal distance). Assign a positive weight to each vertex of P in an arbitrary fashion, with the total weight not exceeding 1. Roughly speaking, the polygon-cutting theorem states that there exists two vertices a, b such that the segment ab lies entirely inside the polygon P and partitions it into two polygons, neither of whose weights exceeds 2/3.

If we allow additional vertices to be introduced, then a successful approach to the triangulation problem is the *line-sweep* paradigm (Bentley & Ottmann, 1979; Shamos, 1978). Imagine a vertical line L sweeping the plane from left to right; the segments of the intersection $L \cap P$ are kept dynamically in a balanced search structure (e.g., AVL, Knuth, 1973), red-black, Sedgewick, 1983, trees). An *event* takes place every time the line encounters a new vertex. At that point, the search structure is updated to record the fact that a segment is about to vanish, to

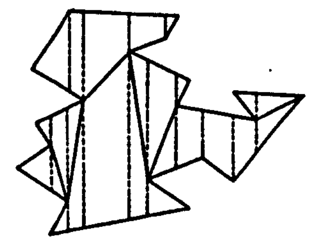


FIG. 4.15.

be created, or to have one of its endpoints switch to a different edge of P (Fig. 4.15). The net result of this line-sweep process is the so-called *vertical decomposition* of P, a set of vertical segments obtained by connecting every vertex of P to the edge immediately above or below it, in such a way that the segment created lies inside P (dashed lines in Fig. 4.15). It is easy to derive a triangulation of P from its vertical decomposition (note that the triangulation is not uniquely defined, but the vertical decomposition is). In a nutshell, the idea is to identify the *cusps* of P, i.e., the vertices that are locally in leftmost or rightmost position. For each trapezoid generated by the line-sweep procedure, connect each cusp to a vertex of the other vertical side (solid interior lines in Fig. 4.15). Removing all vertical segments produces a decomposition of P into monotone polygons (with respect to the horizontal direction). The linear time postprocessing of Garey et al.'s algorithm can then be applied to complete the triangulation.

Using a fairly similar method, Hertel and Mehlhorn (1983) succeeded in lowering the $O(n\log n)$ running time to $O(n + r\log r)$, where r is the number of reflex angles in P. Their method differs from the previous one in two important aspects. First, it performs the triangulation on the fly, i.e., it computes the triangles during the line-sweep process. Second, it skips over vertices that do not display reflex angles. The import of Hertel and Mehlhorn's result is to express the running time of the algorithm not only as a function of the input size but also as a function of a parameter which reflects a morphological property of the polygon.

In the same spirit, Chazelle and Incerpi (1984) developed a radically different algorithm based on divide-and-conquer. Their method requires $O(n\log s)$ time, with $s < n$. 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 (e.g., $s = 2$ in the case of Fig. 4.15), even for extremely winding polygons. The running time of the algorithm depends primarily 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. Aside from the notion of sinuosity, Chazelle and Incerpi have also characterized a large class of polygons for which the algorithm can be proven to run in $O(n\log\log n)$ time. Implementation of the algorithm has confirmed its theoretical claim to efficiency.

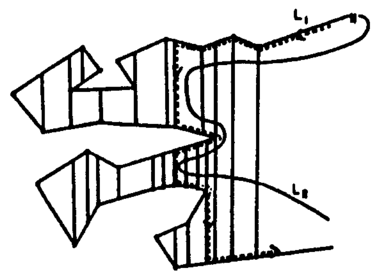


FIG. 4.16.

Briefly, the algorithm starts with the observation that vertical decompositions can be defined with respect to the boundary of P as opposed to the polygonal region P. This distinction enables a divide-and-conquer strategy involving (1) the computation of the vertical decompositions of $\{p_i, \ldots, p_{\lfloor \frac{i+j}{2} \rfloor}\}$ and $\{p_{\lfloor \frac{i+j}{2} \rfloor}, \ldots, p_j\}$; (2) the merge of these decompositions into the vertical decomposition of $\{p_i, \ldots, p_j\}$. The key to efficiency is to take *shortcuts* in the execution of the merge. Figure 4.16 illustrates this notion of shortcuts. L_1 and L_2 are the two polygonal lines whose vertical decompositions are being merged. The algorithm traverses the trapezoids of both decompositions, but visits *only* those that will be modified by the merge. For example, the trapezoids on the left-hand side of Fig. 4.16 will not be examined. The dashed line traces the steps of the merge (see Chazelle and Incerpi, 1985, for details).

3.2. Decompositions into Special Shapes

Among primitive shapes of particular interest, we also have quadrilaterals and convex or *star-shaped* polygons. We say that a polygon P is star-shaped if it contains at least one point p such that for every point q inside P the segment pq lies inside P. Sack and Toussaint (1981) have shown that any isothetic polygon with n vertices which is star-shaped can be partitioned into convex quadrilaterals in $O(n)$ time. Recall that a polygon is isothetic (also sometimes referred to as *rectilinear*) if its sides are parallel to two orthogonal axes. See Chazelle (1980, 1982), Chazelle and Dobkin (1985), Greene (1983), Hertel and Mehlhorn (1983) for examples of algorithms for decomposing a polygon into a number of convex polygons within a constant factor from the minimum number.

Kahn, Klawe, and Kleitman (1980) have shown that any isothetic polygon can always be partitioned into convex quadrilaterals. Building on this result, Sack (1982) has given an $O(n \log n)$ time algorithm for computing the decomposition. In a nutshell, the algorithm involves decomposing the polygon into monotone polygons, and then decomposing each of these polygons into quadrilaterals.

Avis and Toussaint (1981) have obtained similar results concerning star-shaped polygons. They have shown that any n-gon can be partitioned into star-shaped polygons in $O(n \log n)$ time.

3.3. Minimum Decompositions

Problem OCD (Optimal Convex Decomposition): Given a simple polygon P, which is the minimum number of convex polygons which form a partition of P?

Let's briefly review the main results relating to this problem. Pioneering work appears in Feng and Pavlidis (1975) and Schachter (1978). with the design of heuristics for computing decompositions of shapes into a number (not necessarily minimum) of convex pieces.

Minimum Partitioning: One of the earliest results concerning the OCD problem per se was obtained by Chazelle and Dobkin (1979), who showed that the OCD problem was solvable in polynomial time (see an example of optimal decomposition in Fig. 4.18). Interestingly enough, this finding was followed by a stream of NP-hardness results for similar problems. For example, it was shown by Lingas (1982) that the presence of holes in the polygon P was sufficient to make the OCD problem NP-hard. Lingas also showed that a minimum decomposition into triangles was NP-hard. Other variants have been shown to be (most likely) intractable: Once again, assuming that P may have holes, Asano and Asano (1983) proved that decomposing P into a minimum number of trapezoids with two vertical sides is NP-hard. If holes are disallowed, however, they showed that the problem could be solved in $O(n^3)$ time, an upper bound later improved by Asano, Asano, and Imai (1984) to $O(n^2)$.

Minimum Covering: If now instead of partitioning the polygon, one would rather *cover* it with possibly overlapping convex pieces, it might be possible to save a few pieces. For example, a cross can be covered with two rectangles but cannot be partitioned into fewer than three convex polygons. Unfortunately, minimum coverings seem very difficult to compute. O'Rourke and Supowit (1983) have shown that this problem and a large number of its variants are NP-hard. A decision procedure for this problem is given in O'Rourke (1982). If the polygon is isothetic and the covering is made of rectangles, then a quadratic algorithm can be used (Franzblau & Kleitman, 1984).

No Steiner Points: Another class of decomposition problems stipulates that no new vertices should be introduced in the decomposition. These new vertices, called *Steiner points* are often useful—as we will shortly see—in reducing the number of pieces. Steiner points are sometimes undesirable, however. Once again, the presence of holes makes most of the problems NP-hard (see Keil, 1983; Lingas, 1982; O'Rourke & Supowit, 1983). If holes are disallowed, however, the situation is much brighter. Let c be the number of reflex angles in P; Greene (1983) has given an $O(n^2c^2)$ algorithm for decomposing P into a minimum number of convex pieces (not allowing Steiner points), and Keil (1983) has described an $O(c^2 n\log n)$ algorithm for the same problem.

Partitioning into Special Shapes: A decomposition problem of practical interest concerns the partition of an isothetic polygon into a minimum number of rectangles. Lipski (1983) and Imai and Asano (1983a, 1983b), have shown that the problem can be solved in $O(n^{3/2}\log n \log\log n)$ and $O(n^{3/2}\log n)$ time, respectively, by using a reduction to maximum matching of a particular bipartite intersection graph. In both cases, $O(n\log n)$ storage is required. Keil (1983) has shown that it is possible to partition any simple polygon into a minimum number of star-shaped polygons in $O(n^5 c^2 \log n)$ time, where c once again is the number of reflex angles in P. For other work, consult Ferrari, Sankar, Sklansky (1981), Keil and Sack (1985), Toussaint (1980b).

Minimum-Length Partitioning: Other objective functions besides the cardinality of the decomposition have been examined. Lingas (1981) and Lingas, Pinter, Rivest, and Shamir (1982) consider decomposition problems where the amount of "ink" used, i.e., the total length of the added edges is to be minimized.

Higher Dimensions: In three dimensions, the OCD problem becomes NP-hard (Lingas, 1982). A decision procedure based on a combinatorial description of the complex formed by a convex decomposition is given in Chazelle (1983a). Chazelle (1984) has established an $\Omega(n^2)$ lower bound on the time complexity of the problem. If n is the number of vertices in the polyhedron and c is the number of edges displaying reflex angles, in $O(nc^3)$ time it is possible to decompose the polyhedron into a number of pieces which, in the worst case, is minimal up to within a constant factor.

The OCD Problem: We now return to the original OCD problem: Given a simple n-gon P compute a minimum partition of P into convex pieces, using Steiner points if necessary. The remaining of this section is devoted to giving the technical basis of the polynomial time algorithms given in Chazelle (1980), Chazelle and Dobkin (1985). For details, we again refer the reader to the appropriate sources. In the following, a *notch* refers to a vertex of P with a reflex interior angle (e.g., a convex polygon has no notch). We let $\{p_1, \ldots, p_n\}$ be a vertex-list of P in clockwise order, and we assume that c of these n vertices are notches of the polygon. Chazelle and Dobkin have given several algorithms for computing an OCD (optimal convex decomposition) of P, the most efficient of which runs in $O(n + c^3)$ time (Chazelle, 1980; Chazelle & Dobkin, 1985). All the algorithms use two simple observations. First, each notch *can* be removed by the addition of a polygon to the decomposition. Second, *at most* two notches can be removed through the addition of a single polygon. It follows that the minimum number of convex parts always lies between $\lceil c/2 \rceil + 1$ and $c + 1$.

Below, we introduce the notion of *X-pattern*, the essential tool for generating minimum decompositions. An X_k-pattern is a particular interconnection of k notches which removes all reflex angles at the k notches and creates no new

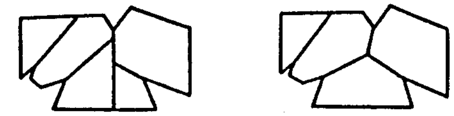


FIG. 4.17.

notches. A decomposition obtained by applying p patterns of type $X_{i_1}, \ldots X_{i_p}$ along with straight-line segments to remove the remaining notches can be shown to yield $c + 1 - p$ convex parts. It follows that the decompositions using the most X-patterns will at the same time minimize the number of convex pieces.

The next question to tackle is whether a given set of notches can be interconnected via an X-pattern. One solution around the multiplicity of combinatorially distinct X-patterns is to constrain the patterns in such a way that their detection becomes tractable. This leads to the introduction of Y-patterns, which we regard as X-patterns endowed with some structural property. A key property of minimum decompositions is that with the exception of X_4-patterns any X-pattern can be advantageously replaced by a Y-pattern. Since, as we will sketch out shortly, Y-patterns can be constructed in polynomial time via dynamic programming, we can prove in this way that th$_s$ OCD problem is in the polynomial class. Further geometric analysis leads to substantial gains in the efficiency of the algorithm, but we will not elaborate on these improvements which are fairly intricate.

As a starter, we define the notion of *naive decomposition*. This is the decomposition obtained by removing each notch in turn by means of a simple line segment *naively* drawn from the notch. Figure 4.17 shows a naive decomposition of a nonconvex polygon next to an improved decomposition. To obtain a naive decomposition of P is easy: consider each notch $v_1, \ldots, v_c$ in turn, and extend a line segment from v_i in order to remove the reflex angle at the notch. Stop the line as soon as it hits another line already in the decomposition. Trivially, the naive decomposition of P produces exactly $c + 1$ convex parts. Next, we define a *nice* class of decompositions to which we may restrict our attention from then on. We say that a polygon is *interior* to P if it lies inside P and at most a finite number of its points lie on the boundary of P. An *X-decomposition* is then defined as any convex decomposition containing no interior polygons and such that no vertex is of degree greater than 3, except for the notches, which may be of degree at most 4. An important result, whose proof we will omit, states that the class of X-decompositions always contains an OCD.

We are now ready to introduce the notion of *X-pattern*. Consider the planar graph formed by the added edges of an X-decomposition. This graph is a forest of trees, where each node is of degree 1 or 3, except for notches which may be of degree 1 or 2. Note that it is a forest because we disallow interior polygons in the decompositions. A straightline embedding of a tree lying inside P (i.e., with no self-intersections) is called an X-pattern if

1. all vertices are of degree 1, 2 or 3.

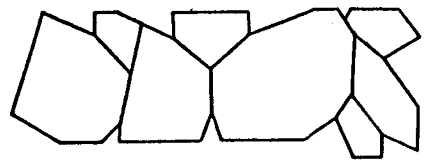


FIG. 4.18.

2. any vertex of degree 1 or 2 coincides with a notch of P, and its (1 or 2) adjacent edges remove its reflex angle.
3. none of the 3 angles around any vertex of degree 3 exceeds 180 degrees.

An X-pattern with k vertices of degree 1 or 2 is called an X_k-pattern. Informally, an X_k-pattern is an interconnection of k notches used to remove them while introducing $k - 1$ additional polygons into the decomposition. Figure 4.18 gives an example of an X-decomposition with one X_3-pattern and one X_4-pattern. Note that the leftmost tree is not an X-pattern because one of its vertices of degree 1 does not coincide with a notch of the polygon. In the following, the vertices of an X-pattern (regarded as tree vertices) of degree 1 (resp. 2,3) will be called N1-nodes (resp. N2, N3-nodes). We justify the introduction of X-patterns with the following observation. An X-decomposition with p X-patterns has at least $c + 1 - p$ convex parts. This suggests that using p X-patterns saves at most p polygons over the naive decomposition. This leads to the definition of *compatible X-patterns*. A set of X-patterns is said to be compatible, if no pair of edges taken from two distinct patterns intersect. It is not hard to show that any compatible set of p X-patterns can be used to produce an X-decomposition with exactly $c + 1 - p$ convex parts. The OCD problem can now be expressed as a generalized matching problem.

Let p be the maximum number of compatible X-patterns. An OCD can be obtained by first applying the p X-patterns and then applying the naive decomposition to any remaining non-convex polygon. Unfortunately the complexity of this approach is prohibitive, given the excessive number of candidates we might have to consider in the process. X-patterns allow Steiner points (i.e., vertices not on the boundary of P) to be adjacent. Looking at any X-pattern as a mechanical system of extendible arms and joints, it appears to be greatly underconstrained. We can show that X-patterns can be in general *reduced* to maximally constrained X-patterns, called Y-patterns. The next step is then to prove that Y-patterns can be computed in polynomial time. Roughly speaking, a Y_k-pattern is an X_k-pattern such that no edge joins two Steiner points. For example, the X_4-pattern of Fig. 4.18 violates this restriction. The utility of Y-patterns comes from the fact (which we will not prove) that all X_k-patterns ($k \neq 4$) can be transformed into Y-patterns. This allows us to limit attention to X-decompositions with only Y and X_4-patterns. The transformations, called *reductions*, involve the stretching, shrinking, and rotating of lines in the original pattern. The next question which we must be able to answer efficiently is:

Given k notches, does there exist an X-pattern connecting them?

Let v be an N3-node of a Y-pattern. Removing the three edges vv_i, vv_j, vv_k adjacent to v disconnects the Y-pattern into three subtrees. Note that the removed edges play the role of an X_3-pattern with respect to the three subtrees. This leads us to introduce the notion of *extended* X-pattern. An extended X_l-pattern ($l = 2, 3$) is either an X_l-pattern by itself or an X_l-pattern formed by a subgraph of an X_m-pattern ($m > l$). It is clear that an algorithm for testing the possibility of an X_l-pattern between a given set of notches can also be used to determine the possibility of an extended X_l-pattern, as long as the angles formed at the notches by the subtrees of the X_m-pattern are known in advance.

To see this, we must define the notion of *extended notch*. Let v_i be a notch of the extended X_l-pattern, and let W be any wedge centered at v_i. The extended notch at v_i with respect to W refers to v_i along with the angular specifications given by W. This means that the X-pattern edge emanating from the extended notch v_i must lie in the intersection of W with the visibility-polygon at v_i (El Gindy & Avis, 1981). In general the wedge W will be taken as the locus of rays (i.e., half-lines) which emanate from v_i and remove the reflex angle at v_i. When dealing with ordinary X-patterns, the wedge W is simply determined by the edges of P adjacent to v_i. In the case of extended patterns, however, the wedge W will take into account the other edges already adjacent to v_i, and will thus be wider than in the previous case. Sometimes, we will also consider cases where the wedge W is taken as the entire plane. This is done if we do not wish to remove a reflex angle at a particular notch. We can show that in all cases it is possible to check for the existence of an (extended) X_l-pattern between l given notches in polynomial time (for $l \leq 4$).

We can now turn to the decomposition algorithm, which as we mentioned earlier is based on dynamic programming. We rely on the observation that if a certain X_k-pattern belongs to an OCD of P, it decomposes P into k subpolygons, $P_1, \ldots, P_k$, so that finding an OCD for each P_i yields an OCD for P. We compute maximal compatible sets of patterns for each P_i. Since the notches of P_i are also notches of P, any X-pattern of P_i is also an X-pattern of P. Conversely, we want to show that any X-pattern of P involving only notches in P_i is also an X-pattern of P_i. This is crucial since dynamic programming proceeds bottom-up. Indeed, the algorithm will always compute maximal sets of compatible patterns involving notches of P_i before it even knows the shape of P_i, i.e. it will rely solely on the notches of P_i and not on its boundary.

To check this point, we define $V(i, j)$ as the set of notches between v_i and v_j in clockwise order. We have $V(i, j) = \{v_i, v_{i+1}, \ldots, v_j\}$, with index arithmetic taken modulo c. Let $z_1, \ldots, z_k$ be the notches of an X-pattern, T, given in clockwise order around the boundary of P, and let $V(i, j)$ be the notches of P between z_u and z_{u+1} in clockwise order ($z_u = v_{i-1}$, $z_{u+1} = v_{j+1}$). It is possible to prove that no X-pattern with its notches in $V(i, j)$ can intersect T. This independence result can be understood in combinatorial terms. It states that two X-patterns are intersection-free if and only if their notch sequences are not intermixed, i.e. one sequence falls completely between two consecutive elements of the other sequence.

Next, we define $S(i, j)$, for every pair of notches v_i, v_j, as a maximum compatible set of X_4 or Y-patterns in $V(i, j)$. The goal is to evaluate $S(1, c)$, which we achieve by computing all values $S(i, j)$ from $\{s(k, l) \mid V(k, l) \subset V(i, j)\}$. This can be done directly if v_i and v_j are not to be connected to the same pattern. For this case, we test all combinations $\{S(i, k), S(k + 1, j)\}$, for each $v_k \in V(i, j - l)$. In the event where v_i and v_j should be connected together, we consider the possibility of an X_4 or a Y-pattern between the two notches.

To handle the latter case, we compute all candidate Y-patterns via dynamic programming. We compute Y-subtrees (i.e., subtrees of Y-patterns) as well as Y-patterns by patching Y-subtrees together. A Y-subtree is considered *not* to be a candidate if at the time it is computed we are ensured of the existence of at least one OCD which does not use this Y-subtree and satisfies previous constraints. As a shorthand we say that a pattern or a Y-subtree lies in $V(i, j)$ if all its notches do. We next give a brief description of the polynomial time algorithm.

Consider a Y-pattern of an OCD with at least one N2-node, v_i. This node splits the Y-pattern into two Y-subtrees, so there exists an index j such that

1. One of the Y-subtrees lies in $V(i, j)$ whereas the other lies in $V(j + 1, i)$.
2. All the other patterns in the OCD lie totally either in $V(i, j)$ or in $V(j + 1, i)$.

The idea is to examine the candidacy of the Y-subtree in $V(i, j)$ immediately after $S(i, j)$ has been computed. Let $v_i, v_{i_1}, \ldots, v_{i_m}$ be a clockwise-order list of the notches in the Y-subtree. The candidacy of this subtree may be rejected if the following equality is not satisfied:

$$|S(i,j)| = |S(i + 1, i_1 - 1)| + \ldots + |S(i_{m-1} + 1, i_m - 1)| + |S(i_m + 1, j)|. \tag{1}$$

$|S(k,l)|$ represents the number of patterns in $S(k, l)$, and the last term of the right-hand side is to be ignored if $i_m = j$. If Relation (1) is not satisfied, it is clear that the right-hand side is strictly smaller than $|S(i, j)|$. Since we only consider the presence of a single pattern connecting notches in both $V(i, j)$ and $V(j + 1, i)$, we can dismiss any Y-subtree that does not satisfy (1). Indeed, using the patterns of $S(i, j)$ would yield a smaller decomposition.

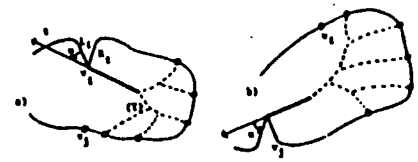


FIG. 4.19.

Another crucial fact will allow us to reduce the number of candidates to keep around. Let L_i (resp. R_i) denote the edge of P adjacent to v_i and preceding (resp. following) v_i in clockwise order. Both L_i and R_i are understood as *directed* outwards with respect to v_i. Let t be the edge adjacent to v_i in the Y-subtree lying in $V(i, j)$. We refer to t as the *arm* of the Y-subtree. When the arm of a Y-subtree enters the expression of an angle, we assume that it is directed towards the notch (here t is directed towards v_i). Among all the Y-subtrees in $V(i, j)$ for which v_i is an N2-node, (1) is true and $u = \angle(L_i, t) < 180$, we may keep the Y-subtree T which minimizes the angle u as the only candidate with respect to $V(i, j)$ (Fig. 4.19a). We then define $B(i, j)$ as a pointer to the arm of T. If no such subtree can be found, $B(i, j)$ is set to 0. The same reasoning applied counterclockwise with respect to $V(i, j)$ (v_i is now an N2-node), leads to $F(i, j)$, defined in a similar fashion (Fig. 4.19b).

We are now ready to present the decomposition algorithm. We define the function $\langle ARG \rangle$ for the purpose of assembling Y-subtrees in the computation of $S(i, j)$. The argument ARG is in general a pair of Y-subtrees in $B(u, v)$ or $F(u, v)$. If the two subtrees can be patched together and form a Y-pattern T, the function $\langle\rangle$ returns (C, T), where C is the maximum number of compatible patterns which can be applied in $V(i, j)$ including T. We discuss the implementation of this function after describing the main algorithm.

Initialization of the data structures is performed in STEP 1, followed by two nested loops setting up the framework for the dynamic programming scheme. Each stage corresponds to the computation of $S(i, j)$ for a given value of i and j. STEP 2 computes the best Y-pattern which connects v_i and v_j, by patching together precomputed candidate Y-subtrees. STEP 3 computes a maximum set of compatible patterns in $V(i, j)$, denoted L, assuming that v_i and v_j do not belong to the same pattern. It computes M, defined similarly, with the only difference that the presence of an X_4-pattern connecting v_i and v_j is now allowed. Finally the Y-pattern of STEP 2 (if any) is used to compute N, and the maximal set among L, M, N is chosen as $S(i, j)$. STEP 4 computes the Y-subtrees which lie in $V(i, j)$ and are considered candidates. These subtrees are to be used in further iterations through STEP 2. Once a maximum compatible set of patterns for P has been found, STEP 5 is able to complete the OCD with the naive decomposition.
NOTE: ***The following pieces of pseudo-code are reproduced from Chazelle and Dobkin (1985) with the permission of the authors.***

Procedure *ConvDec(P)*

beginprocedure

STEP 1:

The preprocessing involves checking that P is simple and nonconvex. We make a list of the notches $v_1, \ldots, v_c$, and we initialize all $B(i, i)$ and $F(i, i)$ to 0.

```
for d = 1, . . . ,c − 1
   for i = 1, . . . , c
      j:= i + d [mod c]
      do STEPS 2, 3, 4
```

STEP 2:

Compute the best Y-pattern connecting v_i and v_j as follows:
For each k; $v_k \in V(i+1, j-1)$, compute the set $Q = \bigcup_{1 \le i \le 4} Q_i$, where
$Q_1 = \{\langle F(i, k), B(k, j)\rangle\}$ /* N2-node on path */
$Q_2 = \{\langle B(i, k-1), F(k, j)\rangle\} \cup \{\langle B(i, j-1), F(j, j)\rangle\}$ /* no N2 or N3 nodes on path */
$Q_3 = \{\langle B(i, k-1), B(k, j-1)\rangle\}$ /* N3-node on path */
$Q_4 = \{\langle F(i+1, k), F(k+1, j)\rangle\}$ /* N3-node on path*/
The elements of Q are pairs of the form (C, T). Let T be the Y-pattern which has the maximum C value in Q.

STEP 3:

Let $S(i, j)$ be the maximum of L, M, N with respect to cardinality, where (max is taken with respect to cardinality)

$L = \max_{v_k \in V(i,j-1)} \{S(i, k) \cup S(k+1, j)\}$
/* corresponds to a patching together of Y-patterns */
$M = \max \{\{x_{i,a,b,j}\} \cup S(i+1, a-1) \cup S(a+1, b-1) \cup S(b+1, j-1)\}$
for all X_4-patterns $x_{i,a,b,j}$ connecting v_i, v_a, v_b, v_j, with $v_a, v_b \in V(i, j)$.
/*corresponds to the use of an X_4-pattern */
$N = \{\text{the } Y\text{-pattern } T \text{ of STEP 2}\} \cup S(i+1, i_1-1) \cup \ldots \cup S(i_{p-1}+1, i_p-1) \cup S(i_p+1, j-1)$
where $v_i, v_{i_1}, \ldots, v_{i_p}, v_j$ are the notches of T in clockwise order.
/* corresponds to the use of the Y-pattern T */

STEP 4:

Compute $B(i, j)$ and $F(i, j)$.

STEP 5:

Finish off the decomposition using the naive decomposition, i.e. adding one polygon for each remaining notch.

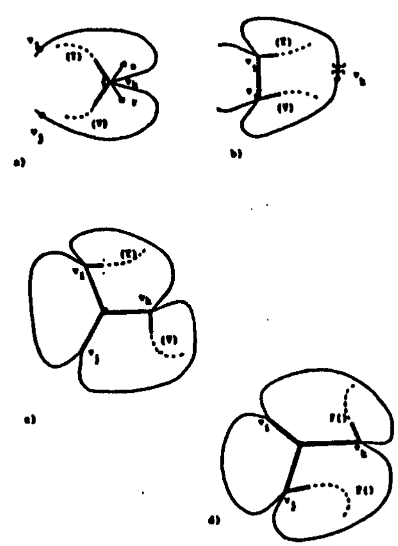


FIG. 4.20.

endprocedure

In the following we explore some of the primitive operations used by the algorithm in greater detail.

1. *Patching Y-subtrees* (STEP 2)

The function $\langle ARG\rangle$ takes two Y-subtrees and constructs a Y-pattern if these two subtrees can be patched together. ARG is any argument of the form: $(F(i, k), B(k, j))$, $(B(i, k-1), F(k, j))$, $(B(i, k-1), B(k, j-1))$ or $(F(i+1, k), F(k+1, j))$, with v_i, v_k, v_j occurring in clockwise order.

Case 1. $\langle F(i, k), B(k, j)\rangle$ (Fig. 4.20a).
Let $F(i, k) = T$ and $B(k, j) = V$, with r and s their respective arms. If $\angle(r, s) < 180$ and $T \ne 0$ and $V \ne 0$, then set $\langle F(i, k), B(k, j)\rangle = (|S(i, k)| + |S(k, j)| + 1$, Y-pattern: $T \cup V)$, else $\langle F(i, k), B(k, j)\rangle = 0$.

Case 2. $\langle B(i, k-1), F(k, j)\rangle$ (Fig. 4.20b).
Let $B(i, k-1) = T$ and $F(k, j) = V$. If an extended X_2-pattern is possible between v_i and v_j then set $\langle B(i, k-1), F(k, j)\rangle = (|S(i, k-1)| + |S(k, j)| + 1$, Y-pattern: $\{v_iv_j\} \cup T \cup V)$, else $\langle B(i, k-1), F(k, j)\rangle = 0$.

Case 3. $\langle B(i, k-1), B(k, j-1)\rangle$ (Fig. 4.20c).
Let $B(i, k-1) = T$ and $B(k, j-1) = V$. If an extended X_3-pattern W is possible between v_i, v_j, v_k, then
$\langle B(i, k-1), B(k, j-1)\rangle = (|S(i, k-1)| + |S(k, j-1)| + 1, T \cup V \cup W)$, else $\langle B(i, k-1), B(k, j-1)\rangle = 0$.

Case 4. $\langle F(i+1, k), F(k+1, j)\rangle$ (Fig. 4.20d).

Let $F(i+1, k) = T$ and $F(k+1, j) = V$. If an extended X_3-pattern W is possible between v_i, v_j, v_k, then

$\langle F(i+1, k), F(k+1, j)\rangle = (|S(i+1, k)| + |S(k+1, j)| + 1, T \cup V \cup W)$, else

$\langle F(i+1, k), F(k+1, j)\rangle = 0$.

Because of Relation (1), it is clear that STEP 2 computes the Y-pattern connecting v_i and v_j (if any is to be found) such that the number of compatible patterns which can be applied in $V(i, j)$ is maximum. All we have to check is that all cases are indeed handled in STEP 2. Consider the path from v_i to v_j in any such Y-pattern. If it contains an N2-node, it will be detected in Q_1. Otherwise one N3-node may appear on this path and all these candidates will be reported in Q_3 and Q_4. The final case, handled by Q_2, assumes that the path from v_i to v_j is free of N2 and N3-nodes.

2. *Computing* $S(i, j)$ (STEP 3)

Assume by induction that $S(k, l)$ has been computed for all v_k, $v_l \in V(i, j)$ (except for $S(i, j)$). The algorithm investigates the three following cases in turn:

1. Disallow the presence of any pattern having both v_i and v_j as vertices.
2. Consider the possibility of an X_4-pattern connecting v_i and v_j.
3. Consider the possibility of a Y-pattern connecting v_i and v_j.

3. *Constructing Y-subtrees* (STEP 4)

We compute $B(i, j)$ and $F(i, j)$ by iteratively patching Y-subtrees together via two functions, $Y(i, ARG)$ and $Y'(i, ARG)$. ARG is an argument of the form $B(a, b)$ or $(B(a, b), B(c, d))$ (or the same with F). We describe these functions with respect to B's only, all other cases being similar.

Case 1. $Y(i, B(a, b))$ (Fig. 4.21a)

The vertices v_a, v_b, v_i occur in clockwise order. Let $T = B(a, b)$. Extend the notch at v_a to take into account the arm of T. Extend the notch at v_i by making its associated wedge be the entire plane. If an extended X_2-pattern is possible between v_a, v_i and if $w = \angle(R_i, v_iv_a) < 180$, set $Y(i, B(a, b)) = (Y\text{-subtree}: \{v_iv_a\} \cup T)$, else $Y(i, B(a, b)) = 0$.

Case 2. $Y(i, (B(a, b), B(c, d))$ (Fig. 4.21b)

The vertices v_a, v_b, v_c, v_d, v_i occur in clockwise order. Let $T = B(a, b)$ and $V = B(c, d)$. Extend the notch at v_a (resp. v_c) to take into account the arm of T (resp. V). Extend the notch at v_i by making its associated wedge be the entire plane. If an extended X_3-pattern is possible between v_a, v_c, v_i, compute the locus of its N3-node. Let S be the point in the locus which maximizes the angle $w = \angle(R_i, v_iS)$.

If $w \in 180$, set $Y(i, (B(a, b), B(c, d)) = (Y\text{-subtree}:\{Sv_i\} \cup \{Sv_a\} \cup \{Sv_c\} \cup T \cup V)$, else $Y(i, (B(a, b), B(c, d)) = 0$.

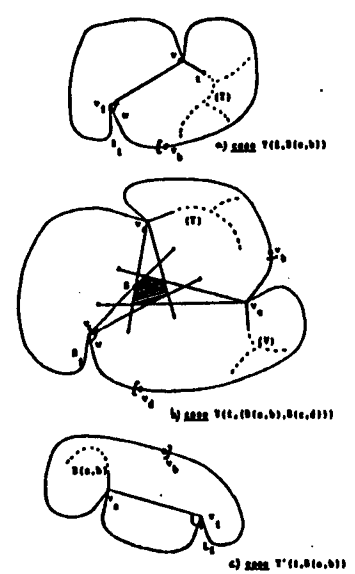


FIG. 4.21.

Case 3. $Y'(i, B(a, b))$ (Fig. 4.21c)

We define $Y'(i, B(a, b))$ in the same way as $Y(i, B(a, b))$, with $\angle(R_i, v_iv_a)$ replaced by $\angle(v_iv_a, L_i)$. $Y'(i, F(a, b))$ is defined similarly, so we omit the details.

We are now ready to implement STEP 4 of the decomposition algorithm. We will only describe the computation of $B(i, j)$, since the case of $F(i, j)$ is strictly similar. We begin by computing the four sets B_1, B_2, B_3, B_4. Let C be the value of $|S(i, j)|$ computed in Step 3.

$B_1 = \{Y\text{-subtree of } B(i, k)\}$, for all $v_k \in V(i, j-1)$ such that $|S(i, k)| + |S(k+1, j)| = C$.

/* v_j is not a notch of the Y-subtree */

$B_2 = \{Y'(i, B(k, j))\}$, for all $v_k \in V(i+1, j-1)$ such that $|S(i+1, k-1)| + |S(k, j)| = C$.

/* v_i's neighbor is an N2-node */

$B_3 = \{Y'(i, F(i+1, j))\}$, if $|S(i+1, j)| = C$.

/* v_i's neighbor is an N2-node */

$B_4 = \{Y'(i, (F(i+1, k), F(k+1, j)))\}$, for all $v_k \in V(i+1, j-1)$ such that $|S(i+1, k)| + |S(k+1, j)| = C$.

/* v_i's neighbor is an N3-node */

Let T be the Y-subtree of $B_1 \cup B_2 \cup B_3 \cup B_4$ which maximizes the angle $u = \angle(t, L_i)$, where t is understood here as the arm of T directed outward from v_i (Fig. 4.22). We define $B(i, j)$ as a pointer to the arm of T (now understood to be directed towards v_i).

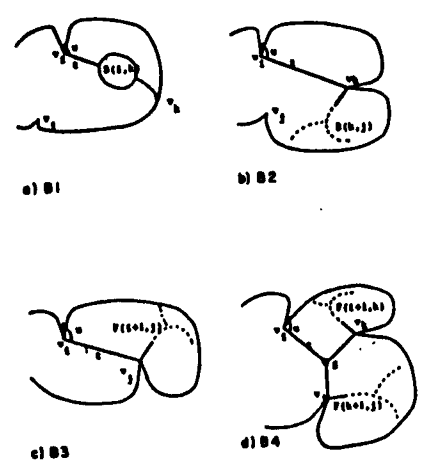


FIG. 4.22.

It follows from previous remarks that $B(i, j)$ can be computed in polynomial time. Y-subtrees can be merged in constant time by linking their respective arms together. B_1 through B_4 evaluate all candidate Y-subtrees adjacent to v_i and lying in $V(i, j)$, and keep a single candidate, i.e., the subtree which has maximum angle u. It is easy to show by induction that it is sufficient to consider only the Y-subtrees in the B's and F's. B_1 considers all subtrees which do not have both v_i and v_j as notches (Fig. 4.22a). B_2 and B_3 compute the subtrees whose vertex adjacent to v_i is an N2-node. Note that B_1 and B_2 may share common subtrees. The two possible configurations are illustrated in Fig. 4.22 b,c. Finally B_4 detects all candidate subtrees such that the vertex adjacent to v_i is an N3-node (Fig. 4.22d).

4. *Completing the OCD* (Step 5)

The last step of procedure *ConvDec* consists of removing the remaining notches with the naive decomposition. This can be done in polynomial time, leading to the main result: an optimal convex decomposition of a simple polygon can be computed in polynomial time.

A rough analysis of the algorithm's complexity shows that the exponent in the polynomial is prohibitively high. Cutting down the complexity to $O(n + c^3)$ is a long and complicated process which we will not address here. The reader can find the basic scheme in Chazelle and Dobkin (1985) and all details in Chazelle (1980).

4. EPILOGUE

Approximations and decompositions of two and three dimensional shapes are only beginning to be understood. Powerful algorithmic techniques are available, and problems once treated in an ad hoc fashion can now be solved with general,

versatile methods. All of the algorithms reviewed here have been analyzed and many of them have been implemented. Their relevance to robotics is compelling, but more work needs to be done to adapt these methods to the context of specific applications. In particular, relatively little is known regarding problems in three dimensions.

The aim of this article has been to present state-of-the-art methods for approximating and decomposing geometric shapes. We have given emphasis to techniques with direct practical applications as well as methods of purely theoretical interest. Our rationale for including the latter has been that these methods often reveal enough insights to suggest simple, practical methods. For example, our long development on the OCD problem readily suggests efficient approximation methods (based on the naive decomposition and, say, X_2-patterns only). It is certainly counterproductive to dismiss complicated theoretical methods solely on the grounds of impracticality. Many recent advances in theoretical computer science might never come into practice as such but may very well, as has often happened in the past, trigger the making of practical breakthroughs. For example, the availability of practical methods for planar point location today (a central operation in many tasks—see chapter by Yap in this volume) owes a great deal to an earlier algorithm by Lipton and Tarjan, theoretically remarkable, but unfit for practical use.

The field of computational geometry is blossoming. One of its challenges is to span the entire spectrum from theory to practice, enabling powerful mathematical constructions to have an impact in practical domains. Robotics along with statistics, computer graphics, and a number of other applications areas serve as the prime providers of fascinating problems to researchers in computational geometry. In return, we believe that robotics is advised to keep an eye on an area that is blossoming today and is likely to come up with many of the algorithmic tools that it needs in the years to come.

ACKNOWLEDGMENTS

I wish to thank A. Aggarwal, J. Incerpi, C. K. Yap, and the referee for many helpful comments and suggestions.

The author was supported in part by NSF grants MCS 83-03925 and the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order No. 4786.

REFERENCES

Aggarwal, A., Chang, J. S., & Yap, C. K. (1985). Minimum area circumscribing polygons. *J. of Computer Graphics, Vision, and Image Processing*.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). *The design and analysis of computer algorithms*. Reading, MA: Addison-Wesley.

Ahuja, N., Chien, R. T., Yen, R., & Birdwell, N. (1980). Interference detection and collision avoidance among three dimensional objects. *Proc. 1st Annual Nat. Conf. on Artificial Intelligence*, Palo Alto, pp. 44–48.

Asano, T., & Asano, T. (1983). Minimum partition of polygonal regions into trapezoids. *Proc. 24th Annual FOCS Symp.*, pp. 233–241.

Asano, T., Asano, T., & Imai, H. (1984). *Partitioning a polygonal region into trapezoids.* (Res. Mem. RMI84-03). Dept. Math. Eng. & Instrumentation Physics, Univ. of Tokyo.

Avis, D., & Toussaint, G. T. (1981). An efficient algorithm for decomposing a polygon into star-shaped polygons. *Pattern Recognition, 13*, 395–398.

Baker, B. S., Fortune, S. J., & Mahaney, S. R. (1984). Inspection by polygon containment. *Proc. 22nd Allerton Conf. on Comm. Control and Computing*, pp. 91–100.

Baker, B. S., Crosse, E., & Rafferty, C. S. (1985). *Non-obtuse triangulation of a polygon.* (Technical Report). Numerical Analysis Manuscript 84-4, AT&T.

Ben-Or, M. (1983). Lower bounds for algebraic computation trees. *Proc. 15th ACM Annual Symp. on Theory of Computing*, pp. 80–86.

Bentley, J. L., Faust, G. M., & Preparata, F. P. Approximation algorithms for convex hulls. *Comm. ACM, 25*, pp. 64–68.

Bentley, J. L., & Ottmann, T. (1979, September). Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comp. C-28, 9*, 643–647.

Bentley, J. L., & Shamos, M. I. (1978). Divide and conquer for linear expected time. *Info. Proc. Lett., 7*, 87–91.

Bhattacharya, B. K., & El Gindy, H. (1984). A new linear convex hull algorithm for simple polygons. *IEEE Trans. Infor. Theory, IT-30*, pp. 85–88.

Boyce, J. E., Dobkin, D. P., Drysdale, R. L., & Guibas, L. J. (1982). Finding extremal polygons. *Proc. 14th ACM Annual Symp. on Theory of Computing*, pp. 282–289.

Chand, D. R., & Kapur, S. S. (1970). An algorithm for convex polytopes, *J. ACM, 17*(1), 78–86.

Chang, J. S., & Yap, C. K. (1984). A polynomial solution for potato-peeling and other polygon inclusion and enclosure problems. *Proc. 25th Annual FOCS Symp.*, pp. 408–416.

Chazelle, B. (1980). Computational geometry and convexity. Doctoral thesis, Yale University (Also available as Technical Report CMU-CS-80-150, Carnegie-Mellon University, Pittsburgh, PA).

Chazelle, B. (1982). A Theorem on polygon cutting with applications. *Proc. 23rd Annual FOCS Symp.*, pp. 339–349.

Chazelle, B. (1983a). A decision procedure for optimal polyhedron partitioning. *Info. Proc. Lett., 16*(2), 75–78.

Chazelle, B. (1983b), The polygon containment problem. *In advances in computing research* (pp. 1–33). Greenwich, CT: JAI Press.

Chazelle, B. (1984). Convex partitions of polyhedra: A lower bound and worst-case optimal algorithm. *SIAM J. on Comput., 13*(3), 488–507.

Chazelle, B. (1985). On the convex layers of a planar set. *IEEE Trans. Inform. Theory*, IT-*31*(4), 509–517.

Chazelle, B., & Dobkin, D. P. (1979). Decomposing a polygon into its convex parts. *Proc. 11th Annual ACM Symp. on Theory of Computing*, pp. 38–48.

Chazelle, B., & Dobkin, D. P. (1985). Optimal convex decompositions. *Computational geometry* (pp. 63–133), North-Holland.

Chazelle, B., Guibas, L. J., & Lee, D. T. (1983). The power of geometric duality. *Proc. 24th Annual FOCS Symp.*, pp. 217–225.

Chazelle, B., & Incerpi, J. (April 1984). Triangulation and shape-complexity. *ACM Trans. on Graphics, 3*(2), 135–152.

Chazelle, B., & Lee, D. T. (1986). On a circle placement problem. *Computing, 36*, 1–16.

DePano, A., & Aggarwal, A. (1984). Finding restricted k-envelopes for convex polygons. *Proc. 22nd Allerton Conf. on Comm. Control, and Computing*, pp. 81–90.

Eddy, W. (1977). A new convex hull algorithm for planar sets. *ACM Trans. Math. Software, 3*(4), pp. 398–403.

El Gindy, H., & Avis, D. (1981). A linear algorithm for computing the visibility polygon from a point. *J. Algorithms, 2*, 186–197.

El Gindy, H., Avis, D., & Toussaint, G. T. (1983). Applications of a two dimensional hidden line algorithm to other geometric problems. *Computing, 31*, 191–202.

Feng, H., & Pavlidis, T. (1975). Decomposition of pollygons into simpler components: Feature generation for syntactic pattern recognition. *IEEE Trans. Comp., C-24*, 636–650.

Ferrari, L., Sankar, P. V., & Sklansky, J. (1981, December). Minimal rectangular partitions of digitized blobs. *Proc. 5th International Conference on Pattern Recognition*, Miami Beach, pp. 1040–1043.

Franzblau, D. S., & Kleitman, D. J. (1984). An algorithm for constructing regions with rectangles: Independence and minimum generating sets for collection of intervals. *Proc. 16th Annual ACM Symp. on Theory of Computing*, pp. 167–174.

Garey, M., Johnson, D. S., Preparata, F. P., & Tarjan, R. E. (1978). Triangulating a simple polygon. *Info. Proc. Lett., 7*(4), 175–180.

Graham, R. L. (1972). An efficient algorithm for determining the convex hull of a finite planar set. *Info. Proc. Lett., 1*, 132–133.

Graham, R. L., & Yao, F. F. (1983). Finding the convex hull of a simple polygon. *J. Algorithms, 4*(4), 324–331.

Greene, D. H. (1983). *The decomposition of polygons into convex parts.* In F. Preparata (Ed.), Advances in computing research (pp. 235–259). Greenwich, CT: JAI Press.

Guibas, L. J., Ramshaw, L., & Stolfi, J. (1983). A kinetic framework for computational geometry. *Proc. 24th Annual FOCS Symp.*, pp. 100–111.

Hertel, S., & Mehlhorn, K. (1983). Fast triangulation of simple polygons. *Proc. FCT'83*, Borgholm, LNCS Springer-Verlag, pp. 207–218.

Imai, H., & Asano, T. *An efficient algorithm for finding a maximum matching of an intersection graph of horizontal and vertical line segments.* (Papers of IECE Tech. Group on Circuits and Systems, CAS), 83–143.

Imai, H., & Asano, T. (1983b, October). *Efficient algorithms for geometric graph search problems.* (Res. Memo. RMI 83-05). Dept. Math. Eng. & Instrumentation Physics, Univ. of Tokyo.

Jarvis, R. A. (1973). On the identification of the convex hull of a finite set of points in the plane. *Info. Proc. Lett., 2*, 18–21.

Kahn, J., Klawe, M., & Kleitman, D. (1980). Traditional galleries require fewer watchmen. *SIAM J. Alg. Disc. Method, 4*(2), 194–206.

Keil, J. M. (1983). *Decomposing polygons into simpler components.* Doctoral thesis, University of Toronto.

Keil, J. M., & Sack, J. R. (1985). Minimum decompositions of polygonal objects. *Computational Geometry*, North-Holland.

Kirkpatrick, D. G., & Seidel, R. (1983, October). *The ultimate planar convex hull algorithm.* (Tech. Rep. No. 83-577). Cornell University. A preliminary version appeared in *Proc. 20th Annual Allerton Conf. on Comm., Control, and Comput.*, pp. 35–42.

Klee, V., & Laskowski, M. C. (1985). Finding the smallest triangles containing a given convex polygon. *J. Algorithms, 6*(3), 359–375.

Knuth, D. E. (1973). *The art of computer programming: Sorting and searching*, (Vol. 3). Reading, MA: Addison-Wesley.

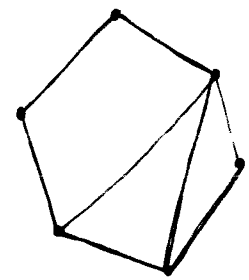
Lee, D. T. (1982). On k-nearest neighbor Voronoi diagrams in the plane. *IEEE Trans. Comput., C-31*, No. 6, 478–487.

Lee, D. T. (1983). On finding the convex hull of a simple polygon. *Int'l J. Comput. and Infor. Sci., 12*(2), 87–98.

Lingas, A. (1981, November). *Heuristics for minimum edge length rectangular decomposition.* Unpub. Manuscript, MIT.

Lingas, A. (1982). The power of non-rectilinear holes. *Proc. 9th Colloquium on Automata, Languages and Programming*, Aarhus, LNCS Springer-Verlag, pp. 369–383.
Lingas, A., Pinter, R., Rivest, R., & Shamir, A. (1982, October). Minimum edge length partitioning of rectilinear polygons. *Proc. 20th Annual Allerton Conf. on Comm., Control and Comput.*, pp. 53–63.
Lipski, W., Jr. (1983). Finding a Manhattan path and related problems. *Networks, 13*, pp. 399–409.
McCallum, D., & Avis, D. (1979). A linear time algorithm for finding the convex hull of a simple polygon. *Info. Proc. Lett.*, *9*, 201–206.
Megiddo, N. (1983). Linear time algorithm for linear programming in $\mathfrak{R}^3$ and related problems. *SIAM J. Comput.*, *12*(4), 759–776.
Megiddo, N., & Supowit, K. J. (1984). On the complexity of some common geometric location problems. *SIAM J. Comput.*, *13*(1), 182–196.
Newman, W. M., & Sproull, R. F. (1979). Principles of interactive computer graphics. New York: McGraw-Hill.
O'Rourke, J. (1982). The complexity of computing minimum convex covers for polygons. *Proc. 20th Annual Allerton Conf. on Comm. Control and Comput.*, pp. 75–84.
O'Rourke, J. (1984). *Finding minimal enclosing tetrahedra*. The Johns Hopkins Univ., Techn. Report.
O'Rourke, J., Aggarwal, A., Maddila, S., & Baldwin, M. (1984). *An optimal algorithm for finding minimal enclosing triangles*. (Tech. Rep. JHU/EECS-84/08). Dept. of EE/CS. Johns Hopkins Univ.
O'Rourke, J., & Supowit, K. J. (1983). Some NP-hard polygon decomposition problems. *IEEE Trans. Inform. Theory, IT-29*(2), 181–190.
Overmars, M. H., & van Leeuwen, J. (1981). Maintenance of configurations in the plane. *JCSS*, *23*, 166–204.
Post, M. J. (1982, May). *Computing minimum spanning ellipses*. (Tech. Rep. No. CS-82-16). Brown Univ. A preliminary version appeared in *Proc. 22nd Annual FOCS Symp.*, pp. 115–122.
Preparata, F. P. (1979). An optimal real-time algorithm for planar convex hulls. *C. ACM*, *22*, 402–405.
Preparata, F. P., & Hong, S. J. (1977). Convex hulls of finite sets of points in two and three dimensions. *Comm. ACM*, *20*(2), 87–93.
Roger, C. A. (1964). *Packing and covering*. Cambridge, England: Cambridge University Press (1964).
Sack, J. R. (1982). An $O(n\log n)$ algorithm for decomposing simple rectilinear polygons into convex quadrilaterals. *Proc. 20th Annual Allerton Conf. on Comm., Control, and Comput.*, pp. 64–74.
Sack, J. R., & Toussaint, G. T. (1981). A linear time algorithm for decomposing rectilinear star-shaped polygons into convex quadrilaterals. *Proc. 19th Annual Allerton Conf. on Comm., Control, and Comput.*, pp. 21–30.
Schachter, B. (1978). Decomposition of polygons into convex sets. *IEEE Trans. on Computers, C-27*, 1078–1082.
Schoone, A. A., & van Leeuwen, J. (1980). *Triangulating a star-shaped polygon*. (Tech. Rep. No. RUV-CS-80-3). Univ. of Utrecht.
Sedgewick, R. (1983). *Algorithms*. Reading, MA: Addison-Wesley.
Seidel, R. (1981). *A convex hull algorithm optimal for points in even dimensions*. M.S. Thesis, Tech. Rep. 81-14, Univ. British Columbia, Canada.
Shamos, M. I. (1978) *Computational geometry*. Doctoral thesis, Yale University.
Shamos, M. I., & Hoey, D. (1975). Closest-point problems. *Proc. 16th Annual FOCS Symp.*, pp. 151–162.
Silverman, B. W., & Titterington, D. M. (1980). Minimum covering ellipses. *SIAM J. Sci. Stat. Comput.*, *1*(4), 401–409.

Toussaint, G. T. (1980a). Pattern recognition and geometrical complexity. *Proc. 5th Int. Conf. on pattern recognition*, Miami Beach, pp. 1324–1347.
Toussaint, G. T. (1980b, October). Decomposing a simple polygon with the relative neighborhood graph. *Proc. 18th Annual Allerton Conf. on Comm., Control, and Comput.*
Toussaint, G. T. (1983, May). Solving geometric problems with the "rotating calipers". *Proc. IEEE MELECON'83*, Athens, Greece.
Toussaint, G. T., & Avis D. (1982). On a convex hull algorithm for polygons and its application to triangulation problems. *Pattern Recognition*, *15*(1), 23–29.
Yao, A. C. (1981). A lower bound to finding convex hulls. *J. ACM*, *28*,(4), 780–787.



20

5

$$\begin{array}{r} 50 \\ 2 \\ \hline 100 \end{array}$$

5