What is Computational Geometry?

Godfried T. Toussaint

Computational Geometry Laboratory
School of Computer Science
McGill University
Montreal, Quebec, Canada

1. Introduction

Consider the point p and the line l arranged in the plane as illustrated in Fig. 1.1. The following question immediately comes to mind. Does the point p lie on, above or below l? This question asks for the answer to a geometric property of the given set of geometric objects $\{p, l\}$. In a narrow sense *computational geometry* is concerned with computing geometric properties of sets of geometric objects in space such as the simple *above/below* relationship of a given point with respect to a given line. In a broader sense computational geometry is concerned with the *design* and *analysis* of algorithms for solving geometric problems. In a deeper sense it is the study of the inherent *computational complexity* of geometric problems under varying models of computation. In this latter sense it pre-supposes the determination of which geometric properties are computable in the first place.

In the above simple problem let us assume that the point p is specified in terms of its x and y coordinates (x_p, y_p) and that the line l is given by the equation $y = ax + b$, where a and b are the slope and y-intercept, respectively. For simplicity of discussion assume that a is not equal to infinity or zero, i.e., the line is non-vertical and non-horizontal. To solve this problem it suffices to compute the intersection point of the vertical line through p with l. Call this point z with coordinates (x_z, y_z) and refer to Fig. 1.1. Then $x_z = x_p$ and y_z may be calculated using the equation $y_z = ax_p$ +

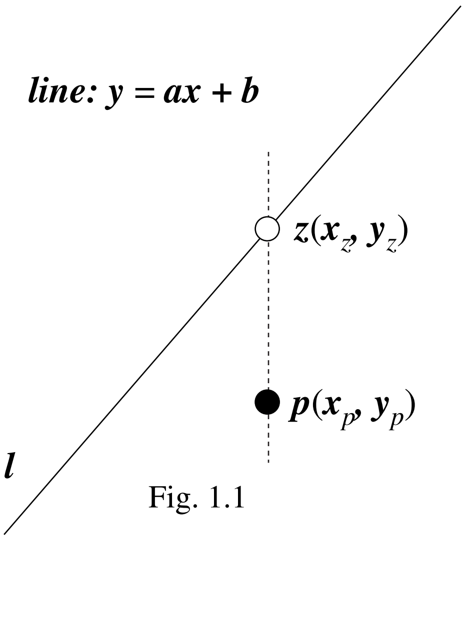

Fig. 1.1

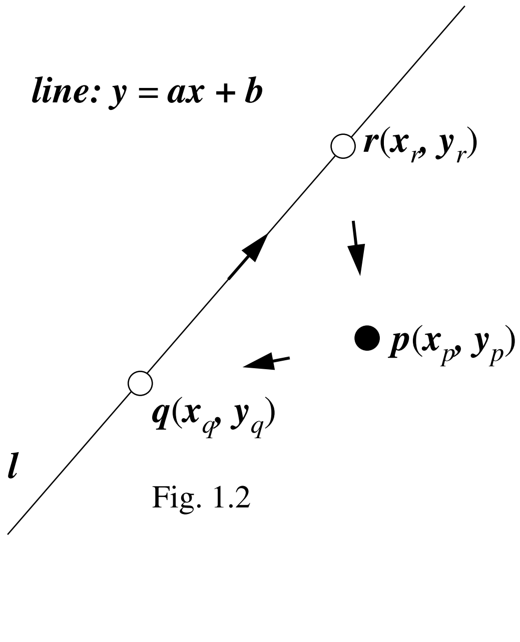

Fig. 1.2

b. If $y_z > y_p$ then p lies below l, if $y_z < y_p$ then p lies above l and if $y_z = y_p$ then p lies on l.

The above algorithm is but one approach to solve this problem. To illustrate a different method let us assume that in our library of basic computational tools we already have available a sub-routine that calculates the area of a triangle when the input triangle is represented by the x and y coordinates of its three vertices. Then we could solve the problem in the following manner. Let l and $p(x_p, y_p)$ denote the line and point, respectively, as before and refer to Fig. 1.2. First we identify two new points $q(x_q, y_q)$ and $r(x_r, y_r)$ on the line l. The three points q, r, p define a triangle. We may compute the area of this triangle using the following formula [Kl39], [St86]:

$$Area = \frac{1}{2} \cdot [x_q(y_r - y_p) + x_r(y_p - y_q) + x_p(y_q - y_r)]$$

Surprisingly, the area will in fact tell us whether p lies below, above or on the line l. Clearly if the area is zero p lies on l. Consider now what happens when the area is not zero noting that in each of the three terms summed in the above formula the indices q, r, and p occur in a clockwise order with respect to the triangle illustrated in Fig.1.2. Actually the expression given above for calculating the area of a triangle does a lot more than that. It calculates the *signed* area, i.e., the area has a *negative* sign if the order of the indices in the formula corresponds to a *clockwise* order of the vertices in the triangle and the area has a *positive* sign if the order of the indices in the formula corresponds to a *counter-clockwise* order of the vertices in the triangle. Furthermore, in order to solve our problem we are interested in the *sign* rather than the absolute value of the area. Finally, note in Fig. 1.2 that if p lies below l the vertices of the triangle appear in clockwise order in the formula whereas if p lies above l the vertices of the triangle appear in counter-clockwise order in the formula. Therefore, if the area is positive we may conclude that p lies above l and if the area is negative that p lies below l.

We have exhibited two very different algorithms for solving one and the same geometric problem. At a low level computational geometry is concerned with the comparative study of fundamental algorithms such as these with the goal of determining, in different computational contexts, which algorithms run faster, which require less memory space and which are more robust with respect to numerical errors. However, the discipline of computational geometry actively pursued in computing science today is concerned mostly with the design and analysis of algorithms at a conceptually much richer level than that described above.

2. Classical Constructive Geometry *versus* Modern Computational Geometry

2.1 Introduction

Today computational geometry is often billed as a new discipline in computing science which many computing scientists would say was born either with the Ph.D. thesis of Michael Shamos at Yale University in 1978 [Sh78], or perhaps his earlier paper on geometric complexity [Sh75]. Others would say it began ten years earlier with the Ph.D. thesis of Robin Forrest at Cambridge University in 1968 [Fo68] or perhaps with his subsequent papers on computational geometry [Fo71], [Fo74]. Still others would claim it began with Minsky and Papert's formal investigation of which geometric properties of a figure can and cannot be recognized (computed) with a

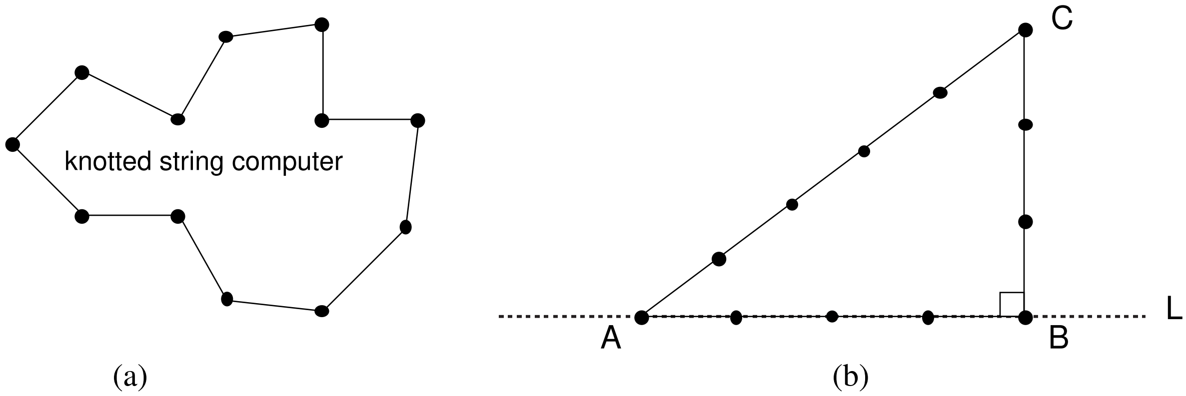

Fig. 2.1 Egyptian computer and algorithm used 4000 years ago for constructing a right angle.

variety of *neural network* models of computation [MP69]. Many tend to associate computer science disciplines with their appearance in the context of the advent of the electronic digital computer, oblivious as to whether such a discipline may have existed before. Computational geometry is a case in point. We show here that computational geometry (the various disciplines of present-day computing science mentioned above) has been around for more than 2600 years starting with the Greeks. In this historical context we illustrate the new features that computing science has added to this developing field during the past thirty years. The electronic digital computer, while having had a strong influence on the field, is an instrument used to give technological wings to computing science. The essence of computing science is constructive (i.e., computational, algorithmic) mathematics independent of the technology of the machines used to implement the algorithms that it yields. To quote the Dutch computing scientist Edsger Dijkstra, "*computer* science should be called *computing* science, for the same reason why surgery is not called knife science." Let us then consider what kind of geometry was done with some computers used in the past well before the introduction of the twentieth century electronic digital computer.

2.2 Knotted Strings, Rulers, Compasses and the Electronic Digital Computer

If you did not have a protractor and you were asked to construct a right angle at a given point on a given line you may remember how to do it with straight edge and compass, a popular computer of the ancient Greeks. Let us assume however that all you have available is some string. The following computer, inexpensive and easily manufactured, affords the implementation of a simple algorithm the correctness of which is based on a sound theory of computational geometry. Take 12 equally long pieces of string and tie them together with knots end-to-end so that they form a "necklace" with 12 knots as in Fig. 2.1 (a). The construction of this *necklace computer* (or knotted string computer) is now finished so let us turn to the algorithm.

Step 1: Take any knot, call it B, and attach it to the point on the line L where it is desired to erect the line at right angles to L, as illustrated in Fig. 2.1 (b).

Step 2: Take the fourth knot away from B (in either direction) on the necklace, call it A, and place it on the line L pulling the string taut so that A is as far as possible from B.

Step 3: Now take the third knot in the other direction away from B along the loose remaining portion of the necklace and pull it away from A and B until both the AC and BC portions of the necklace are taut.

The position of the knots B and C now indicate two points on a line perpendicular to L at B. The line perpendicular to L may now be readily drawn with a straight edge through the points B and C. This is how the Egyptian architects worked more than 4000 thousand years ago [Du90]. Thus the Egyptians, thousands of years earlier than the Greeks, already possessed a rudimentary knowledge of the *converse* of the theorem of Pythagoras, the latter stating that if ABC is a right angle then the square on AB plus the square on BC is equal to the square on AC.

In the third century B.C. Greek mathematicians in Alexandria such as Euclid were heavily involved with computational geometry. Even two hundred years before that geometry was already flourishing in the *Hecademia*, Plato's school in Athens [HT85]. Although Plato was not a mathematician but was in the business of awarding five-year philosophy degrees he felt that geometry was so essential for a sound mind that as a prerequisite to study philosophy his students were required to study geometry for ten years! In fact, the entrance to his school displayed a sign that read, *"Let no one who is un-acquainted with geometry enter here."* So you may ask, is there any difference between the geometry of the ancient Greeks and modern computational geometry? In essence not much! Modern computational geometry resembles closely that work of the Greeks concerned with determining constructions for solving geometric problems. As we shall see many of the key issues addressed in computational geometry today were also of concern to the ancient Greeks. The first difference worth noting is that they dealt with inputs of small size whereas today, due to the necessity of certain applications and the speed offered by electronic digital computers, we deal with inputs with a large number of objects. A concrete example from the field of *facility location theory*, a central concern to transportation and management science, will make the point.

2.3 The Size of the Input to an Algorithm

2.3.1 Heron's Theorem

Consider two farms in the middle of a field in Saskatchewan on the same side of a nearby highway. People are continually travelling from one farm to the other and during their trips would like to take a detour to stop at the highway to obtain supplies. A supply company would like to build a supply store (facility) somewhere along the highway so that the distance the farmers must travel in going from one farm to the other, via at the supply store, is minimized. Where should the supply company build its store and what is the path that the farmers should follow? This problem, which I will call the *highway facility location* problem, was solved by the Alexandrian mathematician Heron around the year A.D. 100. Four hundred years earlier Euclid had laid down in his book the *Catoptrica* the fundamental laws of reflection of light rays from a mirror:

Rule 1: The plane of incidence of the light ray coincides with the plane of its reflection

Rule 2: The angle of incidence of the light ray equals its angle of reflection

Heron was extending Euclid's results and had explained the laws of reflection in simpler more fundamental terms: *light must always travel via the shortest path*. Geometrically we can state the above facility location problem as follows. Let l be a straight line as before (highways in Saskatchewan are well modeled by straight lines) and let p and q denote two points (representing

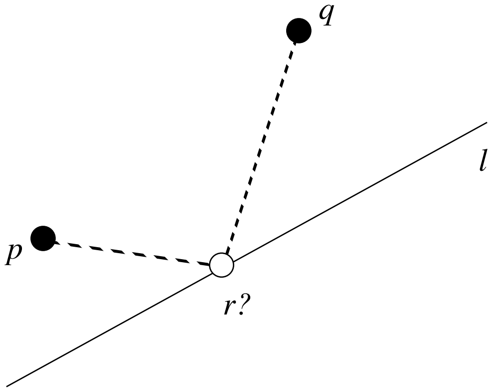

Fig. 2.2 The highway facility location problem.

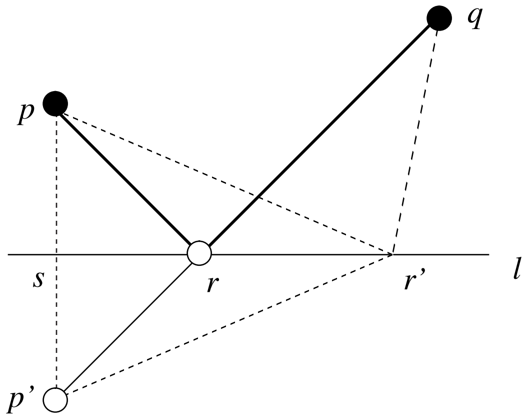

Fig. 2.3 Illustrating the proof of Heron's Theorem.

the farm houses) on the same side of l and refer to Fig. 2.2. Let d(a,b) denote the Euclidean distance between points a and b. Find the point r on l so that $d(p,r) + d(q,r)$ is a *minimum*. Let p' be the mirror image of p reflected across l and refer to Fig. 2.3. Then the solution r lies at the intersection of l with the line through p' and q. This is known as *Heron's Theorem* and often referred to as the *reflection principle*. It has applications to solving a variety of geometric problems [Ka61], one of the most recent being the efficient computation of the shortest tour inside a simple polygon that visits at least one point on every edge of the polygon (*The Aquarium Keeper's Problem*) [Cz91]. To see the correctness of Heron's claim consider any other location r' and assume it is to the right of r (a similar argument applies when r' lies to the left of r). By construction, the line l is the locus of points equidistant from p and p'. Therefore $d(p,r) = d(p',r)$ and $d(p,r') = d(p',r')$. Therefore the length of the path prq is equal to the length of the path $p'rq$, and the length of the path $pr'q$ is equal to the length of the path $p'r'q$. But by the *triangle inequality rule* which states that the sum of two sides of a triangle is greater than the third, it follows that the path $p'r'q$ is longer than the path $p'rq$. Therefore the path $pr'q$ is longer than the path prq establishing that indeed r is the optimal location for the facility.

It can be easily verified by the reader that Heron's problem can be easily solved with the straight edge and compass computer.

2.3.2 The Minimax Facility Location Problem

In the standard version of the *minimax* facility location problem [FW74], [BS67] we are given a set of n points in the plane representing customers, plants to be serviced, schools, markets, towns, distribution sites or what have you, depending on the context in which the problem is embedded, and it is desired to determine the location X (find another point in the plane) where a facility (service, transmitter, dispatcher, etc.) should be located so as to minimize the distance from X to its furthest customer (point). Such a *minimax* criterion is particularly useful in locating emergency facilities, such as police stations, fire-fighting stations and hospitals where it is desired to minimize the worst-case (or maximum) response time [TSRB71]. When transportation between the facility and the customers can be carried out in a straight line path this problem has an elegant and succinct geometrical interpretation: find the smallest circle that encloses a given set of n points

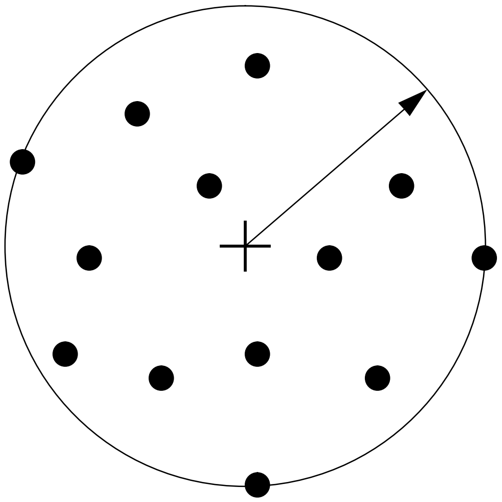

Fig. 2.4 The smallest circle enclosing a set of points.

(see Fig. 2.4). The center of this circle (the cross in Fig. 2.4) is precisely the location of X. The radius of the circle corresponds to the worst-case response time, the maximum distance that must be travelled, the minimum power required of the transmitter, etc. depending on the context.

In this type of problem, typical of those considered today in computational geometry, the size of the input to the problem, i.e., the number n of geometric objects considered can be in the hundreds, thousands, or millions depending on the particular application at hand. Solving such problems by manual *straight-edge-and-compass* constructions as was done in the days of Euclid would clearly have been out of the question and hence such problems were not even attempted. This then is one difference between computational geometry then and now. In the days of the Greeks the number of objects considered in the input to an algorithm was very small compared to today. However it should be noted that this is a *practical* and not an *in-principle* limitation of the methods of ancient computational geometry.

The reader is referred to [LW86] and [RT90] for further details on the fast-growing literature concerning the application of computational geometry to solving facility location problems in transportation and management science.

2.4 The Dimension of a Problem

The highway and minimax facility location problems considered above, like most of the problems considered in say Euclid's *Elements*, were planar, i.e., two dimensional. The Greek geometers limited themselves to problems in two and three dimensions. Today, on the other hand, computational geometry is not limited to the exploration of low dimensional problems. It is just as natural to consider sets of n points in d-dimensional space where $d > 3$ and to ask what is the smallest d-dimensional hyper-sphere that contains the n points. This is another difference between computational geometry then and now and reflects to a large extent the applications that require the development of such tools. The Greeks were concerned with the *physical* world which is well modelled by three Euclidean dimensions. Today on the other hand technologies such as pattern recognition by machine [To80] and robotics [LPS88], [SY87], [SSH87] provide us with many geometrical problems in non-Euclidean and higher dimensional "worlds."

2.5 Models of Computation

2.5.1 Primitive Operations

A *model of computation* is a description or specification of an abstract device, machine or computer in terms of what primitive operations are allowed to be performed on the input data and what is their cost. A model of computation is very useful in classifying and ordering problems and algorithms in terms of their complexity and resulting execution time. Therefore the specification and investigation of such models is a central concern in modern computational geometry.

One of the most popular models of computation used in computational geometry is the *Real RAM (R*andom *A*ccess *M*achine) [AHU74], [PS85]. In the basic version of this model the input data are specified in terms of real numbers and we assume that a real number is stored in a storage location that uses one unit of memory. We also assume that the following operations are primitive and each can be executed in one unit of time:

1. The arithmetic operations (addition, subtraction, multiplication and division).
2. Comparisons between real numbers (=, <, >).

Very often a more powerful RAM is assumed by also allowing additional operations on the list of primitives such as the computation of logarithms, *k-th* roots or trigonometric functions.

2.5.2 Computing Power and the Equivalence of Machines

Specifying a model of computation allows us to ask many interesting questions. For example we may ask which problems can be computed at all with a specified machine in the first place. We can also ask that for a given problem and machine we compare all the known algorithms (for solving the given problem) in terms of the number of primitive operations and units of memory they require. We illustrate these ideas with a concrete example below.

Consider another facility location problem similar to the highway facility location problem solved by Heron of Alexandria that was illustrated in Fig. 2.2. In Heron's problem it was required to compute the shortest path between two given points via a given line. Instead, we now consider two given lines A and B and a point p as illustrated in Fig. 2.5 and we are required to find the shortest *straight* line ab bridging the lines A and B via the point p. This problem is one of the corner stones of a wide class of problems currently under investigation in computational geometry referred to as *stabbing* or *transversal* problems. In a typical instance of the stabbing problem we are given a collection of objects and we ask whether there exists a line that intersects all the objects in the collection. Furthermore, if such lines exist we ask that the shortest line segment intersecting all the objects be reported [BT91], [RT90].

An obvious question now is whether this problem can be solved with the basic real RAM. The answer is negative. However, if we ask this question about a more powerful machine by extending the capability of the basic real RAM by adding to its list of allowed primitive operations the computation of *k-th* roots then, as we shall see, the answer is in the affirmative. This leads us naturally into questions regarding the relative computational power of different machines and the

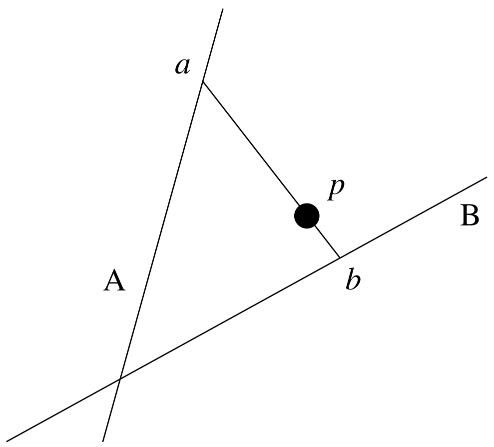

Fig. 2.5 The shortest straight bridge a,b between two lines via a point p.

equivalence of machines as well as equivalence classes of problems, all burning issues in modern computational geometry and discussed in more detail in some of the papers in this special issue. Readers may suspect these concerns to be unique to the electronic digital computer era. However, these questions were just as hot in the days of Euclid.

In classical constructive geometry investigators have also been concerned with defining the *models of computation*, i.e., the characteristics of the machine that will execute the algorithms. Typical machines that have been used in the past starting with Euclid himself include (1) the *straight edge*, (2) the *ruler*, (3) the *collapsing compass*, (4) the *compass*, (5) the *fixed-aperture compass*, (6) the compass with aperture *bounded from above*, and (7) the compass with aperture *bounded from below* just to name a few [Sm61], [Ho70], [CR81], [Ko86]. The reader is referred to the delightful book by William Ransom for a detailed treatment of what can and cannot be computed with a further variety of ancient computing machines [Ra60]. The *collapsing compass* deserves elaboration here. With the regular compass one can open it, lock it at a chosen aperture and lift it off the work-space to some other location to draw a circle with the chosen radius. This operation cannot be done with a collapsing compass. The collapsing compass is, like the other machines, an *idealized* abstract machine (in the same spirit as the Real RAM) which allows the compass to be opened to a chosen radius and a circle drawn, but no distance can be *transferred*. It is as if when the compass is lifted off the work-space it collapses and thus erases any trace of the previous aperture made. More complicated machines can be obtained by combining sets of simple machines. For example in Euclid's *Elements* he selected the *straight edge* and *collapsing compass* (the combination of machines (1) and (3)) as his model of computation. Attempts have also been made to specify the primitive operations allowed with each type of machine [Le02] and to design constructions that require fewer operations than did Euclid's original constructions.

Another active area of research has been to analyze and compare different machine models in terms of their computational power [Ho70], [CR81], [Av87], [Av90]. For example, in 1672 Georg Mohr [Mo1672] and in 1797 the Italian geometer Lorenzo Mascheroni [Ma1797] independently proved that any construction that can be carried out with a straight edge and a compass can be carried out with a compass alone and Jacob Steiner proved in 1833 that the straight edge is equivalent in power to the compass if the former is afforded the use of the compass once [SA48]. To remind the reader that the *straight edge* and *compass* are not yet obsolete computers we should point out that the Mohr-Mascheroni result was strengthened as recently as in 1987 by Arnon Avron [Av87] at the University of Tel Aviv.

The earliest theorem concerning the equivalence of models of geometric computation is due to Euclid in his second proposition of Book I of the *Elements* in which he establishes that the *collapsing compass* is equivalent in power to the regular *compass*. Therefore in the theory of equivalence of the power of models of computation, Euclid's second proposition enjoys a singular place. Here we mention as an aside that like much of Euclid's work and particularly his constructions involving many cases, the correctness of his proof of his second proposition has received a great deal of criticism over the past twenty centuries. In [To92a], [To92b] it is argued that it is Euclid's commentators and translators that are at fault and that Euclid's original algorithm and proof of correctness are beyond reproach.

Let us return however to the problem of finding the shortest *straight* line *ab* bridging the lines A and B via the point *p* as illustrated in Fig. 2.5 (the *shortest bridge* problem). Recall that this problem cannot be solved with the basic Real RAM. Also recall that the related highway facility location problem was solved using the straight edge and compass by Heron of Alexandria in 100 A.D. We may ask if our shortest bridge problem can also be solved with straight edge and compass. Actually the problem of computing the shortest bridge and the problem of whether it can be computed with straight edge and compass have long and interesting histories [He81]. The first computing scientist to find a characterization of the solution was Philon of Byzantium circa 100 B.C. and the shortest such line has come to be known as the *Philo Line* named after him [Ev59]. More interesting is the reason why Philon was interested in solving this problem with the straight edge and compass. Three problems that the Greeks pursued with great passion were whether the straight edge and compass could be used to (1) trisect an angle, (2) square a circle and (3) double a cube. In problem (2) we are given a circle and asked to construct a square with the same area as the circle. The last of these problems asks for constructing a cube with twice the volume of a given cube. In planar terms, given a line segment of length x on the plane, representing the side of the given cube (with volume x^3), construct a line segment of length y, representing the side of the desired cube (with volume y^3), i.e., such that $y^3 = 2x^3$.

Solutions to this problem existed for hundreds of years before Philon's time but using other types of machines. For example Plato designed a computer to double the cube that resembled the instrument that a shoe maker uses to measure the length of the foot [CR41]. But the most fashionable computing device at the time, due no doubt to the impact of Euclid's *Elements*, was the straight edge and compass and no one before the time of Philon could solve this problem with the straight edge and compass. With a brilliant reduction proof (typical of the problem reduction techniques used in modern computational geometry) Philon showed that finding the shortest bridge (Philo Line) is equivalent to doubling the cube and thus he set off to try to find a straight edge and compass solution to the shortest bridge problem. In fact other investigators reduced other problems as well to that of doubling the cube so that an entire class of problems existed such that if any one of them could be solved with straight edge and compass then all of them could be. Thus the ancient Greeks developed a theory of complexity classes in much the same way that modern computer scientists are developing complexity theory [GJ78]. However, Philon never found a solution. In the 17th century Isaac Newton turned his attention to Philon's problem, generalized it, and found a different characterization of it. He did not however find a straight edge and compass solution. In fact no one ever found a straight edge and compass solution because it does not exist. One cannot double the cube with a straight edge and compass but it was not until the advent of modern algebra in the 19th century that this was proved. Because of Philon's reduction theorem it follows that the shortest bridge also cannot be computed with the straight edge and compass. It is interesting that a problem of such importance in 100 B.C. should form the cornerstone of the 20th century shortest transversal algorithms [BT91] and that the 20th century real RAM should have the same computational limitation with respect to this problem as the straight edge and compass. We remark here that the characterization of the Philo line used in [BT91] and [BET91] is different from both

Philon's and Newton's.

Finally, note that doubling the cube is equivalent to solving the equation $y^3 = 2x^3$ mentioned above. This equation can be written as $y = (2)^{1/3}x$. This is why Philon's problem can indeed be solved with the extended version of the Real RAM in which cube roots are allowed in its list of primitive operations.

2.5.3 The Complexity of Algorithms

One of the most fundamental and useful geometric structures that has received a great deal of attention in computational geometry is the *convex hull* of a set. In fact, there are even those who claim that computational geometry began with the classic 1972 paper by Ron Graham on computing the convex hull of a finite set of points in the plane [Gr72]. The convex hull of a set *S* is the smallest convex set containing *S*. Fig. 2.6 (a) illustrates the convex hull of a set of points. Consider the points as nails sticking out of a wooden board. Then, intuitively, you may think of the boundary of the convex hull as the shape taken by an elastic band stretched around the entire set of nails. When *S* is a set of points, the boundary of the convex hull of *S* is a convex polygon consisting of edges connecting some carefully selected pairs of points of *S*. These special pairs are straight forward to characterize (see Fig. 2.6 (b)). A pair contributes an edge to the boundary of the convex hull if, and only if, the line *L* through the pair divides the plane into two regions (one on each side of *L*) such that one of these regions contains no points of *S*. In Fig. 2.6 (b) A,B is such a pair whereas C,D is not.

This characterization leads to a conceptually simple algorithm for computing the convex hull of *S* that uses the algorithm discussed in the introduction for determining if a point is above or below a line. Assume for simplicity of discussion that the *n* points of *S* are specified by their cartesian coordinates as real numbers and that no two points have the same *x* or *y* coordinates. All we have to do is to consider each pair of points separately and compute the line determined by the two points. If all the remaining *n*-2 points all lie above (or all lie below) the computed line then the pair determines an edge of the boundary of the convex hull. Finally all the convex hull edges are concatenated to form a convex polygon.

A natural question now is: how powerful a computer do we need in order that it be able to execute this algorithm? The reader may easily verify that the basic real RAM will suffice.

Another natural question is: for a set *S* of size *n* how many primitive operations are required by the basic real RAM to compute the convex hull of *S* with the above algorithm? In other words, what is the *complexity* of the algorithm? This question is the same as the question Lemoine [Le02] asked for the straight edge and compass construction algorithms of Euclid. Lemoine called the number of primitive operations used in the execution of the algorithm the *simplicity* of the construction. If *S* consists of a given configuration of say 10 points we could just count the number of operations performed by the RAM as the algorithm is executed and report this absolute number as the complexity of the algorithm for these 10 points. This is how the ancient Greeks measured the complexity of their constructions. However, because today the size *n* of the input can be very large and can vary a lot from problem to problem it is more convenient to report the complexity of an algorithm as a function of *n*. This seemingly minor detail is another difference between computational geometry then and now. The philosophically oriented leisure conscious Greeks were content

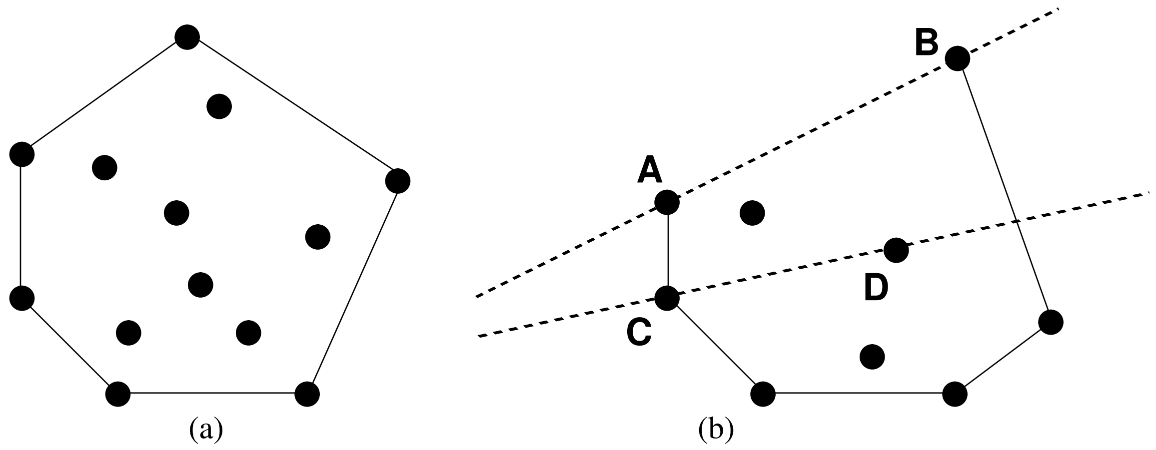

Fig. 2.6 (a) The convex hull of a set of points. (b) Characterizing the edges of the boundary of the convex hull.

with finding one algorithm to solve a problem. Today's society hooked on technological progress and speed wants faster and faster algorithms.

Consider for example the convex hull algorithm described above. For a given pair of points we compute the line passing through them using say k_1 primitive operations. Then we test *n-2* points to determine if each of them lies above or below the line. Assume that each such point-line test takes no more than k_2 operations. Now we must repeat this procedure for every pair of points in *S*. There are $n(n-1)/2$ such pairs. Therefore the total number of primitive operations referred to as the *complexity* of the algorithm and denoted by $C(n)$ is given by:

$$C(n) = [(n(n-1))/2] \times [(n-2)k_2 + k_1]$$

Expanding this expression, rearranging terms, and re-labelling constants yields:

$$C(n) = c_1 n^3 + c_2 n^2 + c_3 n + c_4$$

where the c_i, i=1,2,3,4 are constants (i.e., are not functions of n).

This simple algorithm already yields a polynomial of degree three as the expression describing its complexity. More complicated problems may yield very long and messy formulas making it difficult to compare and contrast algorithms with them. Therefore we use a simple convention to simplify the complexity expressions: we use only the term which dominates all others and drop the constants. This type of notation is called "big O" notation [PS85]. Using "big O" notation the complexity of our algorithm becomes $O(n^3)$. Note that the larger n gets the more $C(n)$ behaves like n^3. Also note that if the execution of one primitive operation takes one micro-second and n is one million then the algorithm will take at least about 10,000 years to execute. Therefore when n is of this magnitude a time complexity of $O(n^3)$ may not be feasible. Many other algorithms exhibiting smaller time complexities exist for computing the convex hull of n points [To85]. Graham's algorithm [Gr72] is one of the fastest and requires no more than $O(n \log n)$ primitive operations.

For most practical purposes O(n log n) is almost equal to O(n). Note again that if n is one million then under the same conditions as above the O(n) algorithm will execute in only one second.

2.5.4 The Inherent Complexity of Geometric Problems

Because of today's pre-occupation with faster and faster algorithms a very natural, and frequently asked, question for the 20th century computational geometers is: what is the fastest algorithm possible for solving a particular problem? For example, we may ask: what is the minimum possible number of primitive operations required to compute the convex hull of n points under a suitable model of computation? Surprisingly, this simple question was never asked by the ancient Greeks, or even Lemoine [Le02], with respect to the number of steps in their straight edge and compass constructions. This is perhaps the single and most important difference between computational geometry then and now. An answer to this question provides what is called a *lower bound on the time complexity of the problem*. Instead of the "Big O" we use the symbol Ω ("Big Omega") to denote lower bound. It should be emphasized that this is a statement not about an algorithm but about a problem. However, it is a statement about *all possible* algorithms that can ever be designed for solving the problem. The complexity of a particular algorithm for solving the problem is also refereed to as an *upper bound on the time complexity of the problem*. Finding lower bounds is one of the most theoretical activities computational geometers indulge in while having the most practical of consequences. For example, if for a given geometric problem and under a suitable model of computation an O(n log n) algorithm exists but only an $\Omega(n)$ lower bound is known then scores of researchers may spend years of precious time (possibly in vain) looking for faster algorithms. On the other hand if someone proves an $\Omega(n \log n)$ lower bound on the problem these researchers can stop their search because no faster algorithm exists (within the constant factor in front of the n log n term). They can then turn, if necessary, to attempt to reduce the constant factor term in the complexity. When the complexity function of an algorithm (upper bound) matches the complexity function of the problem (lower bound) we call such an algorithm *optimal*.

2.5.5 The Expected Complexity of Algorithms

It is also common in analyzing the complexity of an algorithm to report not only the *worst-case* complexity, i.e., the greatest number of primitive operations required over all possible configurations of the input data, but also the *expected* complexity. In the expected complexity analysis we assume the data are random and are generated from an assumed probabilistic model. Then under these assumptions we compute the expected value of the number of primitive operations required. In practice the *expected* complexity is often a more realistic description of the running time performance of the algorithm. However, an expected complexity analysis is usually much more difficult to carry out than the worst-case analysis. For a survey of expected time analysis methods in computational geometry the reader is referred to the excellent survey paper by Luc Devroye [Dev85].

2.5.6 Historical Remarks

The above discussion on complexity helps in resolving the contradictory claims concerning when and with whom computational geometry started. Although by now it is clear that computational geometry did not start with Shamos [Sh75], [Sh78] the contribution that Shamos made to the field was the emphasis on including with each algorithm a complexity analysis in terms of "Big O" notation and the introduction of lower bounds on the complexity of geometric problems. Thus, as the title of his 1975 paper makes clear, he may perhaps fairly be considered to be the father of *geometric complexity theory*.

We should remark that Shamos was not the first to use "Big O" notation to describe the complexity of a geometric algorithm. Graham's 1972 convex hull paper is a case in point [Gr72]. However Graham does not mention lower bounds. Indeed an $\Omega(n \log n)$ lower bound on the convex hull problem for n points was later established [PS85] and thus Graham's algorithm is optimal.

It is also not accurate to say that computational geometry started with the convex hull problem and that Graham had the first convex hull algorithm. Almost the same algorithm as Graham's was published (without a complexity analysis) by Bass and Schubert [BS67] a full five years earlier. For further historical remarks on the convex hull problem see [To85].

We close this section by mentioning that care has to be taken with the definition of the problem in regards to lower bounds. For example the $\Omega(n \log n)$ lower bound for the convex hull problem mentioned above does not imply the same if the input is specified as a simple polygon of n vertices. Indeed, several $O(n)$ algorithms exist for this special case of the input. The first such algorithm was discovered by McCallum and Avis [MA79].

3. The Domain of Computational Geometry

3.1 Introduction

In this section we try to give the reader a wider view of the present-day domain of computational geometry and we locate the papers in this issue within this domain. Computational geometry today is a large field and we cannot do justice to all of it. The papers in this issue touch on only a fraction of the topics of interest to computational geometry. Therefore, as we have already done in the previous section, where ever possible we mention other areas of the field and point the reader to tutorial surveys and books concerning those sub-disciplines.

3.2 Geometric Probing

Assume that you are in the business of manufacturing metal rings in the shape of a circle with a specified diameter D but that your manufacturing process is not perfect and sometimes produces rings that are not circular. However, the process is good enough that it always yields rings that are convex and nicely smooth. You would like to implement a quality control procedure to determine which rings are circular and which are not so that the latter defective rings can be sent back to the shop for re-shaping into the desired circular rings.

You remember that in your high school days when shaping an object with a lathe in the metal work shop you used a set of calipers to measure the width of the object in a given direction such as the x-direction illustrated in Fig. 3.1. Such a measuring test can be considered to be a *probe* to infer the shape of the object. Idealized in a geometrical setting we may view this probe as the result of closing two infinitely long and parallel lines from infinity (one on each side of the object) until they both touch the convex object. In this contact position the lines are referred to as *parallel lines*

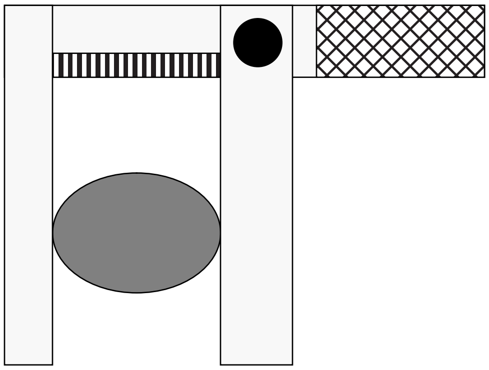

Fig. 3.1 Measuring the width of an object with the calipers.

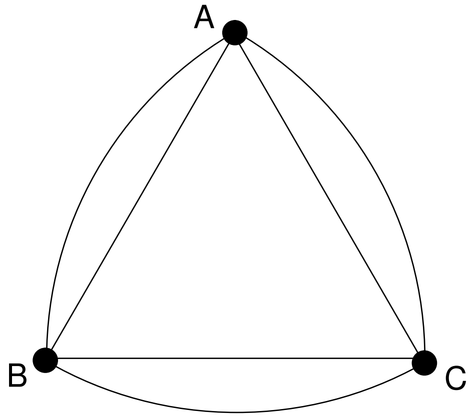

Fig. 3.2 The Reuleaux triangle: a highly non-circular shape of constant width.

of support. The result of the probe, or information obtained, is the minimum separation distance between the parallel lines of support. Let us call this type of probe a *caliper probe*. Clearly, by making one single caliper probe you cannot conclude much about the shape of the object other than that it can be placed within a parallel strip of width *D*. But guided by your strong gut feeling that if a smooth convex ring yields a separation distance of *D* in a sufficient number of directions in which the caliper probe is applied then the ring must be circular, you implement a quality control procedure in which you apply three caliper probes each 60 degrees apart from the others. If each of the three probes yields a separation distance of *D* you conclude that the ring is circular. In retrospect you may ask yourself how good such a quality control procedure is.

The above procedure is precisely the one that was used to determine if the sections of the booster rockets of the space shuttle were circular enough to safely reconstruct the rockets for their re-use in the following launch. A fascinating account of the technical, managerial and political circumstances surrounding the space shuttle *Challenger* disaster that killed seven astronauts is given by the late Nobel prize winning physicist Richard Feynman. Feynman was appointed to the investigating committee for the disaster [Fe89] and wrote candidly about it in his book *"What Do You Care What Other People Think?"*

When the solid-fuel booster rockets do their job they are jettisoned and fall into the ocean. They are picked up, taken apart into sections, and sent to Utah for reconstruction. During the trip by rail, not to mention their fall into the ocean, the rocket sections are squashed a bit and no longer have a circular cross section. Therefore each section is tested for roundness by taking three caliper probes 60 degrees apart using a rod to measure the diameter of a section. If the three diameters are in close enough agreement the sections are considered to be circular and put together again. During Feynman's visit to the Utah plant the workers there complained to him that they often had difficulty putting the sections together again and suspected that the tests were not adequate to guarantee roundness. However, their superiors allegedly ignored their complaints.

It may come as a surprise to those ignorant in geometry that even a convex shape that has a thousand, a million, or even an infinite number of diameters measured in an infinite number of different directions all yielding the same value *D* may still be highly non-circular. The Reuleaux triangle illustrated in Fig. 3.2 is one such example. In Fig. 3.2 the straight line figure ABC forms the standard equilateral triangle. Let the length of each side (straight edge) of the triangle be D. To construct a Reuleaux triangle we substitute the straight sides of the triangle with circular arcs of radius D. For example, the straight side AB is substituted by the arc AB of a circle of radius D centered at C. A similar construction is done for the sides BC and CA with the circle centered at A and B, respectively. It is obvious that for any direction the parallel lines of support have a separation D. In other words all caliper probes have the same diameter. Such a shape is known as a constant diameter shape [Kl71]. If the Reuleaux triangle is placed vertically on the ground like a bicycle wheel and a flat platform is moved parallel to the ground while keeping contact with the top of the triangle it will roll as smoothly as a *circular* wheel without the platform bobbing up and down as might be expected. Nevertheless the shape is highly non-circular. The weakness of this test in measuring the roundness of the booster rocket sections may have contributed significantly to the tragic *Challenger* disaster.

The above example illustrates the application of one type of probe (the caliper probe) to determining the shape of an object. It also demonstrates that any number of such probes is insufficient to determine if a shape is circular. However, there are many other shapes of interest besides circles. For example, we may imagine that a robot with touch sensors such as caliper probes must determine the exact description of an unknown convex polygon of n vertices. We may also assume a more powerful probe that not only reports the separation between the parallel lines of support but also the coordinates of the contact points. Then an unknown convex polygon may be completely determined using a *finite* number of such probes.

The theory of geometric probing is concerned with these types of problems for a wide variety of objects in two and higher dimensions using caliper probes as well as a variety of other types of probes. Two of the most important issues here concern, on the one hand, the determination of the number of probes that are necessary and sufficient to determine an object completely, and secondly, the design of efficient algorithms for actually carrying out the probing strategies. In the first paper in this issue Steve Skiena provides a tutorial survey of this area which should be of interest to researchers in computer vision, pattern recognition, verification and testing, quality control and robotics.

This area of computational geometry also illustrates another difference between the field as practiced now compared to that at the time of the Greeks. The Greeks were concerned with computing properties of objects that were specified completely in advance and did not explore the "learning" aspect exhibited by probing strategies applied to inputs that are only partially specified in advance. This part of computational geometry is clearly inspired by the modern ideas springing up in the field of artificial intelligence.

3.3 Art Gallery Theorems and Algorithms

Let us say you own an art gallery or museum with expensive paintings hanging on the walls and priceless sculptures standing on the floor. You are interested in installing a security system composed of surveillance cameras that are mounted at fixed locations but that can rotate the full 360 degrees so that they can "see" in all directions. Assume for simplicity that you live in "flatland", i.e., the world is the two dimensional plane. In other words we will place the cameras on the floor-plan of the art gallery. The art gallery can then be modelled by a simple polygon such as the

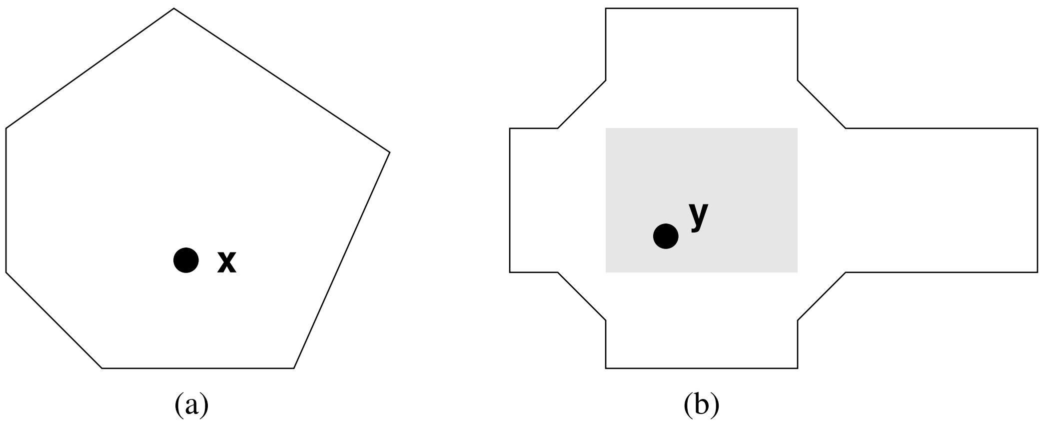

Fig. 3.3 Convex (a) and star-shaped (b) art galleries require no more than one surveillance camera.

two illustrated in Fig. 3.3. If your gallery has a convex shape (Fig. 3.3 (a)) you clearly need only one camera located at the point marked x. Furthermore x can be chosen anywhere in the gallery. A polygon that requires no more than one camera is known as a *star-shaped* polygon. All convex polygons are obviously star-shaped but not all star-shaped polygons are convex. The set of locations in a star-shaped gallery where a single camera may be mounted so as to see everything is called the *kernel* of the gallery. In Fig. 3.3 (b) we have an example of a non-convex gallery that is star-shaped. The camera (point y) may be mounted anywhere in the kernel (shaded region) of the gallery.

Now assume that you have a very large and intricate (non-star-shaped) gallery consisting of n walls and that the number n is quite large. Perhaps your gallery looks something like the one in Fig. 3.4. In 1973 Victor Klee asked the question: given an arbitrarily shaped gallery what is the minimum number of cameras required to guard the interior of an n-wall gallery [Ho76]. Vasek Chvatal soon established what has become known as “Chvatal’s Art Gallery Theorem” (or sometimes “watchman theorem”): $n/3$ cameras are always sufficient and sometimes necessary [Ch75]. In Fig. 3.4 n=36 and $n/3$=12 and therefore 12 cameras are sufficient according to the theorem. However, for this particular art gallery only four are necessary and one possible position for these four is illustrated in the figure. In 1981 Avis and Toussaint exhibited an efficient algorithm for actually finding the locations where the cameras should be mounted [AT81]. We should add that it is tempting to place a camera on every third vertex of the polygon. The reader can easily design a gallery where this placement strategy will fail no matter on which vertex one starts placing the cameras.

This type of problem for different types of cameras (or guards) and different types of environments (galleries) falls into an area of research which has come to be known as *art gallery theorems and algorithms*. Joseph O’Rourke has written a jewel of a monograph devoted entirely to this subject [O’R87].

In the second paper in this issue Thomas Shermer provides a tutorial survey of this area

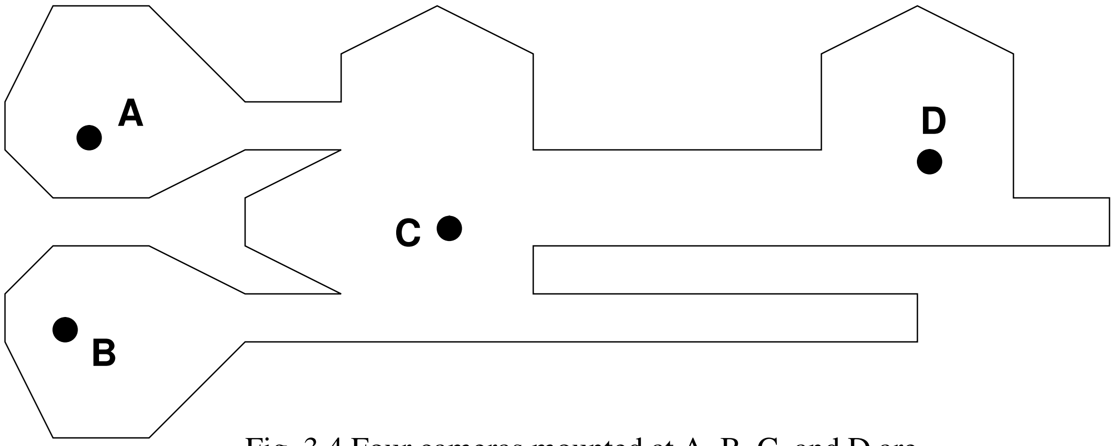

Fig. 3.4 Four cameras mounted at A, B, C, and D are sufficient to guard this gallery.

paying special attention to results obtained since the publication of O'Rourke's book.

3.4 Computer Graphics

3.4.1 The Hidden Line Problem

Consider the art gallery of Fig. 3.4 again where four cameras (A, B, C and D) are installed to "see" the entire gallery. You may be interested in determining exactly which sections of the gallery are visible with a single camera, say C. Such a region, referred to as the *visibility region* from point C is illustrated in Fig. 3.5 as the shaded region. Determining this visibility region is equivalent to removing the un-shaded portions of the polygon hidden from camera C and is known in the computer graphics literature as the *hidden line problem*. Of course in computer graphics we are interested even more in the three dimensional version of this problem known as the *hidden surface removal* problem. In a typical setting we are given a geometrical description of a collection of objects in space (stored in the computer) and if a point V in space is specified as the location of a viewer, it is required to display the objects as they would be seen from point V with the hidden portions of the objects removed. This is one of the many crucial problems that must be solved as part of the obstacles on the road towards the major goal of computer graphics: that of creating visual images. A variety of other problems occur in computer graphics and the reader is referred to standard text books for details [FVD82], [NS79]. However, many of these problems such as the hidden line and hidden surface problems have benefitted considerably from the field of computational geometry. Consider for example the two-dimensional hidden line problem illustrated in Fig. 3.5. One standard algorithm used for solving this problem is the one due to Freeman and Loutrel [FL67] developed more than ten years before Shamos' thesis marking the alleged birth of computational geometry. Freeman and Loutrel did not provide a complexity analysis of their algorithm in terms of "Big O" notation but their algorithm runs in time $O(n^2)$, where n is the number of sides of the polygon, under the usual assumptions made in computational geometry. Using the tools developed for the design of algorithms in computational geometry, ElGindy and Avis [EA81] were able to design an algorithm for this problem that runs in *optimal* time, i.e., $O(n)$. Since their seminal discovery others have also discovered algorithms with this time complexity. When n is large this represents

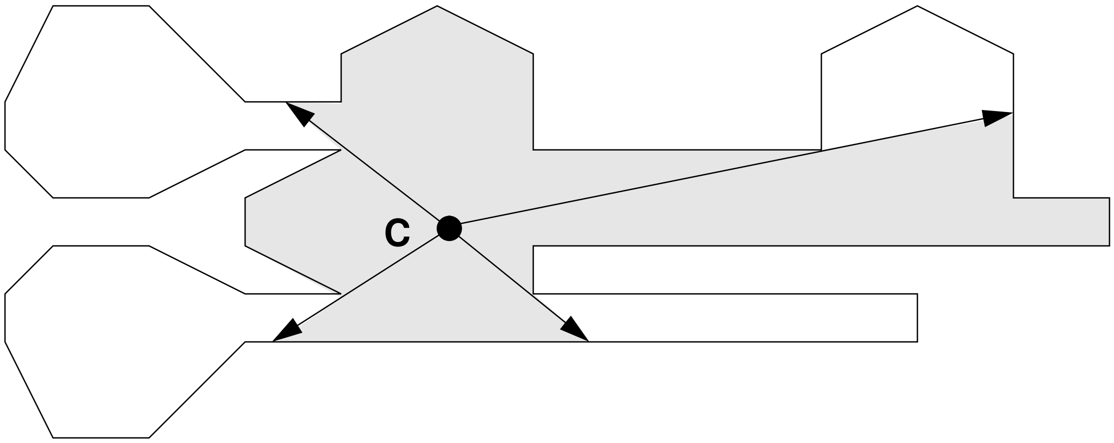

Fig. 3.5 The portion of the gallery visible (shaded portion) with camera C.

a considerable improvement in the speed of the algorithm.

3.4.2 Ray Tracing, Polygonal Approximation and Polygon Triangulation

Three other fundamental problems in computer graphics are: (1) *ray-tracing*, (2) *polygonal approximations of a curve* and (3) *triangulating polygons*. Here also computational geometry has recently made significant contributions [Be92], [II88], [To91].

3.4.3 Computational Geometry and Computer Graphics

That computational geometry and computer graphics have influenced each other has been obvious particularly with respect to the hidden line and surface removal problems. Computer graphics makes practical problems known to the computational geometry community and computational geometry often provides faster algorithms. It is expected that computational geometry will have many other dramatic improvements to the standard algorithms used in computer graphics. Computational geometry also provides new ways for graphics programmers to think about their problems and to do their job [Fi89], [St91]. In the third paper in this issue David Dobkin explores life at the interface of these two fields.

3.5 Dynamic Computational Geometry

Consider again the problem of computing the convex hull of a set of points in the plane. Let us say you were given an input data set of n points and you computed the convex hull with $O(n^3)$ primitive operations using the algorithm described in section 2.5. Assume now that you are given an additional point p that was accidentally left out of the original data and told that you need the convex hull of the entire collection of $n+1$ points. In other words you are asked to *update* the convex hull of the n points by *inserting a* new point. One obvious approach is to disregard all the work you have done on the original n points and to apply the algorithm once again from scratch to the $n+1$ points. This in effect means that you are inserting one new point using $O(n^3)$ primitive oper-

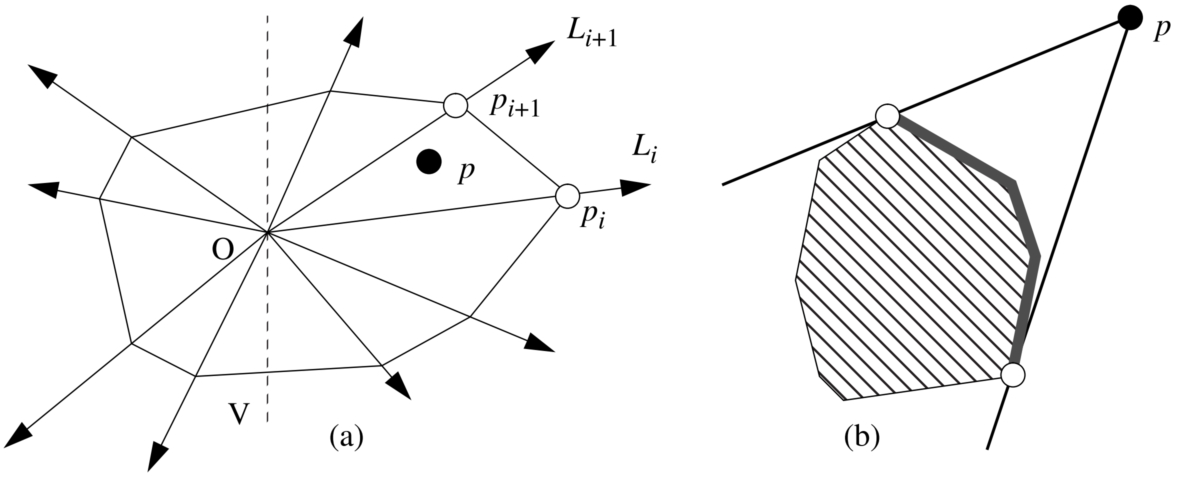

Fig. 3.6 Updating the convex hull by inserting a new point p (a) inside the old hull and (b) outside the old hull.

ations. Note that using the “Big O” notation convention we have that $O((n+1)^3) = O(n^3)$. Not surprisingly this is a rather wasteful approach. A much more efficient method would modify the existing convex hull to reflect the introduction of the new point p. For example, we could first test whether the new point lies in the convex hull of the original data set. If so the new convex hull is the same as the old one and nothing more need be done. The obvious next question is: how fast can we tell whether a point p is inside a convex polygon of n sides? First note that the edges of the convex hull come in two varieties: the *upper* edges and the *lower* edges. Recall that we assumed for simplicity of discussion that no two points have the same x coordinate and hence no edge is vertical. An edge is an upper edge if the interior of the convex hull lies below the line through the edge. An edge is a lower edge if the interior of the convex hull lies above the line through the edge. Clearly, a point p is in the convex hull if, and only if, it lies below all the lines through the upper edges and above all the lines through the lower edges. Therefore we can determine if a point p is inside a convex polygon of n sides by using the algorithm described in the introduction to test the relative position of a point with respect to a line. This approach yields an $O(n)$ algorithm for testing point inclusion in a convex polygon. However, we can do much better than that if we do some *pre-processing* of the convex polygon and store it in an appropriate *data structure*.

First pick an arbitrary point O in the interior of the polygon. This can be done by taking the arithmetic mean of any three vertices of the polygon. Then construct half-lines starting at O and passing through every vertex of the polygon as illustrated in Fig. 3.6 (a). Consider the vertical line V through O. This line divides the half-lines into two groups: those to the right of V and those to the left of V. Assume p lies to the right of V and consider the half-lines to the right of V. Because the polygon is convex all these half-lines occur in *sorted* order by slope as we traverse the ordered vertices of the polygon. Therefore by storing these half-lines in a *sorted array* we can apply *binary search* to determine in only $O(\log n)$ operations the line immediately below our point p. Similarly, we can apply *binary search* to determine in $O(\log n)$ operations the line immediately above our point p. Once we know between which pair of adjacent half-lines our point lies a simple test will tell us if it is inside the polygon. Assume we have determined that p lies between L_i and L_{i+1} as in Fig. 3.6 (a). Then if p_i, p_{i+1} is an upper edge we test if p lies below the line through p_i, p_{i+1}. If it does p is in the convex polygon, otherwise it is not. If p_i, p_{i+1} is a lower edge we test if p lies above the line through p_i, p_{i+1}.

A similar approach is used on the left half-lines if p lies to the left of V. The special cases when p lies above (or below) all the half-lines are taken care of in a similar manner. Therefore with this approach we can determine if a point lies inside a convex polygon of n vertices using only O(log n) primitive operations. If the point lies outside of the convex hull then of course the *updating* procedure is not yet finished as we must modify the convex hull by disconnecting part of the old boundary and adding two new edges to the new one as illustrated in Fig. 3.6 (b). Using analogous techniques to those described above this can also be done in O(log n) operations [PS85].

Let us now change the scenario so that instead of having only one new point to insert, new data points are continually arriving indefinitely. In order to continue to benefit from our pre-processed data structure so that we may continue to insert new points in O(log n) operations we must also update the data structure. Such data structures are referred to as *dynamic* and this sub-discipline of computational geometry is referred to as *dynamic computational geometry*. Furthermore, in fully dynamic situations we want not only to support insertions of new data points as discussed above but also *deletions* of old data points. Some thought on this problem will quickly reveal that deletion is a much more difficult problem.

In the fourth paper of this issue Yi-Jen Chiang and Roberto Tamassia provide a tutorial survey of this area which has important practical applications in circuit layout, computer graphics and computer-aided design. As pointed out in the short description above this area depends heavily on the design of clever data structures. For the latest results on data structures for computational geometry the reader is referred to the excellent book (actually a Ph.D. thesis) by Marc van Kreveld [Kr92].

3.6 Parallel Computational Geometry

3.6.1 Networks of Sequential Computers

In the previous discussion on models of computation it was tacitly assumed that the models were *sequential*, i.e., that all the primitive operations required to obtain a solution to a problem were executed one after the other. It was also pointed out that for some problems, if the input is very large, the time taken for the execution of the entire algorithm may be prohibitive. One is naturally led to the question: if instead of using one machine (processor, computer) on which primitive operations are executed one after the other, we used a collection of k computers that all work on the problem at the same time, how much faster can we solve the problem. Such computers are called *parallel* computers. Because each computer is only solving part of the problem and the computers must communicate with each other in order not to duplicate work and “stitch” the partial solutions obtained by each computer into a complete solution, the algorithms used by parallel computers are different from those used by sequential computers. Such algorithms are called *parallel algorithms*. Remember for example the convex hull algorithm described above that required O(n^3) primitive operations to find the convex hull of n points. On a sequential computer such an algorithm requires O(n^3) units of time. However, if we use O(n^3) computers (simple processors) as our massive parallel computer then the convex hull of n points can be computed in *one* unit of time

[Ak82]. Of course it may be totally infeasible to build a computer with $O(n^3)$ processors communicating with each other. Parallel computational geometry deals with the trade-off problems between the number of processors used in a parallel machine and the time taken by the parallel machine to find the solution to a geometric problem.

In the fifth paper of this issue Mikhail Atallah provides a survey of techniques used for solving geometric problems on parallel machines. These techniques rely heavily on the methodology of the design and analysis of parallel algorithms in general. For an excellent introduction into this topic the reader is referred to the excellent book by Selim Akl [Ak89].

3.6.2 Neural-Network Computational Geometry

We should mention that the classic work on neural networks by Minsky and Papert [MP69] falls into the domain of parallel computational geometry as well. The accent here however is on the ability of such parallel computers (neural networks) to "learn" to recognize certain geometric properties of the input data. In this sense neural network research and computational geometric probing have much in common.

3.6.3 Optical Computational Geometry

We close this section by mentioning that another parallel approach to computational geometry that is radically different from those approaches discussed so far, uses *optical computers*. The complexity analysis of optical algorithms involves the appropriate definition of optical primitives. For details on this new and potentially revolutionary development in computational geometry the reader is referred to [KS92].

3.7 Isothetic Computational Geometry

Isothetic computational geometry (also *rectilinear* computational geometry) deals with input data such as line segments and polygons in which all the edges are either vertical or horizontal. This restriction of the input often greatly simplifies the algorithms for solving geometric problems. Furthermore, in some applications areas such as image processing and VLSI design the data considered are predominantly isothetic polygons. For a survey of isothetic computational geometry the reader is referred to [Wo85].

Perhaps the most important application of isothetic computational geometry is to VLSI and within this field one of the most fundamental problems is the so called *Contour-of-a-Union-of-Rectangles* problem. In this problem we are given a set of isothetic rectangles in the plane and it is required to compute the boundary of the union of the rectangles (see Fig. 3.7). In the sixth paper of this issue Diane Souvaine and Iliana Bjorling-Sachs provide a comparative study of three well known algorithms for a generalized version of this problem paying special attention to practical considerations.

3.8 Numerical Computational Geometry

Consider the following input to a convex hull algorithm. Generate one thousand points on the boundary of a given circle. Then compute the convex hull of these points. Clearly the convex

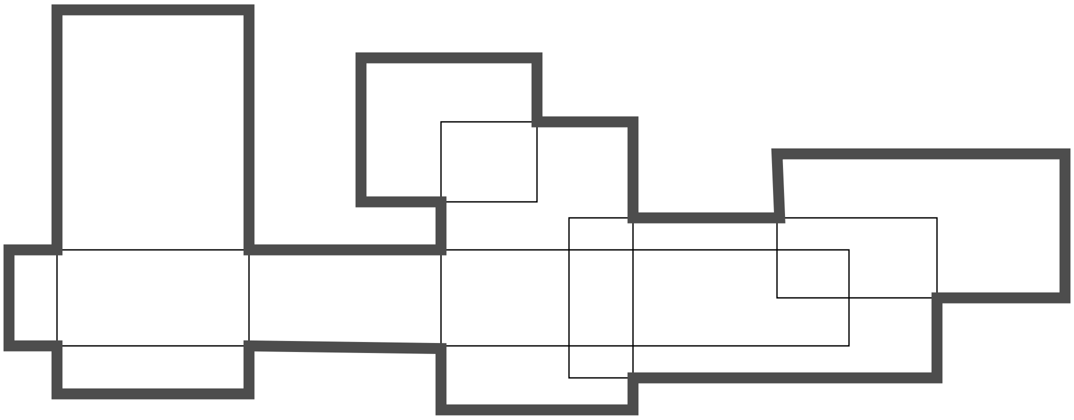

Fig. 3.7 A set of isothetic rectangles and
the contour (bold lines) of their union.

hull should be a convex polygon of one thousand vertices. A typical implementation of a typical convex hull algorithm on the other hand will yield a convex polygon with perhaps only 950 vertices! This illustrates the disconcerting problem of numerical robustness of geometric algorithms, a primary concern to the programmers and users of geometric programs.

Numerical computational geometry deals with the design of numerically robust algorithms for solving geometric problems [DS88], [Mi88]. Several different approaches to this problem are emerging. Edelsbrunner and Mucke [EM88] describe a technique which they call *simulation of simplicity* that simplifies correctness proofs of geometric algorithms by suggesting a uniform framework with which to deal with degeneracies. For alternate approaches see also [GY86], [HHK88] and [OTU87]. Of particular relevance to computer graphics is the work of Karasick [Ka88] and Sugihara [Su87] who consider numerical computational geometry in the context of *solid modelling*.

In the seventh paper in this issue Kokichi Sugihara and Masao Iri present a numerically stable algorithm for constructing Voronoi diagrams, no doubt the single most important geometric structure in computational geometry that has a wide range of applicability across many disciplines. For an in-depth treatment of Voronoi diagrams and its many generalizations the reader is referred to the book by Rolf Klein [Kl89].

3.9 Geometric Modeling

Geometric modeling refers to the process of generating geometric models of real objects or dynamic processes that can be stored in the computer with the goal of either design (CAD), manufacturing (CAM) or process simulation. Not surprizingly there are a variety of sub-problems in geometric modeling where computational geometry plays an important role. For a comprehensive treatment of this field the reader is referred to the book by Michael Mortenson [Mo85]. One of the most important problems in geometric modeling is the automatic generation of a mesh inside a polygon. This also forms a fundamental and indispensable tool for solving systems of partial differential equations with the finite-element method.

In the eighth paper in this issue, Vijay Srinivasan, Lee Nackman, Jun-Mu Tang and Siavash Meshkat show how some tools developed in the field of computational geometry can be used to design new automatic mesh generation techniques that have some advantages over traditional methods.

3.10 Computer Vision

3.10.1 Introduction

Computer vision has flourished now for some forty years as a sub-discipline of artificial intelligence and hundreds of books are readily available on the subject and will not be mentioned here. Relevant to this special issue are the first two books that are the fruit of the marriage between computer vision and computational geometry and these are the monographs by Ahuja & Schacter [AS83] and Sugihara [Su86].

It is useful to decompose the computer vision problem into a series of sub-problems that are usually tackled sequentially and separately in some order such as that illustrated in Fig. 3.8. The purpose of a computer vision program is to analyze a scene in the real world with the aid of an input device which is usually some form of transducer such as a digital camera and to arrive at a description of the scene which is useful for the accomplishment of some task. For example, the scene may consist of an envelope in the post office, the description may consist of a series of numbers supposedly accurately identifying the zip code on the envelope, and the task may be the sorting of the envelopes by geographical region for subsequent distribution. Typically the camera yields a two-dimensional array of numbers each representing the quantized amount of light or brightness of the real world scene at a particular location in the field of view. The first computational stage in the process consists of segmenting the image into meaningful objects. The next stage usually involves processing the objects to enhance certain of their features and to remove noise of one form or another. The third stage consists of feature extraction or measuring the "shape" of the objects. The final stage is concerned with classifying the object into one or more categories on which some subsequent task depends. Computational geometry can contribute significantly to all these sub-problems. Robert Melter, Azriel Rosenfeld and Prabir Bhattacharya have recently edited an excellent collection of papers devoted precisely to the application of computational geometry to different aspects of all these sub-problems [MRB91].

3.10.2 Proximity Graphs and The Shape of a Set of Points

In some contexts such as the analysis of pictures of bubble-chamber events in particle physics the input patterns are not well described by polygons because the pattern may consist essentially of a set of disconnected dots. Such "objects" are called *dot patterns* and are well modeled as sets of points. Thus one of the central problems in shape analysis is extracting or describing the "shape" of a set of points. Let $S=\{x_1, x_2,..., x_n\}$ denote a finite set of points in the plane. A *proximity graph* on a set of points is a graph obtained by connecting two points in the set by an edge if the two points are close, in some sense, to each other. The minimal spanning tree (MST) and the relative neigh-

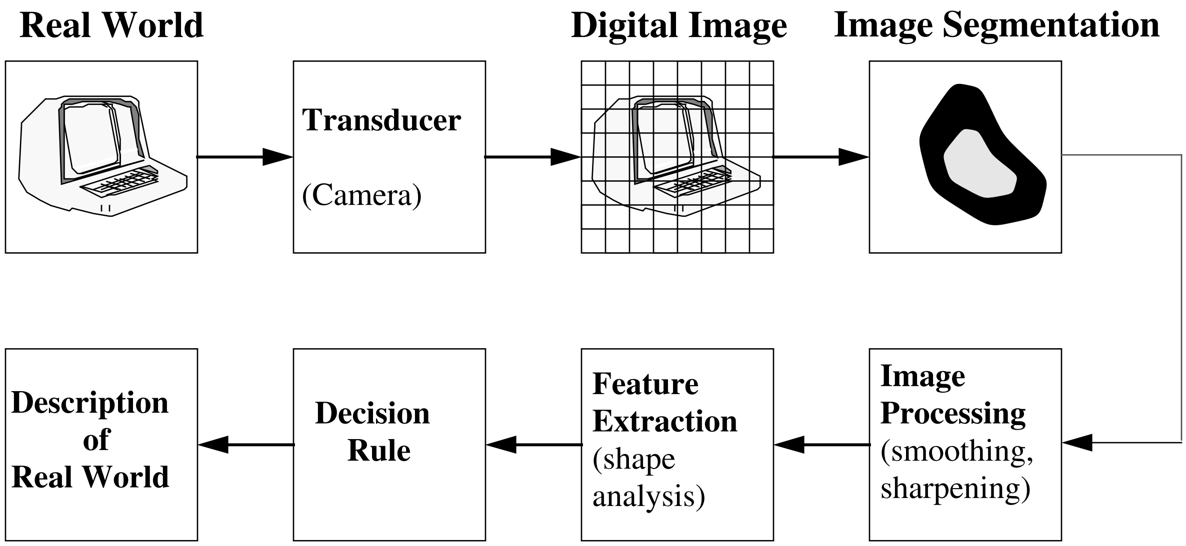

Fig. 3.8 Decomposing the *computer vision* problem into sub-problems.

borhood graph (RNG) are two proximity graphs that have been well investigated in this context.

A spanning tree of a set of points is a tree connecting all the points constructed by adding edges between pairs of points. The length of a tree is the sum of the lengths of all the edges in the tree. The *minimal spanning tree* is the tree that has the minimal length over all spanning trees. It has attractive properties for computer vision and for this reason has been widely employed. Consider the dot pattern in Fig. 3.9 (a). How would you connect these points so that the resulting figure describes the perceptual structure so evident to humans in the dot pattern? Well the minimal spanning tree of the dot pattern is shown in Fig. 3.9 (b) and it clearly does an admirable job on this pattern. However the minimal spanning tree imposes tree structure on every dot pattern it “sees.” On the cyclic dot pattern of Fig. 3.9 (c) it fails because it leaves a gap somewhere. The relative neighborhood graph defined below is much more powerful than the minimal spanning tree in these kinds of problems.

The lune of x_i and x_j, denoted by Lune(x_i, x_j), is defined as the intersection of the two discs centered at x_i and x_j each with radius equal to the distance between x_i and x_j. The relative neighborhood graph is obtained by joining two points x_i and x_j of S with an edge if Lune(x_i, x_j) does not contain any other points of S in its interior. The relative neighborhood graph of a dot pattern that is not a tree is shown in Fig. 3.8 (c). Note that the structure shown in Fig. 3.9 (b) is also the relative neighborhood graph. By generalizing the shape of Lune(x_i, x_j) one obtains generalizations of the relative neighborhood graph. These graphs have not only found applications in computer vision but in many other areas as well. In the last paper of this issue Jerzy Jaromczyk and Godfried Toussaint provide a survey of results on relative neighborhood graphs and their generalizations and point out

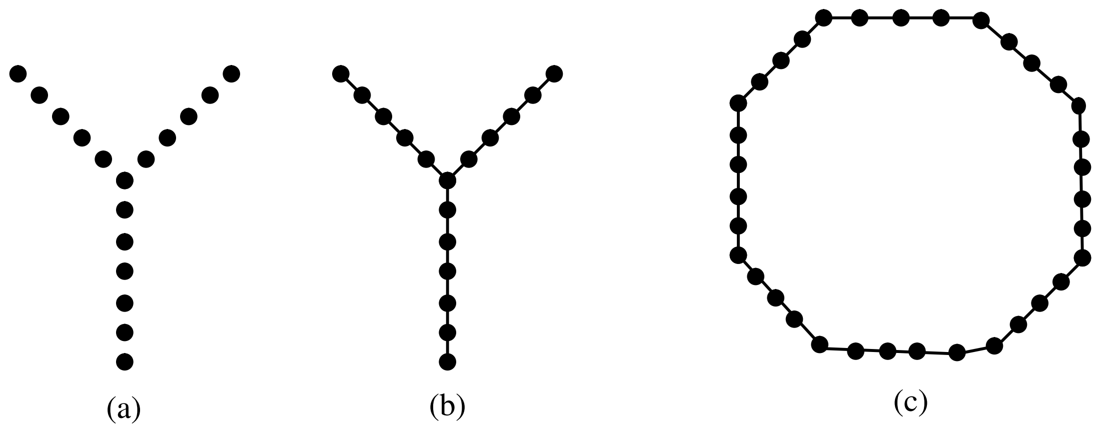

Fig. 3.9 (a) A *dot pattern.* (b) The *minimal-spanning-tree* of the points in (a). (c) The *relative-neighbourhood-graph* of another dot pattern.

the areas in which they have been successfully applied.

3.11 Robotics

No discussion of computational geometry can go very far without mentioning robotics as one of the main applications even though no contribution in this area appears in this special issue. There are many sub-problems of robotics which could be mentioned. However we limit ourselves to three: (1) *motion planning*, (2) *linkages* and (3) *automated assembly*.

In motion planning the typical problem involves a robot modeled as a polygon in two dimensions or a polyhedron in three dimensions which must maneuver in space amongst a collection of obstacles. Here we are usually asked questions such as: can the robot move from point A to point B without colliding with the objects and if the answer is in the affirmative find the shortest such path. For the latest results applying computational geometry to such problems the reader is referred to the books by Jacob Schwartz, Micha Sharir and Chee Yap [SY87], [SSH87] and the tutorial surveys by Helmut Alt, Chee Yap and Sue Whitesides [AY90a], [AY90b], [Wh85].

A linkage is a collection of rigid rods that are fastened together at their endpoints, about which they may rotate freely. The rods may be connected in a variety of ways. For example, the rods may form a chain (*chain linkage*) or a closed polygon (a *poligonal linkage*). Many fascinating questions arise with linkages. For example, we may ask: can a poligonal linkage be turned inside out? For the latest results in this area the reader is referred to the tutorial survey of Sue Whitesides [Wh92].

Automated assembly is a type of motion planning problem where we consider a collection of objects and must answer questions about whether the collection can be separated (taken apart) or brought together into a specified configuration and if so what kinds of motion are guaranteed to yield an answer. A typical question is whether a collection of objects can be disassembled by moving only one object at a time. This is always true for convex objects in the plane but shockingly untrue for convex objects in three dimensions. In Fig. 3.10 is illustrated a configuration of three star-shaped objects such that no single object can be moved without disturbing the others but they can be separated if two of them are moved simultaneously each with its own separate direction and velocity of translation. For a tutorial survey of this field the reader is referred to [To85b].

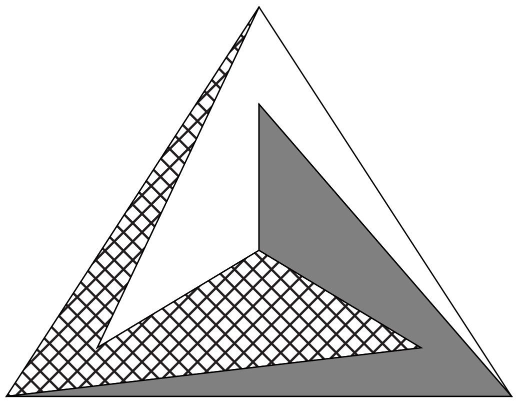

Fig. 3.10 Three star-shaped objects such that no single object can be moved without disturbing the others but they can be separated if two of them are moved simultaneously.

3.12 Geodesic Computational Geometry

Consider again the minimax facility location problem discussed in section 2 but now assume that transportation is to be carried out by ship and the customers are coastal cities in a country such as the Philippines made up of a large number os islands. In this situation the Euclidean distance between two points is no longer a useful measure. What we need is the *geodesic* distance, i.e., the length of the shortest path (*geodesic path*) between the two points that avoids obstacles. In this example the shortest path is restricted to the ocean and the islands are the obstacles which must be circumvented.

A fundamental structure that has proved very useful in scenarios such as these relevant to both facility location and path planning in robotics, is a generalization of the convex hull known as the *relative convex hull* (also *geodesic convex hull*). Given a set S of n points inside a simple polygon P of n vertices, the relative convex hull of S (i.e., relative to P) is the minimum perimeter circuit that lies in P and encloses S (see Fig. 3.11). The relative convex hull of a set of points in a polygon was first investigated in [To86] where an optimal $O(n \log n)$ time algorithm for its computation was given. Since then other similar algorithms with the same time complexity have appeared [GH89], [He92]. For applications of the relative convex hull the reader is referred to [To89].

3.13 Computational Topology

Many fundamental problems in computational geometry have a topological flavor. For example, given a closed polygonal chain in three dimensional space it is natural to ask whether the chain is knotted in some way or whether it is “simple.” Inspired by such problems, a new branch of computational geometry, *computational topology* has developed in the past few years. For details and pointers to the small but significant literature on this subject the reader is referred to [Sc92].

4. Conclusion

When the electronic digital computer was first introduced in the early 1940’s it was used as a “number cruncher” imitating the purpose of its mechanical calculating predecessors built by mathematicians such as Blaise Pascal and Gottfried Liebnitz in the 17th century and Charles Babbage in the 19th century. Computing was numerical then and *numerical analysis* was the primary concern of the design and analysis of numerical algorithms for solving equations. However, it is relatively difficult for human beings to communicate with machines through numbers. A word is

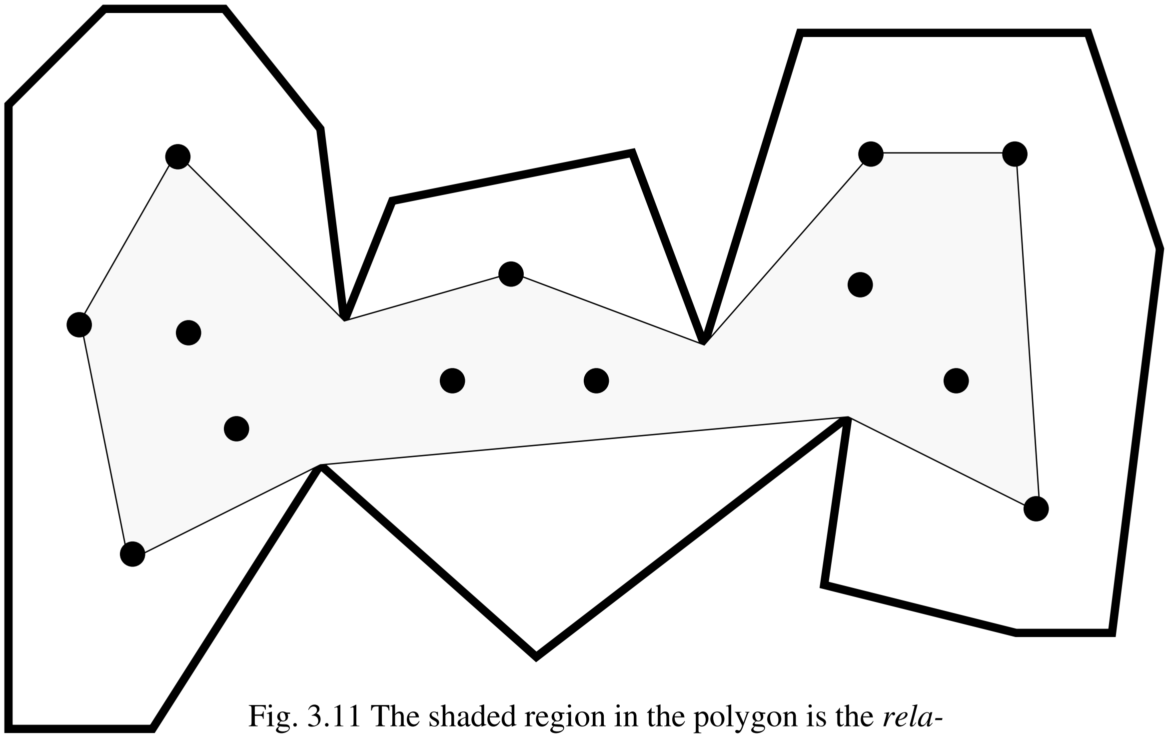

Fig. 3.11 The shaded region in the polygon is the *relative convex hull* of the set of points in the polygon.

worth a thousand digits and the development of programming languages helped considerably to smooth the human-machine interface. But of course a picture is worth a thousand words and human-machine communication today is done almost exclusively through graphic displays or pictures. Computing has become *visual* and the foundation of visual computing is *computational geometry.* I hope this issue gives the reader a taste of the ubiquity, practical relevance and beauty that characterize computational geometry.

5. Acknowledgments

First I would like to thank Theo Pavlidis for conceiving the idea of this special issue. I thank all the invited authors for undertaking the task of writing the papers in spite of their busy schedules and all the referees for taking time off their writing in order to read. I thank Reed Crone, the Executive Editor of *The Proceedings of the IEEE*, for his hospitality and help with the managerial aspects of guest editing. I thank Jit Bose for reading a preliminary version of this manuscript and for making many useful suggestions. Maria Pineda made it all worth while.

6. References

[AHU74] Aho, A. V., Hopcroft, J. E. and Ullman, J. D., *The Design and Analysis of Computer*

Algorithms, Addison-Wesley, Reading, Mass., 1974.

[Ak82] Akl, S., “A constant-time parallel algorithm for computing convex hulls,” *BIT*, vol. 22, 1982, pp. 130-134.

[Ak89] Akl, S., *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[AS83] Ahuja, N., and Schacter, B. J., *Pattern Models,* John Wiley, 1983.

[AT81] Avis, D. and Toussaint, G. T., “An efficient algorithm for decomposing a polygon into star-shaped pieces,” *Pattern Recognition,* vol. 13, 1981, pp. 295-298.

[Av87] Avron, A., “Theorems on strong constructibility with a compass alone,” *Journal of Geometry*, vol. 30, 1987, pp. 28-35.

[Av90] Avron, A., “On strict strong constructibility with a compass alone,” *Journal of Geometry,* vol. 38, 1990, pp. 12-15.

[AY90a] Alt, H. and Yap, C. K., “Algorithmic aspects of motion planning: a tutorial, part 1,” *Algorithms Review*, vol. 1, No. 1, pp. 43-60.

[AY90b] Alt, H. and Yap, C. K., “Algorithmic aspects of motion planning: a tutorial, part 2,” *Algorithms Review*, vol. 1, No. 2, pp. 61-78.

[Be92] de Berg, Mark, *Efficient Algorithms for Ray Shooting and Hidden Surface Removal,* University of Utrecht, 1992.

[BS67] Bass, L. J. and Schubert, S. R., “On finding the disc of minimum radius containing a given set of points,” *Mathematics of Computation*, vol. 21, 1967, pp. 712-714.

[BET91] Bhattacharya, B., Egyed, P. and Toussaint, G. T., “Computing the wingspan of a butterfly,” *Proc. Third Canadian Conference on Computational Geometry*, Vancouver, August 6-10, 1991, pp. 88-91.

[BT91] Bhattacharya, B. and Toussaint, G. T., “Computing shortest transversals,” *Computing*, vol. 46, 1991, pp. 93-119.

[Ch75] Chvatal, V., “A combinatorial theorem in plane geometry,” *Journal of Combinatorial Theory, Series B,* vol. 18, 1975, pp. 39-41.

[CR81] Courant, R. and Robbins, H., *What is Mathematics?* Oxford University Press, 1981.

[Cz91] Czyzowics, J., Egyed, P., Everett, H., Rappaport, D., Shermer, T., Souvaine, D., Toussaint, G. T., and Urrutia, J., “The aquarium keeper’s problem,” *Proc. ACM/SIAM Symposium on Discrete Algorithms*, January 28-30, 1991, pp. 459-464.

[De85] Devroye, L., “Expected time analysis of algorithms in computational geometry,” in *Computational Geometry*, G. T. Toussaint, ed., North-Holland, 1985, pp. 135-151.

[DS88] Dobkin, D. and Silver, D., “Recipes for geometry & numerical analysis - Part I: An empirical study,” *Proc. 4th Annual Symposium on Computational Geometry*, Urbana, June

1988, pp. 93-105.

[Du90] Dunham, W., *Journey Through Genius: The Great Theorems of Mathematics*, John Wiley and Sons, Inc., 1990.

[EA81] ElGindy, H. and Avis, D., "A linear algorithm for computing the visibility polygon from a point," *Journal of Algorithms*, vol. 2, 1981, pp. 186-197.

[EM88] Edelsbrunner, H. and Mucke, E., "Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms," *Proc. 4th Annual Symposium on Computational Geometry*, Urbana, Illinois, June 1988, pp. 118-133.

[Ev59] Eves, H., "Philo's line," *Scripta Mathematica*, vol. 26, 1959, pp. 141-148.

[Fe89] Feynman, R. P., "*What Do You Care What Other People Think?*" Bantam, 1989.

[Fi89] Fiume, E. L., *The Mathematical Structure of Raster Graphics*, Academic Press, Inc., San Diego, 1989.

[FL67] Freeman, H. and Loutrel, P. P., "An algorithm for the two dimensional hidden line problem," *IEEE Transactions on Electronic Computers*, vol. EC-16, 1967, pp. 784-790.

[Fo68] Forrest, A. R., "Curves and surfaces for computer aided design," Ph.D. thesis, University of Cambridge, 1968.

[Fo71] Forrest, A. R., "Computational geometry," *Proceedings of the Royal Society,* London, Series A, vol. 321, 1971, pp. 187-195.

[Fo74] Forrest, A. R., "Computational geometry - Achievements and problems," in *Computer Aided Geometric Design*, Academic Press, New York, 1974, pp. 17-44.

[FVD82] Foley, J. D. and Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.

[FW74] Francis, R. L. and White, J. A., *Facility Layout and Location*: *An Analytical Approach*, Prentice-Hall, Inc., 1974.

[GH89] Guibas, L. and Hershberger, J., "Optimal shortest path queries in a simple polygon," *Journal of Computer and System Sciences*, vol. 39, No. 2, 1989, pp. 126-152.

[GJ78] Garey, M. and Johnson, D., *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman and Co., San Francisco, 1978.

[Gr72] Graham, R. L., "An efficient algorithm for determining the convex hull of a planar set," *Information Processing Letters,* vol. 1, 1972, pp. 132-133.

[GY86] Greene, D. H. and Yao, F. F., "Finite-resolution computational geometry," *Proc. 27th IEEE Symposium on Foundations of Computer Science*, Toronto, October 1986, pp. 143-152.

[He81] Heath, T., *A History of Greek Mathematics*, Dover, New York, 1981.

[He92] Hershberger, J., "Optimal parallel algorithms for triangulated simple polygons," *Proc.*

8th ACM Symposium on Computational Geometry, Berlin, June 10-12, 1992, pp. 33-42.

[HHK88] Hoffmann, C. M., Hopcroft, J. E., and Karasick, M. S., "Towards implementing robust geometric computations," *Proc. 4th Annual Symposium on Computational Geometry*, Urbana, Illinois, June 1988, pp. 106-117.

[Ho70] Honsberger, R., *Ingenuity in Mathematics,* Random House, Inc., 1970.

[Ho76] Honsberger, R., *Mathematical Gems II*, Mathematical Association of America, 1976, pp. 104-110.

[HT85] Hildebrandt, S. and Tromba, A., *Mathematics and Optimal Form*, Scientific American Books, Inc., 1985.

[II88] Imai, H. and Iri, M., "Polygonal approximations of a curve - formulations and algorithms," in *Computational Morphology,* G. T. Toussaint, ed., North-Holland, 1988, pp. 71-86.

[Ka61] Kazarinoff, N. D., *Geometric Inequalities*, The Mathematical Association of America, 1961.

[Ka88] Karasick, M., "On the representation and manipulation of rigid solids," Ph.D. thesis, School of Computer Science, McGill University, Montreal, 1988.

[Kl71] Klee, V., "Shapes of the future," *American Scientist*, vol. 59, January-February 1971, pp. 84-91.

[Kl39] Klein, F., *Elementary Mathematics from an Advanced Standpoint: Geometry*, Dover Publications, Inc., 1939.

[Kl89] Klein, R., *Concrete and Abstract Voronoi Diagrams*, Springer-Verlag, 1989.

[Ko86] Kostovskii, A., *Geometrical Constructions with Compasses Only*, Mir Publishers, Moscow, 1986.

[Kr92] Kreveld, M. van, *New Results on Data Structures in Computational Geometry*, University of Utrecht, 1992.

[KS92] Karasik, Y. B. and Sharir, M., "Optical computational geometry," *Proc. 8th ACM Symposium on Computational Geometry,* Berlin, June 10-12, 1992, pp. 232-241.

[Le02] Lemoine, E., *Geometrographie,* C. Naud, Paris, 1902.

[LPS88] Lenhart, W., Pollack, R., Sack, J., Seidel, R., Sharir, M., Suri, S., Toussaint, G., Whitesides, S. and Yap, C., "Computing the link center of a simple polygon," *Discrete & Computational Geometry*, vol. 3, 1988, pp. 281-293.

[LW86] Lee, D. T. and Wu, Y. F., "Geometric complexity of some location problems," *Algorithmica*, vol. 1, 1986, pp. 193-212.

[Ma1797] Mascheroni, L., *The Geometry of Compasses,* University of Pavia, 1797.

[MA79] McCallum, D. and Avis, D., "A linear time algorithm for finding the convex hull of a simple polygon," *Information Processing Letters,* vol. 8, 1979, pp. 201-205.

[Mi88] Milenkovic, V., "Verifiable implementations of geometric algorithms using finite precision arithmetic," Tech. Rept. CMU-CS-88-168, Carnegie Mellon University, July 1988.

[Mo1672] Mohr, G., *The Danish Euclid*, Amsterdam, 1672.

[Mo85] Mortenson, M. E., *Geometric Modeling*, John Wiley & Sons, 1985.

[MP69] Minsky, M. and Papert, S., *Perceptrons: An Introduction to Computational Geometry*, M.I.T. Press, 1969.

[MRB91] Melter, R. A., Rosenfeld, A. and Bhattacharya, P., eds., *Vision Geometry*, American Mathematical Society, 1991.

[Ni81] Niven, I., *Maxima and Minima Without Calculus,* The Mathematical Association of America, 1981.

[NS79] Newman, W. M. and Sproull, R. F., *Principles of Interactive Computer Graphics,* McGraw-Hill, New York, 1979.

[O'R87] O'Rourke, J., *Art Gallery Theorems and Algorithms,* Oxford University Press, 1987.

[OTU87] Ottmann, T., Thiemt, G., and Ulrich, C., "Numerical stability of geometric algorithms," *Proc. 3rd Symposium on Computational Geometry*, Waterloo, June 1987, pp. 119-125.

[PS85] Preparata, F. P. and Shamos, M. I., *Computational Geometry*, Springer-Verlag, New York, 1985.

[Ra60] Ransom, W. R., *Can and Can't in Geometry*, J. Weston Walch, Portland, Maine, 1960.

[RT90] Robert, J. M. and Toussaint, G. T., "Computational geometry and facility location," *Proc. International Conference on Operations Research and Management Science*, Manila, The Philippines, Dec. 11-15, 1990, pp. B-1 to B-19.

[SA48] Stark, M. E. and Archibald, R. C., "Jacob Steiner's geometrical constructions with a ruler given a fixed circle with its center," (translated from the German 1833 edition), *Scripta Mathematica*, vol. 14, 1948, pp. 189-264.

[Sc92] Schipper, H., "Determining contractibility of curves," *Proc. 8th ACM Symposium on Computational Geometry,* Berlin, June 10-12, 1992, pp. 358-367.

[Sh75] Shamos, M. I., "Geometric complexity," *Proc. 7th ACM Symposium on the Theory of Computing*, 1975, pp. 224-233.

[Sh78] Shamos, M. I., "Computational geometry," Ph.D. thesis, Yale University, 1978.

[Sm61] Smogorzhevskii, A. S., *The Ruler in Geometrical Constructions,* Blaisdell, New York, 1961.

[SSH87] Schwartz, J., Sharir, M. and Hopcroft, J., eds., *Planning Geometry and Complexity of*

Robot Motion, Ablex Publishing Corporation, Norwood, New Jersey, 1987.

[St91] Stolfi, J., *Oriented Projective Geometry: A Framework for Geometric Computations*, Academic Press, Inc., San Diego, 1991.

[St86] Stone, M. G., "A mnemonic for areas of polygons," *American Mathematical Monthly,* June-July, 1986, pp. 479-480.

[Su86] Sugihara, K., *Machine Interpretation of Line Drawings*, MIT Press, Cambridge, 1986.

[Su87] Sugihara, K., "An approach to error-free solid modelling," Notes, *Institute for Mathematics and its Applications*, University of Minnesota, 1987.

[SY87] Schwartz, J. and Yap, C. K., eds., *Algorithmic and Geometric Robotics*, Lawrence Earlbaum Assoc., Hillsdale, New Jersey, 1987.

[To80] Toussaint, G. T., "Pattern recognition and geometrical complexity," *Proc. Fifth International Conf. on Pattern Recognition*, Miami Beach, December 1980.

[To85] Toussaint, G. T., "A historical note on convex hull finding algorithms," *Pattern Recognition Letters,* vol. 3, January 1985, pp. 21-28.

[To85b] Toussaint, G. T., "Movable separability of sets," in *Computational Morphology,* G. T. Toussaint, ed., North-Holland, 1985, pp.335-375.

[To86] Toussaint, G. T., "An optimal algorithm for computing the relative convex hull of a set of points in a polygon," *Signal Processing III: Theories and Applications, Proc. EURASIP-86, Part 2,* North-Holland, September 1986, pp. 853-856.

[To89] Toussaint, G. T., "Computing geodesic properties inside a simple polygon," *Revue D'Intelligence Artificielle,* Vol. 3, No. 2, 1989, pp. 9-42.

[To91] Toussaint, G. T., "Efficient triangulation of simple polygons," *The Visual Computer*, vol. 7, No. 5-5, September 1991, pp. 280-295.

[To92a] Toussaint, G. T., "A new look at Euclid's second proposition," *The Mathematical Intelligencer*, in press, 1992.

[To92b] Toussaint, G. T., "Un nuevo vistazo a la segunda proposición de Euclides," *III Coloquio Internacional de Filosofía e Historia de las Matemáticas*, México City, June 22-26, 1992.

[TSRB71] Toregas, C., Swain, R., Revelle, C., and Bergman, L., "The location of emergency service facilities," *Operations Research,* vol. 19, 1971, pp. 1363-1373.

[Wh85] Whitesides, S., "Computational geometry and motion planning," in *Computational Geometry*, G. T. Toussaint, ed., North-Holland, 1985, pp. 377-427.

[Wh92] Whitesides, S., "Algorithmic issues in the geometry of planar linkage movement," *The Australian Computer Journal*, vol. 24, No. 2, May 1992, pp. 50-58.

[Wo85] Wood, D., "An isothetic view of computational geometry," in *Computational Geometry*, G. T. Toussaint, ed., North-Holland, 1985, pp. 429-459.