TEACHING COMPUTATIONAL GEOMETRY

Mark H. Overmars
Department of Computer Science
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
THE NETHERLANDS
e-mail: markov@cs.ruu.nl

ABSTRACT

This paper describes a possible approach to teaching computational geometry to students. It gives a brief introduction in computational geometry, followed by three possible ways of treating the area: mathematically, algorithmically and application-oriented. Next a course outline is described, introducing computational geometry from an application-oriented point of view. Finally, some remarks are made about implementation issues.

COMPUTATIONAL GEOMETRY

In general terms, computational geometry concerns itself with the study of algorithms for geometric problems involving points, lines, polygons, etc., in the plane and in higher-dimensional space. As an example, consider the following problem: Given a set of n line segments in the plane, does there exist a pair that intersects. An easy solution would test all pairs, thus requiring $O(n^2)$ time in the worst case. A much better approach exists though, requiring only $O(n \log n)$ time (see [Bentley-Ottmann79]).

Since its introduction by Shamos in the mid seventies (see e.g. [Shamos78]) the area of computational geometry has rapidly gained popularity. Nowadays there are special conferences devoted to the topic (e.g. the ACM Symp. on Computational Geometry that in 1994 will have its tenth edition) and there are a number of journals devoted to the area (most notable the journal on "Discrete and Computational Geometry", the journal "Computational Geometry: Theory and Applications" and "International Journal on Computational Geometry and Applications"). The reason for the popularity lies both in the beauty of the problems and solutions and in the fact that computational geometry problems show up in many applications.

Typical application areas where one encounters many geometric problems are computer graphics, geographic information systems (GIS), robotics (in particular in motion planning), computer aided design and manufacturing (CAD and CAM), etc. Let me briefly describe some of the problems that occur.

A prime geometric problem in computer graphics is the hidden surface removal problem. Given a scene consisting of a number of objects we want to know which parts of objects are visible from a particular viewpoint. See e.g. figure 1. The normal approach to solve this problem is to compute for each pixel on the computer screen the object visible at that pixel, often using a hardware Z-buffer. Such a solution is often called an image-space solution because it computes an image (i.e., a bitmap). An alternative approach would be to compute a combinatorial description of the visible parts of the objects, normally called the visibility map. Such an approach is called object-space because it works independent of the display device (screen, laser printer, etc.). Object-space algorithms for graphics problems have been studied extensively in computational geometry. See e.g. [deBerg93] for some very recent results.

Another important application area of computational geometry is "Geographic Information Systems (GIS)". Here people want to store different types of maps (road maps, height maps, etc.) and ask particular questions about them. Examples are: Given a small (rectangular) region, compute the part of the map inside the region, e.g., to display on a screen. Such queries are often referred to as range queries or windowing queries. See figure 2 for an example. Or the map overlay problem: Given two maps (e.g. of roads and of railways) compute all intersections among them. Or, given two points on the map, compute the shortest route between them, taking into account the various properties of the terrain encountered. All these problems are

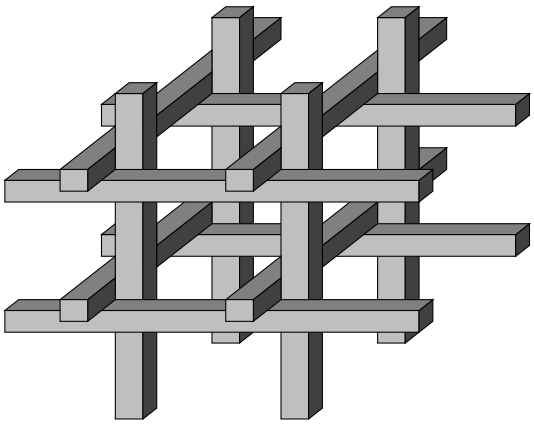

Figure 1: Hidden surface removal.

geometric of nature and computational geometry provides useful solutions to them.

In robotics a crucial geometric problem is the motion planning problem: Given a robot R, a set of obstacles S, and a start and goal position for R, determine a path for the robot from start to goal that does not collide with any obstacle. See figure 3 for an example. This is an extremely difficult problem. One would also like to stay as far away as possible from the obstacles. Computational geometry has provided theory about e.g. arrangements and Voronoi diagrams that can be used for solving the motion planning problem.

For these and many other problems arising in different applications, computational geometry offers a wide spectrum of possible algorithms, data structures and geometric paradigms. Knowledge of these will be very useful for people working in these fields. Hence, within a computer science curriculum a course in computational geometry is important.

AUDIENCES

Computational geometry courses can be taught to different kinds of audiences: students in (combinatorial) mathematics, students in algorithms and students in application areas such as graphics, robotics, and GIS. In each situation the material covered should be quite different. I will briefly review

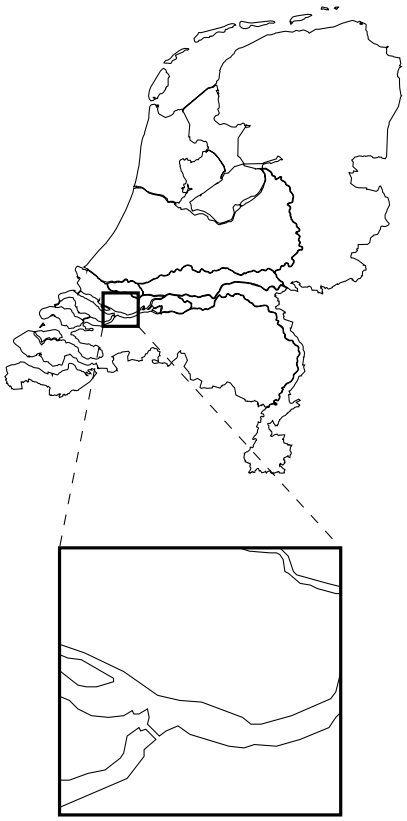

Figure 2: Windowing in the map of the Netherlands.

these three different approaches now.

Combinatorial Geometry

Computational geometry and combinatorial geometry are closely related. Combinatorial geometry typically provides bounds on the combinatorial complexity of certain geometric structures. These bounds are often crucial in the complexity analysis of geometric algorithms. For example, given a set of n lines in the plane and n points, it has been proven that the number of point-line incidences is bounded by $O(n^{4/3})$. As a result a large number of geometric algorithms run in something like $O(n^{4/3})$ time. Also,

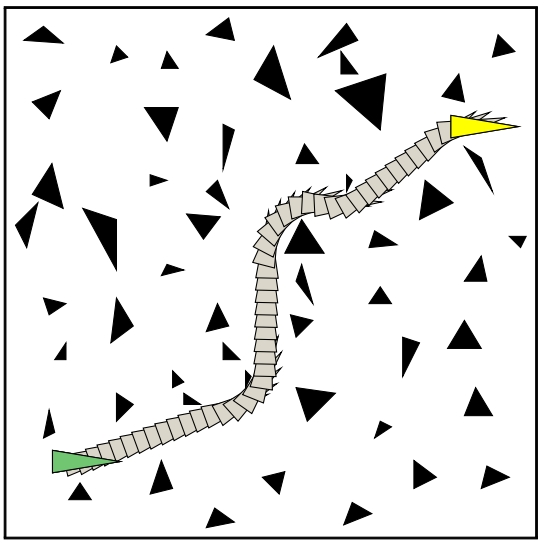

Figure 3: Motion planning in the plane.

combinatorial geometry provides properties that can be used in the design of geometric algorithms and data structures. For example, given a set of blue and red points in the plane, it has been show that there always exists a line that separates simultaneously the red and blue points in equal sized subsets. This result is crucial for certain planar partition trees.

On the other hand, computational geometry provides a rich collection of combinatorial problems that should fascinate mathematicians interested in combinatorics. Also, computational geometry has provided many constructive proofs for combinatorial theorems. Hence, a course on computational and combinatorial geometry would be a very interesting and useful course for students in mathematics. For such a course, the book of [Edelsbrunner85] can provide good material.

Algorithms

Computational geometry is a subfield of algorithms design. Hence, it is very natural to give a course in computational geometry from an algorithmic point of view. Computational geometry provides many nice algorithmic tools like scan-line algorithms, different types of divide-and-conquer, randomized

algorithms, and multi-level data structures. Also, the analysis of geometric algorithms requires many different techniques.

The oldest textbook on computational geometry [Preparata-Shamos85] is still one of the few available to treat the basic techniques and data structures in computational geometry from an algorithmic point of view. A second possibility is [Mehlhorn84]. Both books are a bit old-fashioned now and almost only treat deterministic algorithms. A very recent textbook [Mulmuley93] concentrates on new randomized solutions. None of the books is complete. All of them are also rather difficult and not well suited for e.g. an undergraduate course. Soon a new book [O'Rourke93] will appear that is more directed to an undergraduate audience.

Application-Oriented

As indicated in the previous section computational geometry is useful in many application areas, in particular computer graphics, robotics, and Geographic Information Systems. Hence, it would be good if students in these areas get some background in computational geometry. Unfortunately there are at the moment no good textbooks that introduce computational geometry from the application-oriented perspective. Most algorithms oriented books tend to give little information about the applicability of the results described. Also, they tend to delve too deep into the algorithmic and analysis issues of the field.

A few books do exist that are dedicated to particular aspects of computational geometry in relation to application fields. For example, the books [Samet93a], [Samet93b] give an extensive overview of the use of geometric data structures. [deBerg93] closely studies ray shooting and hidden surface removal from a computational geometry perspective. Finally [Latombe91] concentrates on the motion planning problem. None of these books would be suited for a general application-oriented course in computational geometry. We ourselves are at the moment working on such a general text-book, but it will probably take till 1995 before it is available. (A preliminary version might be available much sooner. Contact me if you are interested.)

A POSSIBLE COURSE

In this section I will give a possible outline for an application-oriented course in computational geometry. It is the basis for the course we plan to teach at our department (although we will treat some other topics as well). For

each of the topics indicated I will describe what applications can be used to introduce them. Of course the list below is definitely not the only possible choice. Computational geometry is such a wide area that there is no way to cover it completely in one course. But I think that the topics indicated below would form a good introduction (and necessary background) for application-oriented students.

Geometric Data Structures

Every course in computational geometry should treat a number of basic geometric data structures. In particular, students should get familiar with quad- or k-d trees, range trees and segment trees. Also, they should see the concept of multi-level data structures. These structures can be introduced using the windowing problem described in the introduction. An extensive overview of geometric data structures and their applications can be found in [Samet93a] and [Samet93b].

Line Segment Intersection

As described in the introduction, the line segment intersection problem asks for all intersections in a set of line segments (or whether there is an intersection). The algorithm by [Bentley-Ottmann79] should be described in any course. It is widely used. It also introduces the plane-sweep technique that is fundamental in computational geometry. A prime application is the computation of map overlays in GIS, but it also is the basic step in an object space hidden surface removal algorithm by [Schmitt81].

Simple Polygons

In a number of applications the sets of objects we have to deal with have some structure that we can exploit to improve our algorithms. In particular one often has to deal with (simple) polygons rather than arbitrary sets of points or line segments. Many geometric problems can be solved more efficient for simple polygons. For example, triangulation (see figure 4) can be solved inside a simple polygon in time $O(n)$ while it takes time $O(n \log n)$ in general (see [Chazelle91]). A typical area of application can be the so-called art-gallery problems. See [O'Rourke87] for an overview.

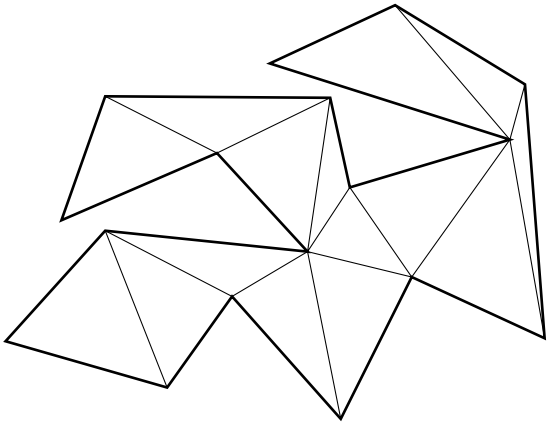

Figure 4: Triangulation of a simple polygon.

Point Location

Point location is an essential type of query in geometric settings. Given a subdivision of the plane (or space) in regions, it asks which region contains a particular query point. Many different solutions exist that store a (planar) subdivision in linear storage and solve point location queries in optimal time $O(\log n)$ (see e.g. [Preparata-Shamos85], chapter 2.2). Point location has many applications. One possible choice is the location of mouse clicks: Assume the user points with his mouse to a particular position in a displayed map, determine which region is selected.

Voronoi Diagrams

Voronoi diagrams are a prime structure in computational geometry. Given a set of points in the plane, the Voronoi diagram gives for each point p the region R_p for which p is the closest point. The regions together form the diagram. See figure 5 for an example. A Voronoi diagram of n points can be computed in $O(n \log n)$ time either using the classical divide-and-conquer algorithm described e.g. in [Preparata-Shamos85], chapter 5.5, or using the plane-sweep algorithm by [Fortune87]. Voronoi diagrams (and their generalizations) play a crucial role in many applications. An immediate example is facility location: What is the nearest, say, post office for a particular position. But the Voronoi diagram can also be used for motion planning be-

cause its edges describe the places with maximal clearance (i.e., the maximal distance to the obstacles).

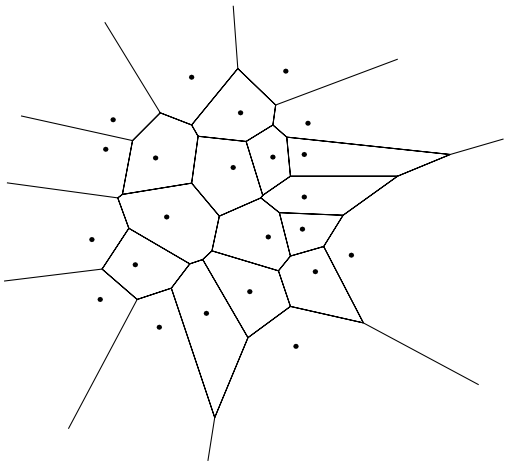

Figure 5: The Voronoi diagram.

Also the dual of the Voronoi diagram, called the Delauney triangulation, is an important structure. It is obtained by connecting each pair of points whose Voronoi regions share an edge. It is a triangulation of the set of points with some nice properties that make it very suited for mesh generation and interpolation purposes.

Linear Programming

Linear programming is a well-known algorithmic technique for solving many different types of problems. A number of geometric problems can be formulated as linear programming with only very few variables. Such problems can be solved in linear time (where the constant in the time bound depends on the number of variables) using the approach of [Megiddo83] (see also [Preparata-Shamos85], chapter 7.2.5). Recently a very nice $O(n)$ randomized expected time algorithm has been devised by [Welzl91]. This approach can also be use for slightly more general problem like the smallest enclosing sphere problem: Given a set of points in space, compute the smallest radius sphere that contains all of them.

Arrangements

An arrangement is the subdivision of the plane (space) formed by a collection of lines, polygons, etc. See e.g. figure 6. Arrangements play an important role in different types of problems. For example, the question whether a collision free path exists for a robot can be formulated as the question whether start and goal configuration lie in the same cell in an arrangement of hyper-surfaces in the space of all possible configuration. Each obstacle defines a collection of such hyper-surfaces. Hence, knowledge about the complexity of arrangements and algorithms for computing them is useful.

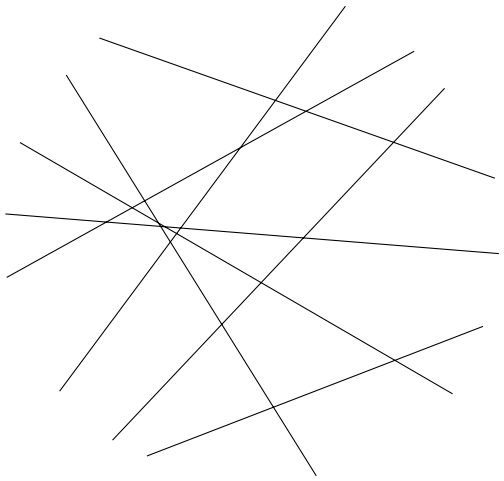

Figure 6: Arrangement of lines.

Also geometric transformations exist that map e.g. points into lines and vice versa. In this way, questions about sets of points can be transformed into questions about sets of lines. E.g., the question whether three points in a set are collinear transforms into the question whether three lines pass through a single point. This can be solved in time $O(n^2)$ by computing the arrangement of the lines using the technique of [Edelsbrunner-Guibas89].

Convex Hulls

The convex hull problem is one of the oldest studied in computational geometry. It asks for the smallest convex polygon containing a given set of

points (or other objects). See figure 7. Many different optimal $O(n \log n)$ algorithms are known for the problem (see e.g. [Preparata-Shamos85], chapter 3). An extension of convex hulls are so-called α-hulls by [Edelsbrunner et al.83] (see also [Edelsbrunner87]). These can be used to determine the shape of a set of points. This is very useful in various pattern recognition applications.

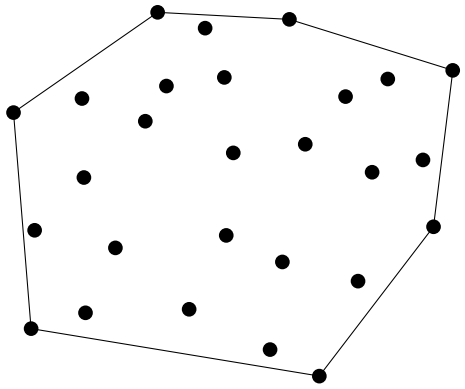

Figure 7: Convex hull.

PRACTICAL WORK

An application oriented course in computational geometry could well be accompanied by some practical work by the students, although it is not necessary for a good understanding of the material. But trying to implement geometric algorithms confronts the students with some problems that are not immediately apparent from the normally high-level descriptions of the algorithms. I will discuss some of these problems here.

Low-Level Geometric Routines

When implementing a geometric algorithm one normally needs lots of low-level routines to determine whether two line segments intersect, whether a point lies to the left or to the right of a line, etc. Writing such routines yourself is time consuming, bound to introduce errors and not very useful from an educational point of view (although it is often a good exercise to let students write at least one such routine to realize the problems involved).

Fortunately there are some libraries available that contain such primitive operations. One of the most extensive libraries is LEDA (see [Mehlhorn-Näher89] and [Näher93], available by ftp from `sbsvax.cs.uni-sb.de`) although the emphasis is more on data structures than on geometry. A second library is PlaGeo (see [Giezeman93], available by ftp from `ftp.cs.ruu.nl`) that only contains basic geometric routines. Both libraries are written in C++. Another possibility is to use systems like the Workbench for Computational Geometry developed at Carleton University (see e.g. [Knight et al.90]) or the XYZ GeoBench developed at the ETH Zürich (see [Nievergelt et al.91], [Schorn91]) which are not really libraries but environments for implementing geometric algorithms, both running on a Macintosh.

Special Cases

In many papers in computational geometry one finds an expression like: "We assume that all points lie in general position. The method can easily be adapted to deal with such degeneracies as well." Even though this is true from the theoretical point of view it is not true when implementing the algorithm. Often most of the resulting code is written for dealing with special cases. Unfortunately, in practice, special cases occur all the time. Hence, the code has to be able to deal with them correctly. Some general approaches exist for this problem, e.g the symbolic perturbation technique that, in a symbolic fashion, perturbs the objects such that they become in general position. See e.g. [Edelsbrunner-Mücke90]. Also, careful analysis of the problem can often turn most special cases into normal cases, thus reducing the amount of code required.

Robustness

Robustness is a difficult problem when implementing geometric algorithms. Most programming languages work with floats of bounded precision. This will introduce round-off errors. Not only will these errors sometimes result in wrong output (e.g. an intersection is reported that does not exist), but they also easily lead to inconsistencies that might crash the program. Of course one could solve this problem using exact arithmetic but the precision required is so high that it makes the computations extremely slow. When carefully coding the algorithms, problems with robustness can be reduced. A good analysis is required to determine which precision is required at particular places. See e.g. [Fortune-Van Wyk93] for a general treatment of this

problem.

Tuning

Algorithms described in books and papers are often designed in a form such that it is easy to prove that they are theoretically efficient. To make them also efficient in practice normally requires quite some work. Just implementing the algorithm the way it is described will often lead to disappointment. Careful implementation, perhaps changing the algorithm at certain points, can easily improve the running time with a factor of 10 or more.

Most algorithms have rather large constants in the time bounds. This makes them unsuited for small sets of objects. So whenever the input size is small a simpler brute-force technique should be used. This e.g. applies to divide-and-conquer algorithms that recursively solve smaller subproblems. Once the subproblems get small enough a different technique should be used, i.e., one should not continue to recurse until the subproblem has size 1. Also in geometric data structures that are defined recursively one should use a simpler structure once the stored sets get small.

Also care has to be taken in implementing the basic operations. For example, consider the following algorithm that computes the closest pair in a set of points: For every pair of points, compute the distance and compare it with the smallest distance stored so far. If it is larger we keep the current pair as closest, otherwise we replace it by the new pair. This algorithm will take $O(n^2)$ time. (An $O(n \log n)$ algorithm exists as well.) The running time of this algorithm will highly depend on the way we compute the distance between two points. Because we only compare distances one does not need to take the square roots. Moreover, in most cases a difference test in x-coordinates only will be enough to decide that the new distance is larger than the current minimal one. This will simplify the test most of the time. Hence, a careful analysis to decide which parts of the algorithm take most of the time is crucial.

Contrary to what some people claim, many geometric algorithms can be implemented without too much trouble and run really fast.

CONCLUSIONS

In this paper I have tried to give some information about computational geometry and the different ways it can be taught to different types of audiences. In particular I have concentrated on a possible outline of a course

directed towards more application-oriented students. Of course the different possibilities discussed are not the only ones. Probably each person who teaches a course in computational geometry will choose a different set of topics and will treat them in a different way. Also computational geometry is still evolving very quickly. New algorithms, approaches, problems and applications are discovered all the time. At the moment for example there is a strong tendency towards randomized algorithms for computational geometry (see e.g. the textbook of [Mulmuley93]) in contrast to the more traditional deterministic approaches. This will all have an impact on future courses.

Again, I would like to point out the importance of computational geometry in application areas like graphics, robotics, and geographic information systems. First of all, techniques and algorithms developed in computational geometry will be useful for people working in these areas. Secondly, computational geometry will give them a better insight in the theory underlying the different geometric questions arising. It will save people from reinventing the wheel or from working too long on impossible missions.

Computational geometry is definitely not only a field of research with beautiful combinatorial and algorithmic results but also a field with lots of practical impact. Possibly due to the lack of good textbooks it is unfortunately too unknown among practitioners. I hope this will change in the near future.

ACKNOWLEDGEMENTS

This work was partially supported by the ESPRIT III Basic Research Action No. 7141 (project ALCOM II) and by the Netherlands Organization for Scientific Research (N.W.O.).

REFERENCES

Bentley, J., Ottmann, T., 1979 Algorithms for reporting and counting geometric intersections, *IEEE Trans. Comput.*, C-28, 643–647.

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

de Berg, M. 1993 *Ray Shooting, Depth Orders and Hidden Surface Removal,* Springer-Verlag, Berlin, Germany.

Edelsbrunner, H. 1987 *Algorithms in Combinatorial Geometry,* Springer-Verlag, Berlin, Germany.

Edelsbrunner, H., Guibas, L., 1989 Topologically sweeping an arrangement, *J. Comput. Syst. Sci.*, 38, 165–194.

Edelsbrunner, H., Kirkpatrick, D., Seidel, R., 1983 On the shape of a set of points in the plane, *IEEE Trans. Inform. Theory*, IT-29, 551–559.

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

Fortune, S., 1987 A sweepline algorithm for Voronoi diagrams, *Algorithmica*, 2, 153–174.

Fortune, S., Van Wyk, C., 1993 Efficient exact arithmetic for computational geometry, *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, 163–172.

Giezeman, G-J., 1993, *PlaGeo: A Library for Planar Geometry,* Dept. of Computer Science, Utrecht University.

Knight, A., May, J., McAffer, M.,. Nguyen, T., Sack, J.-R., 1990 A computational geometry workbench, *Proc. 6th Annu. ACM Sympos. Comput. Geom.,* 370.

Latombe, J-C., 1991 *Robot Motion Planning,* Kluwer Academic Publishers, Boston, U.S.A.

Mehlhorn, K., 1984 *Multi-dimensional Searching and Computational Geometry,* Springer-Verlag, Berlin, Germany.

Mehlhorn, K., Näher, S., 1989 LEDA, a library of efficient data types and algorithms, *Proc. MFCS '89*, Springer-Verlag, Berlin, Germany, 187–202.

Mulmuley, K. 1993 *Computational Geometry, An Introduction Through Randomized Algorithms,* Prentice Hall, Englewood Cliffs, New Jersey, U.S.A.

Näher, S., 1993 *LEDA Manual,* Max Planck Institute für Informatik, Saarbrücken, Germany.

Nievergelt, J., Schorn, P., De Lorenzi, M., Ammann, C., Brüngger, A., 1991 XYZ: A Project in Experimental Geometric Computation, *Proc. Computational Geometry - Methods, Algorithms and Applications,* Springer-Verlag, Berlin, Germany, 171–186.

O'Rourke, J., 1987 *Art Gallery Theorems and Algorithms,* Oxford University Press, New York, U.S.A.

O'Rourke, J., 1993 *Computational Geometry in C,* Cambridge University Press, Cambridge, England.

Preparata, F., Shamos, M., 1985 *Computational Geometry, An Introduction,* Springer-Verlag, New York, U.S.A.

Samet, H. 1990 *The Design and Analysis of Spatial Data Structures,* Addison Wesley, Reading, Massachusetts, U.S.A.

Samet, H. 1990 *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS,* Addison Wesley, Reading, Massachusetts, U.S.A.

Schmitt, A., 1981 Time and space bounds for hidden line and hidden surface algorithms, *Proc. Eurographics 81,* North-Holland, Amsterdam, Netherlands, 43–56.

Schorn, P., 1991 The XYZ GeoBench: A Programming Environment for Geometric Algorithms, *Proc. Computational Geometry - Methods, Algorithms and Applications,* Springer-Verlag, Berlin, Germany, 187–202.

Shamos, M., 1978 *Computational Geometry,* PhD thesis, Dept. of Computer Science, Yale University.

Welzl, E., 1991 Smallest enclosing disks (balls and ellipsoids), in: H. Maurer (Ed.): *New Results and New Trends in Computer Science,* Springer-Verlag, Berlin, Germany, 359–370.