37

Ruler, Compass, and Computer: The Design and Analysis of Geometric Algorithms

by Leonidas J. Guibas and Jorge Stolfi

February 14, 1989

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984 — their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Ruler, Compass, and Computer: The Design and Analysis of Geometric Algorithms

Leonidas J. Guibas and Jorge Stolfi

February 14, 1989

Publication history

An earlier version of this work was presented at the NATO Advanced Study Institute on Theoretical Foundations of Computer Graphics and CAD held at Il Ciocco, Italy, 1987. Springer-Verlag published the proceedings under the title, *Theoretical Foundations of Computer Graphics and CAD.*

Authors' abstract

In this paper, the authors endeavor to convey the flavor of techniques, especially recent ones, that have been found useful in designing and analyzing efficient geometric algorithms. Each technique is presented by means of a worked out example. The paper presupposes some elementary knowledge of algorithmic geometric techniques and a more advanced knowledge of classical data structures. The aim is to share with the reader some of the excitement that permeates one of the most active areas in theoretical computer science today, namely, the field of *Computational Geometry.* The paper is based on a series of lectures delivered at the 1987 NATO Symposium on Theoretical Foundations of Computer Graphics and CAD.

Leonidas J. Guibas and Jorge Stolfi

Contents

Introduction

Computational geometry is the branch of computer science concerned with the design and analysis of algorithms for geometric problems. Among the tools of computational geometry there seems to be a small set of techniques and structures that have such a wide range of applications that they deserve to be called fundamental, in the same sense that balanced binary trees and sorting are fundamental for combinatorial algorithms in general. In this paper we will discuss and illustrate a number of these fundamental techniques of computational geometry. In most instances we give a worked out example of a problem solved by each of the techniques we discuss.

Historical note. The term "computational geometry" was originally coined by Robin Forrest [23] to denote the study of computational techniques in the realm of computer-aided geometric design. More recently, this term has been used to name a somewhat different field, in a way broader and in a way narrower than Forrest's conception, a field which is now considered part of theoretical computer science. This new use of the name "computational geometry," whose origin can be traced to Shamos [65], refers to the design and analysis of algorithms for all kinds of geometric problems, not necessarily in computer-aided design. This is the sense in which the term will be used in this paper.

Influenced by other branches of theoretical computer science, this new field has focused on the design of efficient algorithms when the number of objects in the input is large. This consideration has led to the development of asymptotically efficient (and sometimes practical) algorithms for problems dealing with large numbers of simple underlying objects in two or three dimensions: points, lines, planes, polygons, and polyhedra. This situation is to be contrasted with that of computer-aided design, where the underlying objects are more complex, say bicubic surface patches, but where asymptotic considerations tend to be less dominant. Abstracting away from this dichotomy, we can say that the goal of computational geometry is to collect and study all techniques relevant to the computer description and manipulation of geometric objects, within the wider framework of the analysis of algorithms.

It has been known since the time of Descartes that any geometrical problem can be recast in purely algebraic terms. It may therefore seem unnecessary to have a special discipline for geometric problems, distinct from numerical algebra and analysis. Indeed, some problems in geometry are best solved by algebraic methods, like computing the area of a polygon; but it is equally true that most geometrical concepts and algorithms are best

"seen" and studied in their own framework. Furthermore, most problems in computational geometry are neither purely combinatorial, nor purely numerical, but rather an intimate mixture of both. Computational geometry has therefore its unique flavor, and its unique combination of tools and techniques.

Many fundamental problems and results of computational geometry had been studied well before the field was recognized as a discipline in itself. Actually, we can say that computational geometry is the most ancient branch of computer science: the constructions of Euclidean geometry are legitimate *algorithms*, based on a well-defined, finite set of elementary operations.

In fact, Euclidean geometry is where several of the key concepts of computer science were first introduced: the close relationship between an algorithm and the proof of its correctness, the first examples of "provably unsolvable" problems (doubling the cube, trisecting the angle, squaring the circle), and so forth. Early this century the French mathematician Lemoine introduced (without much success) the idea of "computational complexity" of a geometrical construction, by counting the number of elementary steps it required. Other 19th century geometers, like Mohr and Mascheroni, showed that the ruler and compass of classical geometry could be replaced by other sets of tools (compass alone, ruler and scale, ruler and fixed-aperture compass, and so on). Their work is a close parallel to the theorems establishing equivalence of power for different flavors of Turing machines by simulation.

Because of the long pre-history of the field and its large number of applications, the fundamental results and techniques of computational geometry are widely scattered in publications that range from highly abstract mathematics texts to applications-oriented journals. It is only very recently that we are starting to see journals devoted explicitly (at least in part) to computational geometry, such as *Discrete and Computational Geometry* and *Algorithmica.*

Applications. Computational geometry (in both senses) is currently undergoing rapid growth. The field now has attained some mathematical depth and maturity, yet it encompasses many fundamental questions that are far from being fully resolved. Numerous areas of application contribute also to the vitality of the field; the list below shows a few of the most common ones, and some geometrical problems that are characteristic of each application.

- *Computer graphics*

  geometric sorting in hidden surface elimination
  polygon intersection in clipping

geometric neighbor computations in hit detection

- *Computer-aided design*

  union and intersection of geometric objects

- *VLSI design*

  union, intersection, and near-contact of rectangles
  wire routing

- *Computer-aided manufacturing and robot control*

  determining obstacle-free paths
  automatic milling

- *Pattern recognition and computer vision*

  fitting of geometrical objects to noisy data
  clustering algorithms

- *Statistics, operations research, and numerical analysis*

  classification by nearest-point determination
  finite-element decomposition

In addition, there is an extensive list of problems and applications that, although non-geometrical in origin and nature, can be better visualized and solved by recasting them in geometrical terms. For example, the geometrical equivalent of a database range-query problem (like "find all employees with salary greater than $10,000 and age between 30 and 40") is the selection, from a collection of points in n-space, of those whose projections fall inside a given m-dimensional box, $m \leq n$.

Eventually the techniques of computational geometry are bound to find more and more uses in computer-aided design, robotics, vision, and other applied areas. At the same time, these applied areas can serve as a rich source of problems for those in the field with more theoretical inclinations. It is the authors' hope that enough cross-fertilization will occur that in future years a distinction between the conceptions for the fields that Forrest and Shamos proposed will not need to be drawn.

Overview. The paper is divided into three parts. In the first part we deal mostly with algorithm design methods. We have restricted our attention to techniques that are especially effective in the geometric domain, that make use in some essential way of the geometry of the problem we wish to solve. In the second part we present a few data structuring techniques that are not themselves geometric but which have repeatedly found applications in geometric problems. Finally, in the third part

we discuss a couple of mathematically sophisticated techniques for the analysis of geometric algorithms.

We have not tried to be exhaustive, and indeed we have completely ignored many important results and entire sub-areas of computational geometry. The interested reader will find more systematic coverage in the books by Preparata and Shamos [58] and Mehlhorn [49, vol. III], as well as in the surveys by Lee and Preparata [42], Edelsbrunner [20], and Dobkin and Souvaine [18]. Notable omissions are most range-query type problems and the data structures motivated by them, such as segment trees [4] and interval trees [46]; a good introduction to this area is Overmars' thesis [55]. Another important new technique we have omitted is the use of randomization in geometric algorithms, for which the reader is referred to papers by Clarkson [13, 14]. We have also ignored the practical issues that arise in the implementation of computational geometry algorithms, and in particular the handling of "degenerate" cases where the input objects are not in "general position" (as theoretical discussions usually assume them to be). Nevertheless we hope that our sampling is representative enough that the reader comes away with a sense of the technical excitement that permeates the field today.

Part A: Algorithm Design Techniques

In this first part we discuss techniques for the design of geometric algorithms. Of course, computational geometry makes wide use of standard algorithm design methods, such as divide-and-conquer, dynamic programming, and so forth. However, we will concentrate here on methods and paradigms that seem to be more particular to this field, that in some essential way utilize the geometry of the problem at hand. Still, in many cases, the key insight for an efficient algorithm is an observation that allows us to map a geometric problem to a purely combinatorial one. We will see several instances of this in what follows.

A1. The locus approach

A common technique in solving geometric problems is the so-called *locus approach*. A classic example is the *post office problem*: given n sites in the plane (the post offices) and a query point x, report the site closest to x. If the sites remain fixed over several queries, then it pays to subdivide the plane into regions, each consisting of all points closest to a particular site. This partitioning of the plane is the well-known *Voronoi diagram* of the sites [58]. See figure 1. Once we have the Voronoi diagram, a nearest neighbor query can be answered simply by doing a point location in the diagram. This is the essence of the locus approach: subdivide space into regions such that all points in the same region yield the same answer to the type of query we are interested in. Point-location in this subdivision can then be used to answer any specific query. A method for computing the Voronoi diagram is given in section A4.

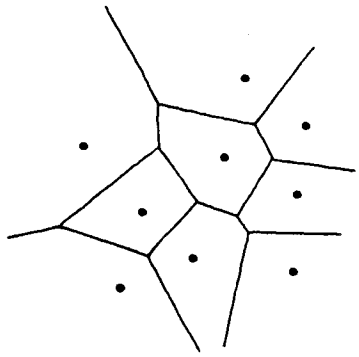

Figure 1.

One drawback of the locus method is that sometimes the size of the partitioning structure that we need to compute is too large. For example, in three dimensions, the Voronoi diagram technique is significantly less appealing for the post office problem than it is in the plane, simply because a three-dimensional Voronoi diagram can have size $\Theta(n^2)$. In such a situation one can trade space for time by building the patitioning structure only for subsets of the given objects, and then querying each patitioning structure in sequence. For example, if we break the sites up into $\sqrt{n}$ groups of size $\sqrt{n}$ each and compute the Voronoi diagram for each group, then the total storage in three dimensions goes down to $O(n\sqrt{n})$, while query time increases by a factor of $\sqrt{n}$.

The locus approach is widely used in computational geometry algorithms, and plays a significant role in several of the examples to be given in the following sections.

A2. Geometric transformations

Geometric transformations can be a very powerful tool in the development of geometric algorithms. In general, a geometric transformation is an *isomorphism* between two geometric spaces (or a geometric space and itself). The simplest kind of transformation is a map that preserves the type and the essential geometric properties of the mapped objects, such as the affine maps (which preserve parallelism), the projective maps (which preserve collinearity), and inversions (which preserve circles). Transformations of this sort can be used to simplify geometric algorithms, both in theory and in practice; for example, with a suitable affine map we can reduce many problems on an arbitrary triangle to problems on the equilateral one. Similarly, a projective map can be used to reduce many problems on a general conic section to problems on a circle. For more sophisticated examples, we can cite the linear programming algorithm of Karmarkar [37] and the minimum-area spanning ellipse algorithm of Post [56]. Both of these arrive at the answer by transforming the problem through a sequence of geometric maps, which eventually reduce the problem to a trivial one.

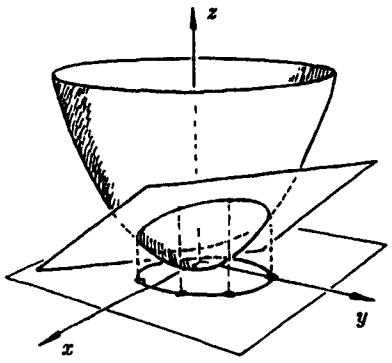

Figure 2.

Even more useful are the geometric transformations that change the nature of the mapped objects. A transformation of this type is an isomorphism between two geometrical structures, or two parts of the same structure, relating in a surprising way two classes of objects which appear to have very little in common. The isomorphism must by definition extend to the predicates and operations defined on those objects, and this is precisely where the the power of the method lies. By mechanically replacing the objects, predicates, and operations by their images, we can with negligible effort transport problems and solutions from one domain to the other. This not only reduces the number of theorems that need to be proved and of subroutines that have to be written, but also allows us to fully exploit in the study of one domain any intuition we may have gained in the other.

Delaunay diagram via convex hull. An example of this class of transformation is the "lifting map"

$$\lambda : (x, y) \mapsto (x, y, x^2 + y^2),$$

which lifts each point on the x, y-plane onto the paraboloid of revolution $z = x^2 + y^2$. See figure 2. This map can be used to transform circular queries in the plane to half-space queries in three-dimensional space [33]. The reason is that any four points A, B, C, and D in the plane are co-circular if and only if $\lambda(A)$, $\lambda(B)$, $\lambda(C)$, and $\lambda(D)$ are coplanar. Therefore the intersection of the paraboloid of revolution $z = x^2 + y^2$ with any plane

meeting it projects into a circle in the x, y-plane. Moreover, the part of the paraboloid below the plane projects to the interior of the corresponding circle in the x, y-plane, and the part above the plane to the exterior of the circle.

Consider now n sites in the plane and their lifted images under the map λ. These images are points on a convex surface, and therefore they are the vertices of a convex polyhedron, which is obviously their convex hull. Now consider a downward-looking face f of this polyhedron: by the definition of convex hull, its supporting plane π passes through the vertices of that face, and below all other vertices of the polyhedron. It follows that the projection of f onto the x, y-plane is a convex polygon (generally a triangle) whose circumcircle contains none of the other sites in its interior. We conclude that the projection of the downward-looking faces of the polyhedron are the faces of the Delaunay diagram of the given sites [7,27,33,40,41]. See figure 3. Similarly, an upward-looking face of the polyhedron corresponds to a convex polygon with vertices on the sites and whose circumcircle encloses all the sites. These are of course the faces of the dual of the *furthest-point* Voronoi diagram [58] for our collection of sites.

It follows that algorithms for computing the convex hull of n points in three-dimensions yield, under this lifting map, algorithms for computing Delaunay triangulations (and therefore Voronoi diagrams) in the plane. Curiously, it took a long time for the relationship between these two problems to become widely known, in spite of the fact that both have been solved independently by essentially the same algorithms. For example, the divide-and-conquer algorithms for Delaunay diagrams by Guibas and Stolfi [33] and Lee and Schachter [43] are essentially the Preparata-Hong algorithm [57] for computing the convex hull of n points in three dimensions, restricted to compute the lower half of the hull.

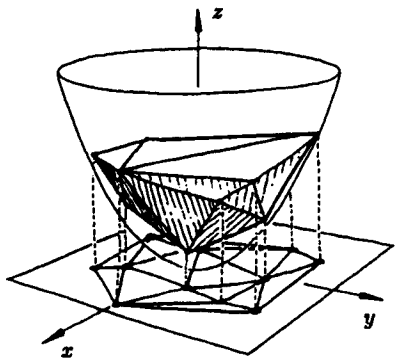

Figure 3.

The effect of the map λ can also be obtained (at somewhat greater computational cost) by lifting the points of the xy plane onto a sphere with inverse stereographic projection [7]. The general idea behind the lifting map λ is replacing non-linear tests on the input data (in our case, distance comparisons) by a non-linear map into a space of higher dimension, followed by linear tests on this transformed data. With a suitable lifting map, for example, we can reduce the problem of finding the minimum spanning ellipse of n points to a minimization problem on the faces of a 5-dimensional convex polyhedron [56]. Further applications of such geometric transforms are discussed in Kevin Brown's thesis [7].

A3. Duality

Duality is a particularly important example of a geometric transform. In general, duality manifests itself as a non-trivial isomorphism between a mathematical structure and itself. As with any geometric transform, this isomorphism must extend to predicates and operations (and therefore to theorems and algorithms) defined on the objects. Familiar examples are the duality between variables and inequalities in linear programming, the laws of DeMorgan in boolean algebra, the current-voltage duality of circuit theory, the Fourier and Laplace transforms of analysis, and the face-vertex duality in the theory of planar graphs.

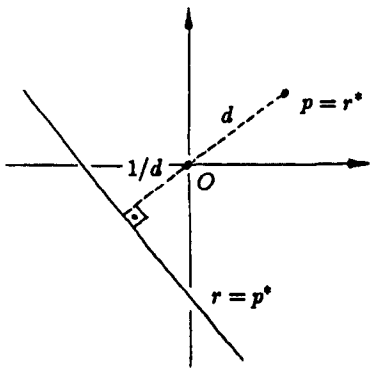

Figure 4.

As one might expect, this powerful concept has not escaped the attention of geometers. A natural duality between points and lines in the projective plane has been known since the eighteenth century [15]. One way to express this duality is to map the point with Cartesian coordinates (a,b) to the line with equation $ax + by + 1 = 0$, and vice-versa. We also map the point at infinity in the direction (a,b) to the line through the origin with equation $ax + by = 0$. Geometrically, if d is the distance between a point p and the origin $O = (0,0)$, the dual of p is the line r perpendicular to Op and passing at distance $1/d$ of O, on the side opposite to p. See figure 4. We will denote the line which is dual to point p by p^*, and the point dual to line r by r^*. (This is not the only possible duality between points and lines, but we will stick with this version in the current section.) It is not hard to show that the mapping thus defined preserves incidence: if point p lies on the line r, then point r^* lies on line p^*. It also preserves any "betweenness" and continuity relations: as point u moves continuously from p to q along the line r, the line u^* turns continuously at the point r^* from p^* to q^*.

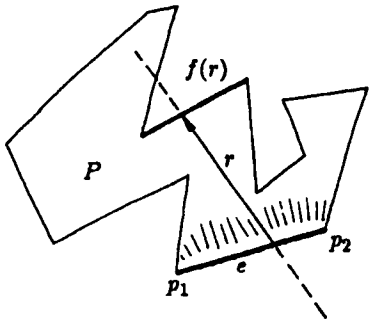

Figure 5.

Application to polygon visibility. For a typical use of duality in the development of geometric algorithms, consider the following visibility problem. Let $P = (p_1, p_2, \ldots p_n)$ be a simple polygon in the plane, and let e denote the edge p_1p_2. Given a line r that enters P through the edge e, we wish to determine the first edge $f(r)$ of P which r encounters after e. For concreteness, we can imagine that P is a room with opaque walls, e is a luminous neon bulb, and we want to know which wall is illuminated by a given light ray emanating from e. See figure 5.

Our approach to this problem is based on the observation that lines, and not points, are the primitive objects to consider in visibility questions. Since points are intuitively easier to grasp than lines, such questions are advantageously recast

in dual space, where the roles of points and lines are interchanged. By using duality, we can obtain an algorithm which, after $O(n \log n)$ preprocessing, allows us to solve this ray shooting problem in $O(n)$ space and $O(\log n)$ response time. These latter bounds are clearly optimal.

To simplify the discussion, let's assume P has been translated, rotated, and scaled so that p_1 is the origin and p_2 is the unit point on the x-axis. Under the duality we defined above, p_1^* is the line at infinity, and p_2^* is the vertical line with equation $x = -1$. Now consider the set E of all lines intersecting e. It is easy to check that these lines dualize to all points with $x < -1$, that is, the half-plane E^* delimited by the line p_2^* that does not contain the origin. See figure 6.

Note again that the lines in E are in one-to-one correspondence with the light rays starting at e and directed into the polygon. Let us then classify each point p of E^* according to the edge $f(p^*)$ that the ray p^* illuminates. This defines a partition $S(E^*)$ of the half-plane E^* into subregions where $f(p^*)$ is constant. Now consider two points p, q in the same region of $S(E^*)$, that is, with $f(p^*) = f(q^*)$. This means the rays p^* and q^* illuminate the same edge of P. Let u be any point on the segment pq; by the properties of duality, the line u^* must intersect e, must be concurrent with p^* and q^*, and must lie in the angle delimited by p^* and q^* that does not include the origin. Since the boundary of P is connected and free from self-intersections, we must have $f(u^*) = f(p^*) = f(q^*)$; that is, u lies in the same region of $S(E^*)$ as p and q. We conclude the following:

Lemma 1. *Each region of the subdivision $S(E^*)$ is a convex polygon.*

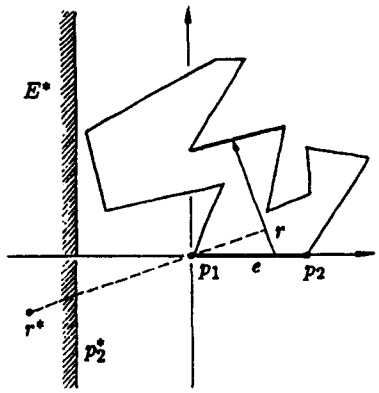

Figure 6.

Clearly the subdivision $S(E^*)$ (which has at most $n-1$ regions) provides the essential information we need to know in order to solve our problem. In order to compute this subdivision we first triangulate our polygon P by a standard $O(n \log n)$ time algorithm [25]. We then consider each of the $n-2$ triangles in turn, beginning with the one adjacent to the luminous edge e, and proceeding at each step across a diagonal from a visited triangle to a new one. In a typical step we enter a new triangle $\Delta = p_i p_j p_k$ which has one side in common, say $p_i p_j$, with a previously handled triangle. At this point we know the region V of E^* corresponding to all rays reaching $p_i p_j$ (a diagonal of P) from e; this region is a convex polygon in the dual plane. These rays will now be subdivided into two groups, according to which other side of Δ they exit from. Dually, the line p_k^* cuts the polygon V into two subpolygons V', V'', which are queued for processing in later steps. Assuming V is represented as a

balanced tree structure, we can perform each splitting operation in $O(\log n)$ time. So we have shown the following result:

Lemma 2. *It is possible to compute the convex subdivision $S(E^*)$ associated with illuminating the simple polygon P from edge e in $O(n \log n)$ time and $O(n)$ space.*

Once we have the subdivision, standard point-location methods can be used to solve the ray-shooting query in linear space and logarithmic time.

Recently Tarjan and Van Wyk [72] have discovered that a triangulation can be computed in $O(n \log\log n)$ time. Also, by using a fancier representation for the polygons V that exploits the finger search trees discussed in section B2, the splitting operation described above can actually be carried out in linear total time. Thus the subdivision $S(E^*)$ can be built in $O(n \log\log n)$ time.

A4. Space sweep techniques

Space sweep is one of the most useful paradigms of computational geometry. Generally speaking, space sweep is a technique that allows us to reduce an n-dimensional static problem to an $(n-1)$-dimensional dynamic problem. The basic idea can be described as follows. Suppose we have one or more objects lying in some Cartesian space, and imagine that space being completely traversed by a moving hyperplane (the *sweep plane*). At any given moment, the sweep plane intersects some subset of the elements, the *active* ones. These intersections evolve continuously in time, except when certain discrete *events* occur: when new objects join the active set, when old objects leave it, or when the configuration of the intersections on the sweep plane undergoes a qualitative change.

A space sweep algorithm is a discrete simulation of this process, using essentially two data structures: a queue of *future events*, and a representation of the active set and its cross-section by the current sweep plane. Each iteration of the algorithm removes the next event from the queue, and updates the cross-section data structure to mirror the effect of the sweep plane advancing past that point. Depending on the algorithm, each iteration may also reveal some future events that were not known at the beginning of the sweep; these must be inserted into the event queue, in the appropriate order, before proceeding to the next iteration.

The space sweep technique is quite old, and has probably been re-discovered several times in the early history of computational geometry. It has been used in algorithms for the following: finding planar convex hulls [58]; decomposing polygons into triangles [25], trapezoids [2], and monotone polygons [25]; intersecting line segments [5, 44, 45, 66], polygons [5], and convex polyhedra [35]; merging convex maps [32, 44, 54]; computing Voronoi diagrams [24, 27]; and many more.

Voronoi via cones. We will describe here an optimal space sweep algorithm for computing the Voronoi diagram of n sites in the plane, due to S. Fortune [24]. His algorithm can be understood more easily with the help of the following construction for the Voronoi diagram. For simplicity, we assume the sites are in general position, so that there are no two sites with the same x or y coordinate, and no four sites are cocircular.

Imagine that an identical opaque cone with vertical axis is grown upwards from each site. See figure 7. A symmetry argument shows that any two cones intersect along a curve (a hyperbola) that is contained in a vertical plane. That plane is simply the perpendicular bisector in three-space of the two corresponding sites.

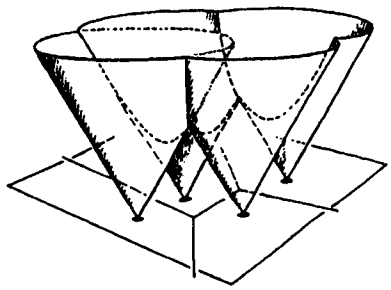

Figure 7.

Now imagine that we look at the collection of cones from below. The intersection of two cones will look like a straight line, the perpendicular bisecting line of the two sites p, q on the xy plane. Not every point of this line will be visible, however: a point a on it will be hidden by some other cone if and only if the site from which the latter grew is closer to a than p and q are. It follows that projecting the visible intersections down onto the plane gives the Voronoi diagram of the n sites.

Sweeping the cones. Consider what happens when we sweep this three dimensional picture from left to right with a plane that is inclined *at the same angle as the cones*, with its top half trailing its bottom half. See figure 8. At any moment, the sweep plane meets the xy plane along a line which we call the *sweep line*. The intersection of a cone and the sweep plane is empty until the sweep line reaches the site p that is the apex of the cone. At that moment the sweep plane becomes tangent to the cone, and their intersection appears in the form of a degenerate parabola of zero width, whose projection is a ray starting at p and pointing towards $x = -\infty$. A moment later that ray opens up to become a narrow parabola, which grows wider and wider as the sweep plane moves further to the right. It is easy to see that the projected parabola on the xy plane will have the site p at the focus and the sweep line as the directrix. Referring to figure 8, we see that triangles pvw and uvw have equal angles and share an edge; therefore they are congruent and $pw = uw$.

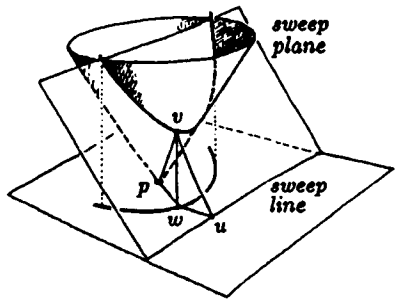

Figure 8.

The parabolic front. Now let's make the sweep plane opaque, too, and look at the whole thing from below. The sweep plane will completely hide the cones whose sites lie to the right of the sweep line. The cones of the sites already swept over will give rise to a set of left-pointing parabolas with parallel axes and various widths. A point on the xy plane is inside (to the left of) a parabola if and only if the cone of the latter lies below the sweep plane at those (x, y) coordinates. Therefore, any part of a parabola that lies inside another one is invisible. The only visible parts of those parabolas are those on the boundary of the union of their interiors. We call this boundary the *parabolic front.* See figure 9.

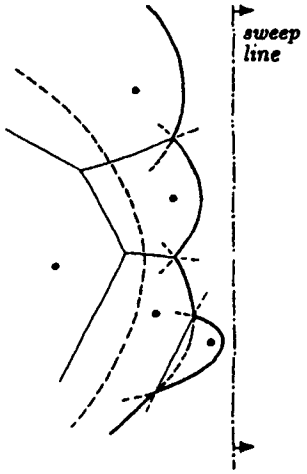

Figure 9.

Note that any horizontal line intersects the parabolic front at exactly one point. Therefore, the front consists of a single chain of arcs, extending from $y = -\infty$ to $y = +\infty$. To the right of the parabolic front, the sweep plane lies below all the cones, and completely blocks the view from the underside. To the left of the front, the plane lies above one or more cones, and cannot hide anything that would be visible otherwise. Looking from below, what we see there is a cut-out portion of the Voronoi diagram of the sites. It is easy to see that each arc of the front lies in some Voronoi region, and each "break" between two consecutive arcs lies on a Voronoi edge.

Evolution of the parabolic front. As the sweep plane advances, the arcs in the parabolic front move to the right and become flatter, and the breakpoints between them move along the corresponding Voronoi edges. It is not hard to show that the more recent of two consecutive arcs (the one whose site lies further to the right) grows in vertical extent at the expense of its older neighbor. We claim that as the sweep plane moves from $x = -\infty$ to $x = +\infty$, the breakpoints of the front will trace out every edge of the Voronoi diagram:

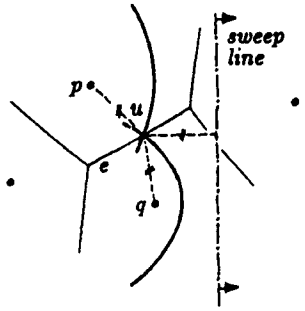

Figure 10.

Lemma 3. *Every point of every edge of the Voronoi diagram is a breakpoint of the parabolic front at some time during the scan.*

Proof: Let e be any Voronoi edge, and u a point on e (other than its endpoints). Consider the situation when the sweep line is already to the right of u, and the distance r between the two is equal to the distance between u and the sites p, q of the Voronoi regions separated by e. See figure 10. From the properties of Voronoi diagrams we know that no other site is closer to u than those two, that is, there are no other sites on or within the circle C with center u and radius r. Obviously, at that moment u will be on the intersection between the parabolas associated with p and q.

The only way u could not be a breakpoint of the front is if there were some other parabola passing between u and the sweep line. Let v be the point on that parabola just to the right of u; the focus of the parabola must then be on the circle with center v that is tangent to the sweep line. But that circle lies inside the circle C, a contradiction. We conclude that u is on the current front. ∎

The discrete events. The continuous evolution of the front is punctuated by discrete events, where new arcs join the front, or old arcs shrink down to nothing and drop out of the picture. The first case happens, for instance, when the sweep line runs over a new site s. See figure 11. That introduces a new degenerate parabola α with zero width (i.e. a twice-traversed horizontal ray) with focus and apex at s. The portion of α that lies to the right of the parabolic front becomes a new arc of the latter, and its insertion will split an existing arc β of the front in two pieces (if the sites are in general position, we can ignore the possibility of the ray passing through a breakpoint). We call this a `site` event, and we make the following claim:

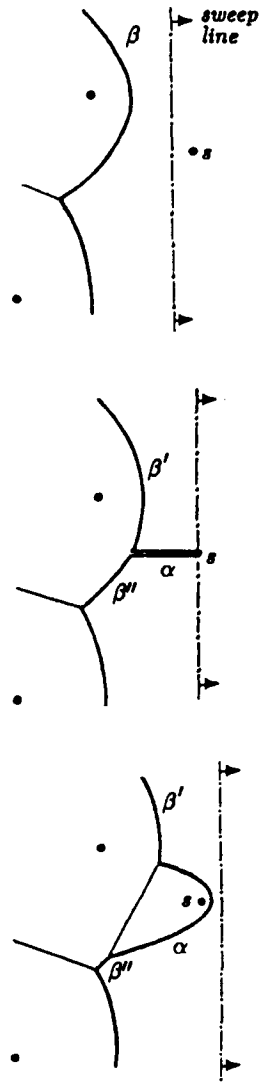

Figure 11.

Lemma 4. *The only way for an arc to enter the parabolic front is through a* `site` *event.*

Proof: The only other alternative is for new arcs to arise due to changes in the shape and position of existing parabolas, that is, due to some parabola overtaking the front and breaking through it. However, this cannot happen. Consider the situation when the parabola α in question is about to break through. It cannot do so in the middle of an arc β, since that would require the curvature of α to be higher than that of β and the focus of α to be further from the sweep line than that of β — which are contradictory statements.

On the other hand, α cannot break through the corner between two arcs β and γ. To do so, it would have to move faster in the x-direction than β and γ at the y-coordinate of their intersection. A little algebra shows that the horizontal speed at which a particular parabola advances at a given y-coordinate is a monotonically decreasing function of its absolute slope $|\sigma|$ (more precisely, it is $\frac{1}{2}(1 + 1/\sigma^2)$). Therefore, for α to overtake β and γ at a point u, the absolute slope of α at u would have to be smaller than those of β and γ, contradicting the fact that the latter are consecutive arcs of the front meeting at u. ∎

Let's now turn to the second class of events, where an existing arc drops out of the front after having shrunk down to a single point u. At that moment there are three parabolas passing through u, namely the disappearing arc β and the arcs α, γ

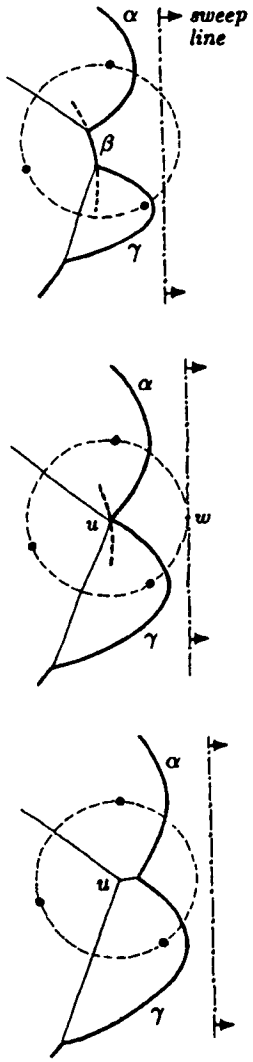

Figure 12.

just above and below it on the front. See figure 12. As in the proof of lemma 4, it is easy to show that α and γ cannot belong to the same parabola. The point u is therefore equidistant from the three corresponding sites and from the current sweep line. In other words, the sweep line is tangent to (and to the right of) the circumcircle of the three sites. We conclude that the point u is a vertex of the Voronoi diagram. (Since the sites are in general position, we can ignore the possibility of four or more of them having the same circumcircle.) We call the rightmost point w of the circumcircle associated with a Voronoi vertex u a `circle` event. From the preceding discussion, the following result is clear:

Lemma 5. *The only way for an arc to leave the parabolic front is through a* `circle` *event.*

The data structures. Now that we understand how the parabolic front evolves during the sweep, we are ready to consider its simulation in the computer. As mentioned before, the algorithm needs two main data structures: a queue of future events, and a description of the current parabolic front. The former contains all `site` events that lie to the right of the sweep line. It also contains some of the `circle` events in that region, namely those that correspond to three *consecutive* arcs of the current parabolic front. Note that some future `circle` events will be missing from this list, either because they involve sites that haven't been swept over yet, or because the corresponding arcs are not yet consecutive elements of the parabolic front. As we will see, these `circle` events will be discovered during the simulation, and inserted in the event queue before the sweep line advances past them. Conversely, not every three consecutive arcs of the current front specify a `circle` event.

The parabolic front is represented by a balanced search tree (or any equivalent structure), with the arcs as leaves and the breakpoints as internal nodes. We will use this tree to efficiently insert and delete arcs of the front, to locate the arc that intersects a given horizontal line, and to locate the arcs immediately above and below a given one.

We also hang from this tree the part of the Voronoi data structure built so far, consisting of the vertices, faces, and edges that lie totally or partially to the left of the current parabolic front. Each leaf of the tree (i.e., each parabolic arc) includes a pointer to the corresponding site, and each internal node (breakpoint) includes a pointer to the record representing the associated Voronoi edge in the incomplete diagram.

Event processing. Given these data structures, the algorithm itself is a straightforward event-driven simulation loop. At each iteration we remove the next event from the queue (the one with smallest x-coordinate), and we simulate the effect of the sweep line advancing to that point. We again use the assumption of the sites being in general position to ignore the possibility of two events having the same x-coordinate.

To process a `site` event s, we add to the tree a new degenerate arc α, consisting of a left-pointing horizontal ray starting at s. This requires us to split an existing arc β in two, and add two new breakpoints. We must also add a new edge to the Voronoi diagram, associated with both breakpoints, which is the frontier between the Voronoi regions of s and of the site associated with β. See figure 11.

Processing of a `circle` event is equally simple. Recall that such an event corresponds to the sweep line reaching the rightmost point of the circumcircle of three sites a, b, c, associated with three consecutive arcs α, β, γ of the parabolic front. At that point the middle arc β has zero length and must be deleted from the tree. Before we do so, we must add to the Voronoi diagram a new vertex u, the circumcenter of the triangle abc. That vertex is incident to the two Voronoi edges currently associated with the breakpoints α–β and β–γ. Also, after deleting β we must add a new Voronoi edge incident to u, and attach it to the new breakpoint α–γ. See figure 12.

Event scheduling. We still haven't said how and when the events get inserted into the queue. `Site` events get inserted all at once, at the beginning of the algorithm. `Circle` events are inserted as the simulation proceeds, when the three associated arcs first become consecutive elements of the front.

More precisely, when adding a new arc β during a `site` event s we look at the two new consecutive arc triplets created by the insertion (having β as the first and last element, respectively). For each triplet we compute the circumcircle of the corresponding sites, and add its rightmost point to the queue as a `circle` event. Note that since the site s is on the current sweep line, the new `circle` event cannot lie to the left of that line.

Similarly, when deleting an arc β between arcs α and γ during the processing of a `circle` event, we check the two new consecutive arc triplets created by the deletion (which start at α and end at γ, respectively). For each triplet we compute the circumcircle of the sites, and add its rightmost point to the queue, *but only if it lies to the right of the sweep line.* Note that it is possible for the circumcircle to lie strictly to the left of the sweep line, as figure 13 shows. That means the center of the circumcircle has already been swept over by the front.

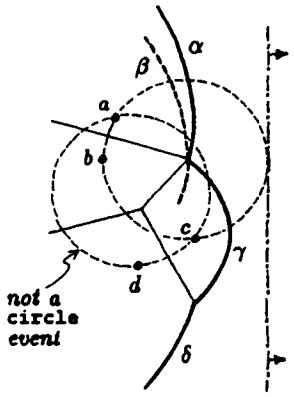

Figure 13.

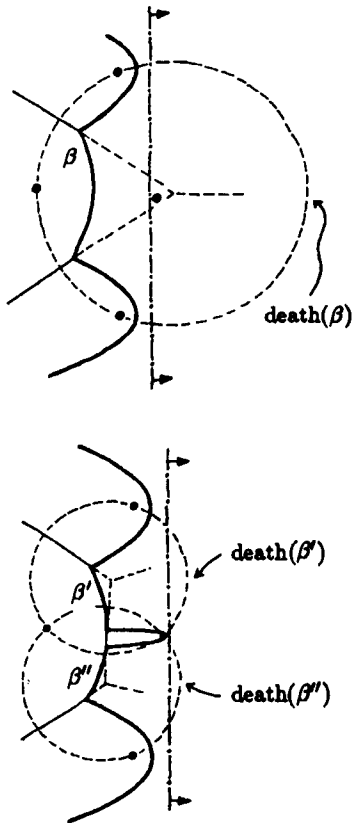

Figure 14.

As discussed below, that center (if it was a Voronoi vertex) will have been noticed and processed at an earlier date.

It is easy to see why this algorithm schedules every discrete event affecting the front in due time, that is, before the sweep line reaches that event. Because of lemma 4, new arcs can enter the front only through `site` events, which are all scheduled beforehand. Because of lemma 5, whenever an arc is about to disappear from the front, the sweep line must be tangent to the circumcircle of the sites associated with it and the two adjacent arcs. Therefore, the three arcs must have become adjacent at some earlier time, and the corresponding `circle` event must have been inserted in the queue at that moment.

On the other hand, the algorithm may schedule spurious `circle` events which do not correspond to actual changes in the front, and therefore to actual Voronoi vertices. This may happen, for example, if three arcs become consecutive elements of the front at some point, but a new arc is inserted in their midst before the sweep line reaches their `circle` event. At the moment the `circle` event was inserted, the Voronoi edges between the three arcs looked as if they would converge at some later point, but the arrival of the new site caused those edges to terminate prematurely. See figure 14. In fact, by the time the `circle` event is reached, some of the three arcs may have already disappeared, engulfed by more distant neighbors or by new arcs inserted in their midst.

To fix this problem, we keep for each parabolic arc β on the front a pointer `death`(β) to an event in the queue, namely the earliest `circle` event where that arc is currently scheduled to disappear. The Voronoi vertex corresponding to that event is the predicted meeting point of the Voronoi edges being traced by the endpoints of β. If those edges diverge, the arc is currently expected to live forever, and its event pointer is `nil`. To avoid spurious `circle` events, it suffices to keep the `death` pointers up to date during the simulation, and to promptly remove from the queue any `circle` event that is no longer pointed to by any arc. More precisely, we must discard the event `death`(α) whenever the arc α is split in two by a `site` event, or whenever one of the two arcs adjacent to α is deleted by a `circle` event. This completes the description of Fortune's algorithm.

Analysis. In order to get the desired bounds on the space and time cost of the algorithm, we have to show that at any time the parabolic front and the event queue contain at most $O(n)$ elements. This is not entirely obvious, since, for example, each parabola may give rise to many arcs on the front. However, the only way the parabolic front increases in size is through `site` events: then a new arc is added and an old arc is split in two. Since there are only n such events, this proves that the front

has linear size. (Another way to derive linearity comes from the fact that the front cannot contain two parabolas α and β that each appear twice in the pattern $\ldots\alpha\ldots\beta\ldots\alpha\ldots\beta\ldots$. The Davenport-Schinzel sequence theory developed in section C2 then implies that the combinatorial complexity of the front is $O(n)$.) Since each event currently in the queue is either a `site` event or a death event for an arc on the front, we conclude that the queue too never contains more than $O(n)$ elements. With a suitable priority queue structure [1] we can insert and delete an event in the queue, or find the one with minimum x, in only $O(\log n)$ time. We conclude that each iteration takes $O(\log n)$ time, and therefore we arrive at the following result:

Theorem 6. *Fortune's algorithm computes the Voronoi diagram of n sites in $O(n \log n)$ time.*

A5. The configuration space approach

The *configuration space* approach is a geometric technique from robotics research that has found application in many other areas of computational geometry [63]. In a typical application, we want to check quickly whether two or more geometric objects (say, a robot and the furniture in a room) intersect when placed in a given relative position. If we have to perform this test for the same set of objects in many different positions, we may consider pre-computing the entire set S of all relative placements (*configurations*) for which the two objects intersect. Then each query reduces to a test whether the proposed placement lies within S, that is, to a point location problem in the space of all configurations.

This approach is practical only when the configuration space has low dimension, such as problems involving two rigid objects in the plane. As the objects' motion becomes less constrained (allowing, say, translations and rotations in three-space, or articulated joints) the dimensionality of the configuration space increases, and the cost of pre-computing, storing, and searching S grows exponentially.

The simplest case that leads to non-trivial algorithms is detecting intersections between two polygonal objects P and Q in the plane that are allowed to move only by parallel translation, without rotation. In this case the configuration space is two-dimensional, and consists of all displacement vectors measured between two reference points fixed on the objects. The set S of configurations leading to "interference" between the two polygons is itself a planar polygonal region.

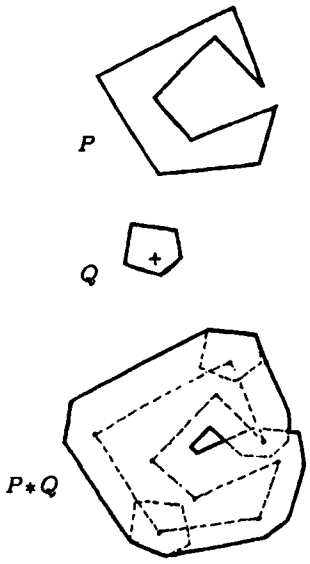

Figure 15.

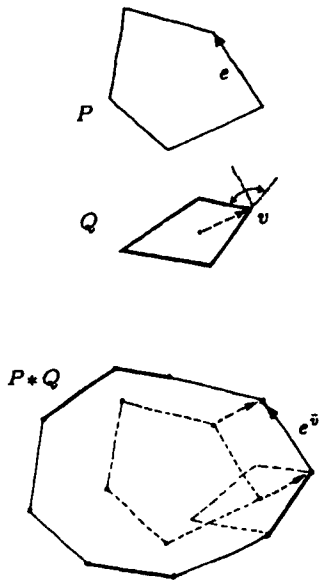

Figure 16.

Even with these restrictions, the proper treatment of this problem for general polygons requires more machinery than we can afford to develop here. The reader is referred to the literature, in particular to the *kinetic framework* proposed by Guibas, Ramshaw, and Stolfi [31]. Henceforth we will restrict ourselves to the case of two *convex* polygons.

The set of all invalid placements can be easily expressed in terms of the *Minkowski sum* or *convolution* of two sets A, B of vectors, which is simply the set of all pairwise sums of their elements:

$$A * B = \{a + b \ : \ a \in A \text{ and } b \in B\}$$

To see how this relates to our problem, we must consider the two given objects as sets of vectors (measured from a common origin). Let's also denote by A^N the set A rotated 180° around the origin, and by $A^{\overline{x}}$ the set A translated by the vector x:

$$A^{\mathrm{N}} = \{-a \ : \ a \in A\}$$
$$A^{\overline{x}} = \{a + x \ : \ a \in A\}$$

The set of forbidden positions is given by the following trivial result:

Lemma 7. *If P and Q are plane objects, then $P^{\overline{x}}$ intersects $Q^{\overline{y}}$ if and only if the vector $x - y$ lies in the set $P^{\mathrm{N}} * Q$ (or, equivalently, $y - x$ lies in $P * Q^{\mathrm{N}}$).*

In other words, $P^{\mathrm{N}} * Q$ is the set of displacements of P relative to Q for which the two figures intersect.

The convolution of two polygonal figures P, Q is also a polygonal figure. If P and Q have p and q edges, respectively, then $P * Q$ (and $P^{\mathrm{N}} * Q$) may have $\Theta(pq)$ edges in the worst case. As figure 15 shows, $P * Q$ need not be simply connected, even when P and Q are simple polygons. If P and Q are convex, however, $P * Q$ is a convex polygon with at most $p + q$ edges.

It is convenient to view each edge of a polygon as a vector directed counterclockwise around the polygon. Observe that the edges around a convex polygon are ordered by "slope" (direction angle). In the case of convex polygons, $P * Q$ has a simple characterization: its edges are those of P and Q, merged in slope order. More precisely, if we imagine the edges directed counterclockwise, then every edge e of P is translated by the vertex v of Q such that the direction of e lies in the exterior angle at v that is facing counterclockwise. See figure 16.

This fact allows us to compute the convolution $P^{\mathrm{N}} * Q$ using only $O(p + q)$ time and space. After that we can test in $O(\log(p + q))$ time whether P displaced by a given vector x

would intersect Q displaced by y, by simply checking whether $x - y$ is inside the convex polygon $P^{\mathrm{N}} * Q$. We conclude the following:

Theorem 8. *It is possible to pre-process two convex polygons P and Q of size n using only $O(n)$ time and space, so that any translated copies of P and Q can be tested for intersection in $O(\log n)$ additional time.*

Intersection without preprocessing. We may be tempted to rest on our laurels at this point. After all, don't we have to look at each edge of P and Q to know if they intersect? Surprisingly, the answer is no. By a suitable kind of binary search, we can test for intersection in time only $O(\log(p+q))$, *without any pre-processing.* Instead of precomputing the convolution $P^{\mathrm{N}} * Q$ explicitly, we compute each edge of the convolution only if and when it is needed.

Imagine that we move P horizontally to the left, all the way to infinity. We call the region swept over this way (including P itself) the *left shadow* of P, denoted by P_L. See figure 17. The *right shadow* P_R or P is similarly defined by sweeping P to the right. Obviously, $P = P_L \cap P_R$. It is easy to verify the following result:

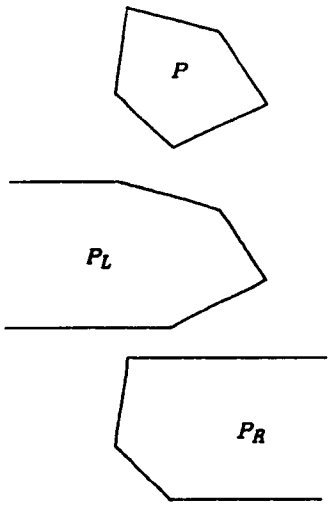

Figure 17.

Lemma 9. *The intersection $P \cap Q$ is non-empty if and only if both $P_L \cap Q_R$ and $P_R \cap Q_L$ are non-empty.*

Thus, to check whether $P^{\overline{x}}$ intersects $Q^{\overline{y}}$ we need only know how to check for intersections between a left shadow $L^{\overline{u}}$ and a right shadow $R^{\overline{v}}$, that is, between $(P_L)^{\overline{x}}$ and $(Q_R)^{\overline{y}}$, or between or $(Q_L)^{\overline{y}}$ and $(P_R)^{\overline{x}}$. By lemma 7, this is equivalent to testing whether $v - u$ lies in $L * R^{\mathrm{N}}$. Note that R^{N} is a *left* shadow, and that the convolution of two left shadows is also a left shadow. This is then the problem we will consider: given two left shadows A and B, discriminate a given point w against $A * B$.

As stated earlier, the edges of the convolution $A * B$ (viewed as vectors directed counterclockwise around the figure) are precisely those of A and B, merged in order of direction. Suppose the direction of an edge e from A lies between the directions of two consecutive edges f, g of B, both incident to the same vertex v. Then the edge e appears in the convolution $A * B$ displaced by the vector v. In the same way, each edge of B gets displaced by some vertex of A.

In what follows, *placing e in the convolution* means determining this vertex v, and hence the position of e in $A * B$. Let n_A and n_B be the number of edges in A and B, respectively, and let $n = n_A + n_B$ be the number of edges in the

convolution. If the coordinates of the vertices of each chain are stored in a linear array, in counterclockwise order, we can use binary search to place any edge e in the convolution, in time $O(\max\{\log n_A, \log n_B\}) = O(\log n)$. Once e has been placed, we can test w against its supporting line l and its endpoints p, q in $O(1)$ additional time. See figure 18. If w lies to the right of l, then w is outside $A * B$, and we can stop. If w lies in the left shadow of e, then it definitely lies inside $A * B$, and we can stop.

If neither case holds, then we must continue checking w against other edges of $A * B$; specifically, either the edges that follow e or those that precede e in counter-clockwise order, depending on whether w lies above or below e's shadow. Suppose e is an edge from A that was displaced by a vertex v of B, and w lies above its shadow; then the edges of $A * B$ we must consider next are those that follow e in A, plus those that follow v in B. The other cases are symmetrical. In other words, we have managed to transform the original problem into a similar (but smaller) one: discriminate w against the convolution of two convex left shadows, each defined by a string of consecutive edges from A or B. By repeating this process we will eventually reduce the convolution to a single edge, which either shadows w or leaves w to its right.

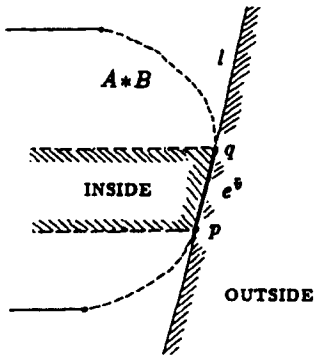

Figure 18.

How many iterations will this take? If the edge e we discriminate against is always the median edge of $A * B$, then at each iteration the problem size gets halved, and the number of iterations will be at most $\log_2 n$. Finding the median edge of $A * B$ is a bit tricky, given that A and B are stored in separate arrays. Nevertheless, by a binary search procedure it is possible to find this median in $O(\log n)$ time. We will not bother to give the details, however, since a much simpler method gives the same asymptotic result.

Note that it is not necessary to find the *exact* median of $A * B$ to guarantee a logarithmic number of steps. It suffices to choose e so that at each stage we always eliminate at least some fixed fraction α of the edges. In particular, if we let e be the median of the longest of the two chains (which can obviously be found i onstant time), then at least $1/4$ of the edges of $A * B$ must come before e, and at least $1/4$ must come after it. Therefore at each stage the size of the problem is reduced by a factor $\leq 3/4$. The number of stages will be at most $\log_{4/3} n$, which is bigger than $\log_2 n$ by a factor of $\log(4/3)/\log 2 = 2.409+$, but is still $O(\log n)$. Since each stage (placing e in the convolution and testing w against it) takes $O(\log n)$ time, it follows that testing whether w lies in $A * B$ can be done in $O((\log n)^2)$ time. We conclude the following:

Theorem 10. *With no preprocessing, it is possible to test whether two convex polygons of size n intersect in $O((\log n)^2)$ time.*

Intersection in O(log n) time. Surprisingly, we can still improve on this result. We will show that we can actually discriminate the point w against the convolution $A * B$ in time only $O(\log n)$. To do this, we cannot afford to spend time $O(\log n)$ on every test just to place the edge e in the convolution. Instead we wish to find a constant time test that allows us to get rid of some fraction of the edges of A or B. With this in mind, let us look at the convolution from yet another viewpoint.

Let a', a'' be the lower and upper endpoints of the chain A, and let b', b'' be those of B. The convolution $A * B$ will be a right-convex path on the plane from $a' + b'$ to $a'' + b''$. Now consider all possible ways to interleave the edges of A with the edges of B, while maintaining the relative order of any two edges from the same chain. Each interleaving defines a path on the plane from $a' + b'$ to $a'' + b''$ (since all paths consist of the same set of vectors, their total displacement is the same). Every path consistently moves in the general direction of increasing y, but may turn left and right several times. Only one path will be convex to the right: the one in which all edges are sorted by direction, that is, $A * B$.

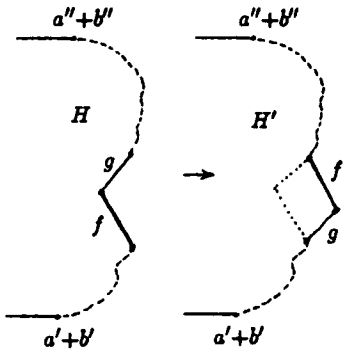

Figure 19.

Note that $A * B$ is also the rightmost of those paths, in the sense that no point on any other path lies to the right of $A * B$, and any other path has at least one point that is strictly to the left of $A * B$. To see why this is true, observe that if f and g are consecutive edges in a path H and are not in the right order, then swapping them will produce another valid path H' that lies to the right of H. See figure 19.

Now let f denote the median edge of A, and g the median edge of B. Without loss of generality, we can assume that f precedes g in slope order, and therefore also in the convolution $A * B$. Let A_H, A_L, B_H, B_L denote respectively the high and low halves of A and B that are separated by f and g.

Consider next the chain D which consists of $A_L * B_L$, followed by f, and g, followed $A_H * B_H$, in this order. See figure 20. Since the displacement of $A_L * B_L$ is the sum of the displacements of A_L and of B_L, the position of f in the chain D can be computed in constant time.

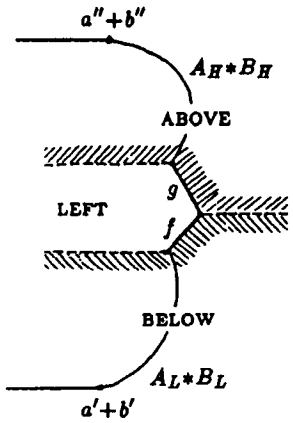

Figure 20.

Let us then test w against the edges f and g, positioned as in the path D. If the point w falls in region *LEFT* (see figure), then clearly w is to the left of the convolution, and we are finished. If w falls in region *ABOVE*, then what can we conclude? Observe that in the convolution $A * B$, the edge f and all the edges in A_L must lie below the dashed line. The reason is that f precedes g in slope order, hence it precedes

also all edges of A_H and B_H. Therefore, the path leading to f in the convolution is a subset of $A_L * B_L$, which is the path leading to f in D.

Therefore, if w falls in region *ABOVE*, we can discard f and all edges in A_L, since they cannot affect the classification of w. Similarly, if w falls in region *BELOW*, then in the convolution w will be below g and all edges of B_H, so these edges can be discarded. By this constant time test we are able to eliminate at least half of the chain A or half of the chain B. We can repeat this step until one of the chains (say, A) gets reduced to a single vertex v.

Note that we can eliminate only half of *one* chain, and we don't get to choose which one. Since the two chains may have very different lengths, we cannot guarantee that the size $n_A + n_B$ of the convolution will decrease by a constant fraction at each stage. However, after $O(\log n_A) + O(\log n_B) = O(\log n)$ such steps one of the two chains (say, A) will be reduced to a single vertex v. This means the convex shadow bounded by that chain is a single horizontal ray extending from v to the left. The convolution of the two chains is therefore the other chain (B) displaced by the vector v. Checking w against the convolution is then equivalent to checking $w - v$ against the chain B, which can be done in $O(\log n)$ additional time by a straightforward binary search. The total time for checking w against $A * B$ is therefore $O(\log n) + O(\log n) = O(\log n)$. So we have proved the following result:

Theorem 11. *It is possible to test if two convex polygons of size $O(n)$ intersect in time $O(\log n)$.*

The same result was obtained by Chazelle and Dobkin, using different methods [9].

To conclude this example, let's comment briefly on the form in which the answer should be presented. In many practical applications, just knowing whether P and Q intersect is not enough. We often need a *witness* or *certificate* that corroborates the answer: a common point if the polygons do intersect, or a separating line if they don't. Our algorithm for testing whether a point w lies in the intersection of two convex shadows A and B returns such a witness, namely an edge e of $A * B$ such that w is either in the shadow of e or to the right of its supporting line. By that time we also know the chain where e came from, and the vertex v of the other chain by which e was displaced. From these data we can easily recover a witness for the intersection or disjointness of the original polygons.

A6. Separator techniques

As in graph theory, the existence of "small" separators often leads to efficient divide-and-conquer algorithms. Perhaps the best known result of this kind in computational geometry is Chazelle's polygon cutting theorem [8]. That theorem states that a simple polygon admits of a diagonal that cuts it into two subpolygons such that (roughly) the larger is not more than twice as big as the smaller. See figure 21. (As usual, the size of a polygon is its vertex count, not its length or area.) This kind of balanced decomposition is crucial when we seek efficient divide-and-conquer methods.

The existence of such a diagonal for a simple polygon follows from a corresponding theorem for trees. Triangulate the polygon and look at the tree T that is the dual graph of the triangulation. The nodes of T have degree three; by a well-known result of graph theory, T has an edge whose removal cuts it in a balanced fashion. The dual of this edge is the diagonal we want. Chazelle showed also that such a cutting diagonal can be found in time proportional to the size of the polygon.

Suppose that our polygon P has n sides. If we cut P along a cutting diagonal and then recursively apply the technique to the two subpolygons thus formed, we will obtain a tree S which forms a *balanced hierarchical decomposition* of P. The leaves of this tree are the triangles of a triangulation of P, while internal nodes correspond to diagonals used in the triangulation. The whole process can be carried out in $O(n \log n)$ time. If all the produced subpolygons are saved within the appropriate node of S, then the total storage will also be $O(n \log n)$. Such a balanced hierarchical decomposition is very useful in computing shortest paths inside P [29], or in visibility questions inside P [11]. In most applications the redundancy involved in storing all the subpolygons can be removed with some cleverness so the storage used can be made linear in n.

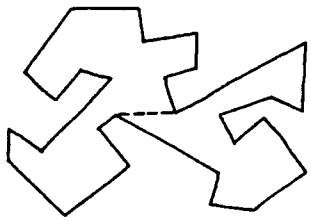

Figure 21.

A7. Decimation

Decimation is our name for a programming technique due to Nimrod Megiddo that has yielded surprisingly efficient algorithms for a variety of computational geometry and operations research problems. The technique has also been called *the Megiddo method* or *prune-and-search* by other authors. Some applications are computing the smallest enclosing circle [47] and the convex hull [39] of n points on the plane, and solving linear programming problems in spaces of fixed dimension [48].

Problems where the decimation technique is applicable seem to be those where the answer is some simple geometric object

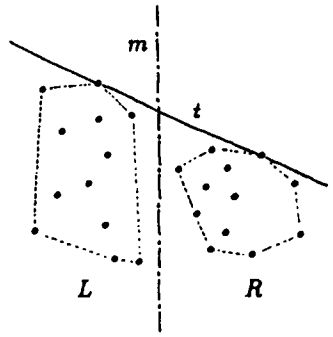

Figure 22.

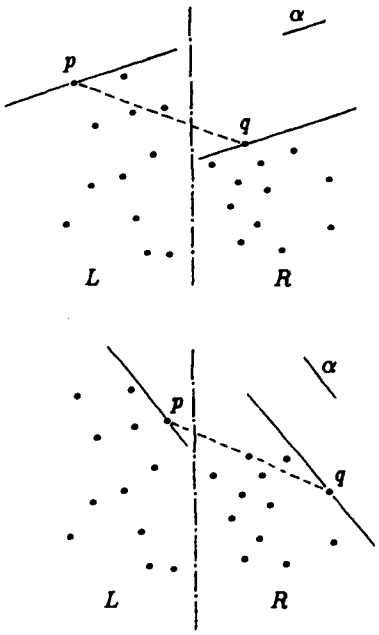

Figure 23.

that ultimately depends on only a few of the input data. The technique depends on the existence of some quick criterion that allows us to eliminate in linear time some fixed fraction of the input data. By repeatedly applying this process with the remaining data, we will be left eventually with a trivial problem that can be solved efficiently by some other method. If each pass reduces the size of the problem by a factor $\rho < 1$, the total time required by the algorithm will be $O(n + \rho n + \rho^2 n + \cdots)$, which is still $O(n)$.

A good example is that of computing the smallest circle enclosing n given points in the plane. Such a circle is always determined by either two or three of the given points. This classic problem of computational geometry was conjectured to have $\Omega(n \log n)$ worst-case cost, until Megiddo found a decimation method that on each pass eliminates a fixed fraction of the points which do not support the minimum circle [47].

The common tangent problem. We will describe here in detail another and somewhat easier example, the computation of the *common upper tangent* of two sets of points. The problem is this: we are given a collection P of n points in the plane, which is separated into two non-empty subsets L and R by a known vertical line m, with L on the left and R on the right. We wish to find a line t passing through one point from each subset, such that none of the given points lies above t. See figure 22. In other words, we want the upper exterior common tangent of the convex hulls of L and R.

If the convex hulls of L and R were known, the common tangents could easily be found in linear time [57]. However, computing the convex hull of n points costs $\Theta(n \log n)$ in the worst case [73]. Is this also a lower bound to computing the common tangent? The answer is no: as Kirkpatrick and Seidel showed, the upper common tangent of L and R can be computed in time linear in n, without knowledge of their convex hulls [39]. The key to their algorithm is a simple decimation criterion that in linear time allows us to eliminate a good many of the points that do not define the common tangent.

Let us fix for the moment our attention on lines of a particular slope α. We can compute in linear time a point p of L that is extremal for this slope, i.e. such that a line through p with slope α leaves all other points of L below it. We can do the same for R and obtain a corresponding extremal point q. Now if the line pq has slope less than α, then so must the common tangent t; similarly, if pq has slope greater than α then so does t; and if pq has slope equal to α then $t = pq$. See figure 23.

Now suppose the first case holds, so the slope of t is less than α. Let r, s be any two points of $L \cup R$, such that r is to the left of s and the line rs has slope *greater* than α. Then we

can conclude that t cannot pass through r, because a line of slope less than α through r must pass below s. See figure 24. The second case, where the slope of t is greater α, is entirely symmetrical: we can eliminate the second member of any pair r,s, with r to the left of s, if the line rs has slope less than α.

These remarks suggest the following decimation method. Pair up the n given points in an arbitrary way, and find the *median* slope α of the $n/2$ lines defined by those pairs. Now compute the extremal points p and q for the slope α, and eliminate one point from every pair that satisfies the criteria described above. It should be obvious why the median is a good choice: since half the pairs have slope less than α, and half have slope greater than α, then half the pairs will satisfy the criterion, no matter whether the slope of pq is greater or less than α. So, in either case we eliminate one point from half the pairs. Of course if pq has slope α then we are done.

It is possible to find the median in time linear in n using the rather elaborate technique of Blum and others [6]. The other operations clearly take $O(n)$ time. Therefore, in linear time we either stop, or eliminate $1/4$ of the original points. If we repeatedly apply this elimination process on the remaining points, we are guaranteed to find the common tangent, at a total cost of $O(n + (3/4)n + (3/4)^2 n + \cdots) = O(n)$ Note that we may eliminate a different number of points from L than from R, but this does not affect the analysis.

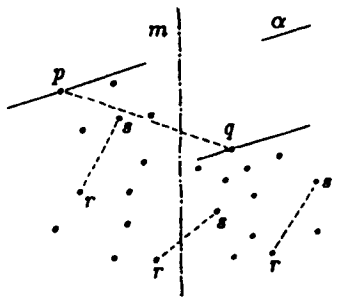

Figure 24.

Theorem 12. *The upper exterior common tangent of two vertically separated point sets can be computed in linear time.*

Kirkpatrick and Seidel used this result to obtain a planar convex hull algorithm that runs in time $O(n \log h)$, where h is the size of the hull [39]. The one step that makes the algorithm above impractical is finding the median. Unfortunately, such a step is almost always present in decimation algorithms. For practical purposes we can substitute a randomized median (or other quartile) finding method. This yields algorithms that are linear-time only in a probabilistic sense but are much simpler to implement.

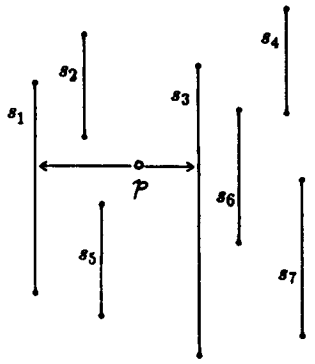

Figure 25.

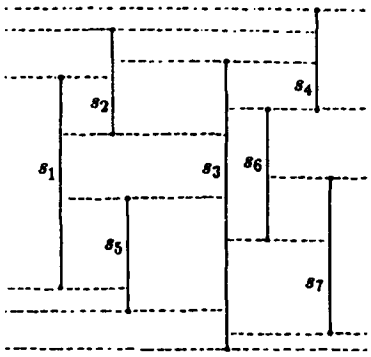

Figure 26.

A8. Hierarchical structures

Several geometric problems have been successfully solved by the following approach. Given an instance P of the problem, we may attempt to extract from it a "coarsened sample" P', another instance of the same problem such that (1) the size of P' is only a fraction of the size of P, and (2) a solution to P' yields a solution P with only little additional effort. If we can define and effectively construct such samples, we can solve P by sampling it, in turn sampling the sample, and so on, till we obtain an instance small enough to be solved by some trivial method. We then back up through the sample hierarchy we have constructed, obtaining the solution for increasingly finer samples, till we come to the original problem P.

David Kirkpatrick was the first to popularize such *hierarchical decomposition methods* in computational geometry. He used this approach in the first practical logarithmic point location method for planar subdivisions [38]. He and Dobkin also gave many applications to intersection questions about convex polyhedra [16, 17].

A segment visibility problem. Let's illustrate this hierarchical decomposition technique on a very simple problem. Suppose we are given a set S of n vertical line segments in the plane. For ease of exposition we assume that the x-coordinates of all these segments are distinct, so no overlaps among them are possible. We are interested in organizing these segments into a structure that allows us to answer efficiently queries of this form: given a point p in the plane, report the segments in S (if any) immediately to its left and to its right. In other words, if we extend horizontal rays from p left and right, we wish to know the first segments of S that will be hit. See figure 25.

Notice that this is a point location problem. From each segment endpoint, let's extend rays to the left and right till we encounter another segment of S. The segments of S together with these rays form a rectangular partition of the plane. See figure 26. All points lying in the same rectangle of this partition yield the same answer to our "left-and-right neighbor" query. By sweeping the plane with a horizontal line, we can construct this rectangular partition in $O(n \log n)$ time. We could then solve our problem by triangulating the regions and using Kirkpatrick's original point location method. It is preferable, however, to apply the sampling idea directly to the original problem, so as to avoid the complexity of the triangulation algorithm and the numerical difficulties involved in testing a point against oblique lines.

Sampling the segments. How are we to define a sample of S? Let us name the segments of S as $s_1, s_2, \ldots, s_n$ in some order. We say that segment s_i "sees" segment s_j if there is some ordinate at which these two vertical segments are adjacent in the horizontal ordering. In more informal terms, "seeing" means that if every point of s_i sent out horizontal light rays then some of them would hit s_j. (Note that the relation is symmetric, even though this description isn't). In the example collection shown in figure 26, s_2 and s_4 see each other, while s_1 and s_6 do not. This visibility relation between segments can be useful to us because, if we remove from S a collection of mutually invisible segments to obtain a reduced set S', then the answer to the left-and-right neighbor problem in S' is not too different from that in S. Specifically, if we think of the deleted segments as having become transparent, then the rays we emit from our query point can only pass through at most one transparent segment before they encounter some segment of S' (or go off to infinity).

Assume that we have preprocessed our collection of segments as mentioned above so we have for each segment endpoint its left and right neighbors. It is most useful to record this information on the neighbors: for each segment s we maintain two lists ordered in y that describe the segment endpoints for which s is the left or right visible segment, respectively. These are called the *visibility lists* for s and their total length is called the *degree* of s. We require that a left-and-right neighbor query from a point p report not only the two segments stopping the rays from p but also the specific interval in the right visibility list of the left neighbor and the left visibility list of the right neighbor where these rays fall. Note that the total size of the visibility lists, summed over intervals, is at most $4n$; furthermore, these lists can be set up in $O(n \log n)$ time by sweeping S horizontally.

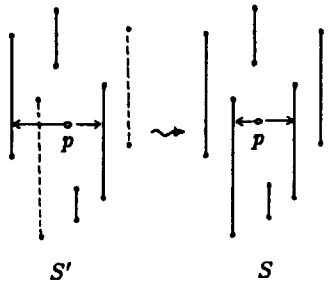

Figure 27.

Now the following sampling process suggests itself: remove a large collection of mutually invisible segments from S to obtain a coarsened subcollection S'. For this subcollection S' we store separately the original visibility lists in S of the segments in S'. We annotate these lists by giving for each segment s in S', and for each pair of adjacent y values in the left and right visibility lists of s, the name of the neighboring deleted segment, if any. This is the deleted segment that lies immediately to the right or left of the appropriate interval in the lists for s. We refer to these auxiliary structures as the *support* of S'. If we have the support of S', then we can take the answer to a left-and-right neighbor query in S' and compute the corresponding answer in S in constant extra time: we need to check only the single deleted segment of S that might have gotten in the way in each direction. See figure 27.

From the visibility lists for S we can derive the visibility lists for S' by a pretty straightforward process. When segment s is to be deleted, its left visibility list has to be merged with those of its right neighbors, and correspondingly for the right visibility list. These neighbors are visible from s, so they are not themselves deleted. The merging process can be carried out in time proportional to the sum of the lengths of the two visibility lists for s. Once we have the proper visibility lists for S' itself we are back where we started, so we can repeat the whole sampling process over again.

Selecting the sample. For this approach to work we have to make sure that we can always find a sufficiently large set of mutually invisible segments to remove, and then do the list updating efficiently. We have already remarked that the total size of all the visibility lists for all segments is less that $4n$. Thus the average segment has degree less than 4. Equivalently, there are at least $n/2$ segments of degree less than 8; let us call their set T. From among the segments in T we would like to extract a large set of mutually invisible segments. So let us take the first segment from T, place it in a bag, and then delete from T each of the at most 7 other segments that it sees. Now continue this process till no more segments are left in T. Since at each step at most 8 segments are removed altogether, there will certainly be at least $n/(2 \times 8) = n/16$ mutually invisible segments placed in the bag. These segments form the set that we remove from S to obtain S'. Note that all the removed segments have degree less than 8.

The process of computing the segments to be removed from S clearly takes linear time. The auxiliary support structure for S' can also obviously be stored in linear space. Now the set S' is at most 15/16 of the size of S. Furthermore, its visibility structures can be computed from those of S in time linear in n, since all the merging steps that have to be done involve constant size lists.

The hierarchical structure. Now our solution is clear: we iterate this sampling process till we get to some constant-size collection of segments R. We store the support structures for all the intermediate samples up to and including R. The cost of computing and storing each level is proportional to the number of segments in it, which is at most a constant fraction $\rho = 15/16$ of the segments in the previous level. Therefore, the total cost (time and space) of all these samples is $O(n + \rho n + \rho^2 n + \cdots) = O(n/(1-\rho)) = O(n)$.

In order to answer a left-and-right neighbor query we use an exhaustive search on R. Then, using the support structures we have saved, we back up through this hierarchy of samples,

at constant cost per level. From the previous discussion it is apparent that the number of levels is $O(\log n)$, which is also the cost of answering a query. We have shown the following:

Theorem 13. *It is possible to preprocess n vertical segments in $O(n \log n)$ time and $O(n)$ space so that any left-and-right neighbor query can be answered in $O(\log n)$ time.*

Rectangular point location. A minor variation of the technique presented here can be used for solving the problem of point location in a rectangular subdivision of the plane. This problem has obvious applications to VLSI design tools and multi-window user interfaces. We look at the vertical sides of our rectangles and then lump them together into maximal vertical segments. The rectangular partition defined by these vertical segments is almost the original subdivision. It only fails to account properly for vertical towers of rectangles with identical width. However, those are not difficult to incorporate into the technique shown. The result is a point location algorithm as asymptotically efficient as that of Kirkpatrick, but which stays entirely in the rectangular domain and does not require any coordinate operations other than simple comparisons.

Part B: Data Structures

In this part we focus on a number of data-structuring techniques that have been found especially useful in designing efficient geometric algorithms.

B1. Fractional cascading

In computational geometry many search problems and range queries can be solved by performing an iterative search for the same key in separate ordered lists. In this section we demonstrate a technique known as *fractional cascading* [10] for efficiently solving such problems. Fractional cascading is a data-structuring technique for solving the *iterative search* problem, which we formulate as follows: let G be a directed graph whose vertices are in one-to-one correspondence with a set of sorted lists; given a query consisting of a key q and a connected subgraph π of G, search for q in each of the lists associated with the vertices of π. This problem has a trivial solution involving repeated binary searches, at a logarithmic cost per list. Fractional cascading establishes that it is possible to do much better: under the assumption that the graph G has bounded degree, we are able to organize the set of lists so that we have to pay the logarithmic cost only once, when searching the first list, and then pay only a fixed amount to look up q in each of the remaining lists. This can be done using only a linear amount of extra storage. These bounds are clearly best possible.

The interval stabbing problem. We now consider the following problem: given a collection of n intervals on the real line, organize them into a data structure so that the number of them containing an arbitrary query point can be efficiently computed. We would like a method that uses linear space and solves this problem in time $O(\log n)$. In the sequel we assume that the given intervals have distinct endpoints.

We organize our intervals into a tree T using an idea due to McCreight [45,46]. First we normalize all the endpoints so that they become the rationals $1/(2n+1)$ to $2n/(2n+1)$, in left-to-right order. These rationals are all in the interval $(0,1)$. We put in the root of T the collection of intervals containing the rational $1/2$. In the left child of the root we put the intervals not already placed that contain the rational $1/4$, and in the right child of the root we put those not already placed that contain $3/4$. Note that any interval containing both $1/4$ and $3/4$ already contains $1/2$, so it has been allocated in the root node. We continue in this process until every interval has been allocated to exactly one node of the tree T thus formed. The

normalization assumption implies that the depth of T will be only $O(\log n)$.

Each node v of T thus ends up containing a collection of intervals. We construct two ordered lists from these intervals and store them in v. One list contains all the left endpoints of the intervals in v sorted by x value, and the other the right endpoints, similarly sorted. Clearly the total space that the tree structure takes, together with these ordered lists, is $O(n)$.

Processing a query. A query asking for the number of intervals containing some point p can then be answered as follows. We start at the root and traverse a path down the tree, essentially emulating a binary tree search for p, where the key value of the root is $1/2$, of its left child $1/4$, etc. For each node v visited we compare p to its key value. If p is greater than the key value, then we locate p by a binary search among the right endpoints of the intervals stored with v. Once we know the location of p, we can easily obtain the number of intervals in v containing p. Then we continue the search with the right child of v. We perform the symmetric operations if p turns out to be less than the key value in v. This method works because once we have (say) branched right from a node v we can be sure that no interval in a node that is in the left subtree of v can possibly contain p.

The cost of this method is $O(\log^2 n)$: we have $O(\log n)$ levels to traverse, and at each level a binary search to perform. How could we reduce the $O(\log^2 n)$ overhead to only $O(\log n)$?

Using fractional cascading. Here is where fractional cascading comes in. The underlying graph G is the tree T. The sorted lists are the ordered end point lists (we treat the list of left endpoints entirely separately from the list of right endpoints). The degree is bounded by three. So here is what we can do. Consider first the left endpoint lists only. Each leaf of T extracts *every other one* of its list elements and merges them into the list of its father. A node with two leaves as children has its own list merged with that of the "sampled" lists from each of its two children. Now each of these fathers with two children leaves in turn samples his own new list by taking every other element and sends that sample up to be merged with the list of his own father. This process continues all the way up the tree: whenever a node has merged with samples from both of his children, he samples himself and sends the sample to his father. Because of the acyclic structure of the tree T this process eventually terminates. When it has done so, each node of v contains an augmented list of left endpoints. This list for v still contains all the left endpoints of intervals stored in v, but it also contains a

sample of those in its children, a smaller sample of those in its grandchildren, and so on.

Analysis. First of all it is clear that if we know where we are in the augmented list of a node v, then we can easily compute where we are in the regular list of v. Second, if we know where we are in the augmented list for v, we nearly know where we are in the augmented lists of either of its children: at most one comparison with an unsampled value in the interval that we are in will tell us our position exactly in a child. This is an important accomplishment, because once we have done a binary search to locate p in the augmented list of the root, we can then descend the tree T and locate p in the relevant lists at only constant additional cost per node visited. Thus this propagation of samples has made neighboring lists sufficiently similar that the search overhead is reduced to only $O(\log n)$: an initial binary search, and then constant cost per level.

But what about storage? Hasn't all the propagation of samples increased storage use significantly? The answer is no. For each list, half of it went to its father, half of that went to the grandfather, and so on. So that the total size of all the samples of a particular list floating around is a geometric series in the original length of the list, and therefore linear in that length. This argument is not quite rigorous, but it conveys the intuition why this sampling process has in fact not more than doubled the storage used. For a rigorous argument the reader is referred to the paper by Chazelle and Guibas [10]. This notion of propagating cascaded fractions of the whole forms the essence of the idea behind fractional cascading. Thus we have shown the following:

Theorem 14. *In $O(n \log n)$ time it is possible to organize n intervals into a linear space data structure such that the number of intervals containing a query point can be found in time $O(\log n)$.*

B2. Finger trees

A classical data structure that has found a number of important uses in computational geometry is the *finger search tree.* A finger search tree is like an ordinary balanced tree used to implement the dictionary operations of *find*, *insert*, and *delete.* Some additional structure is provided, however, so that these operations are especially efficient in the vicinity of certain preferred positions, indicated by *fingers.* Let n be the size of the dictionary; while in an ordinary balanced tree each operation

requires time $O(\log n)$, in a finger search tree the time is only $O(\log d)$, where d is the distance from the finger where the search begins. Thus if there is locality of reference, by keeping a finger in the active region of the key space we can speed up all three operations. The finger search tree structure was introduced by Guibas, McCreight, Plass, and Roberts [30] and has since been further developed by many researchers.

For our discussion here we will need the version described by Tarjan and Van Wyk under the name of *homogeneous finger search tree* [72]. The structure consists of a 2,4-tree (or, equivalently, a red-black tree) with additional internal links from each node to its parent, and to its left and right neighbors on the same level of the tree. See figure 28. These extra pointers allow fast searches starting from *any* given node. More specifically, the cost of locating an item that is d items away from a given node (in the linear order of nodes) is only $O(\log d)$. In fact, if we maintain two fingers pointing at the first and last leaves of the tree (or, alternatively, if we make the left/right neighbor links circular in each level), we can reduce this to $O(\log \min\{d, n-d\})$, where n is the total number of nodes in the tree. To achieve this bound, we start searching simultaneously in both directions from the given node, wrapping around when we reach either end, and stopping both searches as soon as one of them succeeds.

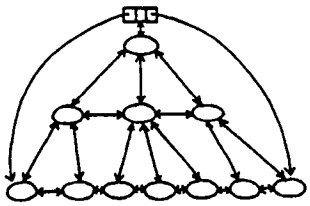

Figure 28.

Homogeneous finger search trees also allow us to perform efficiently a few other list operations, such as insertion, deletion, splitting, and concatenation. In particular, once we have located the position of a given key in the tree, we can insert or delete the corresponding node in only $O(1)$ additional time. Also, given any pair x, y of leaves, we can split the tree into two smaller trees, with all the nodes between x and y in one tree and the balance in the other, in time $O(\log \min\{d, n-d\})$, where d is the number of nodes between the two leaves. Conversely, if we are given two trees of sizes d and $n-d$ whose keys lie in disjoint segments of the key space, we can join them into a single tree in the same time bound. For a careful description of these algorithms see the appendix to the triangulation paper of Tarjan and van Wyk [72].

Note that these are *amortized* bounds. What this means is the following. Consider any sequence of searches, insertions, deletions, splits and joins starting with an empty tree. Let n_i be the total size of the operands for the ith operation, and d_i be either the distance between the starting node and the affected node (for find, insert, delete), or the size of the first operand (for join), or the size of the first result (for split). What the amortized bounds say is that the total cost of the whole sequence

will be $O(\sum_i \log \min\{d_i, n_i - d_i\})$, even though the ith operation by itself may cost much more than $O(\log \min\{d_i, n_i - d_i\})$. We will discuss amortized analysis in more detail in section C3.

Jordan sorting. The application which we will use to demonstrate the use of finger search trees is known as *Jordan sorting.* Suppose we are given a simple polygon P of n sides and an (infinite) straight line l. We wish to compute the intersections of P and l, sorted in the order in which they occur along l. It is possible to do this in only $O(n)$ time, by an elegant algorithm due to Hoffman, Mehlhorn, Rosenstiehl, and Tarjan [36]. In what follows we outline their solution.

Consider what happens to one side of l; the other side is symmetric. The line l breaks up the polygon into a bunch or arcs. See figure 29. These arcs are either disjoint or nested, since the polygon is simple. We represent the containment relationships by a tree S, as shown. Each node is an arc and its children are all arcs directly nested within it, sorted by the order in which they occur along l. The plan is to follow the polygon P, and, as each arc of P on the current side of l is completed, update the tree S. We do this on both sides of l simultaneously, with pointers connecting every arc endpoint on one side to the corresponding arc endpoint on the other side. When we are finished we can just make a symmetric order traversal of S and derive the sorted order of intersections of P with l.

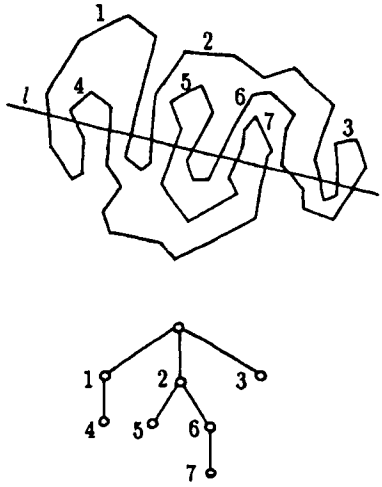

Figure 29.

So how do we update S? Whenever we start processing a new arc γ, we look for the innermost pair of previously processed arcs α, α' whose endpoints bracket the starting point of γ. As we will see, while processing the previous arc (on the other side of l) we will have determined the nearest arcs that bracketed its exit point; therefore, we can get α and α' in $O(1)$ time by following the pointers mentioned above. If the neighbors are nested, that is, they are parent and child in S, then the outermost of the two will become the father φ of the new arc γ. If they are not nested, they must be siblings in S, and their common father φ will become the father of γ. See figure 30.

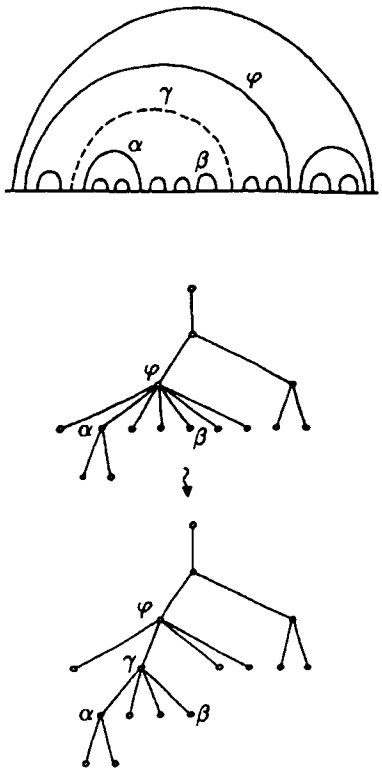

Figure 30.

Next, we follow γ until it reaches l again. The point where this happens must lie in the gap between two children β, β' of φ, or between φ and its first or last child. We locate β and β' in the children list, and then we update S by inserting the new node γ as a child of φ, and demoting all children of φ that are enclosed by γ (α through β in the figure) to be children of γ. In order to do these operations efficiently, we represent the children list for each node of S as a homogeneous finger search tree. (These tree structures should not be confused with the tree S. Rather, we should think of each finger tree as a linear list of children, with some hidden machinery that allows us to

efficiently perform certain search and update operations on the list.)

Analysis. It remains to analyze the complexity of this algorithm. Observe that the insertion of each arc γ requires locating its exit point among the children of φ, splitting off all the enclosed children (if any), and inserting γ in their place. The problem is finding an upper bound to the total cost of these operations.

Again, let's consider only what happens on one side of the line l, and let S be the *final* nesting tree for this side. Now consider the situation after the algorithm has just added some arc φ to the tree. It is helpful to think that all the arcs in S are initially "penciled in" in the plane, and then redrawn in ink in the order in which the algorithm inserts them. Suppose φ has m descendants, excluding itself, in the final tree S. In other words, φ encloses m other arcs, penciled or inked, arbitrarily nested. Suppose also that of these descendants k are still penciled in. We call the quantity $c = m + k$ the *complexity of* φ. We wish to determine the maximum cost $T(c)$ (as a function of the complexity c) for completing the subtree rooted at φ, that is, for adding the arcs that are descendants of φ in S but haven't been processed yet. Note that some of the descendants of φ may have been inked before φ; we aren't interested in how much their insertion cost, nor in how much it cost to insert φ—we just care about the cost of completing the inking process. Note also that the cost of inking the arcs under φ is not affected by any arcs, inked or not, that lie outside φ; therefore, we can ignore them completely in the determination of $T(c)$.

Now let γ be the first descendant of φ to be added after φ; suppose that γ has p, $p < m$, descendants, of which q, $q < k$, are still penciled in. The complexity of γ is then $d = p + q < c$. Since we are representing the inked children of φ by finger trees, if φ currently has i inked children, and γ covers j of them, the (amortized) cost of adding γ is $O(\log \min \{j + 1, i - j + 1\})$. Obviously, the number j is no greater than the total number p of descendants of γ in S. Similarly, $i - j + 1$ is no greater than $m - p$, the descendants of φ that are not descendents of γ. Since $p \leq p + q = d$, and $m - p \leq (m + k) - (p + q) = c - d$, the cost of adding γ is at most $O(\log \min \{d + 1, c - d\})$.

Once γ is inked, what happens to the arcs below γ is independent of what happens to those between γ and φ. The cost of finishing the former is by definition at most $T(d)$. The cost of finishing the latter would be the same if the descendants of γ did not exist, i.e. it is as if φ enclosed only $m - p$ arcs, of which $k - q - 1$ were still penciled in (the -1 is there because we just inked γ). Since $(m - p) + (k - q - 1) = c - d - 1$, this cost

is at most $T(c-d-1)$. Therefore we can write the following recurrence for the cost $T(c)$:

$$T(c) \leq \max_{0 \leq d < c} \left(T(d) + T(c-d-1) + O(\log \min \{d+1, c-d\})\right).$$

The solution to this recurrence is $T(c) = O(c)$ [49, vol. I]. Since the complexity of the first arc is less than twice the number of arcs n, the cost of building the entire tree S is only $O(n)$. We conclude the following:

Theorem 15. *The intersections of a simple polygon and a straight line, in the order in which they occur along the line, can be computed in linear time.*

B3. Persistent data structures

When we discussed the plane sweep technique in section A4, we observed that it transforms a static two-dimensional problem into a dynamic one-dimensional one. Objects enter the active list as the sweep line hits them, and exit the list as the sweep line moves past them. In some sense, the history of the active list represents all the data present in the original problem. Some geometric problems can be solved efficiently by encoding this history in a compact data structure that allows fast access to any of the active lists recorded during the sweep. Noe that this is in a sense the opposite of the plane sweep paradigm, for we are transforming a dynamic one-dimensional problem into a static two-dimensional structure.

In general, a *persistent data structure* is one that accepts an arbitrarily long sequence of updates, but is able to remember at any time all its earlier versions. Driscoll, Sarnak, Sleator, and Tarjan have given general techniques for taking ordinary ephemeral data structures and making them persistent [19]. An especially useful example is that of *persistent sorted sets.* In such a set we maintain a linearly ordered set of items. Items are always inserted and deleted from the "last version" of our sorted set. Queries as to the position of a particular item, however, can be made either in the present or in the past: we are allowed to ask for the place of an item in the sorted list as it was in some earlier version. If the current list contains n items and a total of m updates have been made, then a structure proposed by Sarnak and Tarjan [62] achieves update time of $O(\log n)$, access time to any version in the present or past in $O(\log m)$ time, and needs $O(1)$ amortized space per update, starting from an empty set.

Application to point location. Persistent sorted lists can be the basis of an alternative point-location algorithm for planar subdivisions. Let S be a subdivision of the plane into polygonal regions with a total of n edges. We assume that no edges of S are vertical. Let's cut the plane into a sequence of vertical slabs, by drawing a vertical line through every vertex of S. Within each slab S looks like a stack of trapezoids and triangles separated by line segmens that cut completely through the slab, and thus have a well-defined top-to-bottom order. See figure 31. If we could afford to represent the segments in each slab by a separate sorted list, then we could locate a given query point x in $O(\log n)$ time, by first locating the slab containing x, and then locating the two consecutive segments in that slab that bracket the point x.

Unfortunately storing the slabs separately would require $\Theta(n^2)$ storage in the worst case. However, we can reduce the space by noting that the sorted sets of line segments that form the trapezoid bounding edges in contiguous slabs are similar. Consider sweeping the plane from left to right by a vertical line. While the sweep line is inside a slab, the active list contains precisely the segments that cross the slab. As we sweep over the boundary from one slab to the next, some segments are deleted from the activel list and some are added. Over the entire sweep, there will be $O(n)$ insertions and deletions, one insertion and one deletion per edge of S. Therefore, if we represent the active list as a persistent sorted set of segments, by the end of the sweep this data structure will use only $O(n)$ space, and will allow us to search any slab in $O(\log n)$ time.

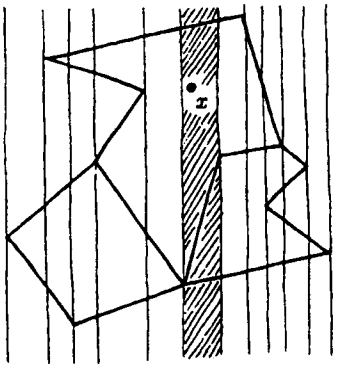

Figure 31.

Theorem 16. *Given a polygonal subdivision S with n edges, we can build in $O(n \log n)$ time and $O(n)$ space a search data structure that allows us to find the region of S containing an arbitrary query point x in only $O(\log n)$ time.*

Part C: Analysis Techniques

For the major part, the techniques used for the analysis of geometric algorithms are not intrinsically different from those used in the analysis of algorithms in general. It is often the case, however, that some of the quantities needed in the analysis depend on non-trivial results of topology and combinatorial geometry. Below we present two such results, having to do with the length of Davenport-Schinzel sequences (section C2), and the total complexity of the faces incident to a line in a line arrangement (section C1), which have been used recently in the analysis of many geometric algorithms.

In the theoretical kind of computational geometry that we are considering in this paper, the efficiency of an algorithm is most often measured by its asymptotic worst-case time and space requirements, viewed as a function of the input size, output size, or some similar measure of the problem's complexity. In fact, the large variability in the output size of many geometric algorithms makes this quantity an important parameter of the analysis. The reader is referred to Seidel's dissertation [64] for many examples of output-sensitive algorithms. An increasingly common tool is amortized analysis, which we already used in section B2 to derive a linear bound for Jordan sorting. In section C3 we show a much simpler application of this technique to the analysis Graham's convex hull algorithm.

Average-case analysis of geometric algorithms is rarely done, and few general-purpose techniques have been developed for this area. Section C4 shows one of the few results obtained so far, an asymptotic analysis of the expect number of vertices on the convex hull of n points uniformly distributed in a triangle.

C1. Exploiting results from combinatorial geometry

In the analysis of algorithms in computational geometry one often must make use of results of a combinatorial nature dealing with geometrical entities. Such results properly form the realm of a field known as *combinatorial geometry*. Grünbaum's book on convex polyhedra [28] is a good example of a work in this area.

We illustrate the use of such techniques in constructing the *arrangement of n lines* in the plane, that is, the subdivision of the plane determined by n given lines. An interesting combinatorial fact about such arrangements is the following: suppose we look at all faces of the arrangement adjacent to one of the lines in the arrangement, the so-called *horizon* of the line. Then the total number of edges on all these faces is only $O(n)$. Notice

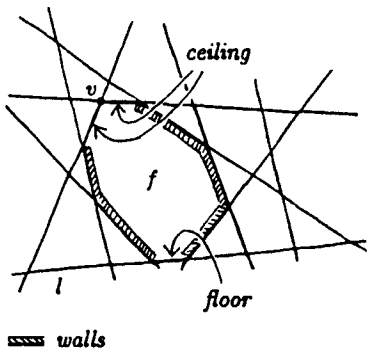

Figure 32.

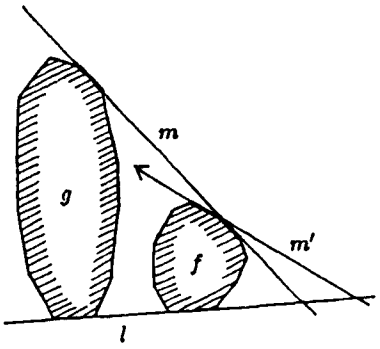

Figure 33.

that even one face can have n edges, so this bound is not as trivial as it may seem.

Based on this combinatorial fact we can design an algorithm for building the arrangement of the n lines by adding them in one at a time. In order to add a line l, we start at the appropriate infinite face R (one of the two, actually) and find the unique edge e through which l exits R. Then we cross over to the region on the other side of e and proceed in the same way. At each step, we find the edge where l exits the current region by simply walking along the boundary of the latter and testing each edge against l. Clearly, the total cost of this process is proportional to the total number of edges on the line's horizon, which as we have said above is only $O(n)$. Therefore the total cost (in both time and space) for inserting all n lines is $O(n^2)$. If our aim is to build and save the arrangement, then this is clearly best possible, as the arrangement will contain $\Theta(n^2)$ vertices, edges, and regions if the lines are in general position. This algorithm was given by Chazelle, Guibas and Lee [12], and Edelsbrunner, O'Rourke and Seidel [22].

Now we prove the combinatorial result mentioned above. We assume that the arrangement contains no parallel lines. Let l be the chosen line. Consider first the faces of the arrangement that are adjacent to and above l. Since the other $n-1$ lines cut l into n segments, there are exactly n such faces. For each face f, let v be the vertex furthest from l; That point is unique, by our non-parallelism assumption. Call the edge that lies on l the *floor* of f. The floor and the vertex v separate the remaining edges into the *left* and *right sides* of f. The edges adjacent to v are the *ceiling* of f. (If the face is unbounded, its ceiling is by definition the one or two edges that go off to infinity). The edges that are neither floor nor ceiling, if any, are called the *walls* of f. See figure 32.

We claim that each line of the arrangement can occur in the wall of at most two faces, once as left wall and once as right wall. To see why, suppose a line m occurred twice as a right wall, on two faces f and g. Without loss of generality, suppose f is the one closer to the intersection of m and l. See figure 33. Let m' be the next line on the boundary of f, counterclockwise from m (which may be either a wall or a ceiling of f). Since m and m' are on the right side of f, we conclude m' meets l to the right of m. Hence, to the left of its intersection with m the line m' lies always between l and m. Therefore it must cut through the face g, a contradiction.

A similar argument holds for left wall occurrences. We conclude that the total number of *wall* edges in all regions under consideration is at most twice the number of lines, that is, $2n-2$. Actually, the total is at most $2n-4$, since the line that crosses

l furthest to the left cannot appear as a left wall, and symmetrically for the right. Moreover, each face has one floor edge and at most two ceiling edges (with the first and last regions having only one). The total number of edges in all those regions is therefore at most $5n - 6 = O(n)$. (This result can be proved also by a Davenport-Schinzel argument, or through the use of Theorem 17; details are left to the reader.) We conclude that

Theorem 17. *The arrangement of n lines in the plane can be built in $O(n^2)$ time and space.*

C2. Davenport-Schinzel sequences

Davenport-Schinzel sequences are interesting and powerful combinatorial structures that arise in the analysis and calculation of the lower envelope of collections of functions, and therefore have applications in many geometric problems that can be reduced to the calculation of such an envelope. Roughly speaking, an (n, s) Davenport-Schinzel sequence is a sequence composed of n symbols with the properties that no two adjacent elements are equal and that it does not contain as a subsequence any alternation of two distinct symbols of length $s + 2$ (e.g. an $(n, 3)$ sequence is not allowed to contain any subsequence of the form $a \ldots b \ldots a \ldots b \ldots a$). The main goal in the analysis of these sequences is to estimate their maximal possible length for any given values of the parameters n and s.

The importance of Davenport-Schinzel sequences lies in their relationship to the combinatorial structure of the lower envelope of a set of functions. Let $f_1, f_2, \ldots, f_n$ be n real-valued continuous functions defined on the real line, having the property that each pair of them intersect in at most s points. The graph of their lower envelope f, defined by $f(x) = \min_i f_i(x)$, is the concatenation of a finite number of arcs, where the kth arc belonging to some function f_{i_k}. The sequence of indices $i_1, i_2, \ldots, i_k$ from left to right is an (n, s) Davenport-Schinzel sequence. Conversely, any (n, s) Davenport-Schinzel sequence can be realized in this way for an appropriate collection of n continuous univariate functions each pair of which intersect in at most s points.

The crucial property of these sequences is that, for a fixed s, the maximum length $\lambda_s(n)$ of an (n, s) sequence is "practically linear" in n. More precisely, $\lambda_1(n) = n$, $\lambda_2(n) = 2n - 1$, and $\lambda_3(n) = \Theta(n\alpha(n))$, where $\alpha(n)$ is an extremely slow-growing function of n, the functional inverse of Ackermann's function. Similar bounds have been proved for $\lambda_s(n)$ when $s > 3$ [34, 67, 69].

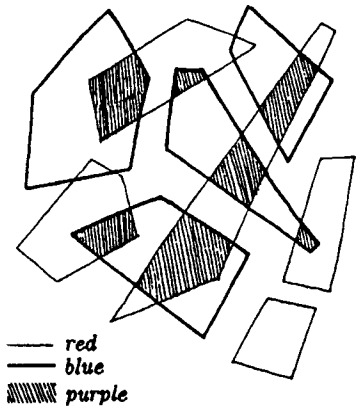

Figure 34.

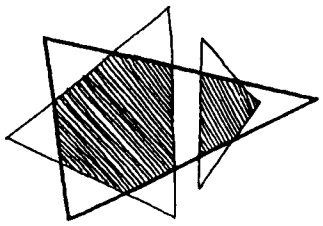

Figure 35.

This property makes Davenport-Schinzel sequences a useful and powerful tool for solving numerous problems in discrete and computational geometry. Indeed, the original motivation for the introduction of these sequences by Davenport and Schinzel was a geometric problem arising in analysis of the combinatorial properties of the pointwise maximum of a collection of solutions to a linear differential equation. In the last few years these sequences have been extensively studied by Micha Sharir and various coworkers. They have found application to a diverse set of geometric problems, including an analysis of the exterior boundary of the union of (intersecting) segments in the plane, the calculation of Euclidean shortest paths in 3-space amidst polyhedral obstacles, the estimation and computation of the configuration space used in solving various motion planning problems amidst polygonal obstacles, the analysis of certain time-varying geometric configurations, and many others [3, 63, 68, 70].

Complexity of red-blue intersections. We will illustrate the use of Davenport-Schinzel sequences on the following problem. Suppose we are given two sets of convex polygons, "red" and "blue," such that no two polygons with the same color intersect. See figure 34. A red and blue pair may have points in common, in which case their intersection is a "purple" convex polygon. Observe that the purple polygons are pairwise disjoint. Let the red and blue polygons have total size m and n, respectively (where size = number of edges). Now suppose we take a fixed number k of purple polygons. Our problem is to estimate the maximum total number of edges $\sigma(m,n,k)$ of these k polygons.

The difficulty here is that a red or blue edge may give rise to many purple edges. It is easy to see that a purple region P can have $m+n$ sides (exactly) in the worst case, since each red or blue edge contributes at most one edge to P. On the other hand, the number of purple polygons can be as high as $\Theta(mn)$ in the worst case (consider for example a grid of $m/4$ vertical red stripes and $n/4$ horizontal blue ones), but then their total size is only $\Theta(mn)$ instead of $\Theta(mn(m+n))$. That is, one purple region may have many edges, but if we take a lot of regions their average size will be quite small. For intermediate values of k, the answer is far from obvious. For example, figure 35 shows that $\sigma(6,3,2) \geq 11$. Can we get $\sigma(m,n,2) \geq c(m+n)$ for some $c > 1$? Surprisingly, the answer is no. It turns out that $\sigma(m,n,k) \leq m+n+4(k-1)$ for all $m,n \geq 3$ and $k > 0$.

We will now prove this result. To avoid confusion, let's call the purple regions *cells* and their sides *arcs*, and reserve the name *edge* to those of the red and blue polygons. So, let K be some set of k cells. Let us fix our attention on some particular red polygon P_i with m_i edges. Let K_i be the set of those cells

from K that are contained in P_i, and k_i the number of such cells. See figure 36. Let us classify the arcs of K_i as "red" or "blue" depending on their origin. We wish to show that the number of red arcs in K_i is at most $m_i + 2(k_i - 1)$.

Label each red arc of K_i with the name of the unique blue polygon that contains it. Now consider the sequence of labels we encounter as we go once around the polygon P_i. Of course several arcs may carry the same label, since two convex polygons may weave in and out of each other arbitrarily often. However, between any two *consecutive* arcs with the same label the polygon P_i must have at least one vertex. We "charge" the second of the two arcs to that vertex, and delete the corresponding entry from the label sequence. Clearly, every vertex of P_i will be charged at most once, which means at most m_i arcs are accounted for this way.

How many red arcs may still be unaccounted for? Here is where the Davenport-Schinzel theory comes to the rescue. Consider the sequence of labels remaining after the deletions above. Each letter may still appear several times, but not twice in a row. Moreover, we cannot have two pairs of repeated letters in the pattern $\ldots a \ldots b \ldots a \ldots b \ldots$. If that occurred, then convexity would imply that the blue polygons a and b intersect, which is a contradiction. See figure 37. This means the string of unaccounted red edges is a circular Davenport-Schinzel sequence on k_i letters with parameter $s = 2$. As mentioned earlier, the length of such a sequence is at most $2(k_i - 1)$. We conclude K_i has a total of at most $m_i + 2(k_i - 1)$ red arcs.

If we sum this bound over all red polygons P_i that contain at least one cell we conclude that the number of red arcs in the k given cells is at most $m + 2(k - 1)$. Similarly, the number of blue arcs is at most $n + 2(k - 1)$. We have therefore proved the following claim:

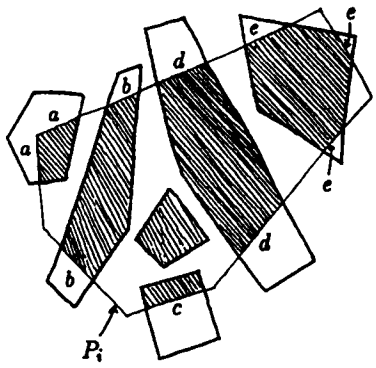

Figure 36.

Theorem 18. *The total size of any k of the cells formed by intersecting two collections of disjoint convex polygons of total size m and n is at most $m + n + 4(k - 1)$.*

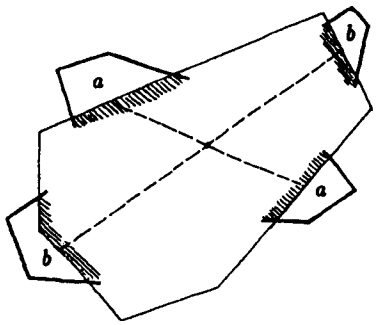

Figure 37.

C3. Amortized analysis

We already encountered the concept of amortized complexity in section B2, in the analysis of finger search trees. The techniques of amortized analysis [71] have also found a number of interesting uses in the analysis of geometric algorithms. Perhaps the earliest significant algorithm in the field, Graham's method of computing the convex hull of n points in the plane [26] already offers an example of amortized analysis. In that method

one first sorts the n points polarly around some point that is interior to the convex hull. Following that, one performs what has come to be called the *Graham scan*. This is a pass over the points in polar order where we maintain a convex chain as we go along; this chain is our best guess as to what the convex hull is, based on points we have seen so far. If the next point considered destroys the convexity of the chain, then one has to back up through the previous points in the chain and discard points till convexity is restored. This is most conveniently done by keeping the current convex chain as a collection of points in a stack. If the new point fits the chain, then it is just stacked as well. If, on the other hand, the new point destroys the convexity, then points are successively popped from the stack till convexity is restored between the stack contents and the current point. That point is then pushed onto the stack and the process repeats. Since each point is stacked and unstacked at most once, the whole process takes time linear in n.

Notice that the processing time for each point during the Graham scan can be highly variable. Some points may use $O(1)$ time, and others may take $\Omega(n)$ time. Still, the charging scheme of putting the cost of the operations on the points themselves as they are stacked or unstacked and not on the points that cause the stacking or unstacking is an example of amortized complexity analysis. Such arguments frequently involve clever charging schemes, such as the use of potential functions. Another example where amortized complexity comes in is the analysis of the topological plane sweep method for an arrangement of n lines by Edelsbrunner and Guibas [21].

C4. Average-case analysis

The field of average-case analysis of geometric algorithms is less well developed than its counterpart in combinatorial algorithms. One difficulty is that most geometric problems have infinite input domains, for which there are no natural probability distributions. For example, what does it mean to choose n random points on the plane, or a random simple polygon with n vertices? Moreover, any given input distribution is unlikely to be represetative of data to be encountered in practice: the simple polygons typically encountered in cartography applications are statistically quite different from those typical of engineering design. Finally, the non-linearity of most geometric operations causes internal results to have extremely complicated distributions. Just to mention a simple example, suppose we take two points p and q with random coordinates, independently drawn from a Gaussian probability distribution. The coefficients of the line joining p and q will be second-degree rational functions of

all four coordinates; as such, they will *not* be independent of each other, and they will *not* follow a Gaussian distribution, or any of the distributions normally studied in statistics books.

The study of this kind of problems is known as *integral* or *stochastic geometry*. For an introduction to this field, see the work of Santaló [60,61] and Miles [50,51,52,53]. We expect that techniques from this field will find more applications in the analysis of geometric algorithms in the future.

Random convex hull. As an example of average-case analysis, we will derive an asymptotic expression for $\bar{h}(n)$, the expected number of vertices of the convex hull of n random points in the plane. The importance of this quantity stems from the fact that many geometric algorithms start by computing the convex hull of n given points, and often the cost of subsequent steps is determined by the size h of the hull, rather than the total number of input points.

The value of $\bar{h}(n)$ obviously depends on the probability distribution of the given points, as well as on their number. In this section we will consider a very simple case: we will assume the n given points are uniformly and independently distributed inside a triangular region of the plane. Following the approach used in 1963 by Rényi and Sulanke [59], we will prove the following result:

Theorem 19. *The average number of vertices in the convex hull of n random points uniformly and independently distributed in a triangle is*

$$\begin{aligned}\bar{h}(n) &= 2H_{n-1} \\ &= 2(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n-1}) \\ &= 2(\ln n+\gamma)+o(1) \qquad (1)\end{aligned}$$

Proof: First, observe that affine maps preserve convexity, "inside-outside" relations, and uniform distributions. Moreover, for any two triangles there is an affine map that takes one into the other. We conclude that if the theorem holds for the equilateral triangle T with unit side, it will automatically hold for any other triangle.

Let $P=\{p_1,p_2,\ldots,p_n\}$ be the input points. If we define the random variables

$$\varepsilon_{ij}=\begin{cases}1 & \text{if } p_ip_j \text{ is an edge of hull}(P),\\ 0 & \text{otherwise.}\end{cases} \qquad (2)$$

then the number of edges (vertices) of hull(P) will be

$$h = \sum_{1 \le i < j \le n} \varepsilon_{ij} \tag{3}$$

and therefore

$$\bar{h}(n) = \sum_{1 \le i < j \le n} \Pr(\varepsilon_{ij}=1) \tag{4}$$

Since all points are drawn from the same distribution, the probabilities $Pr(\varepsilon_{ij}=1)$ are all equal, and it suffices to compute one of them, say $\Pr(\varepsilon_{12} = 1)$. This number is the probability that points $p_3, p_4, \ldots p_n$ fall on the same side of the line p_1p_2.

With probability one, the line p_1p_2 divides the triangle T into a smaller triangle T_{12} and a convex quadrilateral Q_{12}. See figure 38. Let u and v be the lengths of the two sides of T_{12} that are not on the line p_1p_2. The areas of T and T_{12} are then

$$\begin{aligned} |T| &= \frac{1}{2} \sin \frac{\pi}{3} = \frac{\sqrt{3}}{4} \\ |T_{12}| &= \frac{1}{2} uv \sin \frac{\pi}{3} = uv \frac{\sqrt{3}}{4} \end{aligned} \tag{5}$$

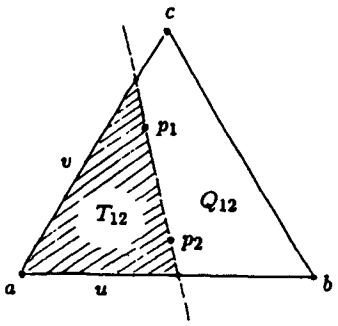

Figure 38.

Therefore, $|T_{12}| / |T| = uv$, and $|Q_{12}| / |T| = 1 - uv$. The probability $\Pr(\varepsilon_{12}=1)$ of all remaining points falling on one side of p_1p_2 is then

$$\int_T \int_T \left((uv)^{n-2} + (1-uv)^{n-2}\right) \Pr(p_1)\, dp_1\, \Pr(p_2)\, dp_2 \tag{6}$$

where $\Pr(p)$ is the probability density of the input points at p, namely $1/|T| = 4/\sqrt{3}$. Note that each $\int$ here denotes a *two-dimensional* integral, and that u and v depend on p_1 and p_2. Formula (6) simplifies to

$$\Pr(\varepsilon_{12}=1) = \frac{16}{3} \int_T \int_T \left((uv)^{n-2} + (1-uv)^{n-2}\right)\, dp_1\, dp_2 \tag{7}$$

Let a, b, and c be the vertices of T. We can partition the domain of integration of (7), namely the set $T \times T$ of all pairs (p_1, p_2) in the triangle, into three subdomains A, B, C according to which vertex of T is also a vertex of T_{12}. Because of symmetry, the integral (7) will be the same on each subdomain, and we can write

$$\Pr(\varepsilon_{12}=1) = 16 \int_A \int \left((uv)^{n-2} + (1-uv)^{n-2}\right)\, dp_1\, dp_2 \tag{8}$$

To compute (8) we will perform a change of variables, replacing p_1 and p_2 by the distances u and v. This requires that

we rewrite the differential $dp_1\,dp_2$ in terms of u, v, du and dv. In other words, we must compute the (4-dimensional) measure of the set of point pairs p, q in $T \times T$ such that the line pq cuts the side ab at u' between u and $u+du$, and cuts ac at v' between v and $v+dv$. Instead of computing this directly, we will measure the set $A'(u,v)$ of pairs p,q for which $u' < u$ and $v' < v$. See figure 39. We then obtain the desired answer by diferentiating the measure $|A'(u,v)|$ of this set with respect to u and v.

What is the set $A'(u,v)$? In order for $u' < u$ and $v' < v$, the points p, q must be inside T_{12}. Furthermore for each choice of p, the point q must be in regions I or IV of figure 40. The measure $|A'(u,v)|$ is therefore the area of T_{12} times the average area of these two regions when p_1 ranges over the whole T_{12}. If T_{12} were an equilateral triangle, then from symmetry alone it would follow that the average area of each region I–VI is exactly one sixth of the area of T_{12}. However, since we can make T_{12} equilateral by an affine map, and every affine map preserves the ratio of areas, we conclude that the same is true no matter what is the shape of T_{12}. Therefore, we have

$$|A'(u,v)| \;=\; |T_{12}| \cdot \frac{2}{6}\,|T_{12}| \;=\; \frac{u^2v^2}{16} \qquad (9)$$

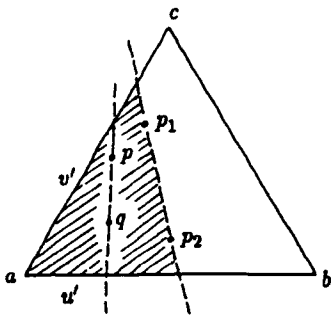

Figure 39.

Differentiation of the formula above with respect to u and v gives $dp_1\,dp_2 = \frac{1}{4}uv\,du\,dv$. Substituting this into (8) we get

$$\begin{aligned}\Pr(\varepsilon_{12}{=}1) &= 16\int_0^1\int_0^1 \left((uv)^{n-2} + (1-uv)^{n-2}\right)\frac{uv}{4}\,du\,dv \\ &= 4\int_0^1\int_0^1 \left((uv)^{n-1} + (1-uv)^{n-2}uv\right)du\,dv \qquad (10)\end{aligned}$$

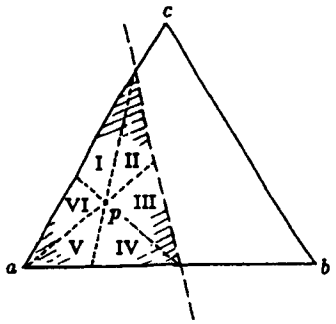

Figure 40.

The evaluation of (10) presents no great problems. For the term $(uv)^{n-1}$ we have

$$\int_0^1\int_0^1 (uv)^{n-1}\,du\,dv = \left(\int_0^1 u^{n-1}du\right)\left(\int_0^1 v^{n-1}dv\right) = \frac{1}{n^2} \qquad (11)$$

As for the second term $(1-uv)^{n-2}uv$, we first integrate it with respect to u by substituting z for $1-uv$:

$$\int_0^1 (1-uv)^{n-2}\,uv\,du$$

$$= -\frac{1}{v}\int_1^{1-v} z^{n-2}(1-z)\,dz$$

$$= -\frac{1}{v}\left[\frac{z^{n-1}}{n-1} - \frac{z^n}{n}\right]_1^{1-v}$$

$$= \frac{1-(1-v)^{n-1}}{n(n-1)v} - \frac{1-(1-v)^{n-1}}{n} \qquad (12)$$

Now we integrate expression (12) with respect to v, substituting w for $(1-v)$:

$$\int_0^1\int_0^1 (1-uv)^{n-2}uv\,du\,dv$$

$$= \int_0^1 \left(\frac{1-(1-v)^{n-1}}{n(n-1)v} - \frac{1-(1-v)^{n-1}}{n}\right) dv$$

$$= -\int_1^0 \left(\frac{1-w^{n-1}}{n(n-1)(1-w)} - \frac{w^{n-1}}{n}\right) dw$$

$$= \int_0^1 \left(\frac{1+w+w^2+\cdots+w^{n-2}}{n(n-1)} - \frac{w^{n-1}}{n}\right) dw$$

$$= \frac{1}{n(n-1)}\left(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n-1}\right) - \frac{1}{n^2}$$

$$= \frac{H_{n-1}}{n(n-1)} - \frac{1}{n^2} \qquad (13)$$

From (11) and (13) we get

$$\Pr(\varepsilon_{12}=1) = \sum_{1\le i<j\le n} \Pr(\varepsilon_{ij}=1)$$

$$= 4\left(\frac{1}{n^2} + \frac{H_{n-1}}{n(n-1)} - \frac{1}{n^2}\right)$$

$$= \frac{4H_{n-1}}{n(n-1)} \qquad (14)$$

We conclude that

$$\bar{h}(n) = \binom{n}{2}\frac{4H_{n-1}}{n(n-1)} = 2H_{n-1}$$

∎

Other distributions. Rényi and Sulanke were also able to compute the asymptotic expansion of $\bar{h}(n)$ for a few other distributions. If the points are uniformly distributed over a fixed convex polygon K with vertices $a_1, a_2, \ldots, a_r$, then we have

$$\bar{h}(n) = \frac{2r}{3}(\ln n + \gamma) + \frac{2}{3}\left(\sum_{1 \le i \le r} \ln \frac{|T_i|}{|K|}\right) + o(1) \qquad (15)$$

where T_i is the triangle $a_{i-1}a_ia_{i+1}$.

This result may seem paradoxical: let K be a triangle, and suppose we remove an infinitesimally small piece K' from one of its corners. According to formulas (1) and (15), the average number of vertices in the convex hull jumps from $2\ln n + o(1)$ to $\frac{8}{3}\ln n + o(1)$. What happens is that for small n the $o(1)$ terms are still large enough to obliterate the difference between $2\ln n$ and $\frac{8}{3}\ln n$. When n is large enough to make the $o(1)$ terms negligible, the probability of finding a point in K' has become significant — i.e., K' is no longer infinitesimally small.

If the points are uniformly distributed on a circle, the average value of h is

$$\bar{h}(n) = 2\,\Gamma(5/3)\left(\frac{2\pi^2}{3}\right)^{1/3} n^{1/3}\,(1 + o(1)) \qquad (16)$$

In fact, we have $\bar{h}(n) = O(n^{1/3})$ whenever the points are uniformly distributed over an *arbitrary* convex figure with smooth boundary; the shape of the curve changes only the proportionality factor. Rényi and Sulanke also show that

$$\bar{h}(n) = 2\sqrt{2\pi \ln n}\,(1 + o(n)) \qquad (17)$$

if the points come from a two-dimensional normal distribution.

Epilogue

In this paper we have attempted to survey a number of different paradigms for the design and analysis of geometric algorithms. We have illustrated the use of traditional algorithm design methods and data structuring techniques, as well as the use of a number of tools that are more intrinsically geometric. Many algorithms employ several of these tools in combination. We hope that in time more and more techniques from geometry (as a branch of mathematics) will find use in this computational domain, thus linking it ever more tightly with a discipline that is as old as human thought itself.

Acknowledgements We gratefully acknowledge the help of Jack Snoeyink in the space sweep section, John Hershberger in the finger tree section, and Micha Sharir in the Davenport-Schinzel sequences section. We thank also Cynthia Hibbard and Lyle Ramshaw for valuable advice and for their critical reading of the manuscript.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman: *The design and analysis of computer algorithms.* [Book] Addison-Wesley, Reading Mass. (1974).

[2] Te. Asano and Ta. Asano: *Minimum partition of polygonal regions into trapezoids.* Proc. of the 24th Annual IEEE Symp. on Foundations of Computer Science (1983), 233–241.

[3] A. Baltsan and M. Sharir: *On shortest paths between two convex polyhedra.* Technical Report no. 180 (Robotics report no. 52), Courant Inst. of Mathematical Sciences, New York Univ. (September, 1985).

[4] J. L. Bentley: *Algorithms for Klee's Rectangle problems.* [Manuscript] CS Department, Carnegie-Mellon Univ. (1977).

[5] J. L. Bentley and Th. Ottman: *Algorithms for reporting and counting geometric intersections.* IEEE Trans. on Computers vol. C-28 no. 9 (Sep. 1979), 643–647.

[6] M. Blum, R. W. Floyd, V. Pratt, R. Rivest, and R. E. Tarjan: *Time bounds for selection.* J. of Computer and Systems Science vol. 7 (1973) 448-461.

[7] K. Q. Brown: *Geometric transforms for fast geometric algorithms.* [Ph. D. Thesis.] Technical report CMU-CS-80-101, Computer Science Dept., Carnegie-Mellon Univ. (1980).

[8] B. Chazelle: *A theorem on polygon cutting with applications.* Proc. 23rd Annual IEEE Symp. on Foundations of Computer Science (Nov. 1982), 339–349.

[9] B. Chazelle and D. Dobkin: *Detection is easier than computation.* Proc. 12th Annual ACM Symp. on Theory of Computing (Apr. 1980), 146–152.

[10] B. Chazelle and L. J. Guibas: *Fractional cascading.* Algorithmica vol. 1 no. 2 (1986), 133–191.

[11] B. Chazelle and L. J. Guibas: *Visibility and intersection problems in plane geometry.* Proc. 1st ACM Symp. on Computational Geometry (July 1985), 135–146.

[12] B. Chazelle, L. J. Guibas, and D. T. Lee: *The power of geometric duality.* BIT vol. 25 (1985), 76–90.

[13] K. L. Clarkson: *New applications of random sampling in computational geometry.* Discrete Computational Geometry vol. 2 (1987), 195–222.

[14] K. L. Clarkson: *Applications of random sampling in computational geometry, II.* Proc. 4th ACM Symp. on Computational Geometry (1988), 1–11.

[15] H. S. M. Coxeter: *The real projective plane.* [Book] McGraw-Hill (1949).

[16] D. P. Dobkin and D. G. Kirkpatrick: *Fast detection of polyhedral intersections.* [Manuscript] EECS Department, Princeton Univ. (November 1981).

[17] D. P. Dobkin and D. G. Kirkpatrick: *A linear algorithm for determining the separation of convex polyhedra.* J. of Algorithms vol. 6 (1985), 381–392.

[18] D. P. Dobkin and D. L. Souvaine: *Computational geometry — A user's guide.* Technical report 334, EE/CS Department, Princeton University (February 1, 1985).

[19] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan: *Making data structures persistent.* Proc. of the 18th Annual ACM Symp. on Theory of Computing (May 1986), 109–121.

[20] H. Edelsbrunner: *Key-problems and key-methods in computational geometry.* [Manuscript] Institut für Informationsverarbeitung, Graz Technical Univ., Austria.

[21] H. Edelsbrunner and L. J. Guibas: *Topologically sweeping an arrangement.* Proc. of the 18th Annual ACM Symp. on Theory of Computing (May 1986), 389–403.

[22] H. Edelsbrunner, J. O'Rourke, and R. Seidel: *Constructing arrangements of lines and hyperplanes with applications.* Proc. of the 24th Annual IEEE Symp. on Foundations of Computer Science (1983), 83–91.

[23] A. R. Forrest: *Computational Geometry* Proceedings of the Royal Society of London vol. 321 series A (1971), 187–195.

[24] S. Fortune: *A sweepline algorithm for Voronoi diagrams.* Proc. 2nd ACM Symp. on Computational Geometry (June 1986), 313–322.

[25] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan: *Triangulating a simple polygon.* Information Processing Letters, vol. 7 no. 4 (Jun. 1978), 175–179.

[26] R. L. Graham: *An efficient algorithm for determining the convex hull of a finite planar set.* Information Processing Letters vol. 1 (1972), 132–133.

[27] P. J. Green and R. Sibson: *Computing Dirichlet tesselations in the plane.* Comp. J. vol. 21 no. 2 (1977), 168–173.

[28] B. Grünbaum: *Convex polytopes.* [Book] John Wiley & Sons, London (1967).

[29] L. J. Guibas and J. Hershberger: *Optimal shortest path queries in a simple polygon.* [Manuscript] To appear in Proc. of 19th Annual ACM Symp. on Theory of Computation (July 1987).

[30] L. J. Guibas, E. M. McCreight, M. F. Plass, J. R. Roberts: *A new representation for linear lists.* Proc. 9th ACM Symp. on Theory of Computing (May 1977), 49–60.

[31] L. J. Guibas, L. Ramshaw, and J. Stolfi: *A kinetic framework for computational geometry.* Proc. of the 24th IEEE Annual Symp. on Foundations of Computer Science (November 1983), 100–111.

[32] L. J. Guibas and R. Seidel: *Computing convolutions by reciprocal search.* Proc. 2nd ACM Symp. on Computational Geometry (June 1986), 90–99.

[33] L. J. Guibas and J. Stolfi: *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams.* ACM Trans. on Graphics vol. 4 no. 2 (April 1985), 74–123.

[34] S. Hart and M. Sharir: *Nonlinearity of Davenport-Schnitzel sequences and of generalized path-compression schemes.* Technical report 84-011, Inst. of Computer Sciences, Tel-Aviv Univ. (August 1984).

[35] S. Hertel, K. Mehlhorn, M. J. Mäntylä, and J. Nievergelt: *Space sweep solves intersection of polyhedra elegantly.* Report HTKK-TKO-B55, Laboratory of Information Processing, Helsinki Univ. of Technology (September 6, 1983).

[36] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan: *Sorting Jordan sequences in linear time using level-linked search trees.* Information and Control vol. 68 (1986), 170–184.

[37] N. Karmarkar: *A new polynomial-time algorithm for linear programming.* Proc. of the 16th Annual ACM Symp. on Theory of Computing (1984), 302–311.

[38] D. G. Kirkpatrick: *Optimal search in planar subdivisions.* SIAM J. on Computing vol. 12 no. 1 (Feb. 1983), 28–35.

[39] D. G. Kirkpatrick and R. Seidel: *The ultimate planar convex hull algorithm.* Technical report 83-577, CS Department, Cornell Univ. (October 1983).

[40] C. L. Lawson: *Generation of a triangular grid with applications to contour plotting.* Technical Memo 299, Jet Propulsion Laboratory, Pasadena (Feb. 1972).

[41] D. T. Lee: *Proximity and Reachability in the Plane* (Ph. D. Thesis). Technical report R-831, Coordinated Science Lab., Univ. of Illinois, Urbana (Nov. 1978).

[42] D. T. Lee and F. P. Preparata: *Computational geometry — A survey.* IEEE Trans. on Computers vol. C-33 no. 12 (1984), 1072-1101.

[43] D. T. Lee and B. J. Schachter: *Two Algorithms for constructing the Delaunay Triangulation.* International J. of Computer and Information Sciences, Vol 9 no. 3 (1980), 219–242.

[44] H. G. Mairson and J. Stolfi: *Reporting and counting line segment intersections (extended abstract).* [Manuscript]. DEC Systems Research Center (1984). Submitted to the 1987 NATO ASI on Theoretical Foundations of Computer Graphics and CAD (July 1987).

[45] E. M. McCreight: *Efficient algorithms for enumerating intersecting intervals and rectangles.* Technical report CSL-80-9, Xerox Palo Alto Research Centers (1980).

[46] E. M. McCreight: *Priority search trees.* Technical report CSL-81-5. Xerox Palo Alto Researc Centers (1981).

[47] N. Megiddo: *Linear-time algorithms for linear programming in R^3 and related problems.* Proc. 23rd Annual IEEE Symp. on Foundations of Computer Science (Nov. 1982), 329–338.

[48] N. Megiddo: *Solving linear programming in linear-time when the dimension is fixed.* [Manuscript] Statistics Department, Tel Aviv Univ. (Israel), April 1982.

[49] K. Mehlhorn: *Data structures and algorithms.* [Book] Springer-Verlag, Berlin (1984).

[50] R. E. Miles: *Random polygons determined by random lines in a plane.* Proc. National Academy of Sciences vol. 52 (1964), 901–907 and 1157–1160.

[51] R. E. Miles: *A wide class of distributions in geometric probability.* Annals of Mathematical Statistics vol. 35 (1964), 1407–1415.

[52] R. E. Miles: *Solution to problem 67-15 (Probability distribution of a network of triangles).* SIAM Review vol. 11 (1969) 399–402.

[53] R. E. Miles: *On the homogeneous planar Poisson point-process.* Math. Biosciences vol 6. (1970), 85–127.

[54] J. Nievergelt and F. P. Preparata: *Plane-sweep algorithms for intersecting geometric figures.* Comm. ACM vol. 25 (1982), 739–747.

[55] M. H. Overmars: *The design of dynamic data structures.* [Ph. D. thesis] Univ. of Utrecht (21 March 1983).

[56] M. J. Post: *A Minimum spanning ellipse algorithm.* Proc. 22nd Annual IEEE Symp. on Foundations of Computer Science (Oct. 1981), 115–122.

[57] F. P. Preparata and S. J. Hong: *Convex hulls of finite sets of points in two and three dimensions.* Comm. ACM vol. 20 no. 2 (Feb. 1977), 87–93.

[58] F. P. Preparata and M. I. Shamos: *Computational geometry: An introduction* [Book] Springer Verlag, New York (1985).

[59] A. Rényi and R. Sulanke: *Über die konvexe Hülle von n zufällig gewählten Punkten.* Z. Wahrscheinlichkeitstheorie vol. 2 (1963), 75–84.

[60] L. A. Santaló: *Introduction to integral geometry.* [Book] Hermann, Paris (1953).

[61] L. A. Santaló: *Integral geometry and geometric probability.* [Book]. Addison-Wesley (1976).

[62] N. Sarnak and R. E. Tarjan: *Planar point location using persistent search trees* (extended abstract). Communications of the ACM vol. 29 no. 7 (July 1986), 669–.

[63] J. T. Schwartz, M. Sharir, J. Hopcroft: *Planning, geometry, and complexity of robot motion.* [Book] Ablex, Norwood NJ (1986).

[64] R. Seidel: *Output-size sensitive algorithms for constructive problems in computational geometry.* [Ph. D. Thesis.] Technical report CS-TR-86-784, Cornell University (September 1986).

[65] M. I. Shamos: *Computational geometry.* [Ph. D. thesis] Yale Univ. (Dec. 1977).

[66] M. I. Shamos and D. Hoey: *Geometric intersection problems.* Proc. 17th IEEE Symp. on Foundations of Computer Science (October 1976), 208–215.

[67] M. Sharir: *Almost linear upper bounds on the length of general Davenport-Schinzel sequences.* Technical report 29/85, Institute of Computer Sciences, Tel Aviv University (February 1985).

[68] M. Sharir: *On the two-dimensional Davenport Schinzel problem.* Technical report 193, Courant Inst. of Mathematical Sciences, New York Univ. (December 1985).

[69] M. Sharir: *Improved lower bounds on the length of Davenport-Schinzel sequences.* Technical report 204, Courant Inst. of Mathematical Sciences, New York Univ. (February 1986).

[70] M. Sharir and R. Livne: *On minima of functions, intersection pattern of curves, and Davenport-Schinzel sequences.* Proc. 26th IEEE Symp. on Foundations of Computer Science (1985), 312–330.

[71] R. E. Tarjan: *Amortized computational complexity.* SIAM J. of Algorithms and Discrete Methods vol. 6 no. 2 (April 1985), 306–318.

[72] R. E. Tarjan and C. J. Van Wyk: *An $O(n \log\log n)$-time algorithm for triangularizing simple polygons.* [Manuscript]. CS Department, Princeton Univ. (July 1986).

[73] A. C. Yao: *A lower bound to finding convex hulls.* J. of the ACM vol. 28 (1981), 780–787.

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Research Report 1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Research Report 2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Research Report 3, January 11, 1985.

"Eliminating **go to**'s while Preserving Program Structure."
Lyle Ramshaw.
Research Report 4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Research Report 5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Research Report 6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Research Report 7, November 14, 1985.

"On Interprocess Communication."
Leslie Lamport.
Research Report 8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Research Report 9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Research Report 10, May 1, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Research Report 11, May 5, 1986.

"Fractional Cascading."
Bernard Chazelle and Leonidas J. Guibas.
Research Report 12, June 23, 1986.

"Retiming Synchronous Circuitry."
Charles E. Leiserson and James B. Saxe.
Research Report 13, August 20, 1986.

"An $O(n^2)$ Shortest Path Algorithm for a Non-Rotating Convex Body."
John Hershberger and Leonidas J. Guibas.
Research Report 14, November 27, 1986.

"A Simple Approach to Specifying Concurrent Systems."
Leslie Lamport.
Research Report 15, December 25, 1986. Revised January 26, 1988

"A Generalization of Dijkstra's Calculus."
Greg Nelson.
Research Report 16, April 2, 1987.

"*win* and *sin*: Predicate Transformers for Concurrency."
Leslie Lamport.
Research Report 17, May 1, 1987.

"Synchronising Time Servers."
Leslie Lamport.
Research Report 18, June 1, 1987.

"Blossoming: A Connect-the-Dots Approach to Splines."
Lyle Ramshaw.
Research Report 19, June 21, 1987.

"Synchronization Primitives for a Multiprocessor: A Formal Specification."
A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin.
Research Report 20, August 20, 1987.

"Evolving the UNIX System Interface to Support Multithreaded Programs."
Paul R. McJones and Garret F. Swart.
Research Report 21, September 28, 1987.

"Building User Interfaces by Direct Manipulation."
Luca Cardelli.
Research Report 22, October 2, 1987.

"Firefly: A Multiprocessor Workstation."
C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr.
Research Report 23, December 30, 1987.

"A Simple and Efficient Implementation for Small Databases."
Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber.
Research Report 24, January 30, 1988.

"Real-time Concurrent Collection on Stock Multiprocessors."
John R. Ellis, Kai Li, and Andrew W. Appel.
Research Report 25, February 14, 1988.

"Parallel Compilation on a Tightly Coupled Multiprocessor."
Mark Thierry Vandevoorde.
Research Report 26, March 1, 1988.

"Concurrent Reading and Writing of Clocks."
Leslie Lamport.
Research Report 27, April 1, 1988.

"A Theorem on Atomicity in Distributed Algorithms."
Leslie Lamport.
Research Report 28, May 1, 1988.

"The Existence of Refinement Mappings."
Martín Abadi and Leslie Lamport.
Research Report 29, August 14, 1988.

"The Power of Temporal Proofs."
Martín Abadi.
Research Report 30, August 15, 1988.

"Modula-3 Report."
Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson.
Research Report 31, August 25, 1988.

"Bounds on the Cover Time."
Andrei Broder and Anna Karlin.
Research Report 32, October 15, 1988.

"A Two-view Document Editor with User-definable Document Structure."
Kenneth Brooks.
Research Report 33, November 1, 1988.

"Blossoms are Polar Forms."
Lyle Ramshaw.
Research Report 34, January 2, 1989.

"An Introduction to Programming with Threads."
Andrew Birrell.
Research Report 35, January 6, 1989.

"Primitives for Computational Geometry."
Jorge Stolfi.
Research Report 36, January 27, 1989.

The Design and Analysis of Geometric Algorithms

by Leonidas J. Guibas and Jorge Stolfi

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301