Trends and Developments in Computational Geometry

M. de Berg

UU-CS-1995-21
May 1995

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Trends and Developments in Computational Geometry

M. de Berg

Technical Report UU-CS-1995-21
May 1995

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Trends and Developments in Computational Geometry*

Mark de Berg

Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, the Netherlands.

Abstract

This report discusses some trends and achievements in computational geometry during the past five years, with emphasis on problems related to computer graphics. Furthermore, a direction of research in computational geometry is discussed, which could help in bringing the fields of computational geometry and computer graphics closer together.

1 Introduction

Computational geometry is the part of theoretical computer science that studies problems involving points, lines, polygons, and other geometric objects. Typical examples are the line segment intersection problem (compute all intersections in a set of line segments in the plane) and the convex hull problem (compute the convex hull of a set of points in two- or higher-dimensional space). Computational geometry developed in the late seventies as an off-spring of algorithms research. As such, it tries to determine the algorithmic complexity of geometric problems: the asymptotic running time of algorithms is usually analyzed and expressed using O-notation, lower bounds are proved, the combinatorial complexity of the problems is investigated, and so on. It is one of the more popular areas of algorithms research: every major conference and journal on theoretical computer science lists computational geometry as one of its topics, and there are several conferences and journals devoted entirely to computational geometry. This has lead to a rapid growth of amount of literature. Currently the on-line database of papers on computational geometry contains around six thousand entries, of which almost two thousand were published during the past five years. Furthermore, a number of new textbooks has appeared. Besides the well known book by Preparata and Shamos [122] and the more mathematically oriented book by Edelsbrunner, there are now textbooks by O'Rourke [113] and Mulmuley [108], and more books on computational geometry or closely related topics will appear in the near future [6, 24, 32, 116].

The popularity of computational geometry has two reasons. The first reason, and perhaps the most important one, is the beauty of the field: because of the often simple and intuitive formulations of the problems, geometry has been appealing to mathematically oriented people since the times of the ancient Greeks. The second reason is the wide applicability of computational geometry: the world around us is a three-dimensional space filled with various objects, so it is not surprising that geometric computations are fundamental in many areas. Robotics, geographic information systems, and CAD/CAM are obvious examples. Computer graphics is another area where geometric problems clearly play an important role: hidden surface removal, ray tracing, and form factor computations for radiosity are examples of topics in computer graphics where geometric algorithms and data structures are required. Even seemingly non-geometric problems can often be interpreted geometrically. Database queries are a well-known example: items that have numerical information associated with them can be viewed as points in some (higher-dimensional) space.

Initially most problems studied in computational geometry were two-dimensional. By the mid-eighties most planar problems were well understood, and the attention shifted to three- and

*This work was supported by the Dutch Organisation for Scientific Research (N.W.O.) and by ESPRIT Basic Research Action No. 7141 (project ALCOM II: *Algorithms and Complexity*)

higher-dimensional problems. These problems turned out to be much more difficult, and their solutions required several new techniques. During the past five years many of these techniques have been developed. Another recent trend is towards experimental research and more practical algorithms. This report discusses a number of the new techniques and results, with special attention to problems relevant to computer graphics. I have not tried to be complete. Several topics are left uncovered; examples are combinatorial geometry, computational topology, and applications of computational geometry in other areas than graphics. I have also tried to do more (and less) than just list a large number of results. Rather than that I have described some of the results and techniques in more detail, while omitting others, or only mentioning them briefly. This way the reader will hopefully get a better feeling for the field. The following topics will be discussed.

We start with randomized algorithms. This type of algorithms has become very popular in computational geometry during the last few years. One of the design paradigms for randomized algorithms, randomized incremental construction, will be explained with case studies for two basic geometric problems: linear programming and the computation of Voronoi diagrams. We shall also discuss another important general technique, namely random sampling.

The shift from two- to higher-dimensional problems in computational geometry can be clearly seen in the area of geometric data structures: in recent years two important tools, simplicial partitions and cuttings, have been developed that form the basis of many higher-dimensional search structures. Together with the concept of multi-level structures this yields a powerful technique for the design of higher-dimensional data structures. This will be discussed in Section 3.

Decomposing a space or a geometric object into smaller and simpler pieces is important in various applications. In Section 4 we will look at some decomposition techniques that have been studied in computational geometry. In particular, we study BSP trees, meshing, and polygon triangulation.

Some other problems that arise in computer graphics—hidden surface removal, ray shooting, and discrepancy—are discussed in Section 6.

Robustness is a major problem in the implementation of geometric algorithms. Round-off errors may cause programs to produce the wrong output or to crash, and often there are many special (degenerate) cases that must be dealt with. Section 5 discusses some of the research that has been done to handle these problems in a general way.

Section 7 discusses a direction for future research and some current initiatives to increase the practical applicability of computational geometry. Although this section constitutes only a small part of the report, it is at least as important as the rest. It should convince people from areas like computer graphics that computational geometry is giving more and more attention to practical issues, and it should encourage people from computational geometry to increase their application oriented research. Thus it will hopefully help in bringing computational geometry and its application areas closer together.

2 Randomized algorithms

A major trend in computational geometry—indeed, in algorithms research in general—is the use of randomized algorithms. A randomized algorithm is an algorithm that at certain moments makes random choices. An easy and well known example is randomized quicksort, an algorithm for sorting a set S of n numbers: Pick a random pivot element $x \in S$, partition S into a subset of elements less than x and a subset of elements greater than x, and sort these two subsets recursively. If we have bad luck, then the partitioning is very unbalanced at every step of the algorithm, leading to quadratic time algorithm. But since we pick x at random, we expect it to be close to the median of S. One can prove that the *expected* running time of the algorithm is $O(n \log n)$. The expectancy in the running time has nothing to do with the distribution of the input points, it is solely with respect to the random choices made by the algorithm. In other words, the expected running time is $O(n \log n)$ for every input; there are no 'bad' inputs, only 'bad' random choices made by the algorithm. In a deterministic version of quicksort we have to use a subroutine for finding the median of a set of n numbers in linear time, if we want to have an $O(n \log n)$ algorithm. Such

a routine exists, but it is rather complicated. This example illustrates the power of randomized algorithms: they are often much simpler than their deterministic counterparts, which makes them easier to implement and often faster in practice.

Since the pioneering work of Haussler and Welzl [81], Clarkson and Shor [55], and Mulmuley [104], randomized algorithm have become a major design paradigm in computational geometry. They have become so popular that there is now a textbook on computational geometry that focusses on randomized algorithms [108]. In this section we will not consider very recent developments such as dynamic randomized algorithms [129, 138]. Instead, we will focus on the two basic techniques for the design of randomized algorithms: randomized incremental construction and random sampling. Randomized incremental algorithms are introduced through two important geometric problems: linear programming and the computation of Voronoi diagrams. Random sampling is explained on the basis of cuttings, a concept we will need in Section 3.

Randomized algorithms need to be able to perform certain random choices, such as picking an element from a set at random. We therefore assume that we have a random number generator available, which, given an integer n, can produce a random integer in the range $1 \cdots n$ in constant time. Theoretically this is perhaps not very satisfactory, since random number generators are not truly random. In practice, however, this is a reasonable assumption. (Chapter 10 of Mulmuley's textbook [108] discusses discusses these issues in detail.)

2.1 Linear programming

Linear programming is one of the classical optimization problems. In a linear programming problem one wants to minimize a linear *cost function* $c : \mathbb{R}^d \to \mathbb{R}$ subject to n linear constraints. It can be formulated as follows:

$$\begin{aligned} \text{Minimize} \quad & c_1 x_1 + c_2 x_2 + \cdots + c_d x_d \\ \text{Subject to} \quad & b_{1,1} x_1 + \cdots + b_{1,d} x_d \leqslant b_{1,d+1} \\ & b_{2,1} x_1 + \cdots + b_{2,d} x_d \leqslant b_{2,d+1} \\ & \qquad \vdots \\ & b_{n,1} x_1 + \cdots + b_{n,d} x_d \leqslant b_{n,d+1} \end{aligned}$$

where the c_i and $b_{i,j}$ are real numbers, which form the input to the problem.

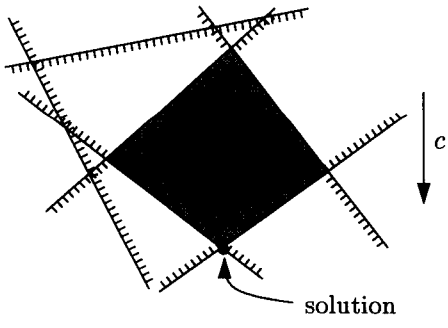

Figure 1: Two-dimensional linear programming.

The constraints in a linear programming problem are half-spaces in d-dimensional space. Hence, the *feasible region* of a linear programming problem—the region where all constraints are satisfied—is the intersection of n half-spaces and thus a convex polytope. This means that we are looking for a point in the polytope that is extreme in a direction specified by the cost function. Fig. 1 illustrates this in the plane. The half-planes are indicated by their bounding lines and a hatched pattern to specify on which side of their bounding line they lie.

In many applications the dimension d of the problem, that is, the number of variables, is quite large; most of the traditional approaches consider this situation. Nevertheless, there are also cases where the dimension is a small constant, say two or three. This setting has been studied extensively in computational geometry. Already in the early 1980s Megiddo [97, 98] and Dyer [65] showed that linear programming can be done in $O(n)$ time when d is a constant. Unfortunately, their algorithms were quite complicated, and the dependency of the running time on d, which is hidden in the O-notation when d is assumed to be a constant, was doubly exponential. Randomization turned out to be a key

factor in improving these results: in recent years, new randomized linear programming algorithms have been developed, which are much simpler than the algorithms of Megiddo and Dyer and have a better dependency on d, while still being linear in n [54, 66, 95, 131, 133]. We shall describe the algorithm by Seidel [131].

A simple randomized algorithm. Consider the two-dimensional linear programming problem. Through a linear transformation we can ensure that the cost function we want to minimize is the function $c : (x_1, x_2) \mapsto x_2$. Thus we want to find a lowest point in the feasible region. Let H be the set of half-spaces representing the constraints of the problem. We add two additional constraints $x_1 \geqslant 0$ and $x_2 \geqslant 0$, so that we know that the solution will be bounded. (This restriction is not essential but simplifies the presentation.) These additional half-planes are denoted h_0 and h_1, respectively. To make the solution unique we shall look for the leftmost lowest point in the feasible region.

The algorithm we describe follows the basic paradigm called *randomized incremental construction*: we treat the half-planes one by one *in random order*, meanwhile maintaining the optimal solution with respect to the current set of half-planes. Let $h_2, \ldots, h_{n+1}$ be a random permutation of the set H of half-planes. (We used h_0 and h_1 for the additional constraints $x_1 \geqslant 0$ and $x_2 \geqslant 0$.) Let v_i denote the solution of the problem after inserting the i-th half-plane, that is, v_i is the point that minimizes the cost function subject to the constraints h_0 up to h_i. Note that v_i is a vertex of the convex polygon defined by $\{h_0, \ldots, h_i\}$. We call v_i the *optimal vertex* of the current polygon. To deal with the next half-plane h_{i+1} we must compute the new optimal vertex v_{i+1}. If $v_i \in h_{i+1}$

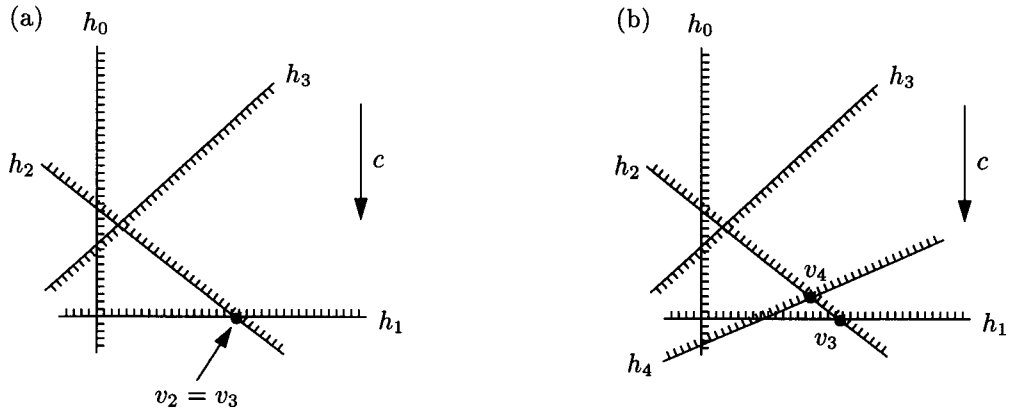

Figure 2: The two cases when adding the next half-plane.

then this is easy: the optimal vertex does not change. This is illustrated in Fig. 2(a); adding h_3 does not change the solution. But it can also happen, as with the addition of h_4 in Fig. 2(b), that h_{i+1} cuts off v_i, in which case we must find a new optimal vertex. This new optimal vertex must lie on the line ℓ_{i+1} that bounds h_{i+1}. Hence, we can find it by considering the intersections of the half-planes $h_0, \ldots, h_i$ with ℓ_{i+1}. More precisely, we can find it by solving a one-dimensional linear program on ℓ_{i+1}, where we wish to find a lowest point on ℓ_i subject to the constraints $h_j \cap \ell_i$, with $0 \leqslant j \leqslant i$. One-dimensional linear programming is quite easy to do in linear time. We now summarize the algorithm in pseudo-code:

2DLINEARPROGRAMMING
Input: A set H of n half-planes in $\mathbb{R}^2$.
Output: The leftmost lowest point in the region $(\bigcap H) \cap \{x_1 \geqslant 0\} \cap \{x_2 \geqslant 0\}$.
1. Let $h_0 := x_1 \geqslant 0$ and let $h_1 := x_2 \geqslant 0$.
2. Let $v_1 := (0, 0)$.
3. Compute a random permutation $h_2, \ldots, h_{n+1}$ of the half-planes in H.
4. **for** $i := 2$ **to** $n + 1$
5. **do if** $v_{i-1} \in h_i$
6. **then** $v_i := v_{i-1}$
7. **else** Solve a one-dimensional linear program on the line bounding h_i with constraints defined by $h_0, \ldots, h_{i-1}$.
8. **if** this one-dimensional linear program has a solution v^*
9. **then** $v_i := v^*$
10. **else** report that the linear program is infeasible and quit.
11. Report v_{n+1} as the solution to the linear program.

What is the running time of this algorithm? Let's start with step 3 of the algorithm: the generation of a random permutation of the set of input constraints. Under the assumption that we can generate random numbers in constant time, this takes only $O(n)$ time. Now consider the amount of work we have to do when we treat a half-plane h_i. If $v_{i-1} \in h_i$ then we only have to do $O(1)$ work. If we have bad luck, however, $v_{i-1} \notin h_i$. In this case we have to solve a one-dimensional linear program with $i + 1$ constraints (this includes the two extra constraints we added), which takes $O(i)$ time. Hence,

$$\text{time to treat } h_i = \begin{cases} O(1) & \text{if } v_{i-1} \in h_i \\ O(i) & \text{if } v_{i-1} \notin h_i \end{cases}$$

Which of these two cases occurs depends on the random order that we generated in step 3: for some random orders we will be lucky, and for other we won't. So let's look at the *expected* time that we spend to treat h_i. Thus we are interested in

$$\mathrm{E}[\text{time to treat } h_i] = \Pr[v_{i-1} \in h_i] \cdot O(1) + \Pr[v_{i-1} \notin h_i] \cdot O(i),$$

where $\mathrm{E}[X]$ denotes the expected value of the random variable X, and $\Pr[Y]$ denotes the probability of the event Y. It is important to realize that the expectation here is with respect to the random permutation generated in step 3; it has nothing to do with the geometric distribution of the constraints.

What is the probability that $v_{i-1} \notin h_i$? Recall that the optimal solution is a vertex of the polygon defined by the constraints. In other words, it is the intersection of the boundary lines of two constraints. When $v_{i-1} \notin h_i$, then the new optimum is computed with a one-dimensional linear program on the line bounding h_i. This implies that the line bounding h_i is one of the two lines defining the new optimal vertex. This observation allows us to apply a trick called *backwards analysis* to bound $\Pr[v_{i-1} \notin h_i]$. Consider the situation *after* h_i has been treated. The optimal vertex is now defined by two half-planes. The case $v_{i-1} \notin h_i$ can only have occurred when h_i was one of these two half-planes. Since the order of the half-planes $h_2, \ldots, h_i$ is random, the probability that h_i is one of the two 'special' half-planes is only $O(1/i)$. Hence, the expected amount of time we spend to treat h_i is $O(1)$, and the total expected time for the linear programming algorithm is $O(n)$.

The generalization of the algorithm to linear programming in $\mathbb{R}^d$ for $d > 2$ is straightforward: the main difference is that we now need to solve a $(d-1)$-dimensional linear program when $v_{i-1} \notin h_i$. (Another, minor difference is that we now need to add d additional constraints, namely $x_j \geqslant 0$ for $1 \leqslant j \leqslant d$, instead of two.) The $(d-1)$-dimensional linear program is solved recursively with the same algorithm. This leads to a running time of $O(d!n)$, which is linear in n for any constant d. Note, however, that the dependency on d makes that the algorithm is efficient only when d is a small constant. There are other randomized algorithm that have a better dependency on d [54, 95].

Smallest enclosing balls. In several applications it is useful to compute a bounding volume for a set of objects. Usually the bounding volume should have a simple shape, contain all the objects, and be as small as possible.

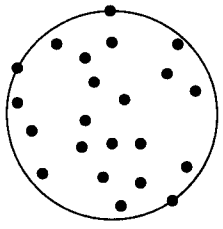

Figure 3: A smallest enclosing ball of a set of points.

A well known example is to compute the smallest disc containing a set of points in the plane, or the smallest ball containing a set of points in three-dimensional space. (Notice that the smallest ball containing a set of polyhedral objects in three-dimensional space can be found by computing the smallest enclosing ball for the vertices of the objects.) This is clearly an optimization problem: the points define the constraints and the function to be optimized is the radius of the enclosing disc or ball. Although the problem is similar to a linear programming problem, it is slightly different. Nevertheless, Welzl [141] showed that the smallest enclosing ball can be computed in linear time with a randomized algorithm. The algorithm is quite simple and experiments show that it performs well in practice. It turns out that there is a whole class of optimization problems that can be solved in linear time with a simple randomized algorithm. Sharir and Welzl [133] presented such an algorithm in an abstract framework. Among the problems to which their framework applies are linear programming, convex programming, and computing smallest enclosing balls or ellipsoids. When the dimension is bounded, all these problems can be solved in linear time. When the dimension is large the algorithm also turns out to be quite efficient, as experiments and the analysis of Matoušek et al. [95] show.

2.2 Voronoi diagrams and Delaunay triangulations

Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in $\mathbb{R}^d$. We call these points *sites*. We denote the Euclidean distance between two points q and r in $\mathbb{R}^d$ by $\text{dist}(q, r)$. The *Voronoi diagram* of S is the subdivision of $\mathbb{R}^d$ into n cells, one per site in S, with the property that a point q lies in the cell corresponding to a site $p_i \in S$ if and only if $\text{dist}(q, p_i) < \text{dist}(q, p_j)$ for each $p_j \in S$ with $j \neq i$. In other words, the cell of a site p_i contains all points for which p_i is the closest site. The edges

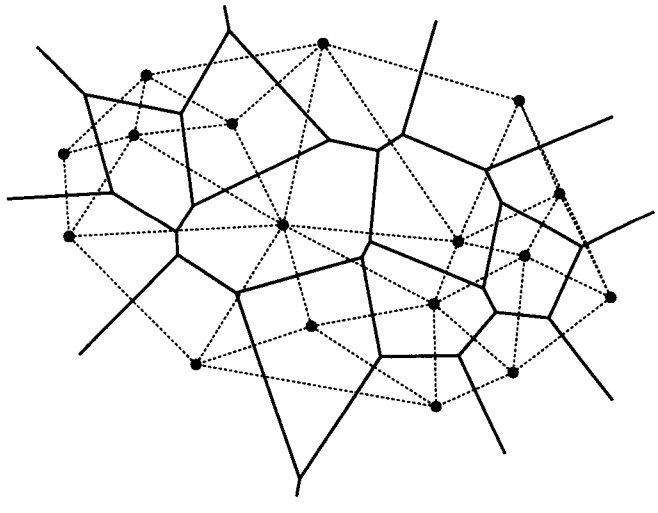

Figure 4: The Voronoi diagram and Delaunay triangulation of a set of points in the plane.

of the Voronoi diagram are formed by the points with two closest sites, and the vertices are the points with three or more closest sites. This is illustrated in Fig. 4, where the the Voronoi diagram of the points is shown solid.

Consider the dual graph of the Voronoi diagram. The nodes in this graph are the sites in S, and two sites are connected by an arc if and only if their cells in the Voronoi diagram are adjacent. In Fig. 4 the dual graph is shown dotted. Surprisingly, this graph is always a triangulation of the convex hull of the set S. (In fact, this is not entirely true: in degenerate situations some of the faces of this graph can be convex k-gons for $k > 3$. But in such cases we can complete the graph into a triangulation by adding extra edges.) This dual graph is called the *Delaunay triangulation* of the set S. It has a number of nice properties. For instance, the circumscribing circle of any triangle in the Delaunay triangulation does not contain any site of S in its interior. The reverse is true as well: any triple of sites whose circumscribing circle has an empty interior forms a triangle in the Delaunay triangulation. (Again, this property should be formulated a little more carefully in degenerate cases.) This implies that the angles of the triangles are not too small. In fact, it can be shown that the Delaunay triangulation maximizes the minimal angle in the triangulation over all possible triangulations of S [67].

Voronoi diagrams and Delaunay triangulations have been studied for a long time; the concept already appeared in Descartes's treatment of cosmic fragmentation in Part III of his *Principia Philosophiae*, published in 1644. Over the years Voronoi diagrams have been used and studied in many different areas, especially in computational geometry. There are many interesting generalizations of Voronoi diagrams: one can consider other metrics than the Euclidean metric, other sites than point sites, and so on. It is beyond the scope of this paper to give an extensive survey of Voronoi diagrams; interested readers should consult the book by Okabe et al. [112] or the survey paper by Aurenhammer [9]. We only consider Voronoi diagrams of sets of points in the plane.

Computing Voronoi diagrams and Delaunay triangulations. The first optimal $O(n \log n)$ time algorithm to compute the Voronoi diagram of a set of n point sites in the plane was a rather complicated divide-and-conquer algorithm by Shamos and Hoey [132]. Since then many other algorithms have been developed. To illustrate the power of randomized algorithms, we describe a simple yet optimal randomized algorithm due to Guibas et al. [78].

The algorithm constructs the Delaunay triangulation of the set S. From this one can easily compute the Voronoi diagram of S in linear time, if needed. As for linear programming, we use the randomized incremental approach. Thus we insert the sites in S in a random order, and we maintain the Delaunay triangulation. The insertion of a site p_i is done as follows. First, we locate the triangle in the current triangulation that contains p_i. We connect p_i to the three vertices of

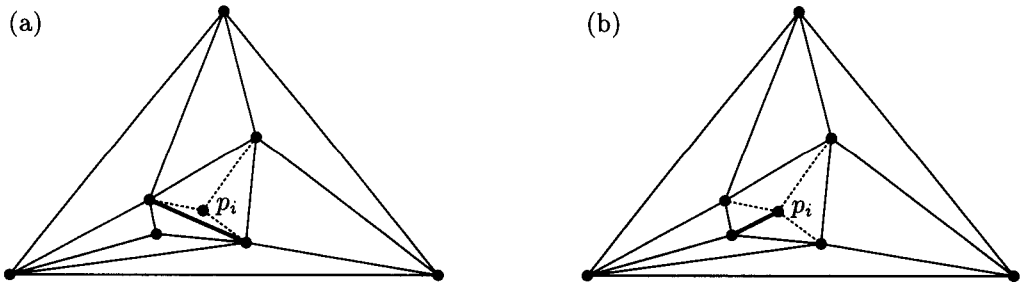

Figure 5: Updating the Delaunay triangulation after the insertion of a point.

this new triangle, as in Fig. 5(a). We now have a valid triangulation of the current point set, but this need not be the Delaunay triangulation. To remedy this situation we use *edge-flipping*: Consider an edge e of the triangulation, and let Δ_1 and Δ_2 be the two adjacent triangles. Suppose that the triangles Δ_1 and Δ_2 together form a convex quadrilateral Q. The edge e is one of the two diagonals of Q. If we replace e by the other diagonal e', then we still have a valid triangulation. We call this operation an edge-flip. It turns out that we can always turn the triangulation we have obtained after the insertion of p_i into the Delaunay triangulation by performing a number of edge-flips. In Fig. 5(a), for instance, we have to flip one edge to get the Delaunay triangulation

of Fig. 5(b); the edge that is being flipped and its replacement are shown in bold. The algorithm can be summarized as follows.

DELAUNAYTRIANGULATION
Input: A set S of n point sites in $\mathbb{R}^2$.
Output: The Delaunay triangulation of S.
1. Let Δ_0 be a large triangle that contains al points in S.
2. Compute a random permutation $p_1, \ldots, p_n$ of the points in S.
3. **for** $i := 1$ **to** n
4. **do** Find the triangle Δ in the current triangulation that contains p_i.
5. Replace Δ by three new triangles by connecting p_i to the vertices of Δ.
6. Turn the new triangulation into a Delaunay triangulation by edge-flipping.

The simple procedure to find out which edge-flips to perform is described in detail by Guibas et al. [78]. The only thing we have swept under the rug so far is step 4 of the algorithm, where we have to locate the triangle that contains p_i. Of course we could simply check all triangles, but this would lead to a quadratic algorithm. (This shows that one still has to be careful when designing randomized algorithms; doing one step the wrong way may destroy the efficiency of the algorithm.) Instead, we use a concept called the *history graph*, or *I-DAG*. This is a directed acyclic graph that records the history of the construction. Initially the graph consists of one node, which represents the triangle Δ_0 that contains all the points in S. Whenever the triangulation changes, we create a new node for each of the new triangles that appear. This way every triangle that was ever created during the course of the algorithm will correspond to a node in the history graph. The triangles in the current triangulation correspond to the leaf nodes in the history graph. The idea is that we can locate the triangle in the current triangulation containing a point p_i by walking down the history graph until we reach the leaf corresponding to this triangle. To understand how this works, consider the insertion of a point p_i. The first thing we did was to replace the triangle Δ containing p_i with three new triangles. This change is reflected in the history graph by creating three new leaf nodes, which get incoming pointers from the node that represented Δ. Next we started the edge-flipping procedure. An edge-flip is reflected in the graph by creating two new leaf nodes corresponding to the two new triangles that are created; these leaf nodes get incoming arcs from the two disappearing triangles. Hence, whenever a triangle disappears it has outgoing arcs to the (three or two) new triangles that intersect it. Fig. 6 shows again the changes made

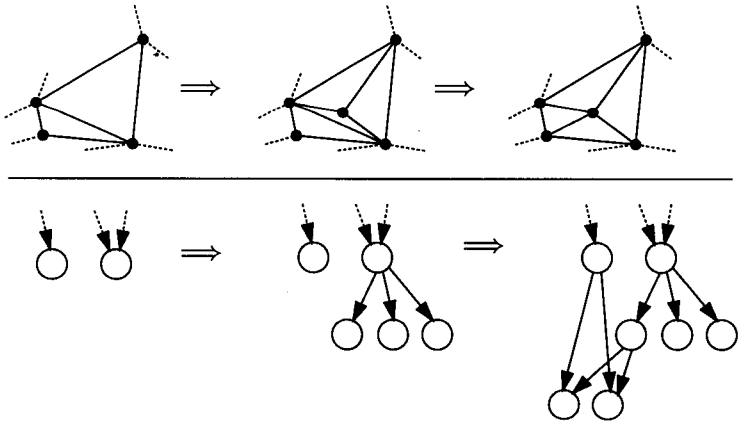

Figure 6: Changes in the history graph due to an insertion.

in Fig. 5, and how these changes are reflected in the history graph. This means that we can locate the triangle containing a point p_i as follows. We start the search at the root of the history

graph, which corresponds to the initial triangle Δ_0. Clearly this triangle contains p_i. The root has three children in the history graph, corresponding to the three triangles that replaced Δ_0 after the insertion of p_1. We simply test each of these three triangles to see which one contains p_i and we descend to the corresponding child node. This node has again a number of children (two or three); we test the corresponding triangles to see which one contains p_i and we descend to the corresponding child node. This process is repeated until we reach a leaf of the history graph. The triangle corresponding to this leaf is the triangle of the current triangle that contains p_i. This finishes our sketch of the algorithm.

The running time of the algorithm depends on the random permutation generated in step 2: for some permutations we create more triangles during the construction process than for other permutations, and for some permutations walking down the history graph will take more time than for others. But it can be shown that the *expected* number of triangles ever created is $O(n)$, and that the *expected* running time is $O(n \log n)$. As in all randomized algorithms, the expectation in the bounds has nothing to do with the distribution of the input points, but it is with respect the random permutation.

We used the history graph in the algorithm above to locate the next point to be inserted or, in other words, to locate the place where the Delaunay triangulation had to be changed. A similar approach works for many problems. Boissonnat et al. [31] describe the use of history graphs in combination with randomized incremental construction in an abstract framework. Among the applications where randomized incremental construction leads to simple and optimal algorithms are the computation of convex hulls [55, 131] and computing all intersections in a set of line segments or curves in the plane [55, 104, 106]. It is interesting to note that until recently [38] there was no optimal deterministic algorithm for convex hulls in dimensions greater than three. Also for the problem of computing the intersections in a set of curves in the plane an optimal deterministic algorithm was found only very recently [11]. Thus randomized algorithms are not only often simpler than their deterministic counterparts, they can also be asymptotically more efficient.

2.3 Random sampling

The two examples that we gave above were both based on the *randomized incremental construction* paradigm: the input objects were treated one by one in random order, and the 'solution' to the problem was updated every time. Randomization can also help in divide-and-conquer algorithms.

In randomized divide-and-conquer algorithms a randomly selected subset of the input objects is used to do the divide-step. We already saw an example of this in the introduction to this section, when we selected a random pivot element to do the divide-step in randomized quicksort. Here we shall describe a more geometric application of random sampling, which we will need in the next chapter.

Let L be a set of n lines in the plane. Suppose that we want to partition the plane into a small number of simple regions, triangles for instance, such that each region is intersected by only a quarter of the lines in L. More generally, given some parameter r we want to partition the plane into triangles, some of which may be unbounded, such that the interior of each triangle is intersected by at most n/r lines. Such a partition is called a $(1/r)$-*cutting*. The goal is to find a $(1/r)$-cutting consisting of as few triangles as possible.

Clarkson [53] showed that random sampling provides a very simple way of constructing such a cutting: we simply take a random sample R of the lines in L, and triangulate the faces of the arrangement induced by R. Fig. 7 illustrates this; the solid lines are the lines in the random sample, the dotted lines the remaining lines, and the dashed segments are added to triangulate the sample arrangement. Before we discuss how large we must choose the sample set R, let's try to understand why such a random sampling approach could work. The reason is that the interiors of the triangles we have created are intersected by none of the lines in R; since R is a random sample of L, this is good evidence that the triangles are likely to be intersected by only few of

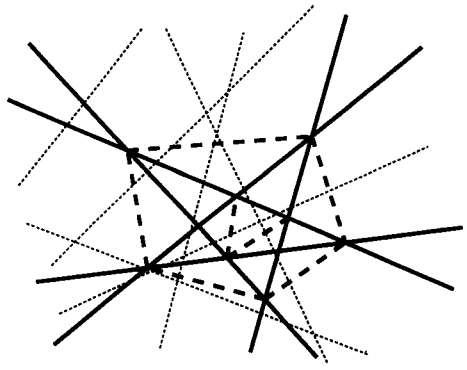

Figure 7: Constructing a cutting by random sampling.

the lines in L. It turns out that there is a constant c (independent of r) such that if we choose $cr \log r$ lines at random, then the triangles that we create are expected to be intersected by at most n/r lines from L. More precisely, the probability that there is a triangle intersected by more than n/r lines is less than $1/2$. To construct a $(1/r)$-cutting we thus choose a random sample of size $cr \log r$ and triangulate its arrangement. Then we check every triangle to see if it intersected by more than n/r lines. If there is such a triangle, we forget the current sample and try another one, and so on, until we succeed in finding a $(1/r)$-cutting. Because the probability that we have to try again is less than $1/2$ at each step, we expect to find a good sample in a few steps. Because the arrangement induced by $cr \log r$ lines has $O(r^2 \log^2 r)$ complexity, we will find a cutting of consisting of $O(r^2 \log^2 r)$ triangles. Using a two-level sampling strategy it is possible to construct a cutting that consist of only $O(r^2)$ triangles [43].

Notice that the number of intersections inside any of the triangles is $O(n^2/r^2)$, because any triangle is intersected by at most n/r lines. On the other hand, the total number of intersections among the lines in L is $\Omega(n^2)$. This implies that we need at least $\Omega(r^2)$ triangles in a $(1/r)$-cutting, so the result that can be obtained using the two-level sampling strategy is optimal.

The same approach works to construct cuttings in higher dimensions. There are also deterministic algorithms to construct cuttings [37, 91, 90] but these algorithms are more complicated than the randomized algorithms.

3 Geometric data structures

The design of data structures has always been an important topic in algorithms research. It is therefore not surprising that geometric data structures have received a lot of attention in computational geometry. Traditionally, geometric data structures dealt mostly with orthogonal objects. Examples of such data structures are range trees and segment trees [99, 114, 122]. There were also data structures for non-orthogonal objects, but their worst-case behavior was not very satisfactory. In recent years new tools have been developed, *cuttings* and *simplicial partitions*, which can be used as a basis for data structures dealing with non-orthogonal objects. It turns out that the asymptotic performance of these data structures is (close to) optimal. Another nice aspect of the data structures is that they can be used to construct multi-level data structures that can solve queries composed of a number of 'sub-queries'. In this section we sketch how these data structures are designed. Readers who want to know more details should consult the survey by Matoušek [94]. Although problems in three- and higher-dimensional spaces can also be tackled, we shall explain the principles on the basis of two-dimensional data structures.

3.1 Triangular range searching

In a range searching problem we are given a set of objects that we want to store in a data structure such that the objects intersecting a query region can be found quickly. Range queries, also called windowing queries, have many applications in graphics, geographic information systems, and other areas. In this section we mainly consider triangular range searching, where the objects are points in the plane and the query region is a triangle.

Data structures with linear space. The general idea behind the data structure we shall describe is simple and well known. We partition the plane into a number of simple regions, for example triangles or squares. With each region we store the points lying inside it.

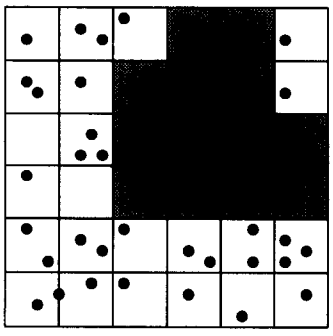

Figure 8: A simple range searching structure.

Let Δ be the query triangle. If a region is fully contained in Δ then we know that any point inside that region must also lie inside Δ, so we can report all points in the region. Similarly, if a region is fully disjoint from Δ, then we can ignore all the points in the region. The only difficult regions are the ones that are only partially inside Δ: points inside such regions may or may not be inside Δ. We say that such a region is *crossed* by Δ. Fig. 8 gives a simple example; the white cells are outside Δ and can be ignored, the dark gray cell is inside Δ and all points inside it can be reported, and the light grey cells are crossed by Δ and need further treatment. We can test all the points inside the crossed regions explicitly, but it is better to treat them recursively. Thus each region is partitioned into subregions. If we find that Δ crosses a region, then we check its subregions, and so on. (This is not shown in Fig. 8.) This approach gives rise to a tree whose nodes correspond to regions of the plane. The children of a node correspond to the subregions into which a region is partitioned. Quad trees and *kd*-trees [126, 127] are examples of data structures that work according to this principle.

The efficiency of this approach depends on the number of subregions that we have to visit or, more precisely, on the total number of points in these regions. Thus we want to find a partitioning of the plane such that for any query triangle the total number of points in the crossed regions is small. It turns out that *kd*-tree or quad tree partitionings give no good worst-case bound on the number of points in the crossed regions; because they only use horizontal and vertical lines in the partitioning they are not flexible enough. Hence, these structures have no good worst-case query time for triangular range searching. *Partition trees*, which we describe next, do better.

Let P be a set of n points in the plane. A *simplicial partition* for P is a collection of pairs $\{(P_1, t_1), \ldots, (P_r, t_r)\}$, where the P_i's are disjoint subsets of P whose union is P, and t_i is a triangle containing P_i. The subsets P_i are called *classes*.[1] We call r, the number of triangles, the *size* of the simplicial partition. Fig. 9 gives an example of a simplicial partition of size five; different shades of grey are used to indicate the different classes. The dotted triangles form a recursively computed simplicial partition for one of the classes of the initial simplicial partition. A range searching data structure based on simplicial partitions, called a *partition tree*, is constructed as follows. The root of the partition tree corresponds to the whole plane. We construct a simplicial partition of size r for the set P, for some (large enough) constant r. Each triangle t_i in the partition corresponds to a child of the root. This child is the root of a recursively defined structure on the class P_i. If P_i contains only one point, then the child corresponding to P_i is a leaf node where the point is stored—see Fig. 9. A partition tree has n leaves (one per point in P), which implies that it uses $O(n)$ storage.

A query with a triangle Δ is performed by walking down the partition tree in the way described earlier. Thus we test the triangle corresponding to each of the children of the root of the tree.

[1] This definition is slightly more general than what we used before: we do not require the triangles of the partition to be disjoint, nor do we require them to cover the whole plane. Because the triangles can intersect, a point could lie in more than one triangle. Still, such a point is a member of only one class.

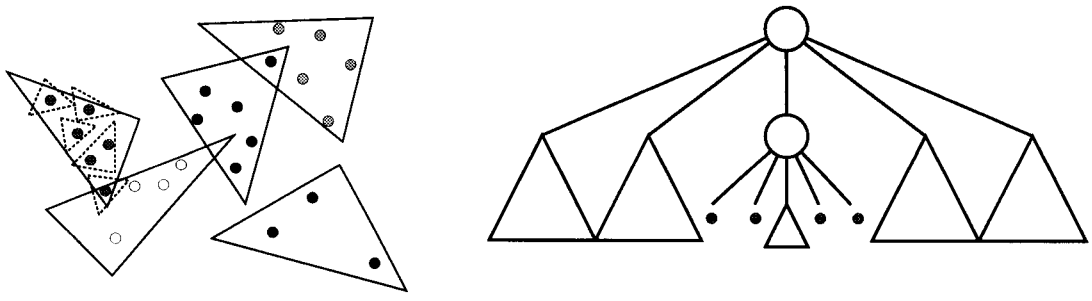

Figure 9: A simplicial partition and the corresponding tree.

If the triangle of a node ν is completely inside Δ we can report all points stored in the leaves of the subtree corresponding to ν, if the triangle is disjoint from Δ we can ignore the subtree, and if the triangle is crossed we visit the subtree recursively. As stated earlier, the query time depends on the *crossing number* of Δ, that is, the number of triangles of the simplicial partition crossed by Δ, and on the number of points in them. Matoušek [92] showed how to construct a simplicial partition such that any query triangle crosses $O(\sqrt{r})$ triangles of the partition. Each of the classes in his partition contains at most $2n/r$ points; hence, the partition is nicely balanced. Using this result one can construct partition trees whose query time is $O(n^{1/2+\varepsilon} + k)$, where ε is a small constant (in fact, ε can be made arbitrarily small) and k is the number of reported points. Using some additional tricks the query time can be improved to $O(\sqrt{n} + k)$ [93].

The data structure can not only be used to report the points in a query triangle, it can also count the number of points without explicitly listing them. The query time for such *range counting queries* is $O(n^{1/2+\varepsilon})$; again, this can be improved slightly to $O(\sqrt{n})$. This is very close to optimal: Chazelle [35] proved that any data structure for triangular range counting that uses $O(n)$ storage must have $\Omega(\sqrt{n})$ query time. The lower bound for range *reporting* is almost the same [47], so the structure is almost optimal for range reporting as well.

Data structures with logarithmic query time. The query time of the partition trees described above is $O(n^{1/2+\varepsilon}+k)$, which is perhaps not what one would hope for. A better query time can be obtained if one is willing to use more storage. We briefly indicate how this can be achieved. Instead of triangular range searching, we will consider *half-plane range searching*, where the query region is not a triangle, but a half-plane; in the next section we shall see how the data structure for half-plane range queries can serve as the basis for a structure for triangular range queries. So we want to store the set P in a data structure such that the points in P above (or below) a query line can be reported quickly. Using *dualization* [67] we can reverse the roles of points and lines, that is, we can transform the half-plane range searching problem to the following problem: store a set L of n lines in a data structure such that the lines above (or below) a query point can be reported quickly. This dual version of the problem can be attacked using *cuttings*, a concept we saw in the previous section. Recall that a $(1/r)$-cutting of a set of n lines is a partitioning of the plane into triangles (here the triangles must be disjoint and cover the whole plane) such that any triangle is intersected by at most n/r lines. Suppose that we have a $(1/r)$-cutting for our set L of lines. Consider a triangle t in the cutting. We can classify each of the lines in L with respect to t into a class $L^+(t)$ of lines that lie above t, a class $L^-(t)$ of lines that lie below t, and a class $L^\times(t)$ of lines that intersect t. Suppose that we want to report all lines above a query point q, and suppose that q lies in the triangle t. Then we can report all the lines in $L^+(t)$, and ignore all the lines in $L^-(t)$. Thus we only have to worry about the lines in $L^\times(t)$, of which there are at most n/r. These lines are treated recursively. This way we get a structure that is very similar to a partition tree. Because the query point lies in only one triangle of the cutting, we only have to recurse into one subtree. Hence, we get $O(\log n + k)$ query time, where k is the number of

reported lines. On the other hand, the lines—which are now the data objects—intersect many triangles of the cutting. Hence, a line is stored in several subtrees. This means that the amount of storage is no longer linear. One can show that the amount of storage becomes $O(n^{2+\varepsilon})$, where ε is a positive constant that can be made arbitrarily small. A slightly better bound is achieved by Matoušek [93].

Again, the data structure can also be used for range counting. The result for range counting is close to optimal, because there is an $\Omega(n^2/\log^2 n)$ lower bound on the amount of storage of any data structure for half-plane range counting with $O(\log n)$ query time [35]. It is interesting that a much better solution can be obtained for half-plane range reporting: Chazelle et al. [46] showed that one can achieve $O(\log n + k)$ query time using only linear storage. With this structure it is not possible, however, to do range counting without explicitly listing all the points, nor is it possible to extend the structure to triangular range searching.

Trade-offs and higher dimensional results. We saw two data structures for triangular range searching: one using linear storage with $O(n^{1/2+\varepsilon})$ query time, and one using $O(n^{2+\varepsilon})$ storage with $O(\log n)$ query time. (Actually, the latter structure was described for half-plane range searching, but it can be extended to triangular range searching, as explained later.) It turns out to be possible to construct structures whose performance lies between these two extremes: for any m with $n \leqslant m \leqslant n^2$ there is a data structure that uses $O(m)$ storage and has $O(n^{1+\varepsilon}/\sqrt{m})$ storage [48].

It is also possible to generalize the results to *simplex range searching* in higher-dimensional spaces: it is possible to report the points inside a query simplex in $\mathbb{R}^d$ in $O(n^{1-1/d} + k)$ time with a data structure that uses $O(n)$ storage [93], or to report the points in $O(\log n + k)$ time with a data structure that uses $O(n^{d+\varepsilon})$ storage [48]. Trade-offs are possible as well. Notice that the performance of the data structure deteriorates quickly when the dimension gets higher. Unfortunately this cannot be unavoided, as follows from the lower bounds proved by Chazelle [35].

3.2 Multi-level data structures

Multi-level data structures solve queries that are composed of several 'sub-queries'. An example of such a composed query is *rectangular range searching* on a set of points in the plane: report the points inside a rectangular window $W = [x_1 : x_2] \times [y_1 : y_2]$. This query is composed of two sub-queries, one on x-coordinate and one on y-coordinate. In general, we want to report objects that satisfy a number of restrictions simultaneously. Each level of the data structure is used to select those objects that satisfy one of the restrictions. The levels work together in such a way that finally all objects are selected that satisfy all restrictions. In the rectangular range searching example, we would need a two-level data structure; the first level selects those points whose x-coordinate lies in the range $[x_1 : x_2]$, and the second level selects out of those points the ones whose y-coordinate lies in the range $[y_1 : y_2]$. For rectangular range searching this leads to a data structure called a *range tree* [122]. We now sketch the technique in a general framework.

Let S be a set of objects, and let $\mathcal{Q}$ be a family of query objects. In our running example, rectangular range searching, S is a set of points in the plane and $\mathcal{Q}$ is the infinite family of all possible axis-parallel rectangles in the plane. Suppose that for an object $s \in S$ and a query object $q \in \mathcal{Q}$ there are two predicates $P_1(s, q)$ and $P_2(s, q)$. In the example we have

$$P_1((p_x, p_y), [x_1 : x_2] \times [y_1 : y_2]) \quad == \quad (x_1 \leqslant p_x \leqslant x_2)$$

and

$$P_2((p_x, p_y), [x_1 : x_2] \times [y_1 : y_2]) \quad == \quad (y_1 \leqslant p_y \leqslant y_2).$$

Our goal is to report those objects that satisfy $P_1(s, q)$ and $P_2(s, q)$. Let's assume for the moment that we have a large pool of data structures available for various subsets of S. These data structures should be able to report objects satisfying the second predicate. We call these data structures *second-level data structures*, and we call the subsets for which they are constructed *canonical*

subsets. In our example, the second-level data structures must be able to report points whose y-coordinate lies in a query range; a binary search tree can be used to report those points in logarithmic time, plus additional time that is linear in the number of reported points. Let the second-level data structures be numbered $\mathcal{D}_1, \mathcal{D}_2, \ldots$ and denote the canonical subset of S for which $\mathcal{D}_i$ is constructed by S_i. Suppose that, by some magic procedure, we could select a number of data structures $\mathcal{D}_{j_1}, \mathcal{D}_{j_2}, \ldots, \mathcal{D}_{j_m}$ from our pool such that the canonical subsets $S_{j_1}, S_{j_2}, \ldots, S_{j_m}$ are pairwise disjoint and their union contains exactly those objects that satisfy the first predicate. Then we can query each of these data structures and report the objects satisfying the second predicate. Because the objects in the selected data structures are exactly the ones satisfying the first predicate this will solve our composed query. If the number of selected data structures is small, then the total query time will be reasonable. But how do we get such a magic pool of data structures? This is often possible if we have a data structure, the *first-level data structure*, that can report points satisfying the first predicate.

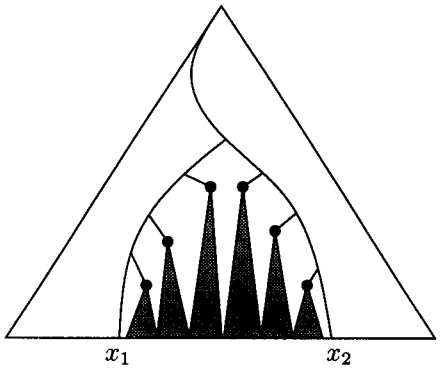

Figure 10: The points with x-coordinates between x_1 and x_2 lie in $O(\log n)$ subtrees.

In our example, the objects are points in the plane, and the first predicate for a point p is $x_1 \leqslant p_x \leqslant x_2$, where x_1 and x_2 are query values. Such a one-dimensional range query can be answered in $O(\log n)$ time with a balanced binary search tree on the x-coordinates of the points; the points whose x-coordinate lies in the correct range lie in between the leaves where the search paths to x_1 and x_2 end—see Fig. 10. Notice that the points are exactly the leaves of a logarithmic number of subtrees of the search tree; in Fig. 10 these subtrees are shown grey. Define the canonical subset of a node ν in the tree to be the subset of points in S stored in the leaves of the subtree rooted at ν. The observation we just made can now be restated as follows: there are $O(\log n)$ nodes whose canonical subsets exactly contain the points whose x-coordinates lie in the correct range. So if we have for each node of the search tree a pointer to a data structure on its canonical subset that can report points satisfying the second predicate, then we have our pool of data structures. Notice that the number of second-level data structures that we have to query is only $O(\log n)$. It follows that the total query time to report the points inside an axis-parallel query window is $O(\log^2 n + k)$, where k is the number of reported points. (Using a technique called *fractional cascading* [44, 45] this can be reduced to $O(\log n + k)$.) It seems at first sight that the data structure uses a lot of storage, because there is a linear number of second-level data structures—one per node in the first-level tree. Fortunately, many of the second-level data structures contain only a small number of points. More precisely, the second-level data structures associated with all the nodes on one level together only contain $O(n)$ points. This implies that the total amount of storage used by all second-level data structures is $O(n \log n)$. In general, the amount of storage used by multi-level data structures is kept small because there are not too many large canonical subsets.

Let's consider one more example. The objects are lines in the plane, and the first predicate is that the points should lie above a query point. We do not specify what the second predicate is, but we assume we can construct a data structure for it. How do we define our pool of second-level data structures? As before, we use a data structure for reporting the points satisfying the first predicate for this purpose. In the previous subsection we saw a data structure for reporting the points above a query line. This structure was based on cuttings; each node ν in the structure corresponds to a triangle t_ν in a cutting. Our pool will contain a second-level data structure for each node ν, built for the set $L^+(t_\nu)$ of lines that lie above t_ν.

We promised earlier that the data structure for half-plane range queries can be used to answer triangular range searching. Here is how it is done. A point lies inside a triangle if and only if it lies on a certain side of each of the three lines through the edges of the triangle. Hence, we wish

to report the points satisfying three restrictions. Each restriction states that the data point must lie on a specified side of a query line. If we exchange the roles of points and lines by applying a duality transform, then each of the restrictions states that the data line must lie on a specified side of a query point. We can answer such a query with a three-level data structure, where each level is a structure based on cuttings. This will lead to a data structure with $O(\log^3 n + k)$ query time that uses $O(n^{2+\varepsilon})$ storage. By applying an extra trick, the query time reduces to $O(\log n + k)$ [48].

Multi-level data structuring is now a standard tool in computational geometry. Basically, any query that is composed of a constant number of sub-queries that are half-space range queries in some two-, three-, or higher-dimensional space can be answered with a multi-level data structure. The efficiency of the multi-level data structure depends mainly on the dimension of the half-space queries. Van Kreveld [87] presents the theory of multi-level data structures in a general way and gives many sophisticated applications. Matoušek's survey [94] also discusses multi-level structures.

4 Decomposition techniques

Divide-and-conquer is a technique that has proved useful for a variety of problems. In a geometric setting the divide-step is often performed by decomposing the space into regions. For a number of different cases algorithms have been developed to perform the decomposition step. In this section we discuss the new results that have been obtained in this area. We shall concentrate our attention on BSP trees, meshing algorithm, and polygon triangulation.

4.1 BSP trees

A binary space partition, or BSP, for a set of objects in d-dimensional space is obtained by recursively splitting the space with a hyperplane until there is only one (fragment of) an object left in each cell of the partitioning. Fig. 11 shows a BSP for a set of objects in the plane: first the plane is partitioned with ℓ_1, then we split the half-plane above ℓ_1 with ℓ_2 and the half-plane below ℓ_1 with ℓ_3, and so on. The splitting lines not only partition the plane, they may also cut objects into fragments. The splitting continues until there is only one fragment left in the interior of each region.[2] This process is naturally modeled as a binary tree. Each leaf of this tree corresponds

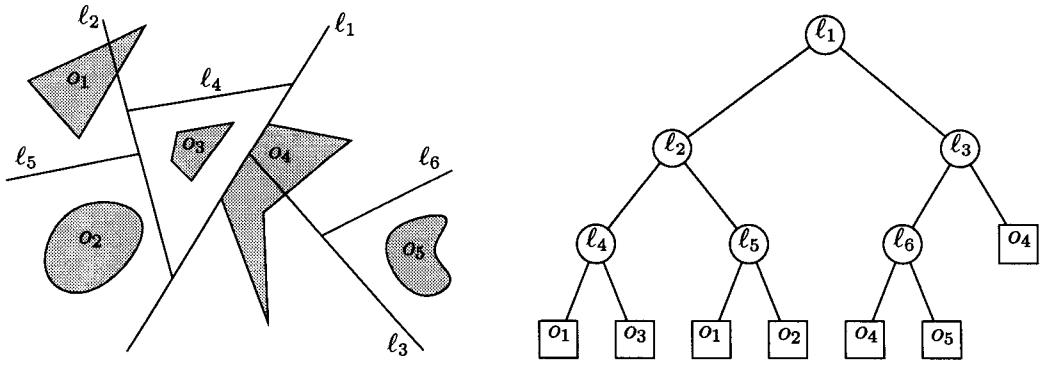

Figure 11: A binary space partition and the corresponding tree.

to a face of the final subdivision; the object fragment that lies in the face (that is, a completely description of the fragment) is stored at the leaf. Each internal node corresponds to a splitting

[2]This implies that the objects must have disjoint interiors, since otherwise it is not possible to obtain a partitioning with only one object per cell. If one wants to allow intersecting objects then the definition of the BSP has to be adapted. We assume in this section that the objects have disjoint interiors.

line; this line is stored at the node. When there are one-dimensional objects (line segments) in the scene then objects could be contained in a splitting line; in that case the corresponding internal node stores these objects in a list.

Formally, a binary space partition tree, or BSP tree, for a set S of objects in d-dimensional space is a binary tree $\mathcal{T}$ with the following properties. Let $|S|$ denote the cardinality of the set S.

- If $|S| \leqslant 1$ then $\mathcal{T}$ is a leaf; the object fragment in S (if it exists) is stored explicitly at this leaf.
- If $|S| > 1$ then the root ν of $\mathcal{T}$ stores a hyperplane h_ν, together with the set of objects that are fully contained in h_ν. The left child of ν is the root of a BSP tree $\mathcal{T}^+$ for the set $S^+ := \{h_\nu^+ \cap s : s \in S\}$, where h_ν^+ is the region above h_ν. The right child of ν is the root of a BSP tree $\mathcal{T}^-$ for the set $S^- := \{h_\nu^- \cap s : s \in S\}$, where h_ν^- is the region below h_ν.

We denote the set of objects stored at a node ν (either an internal node or a leaf) by $S(\nu)$. The *size* of a BSP tree is the total size of the sets $S(\nu)$ over all nodes ν of the BSP tree. In other words, the size of a BSP tree is the total number of object fragments that are generated. If the BSP does not contain useless splitting lines—lines that split off an empty subspace—then the number of nodes of the tree is at most linear in the size of the BSP tree.

Binary space partitions have been used for a large number of purposes in computer graphics: they are used for hidden surface removal with the painter's algorithm [74], for shadow generation [52], for set operations on polyhedra [109, 140], and for visibility preprocessing for interactive walkthroughs [139].

The efficiency of algorithms based on BSPs depends crucially on the size of the BSP. Hence, when constructing a BSP of a given scene, one should choose the splitting hyperplanes carefully, so that the fragmentation of the objects is kept small. Paterson and Yao [117, 118] gave good strategies for choosing the splitting hyperplanes and proved bounds on the worst-case size of the resulting BSPs. Next we discuss their results.

BSP trees in the plane. First, let's consider the planar problem where the set S of objects consists of n line segments. This situation arises when one wants to construct a BSP for the walls in the floor plan of a building. Paterson and Yao describe two methods for this case.

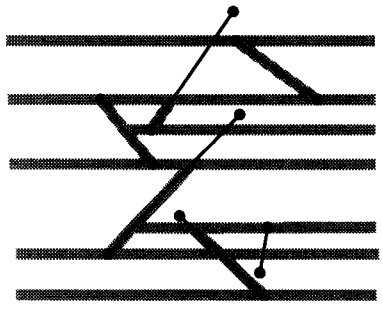

Figure 12: A BSP created with the trapezoid method.

The first method is deterministic. It is actually the same as the trapezoid method for planar point location [122]. It works as follows. The subspaces created by the algorithm will always be trapezoids; two of the edges of the trapezoids will be horizontal. The trapezoids can be unbounded on one or more sides, and they can degenerate into triangles. The initial trapezoid is simply the whole plane; thus it is a trapezoid that is unbounded on all four sides. Trapezoids are split according to the following rules.

- If there is only one segment left inside the trapezoid then nothing is done: the trapezoid will be a cell in the final BSP.
- If there is more than one segment inside the trapezoid and there is a segment s that completely crosses it—that is, a segment that intersects the trapezoid without having an endpoint in its interior—then the trapezoid is split using the segment s. (In case there are more such segments, an arbitrary one can be chosen.) Such a split is called a *free split.*
- If neither of the first two cases occurs, then the trapezoid is split with a horizontal line. This line is chosen such that the number of segment endpoints above it and the number of segment endpoints below it are at most half the total number of segment endpoints in the trapezoid.

Fig. 12 gives an example of a BSP produced with this method. One can prove that this approach produces a BSP tree of size $O(n)$. Furthermore, the construction can be done in $O(n \log n)$ time [122, 117].

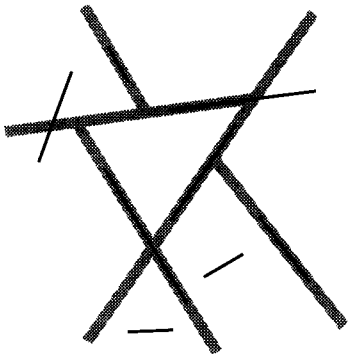

Figure 13: An auto-partition.

The second method uses *auto-partitions.* These are partitions where every splitting line contains one of the input segments, as in Fig. 13. In the first method this is obviously not the case, because the horizontal splitting lines do not contain input segments. Paterson and Yao create an auto-partition by simply adding the splitting lines through the input segments in random order. More precisely, at each step in the algorithm they pick a random segment s that has not been selected before, and they split all the cells that are intersected by this segment with the line through s. (For cells where s is the only segment left, the splitting is not necessary.) One can prove that the expected size of the resulting BSP is $O(n \log n)$. Paterson and Yao also showed that this randomized method can be made deterministic, and they show that the construction can be done in $O(n^2)$ time in the worst case.

Paterson and Yao also studied the special case of line segments that are axis-parallel, that is, segments that are either parallel to the x-axis or parallel to the y-axis. For this case they were able to give a partitioning strategy that is guaranteed to produce a BSP of linear size. D'Amore and Franciosa [7] get the same result, with a slightly better constant. Another special case where a linear size BSP can always be constructed is when the longest segment is only a constant times longer than the shortest one [20]. It is still open whether any set of segments in the plane admits a linear size BSP; the best known bound is still the $O(n \log n)$ bound by Paterson and Yao.

BSP trees in three-dimensional space. Paterson and Yao also studied BSPs for sets of planar faces in three-dimensional space. Let S be a set of n triangles in $\mathbb{R}^3$ with disjoint interiors. The randomized approach that we described earlier for planar BSPs can easily be adapted for the three-dimensional case: at each step, choose one of the not yet selected triangles at random, and use the plane through this triangle as the next splitting plane. The expected size of the resulting BSP is $O(n^2)$. As in the planar case, this randomized algorithm can be made deterministic. This result is rather disappointing: a quadratic size BSP is too large to be useful in practice. Unfortunately, this result is the best possible worst-case result, since there are sets of triangles for which any BSP must have size $\Omega(n^2)$ [117]. For axis-parallel rectangles the situation is slightly better: in this case it is always possible to construct a BSP of size $O(n\sqrt{n})$. Again, this result is the best possible [118].

The lower bound constructions for both the case of triangles and the case of rectangles are very artificial. Hence, one would hope that better results are possible in practical situations. Section 7 discusses this issue further.

4.2 Meshing algorithms

Meshing (subdividing a domain into smaller, simpler pieces) is the first step in many computational problems that are too complex to be tackled directly. A well known example is solving partial differential equations over a domain; solving the PDE analytically is infeasible in most cases, so one usually confines oneself to finite element methods. The first step in finite element methods is to compute a mesh of the given domain.

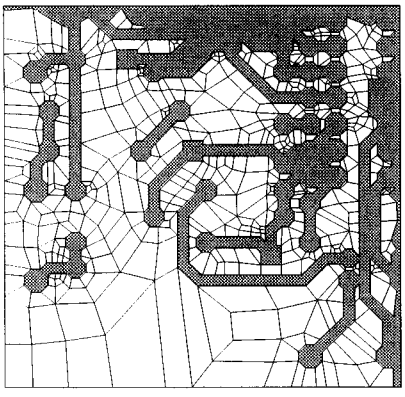

Figure 14: A quadrilateral mesh for a printed circuit board.

Fig. 14 shows an example of a two-dimensional mesh; the domain is a printed circuit board, and the mesh-elements are quadrilaterals. (The mesh from this example was used in simulations to test if the the board doesn't emit too much radiance.) An example in computer graphics where meshing plays an important role is radiosity: here the first step is to subdivide the patches in the scene into smaller elements. In the past few years there have been a number of papers in computational geometry dealing with the meshing problem. Bern and Eppstein [27] give an extensive overview of this area. We shall briefly describe a few of the results.

One can distinguish *structured* and *unstructured* meshes. Structured meshes are usually (deformed) grids; unstructured meshes are often triangulations. We shall restrict our discussion to unstructured grids. Furthermore, we mainly concentrate on the case where the domain to be meshed is two-dimensional and polygonal. For most applications it is important that the mesh be *conforming*, which means that there is no vertex of one triangle lying in the interior of an edge of another triangle. In other words, T-vertices are not allowed in the mesh. (Sometimes the term 'consistent' is used instead of 'conforming'.) The vertices of the triangles in the mesh can either be vertices of the domain, or they can be vertices added by the meshing algorithm. The latter type of vertices is called *Steiner vertices*. Steiner vertices are needed because one usually wants the mesh to meet one or both of the following criteria:

- There should be *no small angles*, that is, every angle of every triangle in the mesh should be at least some fixed (not too small) constant θ. This constant should not be larger than the smallest angle of the input domain, because we cannot avoid the input angles in the mesh.
- There should be *no obtuse angles*, that is, no angles larger than 90°.

Often the goal is to minimize the number of mesh elements under the given conditions.

No small angles. Let's first consider minimizing the number of triangles in the mesh under the condition that there be no small angles. The number of triangles that we need to mesh a polygonal domain under this condition does not only depend on the number of vertices of the domain; it also depends on the shape of the domain. To see this we introduce a parameter that is closely related to the minimum angle of a triangle, namely the *aspect ratio* of the triangle. This is the ratio of the length of the longest side of the triangle to the height of the triangle, where the height of a triangle is the Euclidean distance of the longest edge to its opposite vertex. If the smallest angle of a triangle is θ then the aspect ratio is between $1/\sin\theta$ and $2/\sin\theta$. Now consider a rectangular domain whose shorter sides have length 1 and whose longer sides have length A. Suppose we require that the minimum angle be, say, 30°. This implies that the aspect ratio of any triangle in the mesh must be less than or equal to $2/\sin 30 = 4$. Furthermore, the height of any triangle in the domain is at most one. Hence, the area of any triangle is $O(1)$. Because the total area of the rectangular domain is A, this implies that we need at least $\Omega(A)$ triangles in the mesh. Bern et al. [29] describe a method based on quad trees that produces an asymptotically optimal number of triangles.

Minimizing the number of triangles is not always the goal of meshing algorithms. It can also be important to be able to control the mesh density, so that one can have a dense mesh in interesting areas and a coarse mesh in uninteresting areas. This is the setting studied by Chew [51]. He describes a meshing algorithm that allows the user to define a function that determines whether a triangle of the mesh is fine enough. The angles of the triangles produced by his algorithm are between 30° and 120°. Another nice aspect of his work is that the algorithm not only deals with planar regions, but also with regions on surface patches.

No obtuse angles. If the only requirement is that the triangles in the mesh are non-obtuse then it turns out to be possible to construct a mesh for a given polygonal domain whose number of triangles only depends on the number of vertices of the domain. More precisely, Bern and Eppstein [28] have shown that for any polygonal domain with n vertices there is a mesh consisting of $O(n^2)$ non-obtuse triangles. Quite recently, Bern et al. [30] improved this bound to $O(n)$.

No small and no obtuse angles. If we want both criteria to be satisfied, then the number of triangles that we need again depends on the shape of the domain. Melissaratos and Souvaine [101] extended the approach of Bern et al. [29] for computing a mesh without small angles so that it also avoids obtuse triangles. The number of triangles in the mesh is still at most a constant factor from optimal.

4.3 Triangulating simple polygons and polyhedra

Perhaps the best known decomposition problem is polygon triangulation: partition a simple polygon in the plane into triangles—see Fig. 15.

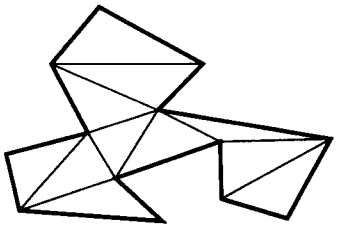

Figure 15: A triangulation of a simple polygon.

The vertices of the triangles in the triangulation should be vertices of the original polygon. This is the major difference with meshing, where it is allowed to add Steiner vertices. It is not difficult to show that a triangulation of a simple polygon always exists. It has been known for a long time that an n-vertex simple polygon can be triangulated in $O(n \log n)$ time [75]. Tarjan and Van Wyk [137] improved this to $O(n \log\log n)$, which was subsequently improved by Clarkson et al. [56] to $O(n \log^* n)$ with a randomized algorithm. (Here $\log^* n$ is the iterated logarithm, a very slowly growing function [59].) It was one of the major challenges in computational geometry to develop a linear time triangulation algorithm. Chazelle [36] managed to do this. His algorithm is quite complicated, so although it runs in linear time it is not very useful in practice. Seidel [130] presented a very simple randomized algorithm whose running time is $O(n \log^* n)$.

In the three-dimensional version of the triangulation problem we are given a simple polyhedron that must be decomposed into tetrahedra. Again, one is only allowed to use the vertices of the polyhedron as vertices of the tetrahedra. Unlike planar polygons, simple polyhedra cannot always be decomposed without adding Steiner vertices: there are polyhedra that require extra vertices to obtain a decomposition into tetrahedra. Ruppert and Seidel [125] have shown that it is NP-complete to decide if a polyhedron can be tetrahedralized without Steiner points.

5 Robustness in geometric computations

The model of computation usually adopted in computational geometry is that of a RAM (random access machine) with real arithmetic. Unfortunately, real arithmetic is not available on real machines. Another problem that arises when implementing geometric algorithms from the literature is that degenerate cases are often omitted from the descriptions. In this section we discuss some of the research that has been done to deal with these problems.

Dealing with finite precision. Most papers in computational geometry assume that computations with real number can be performed exactly and at unit cost. Time bounds for and correctness of the algorithms are proved under this assumption. Implementing the algorithms using floating point arithmetic is thus problematic: round-off errors can cause the program to produce the wrong output or even to crash.

Many basic geometric decisions (such as deciding whether a point lies on a line, or whether a point lies inside the circle defined by three other points) boil down to the evaluation of the sign of a certain determinant. A possible way to deal with the inexactness in floating point arithmetic when evaluating the sign of a determinant is to choose a small threshold value ε and to say that the determinant is zero when the outcome of the floating point computation is less than ε. When implemented naively, this can lead to inconsistencies (for instance, for three points a, b, c we may decide that $a = b$ and $b = c$ but $a \neq c$) that cause the program to fail. Guibas et al. [79] showed that combining such an approach with interval arithmetic and backwards error analysis can give robust algorithms. Another option is to use *exact arithmetic.* Here one computes as many bits of the determinant as are needed to determine its sign. This will slow down the computation, but techniques have been developed to keep the performance penalty relatively small [73, 143]. Besides these general approaches, there have been a number papers dealing with robust computation in specific problems [10, 13, 34, 60, 71, 72, 84, 102].

Dealing with degeneracies. Most algorithms described in the computational geometry literature make the assumption that the input is in general position. For example, for computing the intersections in a set of line segments it is often assumed that no three segments meet in a common point, that no two endpoints have the same x-coordinate, etc. This assumption is usually accompanied by the statement that "the algorithm can be modified so that it also works in degenerate situations". Although this might be true, it is not very helpful when one wants to implement the algorithm.

One way to deal with this problem, is *symbolic perturbation.* In this technique, introduced by Edelsbrunner and Mücke [68] and later refined by Yap [142] and Emiris and Canny [69, 70], the input is perturbed slightly so that degeneracies disappear. Suppose that the input consists of n points in $\mathbb{R}^d$. Let $p_i = (p_{i,1}, p_{i,2}, \ldots, p_{i,d})$ be the i-th input point. Symbolic perturbation replaces each input point p_i by a point $\hat{p}_i$ lying very close to p_i. For example, the scheme of Emiris and Canny replaces the point p_i by $\hat{p}_i = (p_{i,1} + \varepsilon \cdot i, p_{i,2} + \varepsilon \cdot i^2, \ldots, p_{i,d} + \varepsilon \cdot i^d)$, where ε is a positive infinitesimal. The perturbation is done symbolically. Thus one does *not* substitute a very small value for ε, but one computes with ε in a symbolic way. Formally, ε is an extension of the reals, larger than zero but smaller than any positive real number. Because $i_1^{j_1} \neq i_2^{j_2}$ unless $i_1 = i_2$ and $j_1 = j_2$, the perturbation of any two coordinates is different. It can be shown that this and the fact that ε is an infinitesimal imply that no two perturbed points are on a common vertical line, no three points are collinear, etc.

Symbolic perturbation is a very elegant method. Its power lies in the fact that degeneracies are taken care of by a separate subroutine, which can be used by a variety of geometric algorithms. Unfortunately, symbolic perturbation also has some of drawbacks. An obvious drawback is in the performance: the use of a symbolic perturbation library slows down the algorithm. Another drawback arises when the 'perturbed result' (that is, the result of the algorithm run on the perturbed points $\hat{p}_i$) is not good enough. If this is the case one needs to recover the 'unperturbed result' from the 'perturbed result' in a postprocessing step. This can be non-trivial and time-consuming. These drawbacks led Burnikel et al. [33] to the claim that it is both simpler (in terms of programming effort) and more efficient (in terms of running time) to deal directly with degenerate inputs. They support their claim by two case studies, one for the line segment intersection problem and one for the convex hull problem.

6 Graphics-related problems

Geometric problems arising in computer graphics have always been popular in computational geometry. Because of the nature of computational geometry, these problems are studied in an object space setting. In recent years there has been a number of interesting results and developments in this area. We already saw some of them in previous sections, for example when BSP trees were discussed. In this section we discuss a number of other graphics-related problems that have been studied. The book by de Berg [18] contains an extensive treatment of many of the problems discussed in this section.

6.1 Hidden surface removal

Hidden surface removal is one of the basic steps required to render a three-dimensional scene. There are two fundamentally different types of algorithms for hidden surface removal [136]: image-space algorithms and object-space algorithms. Image-space methods calculate for each pixel in the image which object is visible. Object-space algorithms, on the other hand, compute for each input object which parts are visible. Almost all current graphics systems use the Z-buffer algorithm for hidden surface removal. This image-space algorithm is very simple and, hence, very fast. Nevertheless, object-space hidden surface removal offers a number of advantages. In recent years a number of new object-space hidden surface removal algorithms have been developed in computational geometry. Dorward [64] gives a nice survey of most of the results.

Output-sensitive hidden surface removal. Let S be a set of objects in $\mathbb{R}^3$, let p_{view} be the viewpoint, and let h be the view plane. We assume that the objects are polyhedral and non-intersecting. The projections of the visible portions of the objects in S onto h define a planar subdivision, called the *visibility map*. Each face in this subdivision corresponds to a maximal connected region where one object is visible, or none of the objects is visible. Object-space algorithms compute a combinatorial description of the map: they explicitly compute the vertices, edges, and faces of the map, and they label each face with the corresponding visible object. Note that the *combinatorial complexity* of the visibility map can range from constant (if there is one large polygon obscuring all the others) to quadratic (if the input consists of n long and thin triangles that are arranged in a grid-like pattern).

Every vertex of the visibility map is either a projected vertex of one of the input polyhedra, or the intersection of two projected edges. Many of the early hidden surface removal algorithms compute the visibility map as follows: they project the edges of the input polyhedra, compute all intersections among the projected edges, and determine which intersection points and pieces of projected edges are visible [76, 96, 111, 128]. These algorithms spend time to compute every intersection between two projected edges. Ideally, one would like to spend time only on the intersections that are visible. In other words, one would like to have an algorithm whose running time not only depends on n, the total number of vertices in the input, but also on k, the combinatorial complexity of the visibility map. Such algorithms are called *output-sensitive*. Most of the recent work in computational geometry has focussed on output-sensitive algorithms.

The first output-sensitive algorithm was described by Güting and Ottmann [80]. It roughly works as follows. The polygons are treated one by one, in order of increasing distance to the viewpoint. Hence, when a new polygon is processed, all the others that can hide it already have been treated. The algorithm maintains the union of their projections onto the viewing plane. The part of the new polygon that is visible is exactly the part whose projection lies outside the current union. So what is needed is a suitable data structure for storing the union, which allows to quickly compute the part of a query polygon outside the current union. Güting and Ottmann presented such a data structure for the case of *window rendering*, where the input polygons are rectangles parallel to the view plane. (Actually, their solution is more general: it can handle polygons parallel to the view plane whose edges have only a limited number of different orientations.) This way they were able to solve the window rendering problem in $O((n+k)\log^2 n)$ time. This result was later improved to $O((n+k)\log n)$ by various authors [18, 22, 25, 77].

Another special case where the union maintenance approach is successful is for *polyhedral terrains* [123, 124]. However, the fastest algorithm for terrains, presented by Katz et al. [86], uses a different method. This algorithm runs in time $O((n\alpha(n)+k)\log n)$, where $\alpha(n)$ is the extremely slowly growing functional inverse of Ackermann's function. It can also be used in some other special cases.

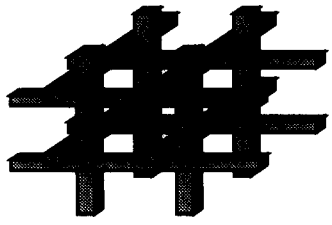

Figure 16: Blocks with cyclic overlap.

The strategy of maintaining the union has also been used for the general problem, where the edges of the polygons have arbitrary orientations. There is one major problem with the approach: the polygons must be processed in order of increasing distance to the viewpoint. This means that a depth order on the polygons is needed. Such an order is difficult to compute and, which is worse, it does not always exist: even in the simple case of axis-parallel blocks there can be cyclic overlap, as in Fig. 16. The main challenge was to develop an output-sensitive algorithm that could deal with cyclic overlap. De Berg and Overmars [22, 23] presented such an algorithm for the case of axis-parallel polygons. Later de Berg et al. [21] extended the technique to arbitrary polyhedra. The algorithm uses *map tracing*, an approach first suggested by Overmars and Sharir [115].

The basic idea is as follows. Suppose we know that the intersection of two projected edges e and e' is visible. In other words, we know a vertex v of the visibility map. Starting from this vertex we search for the other vertices of the visibility map that are adjacent to it. Thus we follow the visible portions of e (and e'), starting from their intersection, until we reach a new vertex of the visibility map. Such a vertex is either the projection of an endpoint of e, or it is the intersection of the projection of e with the projection of a third edge e''. However, not all such intersections are visible—see Fig. 17. Hence, one needs a way to skip invisible intersections. De Berg and

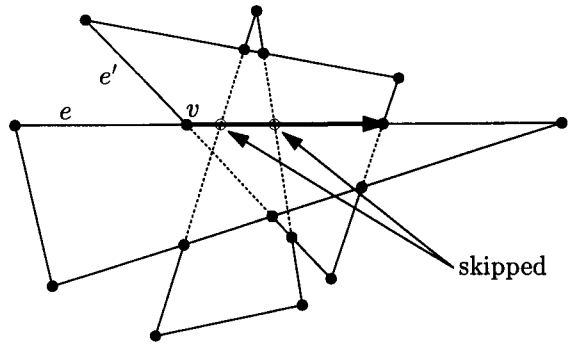

Figure 17: Map tracing.

Overmars showed how to store the polygons into a data structure such that the next vertex of the visibility map can be found efficiently, while skipping invisible intersections. Their approach leads to an $O(n^{1+\varepsilon}\sqrt{k})$ time algorithm for hidden surface removal in a set of non-intersecting triangles. Here ε is a positive constant that can be made arbitrarily small (at the cost of increasing the constant factor hidden in the O-notation). Notice that the running time of the algorithm is close to linear when k is constant, while it is close to quadratic when k is quadratic. Later Agarwal and Matoušek [3] improved this to $O(n^{1+\varepsilon}+n^{2/3+\varepsilon}k^{2/3})$. This is the best time bound known so far for the output-sensitive hidden surface removal for a set of triangles. A detailed description of the algorithm and an extension to intersecting polyhedra is given by de Berg [18].

Mulmuley [105] presents a randomized hidden surface removal algorithm. His algorithm is not output-sensitive, because it also spends time on some invisible intersections. The probability that one spends time on an invisible intersection decreases with the number of objects hiding it from

the viewpoint. For this reason the algorithm is sometimes called *quasi output-sensitive.*

Variants of the hidden surface removal problem. A number of papers deal with *dynamic* variants of the hidden surface removal problem. Some of them consider the situation where objects can be inserted into or deleted from the scene [15, 18, 25, 49]. Other papers study changing viewpoints [26, 89, 107, 121]. Another interesting variant is *generalized hidden surface removal* [17]: given a set of objects, a viewpoint, and a (point) light source, compute which parts of the objects are visible, subdivided into portions that are lit and portions that are in shadow.

Depth sorting. A possible approach to hidden surface removal is to sort the objects in the scene from back to front (with respect to the viewpoint) and then display the objects in that order [110]. This algorithm is called the *painter's algorithm* and the order that is needed is called a *depth order.* Depth orders are not only useful for the painter's algorithm, they are also needed in some of the output-sensitive algorithms discussed above. There we saw that a depth order does not always exist, because there can be cyclic overlap—see Fig. 16. In such cases the objects should be cut into pieces for which a depth order exists; a possible way to do this is by constructing a BSP on the set of objects. Using BSP trees one can in fact easily obtain a depth order for any viewpoint.

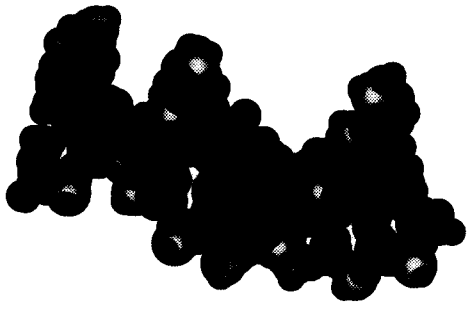

Figure 18: A piece of DNA.

There has been a number of papers dealing with the computation of depth orders, both in the plane and in three-dimensional space. De Berg [18] surveys most of this work. We restrict ourselves here to one recent result by Halperin and Overmars [82].

Halperin and Overmars study the rendering of molecular models in the *hard sphere model.* In this model the atoms are represented by spheres, and atoms with a common binding interpenetrate each other slightly—see Fig. 18. They show how to cut the spheres into $O(n)$ pieces for which a depth order exists. Their algorithm for computing the pieces and a depth order on them runs in $O(n \log n)$ time. Experiments show that the algorithm is quite fast is practice and that it produces very nice results.

6.2 Ray tracing

The basic operation used in ray tracing is to determine the first object hit by a query ray. Since this operation is performed millions of times by a ray tracer, it is advantageous to preprocess the objects into a data structure such that these basic operations, or *ray shooting queries*, can be performed very efficiently. An octree [127] is an example of such a data structure. Although octrees perform well in many practical situations, their worst-case behavior can be quite bad. People in computational geometry have developed data structures that are guaranteed to have a good query time in all situations. Initially, these data structures were mostly for planar versions of the ray shooting problem [1, 12, 41, 50, 83]. The introduction of Plücker coordinates to computational

geometry by Chazelle et al. [42] and the development of multi-level partition trees (see Section 3) opened the way to tackle the three-dimensional problem. For the general problem, where the objects are arbitrary triangles in three-dimensional space, it appears to be quite hard to get a good worst-case query time. The best known structures achieve $O(\log n)$ query time at the cost of $O(n^{4+\varepsilon})$ storage [2, 5, 21, 120]. This amount of storage makes these data structures useless in practical situations. If one wants to use only roughly linear space, then the fastest worst-case query time that has been achieved is $O(n^{3/4})$ [4]. Better solutions have been obtained for several special cases, such as axis-parallel polyhedra and polyhedral terrains [18, 42, 119]. There are reasons to believe that these results are close to to optimal. This is rather discouraging; apparently one cannot hope to design data structures for the ray shooting problem whose worst-case performance is acceptable in practical situations. This has led Mitchell et al. [103] to develop an octree-like data structure whose worst-case query time can be bad, but which performs good under a different, more practical complexity measure—see Section 7.

6.3 Discrepancy

For many computational problems in computer graphics a closed form solution is not available or extremely expensive to compute. In such cases statistical sampling methods can often be used to compute an approximate solution of the problem.

Consider as an example the rendering of a three-dimensional scene using ray tracing. The intensity of a pixel on the screen should correspond to the amount of light that is 'visible' in the finite area of that pixel. More precisely, one should integrate the intensity function over the pixel area. This integral is difficult to compute exactly, so in distributed ray tracing [58] the intensity of a pixel is approximated by sampling the intensity at a finite number of points.

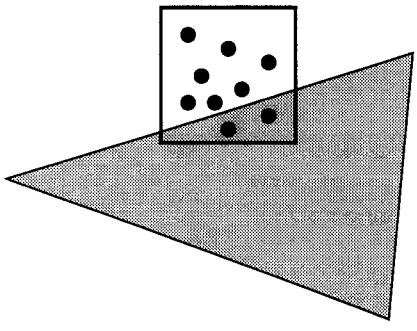

Figure 19: The polygon covers about 20% of the pixel, so we want a sample such that 20% of the points is within the polygon.

The quality of an approximation based on sampling depends on the distribution of the sample points. In distributed ray tracing, for example, shooting rays through points that are all very close to one corner of a pixel usually leads to a bad approximation. At first glance it may seem that uniform sample patterns—that is, grid-like patterns—will perform fine. Unfortunately, uniform patterns can lead to aliasing, and artifacts such as Moiré patterns can result. Nonuniform sampling methods are therefore to be preferred in most applications. It is clear that not all nonuniform sample patterns perform equally well, so the question arises of what constitutes a good pattern and how to find such a pattern. One way of doing this is to use variance-reducing techniques from statistical sampling theory [85, 88]. Other methods use techniques from image sampling theory [57, 63]. A third measure for the quality of sampling patterns is discrepancy [14].

Suppose that a pixel is partially covered by a polygon, as in Fig. 19. When we sample this pixel we would like the fraction of the pixel area covered by the polygon to be the same as the fraction of the sample points inside the polygon. The concept of discrepancy captures this type of quality measure.

Let $U = [0:1] \times [0:1]$ be the unit square in the plane, and let $S = \{p_1, \ldots, p_n\}$ be a set of n points inside the unit square. Let $\mathcal{F}$ be a family of objects in the plane. For an object $o \in \mathcal{F}$ we define $\mu(o)$, the *continuous measure* of o, to be the measure—that is, the area—of $o \cap U$. We define $\mu_S(o)$, the *discrete measure* of o with respect to S, as $|S \cap o|/|S|$. Here $|\cdot|$ is used to denote the cardinality of a set. Thus the continuous measure of o is the fraction of U that is covered by o, and the discrete measure of o is the fraction of the points in S that are contained in o. The discrepancy $\Delta_S(o)$ of o with respect to S is given by

$$\Delta_S(o) := |\mu(o) - \mu_S(o)|.$$

Finally, the discrepancy $\Delta_{\mathcal{F}}(S)$ of the point set S with respect to the family $\mathcal{F}$ is the maximum discrepancy of any object in $\mathcal{F}$:

$$\Delta_{\mathcal{F}}(S) := \sup_{o \in \mathcal{F}} \Delta_S(o).$$

Discrepancy was introduced to computer graphics by Shirley [134]. His experiments show that samples with low discrepancy indeed perform well in distributed ray tracing. The discrepancy that he used is the quadrant discrepancy, where the family $\mathcal{F}$ consists of all axis-parallel quadrants or, equivalently, of all axis-parallel rectangles having at least one corner coincident with a corner of U. Fig. 19 suggests that the set of all quadrants may not be the best choice for the set $\mathcal{F}$. Hence, Dobkin and Mitchell [62] propose to use half-plane discrepancy, where $\mathcal{F}$ consists of all possible half-planes.

A number of papers in computational geometry deal with the computation of the discrepancy of a given point set with respect to a some family $\mathcal{F}$. Dobkin and Mitchell [62] describe such an algorithm for the family of half-planes. Their algorithm computes the half-plane discrepancy of a set of n points in $O(n^2)$ time. They used their algorithm to compute the half-plane discrepancy of several popular sampling patterns (Zaremba, jittered, dart-throwing, and others). Chazelle [39] gives a fast approximation algorithm to compute the discrepancy. Dobkin and Eppstein [61] showed how to compute quadrant discrepancy in the plane in $O(n \log^2 n)$ time. They also obtained results on higher-dimensional versions of the problem. De Berg [16] shows how to compute the discrepancy with respect to the family of strips.

A possible heuristic to find low-discrepancy point sets is to iteratively add random points. Dobkin and Mitchell study the following variant of this process: at each iteration a small set of random points is generated, and the point whose addition results in the lowest discrepancy is added to the current set. This way they were able to construct sets with a very small discrepancy. One can compute the discrepancy of each of the sample sets in such an iterative procedure from scratch, but it would be nice if one could speed up the computation by using that the sample sets are almost identical. These considerations led Dobkin and Eppstein to study the dynamic version of the discrepancy problem. They show that the half-plane discrepancy can be updated in $O(n \log^2 n)$ time after an insertion or deletion of a point in S. This is considerably faster than the $O(n^2)$ time that is needed when the discrepancy has to be computed from scratch. De Berg [16] improved the update time to $O(n \log n)$.

7 Towards a practical theory

In the past years computational geometry has made tremendous progress. Almost all major problems in planar geometry have been solved (almost) optimally, and also in higher dimensions the tools have been developed to tackle most problems. There is now a solid theoretical foundation for the design and the analysis of geometric algorithms. The question is now: what next? Of course, there will always be theoretical open problems left, and it is important that people work on these problems. But we feel that it is more important to try and turn the ideas and techniques from computational geometry into solutions that are not only good in theory, but also in practice. In this section we discuss some of the ongoing (and hopefully future) research that may help to achieve this goal.

7.1 Realistic models

Many of the problems that are studied in computational geometry come from other areas, such as computer graphics. These problems are studied in an abstract setting. There are two dangers in the abstraction process. The first, well known danger is that not all aspects of the problem are modelled; thus the abstract version is a simplified version of the real problem. This sometimes makes that the solution to the abstract version of the problem is not applicable in practice. Most people are well aware of this problem, and they try to get rid of unrealistic assumptions as much as possible. The second danger is that the abstract version of the problem is *too* general, so that

useful properties of are lost. As a result, the algorithms have to be prepared for very complicated, hypothetical inputs. Actually, the situation is even worse: because one is usually interested in the *worst-case* performance, the algorithms are geared to these unrealistic inputs. Thus they become needlessly complicated and slow.

Consider as an example the construction of a BSP tree for a set of disjoint triangles in $\mathbb{R}^3$. In Section 4.1 we saw that it is always possible to construct a BSP of size $O(n^2)$. Moreover, there are sets of triangles for which any BSP must have quadratic size, so the $O(n^2)$ upper bound is the best we can hope for *in the worst case.* The lower bound example, however, is very artificial: it consists of n lines that are arranged in a very special, grid-like fashion. Hence, it does not say anything about what can be achieved in practice. Perhaps realistic scenes have some special property that makes it always possible to construct a BSP of linear size. Indeed, de Berg [19] has shown that under some weak conditions on the input it is always possible to construct a linear size BSP. In fact, the method works for general scenes, but the size of the resulting BSP depends on a certain parameter. If this parameter is constant—and it is claimed that this is the case in most practical applications—then the BSP has linear size.

Another example comes from ray tracing. We saw in Section 6 that ray shooting data structures have been developed that have roughly $O(n^{3/4})$ query time and use roughly linear storage. It is believed that this is theoretically close to optimal. But octrees, for instance, have been observed to behave much better in most practical situations. Apparently ray shooting queries are in practice not as difficult as they are in theory. Mitchell et al. [103] therefore defined a new complexity measure for ray shooting queries, the *simple cover complexity.* This measure reflects the difficulty of a given query: query rays that miss many small objects by just a small distance have a high simple cover complexity, whereas rays that stay for away from most objects have a low simple cover complexity. Mitchell et al. present a data structure whose query time depends on the simple cover complexity of the query ray, so that 'easy' queries are answered quickly. The expectation is that in a practical situation most rays have low simple cover complexity.

These two examples point in a promising research direction, where problems are studied under realistic assumptions, and algorithms analyzed using different, more practical complexity measures. This type of research can lead to algorithms and data structures that are *provably efficient* in *realistic situations.* Hence, the research is interesting from a theoretical as well as from a practical point of view. The crucial factor in this approach lies in the characterization of realistic inputs. Finding the right characterization, which is a key factor in understanding why certain approaches will or will not work in practice, is a topic for (experimental) research in itself. Of course, the characterization may depend on the application domain: architectural models have different properties than molecular models, for instance.

7.2 Experimental research

The final answer to the question if an algorithm is useful in practice can only be given after experimental research. Hence, algorithms should be implemented and tested to see how they behave in practical situations. Fortunately, many people in computational geometry feel the same way and more implementations are becoming available. This is reflected in the ACM Symposium on Computational Geometry, the most important conference in the area: since two years there is a video program with the symposium, and starting this year there will be short communications on experiences with computational geometry techniques in application areas. The change in attitude is also visible in the regular program: the number of papers in this year's symposium reporting on experimental research is much larger than five years ago. Examples of such papers are "Strategies for Polyhedral Surface Decomposition: An Experimental Study" [40], "How Good Are Convex Hull Algorithms" [8], and "A Comparison of Sequential Delaunay Triangulation Algorithms" [135].

One of the problems one faces when implementing an algorithm from computational geometry is that there is no good software library of geometric algorithms. There is a library for combinatorial computing, called LEDA [100], but this library concentrates on data structures and graph algorithms; its geometric part is not very extensive. This means that one has to implement all the basic routines—from convex hulls to low level routines such as the computation of the intersection

point of two line segments—oneself. Hopefully this situation will change in the very near future: a group of seven computational geometry sites in Europe (Berlin, Linz, Saarbrücken, Sophia-Antipolis, Tel-Aviv, Utrecht, and Zürich) has just started to develop such a library. The library, which will be called CGAL, will contain large number of geometric routines and data types. For many routines there will be an exact version as well as a version based on floating-point arithmetic.

8 Conclusions

Computational geometry has now reached an important point: many of the open problems have been (almost) solved, and standard techniques are available for a variety of problems. This report describes several of these results and techniques, emphasizing the results relevant to computer graphics. Of course there still remain a number of theoretical challenges. The challenges from practice, however, are at least as big, and more important: the time has come to turn the ideas and methods developed in computational geometry into solutions that are not only efficient in theory, but also in practice. For some problems this has already been done; I hope and expect that many others will follow.

References

[1] P. K. Agarwal. Ray shooting and other applications of spanning trees and low stabbing number. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 315–325, 1989.

[2] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. In *Proc. 24th Annu. ACM Sympos. Theory Comput.*, pages 517–526, 1992.

[3] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22(4):794–806, 1993.

[4] P. K. Agarwal and J. Matoušek. On range searching with semialgebraic sets. In *Proc. 17th Internat. Sympos. Math. Found. Comput. Sci.*, volume 629 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1992. Also to appear in *Discrete Comput. Geom.*

[5] P. K. Agarwal and M. Sharir. Applications of a new partitioning scheme. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes in Computer Science*, pages 379–391. Springer-Verlag, 1991.

[6] P. K. Agarwal and M. Sharir. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, Cambridge, UK, 1994. In press.

[7] F. d'Amore and P.G. Franciosa. On the optimal binary plane partition for sets of isothetic rectangles. *Inform. Process. Lett.*, 44:255–259, 1992.

[8] D. Avis and D. Bremner. How Good Are Convex Hull Algorithms. To appear in *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995.

[9] F. Aurenhammer. Voronoi diagrams: a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23:345–405, 1991.

[10] C. Bajaj and T. K. Dey. Convex decomposition of polyhedra and robustness. *SIAM J. Comput.*, 21:339–364, 1992.

[11] I. Balaban. An optimal algorithm for finding segments intersects. To appear in *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995.

[12] R. Bar-Yehuda and S. Fogel. Good splitters with applications to ray shooting. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 81–84, 1990.

[13] B. Barber and M. Hirsch. A robust algorithm for point in polyhedron. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 479–484, Waterloo, Canada, 1993.

[14] J. Beck and W. Chen. *Irregularities of distribution.* Cambridge University Press, 1987.

[15] M. de Berg. Dynamic output-sensitive hidden surface removal for c-oriented polyhedra. *Comput. Geom. Theory Appl.*, 2(3):119–140, 1992.

[16] M. de Berg. Computing half-plane and strip discrepancy of planar point sets. In *Proc. 3rd Internat. Conf. on Comput. Graphics and Visualization Techniques*, pages 294–303, 1993.

[17] M. de Berg. Generalized hidden surface removal. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 1–10, 1993.

[18] M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*, volume 703 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1993.

[19] M. de Berg. *Linear size binary space partitions for fat objects.* Manuscript, 1995.

[20] M. de Berg, M. de Groot, and M. Overmars. New results on binary space partitions in the plane. In *Proc. 4th Scand. Workshop Algorithm Theory*, volume 824 of *Lecture Notes in Computer Science*, 1994.

[21] M. de Berg, D. Halperin, M. Overmers,, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. *Algorithmica*, 12:30–53, 1994.

[22] M. de Berg and M. Overmars. Hidden surface removal for c-oriented polyhedra. *Comput. Geom. Theory Appl.*, 1(5):247–268, 1992.

[23] M. de Berg and M. H. Overmars. Hidden surface removal for axis-parallel polyhedra. In *Proc. 31st Annu. IEEE Sympos. Found. Comput. Sci.*, pages 252–261, 1990.

[24] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry by example.* In preparation.

[25] M. Bern. Hidden surface removal for rectangles. *J. Comput. Syst. Sci.*, 40:49–59, 1989.

[26] M. Bern, D. Dobkin, D. Eppstein, and R. Grossman. Visibility with a moving point of view. *Algorithmica*, 11:360–378, 1994.

[27] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 23–90. World Scientific, Singapore, 1992.

[28] M. Bern and D. Eppstein. Polynomial-size nonobtuse triangulation of polygons. *Internat. J. Comput. Geom. Appl.*, 2(3):241–255, 1992.

[29] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. In *Proc. 31st Annu. IEEE Sympos. Found. Comput. Sci.*, pages 231–241, 1990.

[30] M. Bern, S. Mitchell, and J. Ruppert. Linear-size nonobtuse triangulation of polygons. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 221–230, 1994.

[31] J.-D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. Applications of random sampling to on-line algorithms in computational geometry. *Discrete Comput. Geom.*, 8:51–71, 1992.

[32] J.-D. Boissonnat and M. Yvinec. Géométrie Algorithmique. Ediscience International, Paris. 1995.

[33] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms (SODA '93)*, pages 16–23, 1993.

[34] J. Canny, B. R. Donald, and E. K. Ressler. A rational rotation method for robust geometric algorithms. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 251–260, 1992.

[35] B. Chazelle. Lower bounds on the complexity of polytope range searching. *J. Amer. Math. Soc.*, 2:637–666, 1989.

[36] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6:485–524, 1991.

[37] B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9(2):145–158, 1993.

[38] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete Comput. Geom.*, 10:377–409, 1993.

[39] B. Chazelle. Geometric discrepancy revisited. In *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS 93)*, pages 392–399, 1993.

[40] B. Chazelle, D. Dobkin, N. Shouraboura, and A. Tal. Strategies for Polyhedral Surface Decomposition: An Experimental Study. To appear in *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995.

[41] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. In *Proc. 18th Internat. Colloq. Automata Lang. Program.*, volume 510 of *Lecture Notes in Computer Science*, pages 661–673. Springer-Verlag, 1991.

[42] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. Lines in space: combinatorics, algorithms, and applications. In *Proc. 21st Annu. ACM Sympos. Theory Comput.*, pages 382–393, 1989.

[43] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3):229–249, 1990.

[44] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[45] B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986.

[46] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 217–225, 1983.

[47] B. Chazelle and B. Rosenberg. Lower bounds on the complexity of simplex range reporting on a pointer machine. In *Proc. 19th International Colloquium on Automata, Languages, and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 439–449. Springer-Verlag, 1992. Also to appear in *Comput. Geom. Theory Appl.*

[48] B. Chazelle, M. Sharir, and E. Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. *Algorithmica*, 8:407–429, 1992.

[49] S.W. Cheng. *Dynamic hidden line elimination.* Manuscript, 1991.

[50] S. W. Cheng and R. Janardan. Algorithms for ray-shooting and intersection searching. *J. Algorithms*, 13:670–692, 1992.

[51] L. P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 274–280, 1993.

[52] N. Chin and S. Feiner. Near real time shadow generation using bsp trees. In *Proc. SIGGRAPH'89*, pages 99–106, 1989.

[53] K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987.

[54] K. L. Clarkson. A Las Vegas algorithm for linear programming when the dimension is small. In *Proc. 29th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 452–456, 1988.

[55] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[56] K. Clarkson, R. E. Tarjan, and C. J. Van Wyk. A fast Las Vegas algorithm for triangulating a simple polygon. *Discrete Comput. Geom.*, 4:423–432, 1989.

[57] R.L. Cook, Stochastic sampling in computer graphics. *ACM Trans. Graphics*, 5:51–72, 1986.

[58] R.L. Cook, T. Porter and L. Carpenter. Distributed ray tracing. *Computer Graphics*, 18:137–145, 1984.

[59] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[60] T. K. Dey, K. Sugihara, and C. L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Comput. Aided Geom. Design*, 9:457–470, 1992.

[61] D. Dobkin and D. Eppstein. Computing the discrepancy. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 47–52, 1993.

[62] D. Dobkin and D. Mitchell, Random-edge discrepancy of supersampling patterns. In *Graphics Interface '93*, York, Ontario, May, 1993.

[63] M.A.Z. Dippé and E.H. Wold. Antialiasing through stochastic sampling. *Computer Graphics*, 19:69–78, 1985.

[64] S. Dorward. A survey of object-space hidden surface removal. To appear in *Internat. J. Comput. Geom. Appl.*

[65] M. E. Dyer. Linear time algorithms for two- and three-variable linear programs. *SIAM J. Comput.*, 13:31–45, 1984.

[66] M. E. Dyer. On a multidimensional search technique and its application to the Euclidean one-center problem. *SIAM J. Comput.*, 15:725–738, 1986.

[67] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[68] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9:66–104, 1990.

[69] I. Emiris and J. Canny. A general approach to removing degeneracies. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 405–413, 1991.

[70] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 74–82, 1992.

[71] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations and Voronoi diagrams. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 83–92, 1992.

[72] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 334–341, 1991.

[73] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, 1993.

[74] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980.

[75] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. *Inform. Process. Lett.*, 7:175–179, 1978.

[76] M. T. Goodrich. A polygonal approach to hidden-line elimination. Report TR 87-18, Dept. Comput. Sci., Johns Hopkins Univ., Baltimore, MD, 1987.

[77] M. T. Goodrich, M. J. Atallah, and M. H. Overmars. Output-sensitive methods for rectilinear hidden surface removal. *Inform. Comput.*, 107:1–24, 1993.

[78] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[79] L. J. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 208–217, 1989.

[80] R. H. Güting and T. Ottmann. New algorithms for special cases of the hidden line elimination problem. *Comput. Vision Graph. Image Process.*, 40:188–204, 1987.

[81] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete Comput. Geom.*, 2:127–151, 1987.

[82] D. Halperin and M.H. Overmars. Spheres, molecules, and hidden surface removal. computations. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 113–122, 1994.

[83] J. Hershberger and S. Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 54–63, 1993.

[84] H. Inagaki and K. Sugihara. Numerically robust algorithm for contructing constrained Delaunay triangulation. In *Proc. 6th Canad. Conf. Comput. Geom.*, pages 171–176, 1994.

[85] J.T. Kajiya. The rendering equation. *Computer Graphics*, 20:143–150, 1986.

[86] M. J. Katz, M. Overmars, and M. Sharir. Efficient output sensitive hidden surface removal for objects with small union size. *Comput. Geom. Theory Appl.*, 2:223–234, 1992.

[87] M. J. van Kreveld. *New results on data structures in computational geometry.* Ph.D. dissertation, Dept. Comput. Sci., Utrecht Univ., Utrecht, Netherlands, 1992.

[88] M. Lee, R.A. Redner and S.P. Uselton. Statistically optimized sampling for distributed ray tracing. *Computer Graphics*, 19:61–67, 1985.

[89] H.-P. Lenhof and M. Smid. Maintaining the visibility map of spheres while moving the viewpoint on a circle at infinity. In *Proc. 3rd Scand. Workshop Algorithm Theory*, volume 621 of *Lecture Notes in Computer Science*, pages 388–398. Springer-Verlag, 1992.

[90] J. Matoušek. Approximations and optimal geometric divide-and-conquer. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 505–511, 1991. Also to appear in *J. Comput. Syst. Sci.*

[91] J. Matoušek. Cutting hyperplane arrangements. *Discrete Comput. Geom.*, 6:385–406, 1991.

[92] J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.

[93] J. Matoušek. Range searching with efficient hierarchical cuttings. *Discrete Comput. Geom.*, 10(2):157–182, 1993.

[94] J. Matoušek. Geometric range searching. *ACM Comput. Surveys* 26:421–461, 1994.

[95] J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 1–8, 1992.

[96] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Trans. Graph.*, 6:19–28, 1987.

[97] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. In *Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 329–338, 1982.

[98] N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. ACM*, 31:114–127, 1984.

[99] K. Mehlhorn. *Data structures and algorithms 3: Multi-dimensional searching and computational geometry.* Springer-Verlag, Heidelberg, Germany, 1984.

[100] K. Mehlhorn and S.Näher. LEDA: A platform for combinatorial and geometric computing. *Comm. ACM*, 38:96–102, 1995.

[101] E. A. Melissaratos and D. L. Souvaine. Coping with inconsistencies: a new approach to produce quality triangulations of polygonal domains with holes. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 202–211, 1992.

[102] V. Milenkovic. Robust construction of the Voronoi diagram of a polyhedron. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 473–478, Waterloo, Canada, 1993.

[103] J. S. B. Mitchell, D. M. Mount, and S. Suri. Query-sensitive ray shooting. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 359–368, 1994.

[104] K. Mulmuley. A fast planar partition algorithm, I. In *Proc. 29th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 580–589, 1988.

[105] K. Mulmuley. An efficient algorithm for hidden surface removal. *Comput. Graph.*, 23(3):379–388, 1989.

[106] K. Mulmuley. A fast planar partition algorithm, II. *J. ACM*, 38:74–103, 1991.

[107] K. Mulmuley. Hidden surface removal with respect to a moving point. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 512–522, 1991.

[108] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, New York, 1993.

[109] B. Naylor, J. A. Amatodes, and W. Thibault. Merging BSP trees yields polyhedral set operations. *Comput. Graph.*, 24(4):115–124, August 1990.

[110] M.E. Newell, R.G. Newell, and T.G. Sancha. A solution to the hidden surface removal problem. In *Proc. ACM National Conference*, pages 443–450, 1972.

[111] O. Nurmi. A fast line-sweep algorithm for hidden line elimination. *BIT*, 25:466–472, 1985.

[112] Atsuyuki Okabe, Barry Boots, and Kokichki Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley & Sons, Chichester, England, 1992.

[113] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994. ISBN 0-521-44592-2/Pb $24.95, ISBN 0-521-44034-3/Hc $49.95. Cambridge University Press 40 West 20th Street New York, NY 10011-4211 1-800-872-7423 346+xi pages, 228 exercises, 200 figures, 219 references.

[114] M.H. Overmars. *The design of dynamic data structures*, volume 156 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1983.

[115] M. Overmars and M. Sharir. Output-sensitive hidden surface removal. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 598–603, 1989.

[116] J. Pach and P. K. Agarwal. *Combinatorial Geometry.* J. Wiley & Sons, 1994. To appear. Preliminary version: DIMACS Tech. Report 91–51.

[117] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[118] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *J. Algorithms*, 13:99–113, 1992.

[119] M. Pellegrini. Ray-shooting and isotopy classes of lines in 3-dimensional space. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes in Computer Science*, pages 20–31, 1991.

[120] M. Pellegrini. Ray shooting on triangles in 3-space. *Algorithmica*, 9:471–494, 1993.

[121] M. Pellegrini. Repetitive hidden surface removal for polyhedral scenes. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes in Computer Science*, pages 541–552, 1993.

[122] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, New York, NY, 1985.

[123] F. P. Preparata and J. S. Vitter. A simplified technique for hidden-line elimination in terrains. *Internat. J. Comput. Geom. Appl.*, 3:167–181, 1993.

[124] J. H. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithms and its parallelization. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 193–200, 1988.

[125] J. Ruppert and R. Seidel. On the difficulty of triangulating three-dimensional non-convex polyhedra. *Discrete Comput. Geom.*, 7:227–253, 1992.

[126] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, 1990.

[127] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[128] A. Schmitt. On the time and space complexity of certain exact hidden line algorithms. Report 24/81, Fakultät Inform., Univ. Karlsruhe, Karlsruhe, West Germany, 1981.

[129] O. Schwarzkopf. *Dynamic Maintenance of Convex Polytopes and Related Structures.* Ph.D. thesis, Fachbereich Mathematik, Freie Universität Berlin, Berlin, Germany, June 1992.

[130] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1:51–64, 1991.

[131] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434, 1991.

[132] M. I. Shamos and D. Hoey. Closest-point problems. In *Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 151–162, 1975.

[133] M. Sharir and E. Welzl. A combinatorial bound for linear programming and related problems. In *Proc. 9th Sympos. Theoret. Aspects Comput. Sci.*, volume 577 of *Lecture Notes in Computer Science*, pages 569–579. Springer-Verlag, 1992.

[134] P. Shirley. Discrepancy as a quality measure for sample distributions. In F. H. Post and W. Barth, editors, *Proc. Eurographics'91*, pages 183–194, Vienna, Austria, September 1991. Elsevier Science.

[135] P. Su and R.L. Drysdale. A Comparison of Sequential Delaunay Triangulation Algorithms. To appear in *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995.

[136] I. E. Sutherland, R. F. Sproull, and R. A. Shumacker. A characterization of ten hidden surface algorithms. *ACM Comput. Surv.*, 6:1–55, 1974.

[137] R. E. Tarjan and C. J. Van Wyk. An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM J. Comput.*, 17:143–178, 1988. Erratum in 17 (1988), 106.

[138] M. Teillaud. *Towards dynamic randomized algorithms in computational geometry*, volume 758 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.

[139] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. *Comput. Graph.*, 25(4):61–69, July 1991.

[140] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *Proc. SIGGRAPH'87*, pages 153–162, 1987.

[141] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes in Computer Science*, pages 359–370. Springer-Verlag, 1991.

[142] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *J. Comput. Syst. Sci.*, 40:2–18, 1990.

[143] C. K. Yap. Towards exact geometric computation. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 405–419, 1993.

Recent technical reports from the Department of Computer Science, Utrecht University

Requests for Technical Reports can be directed to

Librarian
Department of Computer Science
Utrecht University
P.O. Box 80.089
Utrecht University
the Netherlands
Or by e-mail: guus@cs.ruu.nl

Many technical reports are also available via ftp (**ftp ftp.cs.ruu.nl**, login as **anonymous** or **ftp**, directory: **/pub/RUU/CS/techreps**).

The archive of technical reports is also accessible via the World Wide Web, URL-address:

`http://www.cs.ruu.nl/res/publication/TechRep.html`

[UU-CS-1994-01] J. Jeuring and D. Swierstra. Bottom-up grammar analysis - a functional formulation.

[UU-CS-1994-02] M. de Berg and M. van Kreveld. Trekking in the alps without freezing or getting tired.

[UU-CS-1994-03] M.H. Overmars and P. Švestka. A probabilistic learning approach to motion planning.

[UU-CS-1994-04] G. Hutton and E. Meijer. Back to basics: Deriving presentations changers without relations.

[UU-CS-1994-05] E. Meijer. More advice on proving a compiler correct: Improve a correct compiler.

[UU-CS-1994-06] E. Meijer. Hazard algebra for asynchronous circuits.

[UU-CS-1994-07] J.J.-Ch. Meyer and W. van der Hoek. A modal contrastive logic: The logic of 'but'.

[UU-CS-1994-08] W. van der Hoek B. van Linder and J.-J. Ch. Meyer. Tests as epistimic updates pursuit of knowledge.

[UU-CS-1994-09] M. de Berg, L. Guibas, D. Halperin, M. Overmars, O. Schwarzkopf, M. Sharir, and M. Tillaud. Reaching a goal with directional uncertainty.

[UU-CS-1994-10] P.K. Agarwal and M. van Kreveld. Connected component and simple polygon intersection searching.

[UU-CS-1994-11] H.L. Bodlaender and J. Engelfriet. Domino treewidth.

[UU-CS-1994-12] M. de Berg, Katrin Dobrindt, and O. Schwarzkopf. On lazy randomized incremental construction.

[UU-CS-1994-13] L.C. van der Gaag and C. de Koning. Reason maintenance for production systems.

[UU-CS-1994-14] H.L. Bodlaender and M.R. Fellows. W[2]-hardness of precedence constrained κ-processor scheduling.

[UU-CS-1994-15] T.W.C. Huibers and P.D. Bruza. Situations, a general framework for studying information retrieval.

[UU-CS-1994-16] R.R. Bouckaert. A stratified simulation scheme for inference in bayesian belief networks.

[UU-CS-1994-17] P. Bose, L. Guibas, A. Lubiw, M. Overmars, D. Souvaine, and J. Urrutia. The floodlight problem.

[UU-CS-1994-18] A.S. Rao and K.Y. Goldberg. Computing grasp functions.

[UU-CS-1994-19] I.S.W.B. Prasetya. Mechanization of substitution rule and compostionality of unity in hol.

[UU-CS-1994-20] T. Arts and H. Zantema. Termination of logic programs via labelled term rewrite systems.

[UU-CS-1994-21] M. Kreveld. Efficient methods for isoline extraction from a digital elevation model based on triangulated irregular networks.

[UU-CS-1994-22] R.R. Bouckaert. Idags: a perfect map for any distribution.

[UU-CS-1994-23] L. v.d. Gaag and M. Wessels. Efficient multiple-disorder diagnosis by strategic focusing.

[UU-CS-1994-24] A.S. Rao and K.Y. Goldberg. Friction and part curvature in parallel-jaw grasping.

[UU-CS-1994-25] B. Asberg, G. Blanco, P. Bose, J. Garcia-Lopez, M. Overmars, G. Toussaint, G. Wilfong, and B. Zhu. Feasibility of design in stereolithography.

[UU-CS-1994-26] P. Bose, D. Bremmer, and M. van Kreveld. Determining the castability of simple polyhedra.

[UU-CS-1994-27] R.R. Bouckaert. Probabilistic network construction using the minimum description length principle.

[UU-CS-1994-28] M.C.F. Ferreira and H. Zantema. Syntactical analysis of total termination.

[UU-CS-1994-29] M. de Berg, L.J. Guibas, and D. Halperin. Vertical decompositions for triangles in 3-space.

[UU-CS-1994-30] M.H. Overmars and A.F. van der Stappen. Range searching and point location among fat objects.

[UU-CS-1994-31] V. Ferrucci, M. Overmars, A. Rao, and J. Vleugels. Hunting voronoi vertices.

[UU-CS-1994-32] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. Probalistic roadmaps for path planning in high-dimensional configuration spaces.

[UU-CS-1994-33] P. Svestka and M. Overmars. Motion planning for car-like robots using a probalistic learning approach.

[UU-CS-1994-34] M. de Berg. Computing half-plane and strip discrepancy of planar point sets.

[UU-CS-1994-35] R.R. Bouckaert. Properties of measures for bayesian belief network learning.

[UU-CS-1994-36] D. Halperin and M.H. Overmars. Spheres, molecules, and hidden surface removal.

[UU-CS-1994-37] T.W.C. Huibers, B. van Linder, and P.D. Bruza. Een theorie voor het bestuderen van information retrieval modellen.

[UU-CS-1994-38] J.-J. Ch. Meyer, F.P.M. Dignum, and R.J. Wieringa. The paradoxes of deontic logic revisited: A computer science perspective (or: Should computer scientists be bothered by the concerns of philosophers?).

[UU-CS-1994-39] P.K. Agarwal, J. Matousek, and O. Schwarzkopf. Computing many faces in arrangements of lines and segments.

[UU-CS-1994-40] P.K. Agarwal, O. Schwarzkopf, and M. Sharir. Computing many faces in arrangements of lines and segments.

[UU-CS-1994-41] M. van Kreveld, J. Snoeyink, and S. Whitesides. Folding rulers inside triangles.

[UU-CS-1994-42] L.C. van der Gaag. Evidence absorption - experiments on different classes of randomly generated belief networks.

[UU-CS-1994-43] H.R. Walters and H. Zantema. Rewrite systems for integer arithmetic.

[UU-CS-1994-44] H. Zantema and A. Geser. A complete characterization of termination of $0^p1^q \rightarrow 1^r0^s$.

[UU-CS-1994-45] F.S. de Boer and M. van Hulst. A proof system for asynchronously communicating deterministic processes.

[UU-CS-1994-46] M.C.F. Ferreira and H. Zantema. Well-foundedness of term orderings.

[UU-CS-1994-47] M.C.F. Ferreira and H. Zantema. Dummy elimination: Making termination easier.

[UU-CS-1994-48] B. van Linder, W. van der Hoek, and J.-J. Ch. Meyer. The dynamics of default reasoning.

[UU-CS-1994-49] A. Rao, D. Kriegman, and K. Goldberg. Complete algorithms for feeding polyhedral parts using pivot grasps.

[UU-CS-1994-50] G. Florijn. Modelling office processes with functional parsers.

[UU-CS-1994-51] M. de Berg, M. de Groot, and M. Overmars. New results on binary space partitions in the plane.

[UU-CS-1994-52] C. Soares. Evolutionary computation for the job-shop scheduling problem.

[UU-CS-1994-53] B. van Linder, W. van der Hoek, and J.-J. Ch. Meyer. Actions that make you change your mind: Belief revision in an agent-oriented setting.

[UU-CS-1994-54] W. van der Hoek, J.-J. Ch. Meyer, and J. Treur. Temporalizing epistemic default logic.

[UU-CS-1994-55] H. Zantema. Total termination of term rewriting is undecidable.

[UU-CS-1994-56] C. Witteveen and W. van der Hoek. Revision by communication: Program by consulting weaker semantics.

[UU-CS-1995-01] H.L. Bodlaender, R.G. Downey, M.R. Fellows, and H.T. Wareham. The parameterized complexity of sequence alignment and consensus.

[UU-CS-1995-02] H.L. Bodlaender and D.M. Thilikos. Treewidth and small seperators for graphs with small chordality.

[UU-CS-1995-03] H.L. Bodlaender, J.S. Deogun, K. Jansen, T. Kloks, D. Kratsch, H. Müller, and Zs. Tuza. Rankings of graphs.

[UU-CS-1995-04] T. Biedl and G. Kant. A better heuristic for orthogonal graph drawings.

[UU-CS-1995-05] J. van Leeuwen and R.B. Tan. Compact routing methods: A survey.

[UU-CS-1995-06] P.K. Agarwal, M. de Berg, J. Matoušek, and O. Schwarzkopf. Constructing levels in arrangements and higher order voronoi diagrams.

[UU-CS-1995-07] I.S.W.B. Prasetya and S.D. Swierstra. Formal design of self-stabilizing programs.

[UU-CS-1995-08] B. van Linder, W. van der Hoek, and J.-J. Ch. Meyer. Seeing is believing and so are hearing and jumping.

[UU-CS-1995-09] P.D. Bruza and T.W.C. Huibers. How nonmonotonic is aboutness?

[UU-CS-1995-10] W. Fokkink and H. Zantema. A complete equational axiomatization for bpa$\delta\epsilon$ with prefix iteration.

[UU-CS-1995-11] N.B. Peek and L.C. van der Gaag. A case-based filter for diagostic belief networks.

[UU-CS-1995-12] M. de Berg and K.T.G. Dobrindt. On levels of detail in terrains.

[UU-CS-1995-13] J. van Leeuwen, N. Santoro, J. Urrutia, and S. Zaks. Guessing games, binomial sum trees and distributed computations in synchronous networks.

[UU-CS-1995-14] J. Vleugels and M. Overmars. Approximating generalized voronoi diagrams in any dimension.

[UU-CS-1995-15] H.L. Bodlaender and B. de Fluiter. Intervalizing k-colored graphs.

[UU-CS-1995-16] M. Flammini, J. van Leeuwen, and A. Marchetti-Spaccamela. The complexity of interval routing on random graphs.

[UU-CS-1995-17] T. Arts and H. Zantema. Termination of constructor systems using semantic unification.