Reprinted from

Chandrajit L. Bajaj
Editor

Algebraic Geometry and Its Applications

Springer-Verlag
New York Berlin Heidelberg London Paris
Tokyo Hong Kong Barcelona Budapest

27 Decomposition Algorithms in Geometry

Bernard Chazelle
Leonidas Palios

27.1 Introduction

Decomposing complex shapes into simpler components has always been a focus of attention in computational geometry. The reason is obvious: most geometric algorithms perform more efficiently and are easier to implement and debug if the objects have simple shapes. For example, mesh-generation is a standard staple of the finite-element method; partitioning polygons or polyhedra into convex pieces or simplices is a typical preprocessing step in automated design, robotics, and pattern recognition. In computer graphics, decompositions of two-dimensional scenes are used in contour filling, hit detection, clipping and windowing; polyhedra are decomposed into smaller parts to perform hidden surface removal and ray-tracing.

Methods can be classified into two broad categories: In *image space* techniques the underlying space is discretized, sometimes in a hierarchical manner (e.g., quad-trees, oct-trees) and the discretization guides the decomposition algorithms. In this survey we will focus entirely on *object space* methods, which assume that the underlying universe is continuous. These methods focus entirely on the objects themselves and not so much on the universe in which they are embedded. Image space methods tend to be more hardware-oriented, as they can often be hardwired into special-purpose processors; in practice, they are also more directly tied to the needs of computer graphics and image processing. Object space methods are more universal; they make fewer assumptions about the world and and often enjoy a richer mathematical structure. On the other hand they might be less efficient on particular problems for which the hardware lends itself to a particular discretization scheme.

This paper reviews a large number of state-of-the-art decomposition methods. It does not pretend to be a comprehensive survey. Instead, it focusses its attention on methods that perhaps may not always be the solutions of choice in practice but reveal some of the most important current ideas in the design of decomposition algorithms.

27.2 The Two-Dimensional Case

The notion of decomposition related to a certain two-dimensional planar scene can be understood either as the partition of the plane (or a portion of it) into simple pieces, or as the explicit description of the plane tessellation induced by the elements of the scene. The former interpretation leads to problems involving convex partitions (and in particular triangulations), and related optimization questions; the latter leads to problems pertaining to line arrangements, Voronoi diagrams, or line segment and curve intersections.

27.2.1 Polygons

A *polygon* is a subset of the plane that is bounded by a finite set of nonintersecting closed polygonal lines. The segments that constitute these polygonal lines are the *edges* of the polygon, while their endpoints are its *vertices.* A polygon is *simple* if no two edges share any point other than a common endpoint. A simple polygon P is *convex* if it is a convex set, i.e., the line segment connecting any two of its points lies entirely in P, and is *star-shaped* if there exists a point p in P such that the line segment connecting p to any other point of P lies in P. In our discussion below, we will restrict our attention to simple polygons, and so we will omit the modifier "simple".

Triangulation

By *triangulation* of a polygon we mean its partition into non-overlapping triangles whose vertices belong to the vertex set of the given polygon. The triangulation problem has long been one of the central problems in two-dimensional computational geometry; not only does it have a number of applications in computer graphics, computer aided design and manufacturing, and robotics, but it is a very common preprocessing step in a large number of algorithms operating on polygons or planar subdivisions.

Convex polygons can be easily triangulated in linear time by drawing the diagonals that connect a given vertex of the polygon to all its nonadjacent vertices. The same procedure does not apply, however, in the general case, as such diagonals may cross the boundary of the polygon, or lie completely outside it.

An $O(n \log n)$ time algorithm:
The first subquadratic algorithm for the triangulation of an n vertex polygon was given by Garey, Johnson, Preparata and Tarjan [23] in 1978. Their algorithm works in two phases: during the first one, the polygon is *regularized*, meaning that it is partitioned into pieces that are monotone with respect to a fixed direction; then, in the second phase, each monotone piece is decomposed into triangles, thus completing the triangulation. A

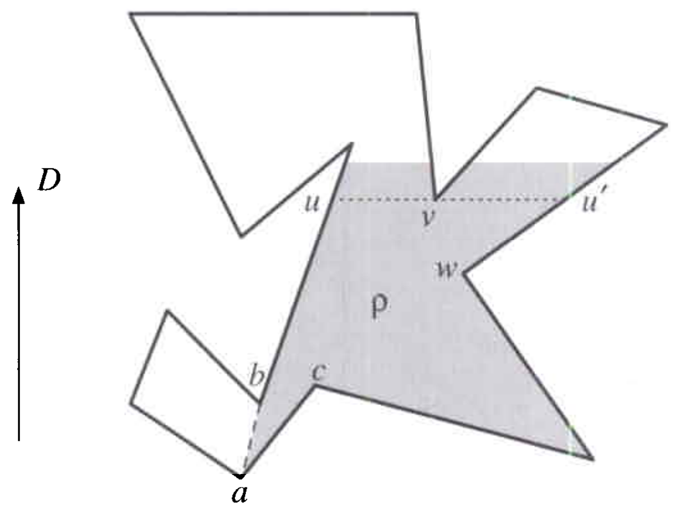

FIGURE 1. Decomposition into Regions Monotone with respect to D.

polygonal chain is said to be *monotone* with respect to a direction D if any line normal to D intersects the chain in no more than one point. A polygon is *monotone* with respect to D if its boundary can be decomposed into two monotone chains (with respect to D). In other words, the source of evil in a non-monotone polygon is the existence of vertices whose incident edges form a chain that is not monotone with respect to D.

The regularization phase fixes a direction D, and employs an algorithm of Lee and Preparata [33] to "resolve" these vertices by adding diagonals incident upon them. It proceeds by sweeping the polygon twice along D, moving in opposite directions. At the general step, the situation is as shown in Figure 1; we assume that the sweepline is moving from top to bottom, in which case only "bad" vertices like b and v, but not c, are resolved. (For instance, vertex b has already been resolved through the diagonal ab.) As a result, the interior of the polygon is split into a number of regions which will eventually yield the monotone partition. Let v, belonging to such a region ρ be the next "bad" vertex to be processed. The line normal to D that passes through v intersects the boundary of ρ in at least two points, two among which, say u and u', define the shortest line segment containing v. If σ is the part of the boundary of ρ that is delimited by u and u' and does not contain v, then v can be resolved by adding a diagonal connecting it to the lowermost vertex of σ (vertex w in the case of Figure 1).

Except for the initial sorting of the vertices of the polygon along D, the

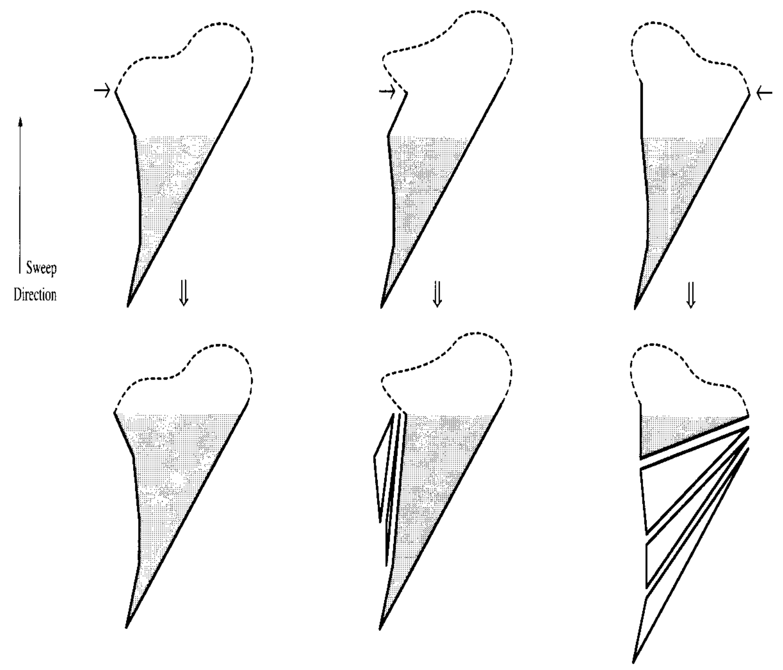

FIGURE 2. Triangulating a Monotone Polygon.

algorithm spends $O(\log n)$ time in each vertex, as it performs at most one Insert, or Delete operation into a balanced tree structure, such as a 2-3 tree, that holds all the information about the monotone regions in which the polygon has been partitioned. This accounts for a total of $O(n \log n)$ time spent in this phase. In fact, Lee and Preparata's algorithm is slightly more general than to our description suggests, and may introduce diagonals that lie outside the polygon; these can be easily located and deleted in linear time, however, or the regularization procedure may be made to ignore them.

The final partition is achieved by triangulating each monotone piece that the regularization phase produces. This can be done in time linear in the number of vertices of the piece for a total of $O(n)$ time. We sweep the piece with a line normal to the direction of monotonicity, maintaining in a stack vertices that form a concave chain. If the next vertex to be processed forms a concave angle with the chain, it is pushed on the stack, thus maintaining the invariant. Otherwise, vertices are popped from the stack (and triangles are reported) until either the vertex forms a concave chain with the rest of the chain, or the latter is reduced to a single vertex. The cases that may arise are shown in Figure 2.

The search for faster algorithms shifted the attention of researchers toward either special classes of polygons, or the discovery and use of the parameters that characterize the complexity of the polygon. In the context of the former trend, Toussaint and Avis [43] presented a linear time triangulation algorithm for edge-visible (a polygon P is edge-visible if there is an edge e of P such that for each point p of P one can always find a point q in e with the line segment pq lying entirely in P) or monotone separable polygons. See also the survey paper of Toussaint [42]. Advances were also made by Hertel and Mehlhorn [27], Fournier and Montuno [21], and Chazelle and Incerpi [11]. In particular, Hertel and Mehlhorn's algorithm has a running time of $O(n + r \log r)$, where r is the number of *cusps* of the polygon, i.e., the vertices of the polygon whose incident edges form an interior angle exceeding π. It is based on the sweep-line paradigm, and combines the two phases of the previous algorithm into one. The idea is to select and sort only a subset of the polygon's vertex set of size proportional to the number of cusps, and then process them in order; all other vertices are reached as we move along the boundary of the polygon and require constant processing time each.

Fournier and Montuno discovered an $O(n \log n)$ time sweep-line method for computing the horizontal (or vertical) *visibility map* of a polygon, which yields its triangulation in linear time postprocessing. The map is obtained by drawing horizontal segments, *chords*, inside the polygon, starting from each vertex and extending all the way across the polygon. The idea is to sort the vertices and process each of them in order, updating the sorted list of active trapezoids stored in a balanced tree.

The algorithm by Chazelle and Incerpi relies on the fact that the notion of visibility map can be extended to polygonal chains, where the horizontal lines that yield the decomposition may extend to infinity. It runs in $O(n \log s)$ time, where s is the *sinuosity* of the polygon, i.e., the number of times its boundary alternates between spirals of opposite orientation. Unlike its predecessors, this algorithm does not sort or sweep the plane. Instead, it computes visibility maps bottom-up by following the mergesort paradigm: the maps of smaller chains of the polygon's boundary are merged together two-by-two in a balanced manner. To achieve linear or even sublinear time, the merging relies on taking shortcuts along the added chords, thus bypassing long portions of the polygon's boundary. As before, a linear time postprocessing step converts the visibility map into a triangulation.

It should be noted that both the sinuosity s and the number of cusps r of a polygon can be as large as $\Omega(n)$, so that none of the algorithms presented so far beats the $\Theta(n \log n)$ time barrier in the worst case. As a matter of fact, this can be shown to be optimal in the case where the polygon contains holes, by using a straightforward reduction from sorting. However, the question of whether the triangulation of a simple polygon without holes was indeed as hard as sorting still remained open for many years. This question was finally settled negatively in 1986 by Tarjan and Van Wyk [41], who presented an $O(n \log \log n)$ time algorithm for the problem.

$O(n \log \log n)$ **time algorithms:**
The visibility map is computed by a clever combination of a linear time *Jordan sorting algorithm* for simple curves (Hoffman, et al. [28]), *divide-and-conquer*, and the fast processing of *finger search trees.* The basic idea in the algorithm is to slice the polygon into smaller pieces by means of a horizontal line L passing through one of its vertices v, and then to apply the technique recursively in the subpolygons produced. The Jordan sorting procedure helps to order along the horizontal line its intersection points with the boundary of the polygon. It is not difficult to see, however, that this approach is doomed if all intersection points are to be reported, as it may be the case that a quadratic number of such points exists. The key is to consider only the *essential* intersection points, i.e., those separating teeth-like portions of the boundary. As a result, a total of $O(n)$ visible pairs are reported. Of course, the Jordan sorting routine has to be appropriately updated to handle this partial information, yielding the *Jordan sorting with error correction.* Apart from that, the balanced divide-and-conquer (that is, making sure that each of the subpolygons that result from the slicing has no more than a constant fraction of the features of the sliced polygon) is used to guarantee that the total number of boundary segments in all regions is $O(n \log n)$. The techniques above in conjunction with the fast access, insertion and concatenation operations in finger search trees result in an $O(n \log \log n)$ time algorithm. Despite the fact that the algorithm has been implemented [32], its intricacy makes it impractical and is only of theoretical interest.

Kirkpatrick, Klawe and Tarjan [31]recently discovered an other $O(n \log \log n)$ algorithm that avoids the use of finger searching. Like Tarjan and Van Wyk's algorithm, it is also recursive in nature and uses a balanced divide-and-conquer approach. To add consistency to the visibility map, it introduces the notion of a *wrap-around partition* of a polygon (or chain): that is the decomposition of the plane induced by horizontal rays through the vertices of the polygon that extend to the left and right until they hit the polygon again. This last condition is always met, since it is assumed that the plane is replaced by an infinite vertical cylinder so that rays extending to the far right emerge again from the far left. Also, the boundary of the polygon is "thickened" so as to make it into a thin snake-like region.

The algorithm works as follows: for $k \geq (n/2)^{2/3}$, the boundary of the given polygon is partitioned into segments of length between k and $k/2$, yielding its *k-uniform partition.* Then the entire polygon is partitioned into *chunks* with the addition of one horizontal chord per segment. The chord associated with a certain segment emanates from a *special point* that splits the segment into two parts containing at least $k/36$ of the vertices of the segment each. Additionally, the endpoints of the chord should split the boundary of the polygon into two chains of size at least $k/36$ each. The special points are found easily, if $k/36$ edges are chopped off from either end of each segment and points are selected from the remaining parts. For a polygon of t vertices, there are $O(t/k)$ special points to be found; locating them and drawing the corresponding horizontal chords takes $O(t)$ time (remember that $k \geq (t/2)^{2/3}$), using standard planar point location techniques. Furthermore, the wrap-around partitions of those chunks that are bounded by at most 2 horizontal chords require another $O(t)$ time. The wrap-around partitions of the remaining chunks are computed by recursive application of the above procedure. Note that each of these chunks is of size at most $t^{2/3}$, while the total number of all their vertices is less than a constant fraction of t. Finally, when all the partial partitions are available, the wrap-around partition of the entire polygon can be constructed applying a linear time merging algorithm similar to [11]. So, each recursive step takes time linear in the size of the current polygon, which amounts to a total of $O(n \log\log n)$ time for the entire algorithm. At least on a conceptual level, this algorithm is much simpler than the Tarjan and Van Wyk method.

$O(n \log^* n)$ **expected time probabilistic algorithms:**
Clarkson, Tarjan and Van Wyk [16] gave an $O(n \log^* n)$ Las Vegas algorithm, based on random sampling. (A randomized algorithm is *Las Vegas* if it always returns the correct answer but its running time is a random variable.) The algorithm starts by picking a random sample of the polygon edges of the appropriate cardinality, and constructs their visibility map. Then, each of the remaining edges is intersected with the plane partition, and the intersection points of each region are Jordan sorted along its boundary. The information collected during Jordan sorting (the *family trees*) allows the refinement of each region into subregions induced by the non-sampled edges of the polygon. As long as there is some vertex of the polygon whose associated horizontal line has not been drawn, the algorithm recurs on the region enclosing it. The time analysis of the algorithm relies on the observation that, for a random sample of size r out of s noncrossing line segments, with probability greater than some fixed constant, the number of features refining each region is $O(r(\log r)/s)$, while the descriptive size of the entire refined partition of the plane does not exceed $O(s)$. As a result, the Jordan sorting and the region refinement at each level of recursion will take time linear to the size of the input polygon. Finally, selecting a random sample of size $r = s/\log s$ each time the algorithm is applied on a polygon of size s ensures that the vertical visibility decomposition can be computed in $O(r \log r) = O(s)$ time too. It further guarantees that for a polygon of n vertices the recursion depth will be $O(\log^* n)$, achieving the stated time complexity.

Recently, Clarkson and Cole [14] and independently Seidel [40] have announced considerable simplifications of the algorithm which circumvent the use of Jordan sorting.

A Linear Time Algorithm:
An optimal algorithm was recently discovered by Chazelle [7]. The starting observation is that the Chazelle-Incerpi bottom-up method computes too much intermediate information. To speed up the merging process, therefore, only a sample of the chords must be kept around. This sample must be carefully chosen so that the resulting submap represents a balanced approximation of the full-fledged visibility map. Merging is now more difficult because of the missing information, so various oracles are introduced to make it possible. One of these oracles is a primitive that allows us to shoot a ray from within any one of the computed regions and discover which edge (if any) of a given piece of the polygon is hit by the ray. Implementing this oracle requires the implementation of Lipton and Tarjan's planar separator theorem. A major difficulty is that the quality (i.e., the evenness) of the approximation scheme provided by submaps decays as we keep merging submaps together. To repair the damage the algorithm takes advantage of the fact that the duals of submaps have tree structures. It applies a normalization procedure that is best described as a geometric analog of the rotations needed to keep binary trees balanced under dynamic operations. Once a submap of the whole polygon is available, a refinement procedure is called upon to add all the missing chords. This is achieved in a top-down phase that makes use of the auxiliary structures built during the previous merges. Although the algorithm is not a good candidate for practical implementation, it is believed that heuristics based on it will turn out to be useful in practice.

Voronoi Diagrams—Skeletons.

Several geometric problems involve the computation and often the minimization of distances between a given set S of objects. Working on these problems is usually greatly facilitated if for a particular object in S one knows the locus of the points that are closer to it than any other member of S. Such a subset of the plane is called the *Voronoi region* of that object. It is clear from the definition that no pair of Voronoi regions intersect. The tessellation of the plane into Voronoi regions associated with the members of a set S is called the *Voronoi diagram* of S.

Voronoi diagrams are commonly defined with respect to point sets; extensions to segments of straight lines or curves and polygons have also been studied. In the case of polygons, the Voronoi diagram forms their so-called *internal* or *external skeletons.* The internal skeleton of a figure can be more formally defined as the locus of the centers of all the maximal inscribed circles, while the external skeleton of a set of figures coincides with the internal skeleton of their complement [30]. Examples of the Voronoi diagram of a point set, and the internal and external skeletons of two polygons are shown in Figure 3. Finally, it should be noted that the Voronoi diagram of a polygon coincides with the Voronoi diagram of its edges, so that an algo-

FIGURE 3. Voronoi Diagrams of (a) Eight Points, and (b) two quadrilaterals.

rithm operating on line segments can deal with the case of polygons too. Several $O(n \log n)$ algorithms for the computation of the Voronoi diagram of a point set have been proposed, as for example those of Guibas and Stolfi [26], and Fortune [20]. The latter is a sweep-line algorithm, while the former uses divide-and-conquer approach with a linear time merging step of two Voronoi diagrams. Let us mention that another linear time algorithm for merging two Voronoi diagrams follows directly from Chazelle's linear time algorithm for computing the intersection of two convex polytopes in 3-space [6].

The two algorithms that we describe below are applicable not only to point sets, but to sets including line segments or pieces of curves as well. They are both based on the divide-and-conquer paradigm, and they build upon the notion of the *contour* of two sets of the above mentioned objects in the plane. The contour of two such sets P and Q is defined as the locus of all the points in the plane whose minimum distances from both P and Q are equal. If P and Q contain only points, their contour is a set of simple (possibly closed) polygonal lines; if, however, line segments or pieces of curves are allowed in P and Q, the contour may include parabolic, hyperbolic, or elliptic arcs. It should be obvious that in order to merge the Voronoi diagrams of P and Q, we need only to compute their contour, since the Voronoi diagram of $P \cup Q$ coincides with the Voronoi diagram of either P or Q within each region that the contour defines. So, a linear time algorithm to find the contour of two sets guarantees linear time merging of their Voronoi diagrams, resulting in an $O(n \log n)$ time algorithm for the Voronoi diagram computation problem.

Kirkpatrick's algorithm:
In 1979, Kirkpatrick [30] proposed a divide-and-conquer algorithm for the construction of the Voronoi diagram of a point set. The algorithm computes the contour of two point sets using their *Euclidean minimum spanning trees*. The Euclidean minimum spanning trees (EMST) possess two very useful properties: (i) such a tree is a subgraph of the *Delaunay triangulation*, i.e., the dual of the Voronoi diagram of the underlying point set, and can be constructed from it in linear time, and (ii) considering two point sets P and Q on the plane, each edge of the EMST of P or Q intersects either zero or two edges of the contour of P and Q. The latter property can be (and actually is) used to allow the location of points on the contour, and to test whether the entire contour has been computed. Let us briefly outline the merging procedure. We assume that the Voronoi diagrams of two point sets P and Q need to be merged. First, their EMSTs are computed, which are subsequently augmented by a special point far away from the members in P and Q, so that it does not interfere with the edges already in the EMSTs. Clearly, one of the two edges upon which the special point is incident in the trees crosses an edge of the contour, and the point of intersection can in fact be located by moving along the edge in the Voronoi diagrams of P and Q. As soon as this starting point has been located, the corresponding contour component can be traced by following the perpendicular bisector of the current closest pair formed by one point from either point set. If the boundary of a Voronoi region is crossed in any of the partial Voronoi diagrams, the closest pair information is updated. The walk continues until either the starting point is met, meaning that the particular contour component is a closed polygonal line, or the current edge does not cross any other region boundary, and thus extends to infinity. Finally, the contour has been computed in its entirety after each edge of one of the EMSTs has been checked for intersection with the contour and either two or zero such points have been found. In order to estimate the cost of the contour tracing, Kirkpatrick introduces the notion of *spokes*. These are line segments that connect each point of a point set P with the vertices of the Voronoi region enclosing it in the Voronoi diagram of P. The introduction of spokes in a Voronoi diagram restricts the descriptive size of each region to a constant. Then, the cost of tracing a path is equal to the number of spokes and region boundaries crossed, which implies that the total time for the contour construction is linear in the combined size of the two point sets. The Voronoi computation can subsequently be finished off by clipping the partial Voronoi diagrams with the contour components. As the merging is carried out in linear time, the overall time required to compute the Voronoi diagram of n points is $O(n \log n)$.

Kirkpatrick then turned his attention to a generalized Voronoi diagram, that extends the known definition to both points and open line segments (line segments without their endpoints). In this case, the Voronoi regions may be bounded by line segments, half-lines, and parabolic segments. The previous algorithm can be extended to allow the construction of generalized Voronoi diagrams. As we are this time dealing both with points and open line segments, EMSTs are not applicable. Instead, *pseudo-minimum spanning trees*, defined next, can be used. Let us assume that a constant number of visibility directions have been specified, and the visible pairs within a set P of points and open line segments have been computed. Then, if the closest points between the members of any such pair are not endpoints, artificial endpoints are introduced at their locations. The minimum spanning tree of the set of points and of the enlarged set of endpoints of the line segments in P forms its pseudo-minimum spanning tree. The algorithm for the generalized Voronoi diagram computation can be then described as follows: for a given set of points and open line segments, its pseudo-minimum spanning tree is computed, and is separated into two connected components each of which contains at least a constant fraction of the elements in the initial set. After the generalized Voronoi diagrams of the two subsets that correspond to these connected components have been constructed, they are merged in the same way that was used in the standard Voronoi diagram construction, with the exception that EMSTs are replaced by pseudo-minimum spanning trees. The algorithm again runs in $O(n \log n)$ time.

Yap's algorithm:

Eight years later, C.K. Yap [44] proposed a different algorithm for the construction of the Voronoi diagram of n circular and straight line segments that intersect only at their endpoints. As mentioned earlier, in this case the Voronoi diagram consists of straight line, parabolic, hyperbolic and elliptic segments. The algorithm is based on the divide-and-conquer paradigm, too: vertical lines are introduced through the segment endpoints decomposing the plane into *slabs*; then the Voronoi diagram of the entire arrangement is computed by constructing the Voronoi diagram of each slab, and merging pairs of adjacent slabs in a tree-like fashion. If, however, segments contribute pieces in many slabs, this approach may take as much as $\Omega\left(n^2\right)$ time if the Voronoi diagram in each slab is computed in its entirety. The way to avoid that is by computing only the essential part in each slab, whose size is proportional to the number of segment endpoints lying in the slab. In particular, segments that cross both the bounding vertical edges of a slab subdivide it into regions called *quads*. The computation of the Voronoi diagram in the slab is then restricted to only those quads that contain at least one endpoint of some segment. This complicates the merging process, which can still, however, be carried out in linear time, resulting in an overall $O(n \log n)$ running time. It must be noted that, as in the algorithm of Kirkpatrick, the merging process introduces spokes splitting the Voronoi regions into subregions of no more than four sides, and also relies on the contour computation.

Optimization Problems.

Several decomposition problems, such as polygon triangulation, admit more than one valid solution. Quality criteria can then be used to select the best one among them, thus giving rise to optimization questions. For the polygon decomposition problem, such criteria usually involve the number of pieces produced, the total length of the cuts introduced, or the minimum angle between two edges. Interestingly, the existence of "holes" in the polygons makes a big difference in the complexity of the problems. We analyze these two cases separately.

Hole-free Polygons:
The most natural optimization question regarding a given polygon involves the minimization of the number of simpler pieces in which it can be decomposed. Such pieces may be convex, monotone, or star-shaped polygons, triangles, or trapezoids. Nearly all algorithms suggested for the resolution of these questions invariably employ dynamic programming.

Partition into the minimum number of convex polygons:
Chazelle and Dobkin [8] in 1979 presented an algorithm for achieving a partition of a polygon into the minimum number of convex pieces, establishing that this problem can indeed be solved in polynomial time. They introduce the notion of X-patterns that can be used to resolve several notches (i.e., vertices with reflex angles) at the same time. In particular, an X_k-pattern is an interconnection of k notches which removes all reflex angles at the notches involved without creating any new ones (Figure 4). A very important observation is that if the r notches of a polygon are resolved by m X-patterns and some additional line segments, the number of convex pieces produced is at most $r+1-m$; so, the more X-patterns, the fewer convex pieces. Unfortunately, X-patterns have many degrees of freedom, thus making the minimization computation extremely time-consuming. Chazelle and Dobkin introduce a more constrained version of the X-patterns, the Y-patterns. A Y_k-pattern is an X_k-pattern such that no edge joins two Steiner points. For example, the X_3-pattern of Figure 4 is a Y_3-pattern, whereas the X_5-pattern of the same figure is not a Y_5-pattern. The crucial property of Y-patterns is that all X_k-patterns, except for $k = 4$, can be transformed into Y-patterns. The desired set of X_4- and Y-patterns is constructed by applying dynamic programming. The running time of the algorithm is $O\left(n + r^3\right)$, where r is the number of notches of the polygon.

Partition into the minimum number of trapezoids:
Asano and Asano [1] gave an $O(n^3)$ time algorithm, for the case where the pieces of the partition are restricted to trapezoids with their parallel edges parallel to a given direction D. The algorithm makes use of the *minimally effective* diagonals. These are diagonals that either are parallel to D and

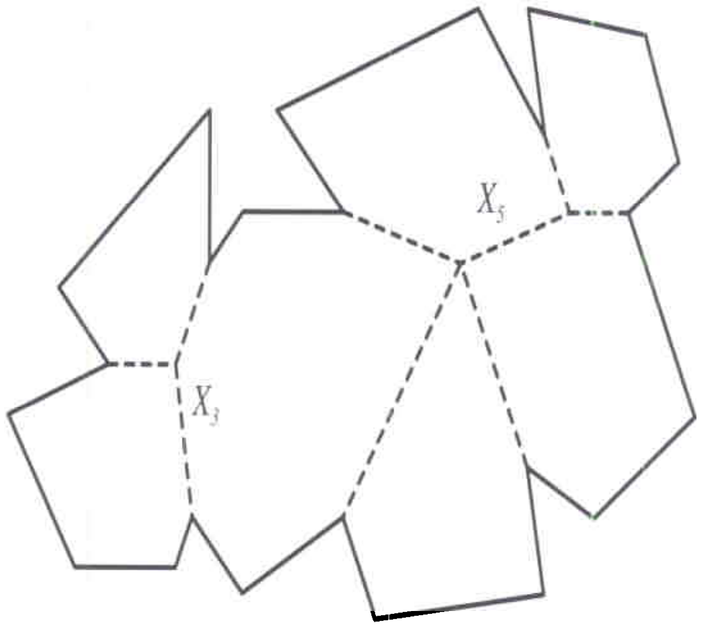

FIGURE 4. A Decomposition using X-patterns.

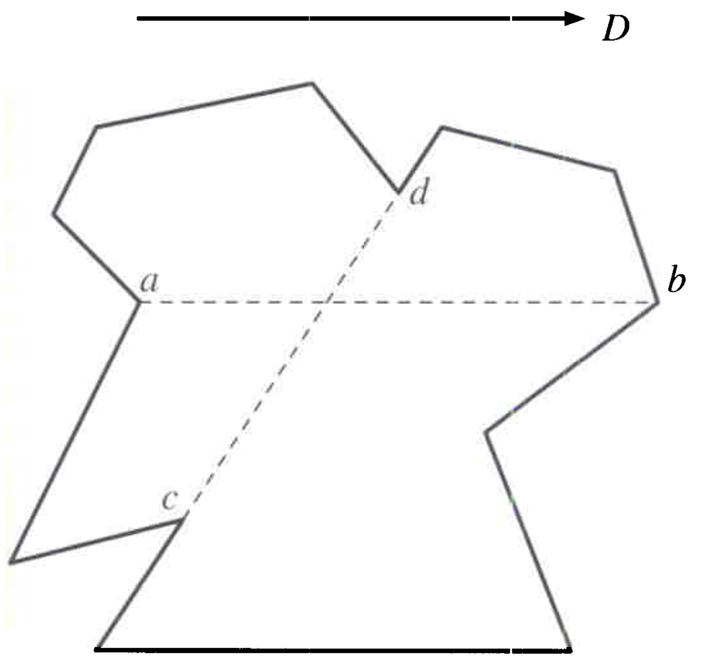

FIGURE 5. The Diagonals *ab* and *cd* are Minimally Effective.

connect two vertices of the polygon, or join two vertices that are incident upon collinear edges (Figure 5).
Since each such diagonal saves exactly one trapezoid in the final decomposition, the largest subset of non-crossing such diagonals ensures a minimum-size decomposition. The algorithm starts by identifying the set of minimally effective diagonals, and reduces the problem of selecting the maximum-cardinality non-intersecting subset among them to finding the maximum independent set of a *circle graph*. The nodes of this graph correspond to the diagonals, while the intersection information is captured in the edge set. The latter problem can be solved by using an algorithm of Gavril [24]. The key observation is that one can reduce the task of finding a maximum independent set of a circle graph of m nodes to m iterations of a routine for finding a maximum independent set on an interval graph. Gavril gave a quadratic time routine for the latter problem. After the appropriate diagonals have been selected, they are added into the polygon. Each subpolygon thus produced can be easily split further into trapezoids by drawing line segments parallel to D emanating from its vertices. The most

expensive component of the algorithm is the maximum independent set routine. Since there may be as many as $O(n)$ minimally effective diagonals it accounts for $O\left(n^3\right)$ time in the worst case, which is also the worst-case time performance of the entire algorithm.

The time complexity was improved to $O\left(n^2\right)$ in a later paper of Asano, Asano, and Imai [2], where a different routine for the selection of the maximum independent set of an interval graph is used. This routine involves a sorting step and then proceeds in linear time. In this way, all n iterations can be carried out in a total of $O\left(n^2\right)$ time.

The algorithms above assume that *Steiner points* are allowed in the decomposition, that is points other than the vertices of the given polygon may be used as vertices of the pieces in the decomposition. For the case in which Steiner points are disallowed, Greene [25] gave an $O(r^2n^2)$ time algorithm for the partition of a polygon into the minimum number of convex pieces. Later, Keil [29], improving on an earlier algorithm, achieved a worst-case time performance of $O\left(r^2n\log n\right)$. The algorithm is based on the observation that each of the diagonals in the partition must be incident upon at least one notch of the polygon. So, the algorithm starts by locating all the notches and finding the vertices of the polygon that are visible from them. Subsequently, a dynamic programming routine computes the minimum decompositions of the subpolygons that the previously computed visible pairs define. The subpolygons are processed in order of increasing size. Unfortunately, the number of valid minimum decompositions may be extremely large; the key is to partition them into equivalence classes and retain a single representative for each such class. This allows the computation of the minimum piece convex partition of a polygon of n vertices and r notches in $O\left(r^2n\log n\right)$ time. The technique can be extended to deal with the partition of a polygon into the minimum number of star-shaped, spiral, or monotone pieces ([29]).

Finally, one may ask for a partition of a polygon that minimizes the total length of the cuts. In the case of a convex partition, and under the additional restriction that Steiner points are disallowed, Keil's dynamic programming approach yields an algorithm requiring $O\left(r^2n^2\log n\right)$ time in the worst case [29].

Polygons with holes:

In contrast to the previous paragraph, all minimization questions discussed become NP-hard when applied to polygons containing holes. Reductions were drawn from 3SAT or planar-3SAT both known to be NP-complete ([22], [34]).

In the case where Steiner points are allowed in the decomposition, Lingas [35] gave a reduction from planar-3SAT showing that the partition of a polygon in the minimum number of convex pieces is NP-hard. The problem remains NP-hard if the convex pieces are restricted to triangles. Furthermore, O'Rourke and Supowit [38] showed that covering a polygon containing holes with the minimum number of convex pieces (that may be overlapping) is NP-hard by reducing 3SAT to it. Finally, Lingas et al. [36] showed that the problem of minimum edge-length partition of a polygon with holes is NP-hard using a transformation from planar-3SAT.

Disallowing Steiner points does not help in obtaining polynomial time algorithms for these questions. Instead, Lingas [35] showed that the convex partition problem remains NP-complete. NP-completeness results have also been established by O'Rourke and Supowit [38] for the minimum piece covering of a polygon with holes, and by Keil [29] for the minimum edge length convex partition.

27.2.2 Line Segments and Curves.

The problem of line segment intersection can be phrased as follows: Given a set of n line segments, report all their pairwise intersections. Let k denote the number of these intersections. This counts intersections that are combinatorially but not necessarily geometrically distinct. For example, if three segments intersect in one point, k is $\binom{3}{2} = 3$. For line segments, the quantity k may range from a constant up to $O\left(n^2\right)$. As a result, the complexities of the presented algorithms are expressed in terms of both n and k. Although most recent work has been restricted to line segments, extensions to curves have also been described.

An $O\left((n+k)\log n\right)$ time algorithm:
The first non-naive algorithm for line segment intersection is due to Bentley and Ottmann [4]. It runs in $O\left((n+k)\log n\right)$ time and requires $O(n+k)$ space. The algorithm employs the sweep-line paradigm: the endpoints of the segments are sorted according to their x-coordinates; then a vertical line sweeps the plane, stopping at each endpoint or intersection point detected during the sweep. Newly discovered intersection points are inserted into a priority queue using their x-coordinates as ranks. At all times, a balanced binary tree stores the ordered list of segments that are crossed by the vertical sweep-line. Processing a left endpoint involves inserting the corresponding line segment into the tree, and checking it for intersection with its neighbors above and below; processing a right endpoint involves deleting the corresponding line segment from the tree and checking its immediate neighbors for intersection. If the tests for intersection discover intersection points, the latter are inserted in order in the list of points. Processing an intersection point involves switching the order of the intersecting line segments in the tree, and checking no more than two pairs of line segments for intersections, again inserting them (if any) appropriately. The processing of an endpoint or intersection point takes $O(\log n)$ time, so that the total running time is $O\left((n+k)\log n\right)$. The same approach can be extended to curves that are monotone in the x-direction provided that their intersections among themselves and with vertical line segments can be computed

effectively.

A modification suggested by Brown [5] reduces the space requirement of the previous algorithm from $O(n+k)$ to $O(n)$. The idea is to reduce the number of intersection points stored in the ordered list of points pending processing down to $O(n)$. The solution is to maintain at most one intersection point per line segment, which will of course be the one closest to the sweep-line. The other intersections detected can be ignored at that time, for they will be rediscovered and reported later.

An $O(n \log n + k)$ time algorithm:

The first time-optimal algorithm for the line segment intersection problem was proposed by Chazelle and Edelsbrunner [9]. Although it, too, employs the sweep-line paradigm, a number of enhancements and additional techniques are used to guarantee the $O(n \log n + k)$ running time. The algorithm processes the segments in increasing abscissae of their leftmost endpoints, and maintains their (vertical) visibility map. Processing a segment involves locating its left endpoint in the map, in which it is inserted by traversing the regions that it intersects. The problem that arises is that a segment may cross many visibility chords, and the incurred cost may be prohibitive. To overcome that difficulty, Chazelle and Edelsbrunner split the segments into $O(\log n)$ pieces each, using a segment tree. Although visibility chords are attached to these artificial endpoints, as if the pieces resulting from the same initial line segment were independent, each piece is still marked with the index number of the segment it came from. This fragmentation of the line segments implies that many subsegments may have endpoints with the same x-coordinate, so that the visibility chords at these artificial endpoints consist of a large number of small edges, making it costly to scan them all every time such a chord is traversed. So, pointers are associated with these edges enabling fast location of the immediately previous and next non-vertical edges of the neighboring regions.

The location of the region of the visibility map in which the left endpoint of a segment falls, is carried out with the help of the *sweep tree*, a balanced tree that stores the edges that the current sweep-line intersects. It should be noted at this point that unlike most sweep-line algorithms, the sweep-line is not assumed to be straight or even monotone for that matter. In order to describe how the sweep tree is used, the concept of *x-walks* needs to be presented. Let us assume for simplicity that no three line segments intersect at the same point, and for a point p in a segment s let us consider the following path: starting from p, we proceed along s from left to right, up to either the right endpoint of s, in which case the path ends, or the intersection point of s with a segment s', in which case the path continues along s' from left to right. The path that is thus traced is an x-walk starting at p. Figure 6 displays the different x-walks defined on an arrangement of line segments. These walks partition the set of edges of the arrangement into equivalence classes, where two edges belong to the same class, if they belong to the same x-walk. If we want to find the relative position of a point p with respect to the edges that the vertical line through p intersects, we need not have stored these edges explicitly. We only need one representative of each of the equivalence classes in which these edges belong. Putting it another way, the sweep tree invariant is that no edge in the tree lies completely to the right of the vertical line through the current point, and for each edge e that this line intersects there exists an x-walk starting from some edge in the tree and leading to e. This is due to the lazy way in which the tree is updated. Namely, suppose that a segment s is to be inserted in the visibility map. If s is not the leftmost piece of an initial segment t (resulting from the segment tree fragmentation) then its position can be found from the previous piece of t, by means of an $O(n)$ size *appearance table*. Otherwise, the tree is traversed starting at its root. If the edge e stored in some node t of the tree that is visited during the traversal intersects the vertical line through the left endpoint of s, then the test above/below between s and e can be readily carried out, and a decision is made regarding which child of t must be visited next. If not, then starting from e an x-walk is traced until an edge intersecting the current vertical line is found, which reduces to the previous case; alternatively, the x-walk may terminate, in which case the tree node is deleted, and a new tree traversal according to the above guidelines starts at the root of the tree. As a matter of fact, the contents of the tree lag well behind the current vertical line, as only the absolutely necessary part of the tree is updated during insertions. Note that if the right endpoint of a line segment is reached no deletion operation is issued in the tree; deletion may happen later as a side effect of some insertion.

Finally, moving from region to region of the visibility map requires the ability to locate the intersection point of a segment and such a region. This can be carried out by walking along the boundary of the region until the point is found (if it exists). A better way, however, is to use *dovetailing*, where two walks proceeding concurrently in opposite directions along the boundary are initiated. The walks stop as soon as one of them locates the point of intersection, or they cross in which case no such point exists. Dovetailing serves also another purpose; it adds vertical *shortcut* edges in some regions, which speeds up subsequent walks along their boundaries.

Although the algorithm is not very complicated (it was implemented with no great difficulty), its time analysis requires an intricate amortization argument. The time analysis of the algorithm is rather complicated. Since the algorithm produces the complete description of the map of the fragmented line segments, the space required is $O(n \log n + k)$, but Chazelle and Edelsbrunner note that it can be easily reduced to $O(n+k)$. To obtain an $O(n)$ bound remains an open problem.

Probabilistic Algorithms:

In 1989, Clarkson and Shor [15] described three randomized algorithms, each running in $O(n \log n + k)$ expected time. We will describe the most

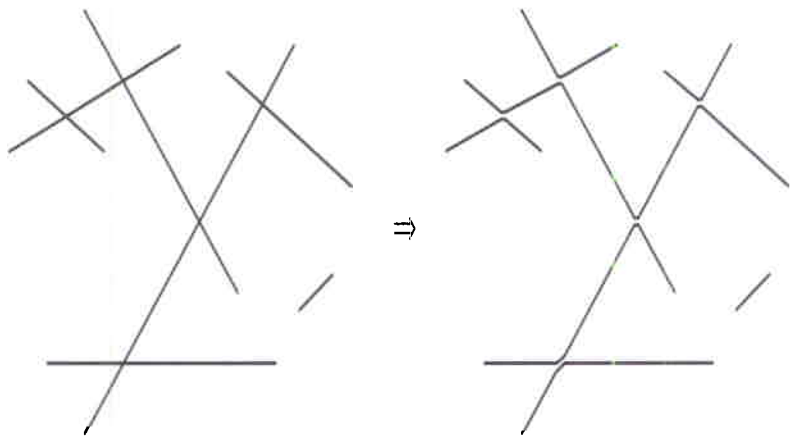
FIGURE 6. A Set of Seven Line Segments and its x-walks.

space efficient one, requiring only $O(n)$ space in the worst case. Let us outline before, however, one of the other two algorithms that is used in the space optimal algorithm. This algorithm processes the segments in random order, and maintains their visibility map in one of the standard planar subdivision representations. In addition, a "conflict" graph is maintained storing pairs of trapezoids in the map and segments that have not been processed yet. In particular, each trapezoid t is associated with a list of segments with an endpoint lying in t, whereas each segment is associated with the trapezoids that contain its endpoints. The addition of a new segment s is carried out as follows: the trapezoid that contains an endpoint of s is found using the conflict graph; then walking along s, we move from trapezoid to trapezoid that s intersects. Trapezoids may be split into no more than four pieces, some of which may merge with adjacent ones into larger cells, as visibility chords may shorten due to the presence of s. In either case, the planar reconfiguration is followed by appropriate updating of the conflict graph. Processing n line segments in this way requires $O(n \log n + k)$ expected time and $O(n + k)$ space.

We are ready now to present the algorithm that requires linear space in n. Clearly, being entitled to only that much space, one cannot store the visibility map of the line segments in its entirety. Instead, divide-and-conquer is applied, until the size of the subproblems becomes $O(\sqrt{n})$, the space sufficing for the entire map of each such subproblem. The algorithm begins by picking a "good" random sample of segments of size roughly $\sqrt{n}$;

a sample R of size r is considered *good* if any trapezoid in the visibility map of R intersects $O\left(n(\log r)/r\right)$ line segments, while the total number of intersections over all trapezoids is $O(n)$. For each such trapezoid T, the list of line segments S_T that intersect it is constructed, and the algorithm recurs on S_T. A second recursive step on the trapezoids that a good random sample of S_T defines brings the number of line segments conflicting with a particular trapezoid down to $O\left(n^{1/4}(\log n)^{3/2}\right)$. Then, the algorithm that we outlined earlier in this paragraph can be applied. The question that remains is how to pick a good sample, and compute the lists of intersections. To do that, two methods running in parallel are used. The first one works in a brute-force fashion: picks a random sample, constructs its visibility map, and computs the sizes of the intersection lists for each cell in the visibility map. If this computation implies that the sample is not good, a new sample is chosen, and the process is repeated. Otherwise, the chosen sample is adequate, and the actual lists of intersections for the trapezoids are constructed. The second method picks a random sample too, and applies the algorithm that we outlined earlier processing the segments in the sample first. Again, if at any time during the processing of the non-sampled segments, the number of conflicts disqualifies the chosen sample from being good, a new sample is selected, and the method is restarted.

The time analysis of the algorithm relies on two important observations pertaining to a random sample of size r out of a set of n line segments that have a total of k intersections. The first provides a bound of $O\left(kr^2/n^2\right)$ on the expected number of intersections among the members of the sample. The second states that (i) with probability at least 3/4, the total number of conflicts is $O(n+kr/n)$, and (ii) with probability at least $1-1/n^{10}$, each individual trapezoid conflicts with $O\left(n(\log n)/r\right)$ segments. The conjunction of these two observations guarantees that a good random sample is found in $O(n \log n + k)$ time; if $k > n\sqrt{n}$ the first method succeeds, otherwise it is the second one that succeeds. Since the two methods run in parallel, the total time for the computation of the conflict lists is $O(n \log n + k)$, and the stated time complexity follows by a two-level induction.

A similar randomized algorithm for the line segment intersection problem has been proposed by Mulmuley [37]. Its expected running time is $O(n \log n + k)$ and its space requirement is $O(n + k)$. Mulmuley assumes the existence of a rectangular box, the *window*, that encloses all the line segments. The algorithm computes the visibility map of the segments clipped within the window. Note that visibility chords, called *attachments*, are drawn through all the endpoints and points of intersection. A key point, is that each face of the partition is associated with only those of the vertices on its boundary that witness a tangent discontinuity. For example, in the Figure 7, the face F_1 is considered to have only u, v, w and z as vertices. In this way, each face has no more than 4 vertices. Initially, the partition is nothing but a collection of $2n + 2$ strips induced by the attachments through the endpoints of the line segments. Then, the line segments are

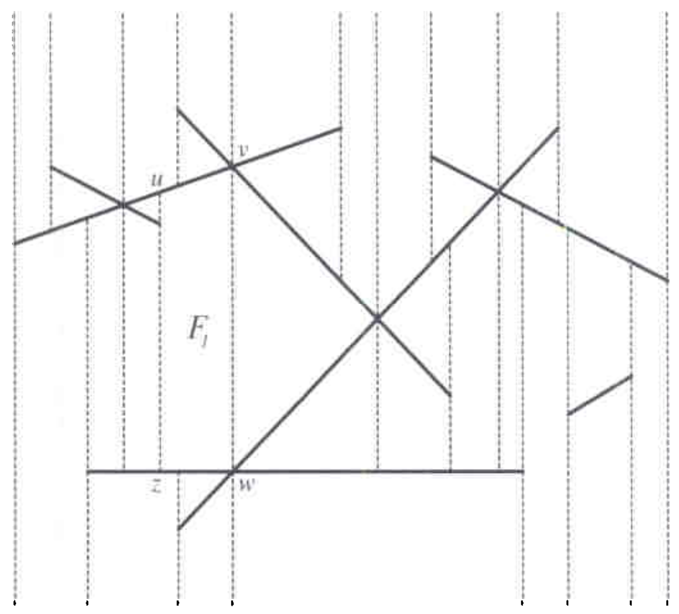

FIGURE 7. The Visibility Map of a Set of Seven Line Segments.

picked, one at a time, and are added to the partition, causing faces to split or merge, and attachments to contract. For a particular segment s, one of its endpoints is used to determine the first face F that s crosses. Then, the boundary ∂F of F is visited until the point of intersection of s and ∂F is found. Now F needs to be split. Furthermore, unless the point of intersection is the other endpoint of s, an attachment needs to be built at that point, and the same process needs to be pursued in the next face that s enters. If the crossing occurs on an attachment, this next face can be located in constant time; otherwise, the segment bounding the two faces is followed until a point of intersection or an endpoint is found. Note that only these points contain the necessary adjacency information to allow the crossing. The time bound of $O(n \log n + k)$ follows from an upper bound of $O(n \log m + k)$ on the expected number of points of attachment visited during the face transitions in the processing of all n line segments, and an upper bound of $5k + 3n + 2n \ln m$ on the expected number of face splits, where $m = O(n)$ is the *average span length*, i.e., the average number of segments that a vertical line through some endpoint intersects.

27.3 The Three-dimensional Case

While two-dimensional decomposition problems are reasonably well-understood, the same cannot be said of their three-dimensional counterparts. For on thing the complexity of the problems tends increases substantially as we move from two to three dimensions. A classical example is the fact that although any polygon can be partitioned into triangles without using Steiner points, not all polytopes can be partitioned into tetrahedra if no Steiner points are allowed. Actually, to decide whether a polytope admits such a partition is NP-complete, as was shown recently by Ruppert and Seidel [39].

Allowing Steiner points, however, reduces the complexity of the problem to polynomial in the size of the polytope. Indeed, Chazelle and Palios [12] have given an $O\left((n+r^2)\log r\right)$ time algorithm for partitioning an n-vertex polytope with r reflex edges into simplices. The algorithm proceeds in two phases: in the first one, the *pull-off* phase, the size of the polytope is reduced to $O(r)$; in the second, the *fence-off* phase, the resulting polytope is partitioned into at most a quadratic number of cylindrical pieces. Central to the pull-off phase is the notion of the *cup* of a vertex that is not incident upon a reflex edge. Let H_v be the convex hull of the vertices adjacent to such a vertex v. The facets incident upon v touch the boundary of H_v and separate it into two patches, one of which, say π, lies between the other patch and v. The cup of v is the polytope bounded by π and the facets of the polytope incident upon v (Figure 8). The idea is that if the cup of v is not hindered (i.e., no vertex of the polytope lies in the interior of the cup(v) or on the cup's boundary that is not adjacent to v), then vertex v and its cup can be removed from the polytope. In fact, there exist at most $2r$ vertices whose cups are hindered. If one subtracts also the vertices that are incident upon reflex edges, which are no more than $2r$ in number, one is left with at least $n-4r$ vertices whose cups can be removed from the polytope. If among these vertices we always select and remove those that have degree bounded above by some prespecified constant, and iterate on what is left, the tetrahedralization of the removed cups produces a number of tetrahedra linear in n. The entire
phase is carried out in $O\left((n+r^2)\log r\right)$ time. In the fence-off phase, vertical *fences* are erected through each edge of the triangulated boundary of the polytope, partitioning it into cylindrical pieces. A triangulation of one of the bases of such a piece is used to yield its decomposition into tetrahedra. This phase runs in $O\left(r^2\log r\right)$ time, and produces $O\left(r^2\right)$ tetrahedra. Summarizing, the running time of the entire algorithm is $O\left((n+r^2)\log r\right)$, and the total number of tetrahedra produced is $O\left(n+r^2\right)$, which is optimal in the worst case.

The application of decomposition algorithms brought up two important issues: that of the robustness of the algorithm due to finite precision arith-

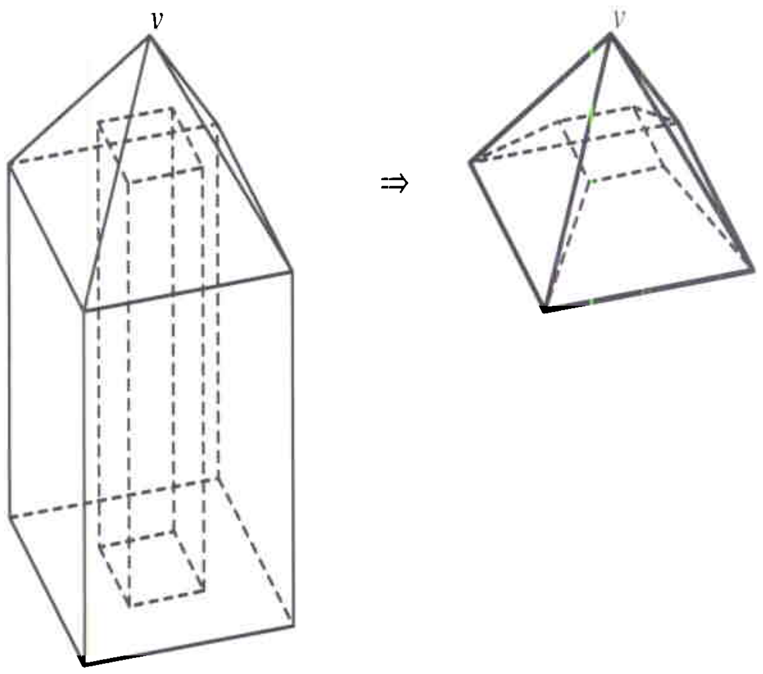

FIGURE 8. A polytope and the cup of vertex v.

metic, and of the "quality" of the produced decomposition. The question of robustness has been receiving considerable attention, and researchers have proposed several sets of primitives to deal with numerical innacuracies both in a general setting and for particular problems. For the problem of partitioning a nonconvex polytope into convex pieces, a numerically robust algorithm was presented by Bajaj and Dey [3]. Finally, in a recent paper, Dey, Bajaj, and Sugihara [19] described an algorithm that triangulates the convex hull of a point set in three dimensions avoiding the creation of flat or long and thin tetrahedra. The algorithm is an extension of an earlier algorithm of Chew [13] for the two-dimensional case, and is based on constrained Delaunay Triangulations.

The idea of a cylindrical decomposition is also exploited by Collins [17] in order to eliminate quantifiers from a prenex formula in first-order logic over the reals. Collins' idea is to partition $\mathbf{R}^d$, where d is the number of variables, into a regular cell complex whose faces are sign-invariant with respect to the polynomials of the formula. The faces of the decomposition are built by lifting cylindrically faces of a base decomposition defined recursively in dimension $d-1$. Each face of the decomposition of $\mathbf{R}^{d-1}$ is made the base of a cylinder, say, in direction x_1, which is then cut off by the real zerosets of the polynomials: in this way, when regarded as polynomials in x_1 (with coefficients in the ring of $(d-1)$-variate polynomials with rational coefficients) the real roots of the polynomials are delinated along any line parallel to x_1 that cuts a given face of the base decomposition. The entire construction produces $O\left(n^{2^d-1}\right)$ cells, and requires $O\left(n^{2^{d+6}}\right)$ time, where n is the number of polynomials in the given formula, and d is the number of variables involved. It is worth noting that the doubly-exponential time bound of Collins' construction is essentially the best one can hope to achieve for quantifier elimination, as shown by Davenport and Heintz [18]. Of course, this is not saying that any sign-invariant cell complex must suffer from this asymptotic blow-up.

Indication to the contrary was given by Chazelle, Edelsbrunner, Guibas, and Sharir [10], who described an algorithm for constructing a sign-invariant stratification of size $O\left(n^{2d-2}\right)$ in $O\left(n^{2d-1}\log n\right)$ time. This is a sign-invariant partition of $\mathbf{R}^d$ into smooth manifolds of constant complexity. It is not a regular cell complex, however, as the cells do not "glue" together; in particular the intersection of the closures of two adjacent cells need not be itself a cell of the stratification. Another drawback is that, although the size is singly exponential in d (which takes us closer to the obvious $\Omega\left(n^d\right)$ lower bound and the Thom-Milnor upper bound of $O\left(n^d\right)$ on the number of connected components in the complement of the varieties defined by the polynomials), the degrees of the intermediate polynomials needed to define the stratification are doubly-exponential in d.

Like Collins', the method has a recursive structure. Let F be a collection $\{f_1, \ldots, f_n\}$ of polynomials with rational coefficients. The sign-invariant stratification of $\mathbf{R}^d$ for F is constructed by lifting cells from a base decomposition. The difference with Collins' method is that we use several base decompositions, so as to provide a pool of cells from which we can choose the ones to be kept in the final stratification. Each cell has a *roof* and a *ceiling*, so we consider all pairs of polynomials to identify these roof/ceiling pairs. Let $\bigvee f_i$ denote the real zeroset of f_i. For each pair $i \leq j$, we project onto $\mathbf{R}^{d-1}$ the intersection of the variety $\bigvee f_i \times f_j$ with each of the $\bigvee f_k$. We also project the silhouettes of all the varieties (i.e., the critical sets of their projection maps). By expressing these projections as subresultants, we can treat them by means of polynomials in $d-1$ variables. This allows us to proceed recursively and thus produce a sign-invariant cell decomposition of $\mathbf{R}^{d-1}$. Next, we lift this decomposition cylindrically into a cell decomposition of $\mathbf{R}^d$. Finally, we use the variety $\bigvee f_i \times f_j$ to chop off the vertical cylinders into cylindrical cells. This provides us with all the cells of the final stratification whose roof/ceiling pair is defined by f_i and f_j (and many other useless ones, of course). We repeat the entire procedure for all pairs f_i, f_j, which gives us a total of $\binom{n+1}{2}$ intermediate cell decompositions of $\mathbf{R}^d$. Next, we examine every cell of every cell decomposition in turn, and keep only those that are free of intersections with any variety $\bigvee f_k$. These candidate cells might still be intersecting, so we add one final selection criterion based on the indices of their defining polynomials. This gives us a collection of mutually disjoint cells which, together with their "upper" boundaries, partition $\mathbf{R}^d$ into the desired sign-invariant stratification.

What is the use of a sign-invariant stratification of the sort we just described? It is shown in [10] how it can be used to perform efficient point location among real-algebraic varieties. Given n polynomials f_1, f_2, etc., we wish to build a data structure such that given any point x in d-space we can determine whether it is the root of any of these polynomials in logarithmic time. If the answer is no, in practice we often want a little more information, such as the name of the cell of the stratification in which x lies.

Such a fast point location algorithm gives us the ability to solve almost any multidimensional searching problem in logarithmic time, using a polynomial amount of storage. This also gives us a means to speed up the solution of a whole class of optimization problems (e.g., finding the longest segment in a polygon, finding whether there is any intersection between two sets of blue and red curves). More generally, suppose that we are given a set Z of n a d-variate polynomial f. In $O(n^d)$ time, it is easy to test whether some subset of Z forms a d-tuple that is a zero of f. The point location technique allows us to perform this test in $O(n^{d-\varepsilon})$ time, for some small $\varepsilon > 0$. Lowering this upper bound substantially is an open problem of major importance.

27.4 Concluding Remarks

The last word on decomposition problems has hardly been said. Which method to use in practice for a given application area remains by and large an open problem. Trade-offs between efficiency and simplicity must be resolved by programmers on an ad hoc basis and a taxonomy to guide us through the maze of decomposition techniques remains to be established. A number of fundamental theoretical questions are still open, especially ones regarding nonlinear higher-dimensional shapes. Also, the issue of robustness, such as how to deal with singularities, degeneracies, and round-off errors, has only begun to be explored, and despite noteworthy efforts, much unknown remains in that area, too.

27.5 References

[1] T. Asano, and T. Asano, Partitioning Polygonal Regions into Trapezoids, *Proc. 24th Annual IEEE Symposium on Foundations of Computer Science* (1983), 233–241.

[2] T. Asano, T. Asano, and H. Imai, Partitioning a Polygonal Region into Trapezoids, *Journal of the ACM* 33 (1986), 290–312.

[3] C.L. Bajaj, and T.K. Dey, Convex Decompositions of Polyhedra and Robustness, *SIAM Journal on Computing* 21 (1992), 339–364.

[4] J.L. Bentley, and T. Ottmann, Algorithms for Reporting and Counting Geometric Intersections, *IEEE Transactions on Computing* C-28 (1979), 643–647.

[5] K.Q. Brown, Comments on "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Transactions on Computing* C-30 (1981), 147–148.

[6] B. Chazelle, An Optimal Algorithm for Intersecting Three-Dimensional Convex Polyhedra, *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science* (1989), 586–591.

[7] B. Chazelle, Triangulating a Simple Polygon in Linear Time, *Proc. 31st Annual Symposium on Foundations of Computer Science* (1990), to appear in *Discrete and Computational Geometry.*

[8] B. Chazelle, and D.P. Dobkin, Optimal Convex Decompositions, *Computational Geometry*, North Holland, (1985), 63-133.

[9] B. Chazelle, and H. Edelsbrunner, An Optimal Algorithm for Intersecting Line Segments in the Plane, *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science* (1988), 590–600.

[10] B. Chazelle, H. Edelsbrunner, L.J. Guibas, and M. Sharir, A Singly Exponential Stratification Scheme for real Semi-Algebraic Varieties and its Applications, *Lecture Notes in Computer Science* 372 (1989), 179–193.

[11] B. Chazelle, and J. Incerpi, Triangulation and Shape-Complexity, *ACM Transactions on Graphics* 3 (1984), 135–152.

[12] B. Chazelle, and L. Palios, Triangulating a Nonconvex Polytope, *Discrete and Computational Geometry* 5 (1990), 505–526.

[13] L.P. Chew, Guaranteed-Quality Triangulation Meshes, Tech. Report, Dept. of Computer Science, Cornell University.

[14] K.L. Clarkson, and R. Cole, in preparation.

[15] K.L. Clarkson, and P.W. Shor, Applications of Random Sampling in Computational Geometry, II, *Discrete and Computational Geometry* 4 (1989), 387–421.

[16] K.L. Clarkson, R.E. Tarjan, and C.J. Van Wyk, A Fast Las Vegas Algorithm for Triangulating a Simple Polygon, *Discrete and Computational Geometry* 4 (1989), 423–432.

[17] G.E. Collins, Quantifier Elimination for Real Closed Fields by Cylindric Algebraic Decomposition, *Lecture Notes in Computer Science* 33 (1975), 134–183.

[18] J. Davenport, and J. Heintz, Real Quantifier Elimination is Doubly Exponential, *Journal of Symbolic Computation* 5 (1988), 29–35.

[19] T.K. Dey, C.L. Bajaj, and K. Sugihara, On Good Triangulations in Three Dimensions, *International Journal of Computational Geometry and Applications* 2 (1992), 75–95.

[20] S.J. Fortune, A Sweepline Algorithm for Voronoi Diagrams, *Algorithmica* 2 (1987), 153–174.

[21] A. Fournier, and D.Y. Montuno, Triangulating Simple Polygons and Equivalent Problems, *ACM Transactions on Graphics* 3 (1984), 153–174.

[22] M.R. Garey, and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.

[23] M.R. Garey, D.S. Johnson, F.P. Preparata, and R.E. Tarjan, Triangulating a Simple Polygon, *Information Processing Letters* 7 (1978), 175-179.

[24] F. Gavril, Algorithms for a Maximum Clique and a Maximum Independent Set of a Circle Graph, *Networks* 3 (1973), 261–273.

[25] D.H. Greene, *The Decomposition of Polygons into Convex Parts*, manuscript, Xerox PARC, 1982.

[26] L.J. Guibas, and J. Stolfi, Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams, *ACM Transactions on Graphics* 4 (1985), 74–123.

[27] S. Hertel, and K. Mehlhorn, Fast Triangulation of a Simple Polygon, *Lecture Notes in Computer Science* 158 (1985), 207–218.

[28] K. Hoffman, K. Mehlhorn, P. Rosenstiehl, and R.E. Tarjan, Sorting Jordan Sequences in Linear Time Using Level-Linked Search Trees, *Information and Control* 68 (1986), 170–184.

[29] J.M. Keil, Decomposing a Polygon into Simpler Components, *SIAM Journal on Computing* 14 (1985), 799–811.

[30] D.G. Kirkpatrick, Efficient Computation of Continuous Skeletons, *Proc. 20th Annual IEEE Symposium on Foundations of Computer Science* (1979), 18–27.

[31] D.G. Kirkpatrick, M.M. Klawe, and R.E. Tarjan, Polygon Triangulation in $O(n \log\log n)$ Time with Simple Data Structures, *Proc. 6th Annual ACM Symposium on Computational Geometry* (1990), 34–43.

[32] A. Knight, J. May, J. McAffer, T. Nguyen, and J.-R. Sack, A Computational Geometry Workbench, *Proc. 6th Annual ACM Symposium on Computational Geometry* (1990), 370.

[33] D.T. Lee, and F.P. Preparata, Location of a Point in a Planar Subdivision and its Applications, *SIAM Journal on Computing* 6 (1977), 594–606.

[34] D. Lichtenstein, Planar Formulae and their Uses, *SIAM Journal on Computing* 11 (1982), 329–343.

[35] A. Lingas, The Power of Non-Rectilinear Holes, *Lecture Notes in Computer Science* 140 (1982), 369–383.

[36] A. Lingas, R. Pinter, R. Rivest, and A. Shamir, Minimum Edge Length Partitioning of Rectilinear Polygons, *Proc. 20th Annual Allerton Conference on Communication, Control, and Computing* (1982), 53–63.

[37] K. Mulmuley, A Fast Planar Partition Algorithm, II, *Proc. 5th Annual ACM Symposium on Computational Geometry* (1989), 33–43.

[38] J. O'Rourke, and K.J. Supowit, Some NP-Hard Polygon Decomposition Problems, *IEEE Transactions on Information Theory* IT-29 (1983), 181–190.

[39] J. Ruppert, and R. Seidel, On the Difficulty of Tetrahedralizing 3-Dimensional Non-Convex Polyhedra, *Proc. 5th Annual ACM Symposium on Computational Geometry* (1989), 380–392.

[40] R. Seidel, in preparation.

[41] R.E. Tarjan, and C.J. Van Wyk, An $O(n \log \log n)$-time Algorithm for Triangulating a Simple Polygon, *SIAM Journal on Computing* 17 (1988), 143–178.

[42] G.T. Toussaint, Pattern Recodnition and Geometrical Complexity, *Proc. 5th International Conference on Pattern Recognition* (1980), 1324–1347.

[43] G.T. Toussaint, and D. Avis, On a Convex Hull Algorithm for Polygons and its Application to Triangulation Problems, *Pattern Recognition* 15 (1982), 23–29.

[44] C.K. Yap, An $O(n \log n)$ Algorithm for the Voronoi Diagram of a Set of Simple Curve Segments, *Discrete and Computational Geometry* 2 (1987), 365–393.