1 Computational Geometry

Imagine you are walking on the campus of a university and suddenly you realize you have to make an urgent phone call. There are many public phones on campus and of course you want to go to the nearest one. But which one is the nearest? It would be helpful to have a map on which you could look up the nearest public phone, wherever on campus you are. The map should show a subdivision of the campus into regions, and for each region indicate the nearest public phone. What would these regions look like? And how could we compute them?

Even though this is not such a terribly important issue, it describes the basics of a fundamental geometric concept, which plays a role in many applications. The subdivision of the campus is a so-called *Voronoi diagram*, and it will be studied in Chapter 7 in this book. It can be used to model trading areas of different cities, to guide robots, and even to describe and simulate the growth of crystals. Computing a geometric structure like a Voronoi diagram requires geometric algorithms. Such algorithms form the topic of this book.

A second example. Assume you located the closest public phone. With a campus map in hand you will probably have little problem in getting to the phone along a reasonably short path, without hitting walls and other objects. But programming a robot to perform the same task is a lot more difficult. Again, the heart of the problem is geometric: given a collection of geometric obstacles, we have to find a short connection between two points, avoiding collisions with the obstacles. Solving this so-called *motion planning* problem is of crucial importance in robotics. Chapters 13 and 15 deal with geometric algorithms required for motion planning.

A third example. Assume you don't have one map but two: one with a description of the various buildings, including the public phones, and one indicating the roads on the campus. To plan a motion to the public phone we have to *overlay* these maps, that is, we have to combine the information in the two maps. Overlaying maps is one of the basic operations of geographic information systems. It involves locating the position of objects from one map in the other, computing the intersection of various features, and so on. Chapter 2 deals with this problem.
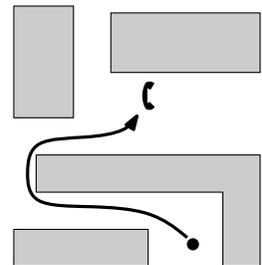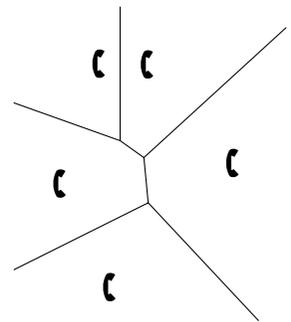
These are just three examples of geometric problems requiring carefully designed geometric algorithms for their solution. In the 1970s the field of computational geometry emerged, dealing with such geometric problems. It can be defined as the systematic study of algorithms and data structures for geometric objects, with a focus on exact algorithms that are asymptotically fast. Many researchers were attracted by the challenges posed by the geometric problems. The road from problem formulation to efficient and elegant solutions has often been long, with many difficult and sub-optimal intermediate results. Today there is a rich collection of geometric algorithms that are efficient, and relatively easy to understand and implement.

This book describes the most important notions, techniques, algorithms, and data structures from computational geometry in a way that we hope will be attractive to readers who are interested in applying results from computational geometry. Each chapter is motivated with a real computational problem that requires geometric algorithms for its solution. To show the wide applicability of computational geometry, the problems were taken from various application areas: robotics, computer graphics, CAD/CAM, and geographic information systems.

You should not expect ready-to-implement software solutions for major problems in the application areas. Every chapter deals with a single concept in computational geometry; the applications only serve to introduce and motivate the concepts. They also illustrate the process of modeling an engineering problem and finding an exact solution.

1.1 An Example: Convex Hulls

Good solutions to algorithmic problems of a geometric nature are mostly based on two ingredients. One is a thorough understanding of the geometric properties of the problem, the other is a proper application of algorithmic techniques and data structures. If you don't understand the geometry of the problem, all the algorithms of the world won't help you to solve it efficiently. On the other hand, even if you perfectly understand the geometry of the problem, it is hard to solve it effectively if you don't know the right algorithmic techniques. This book will give you a thorough understanding of the most important geometric concepts and algorithmic techniques.

To illustrate the issues that arise in developing a geometric algorithm, this section deals with one of the first problems that was studied in computational geometry: the computation of planar convex hulls. We'll skip the motivation for this problem here; if you are interested you can read the introduction to Chapter 11, where we study convex hulls in 3-dimensional space.


convex   not convex
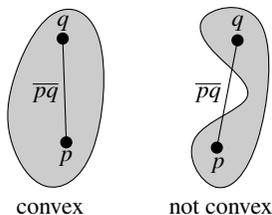
A subset S of the plane is called *convex* if and only if for any pair of points $p, q \in S$ the line segment $\overline{pq}$ is completely contained in S. The *convex hull* $\mathcal{CH}(S)$ of a set S is the smallest convex set that contains S. To be more precise, it is the intersection of all convex sets that contain S.

We will study the problem of computing the convex hull of a finite set P of n points in the plane. We can visualize what the convex hull looks like by a thought experiment. Imagine that the points are nails sticking out of the plane, take an elastic rubber band, hold it around the nails, and let it go. It will snap around the nails, minimizing its length. The area enclosed by the rubber band is the convex hull of P. This leads to an alternative definition of the convex hull of a finite set P of points in the plane: it is the unique convex polygon whose vertices are points from P and that contains all points of P. Of course we should prove rigorously that this is well defined—that is, that the polygon is unique—and that the definition is equivalent to the one given earlier, but let's skip that in this introductory chapter.
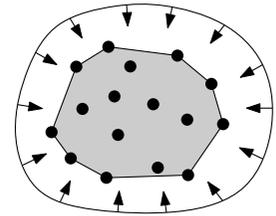
How do we compute the convex hull? Before we can answer this question we must ask another question: what does it mean to compute the convex hull? As we have seen, the convex hull of P is a convex polygon. A natural way to represent a polygon is by listing its vertices in clockwise order, starting with an arbitrary one. So the problem we want to solve is this: given a set $P = \{p_1, p_2, \ldots, p_n\}$ of points in the plane, compute a list that contains those points from P that are the vertices of $\mathcal{CH}(P)$, listed in clockwise order.

input = set of points:

$p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9$

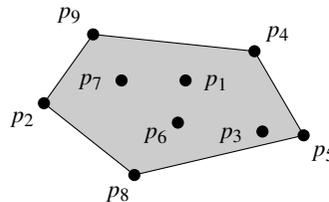output = representation of the convex hull:

p_4, p_5, p_8, p_2, p_9

Figure 1.1
Computing a convex hull

The first definition of convex hulls is of little help when we want to design an algorithm to compute the convex hull. It talks about the intersection of all convex sets containing P, of which there are infinitely many. The observation that $\mathcal{CH}(P)$ is a convex polygon is more useful. Let's see what the edges of $\mathcal{CH}(P)$ are. Both endpoints p and q of such an edge are points of P, and if we direct the line through p and q such that $\mathcal{CH}(P)$ lies to the right, then all the points of P must lie to the right of this line. The reverse is also true: if all points of $P \setminus \{p, q\}$ lie to the right of the directed line through p and q, then $\overline{pq}$ is an edge of $\mathcal{CH}(P)$.
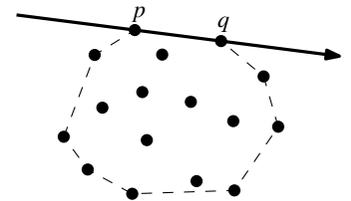
Now that we understand the geometry of the problem a little bit better we can develop an algorithm. We will describe it in a style of pseudocode we will use throughout this book.

Algorithm SLOWCONVEXHULL(P)
Input. A set P of points in the plane.
Output. A list $\mathcal{L}$ containing the vertices of $\mathcal{CH}(P)$ in clockwise order.
1. $E \leftarrow \emptyset$.
2. **for** all ordered pairs $(p, q) \in P \times P$ with p not equal to q
3.   **do** *valid* $\leftarrow$ **true**

4.   **for** all points $r \in P$ not equal to p or q
5.   **do if** r lies to the left of the directed line from p to q
6.   **then** *valid* $\leftarrow$ **false**.
7.   **if** *valid* **then** Add the directed edge $\vec{pq}$ to E.
8.   From the set E of edges construct a list $\mathcal{L}$ of vertices of $\mathcal{CH}(P)$, sorted in clockwise order.

Two steps in the algorithm are perhaps not entirely clear.

    The first one is line 5: how do we test whether a point lies to the left or to the right of a directed line? This is one of the primitive operations required in most geometric algorithms. Throughout this book we assume that such operations are available. It is clear that they can be performed in constant time so the actual implementation will not affect the asymptotic running time in order of magnitude. This is not to say that such primitive operations are unimportant or trivial. They are not easy to implement correctly and their implementation will affect the actual running time of the algorithm. Fortunately, software libraries containing such primitive operations are nowadays available. We conclude that we don't have to worry about the test in line 5; we may assume that we have a function available performing the test for us in constant time.

    The other step of the algorithm that requires some explanation is the last one. In the loop of lines 2–7 we determine the set E of convex hull edges. From E we can construct the list $\mathcal{L}$ as follows. The edges in E are directed, so we can speak about the origin and the destination of an edge. Because the edges are directed such that the other points lie to their right, the destination of an edge comes after its origin when the vertices are listed in clockwise order. Now remove an arbitrary edge $\vec{e}_1$ from E. Put the origin of $\vec{e}_1$ as the first point into $\mathcal{L}$, and the destination as the second point. Find the edge $\vec{e}_2$ in E whose origin is the destination of $\vec{e}_1$, remove it from E, and append its destination to $\mathcal{L}$. Next, find the edge $\vec{e}_3$ whose origin is the destination of $\vec{e}_2$, remove it from E, and append its destination to $\mathcal{L}$. We continue in this manner until there is only one edge left in E. Then we are done; the destination of the remaining edge is necessarily the origin of $\vec{e}_1$, which is already the first point in $\mathcal{L}$. A simple implementation of this procedure takes $O(n^2)$ time. This can easily be improved to $O(n \log n)$, but the time required for the rest of the algorithm dominates the total running time anyway.

    Analyzing the time complexity of SLOWCONVEXHULL is easy. We check $n^2 - n$ pairs of points. For each pair we look at the $n - 2$ other points to see whether they all lie on the right side. This will take $O(n^3)$ time in total. The final step takes $O(n^2)$ time, so the total running time is $O(n^3)$. An algorithm with a cubic running time is too slow to be of practical use for anything but the smallest input sets. The problem is that we did not use any clever algorithmic design techniques; we just translated the geometric insight into an algorithm in a brute-force manner. But before we try to do better, it is useful to make several observations about this algorithm.

We have been a bit careless when deriving the criterion of when a pair p, q defines an edge of $\mathcal{CH}(P)$. A point r does not always lie to the right or to the

destination of $\vec{e}_1$
= origin of $\vec{e}_2$

$\vec{e}_1$

$\vec{e}_2$

origin of $\vec{e}_1$
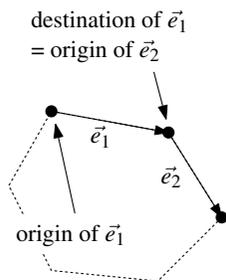
left of the line through p and q, it can also happen that it lies *on* this line. What should we do then? This is what we call a *degenerate case*, or a *degeneracy* for short. We prefer to ignore such situations when we first think about a problem, so that we don't get confused when we try to figure out the geometric properties of a problem. However, these situations do arise in practice. For instance, if we create the points by clicking on a screen with a mouse, all points will have small integer coordinates, and it is quite likely that we will create three points on a line.

To make the algorithm correct in the presence of degeneracies we must reformulate the criterion above as follows: a directed edge $\vec{pq}$ is an edge of $\mathcal{CH}(P)$ if and only if all other points $r \in P$ lie either strictly to the right of the directed line through p and q, or they lie on the open line segment $\overline{pq}$. (We assume that there are no coinciding points in P.) So we have to replace line 5 of the algorithm by this more complicated test.

We have been ignoring another important issue that can influence the correctness of the result of our algorithm. We implicitly assumed that we can somehow test exactly whether a point lies to the right or to the left of a given line. This is not necessarily true: if the points are given in floating point coordinates and the computations are done using floating point arithmetic, then there will be rounding errors that may distort the outcome of tests.

Imagine that there are three points p, q, and r, that are nearly collinear, and that all other points lie far to the right of them. Our algorithm tests the pairs (p,q), (r,q), and (p,r). Since these points are nearly collinear, it is possible that the rounding errors lead us to decide that r lies to the right of the line from p to q, that p lies to the right of the line from r to q, and that q lies to the right of the line from p to r. Of course this is geometrically impossible—but the floating point arithmetic doesn't know that! In this case the algorithm will accept all three edges. Even worse, all three tests could give the opposite answer, in which case the algorithm rejects all three edges, leading to a gap in the boundary of the convex hull. And this leads to a serious problem when we try to construct the sorted list of convex hull vertices in the last step of our algorithm. This step assumes that there is exactly one edge starting in every convex hull vertex, and exactly one edge ending there. Due to the rounding errors there can suddenly be two, or no, edges starting in vertex p. This can cause the program implementing our simple algorithm to crash, since the last step has not been designed to deal with such inconsistent data.

Although we have proven the algorithm to be correct and to handle all special cases, it is not *robust*: small errors in the computations can make it fail in completely unexpected ways. The problem is that we have proven the correctness assuming that we can compute exactly with real numbers.

We have designed our first geometric algorithm. It computes the convex hull of a set of points in the plane. However, it is quite slow—its running time is $O(n^3)$—, it deals with degenerate cases in an awkward way, and it is not robust. We should try to do better.

upper hull

p_1

p_n

lower hull

p_i

points deleted
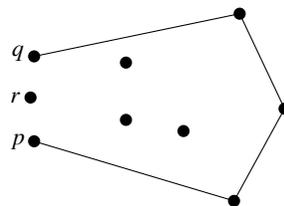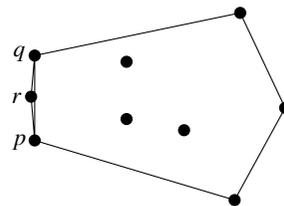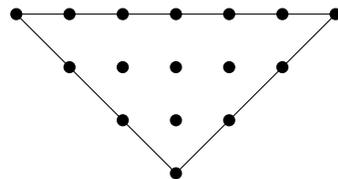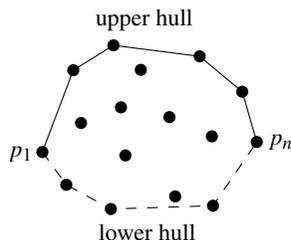
To this end we apply a standard algorithmic design technique: we will develop an *incremental algorithm*. This means that we will add the points in P one by one, updating our solution after each addition. We give this incremental approach a geometric flavor by adding the points from left to right. So we first sort the points by x-coordinate, obtaining a sorted sequence $p_1, \ldots, p_n$, and then we add them in that order. Because we are working from left to right, it would be convenient if the convex hull vertices were also ordered from left to right as they occur along the boundary. But this is not the case. Therefore we first compute only those convex hull vertices that lie on the *upper hull*, which is the part of the convex hull running from the leftmost point p_1 to the rightmost point p_n when the vertices are listed in clockwise order. In other words, the upper hull contains the convex hull edges bounding the convex hull from above. In a second scan, which is performed from right to left, we compute the remaining part of the convex hull, the *lower hull*.
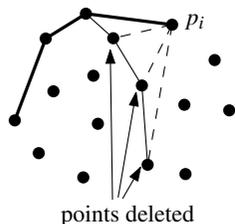
The basic step in the incremental algorithm is the update of the upper hull after adding a point p_i. In other words, given the upper hull of the points $p_1, \ldots, p_{i-1}$, we have to compute the upper hull of $p_1, \ldots, p_i$. This can be done as follows. When we walk around the boundary of a polygon in clockwise order, we make a turn at every vertex. For an arbitrary polygon this can be both a right turn and a left turn, but for a convex polygon every turn must be a right turn. This suggests handling the addition of p_i in the following way. Let $\mathcal{L}_{\text{upper}}$ be a list that stores the upper vertices in left-to-right order. We first append p_i to $\mathcal{L}_{\text{upper}}$. This is correct because p_i is the rightmost point of the ones added so far, so it must be on the upper hull. Next, we check whether the last three points in $\mathcal{L}_{\text{upper}}$ make a right turn. If this is the case there is nothing more to do; $\mathcal{L}_{\text{upper}}$ contains the vertices of the upper hull of $p_1, \ldots, p_i$, and we can proceed to the next point, p_{i+1}. But if the last three points make a left turn, we have to delete the middle one from the upper hull. In this case we are not finished yet: it could be that the new last three points still do not make a right turn, in which case we again have to delete the middle one. We continue in this manner until the last three points make a right turn, or until there are only two points left.

We now give the algorithm in pseudocode. The pseudocode computes both the upper hull and the lower hull. The latter is done by treating the points from right to left, analogous to the computation of the upper hull.

Algorithm CONVEXHULL(P)
Input. A set P of points in the plane.
Output. A list containing the vertices of $\mathcal{CH}(P)$ in clockwise order.
1.   Sort the points by x-coordinate, resulting in a sequence $p_1, \ldots, p_n$.
2.   Put the points p_1 and p_2 in a list $\mathcal{L}_{\text{upper}}$, with p_1 as the first point.
3.   **for** $i \leftarrow 3$ **to** n
4.   **do** Append p_i to $\mathcal{L}_{\text{upper}}$.
5.   **while** $\mathcal{L}_{\text{upper}}$ contains more than two points **and** the last three points in $\mathcal{L}_{\text{upper}}$ do not make a right turn
6.   **do** Delete the middle of the last three points from $\mathcal{L}_{\text{upper}}$.
7.   Put the points p_n and p_{n-1} in a list $\mathcal{L}_{\text{lower}}$, with p_n as the first point.

8.　　**for** $i \leftarrow n - 2$ **downto** 1
9.　　　　**do** Append p_i to $\mathcal{L}_{\text{lower}}$.
10.　　　　　　**while** $\mathcal{L}_{\text{lower}}$ contains more than 2 points **and** the last three points
　　　　　　　　　　　in $\mathcal{L}_{\text{lower}}$ do not make a right turn
11.　　　　　　　　**do** Delete the middle of the last three points from $\mathcal{L}_{\text{lower}}$.
12.　　Remove the first and the last point from $\mathcal{L}_{\text{lower}}$ to avoid duplication of the
　　　points where the upper and lower hull meet.
13.　　Append $\mathcal{L}_{\text{lower}}$ to $\mathcal{L}_{\text{upper}}$, and call the resulting list $\mathcal{L}$.
14.　　**return** $\mathcal{L}$

Once again, when we look closer we realize that the above algorithm is not correct. Without mentioning it, we made the assumption that no two points have the same x-coordinate. If this assumption is not valid the order on x-coordinate is not well defined. Fortunately, this turns out not to be a serious problem. We only have to generalize the ordering in a suitable way: rather than using only the x-coordinate of the points to define the order, we use the lexicographic order. This means that we first sort by x-coordinate, and if points have the same x-coordinate we sort them by y-coordinate.

　　　Another special case we have ignored is that the three points for which we have to determine whether they make a left or a right turn lie on a straight line. In this case the middle point should not occur on the convex hull, so collinear points must be treated as if they make a left turn. In other words, we should use a test that returns true if the three points make a right turn, and false otherwise. (Note that this is simpler than the test required in the previous algorithm when there were collinear points.)

　　　With these modifications the algorithm correctly computes the convex hull: the first scan computes the upper hull, which is now defined as the part of the convex hull running from the lexicographically smallest vertex to the lexicographically largest vertex, and the second scan computes the remaining part of the convex hull.

What does our algorithm do in the presence of rounding errors in the floating point arithmetic? When such errors occur, it can happen that a point is removed from the convex hull although it should be there, or that a point inside the real convex hull is not removed. But the structural integrity of the algorithm is unharmed: it will compute a closed polygonal chain. After all, the output is a list of points that we can interpret as the clockwise listing of the vertices of a polygon, and any three consecutive points form a right turn or, because of the rounding errors, they almost form a right turn. Moreover, no point in P can be far outside the computed hull. The only problem that can still occur is that, when three points lie very close together, a turn that is actually a sharp left turn can be interpreted as a right turn. This might result in a dent in the resulting polygon. A way out of this is to make sure that points in the input that are very close together are considered as being the same point, for example by rounding. Hence, although the result need not be exactly correct—but then, we cannot hope for an exact result if we use inexact arithmetic—it does make sense. For many applications this is good enough. Still, it is wise to be careful in the implementation of the basic test to avoid errors as much as possible.

not a right turn
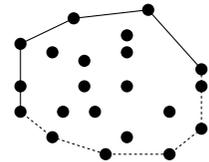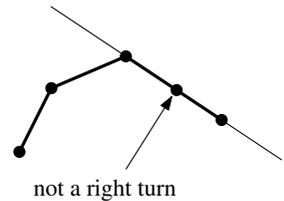
We conclude with the following theorem:

Theorem 1.1 *The convex hull of a set of n points in the plane can be computed in $O(n \log n)$ time.*

Proof. We will prove the correctness of the computation of the upper hull; the lower hull computation can be proved correct using similar arguments. The proof is by induction on the number of point treated. Before the **for**-loop starts, the list $\mathcal{L}_{upper}$ contains the points p_1 and p_2, which trivially form the upper hull of $\{p_1, p_2\}$. Now suppose that $\mathcal{L}_{upper}$ contains the upper hull vertices of $\{p_1, \ldots, p_{i-1}\}$ and consider the addition of p_i. After the execution of the **while**-loop and because of the induction hypothesis, we know that the points in $\mathcal{L}_{upper}$ form a chain that only makes right turns. Moreover, the chain starts at the lexicographically smallest point of $\{p_1, \ldots, p_i\}$ and ends at the lexicographically largest point, namely p_i. If we can show that all points of $\{p_1, \ldots, p_i\}$ that are not in $\mathcal{L}_{upper}$ are below the chain, then $\mathcal{L}_{upper}$ contains the correct points. By induction we know there is no point above the chain that we had before p_i was added. Since the old chain lies below the new chain, the only possibility for a point to lie above the new chain is if it lies in the vertical slab between p_{i-1} and p_i. But this is not possible, since such a point would be in between p_{i-1} and p_i in the lexicographical order. (You should verify that a similar argument holds if p_{i-1} and p_i, or any other points, have the same x-coordinate.)

To prove the time bound, we note that sorting the points lexicographically can be done in $O(n \log n)$ time. Now consider the computation of the upper hull. The **for**-loop is executed a linear number of times. The question that remains is how often the **while**-loop inside it is executed. For each execution of the **for**-loop the **while**-loop is executed at least once. For any extra execution a point is deleted from the current hull. As each point can be deleted only once during the construction of the upper hull, the total number of extra executions over all **for**-loops is bounded by n. Similarly, the computation of the lower hull takes $O(n)$ time. Due to the sorting step, the total time required for computing the convex hull is $O(n \log n)$. $\qquad\square$

The final convex hull algorithm is simple to describe and easy to implement. It only requires lexicographic sorting and a test whether three consecutive points make a right turn. From the original definition of the problem it was far from obvious that such an easy and efficient solution would exist.

1.2 Degeneracies and Robustness

As we have seen in the previous section, the development of a geometric algorithm often goes through three phases.

In the first phase, we try to ignore everything that will clutter our understanding of the geometric concepts we are dealing with. Sometimes collinear points are a nuisance, sometimes vertical line segments are. When first trying to design or understand an algorithm, it is often helpful to ignore these degenerate cases.

empty region

p_i

p_{i-1}
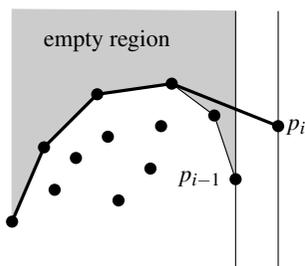
In the second phase, we have to adjust the algorithm designed in the first phase to be correct in the presence of degenerate cases. Beginners tend to do this by adding a huge number of case distinctions to their algorithms. In many situations there is a better way. By considering the geometry of the problem again, one can often integrate special cases with the general case. For example, in the convex hull algorithm we only had to use the lexicographical order instead of the order on *x*-coordinate to deal with points with equal *x*-coordinate. For most algorithms in this book we have tried to take this integrated approach to deal with special cases. Still, it is easier not to think about such cases upon first reading. Only after understanding how the algorithm works in the general case should you think about degeneracies.

If you study the computational geometry literature, you will find that many authors ignore special cases, often by formulating specific assumptions on the input. For example, in the convex hull problem we could have ignored special cases by simply stating that we assume that the input is such that no three points are collinear and no two points have the same *x*-coordinate. From a theoretical point of view, such assumptions are usually justified: the goal is then to establish the computational complexity of a problem and, although it is tedious to work out the details, degenerate cases can almost always be handled without increasing the asymptotic complexity of the algorithm. But special cases definitely increase the complexity of the implementations. Most researchers in computational geometry today are aware that their *general position* assumptions are not satisfied in practical applications and that an integrated treatment of the special cases is normally the best way to handle them. Furthermore, there are general techniques—so-called *symbolic perturbation schemes*—that allow one to ignore special cases during the design and implementation, and still have an algorithm that is correct in the presence of degeneracies.

The third phase is the actual implementation. Now one needs to think about the primitive operations, like testing whether a point lies to the left, to the right, or on a directed line. If you are lucky you have a geometric software library available that contains the operations you need, otherwise you must implement them yourself.

Another issue that arises in the implementation phase is that the assumption of doing exact arithmetic with real numbers breaks down, and it is necessary to understand the consequences. Robustness problems are often a cause of frustration when implementing geometric algorithms. Solving robustness problems is not easy. One solution is to use a package providing exact arithmetic (using integers, rationals, or even algebraic numbers, depending on the type of problem) but this will be slow. Alternatively, one can adapt the algorithm to detect inconsistencies and take appropriate actions to avoid crashing the program. In this case it is not guaranteed that the algorithm produces the correct output, and it is important to establish the exact properties that the output has. This is what we did in the previous section, when we developed the convex hull algorithm: the result might not be a convex polygon but we know that the structure of the output is correct and that the output polygon is very close to the convex hull. Finally, it is possible to predict, based on the input, the precision in

the number representation required to solve the problem correctly.

Which approach is best depends on the application. If speed is not an issue, exact arithmetic is preferred. In other cases it is not so important that the result of the algorithm is precise. For example, when displaying the convex hull of a set of points, it is most likely not noticeable when the polygon deviates slightly from the true convex hull. In this case we can use a careful implementation based on floating point arithmetic.

In the rest of this book we focus on the design phase of geometric algorithms; we won't say much about the problems that arise in the implementation phase.

1.3 Application Domains

As indicated before, we have chosen a motivating example application for every geometric concept, algorithm, or data structure introduced in this book. Most of the applications stem from the areas of computer graphics, robotics, geographic information systems, and CAD/CAM. For those not familiar with these fields, we give a brief description of the areas and indicate some of the geometric problems that arise in them.

Computer graphics. Computer graphics is concerned with creating images of modeled scenes for display on a computer screen, a printer, or other output device. The scenes vary from simple two-dimensional drawings—consisting of lines, polygons, and other primitive objects—to realistic-looking 3-dimensional scenes including light sources, textures, and so on. The latter type of scene can easily contain over a million polygons or curved surface patches.

Because scenes consist of geometric objects, geometric algorithms play an important role in computer graphics.

For 2-dimensional graphics, typical questions involve the intersection of certain primitives, determining the primitive pointed to with the mouse, or determining the subset of primitives that lie within a particular region. Chapters 6, 10, and 16 describe techniques useful for some of these problems.

When dealing with 3-dimensional problems the geometric questions become more complex. A crucial step in displaying a 3-dimensional scene is hidden surface removal: determine the part of a scene visible from a particular viewpoint or, in other words, discard the parts that lie behind other objects. In Chapter 12 we study one approach to this problem.

To create realistic-looking scenes we have to take light into account. This creates many new problems, such as the computation of shadows. Hence, realistic image synthesis requires complicated display techniques, like ray tracing and radiosity. When dealing with moving objects and in virtual reality applications, it is important to detect collisions between objects. All these situations involve geometric problems.

Robotics. The field of robotics studies the design and use of robots. As robots are geometric objects that operate in a 3-dimensional space—the real world—it

is obvious that geometric problems arise at many places. At the beginning of this chapter we already introduced the motion planning problem, where a robot has to find a path in an environment with obstacles. In Chapters 13 and 15 we study some simple cases of motion planning. Motion planning is one aspect of the more general problem of task planning. One would like to give a robot high-level tasks—"vacuum the room"—and let the robot figure out the best way to execute the task. This involves planning motions, planning the order in which to perform subtasks, and so on.

Other geometric problems occur in the design of robots and work cells in which the robot has to operate. Most industrial robots are robot arms with a fixed base. The parts operated on by the robot arm have to be supplied in such a way that the robot can easily grasp them. Some of the parts may have to be immobilized so that the robot can work on them. They may also have to be turned to a known orientation before the robot can work on them. These are all geometric problems, sometimes with a kinematic component. Some of the algorithms described in this book are applicable in such problems. For example, the smallest enclosing disc problem, treated in Section 4.7, can be used for optimal placement of robot arms.
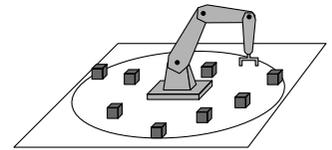
Geographic information systems.   A geographic information system, or GIS for short, stores geographical data like the shape of countries, the height of mountains, the course of rivers, the type of vegetation at different locations, population density, or rainfall. They can also store human-made structures such as cities, roads, railways, electricity lines, or gas pipes. A GIS can be used to extract information about certain regions and, in particular, to obtain information about the relation between different types of data. For example, a biologist may want to relate the average rainfall to the existence of certain plants, and a civil engineer may need to query a GIS to determine whether there are any gas pipes underneath a lot where excavation works are to be performed.

As most geographic information concerns properties of points and regions on the earth's surface, geometric problems occur in abundance here. Moreover, the amount of data is so large that efficient algorithms are a must. Below we mention the GIS-related problems treated in this book.

A first question is how to store geographic data. Suppose that we want to develop a car guidance system, which shows the driver at any moment where she is. This requires storing a huge map of roads and other data. At every moment we have to be able to determine the position of the car on the map and to quickly select a small portion of the map for display on the on-board computer. Efficient data structures are needed for these operations. Chapters 6, 10, and 16 describe computational geometry solutions to these problems.

The information about the height in some mountainous terrain is usually only available at certain sample points. For other positions we have to obtain the heights by interpolating between nearby sample points. But which sample points should we choose? Chapter 9 deals with this problem.

The combination of different types of data is one of the most important operations in a GIS. For example, we may want to check which houses lie in
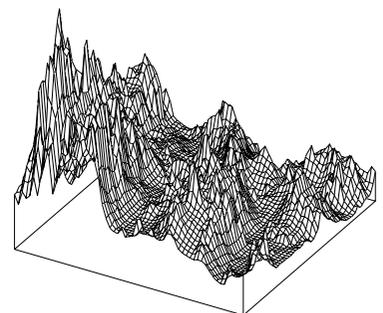
a forest, locate all bridges by checking where roads cross rivers, or determine a good location for a new golf course by finding a slightly hilly, rather cheap area not too far from a particular town. A GIS usually stores different types of data in separate maps. To combine the data we have to overlay different maps. Chapter 2 deals with a problem arising when we want to compute the overlay.

Finally, we mention the same example we gave at the beginning of this chapter: the location of the nearest public phone (or hospital, or any other facility). This requires the computation of a Voronoi diagram, a structure studied in detail in Chapter 7.

CAD/CAM.   Computer aided design (CAD) concerns itself with the design of products with a computer. The products can vary from printed circuit boards, machine parts, or furniture, to complete buildings. In all cases the resulting product is a geometric entity and, hence, it is to be expected that all sorts of geometric problems appear. Indeed, CAD packages have to deal with intersections and unions of objects, with decomposing objects and object boundaries into simpler shapes, and with visualizing the designed products.

To decide whether a design meets the specifications certain tests are needed. Often one does not need to build a prototype for these tests, and a simulation suffices. Chapter 14 deals with a problem arising in the simulation of heat emission by a printed circuit board.

Once an object has been designed and tested, it has to be manufactured. Computer aided manufacturing (CAM) packages can be of assistance here. CAM involves many geometric problems. Chapter 4 studies one of them.

A recent trend is *design for assembly*, where assembly decisions are already taken into account during the design stage. A CAD system supporting this would allow designers to test their design for feasibility, answering questions like: can the product be built easily using a certain manufacturing process? Many of these questions require geometric algorithms to be answered.

Other applications domains.   There are many more application domains where geometric problems occur and geometric algorithms and data structures can be used to solve them.

For example, in molecular modeling, molecules are often represented by collections of intersecting balls in space, one ball for each atom. Typical questions are to compute the union of the atom balls to obtain the molecule surface, or to compute where two molecules can touch each other.

Another area is pattern recognition. Consider for example an optical character recognition system. Such a system scans a paper with text on it with the goal of recognizing the text characters. A basic step is to match the image of a character against a collection of stored characters to find the one that best fits it. This leads to a geometric problem: given two geometric objects, determine how well they resemble each other.

Even certain areas that at first sight do not seem to be geometric can benefit from geometric algorithms, because it is often possible to formulate non-geometric problem in geometric terms. In Chapter 5, for instance, we will see
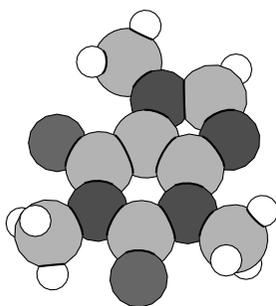
caffeine

how records in a database can be interpreted as points in a higher-dimensional space, and we will present a geometric data structure such that certain queries on the records can be answered efficiently.

We hope that the above collection of geometric problems makes it clear that computational geometry plays a role in many different areas of computer science. The algorithms, data structures, and techniques described in this book will provide you with the tools needed to attack such geometric problems successfully.

1.4 Notes and Comments

Every chapter of this book ends with a section entitled *Notes and Comments*. These sections indicate where the results described in the chapter came from, indicate generalizations and improvements, and provide references. They can be skipped but do contain useful material for those who want to know more about the topic of the chapter. More information can also be found in the *Handbook of Computational Geometry* [331] and the *Handbook of Discrete and Computational Geometry* [191].

In this chapter the geometric problem treated in detail was the computation of the convex hull of a set of points in the plane. This is a classic topic in computational geometry and the amount of literature about it is huge. The algorithm described in this chapter is commonly known as *Graham's scan*, and is based on a modification by Andrew [17] of one of the earliest algorithms by Graham [192]. This is only one of the many $O(n \log n)$ algorithms available for solving the problem. A divide-and-conquer approach was given by Preparata and Hong [322]. Also an incremental method exists that inserts the points one by one in $O(\log n)$ time per insertion [321]. Overmars and van Leeuwen generalized this to a method in which points could be both inserted and deleted in $O(\log^2 n)$ time [305]. Other results on dynamic convex hulls were obtained by Hershberger and Suri [211], Chan [83], and Brodal and Jacob [73].

Even though an $\Omega(n \log n)$ lower bound is known for the problem [393] many authors have tried to improve the result. This makes sense because in many applications the number of points that appear on the convex hull is relatively small, while the lower bound result assumes that (almost) all points show up on the convex hull. Hence, it is useful to look at algorithms whose running time depends on the complexity of the convex hull. Jarvis [221] introduced a wrapping technique, often referred to as *Jarvis's march*, that computes the convex hull in $O(h \cdot n)$ time where h is the complexity of the convex hull. The same worst-case performance is achieved by the algorithm of Overmars and van Leeuwen [303], based on earlier work by Bykat [79], Eddy [156], and Green and Silverman [193]. This algorithm has the advantage that its expected running time is linear for many distributions of points. Finally, Kirkpatrick and Seidel [238] improved the result to $O(n \log h)$, and recently Chan [82] discovered a much simpler algorithm to achieve the same result.

The convex hull can be defined in any dimension. Convex hulls in 3-dimensional space can still be computed in $O(n \log n)$ time, as we will see in Chapter 11. For dimensions higher than 3, however, the complexity of the convex hull is no longer linear in the number of points. See the notes and comments of Chapter 11 for more details.

In the past years a number of general methods for handling special cases have been suggested. These *symbolic perturbation schemes* perturb the input in such a way that all degeneracies disappear. However, the perturbation is only done symbolically. This technique was introduced by Edelsbrunner and Mücke [164] and later refined by Yap [397] and Emiris and Canny [172, 171]. Symbolic perturbation relieves the programmer of the burden of degeneracies, but it has some drawbacks: the use of a symbolic perturbation library slows down the algorithm, and sometimes one needs to recover the "real result" from the "perturbed result", which is not always easy. These drawbacks led Burnikel et al. [78] to claim that it is both simpler (in terms of programming effort) and more efficient (in terms of running time) to deal directly with degenerate inputs.

Robustness in geometric algorithms is a topic that has recently received a lot of interest. Most geometric comparisons can be formulated as computing the sign of some determinant. A possible way to deal with the inexactness in floating point arithmetic when evaluating this sign is to choose a small threshold value ε and to say that the determinant is zero when the outcome of the floating point computation is less than ε. When implemented naively, this can lead to inconsistencies (for instance, for three points a, b, c we may decide that $a = b$ and $b = c$ but $a \neq c$) that cause the program to fail. Guibas et al. [198] showed that combining such an approach with interval arithmetic and backwards error analysis can give robust algorithms. Another option is to use *exact arithmetic*. Here one computes as many bits of the determinant as are needed to determine its sign. This will slow down the computation, but techniques have been developed to keep the performance penalty relatively small [182, 256, 395]. Besides these general approaches, there have been a number papers dealing with robust computation in specific problems [34, 37, 81, 145, 180, 181, 219, 279].

We gave a brief overview of the application domains from which we took our examples, which serve to show the motivation behind the various geometric notions and algorithms studied in this book. Below are some references to textbooks you can consult if you want to know more about the application domains. Of course there are many more good books about these domains than the few we mention.

There is a large number of books on computer graphics. The book by Foley et al. [179] is very extensive and generally considered one of the best books on the topic. Other good books are the ones by Shirley et al. [359] and Watt [381].

An extensive overview of robotics and the motion planning problem can be found in the book of Choset et al. [127], and in the somewhat older books of Latombe [243] and Hopcroft, Schwartz, and Sharir [217]. More information on geometric aspects of robotics is provided by the book of Selig [348].

There is a large collection of books about geographic information systems, but most of them do not consider algorithmic issues in much detail. Some general textbooks are the ones by DeMers [140], Longley et al. [257], and Worboys and Duckham [392]. Data structures for spatial data are described extensively in the book of Samet [335].

The books by Faux and Pratt [175], Mortenson [285], and Hoffmann [216] are good introductory texts on CAD/CAM and geometric modeling.

1.5 Exercises

1.1 The convex hull of a set S is defined to be the intersection of all convex sets that contain S. For the convex hull of a set of points it was indicated that the convex hull is the convex set with smallest perimeter. We want to show that these are equivalent definitions.

   a. Prove that the intersection of two convex sets is again convex. This implies that the intersection of a finite family of convex sets is convex as well.
   b. Prove that the smallest perimeter polygon $\mathcal{P}$ containing a set of points P is convex.
   c. Prove that any convex set containing the set of points P contains the smallest perimeter polygon $\mathcal{P}$.

1.2 Let P be a set of points in the plane. Let $\mathcal{P}$ be the convex polygon whose vertices are points from P and that contains all points in P. Prove that this polygon $\mathcal{P}$ is uniquely defined, and that it is the intersection of all convex sets containing P.

1.3 Let E be an unsorted set of n segments that are the edges of a convex polygon. Describe an $O(n \log n)$ algorithm that computes from E a list containing all vertices of the polygon, sorted in clockwise order.

1.4 For the convex hull algorithm we have to be able to test whether a point r lies left or right of the directed line through two points p and q. Let $p = (p_x, p_y)$, $q = (q_x, q_y)$, and $r = (r_x, r_y)$.

   a. Show that the sign of the determinant

   $$D = \begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}$$

   determines whether r lies left or right of the line.
   b. Show that $|D|$ in fact is twice the surface of the triangle determined by p, q, and r.
   c. Why is this an attractive way to implement the basic test in algorithm CONVEXHULL? Give an argument for both integer and floating point coordinates.

1.5 Verify that the algorithm CONVEXHULL with the indicated modifications correctly computes the convex hull, also of degenerate sets of points. Consider for example such nasty cases as a set of points that all lie on one (vertical) line.

1.6 In many situations we need to compute convex hulls of objects other than points.

   a. Let S be a set of n line segments in the plane. Prove that the convex hull of S is exactly the same as the convex hull of the $2n$ endpoints of the segments.
   b.* Let $\mathcal{P}$ be a non-convex polygon. Describe an algorithm that computes the convex hull of $\mathcal{P}$ in $O(n)$ time. *Hint:* Use a variant of algorithm CONVEXHULL where the vertices are not treated in lexicographical order, but in some other order.

1.7 Consider the following alternative approach to computing the convex hull of a set of points in the plane: We start with the rightmost point. This is the first point p_1 of the convex hull. Now imagine that we start with a vertical line and rotate it clockwise until it hits another point p_2. This is the second point on the convex hull. We continue rotating the line but this time around p_2 until we hit a point p_3. In this way we continue until we reach p_1 again.
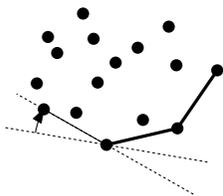
   a. Give pseudocode for this algorithm.
   b. What degenerate cases can occur and how can we deal with them?
   c. Prove that the algorithm correctly computes the convex hull.
   d. Prove that the algorithm can be implemented to run in time $O(n \cdot h)$, where h is the complexity of the convex hull.
   e. What problems might occur when we deal with inexact floating point arithmetic?

1.8 The $O(n \log n)$ algorithm to compute the convex hull of a set of n points in the plane that was described in this chapter is based on the paradigm of incremental construction: add the points one by one, and update the convex hull after each addition. In this exercise we shall develop an algorithm based on another paradigm, namely divide-and-conquer.

   a. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two disjoint convex polygons with n vertices in total. Give an $O(n)$ time algorithm that computes the convex hull of $\mathcal{P}_1 \cup \mathcal{P}_2$.
   b. Use the algorithm from part a to develop an $O(n \log n)$ time divide-and-conquer algorithm to compute the convex hull of a set of n points in the plane.

1.9 Suppose that we have a subroutine CONVEXHULL available for computing the convex hull of a set of points in the plane. Its output is a list of convex hull vertices, sorted in clockwise order. Now let $S = \{x_1, x_2, \ldots, x_n\}$ be a set of n numbers. Show that S can be sorted in $O(n)$ time plus the time needed for one call to CONVEXHULL. Since the sorting problem has an $\Omega(n \log n)$ lower bound, this implies that the convex hull problem

has an $\Omega(n \log n)$ lower bound as well. Hence, the algorithm presented in this chapter is asymptotically optimal.

1.10 Let S be a set of n (possibly intersecting) unit circles in the plane. We want to compute the convex hull of S.

   a. Show that the boundary of the convex hull of S consists of straight line segments and pieces of circles in S.

   b. Show that each circle can occur at most once on the boundary of the convex hull.

   c. Let S' be the set of points that are the centers of the circles in S. Show that a circle in S appears on the boundary of the convex hull if and only if the center of the circle lies on the convex hull of S'.

   d. Give an $O(n \log n)$ algorithm for computing the convex hull of S.

   e.* Give an $O(n \log n)$ algorithm for the case in which the circles in S have different radii.