Graphics, Geometry, and Computing

Jorge Stolfi

DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

ABSTRACT

Computational geometry is a recent discipline with foundations in many branches of mathematics, and which is supposed to serve many applied areas. In this talk, after defining its subject matter, and reviewing its most important tools and paradigms, I will present a quick survey of some recent results that lie right at the frontiers of computational geometry. I hope this survey will give some idea of how computational geometry relates to those disciplines, and what we can and cannot expect from it.

1 What is computational geometry?

Computational geometry (CG) is usually defined as "the mathematical study of algorithms for solving geometric problems."

Like most definitions, this one seems fine, but only until one *really* needs it. When I started preparing this talk I soon realized that this definition was too vague, and could easily be stretched to include most of mathematics an computer science. So, to make my task feasible, I must "clarify" that definition, by imposing four additional conditions.

First, consider the phrase "geometric problem." Obviously, any mathematical problem can be reformulated as a geometric problem in rather trivial ways ("How many triangles is two triangles plus two triangles?"). So, **Condition One** is that geometry must play an essential role—in the statement of the problem, or in the description of the algorithm, or in the accompanying proofs and analyses. Thus, for example, "compute the convex hull of n given points" easily passes this criterion; "sort n given points in left-to-right order" probably does not.

Second, we need to clarify the meaning of "algorithm." You may have noticed that in its origins, and throughout most of its history, geometry was primarily a *computational* discipline. To the ancient Greeks, ruler and compass were not artists' tools, but rather computing instruments—the pocket calculators of Antiquity. The computational character of classical geometry is quite obvious in Archimedes's work on the quadrature of circle and parabolas; but it was already quite evident since the very beginnings. The geometric constructions of Euclid were essentially algorithms (complete with proof of correctness!). In fact, if we define computer science as the science of algorithms, then computational geometry is the most ancient branch of computer science, since the earliest non-trivial algorithm—Euclid's procedure for the greatest common divisor—was originally stated in geometric terms, namely find the greatest common measure of two commensurable line segments.

This computational approach to geometry was still evident during the Renaissance, in the new sciences of perspective and projective geometry, and, a couple of centuries later, of descriptive and Cartesian geometry. However, I have no intention of reviewing several thousand years of geometry in fifty minutes; so I will require as **Condition Two** that the algorithms in question be *computer* algorithms. Specifically, I will not bother with ruler-and-compass constructions, or any similar procedures that are meant to be carried out by hand.

The key to **Condition Three** is the word *mathematical.* I will take this word as a requirement for several things: a precise statement of the problem, in mathematical language; a complete and precise description of the algorithm; an analysis of its relevant parameters (running time, memory size, accuracy, etc.); and a proof that the algorithm is correct. This last item in particular will mercifully exclude an uncountable multitude of ad-hoc geometric "algorithms" invented on the fly by harried computer graphics programmers—including thousands of ingenious but hopelessly wrong point-in-polygon routines. In fact, I will go even further, and demand that the proof of correctness itself be correct—which unfortunately will force me to ignore a few exciting algorithms that have appeared in the literature...

But we aren't done yet; there is still one more restriction that I want to add. To understand why we need it, we must have a look at how the discipline got its name..

1.1 History and sociology

As far as I know, the name "computational geometry" was first used by Robin Forrest [42, 43] in 1971 to describe what is now currently called computer-aided geometric design (CAGD); or, more exactly, the most theoretical parts of CAGD, divorced from software and hardware issues. (The discipline itself is of course quite bit older than the name, witness the work of Bézier and De Casteljau in the sixties.)

Then, in 1974 or so, name "computational geometry" was redefined by Ian Shamos, then a student of Franco Preparata at the University of Illinois. In his Ph.D. thesis [110], Shamos gave algorithms for several geometric problems, such as computing the convex hull of n given points on the plane, and analyzed their asymptotic efficiency. Shamos's approach borrowed the values, methods, paradigms, and language of theoretical computer science, as set forth by Don Knuth [70] which were quite different from those of the CAGD culture. Inspired by Shamos's thesis, many theoretical computer scientists and students (I among them) started working on the same kind of problems and with the same general approach. So was born a second kind of computational geometry, the "asymptotics" school. (Incidentally, Shamos's thesis was later published as a textbook that is still the standard introduction to the subject [112].)

Over the past ten years, people from both sides of computational geometry, inspired by the shared name and by superficial similarity in subject matter, have repeatedly called for the unification of the two branches. So, for instance, the ACM Symposium on Computational Geometry — the major event for Shamos's brand of CG — tried hard for several years to attract papers on CAGD and its applications. Unfortunately, it looks like these attempts were not very successful; the two communities seem as separate as they always were. Indeed this year's ACM Symposium had no CAGD papers at all; I can only presume that the idea has lost some of its momentum.

In retrospect, this failure to unify the two communities is hardly surprising, given the large sociological differences between them. For one thing, people working on the CAGD brand seem to have mostly engineering or applied mathematics background, and are therefore at home with calculus and linear algebra; whereas those working on the asymptotics brand came mainly from other areas of theoretical computer science, and generally prefer to deal with discrete mathematics. Moreover, the CAGD people are (or at least were originally) employed by industry to work on concrete applications like car design; whereas the people from the other camp live in academic or semi-academic institutions, doing fundamental self-directed research.

These differences necessarily impose quite different slants on the contents and values of the two disciplines. In the CAGD school, the emphasis is on application-related issues such as controllability, continuity, practicality of the surface models; and computational complexity is usually measured by counting operations, where constant factors do matter. In contrast, the asymptotics school measures the performance of algorithms with rather abstract metrics which bear only a tenuous relation to actual costs; as a consequence, its algorithms and data structures tend to be quite complex and expensive (in fact, the algorithms often are too complex to be implemented, and exist as paper studies only).

Now perhaps you can appreciate why I want to add another restrictive clause to my definition of CG. Having been raised in the asymptotics school, I naturally know very little about CAGD; therefore, to avoid exposing my ignorance, I will attempt to define that whole area away.

So, how can we differentiate between CAGD and "true" CG? Generally speaking, CAGD problems deal with a small fixed number of complex objects—a single patch of surface of high degree, or a few adjacent patches—or with highly regular arrays of such patches, that interact in highly regular ways. Therefore, as **Condition Four** I will require just the opposite: the problem must involve an unbounded number of simple geometric objects, and—above all–the problem's complexity must lie mainly in the combinatorics of the interaction between those objects, rater then in the objects themselves.

1.2 Values, methods, and paradigms

Computational geometry, as we saw, borrowed many of its values and paradigms from theoretical computer science, specifically from the asymptotic analysis of algorithms. To understanding what CG can and cannot contribute to other disciplines, we need to take a close look at those paradigms, especially the criteria generally used to compare various geometric algorithms.

Asymptotic complexity. Theoretical computer scientists (and hence computational geometers) generally measure the efficiency of an algorithm by *asymptotic* growth of its *worst-case* running time, as a function of the size of the input data. "Asymptotic" here means that they do not care about the running time per se, but only about how fast it grows as problems become larger and larger. "Worst-case" means that they consider only the *maximum* time taken over all inputs of a given size, rather then the average or typical time.

The asymptotic behavior of the running time $t(n)$, as a function of the input size n, is usually expressed in the so-called *big-oh* notation [70]. Specifically, one says that $t(n)$ *is* $O(f(n))$, for some function f, if there exist constants c and N (depending on the particular implementation) such that $t(n) < cf(n)$ for all $n > N$. In other words, $t(n)$ is bounded by some multiple of $f(n)$, provided n is large enough.

Naturally, the big-oh notation can be used also to describe the growth of memory consumption, output size, or any other quantity that is a function of size of the input data. Note, however, that it can only be used to specify *upper* bounds. For lower bounds, one should use the analogous *big-omega* notation: one says that $t(n)$ *is* $\Omega(f(n))$ when there are constants c' and N' such that $t(n) > c'g(n)$, for all $n > N'$. Finally, one says that $t(n)$ is $\Theta(f(n))$ if it is both $O(f(n))$ and $\Omega(f(n))$, that is, $cf(n) < t(n) < c'f(n)$ for some constants c, c' and all sufficiently large n.

The most important thing to remember about the big-oh notation and its relatives is that they do not give any information on how fast the algorithm *actually* is. Asymptotic analysis won't tell you whether a program based on the algorithm in question, running on your favorite computer, will take one second or one year to process 100 data points. At best, it will only give you an idea of what to expect when you double the number of data points—whether the processing time will also double, or quadruple, or become astronomical.

Nonetheless, there are good reasons for using these asymptotic measures, rather then actual running times, when comparing different algorithms for the same problem. For one thing, the actual running time of a program depends a lot on "environmental" factors such as coding style, hardware, compiler, and the size of input data. On the other hand, the asymptotic growth of the running time is unaffected by those factors: an algorithm that takes $\Omega(n^2)$ time when coded in BASIC on a PC will continue to take $\Omega(n^2)$ time when reimplemented in optimized C on a supercomputer. Thus, the asymptotic analysis is an invariant property of the algorithm.

In fact, even from a practical point of view it is far more important to know whether the running time grows as $O(n)$ or $O(n^2)$ or $O(2^n)$, than to know the exact running time for a particular implementation. Even if your supercomputer is a million times faster than your PC, a $\Theta(n)$ algorithm on the latter will surely beat a $\Theta(n^3)$ algorithm on the former, once n gets big enough—perhaps even for $n = 1000$ or so. By the same token, an algorithm that takes $\Theta(2^n)$ time simply cannot be used for problems beyond a given size, because adding one more point doubles the cost.

On the negative side, of course, the asymptotic analysis cannot distinguish between two algorithms that have the same rate of growth (say, both $O(n \log n)$) but one is usually a hundred times slower than the other when implemented with the same care on the same hardware.

Moreover, asymptotic analysis describes the behavior for sufficiently large n, but does not always say what "sufficiently large" means. For some problems, the values of n where the analysis starts to apply may lie well beyond the range of practical applications. The following example is due to Bob Sproull: suppose we have a data base consisting of n points in k-dimensional space $\Re^k$, and we want quickly find the data point that is closest to a given query point p. One way to solve this problem is to represent the data base by a data structure called a k-dimensional tree. Theory say that, if the points are reasonably well distributed, one can find the nearest data point using only $O(\log n)$ operations. This sounds like a great improvement over the $\Theta(n)$ operations required by a straightforward sequential search; however, a more detailed analysis also shows that the $O(\log n)$ bound only starts to apply when n is much larger than 2^k. Indeed, when Sproull tested the k-dimensional tree algorithm with $k = 16$ and $n \approx 75,000$, it still examined some $72,000$ data points. So, even though the asymptotic analysis says that the fancy algorithm is very fast for any k, in practice it is effective only for small k.

Average-case anlysis. Another shortcoming of worst-case analysis is that it reflects the cost of solving the hardest cases of the problem, even if they are extremely unlikely to come up in practice. For example, worst-case analysis will say that an algorithm that takes $\Theta(n^2)$ time for *every* input of size n is better than one that takes $\Theta(n^3)$ time for *one* particular input of size n, and $\Theta(n)$ time for the rest. In order to avoid this problem, theoretical computer scientists sometimes analyze the *average case asymptotic complexity*, where instead of looking at the maximum time over all problems of size n, one looks at the expected (average) time when those problems are taken from a specified probability distribution.

Unfortunately, the average-case asymptotic cost of an algorithm need not be very relevant to practice, either. For one thing, it is still insensitive to constant factors, and makes the same assumption of "sufficiently large n" as the worst-case cost. But the main problem is that the average time usually depends quite strongly on the assumed probability distribution, and the actual distribution will vary drastically from case to case. For example, the well-known Floyd-Eddy convex hull algorithm takes $O(n)$ expected time when the n input points are is uniformly distributed *inside* a circle, and $O(n \log n)$ when the points are uniformly distributed *on* a circle.

This is hardly an extreme example; sometimes, as in the simplex algorithm for linear programming, the average-case performance can vary all the way from linear to exponential and beyond.

Randomized algorithms. In some cases, one *can* get meaningful expected-time analysis by the use of randomized algorithms, where the randomness is generated internally instead of being supplied by the input data.

For example, consider an algorithm that solves a geometric problem on n points—say, computing their convex hull—by the standard "divide and conquer" technique: split the points into two subsets by a vertical line passing through one of them, solve the problem for each subset, and combine the two answers. This approach will be efficient only if the two subsets have roughly the same expected size. However, if the splitting data point is chosen by a simple deterministic criterion—say, always take point $p_{n/2}$—then there will be certain "perverse" data sets that cause all points to end up on one side, every time. Therefore, the algorithm will perform quite poorly in any application where such perverse patterns happen to occur with high frequency.

If, on the other hand, the algorithm picks the splitting data point at random, then for any data set, no matter how contrived, the probability of obtaining many unbalanced splits in a row will be rather small, and the algorithm will perform well in any application. Since the randomness is internally generated, one can analyze the expected time for any fixed input, and then take the maximum of these expected times for all inputs of size n. This *asymptotic worst-case expected time* is therefore an intrinsic property of the algorithm, independent of the application.

To be honest, I should point out in the example above that one *could* choose the splitting point deterministically and still as to guarantee a balanced split, by (for instance) using an efficient median-finding subroutine. However, finding the median of n numbers is much more complicated and expensive than generating a random number between 1 and n. In fact, for many problems (geometric or not) there are simple randomized algorithms whose asymptotic average running time is much less that of any known deterministic algorithm. For these reasons, randomization is a key technique for the development of *practical* CG algorithms.

1.3 Computability

Provably hard problems. We remarked before that there is a vast difference in practicality between an algorithm that whose running time is polynomial in n (that is, $\Theta(n^c)$, for some constant c) and one that takes exponential time ($\Theta(c^n)$). Sometimes, the latter is merely the wrong algorithm. However, one can prove that some problems are so inherently hard that *any* algorithm for them must take exponential time. That is the case, for instance, when the answer itself has exponential size—for example, if one wants a list of all possible paths in a graph with n nodes.

Examples of such *provably hard problems* can be found in computational geometry, too. For example, the convex hull of n points in d-dimensional space may have on the order of $n^{d/2}$ faces [37]; so, if the dimension d is variable, any algorithm that produces a complete description of the hull will perforce take exponential time.

For a less obvious example, consider the problem of drawing a given graph of n vertices on the plane, with straight-line edges, so that the number of edge crossings is as small as possible. Bienstock [17] showed that there are graphs with n vertices such that any straight-line minimum-crossing drawing of them will require an exponential number of bits to specify the coordinates of the vertices.

NP-hard problems. Besides the provably hard problems, there is also a large class of problems (geometric and not) for which we only know of exponential time algorithms, and yet we cannot prove that they must take more than polynomial time. What is most frustrating is that this class includes many problems of great practical importance. One example is *Boolean satisfiability*: given a Boolean expression on n variables, find a set of Boolean values that will make that formula true. Other examples are the *Hamiltonian circuit problem*—given a graph with n nodes, find a path that goes exactly once through each of its vertex; and the *knapsack problem*—given a set of n real numbers, determine if there is some subset whose sum is exactly 1.

A common characteristic of these problems is that there are simple polynomial-time algorithms for *checking* whether a given answer is correct; but the only algorithms we know for *finding* a correct answer basically enumerate an exponential number of candidate answers, and check each one individually. One thing we can prove is that these problems are all equivalent in the sense that if we knew how to solve one of them in polynomial time, we would know how to solve all of them. For reasons that would take too long to explain, these problems are said to be *NP-complete*. ("NP" here stands for "nondeterministic polynomial.") One also uses the term *NP-hard* to mean "NP-complete or perhaps even harder."

As you may expect, theoretical computer scientists have spent a good deal of time and energy trying to find a polynomial-time algorithm for these problems, or trying to prove that there can be no such algorithm. This question — known as the *P=NP question* — is definitely the most important open problem in computer science, equivalent in importance to Riemann's hypothesis in number theory. By now, the P=NP question is considered to be so difficult to settle that showing that a problem is NP-complete is as bad news as proving that it is truly exponential; it effectively kills interest in the problem. It says that looking for an exact and general solution is a rather hopeless quest; if one really needs to, one should look for algorithms that produce only approximate answers, or try to restrict the input so as to avoid all hard instances of the problem.

Computational geometry has its share of NP-complete and NP-hard problems, too. Many of them are problems in combinatorial minimization in a geometric setting, such as computing the shortest network (*Steiner tree*) that connects a given set of n points in the plane [49], or the minimum number of convex polygons whose union is a given simple polygon [45, 28], decomposing a three-dimensional polyhedron into tetrahedra with minimum number of new vertices [102]. Fortunately, most of these negative results are not very problematic for practical applications, because one can often exclude the hard cases or can make do with approximate solutions.

1.4 Computation models

The rounding problem. Rounding errors are another major stumbling block on the road from computational geometry to practical applications. When developing algorithms, or proving their correctness, computational geometers typically assume that tests like "is point p on the right side of line l" or "is pqr a counterclockwise triangle" are evaluated exactly. However, geometric algorithms are usually implemented using approximate arithmetic, either fixed or floating point. As a consequence, the tests will occasionally return incorrect results.

Rounding errors are far more serious in computational geometry than in other typical numerical problems, such as integration or linear algebra. The strong combinatorial aspect of most geometric algorithms means that the desired output is, as a rule, a highly discontinuous function of the inputs. For example, in the convex hull problem, a small change in the position of one data point, or a small rounding error in some geometric test, may add or remove an arbitrary number of vertices of the output polygon.

If this were the whole story, we might still be able to guarantee that a geometric program is "approximately correct" in the sense of backwards error analysis: namely, that the output is the correct solution for some problem that differs only slightly from the given one. Unfortunately, the output of a geometric algorithm is often expected to satisfy some non-trivial combinatorial properties; and these properties may be impossible to satisfy if the required tests cannot be carried out exactly. In that case, the program output may be wrong for *all* conceivable inputs. For example, a convex hull algorithm implemented in floating point may return a polygon that is not convex, or not even simple, and therefore cannot be the hull of any set of points.

Worse yet, the rounding errors may cause the results of geometric tests to be not only wrong, but logically inconsistent with each other; which in turn may cause the algorithms to crash, or loop forever, or output malformed data structures. For example, if we have four points $ABCD$, and the triplets ABC, ABD, and ACD are collinear, then in theory the triplet BCD must also collinear. However, this conclusion may not be valid if the collinearity tests are computed with inexact arithmetic. So, an algorithm that implicitly assumes that property of collinear triplets may end up in "impossible" states when implemented with approximate arithmetic.

A naive response to this problem is to compute the tests in double precision. Intuitively, one would expect that reducing the magnitude of rounding errors by a factor of 10^{-6} would reduce the probability of erroneous test results in the same proportion; but, paradoxically, this is not the case. For example, consider the case of multiple collinear points, which are a rather common phenomenon in real-life data. When encoded in floating point, those points will deviate from a straight line by a few times the machine's rounding error; and that is precisely the situation where the collinearity test is most likely to give incorrect and inconsistent results. Therefore, these cases will remain equally problematic no matter how many digits one uses.

People from both the theoretical and practical side of the fence have tried many times to overcome this obstacle, but the results have been quite modest so far. For some simple problems, one sometimes can obtain simple and robust algorithms by choosing the appropriate representation for the input data [47] or by the judicious use of double precision [93]. However, for slightly harder problems, the robust algorithms one can find in the literature are substantially more complicated than the ones which assume exact arithmetic, and their analysis is by no means trivial [76, 44, 103].

Another approach is to round all input coordinates to a relatively low precision (say, 16 bits), so that all geometric tests can be carried out accurately. However, that merely shifts the difficulty from the algorithm itself to this preprocessing step, which must round the input data without introducing any inconsistencies (such as concavities in convex polygons, crossed edges in triangulations, and so on). Unfortunately, performing such a consistent rounding seems to be as hard as designing robust algorithms with inaccurate tests. While Greene and Yao successfully used this approach to compute intersections between sets of line segments [52], Milenkovic and Nackman proved that consistent rounding of polygons is NP-complete [87]. (However, this result may be an artifact of their particular statement of the problem, so there is still some hope.)

Rational arithmetic. Given the difficulty of building robust geometric algorithms from inaccurate tests, the use of exact multi-precision arithmetic makes a lot of sense, provided one can afford the added time and space costs [50].

In particular, many geometric algorithms require only the four arithmetic operations. Such algorithms are precisely those that can be carried out on paper using only straightedge and parallel rulers, i.e. the constructions of affine and projective geometry; they include convex hull construction, triangulation and intersection of polygons, Voronoi diagrams, plotting parametric polynomial splines, and so on. In those cases, if the input coordinates are rational numbers, then all internal and output quantities will be rational numbers. In fact, Many such problems require only the evaluation of low-order polynomials in the input coordinates, so if the precision of the latter is bounded a priori, all arithmetic can be done using statically allocated multiprecision numbers of fixed size.

Algebraic numbers. Rational arithmetic is not enough for certain operations of Euclidean geometry, such as marking a given distance on an arbitrary line, computing the intersection of two circles, or bisecting a given angle. (However, one *can* compare the lengths of two line segments, or enumerate a dense set of points on a circle, or check whether two angles are equal.) To carry out all construction of ruler and compass, one needs to compute with *surds*, the subset of the algebraic numbers that can be obtained from the integers through arbitrary combinations of $\{+, -, \times, /, \sqrt{\ }\}$. Other desirable geometric operations, such as trisecting an angle, rotating a figure by a rational angle in degrees, or computing the intersections of a Bézier arc with a given line require computations with algebraic numbers more general than surds. The main obstacle to this approach is cost: algorithms for exact arithmetic with algebraic numbers do exist, but they tend to be quite slow. Apart from its practical considerations, the large cost of arithmetic usually dwarfs the cost of the algorithm per se. Exact algebraic arithmetic seems to be interesting only in the cases where the degrees of all inputs, outputs, and intermediate quantities can be bound in advance.

Transcendental numbers. There are of course many geometric operations for which even algebraic numbers are not enough. Examples are computing the length of a circle or Bézier arc, computing points on non-algebraic curves like the catenary or the logarithmic spiral, or rotating a figure by a given number of radians. Problems that depend on the exact comparison of such quantities may well be mathematically unsolvable.

Symbolic algebra. The use of multiple-precision rationals or with algebraic numbers is still 'numerical" computation, where each operand represents a single real number. A few geometric algorithms have gone further by using *symbolic* computation, where the operands may be functions (usually polynomials) with one or more unbound variables.

As we shall see in section 2.2, symbolic computation is used in algorithms that manipulate algebraic surfaces, such as Collins cylindrical decomposition method. However, in a few instances the contributions of symbolic algebra were more direct than that. For instance, symbolic algebra provided a general technique for graceful handling of *degenerate cases*, another longstanding hurdle in the way from theory to practice.

Degenerate cases are all sorts of geometric coincidences in the input data of a geometric algorithm that lie right at the borderline between two markedly different outcomes or computation paths. For example, a typical convex hull algorithm may test whether three points form a clockwise or counterclockwise triangle, and do different things in each case. But what if all three points happen to be collinear? This "degenerate" case often needs to be handled separately from the two "normal" ones; for instance, we may need an extra test to decide which of the three points lies between the other two. Theoreticians usually ignore This issue by assuming, "for simplicity," that such degenerate cases do not occur; but the poor programmer can hardly ever do the same. Since conditional branches tends to multiply rather than add, the possibility of degenerate cases make programs much more complicated than what they should be.

Enter symbolic algebra to the rescue. Edelsbrunner, Mücke and Yap developed a general technique, which they call "simulation of simplicity" or SOS, for automatically handling degenerate cases in many geometric algorithms [39, 127]. The idea is to replace each real number x_i in the input by a polynomial $x_i + a_{i1}\varepsilon_1 + a_{i2}\varepsilon_2 + ...a_{in}\varepsilon_n$, where the ε_i are dummy variables (*virtual perturbations*) and the coefficients a_{ij} are chosen so that the polynomials are algebraically independent. It turns out that, if the polynomials have the right degree and number of variables, then one can always assign arbitrarily small values to the ε_i in such a way that all geometric tests give unambiguous results.

Actually, one does not specify any of the values ε_i; instead, one performs all the computations and tests on the symbolic polynomials themselves.

2 CG and pure mathematics

Let's look now at the frontier between computational geometry and the relevant branches of pure mathematics, especially the various flavors of geometry—classical, projective, analytic, convex, algebraic, combinatorial—and also topology and graph theory.

2.1 Classical geometry

I mentioned above that geometry, from ancient Egypt to the end of the Renaissance, was primarily a computational discipline. By the nineteenth century, however, geometry–like most of mathematics—had evolved into highly abstract science, concerned with describing the *properties* of geometric objects, rather than developing *algorithms* for their manipulation. The objects of study themselves became more abstract and non-constructive—d-dimensional figures, non-Euclidean spaces, geometric transformation groups, general metrics, topological manifolds. A characteristic example of this trend is Felix Klein's *Erlangen program*, in which geometry is defined as the study of properties of a set of points that are invariant under a given group of transformations of the set to itself.

This abstract approach to geometry produced many deep and beautiful theories, and did a lot to improve our understanding of mathematics (and of the world, and of our own minds); but at the same time it widened the gap between pure and applied geometry. By the beginning of this century, the former had became an academic discipline feeding largely upon itself; while the latter became just a standardized set of tools for engineers and architects (such as nomography, descriptive geometry, perspective projection, graph plotting). Of course, this dichotomy between theory and practice in geometry was merely a projection of what was happening to mathematics as a whole. This transformation was presumably a consequence of the recognition of mathematics as a scientific discipline in itself, which led to the creation of mathematics departments in universities, and therefore to a community of academic mathematicians working in a system where publication was valued more than application.

Fortunately or unfortunately, the advent of the computer broke this trend, and has brought pure and applied mathematics closer to each other then they have been in centuries. On one hand, pure mathematicians were attracted by the the formidable intellectual challenge of designing and analyzing computer algorithms. On the other hand, programmers and engineers discovered that their traditional mathematical tools were not up to that task, and so they turned to pure mathematics in search of suitable ones.

This reapproachment was not instantaneous, of course, and did not happen at the same time for all branches of mathematics. Calculus and linear algebra were the first; combinatorics and other discrete mathematics followed a little later. As for geometry, the reconciliation began only in the last ten years or so. One could say that it began officially in 1986, at the Conference on Discrete and Computational Geometry in Santa Cruz, where a few dozen pure and computational geometers met to see what they had in common. The two camps had quite a hard time communicating—they seemed to speak two different languages—but that seed sprouted, grew, and bore fruit. computational geometers are increasingly using the language and results of pure geometry, and pure geometers are becoming more and more interested in computational problems.

In retrospect, it is amazing how little the first computational geometers knew about geometry itself. In Shamos' thesis, and in the proceedings of the first few ACM Symposia on Computational Geometry, one will find rather sophisticated algorithms and data structures, but no advanced geometric concepts—in fact, practically nothing beyond the two-dimensional Euclidean and Cartesian geometry that I had learned in high school. For instance, I am embarrassed to say that when I entered CG as a graduate student I knew next to nothing about projective geometry. Yet, at the time, I didn't feel that I was missing anything important. (In fact, I only began to realize how little I knew after several years of writing a thesis on that same subject!) Considering how clumsily the early CG papers handled points at infinity and geometric duality, I can only conclude that I was not alone in my ignorance.

I must say that the situation has improved considerably since then. Every year the papers presented at CG conferences depend more and more on advanced geometry and mathematics. Today, computational geometers must be familiar with d-dimensional and projective geometry, topology, algebraic and combinatorial geometry, and more. Some of the most efficient algorithms are built on top of rather advanced concepts, like ϵ-nets, cylindrical decompositions, and matroid theory. I will discuss some specific examples later on.

Computing with abstractions. Surprisingly, this shift towards abstract mathematics was greatly helped by advances in the most concrete areas of computer science, namely programming languages and computer hardware.

In the early days of CG, problems and algorithms were often stated explicitly in terms of Cartesian coordinates, and programs would specify all numerical computations explicitly. Nowadays, programs are increasingly specified in more abstract terms: instead of working with arrays of numbers, they work with lists of points or sets of lines. One motivation for this change is is necessity: as the problems get more complex, is becomes more important to hide unessential detail. The other motivation is opportunity: modern programming languages make it easier to deal with geometric objects rather than raw coordinates.

For example, in FORTRAN one would typically compute the midpoint of a segment with the code

```
DIMENSION XP(10), YP(10), ZP(10)
...
XP(I) = (XP(j) + XP(K))/2.0
YP(I) = (YP(j) + YP(K))/2.0
ZP(I) = (ZP(j) + ZP(K))/2.0
```

whereas in Modula one would more likely write

```
TYPE Point = RECORD x, y, z: REAL END;
VAR P: ARRAY [1..10] OF Point;
...
P[i] := midpoint(P[j], P[k]);
```

Note how record types and the ability to return a record as a procedure result encourage us to program in a higher level of abstraction, where we deal with points rather than real numbers. One manifestation of this trend is the increased availability of geometric packages for symbolic algebra systems, and of languages specifically tailored for geometric computation [59].

Another contributing factor to this trend is the increased availability of graphical input and output devices. Surely, it is much easier to think of points (rather than numbers) as the basic objects when the data are input by clicking with the mouse (rater than typing two numbers) and output by painting dots the screen (rather than as three printed numbers).

Projective geometry. Computational geometers have been forced to learn a lot about projective space and related concepts. The primary motivation, I presume, was probably the heavy usage of homogeneous coordinates and projective maps in computer graphics. Another reason is that projective geometry often simplifies the algorithms by removing degenerate cases (such as parallel lines and points at infinity.) There have been a few books and papers specifically devoted to teaching projective geometry to programmers and computer scientists, such as the book by Penna and Patterson [97]. My own Ph.D thesis was on this area, specifically about an oriented version of projective geometry for use in computational geometry and computer graphics [117].

The most useful tool that CG imported from projective geometry is the notion of geometric duality. On the plane, the projective duality maps lines to points, and points to lines, while preserving their incidence relations. Similar dualities can be defined in spaces of any dimension (in three-space it maps points to planes, lines to lines, and planes to points); and it can can be applied to any geometric figures defined in terms of points an lines. Duality is valuable to computational geometers because it lets one use a single geometric algorithm to solve two apparently unrelated problems [22, 117]. For instance, the same algorithm computes the intersection of n half-planes or the convex hull of n points.

Plücker coordinates. Another valuable contribution from projective geometers to their computational brethren is the *Plücker representation* of lines in 3-space [117]. If $[u_0, ..u_3]$ and $[v_0..v_3]$ are the homogeneous coordinates of two points u and v, the Plücker coordinates of the line uv are the six real numbers

$$l_{01} = \begin{vmatrix} u_0 & u_1 \\ v_0 & v_1 \end{vmatrix} \quad l_{02} = \begin{vmatrix} u_0 & u_2 \\ v_0 & v_2 \end{vmatrix} \quad l_{12} = \begin{vmatrix} u_1 & u_2 \\ v_1 & v_2 \end{vmatrix}$$

$$l_{03} = \begin{vmatrix} u_0 & u_3 \\ v_0 & v_3 \end{vmatrix} \quad l_{13} = \begin{vmatrix} u_1 & u_3 \\ v_1 & v_3 \end{vmatrix} \quad l_{23} = \begin{vmatrix} u_2 & u_3 \\ v_2 & v_3 \end{vmatrix}$$

Plücker coordinates for lines have many advantages over other more straightforward representations, such as pairs of the form (*point*, *direction*). For one thing, Plücker coordinates are unique except for a constant scaling factor (just like ordinary homogeneous coordinates), and are well-defined even for lines at infinity. Also, many common geometric operation on lines, planes and points reduce to simple bilinear combinations on their Plücker and homogeneous coordinates—without divisions or square roots. In particular, the relative orientation of two oriented lines in three-space is given by the sign of a modified dot product of the two Plücker tuples. For this reason, Plücker coordinates often allow us to transform problems about lines in three-space into problems about points or hyperplanes in five-space, which are sometimes easier to understand [21].

Hyperbolic geometry. Some adventurous computational geometers have left the projective plane and ventured into more sophisticated domains, such as hyperbolic space. At Princeton University, Bill Thurston (a mathematician) and David Dobkin (a computational geometer) have collaborated in for many years in the exploration of this field, and have produced some stunning computer imagery as a byproduct. Those of you who were at this year's SIGGRAPH conference surely recall the video movie *Not Knot* produced by the Geometry Center of the University of Minnesota [57].

Realizability of configurations. Another example of recent flow of ideas from pure to computational geometry is the work by Sugihara, Crapo, Ryan, Whiteley, and others on the problem of deciding whether a single two-dimensional network of polygons can be the projection of a three-dimensional polyhedron [118, 125, 27].

A more abstract version of this problem is the question of whether there exist an arrangement of geometric objects (points, lines and planes) satisfying a given set of incidence and orientation constraints. For example, given an $m \times n$ matrix σ_{ij} over $\{+1, -1\}$, we can ask whether there are n points $p_1, ..p_n$ in three-space, and m oriented planes $\pi_1, ..\pi_m$, such that σ_{ij} is the position of point p_i relative to plane π_j. As another example, suppose we are given an $n \times n \times n$ matrix σ_{ijk} over $\{+1, -1\}$, and we need to determine whether there are n points $p_1, ..p_n$ on the plane such that σ_{ijk} is the orientation of the triangle $p_i p_j p_k$. The importance of this problem to CG inspired some fundamental work by both pure geometers like Goodman, Pollack, and Sturmfels [51, 18] and quintessential computer scientists like Knuth [71]. In particular, Knuth developed an axiomatic theory for realizable configurations, and used it to develop a simple and efficient convex hull algorithm that is guaranteed to work in the presence of arbitrarily large rounding errors.

2.2 Algebraic geometry

Algebraic geometry has a large body of algorithms of its own, and it is hard to define the frontier between it and symbolic algebra in general. Fortunately for me, most of those results don't qualify as "true" computational geometry, because they do not satisfy Condition Four: they deal only with a few objects at a time, and the complexities of the problems are usually algebraic rather then combinatorial. For example, this criterion disqualifies Kaltofen's algorithm for computing the factors and components of an algebraic curve [65], or Bajaj and Ihm's algorithms for interpolating a surface between specified closed curves [11]—even though those papers were presented at "true CG" conferences. It also excludes various important algorithms for rendering algebraic surfaces [107].

On the other hand, this criterion *does* classify as "true CG" some fundamental algorithms of algebraic geometry, such as Collins cylindrical decomposition algorithm [7]. Given an arrangement of arbitrary algebraic surfaces in $\Re^n$, Collins's algorithm identifies the connected cells of that arrangement, and further decomposes them into simply connected "cylindrical" cells. Loosely speaking, it generalizes the well-known algorithms for "trapezoidalization" of planar polygonal maps to higher dimensions and surfaces of higher degree. Collins algorithm is the basis for several other "true CG" results, such Chazelle and Sharir's data structures for point location in algebraic cell complexes [23].

Another important example of algebraic geometry invading "true CG" is the the paper by Ben-Or [14], where he uses the theorem of Milnor and Thom on the *topological* complexity of algebraic varieties to derive lower-bounds for the *computational* complexity of many geometric problems.

2.3 Differential Geometry

I have found surprisingly little research connecting CG to differential geometry. As in the case of algebraic geometry, there are plenty of algorithms for differential geometry problems, but most of them fail Condition Four and are better classified as CAGD.

In fact, the only valid example I can think of is a paper by Guibas and Seidel [54] on the convolution of polygonal lines or convex polyhedra. The convolution of two curves or surfaces A and B is defined as the locus of all sums $a + b$, where a, b are points on A and B with parallel tangents. Guibas and Seidel give algorithms that compute the convolution in time proportional to the size of input plus the size of output. However, this is not a very satisfactory example, because the differential geometry used is rather elementary. Could this be a new research area waiting to be explored?

2.4 Combinatorial Geometry

Combinatorial geometry can be loosely defined as the science of counting (exactly or in in the asymptotic sense) the number of configurations of geometric object satisfying certain properties. For example, among n points on the plane, how many pairs can be precisely at distance 1 apart? Or, if we are given six points on the plane, two red, two green and two blue, can we always partition them into two red-green-blue triangles so that their interiors overlap?

Not surprisingly, combinatorial geometry is the branch of "pure" mathematics where cross-fertilization with computational geometry has produced the most fruit. In fact I can' hope to offer more than a glimpse of all the work that lies in this boundary area. I have to omit, for example, all the theorems of combinatorial geometry that have been motivated by computational geometry problems.

The reason for this close link between combinatorial and computational geometry should be clear: the efficiency of geometric algorithms is often determined by how many times certain geometric patterns can occur among the data being processed. The close relationship between the two disciplines is similar to that observed in the sixties between compiler engineering and formal language theory.

Epsilon nets. In some cases, certain results from combinatorial geometry gave rise to an entire branch of computational geometry. Perhaps the most obvious example is the theory of *epsilon nets*, which are the key to a number of fast geometric search algorithms [61, 4, 81, 1, 2, 83, 94, 82]. (Unfortunately, the concept itself is a bit too complex to explain here.)

Arrangements of hyperplanes. Another family of results is concerned with enumerating arrangements of lines on the plane, or planes in three-space. For example, an elementary theorem of combinatorial geometry says that n lines on the plane determine $O(n^2)$ regions (more precisely, $\binom{n}{2}+\binom{n}{1}+\binom{n}{0}$). How fast can we compute those regions? A naive analysis would say $O(n^3)$ time, since we need to compute $O(n^2)$ regions, a region can have up to n edges, and each edge takes some fixed time to compute. However, a more careful analysis shows that the total number of edges of all the regions is only $O(n^2)$. In fact, if we look only at the regions adjacent to any line of the arrangement, we will count $2n$ regions but only $O(n)$ total edges, even though any of those regions may have up to n sides. This observation leads to a simple but optimal algorithm for computing all the regions of an arrangement [37].

An extension of these results has to do with counting the total complexity of m arbitrarily selected regions in an arrangement of n lines. The naive bound would be mn, but we already know that this is a poor bound when m is greater than n. In fact, it can be shown that the total complexity is at most $O(n+\min\{m\sqrt{n}, n\sqrt{m}\})$ [40, 37]. This result can be generalized in many ways [8, 58, 6].

A curious result that sits in the frontier between topology and combinatorial geometry was inspired by the following problem of computational geometry. Suppose we are given a closed polygonal line in space that is not knotted; that is, we could untwist it and flatten it out so that it becomes a planar polygon, without any edges ever crossing during this process. Topology tells us that we can always find a topological disk whose boundary is the given polygon—a *spanning surface* for the polygon—that is bent and stretched in such a way that does not intersect itself. The problem is to compute a polyhedral surface with this property. To everybody's surprise, Jack Snoeyink showed that an unknotted polygon with n segments can still be so convoluted that any spanning polyhedral surface for it must have an exponential number of faces [115].

2.5 Convexity Theory

Convex polygons and polyhedra have been a staple of CG since its very beginning. One can see here the same pattern we observed with respect to projective and combinatorial geometry: while the first results in the field used only elementary properties of convex figures, we now find CG practitioners delving deeper and deeper into the rich theory of convex bodies, which was already a major branch of pure geometry well before the age of computers.

As in the other fields, one can see intellectual traffic going both ways between CG and convexity theory: computational problems motivate theoretical results, and theoretical results lead to improved geometric algorithms. For example, Herbert Edelsbrunner and Raimund Seidel, who see themselves as computational geometers, have settled several longstanding open questions of convexity theory about the number of faces and other properties of n-dimensional polytopes [37, 108, 120]

Transversals. An example of the non-trivial results of convexity theory is Helly's theorem that says that if you have a set of bounded convex figures on the plane such that every three bodies have a common point, then all of them have a common point. Similar results hold in spaces of higher dimension. We can generalize this theory by replacing the notion of a point common to several sets with that of a *transversal* for those sets, which is a line or plane that intersects them all. O'Rourke, Edelsbrunner, Guibas, Hershberger and others gave efficient algorithms for computing line transversals of convex sets on the plane [101, 38, 62]. Wenger, Pollack and others have obtained algorithms and existence conditions for line and hyperplane transversals in higher dimensions [20, 99, 10, 124].

2.6 Topology

Topology of two-dimensional maps. In the frontier between computational geometry and topology, we find many algorithms and data structures for boundary representation (B-rep) models. Mathematically, such a model is a subdivision (partition) of some geometric space into a finite number of *elements*—cells, faces, edges, vertices—each having simple topology and geometry.

With the due exceptions, the *geometric* problems arising in this context usually do not qualify as "true CG," since they involve only a single element, or a few adjacent elements at a time. However, the *topological* problems tend to be more global, and the algorithms involve non-trivial combinatorics; so here is where we find the computational geometers.

Because of their relevance to CAGD, subdivisions of two-dimensional surfaces are particularly well studied, and several data structures have been developed for them. As shown by Edmonds, the important elements in such subdivisions are the edges and their ordering; the vertices and faces can easily be derived from this data. This principle was exploited by Baumgart's *winged edge* structure [13], and the *quad-edge* structure of Guibas and Stolfi [55]. Other data structures have used faces as the basic element, either triangular [74] or rectangular [92].

Most data structures for subdivisions are all based on the same general principle: nodes of the structure represent elements of the subdivision, and the links between nodes describe adjacencies and incidences between elements. The differences are all in the details of which elements are represented and which are implied, which relations are primitive and which are computed, and so on. One interesting approach that goes beyond this "elementary" paradigm is the hierarchical structure of De Floriani and Falcidieno, where a node may represent either an element or a group of many elements [30].

Topology of higher-dimensional subdivisions. There are also several data structures for subdivisions of higher-dimensional spaces. One possible approach is to use tetrahedra (in general, simplices) as the basic elements, glued together by their faces. Another approach is that of Dobkin and Laszlo, who developed an analogue of the quad-edge structure for three-dimensional subdivisions [35]. The same approach was eventually generalized to n-dimensional subdivisions by Brisson [19] and independently by Lienhardt [77].

Arrangements of pseudo-lines. Another active area in the frontier between CG and topology was inspired by the observation that certain combinatorial properties of line arrangements do not depend on the straightness of the lines, but only on the fact that any two lines cross at a single point. Therefore, those same combinatorial properties are valid also for any *pseudo-line arrangement*—a family of curves, any two of which intersect in at most one point. These properties were successfully exploited in several algorithms to algorithms for compute the faces and other properties of pseudo-line arrangements [37, 116].

Jordan sorting. It is well-known that sorting n arbitrary real numbers takes $\Theta(n \log n)$ comparisons. However, suppose we are given the coordinates where a simple Jordan curve intersects the x-axis, listed in the order that they appear along the curve. Since Jordan curves do not cross themselves, the order of these points is not arbitrary, but in fact highly constrained; so much so that the proof of the $\Theta(n \log n)$ lower bound is no longer applicable. The true complexity of sorting those intersections was a famous open question for several years, until Hoffman, Melhorn, Rosenstiehl and Tarjan eventually found an algorithm that would sort the intersection in only $O(n)$ time [64, 56, 9].

Spanning a curve with a polyhedral disk. A curious result that sits in the frontier between topology and combinatorial geometry was inspired by the following CG problem. Suppose we are given a closed polygonal line in space that is not knotted; that is, we could untwist it and flatten it out so that it becomes a planar polygon, without any edges ever crossing during this process. Topology tells us that we can always find a topological disk whose boundary is the given polygon—a *spanning surface* for the polygon—that is bent and stretched in such a way that does not intersect itself. The problem is to compute a polyhedral surface with this property. To everybody's surprise, Jack Snoeyink showed that an unknotted polygon with n segments can still be so convoluted that any spanning polyhedral surface for it must have an exponential number of faces [115].

Davenport-Schinzel sequences. Perhaps the most non-intuitive results that ever came out from the interplay of topology and CG are the many theorems and algorithms related to Davenport-Schinzel sequences. A set of n arbitrary line segments on the plane divides the plane into $O(n^2)$ connected regions; the boundary of any region is one or more polygons whose sides are pieces of the original segments. How many sides can a region have? Note that the same line segment may contribute several sides to the boundary of a single region, so the answer is not necessarily "at most n." The obvious bound is $O(n^2)$, but this turns out to be very pessimistic.

The key observation is that there are tight constraints to the order in which the segments may appear as we go once around the perimeter of a region. In particular, after we seen the segments a and b two times each, alternating in the pattern ...a...b...a...b..., topology tells us that we cannot ever see segment a again. So, the maximum number of sides on the boundary of a region is limited by the maximum length of any string over an alphabet of n letters that does not contain any consecutive repeated letters ...aa... or any subsequence of the form ...a...b...a...b...a.... Such strings are known as *Davenport-Schinzel (DS) sequences*, specifically DS sequences on n letters with at most 4 alternations.

The constraints on DS sequences imply that the sequences cannot be arbitrarily long, but how long can they be? Although DS sequences were first defined in 1965 (in the context of differential equations), their maximum length remained unknown until 1984, when Hart and Sharir—motivated by the CG problem above—finally determined a tight upper bound [60, 126]. The answer turned out to be quite bizarre: $O(n\alpha(n))$, where $\alpha(n)$ is the functional inverse of Ackermann's function. The latter is a function that grows amazingly fast, faster than 2^n or 2^{2^n} or $\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}n$ or any other function that one can write with finitely many `for` loops. So the inverse $\alpha(n)$ is an extremely slow-growing function of n. To give an idea, $\alpha(n)$ is less than 6 for any number n that can be written down using all the atoms in the known universe—but for even bigger n, $\alpha(n)$ can be as large as you want, so theoretical computer scientists are not allowed to treat it as a constant!

Anyway, these bounds on the lengths of DS sequences, and therefore on the complexity of any region determined by n segments, eventually led to optimal algorithms for computing such a region [38]. DS sequences soon turned up in all sort of geometric problems having to do with topological features of arrangements of n geometric objects, and consequently in the running time of many geometric algorithms [106].

2.7 Graph Theory

The connection between computational geometry and graph theory are quite extensive, and it is sometimes hard to tell where the boundary between the two lies. **Euclidean weighted graph problems.** Many problems of graph theory that involve undirected graphs with weights on the edges (such as shortest path and minimum spanning tree) can be turned into geometric problems by letting each vertex be a point of Euclidean space, and the weight of each edge be the Euclidean distance between its two endpoints (or some function thereof). So, for example, we have the the minimum Euclidean spanning tree [24, 128], or classical traveling salesman problem [121, 119], or the minimum-length triangulation [78], minimum-length matching [119], and so on.

This "geometrization" of a graph-theoretic problem is not as trivial as it may seem. The crucial thing to observe is that, even though the graph has $O(n^2)$ for n vertices, its description takes only $O(nm)$ bits, where m is the precision (in bits) of each coordinate. Put another way, the weights on the edges are highly constrained. For this reason, we can often find algorithm for problems in Euclidean graphs that are much faster than their analogs for arbitrary weighted graphs—typically, $O(n \log n)$ time, instead of $\Theta(n^2)$ or more.

Visibility graphs. In such problems, one often assumes the complete graph, i.e., every pair of vertices defines a valid edge. Alternatively, one may define the edges implicitly by some geometric criterion, e.g. all the pairs of vertices such that the straight line segment connecting them lies entirely inside a given polygon; or all the edges of a specific triangulation of the given vertices. This is the so-called "visibility graph" for the given points and obstacles. One can then ask for the shortest path in this graph between two given vertices, or the shortest spanning tree, and so forth [53].

2.8 Probability and Statistics

Probability and statistics are of course very relevant for the average-case analysis of randomized algorithms and of deterministic algorithms applied to random inputs. The combination of probability and geometry has been explored by Santaló [105] and a few other authors [12]. A typical result is that the average number of vertices on the convex hull of n points is $O(\sqrt[3]{n})$ if they are uniformly distributed in a circle, and $O(\log n)$ if they are uniformly distributed in a triangle or square. This result has been used to derive several convex hull algorithms with good expected running time [69, 36]. These are just a couple of examples, out of hundreds.

3 CG and applications

Let's now turn our attention to the connections between computational geometry and the applied sciences that are supposed to use its results.

First, I must mention an embarrassing fact. The applications of computational geometry are many, but certainly not as many as they should be. As far as I can tell, a large fraction–probably 75or more–of the published CG algorithms are never put to practical use, not even by the applications that ostensibly motivated them. In fact, many of the published CG algorithms never get coded, much less tested and measured. The main reason for for this poor showing is probably the unrealistic measures of efficiency used by computational geometers when comparing algorithms.

3.1 Computer graphics

This is usually the first think to be mentioned when listing applications of computational geometry; so much so that computational geometry could be defined as "computer graphics without the pretty pictures." Indeed, before it became a discipline on its own, computational geometry was effectively part of computer graphics. So it is not surprising to find many classical algorithms of the former, such as polygon triangulation and convex hull, being heavily used in the latter. One of the most illustrious examples is Bentley and Ottmann's plane sweep technique for computing object intersections and a variety of other graphical operations [15]. Computational geometry also contributed many algorithms based on the quad-tree and oct-tree data structures, useful for problems like point location, finding nearest neighbors, and ray tracing [104, 123].

Peterson-style formulas. D. Peterson of Sandia first noted that every simple polygon with n sides, no matter how convoluted, has a CSG representation whose n atoms are the supporting half-planes of its sides, combined with the boolean operations of Union and Intersection [98]. computational geometers have contributed efficient algorithms for finding such an expressions [34]. Unfortunately it seems that no such simple result exists for three-dimensional polyhedra, but nevertheless we have some partial results which may be practical [32].

Hidden-line elimination. A classical computer graphics problem is the removal of hidden lines and surfaces from images of polyhedral models. There are of course a great many algorithms developed by computer graphics people, which are designed to be fast for "typical" inputs, but whose worst-case performance is either unknown or far from optimal. However, there are also quite a few recent papers by theoreticians on algorithms that are optimal or near-optimal in the worst case asymptotic sense [84, 31, 96, 67, 46, 41, 90].

Ray tracing. The other classical computer graphics problem, ray tracing, has is only now beginning to receive some attention from theoreticians. There are some interesting theoretical developments, but the known algorithms still seem to suffer from the familiar CG shortcomings: artificially simplified problems, impractically large data structures, unknown constant factors [29, 25]

3.2 CAGD

In spite of all the care I took to remove CAGD from computational geometry, I must admit that there are quite a few topics that have been traditionally considered part of CAGD but that, according to my definition, fully deserve to be in "true CG."

Chief among these topics are the algorithms for computing Boolean functions (intersection, union, set difference) of polygons and polyhedra [89, 63, 111, 80, 72]. These algorithms provide a bridge between the two main solid modeling paradigms, namely boundary representation (B-Rep) and constructive solid geometry (CSG).

3.3 Visualization

Visualization and computational geometry can be related in two ways. The trivial way is the use of algorithm animation techniques to visualize the working of geometric algorithms. I say trivial because there is little difference between animating a geometric algorithm and animating a non-geometric one, except that the objects being handled have an obvious visual representation.

The non-trivial way is the use of computational geometry techniques in the visualization of certain kinds of data. Undoubtedly there are many examples of such use, but they are hard to find because they are scattered in the publications of the application disciplines. One example that comes to mind is the use of Voronoi tesselations in archeology and geography, to visualize the "regions of influence" of cities.

Another large class of applications of computational geometry to visualization is the construction of graph layout algorithms; there are quite a few papers on this area, including at least one in this conference [33, 16, 100].

3.4 Robotics

Robotics is another applied field that had relatively close ties with computational geometry. The main geometric problem in robotics is planning the motion of a "robot"—typically, an articulated arm— among a collection of obstacles, given its initial and final positions [73]. This problems has inspired a great number of papers in computational geometry [75, 91, 48, 114, 79, 113, 68].

Other problems from robotics that have inspired work in computational geometry are stable grasping of polygons and polyhedra by mechanical hands, the assembly of multi-part objects, and the reconstruction of a shape from tactile information [5].

However, as in the case of computer graphics, most of the "robotics" algorithms to be found in the CG literature are essentially theoretical exercises with little practical relevance. For example, computational geometers usually simplify the motion planning problem by assuming a highly idealized robot—a convex polygon, a line segment, or just a disk—moving in two dimensions among polygonal obstacles; and they strive to get exact algorithms with asymptotically optimal bounds for running times and storage space, without regard for algorithm complexity, generality, numerical robustness, or actual efficiency. In contrast, papers by robotics researchers tend to consider from the start complex articulated robots moving in three-dimensional scenes; their emphasis is on practical approximate algorithms with acceptable average-case performance (which is often determined by measurement rather than by analysis).

Still, there is a non-negligible intersection between the two populations, and many tools developed by the theoreticians were eventually adopted by the practitioners.

Also, may of the lower-bound results are undoubtedly useful, since they save people from wasting time in the pursuit of impossible goals or by adopting inefficient approaches. Moreover, the proofs supporting those lower bounds often tell the practitioner what are the pathological hard cases of the problem, which she must either exclude from the input, or handle by approximation, randomization, and similar heuristics.

3.5 Operations research

Many classical CG algorithms were directly motivated by problems from operations research, especially those of geographical nature—plant placement, route optimization, network design, and so forth. Indeed, by the time the theoretical computer scientists discovered the subject, the operations research people had already been developing "true CG" algorithms for ten or twenty years.

Linear programming. is perhaps the most famous of problem in this area, and the *simplex algorithm* is its most famous solution. They easily classify as "true CG" problems: even though they are usually stated in the language of matrix algebra, they are much easier to understand and reason about when recast in geometric terms.

There is a vast ocean of literature dealing with linear programming and the simplex method. Obviously, surveying it here is out of the question, and I won't even try. I do have a technical excuse for this cop-out, however: although the simplex algorithm is very fast and effective for practical problems, its *worst-case* running time grows exponentially with the problem size. Therefore, according to the canons of "true CG," it is a completely worthless algorithm.

That settled, let me cite just a few random pieces of research related to this problem. There are of course the polynomial-time linear programming algorithms of Khachiyan and Karmarkar, in which geometry plays an even more important role than in the simplex method itself [66]. There are also many fast (linear-time) algorithms by Megiddo, Seidel, and others for solving linear programming problems with small number of variables, especially 2 and 3 [85, 86, 109]. These algorithms have been used to solve many classical problems of computational geometry, such as finding the smallest circle enclosing a given set of points, or computing convex hulls in d dimensions.

Networks, paths, and matchings. Another famous class of operations research problems is the minimization of various networks and paths connecting a given set of points in the plane. A classical example is the *traveling salesman problem*, where the goal is to find a shortest path that visits every point exactly once. This seems to be a hard problem, perhaps NP-complete; but a few reasonably fast and good approximation algorithms are known [121, 122].

Instead of a path, once an ask for a tree-like network of cables that connects all the given points; that is the *Euclidean minimum spanning tree* problem. Actually, there are two versions of this problem, depending on whether one can add new nodes (*Steiner points*) or use only the old ones. (We already discussed the latter version in section 2.7.) This apparently minor variation has a profound effect on the difficulty of the problem: $O(n \log n)$ time without Steiner points, NP-hard with them.

Cutting and covering. Another large group of classical CG problems stolen from operations research deal to do with the approximation of geometric figures by a minimum number of simpler parts. For example, consider the problem of expressing a simple polygon as the union of convex polygons, using as few terms as possible. The polygons may be required to be disjoint (a *decomposition problem*) or may be allowed to overlap (a *covering problem*). Again, the two variants have vastly different complexities: disjoint decomposition can be done in $O(n^2)$ or less, while covering is NP-hard.

Other problems. There are many other problems that we could list here, such as finding the largest inscribed convex polygon in a simple one (the *potato peeling* or *convex skull* problem), or conversely, finding the smallest rectangle (circle, ellipse, triangle, etc.) that encloses a given figure; locating a "factory" so as to minimize the sum of distances to n given "outlets" in the plane [86]; finding the fastest path between two points across various regions with different travel speeds [88].

3.6 Data bases

Another major application area and source of inspiration for computational geometry is the construction of databases for multidimensional numeric data. For example, a query like "list all employees with $25 < \text{age} < 30$ and $1000 < \text{salary} < 2000$" can be recast as a geometric problem, by viewing each entry of the database as a point of the plane with coordinates 'age' and 'salary,' and interpreting the query as asking for all points that lie inside a given rectangular region of the plane. This admittedly trivial reformulation leads to many fast retrieval algorithms in which geometry plays a non-trivial role.

As a typical example, Paterson and Yao describe a data structure for n points of the plane that takes $O(n^2)$ space and $O(n^2)$ time to build; but once built, it lets one find all the points contained inside an arbitrary k-sided polygon in time $O(k \log n + s)$, where s is the number of points found [95].

4 Conclusions

What conclusions can we extract from this survey, especially with respect to the theme of this conference? I can think of two messages, one strongly positive and one somewhat mixed.

The positive message is that the interaction between computational geometry and the other disciplines was highly profitable to both—perhaps not a wholly unexpected message, but still worth saying.

The mixed one is about the nature of the barriers between the various disciplines. Recall that the two attempts at bringing computational geometry closer to another discipine–first CAGD, then pure geometry—had quite opposite outcomes. My understanding of why one failed and the other succeeded is that what creates barriers between scientific communities is neither the contents of their disciplines, nor the language and notation they use, but it is primarily the social and economic environment in which they work. It is a matter of who pays their salaries, what do the employers expect from them, how is their competence measured, what deeds will advance their careers. So, it was much easier to get the *academics* that do computational geometry to talk to *academics* that do pure geometry, than to the *engineers* that do computational geometry. I believe this cautionary message is something we must keep in mind when trying to bring together all people who do "computational graphics".

References

In the following list, the acronym "SOCG" stands for the proceedings of the Annual ACM Symposium on Computational Geometry.

References

[1] P. K. Agarwal: "A deterministic algorithm for partitioning arrangements of lines and its applications." SOCG 89, 11–22.

[2] P. K. Agarwal: "Partitioning arrangements of lines, I: An efficient deterministic algorithm." Discrete and Computational Geometry vol. 5 no. 5 (1990), 449–484.

[3] A. Aggarwal, H. Imai, N. Katoh, and S. Suri: "Finding k points with minimum diameter and related problems." J. of ALgorithms vol. 12 no. 1 (March 1991), 38–56.

[4] N. Alon, D. Haussler and E. Welzl: "Partitioning and geometric embedding of range spaces of finite Vapnik-Chervonenkis dimension." SOCG 87, 331–340.

[5] E. M. Arkin, R. Connelly, J. S. B. Mitchell: "On monotone paths among obstacles, with applications to planning assemblies." SOCG 89, 334–343.

[6] E. M. Arkin, D. Halperin, K. Kedem, J. S. B. Mitchell, N. Naor: "Arrangements of segments that share endpoints: single-face results." SOCG 91, 324–333.

[7] D. S. Arnon, G. E. Colins, and S. McCallum: "Cylindrical algebraic decomposition I: the basic algorithm." SIAM J. on Computing vol. 13 (9184), 856–877. As cited by [65].

[8] B. Aronov, J. Matoušek, M. Sharir: "On the sum of squares of cell complexities in hyperplane arrangements." SOCG 91, 307–313.

[9] F. Aurenhammer: "Jordan sorting via convex hulls of certain non-simple polygons." SOCG 87, 21–29.

[10] D. Avis and R. Wenger: "Algorithms for line transversals in space." SOCG 87, 300–307.

[11] C. L. Bajaj and I. Ihm: "Hermite interpolation using real algebraic surfaces." SOCG 89, 94–103.

[12] O. Barndorff-Nielsen and M. Sobel: "On the distribution of the number of admissible points in a vector sample." Theory of Probability and its Applications, vol. XI no. 2 (1966) 249–269.

[13] B. G. Baumgart: "A polyhedron representation for computer vision." AFIPS Natl. Comp. Conf. Proc. Vol. 44 (1975), 589–596.

[14] M. Ben-Or: "Lower bounds for algebraic computation trees." 15th STOC — Annual ACM Symposium on Theory of Computing (April 1983), 80–86.

[15] J. L. Bentley and Th. Ottmann: "Algorithms for reporting and counting geometric intersections." IEEE Trans. on Computers Vol C-28 No. 9 (Sep. 1979), 643–647.

[16] P. Bertolazzi and G. Di Battista: "On upward drawing testing of triconnected digraphs." SOCG 91, 272–280.

[17] D. Bienstock: "Some provbably hard crossing number problems." SOCG 90, 253–260.

[18] J. Bokowski, J. Richter, and B. Sturmfels: "Nonrealizability proofs in computational geometry." Discrete and Computational Geometry vol. 5 no. 4 (1990), 333–350.

[19] E. Brisson: "Representing geometric structures in d dimensions." SOCG 89, 218–227.

[20] S. E. Cappell, J. E. Goodman, J. Pach, R. Pollack, M. Sharir and R. Wenger: "The combinatorial complexity of hyperplane trnsversals." SOCG 90, 83–91.

[21] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir: "Lines in space: Combinatorics, algorithms, and applications." Proc. 21st STOC — Annual ACM Symp. on Theory of Computing (1989), 382–393.

[22] B. Chazelle, L. J. Guibas, and D. T. Lee: "The power of geometric duality." Proc. 24th FOCS — Annual IEEE Symp. on Foundations of Computer Science (1983), 217-225.

[23] B. Chazelle and M. Sharir: "An algorithm for generalized point location and its applications." J. of Symbolic Computation, vol. 10 no. 3& 4 (1990), 281–309.

[24] K. L. Clarkson: "Fast expected time and approximation algorithms for geometric minimum spanning trees." Proc. of the 16th STOC — Annual ACM Symp. on Theory of Computing (1984), 342–348.

[25] R. Cole, and M. Sharir: "Visibility problems for polyhedral terrains." J. of Symbolic Computation vol. 7 (1989), 11–30.

[26] C. Coullard and A. Lubiw: "Distance visibility graphs." SOCG 91, 289–296.

[27] H. Crapo and J. Ryan: "Scene analysis and geometric homology." SOCG 86, 125–132.

[28] J. C. Culberson and R. A. Reckhow: "Covering polygons is hard." 29th FOCS — Annual IEEE Symposium on Foundations of Computer Science (White Plains, 1988), 133–137.

[29] M. de Berg, D. Halperin, M. H. Overmars, J. Snoeyink, M. van Kreveld: "Efficient ray shooting and hidden surface removal." SOCG 91, 21–30.

[30] L. De Floriani and B. Falcidieno: "A hierarchical boundary model for solid object representation." ACM Transactions on Graphics vol. 7 no. 1 (January 1988), 42–60.

[31] F. Devai: "Quadratic bounds for hidden-line elimination." SOCG 86, 269–275.

[32] T. K. Dey: "Traingulation and CSG representation of polyhedra with arbitrary genus." SOCG 91, 364–372.

[33] G. Di Battista, R. Tamassia, and I. G. Tollis: "Area requirements and symmetry display in drawing graphs." SOCG 89, 51–60.

[34] D. Dobkin, L. Guibas, J. Hershberger, and J. Snoeyink: "An efficient algorithm for finding the CSG representation of a simple polygon." DEC Systems Research Center technical report 50(a,b) (September 1989).

[35] D. P. Dobkin and M. J. Laszlo: "Primitives for the manipulation of three-dimensional subdivisions." SOCG 87, 86–99.

[36] W. F. Eddy: "A new convex hull algorithm for planar sets." Manuscript (1977).

[37] H. Edelsbrunner: "Algorithms in Combinatorial Geometry." Springer-Verlag (1985).

[38] H. Edelsbrunner, L. J. Guibas, M. Sharir: "The upper envelope of piecewise linear functions: algorithms and applications." Discrete and Computational Geometry vol. 4 (1989), 311–336.

[39] H. Edelsbrunner and E. P. Mücke: "Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms." SOCG 88, 118–133.

[40] H. Edelsbrunner and E. Welzl: "On the maximal number of edges of many faces in an arrangement." Report F99, Institut für Informationsverarbeitung, Graz Technical Unversity, Austria (November 1982).

[41] G. Elber and E. Cohen: "Hidden curve removal for free-form surfaces." SIGGRAPH90, 95–104.

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

[43] R. Forrest: "Guest editor introduction to special issue on computational geometry." ACM Transactions on Graphics vol. 3 no. 4 (October 1984), 241–243.

[44] S. Fortune and V. Milenkovic: "Numerical stability of algorithms for line arrangements." SOCG 91, 334–341.

[45] R. J. Fowler, M. S. Paterson, and S. L. Tanimoto: "Optimal packing and covering in the plane are NP-complete." Information Processing Letters vol. 12 (1981), 133–137.

[46] Wm. R. Franklin and M. S. Kankanhalli: "Parallel object-space hidden surface removal." SIGGRAPH90, 87–94.

[47] Wm. R. Franklin: "Polygon properties calculated from the vertex neighborhoods." SOCG 87, 110–118.

[48] J. Friedman, J. Hershberger, and J. Snoeyink: "Compliant motion in a simple polygon." SOCG 89, 175–186.

[49] M. R. Garey, R. L. Graham, and D. S. Johnson: "The complexity of computing Steiner minimal trees." SIAM J. on Applied Mathematics, vol. 32 no. 4 (June 1977), 835–859.

[50] G. Gentili and M. A. O'Connor: "Rational geometry in space." J. of Symbolic Computation vol. 11 no. 3 (1991), 213–229.

[51] J. E. Goodman and R. Pollack: "Upper bounds for configurations and polytopes in $\Re^d$." Discrete and Computational Geometry vol. 1 no. 3 (1986), 219–228.

[52] D. H. Greene and F. F. Yao: "Finite-resolution computational geometry." Manuscript, Xerox Palo Alto Research Centers (1985).

[53] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and M. Tarjan: "Linear time algorithms for visibility and shortest paths." SOCG 86, 1–13.

[54] L. J. Guibas and R. Seidel: "Computig convolutions by reciprocal search." SOCG 86, 90–99.

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

[56] L. J. Guibas and J. Stolfi: "Ruler, compass and computer: The design and analysis of geometric algorithms." Tech. Rep. #37, DEC Systems Research Center (February 1989).

[57] C. Gunn and D. Maxwell: "Not Knot." [Video] Produced by the Geometry Center, U. of Minnesota. Jones and Bartlett, Boston (1991).

[58] D. Halperin: "On the complexity of a single cell in certain arrangements of surfaces in 3-space." SOCG 91, 314–323.

[59] P. Hanrahan: "A homogeneous geometry calculator." Tech. memo 3-D no. 7, Computer Graphics Laboratory, New York Ins. of Technology (September 1984).

[60] S. Hart and M. Sharir: "Nonlinearity of Davenport--Schinzel sequences and of a generalized path compression scheme." Tech. Rep. 84-011, Comp. Sci. Inst., Tel-Aviv Univ. (August 1984). Also Combinatorica vol. 6 (1986), 151–177.

[61] D. Haussler and E. Welzl: "Epsilon-nets and simplex range queries." SOCG 86, 61–71.

[62] J. Hershberger: "Finding the upper envelope of n line segments in $O(n \log n)$ time." Information Processing Letters vol. 33 (1989), 169–174.

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

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

[65] E. Kaltofen: "Computing the irreducible real factors and components of an algebraic curve." SOCG 89, 79–87.

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

[67] M. J. Katz, M. H. Overmars, and M. Sharir: "Efficient hidden surface removal for objects with small union size." SOCG 91, 31–40.

[68] Y, Ke and J. O'Rourke: "Moving a ladder in three dimensions: upper and lower bounds." SOCG 87, 137–146.

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

[70] D. E. Knuth: "The Art of Computer Programming." Addison-Wesley (1969).

[71] D. E. Knuth: "Axioms and Hulls." Manuscript, Computer Science Dept., Stanford Unversity (1991).

[72] D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes: "Constructive solid geometry for polyhedral models." Proceedings of SIGGRAPH 86, ACM Computer Graphics vol. 20 no. 4 (1986), 161–170.

[73] J.-C. Latombe: "Robot Motion Planning." Kluwer Academic Press, Boston (1991).

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

[75] D. Leven and M. Sharir: "An efficient and simple motion planning algorithm for a ladder amidst polygonal barriers." J. of ALgorithms vol. 8 no. 2 (June 1987), 192–215.

[76] Z. Li and V. Milenkovic: "Constructing strongly convex hulls using exact or rounded arithmetic." SOCG 90, 235–243.

[77] P. Lienhardt: "Subdivisions of n-dimensional spaces and n-dimensional generalized maps." SOCG 89, 228–236.

[78] A. Lingas: "Advances in minimum-weight triangulation." Dissertations no. 97, Software Systems Research Center, Linköping Univ. (Sweden, 1983).

[79] S. R. Maddila and C. K. Yap: "Moving a polygon around a corner in a corridor." SOCG 86, 187–192.

[80] M. Mäntylä: "Boolean operations of 2-manifolds through vertex neighborhood classification." ACM Transactions on Graphics vol. 5 no. 1 (January 1986), 1–29.

[81] J. Matoušek: "Construction of epsilon nets." SOCG 89, 1–10; Discrete and Computational Geometry vol. 5 no. 5 (1990), 427–448.

[82] J. Matoušek, R. Seidel, and E. Welzl: "How to net a lot with a little: small epsilon-nets for disks and halfspaces." SOCG 90, 16–22.

[83] J. Matoušek and E. Welzl: "Good splitters for counting points in triangles." SOCG 89, 124–130.

[84] M. McKenna: "Worst-case optimal hidden-surface removal." ACM Transactions on Graphics vol. 6 no. 1 (January 1987), 19–28.

[85] N. Megiddo: "Linear-time algorithms for linear programming in $\Re^3$ and related problems." SIAM J. on Computing vol. 12 (1983) 759–776.

[86] N. Megiddo and E. Zemel: "An $O(n \log n)$ randomizing algorithm for the weighted Euclidean 1-center problem." J. of ALgorithms vol. 7 no. 3 (September 1986), 358–368.

[87] V. Milenkovic and L. R. Nackman: "Finding compact coordinate representations for polygons and polyhedra." SOCG 90, 244–252.

[88] J. S. B. Mitchell and C. Papadimitriou: "The weighted region problem." SOCG 87, 30–38.

[89] D. E. Muller and F. P. Preparata: "Finding the intersection of two convex polyhedra." Theoretical Computer Science Vol. 7 (1978), 217–236.

[90] K. Mulmuley: "An efficient algorithm for hidden-surface removal." SIGGRAPH89, 379–388.

[91] C. Ó'Dúnlaing and C. Yap: "A "retraction" method for planning th e motion of a disk." J. of ALgorithms vol. 6 no. 1 (March 1985), 104–111.

[92] J. K. Ousterhout: "Corner Stitching: A Data Structuring Technique for VLSI Design Tools." Manuscript (1985?).

[93] T. Ottmann, G. Thiemt, and C. Ullrich: "Numerical stability of geometric algorithms." SOCG 87, 197–207.

[94] J. Pach and G. Woeginger: "Some new bounds for epsilon-nets." SOCG 90, 10–15.

[95] M. S. Paterson and F. F. Yao: "Point retrieval for polygons." J. of ALgorithms vol. 7 no. 3 (September 1986), 441–447.

[96] M. S. Paterson and F. F. Yao: "Binary partitions with applications to hidden-surface removal and solid modeling." SOCG 89, 23–32.

[97] M. A. Penna and R. R. Patterson: "Projective Geometry and its Applications to Computer Graphics." Prentice-Hall (1986).

[98] D. Peterson: "Halfspace representations of extrusions, solids of revolution, and pyramids." Sandia National Laboratories report SAND 84-0572 (1984).

[99] R. Pollack and R. Wenger: "Necessary and sufficient conditions for hyperplane transversals." SOCG 89, 152–155.

[100] P. Rosenstiehl and R. E. Tarjan: "Rectilinear planar layouts and bipolar orientations of planar graphs." Discrete and Computational Geometry vol. 1 no. 4 (1986), 343–353.

[101] J. O'Rourke: "An on-line algorithm for fitting straight lines between data ranges." Communications of the ACM vol. 24 (1981), 574–578.

[102] J. Ruppers and R. Seidel: "On the difficulty of tetrahedralizing 3-dimensional non-convex polyhedra." SOCG 89, 380–392.

[103] D. Salesin: "Epsilon Geometry: Building Robust Algorithms from Imprecise Computations." Ph.D. Thesis, Stanford University (January 1991).

[104] H. Samet and R. E. Webber: "Storing a collection of polygons using quadtrees." ACM Transactions on Graphics vol. 4 no. 3 (July 1985), 182–222.

[105] L. A. Santaló: "Integral geometry and geometric probability." Addison-Wesley (1976).

[106] J. T. Schwartz and M. Sharir: "On the two-dimensional Davenport-Schinzel problem." J. of Symbolic Computation, vol. 10 no. 3& 4 (1990), 371–393.

[107] T. W. Sederberg and A. K. Zundel: "Scan line display of algebraic surfaces." SIGGRAPH'89 Conference Proceedings, ACM Computer Graphics vol. 23 no. 3 (July 1989), 147–156.

[108] R. Seidel: "On the number of faces of higher-dimensional Voronoi diagrams." SOCG 87, 181–185.

[109] R. Seidel: "Linear programming and convex hull smade easy." SOCG 90, 211–215.

[110] M. I. Shamos: "Computational Geometry." Ph. D. Thesis, Yale University (December 1977).

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

[112] M. I. Shamos and F. P. Preparata: "Computational Geometry: An Introduction." Springer-Verlag (1985).

[113] M. Sharir and E. Ariel-Sheffi: "On the piano movers' problem VI: various decomposable two-dimensional motion planning problems." Comm. in Pure and Applied Mathematics, vol 37 (1984), 479–493.

[114] S. Sifrony and M. Sharir: "A new efficient motion planning algorithm for a rod in polygonal space." SOCG 86, 178–186.

[115] J. Snoeyink: "A trivial knot whose spanning disks have exponential size." SOCG 90, 139–147.

[116] J. Snoeyink and J. Hershberger: "Sweeping arrangements of curves." SOCG 89, 354–363.

[117] J. Stolfi: "Oriented Projective Geometry: a Framework for Geometric Computations." Academic Press (1991).

[118] K. Sugihara: "Mathematical structures of line drawings of polyhedra: towards man-machine communication by means of line drawings." Research note RNS 81-02, Fac. Eng., Nagoya University (1981). As cited by [27].

[119] K. J. Supowit, E. M. Reingold, and D. A. Plaisted: "The minimum traveling salesman problem and minimum matching in the unit square." SIAM J. on Computing Vol. 12 No 1 (Feb. 1983), 144–156.

[120] G. Swart: "Finding the convex hull facet by facet." J. of ALgorithms vol. 6 no. 1 (March 1985), 17–48.

[121] R. Terada: "Fast algorithms for NP-hard problems which are optimal or near-optimal with probability one." Ph. Thesis, CS Dept., U. of Winsconsin at Madison (1979).

[122] T. Tokuyama and J. Nakano: "Geometric algorithms for a minimum cost assignment problem." SOCG 91, 262–271.

[123] J. E. Warnock: "A Hidden-Surface Algorithm for Computer-Generated Halftone Pictures." Computer Science Department, University of Utah, TR 4-15 (June 1969).

[124] R. Wenger: "A generalization of Hadwiger's transversal theorem to intersecting sets." Discrete and Computational Geometry vol. 5 no. 4 (1990), 383–388.

[125] W. Whiteley: "Two algorithms for polyhedral pictures." SOCG 86, 142–149.

[126] A. Wiernik and M. Sharir: "Planar realizations of nonlinear Davenport-Schinzel sequences by segments." Discrete and Computational Geometry vol. 3 no. 1 (1988), 15–48.

[127] C. K. Yap: "Symbolic treatment of geometric degeneracies." J. of Symbolic Computation, vol. 10 no. 3& 4 (1990), 349–370.

[128] A. C. Yao: "On constructing the minimum spanning trees in k-dimensional spaces and related problems." SIAM J. on Computing Vol. 11 (1982), 721–736.