

Mark de Berg
Otfried Cheong
Marc van Kreveld
Mark Overmars

Computational Geometry

Algorithms and Applications
Third Edition



Prof. Dr. Mark de Berg
Department of Mathematics
and Computer Science
TU Eindhoven
P.O. Box 513
5600 MB Eindhoven
The Netherlands
mdberg@win.tue.nl

Dr. Otfried Cheong, né Schwarzkopf
Department of Computer Science
KAIST
Gwahangno 335, Yuseong-gu
Daejeon 305-701
Korea
otfried@kaist.edu

Dr. Marc van Kreveld
Department of Information
and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands
marc@cs.uu.nl

Prof. Dr. Mark Overmars
Department of Information
and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands
markov@cs.uu.nl

ISBN 978-3-540-77973-5

e-ISBN 978-3-540-77974-2

DOI 10.1007/978-3-540-77974-2

ACM Computing Classification (1998): F.2.2, I.3.5

Library of Congress Control Number: 2008921564

© 2008, 2000, 1997 Springer-Verlag Berlin Heidelberg

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable for prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover design: KünkelLopka, Heidelberg

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

Contents

1 Computational Geometry	1
Introduction	
1.1 An Example: Convex Hulls	2
1.2 Degeneracies and Robustness	8
1.3 Application Domains	10
1.4 Notes and Comments	13
1.5 Exercises	15
2 Line Segment Intersection	19
Thematic Map Overlay	
2.1 Line Segment Intersection	20
2.2 The Doubly-Connected Edge List	29
2.3 Computing the Overlay of Two Subdivisions	33
2.4 Boolean Operations	39
2.5 Notes and Comments	40
2.6 Exercises	41
3 Polygon Triangulation	45
Guarding an Art Gallery	
3.1 Guarding and Triangulations	46
3.2 Partitioning a Polygon into Monotone Pieces	49
3.3 Triangulating a Monotone Polygon	55
3.4 Notes and Comments	59
3.5 Exercises	60
4 Linear Programming	63
Manufacturing with Molds	
4.1 The Geometry of Casting	64
4.2 Half-Plane Intersection	66
4.3 Incremental Linear Programming	71
4.4 Randomized Linear Programming	76

CONTENTS		
4.5	Unbounded Linear Programs	79
4.6*	Linear Programming in Higher Dimensions	82
4.7*	Smallest Enclosing Discs	86
4.8	Notes and Comments	89
4.9	Exercises	91
5	Orthogonal Range Searching	95
	Querying a Database	
5.1	1-Dimensional Range Searching	96
5.2	Kd-Trees	99
5.3	Range Trees	105
5.4	Higher-Dimensional Range Trees	109
5.5	General Sets of Points	110
5.6*	Fractional Cascading	111
5.7	Notes and Comments	115
5.8	Exercises	117
6	Point Location	121
	Knowing Where You Are	
6.1	Point Location and Trapezoidal Maps	122
6.2	A Randomized Incremental Algorithm	128
6.3	Dealing with Degenerate Cases	137
6.4*	A Tail Estimate	140
6.5	Notes and Comments	143
6.6	Exercises	144
7	Voronoi Diagrams	147
	The Post Office Problem	
7.1	Definition and Basic Properties	148
7.2	Computing the Voronoi Diagram	151
7.3	Voronoi Diagrams of Line Segments	160
7.4	Farthest-Point Voronoi Diagrams	163
7.5	Notes and Comments	167
7.6	Exercises	170
8	Arrangements and Duality	173
	Supersampling in Ray Tracing	
8.1	Computing the Discrepancy	175
8.2	Duality	177
8.3	Arrangements of Lines	179
8.4	Levels and Discrepancy	185

8.5 Notes and Comments	186
8.6 Exercises	188
9 Delaunay Triangulations	191
Height Interpolation	
9.1 Triangulations of Planar Point Sets	193
9.2 The Delaunay Triangulation	196
9.3 Computing the Delaunay Triangulation	199
9.4 The Analysis	205
9.5* A Framework for Randomized Algorithms	208
9.6 Notes and Comments	214
9.7 Exercises	215
10 More Geometric Data Structures	219
Windowing	
10.1 Interval Trees	220
10.2 Priority Search Trees	226
10.3 Segment Trees	231
10.4 Notes and Comments	237
10.5 Exercises	239
11 Convex Hulls	243
Mixing Things	
11.1 The Complexity of Convex Hulls in 3-Space	244
11.2 Computing Convex Hulls in 3-Space	246
11.3* The Analysis	250
11.4* Convex Hulls and Half-Space Intersection	253
11.5* Voronoi Diagrams Revisited	254
11.6 Notes and Comments	256
11.7 Exercises	257
12 Binary Space Partitions	259
The Painter's Algorithm	
12.1 The Definition of BSP Trees	261
12.2 BSP Trees and the Painter's Algorithm	263
12.3 Constructing a BSP Tree	264
12.4* The Size of BSP Trees in 3-Space	268
12.5 BSP Trees for Low-Density Scenes	271
12.6 Notes and Comments	278
12.7 Exercises	279

13 Robot Motion Planning	283
Getting Where You Want to Be	
13.1 Work Space and Configuration Space	284
13.2 A Point Robot	286
13.3 Minkowski Sums	290
13.4 Translational Motion Planning	297
13.5* Motion Planning with Rotations	299
13.6 Notes and Comments	303
13.7 Exercises	305
14 Quadtrees	307
Non-Uniform Mesh Generation	
14.1 Uniform and Non-Uniform Meshes	308
14.2 Quadtrees for Point Sets	309
14.3 From Quadtrees to Meshes	315
14.4 Notes and Comments	318
14.5 Exercises	320
15 Visibility Graphs	323
Finding the Shortest Route	
15.1 Shortest Paths for a Point Robot	324
15.2 Computing the Visibility Graph	326
15.3 Shortest Paths for a Translating Polygonal Robot	330
15.4 Notes and Comments	331
15.5 Exercises	332
16 Simplex Range Searching	335
Windowing Revisited	
16.1 Partition Trees	336
16.2 Multi-Level Partition Trees	343
16.3 Cutting Trees	346
16.4 Notes and Comments	352
16.5 Exercises	353
Bibliography	357
Index	377

1 Computational Geometry

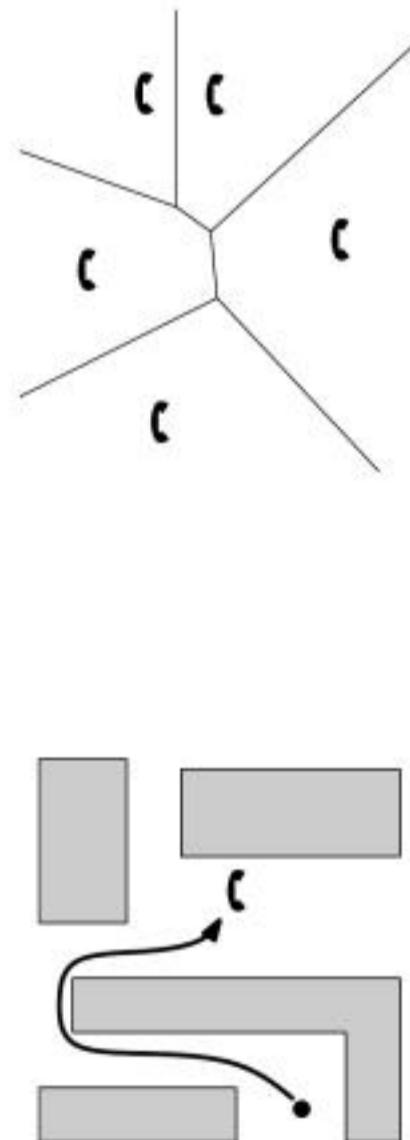
Introduction

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.

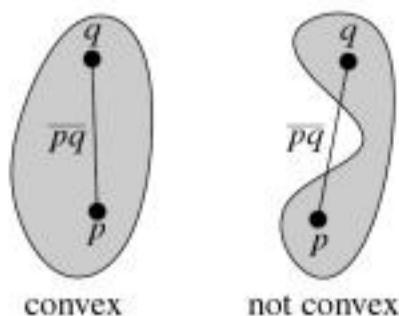
This book describes the most important notions, techniques, algorithms, and data structures from computational geometry in a way that we hope will be attractive to readers who are interested in applying results from computational geometry. Each chapter is motivated with a real computational problem that 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.



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 .



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.

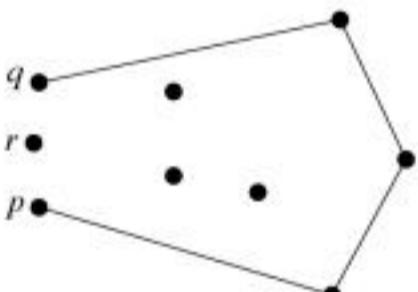
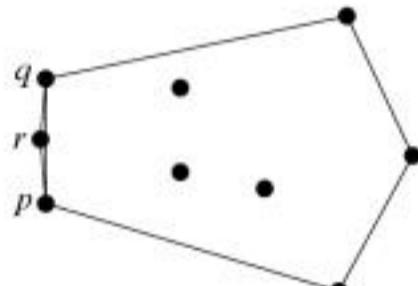
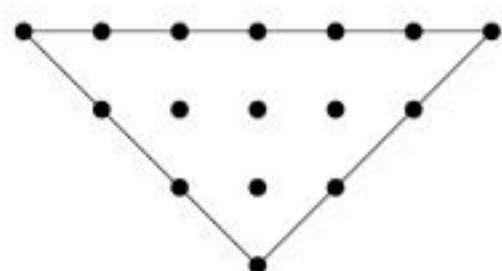
Imagine that there are three points p , q , and r , that are nearly collinear, and that all other points lie far to the right of them. Our algorithm tests the pairs (p, q) , (r, q) , and (p, r) . Since these points are nearly collinear, it is possible that the rounding errors lead us to decide that r 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.

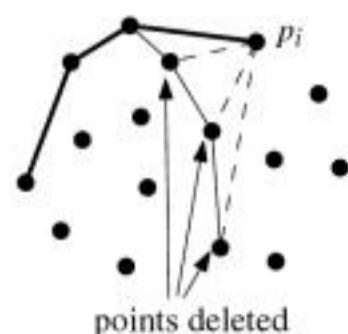
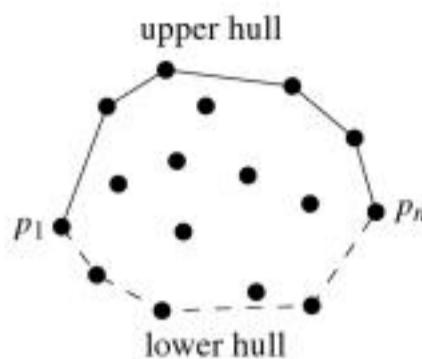
Although we have proven the algorithm to be correct and to handle all special cases, it is not *robust*: small errors in the computations can make it fail in completely unexpected ways. The problem is that we have proven the correctness assuming that we can compute exactly with real numbers.

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.

Section 1.1

AN EXAMPLE: CONVEX HULLS





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, \dots, 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, \dots, p_{i-1} , we have to compute the upper hull of p_1, \dots, 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, \dots, 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, \dots, 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.

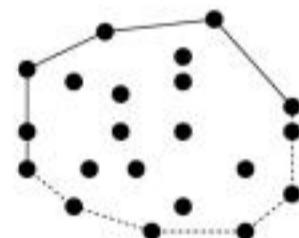
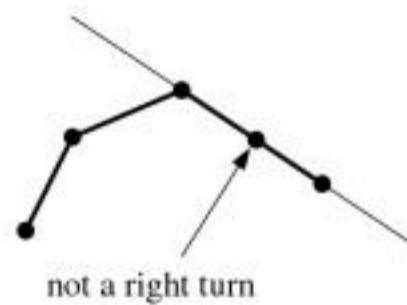
Another special case we have ignored is that the three points for which we have to determine whether they make a left or a right turn lie on a straight line. In this case the middle point should not occur on the convex hull, so collinear points must be treated as if they make a left turn. In other words, 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.

Section 1.1

AN EXAMPLE: CONVEX HULLS



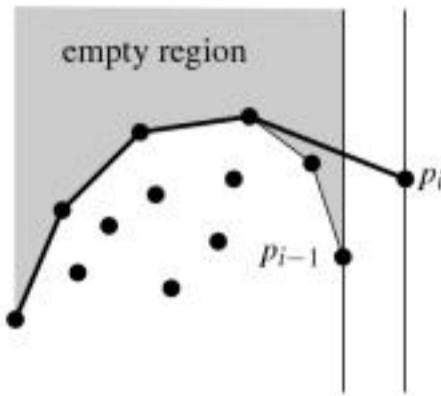
We conclude with the following theorem:

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 $\mathcal{L}_{\text{upper}}$ contains the points p_1 and p_2 , which trivially form the upper hull of $\{p_1, p_2\}$. Now suppose that $\mathcal{L}_{\text{upper}}$ contains the upper hull vertices of $\{p_1, \dots, p_{i-1}\}$ 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 $\mathcal{L}_{\text{upper}}$ form a chain that only makes right turns. Moreover, the chain starts at the lexicographically smallest point of $\{p_1, \dots, p_i\}$ and ends at the lexicographically largest point, namely p_i . If we can show that all points of $\{p_1, \dots, p_i\}$ that are not in $\mathcal{L}_{\text{upper}}$ are below the chain, then $\mathcal{L}_{\text{upper}}$ 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 $O(n \log n)$. \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.

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

Section 1.2

DEGENERACIES AND ROBUSTNESS



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.

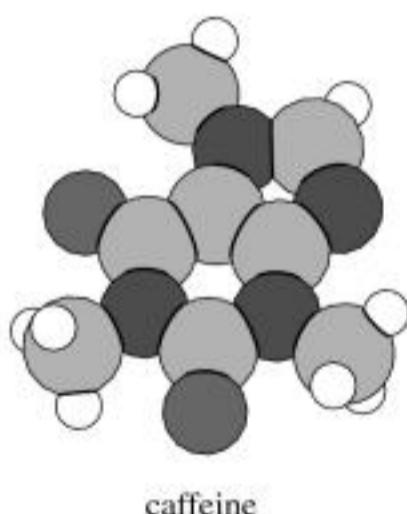
A recent trend is *design for assembly*, where assembly decisions are already taken into account during the design stage. A CAD system supporting this would allow designers to test their design for feasibility, answering questions like: can the product be built easily using a certain manufacturing process?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



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.

Section 1.4
NOTES AND COMMENTS

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.

Section 1.5

EXERCISES

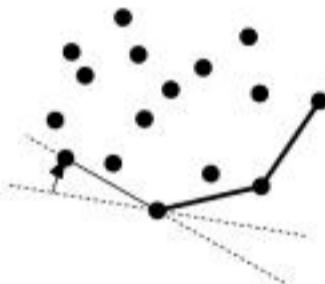
1.5 Exercises

- 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.
- 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.
 - * 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.
- Give pseudocode for this algorithm.
 - What degenerate cases can occur and how can we deal with them?
 - Prove that the algorithm correctly computes the convex hull.
 - Prove that the algorithm can be implemented to run in time $O(n \cdot h)$, where h is the complexity of the convex hull.
 - 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.
- 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 $\mathcal{P}_1 \cup \mathcal{P}_2$.
 - 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 $S = \{x_1, x_2, \dots, x_n\}$ 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





You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



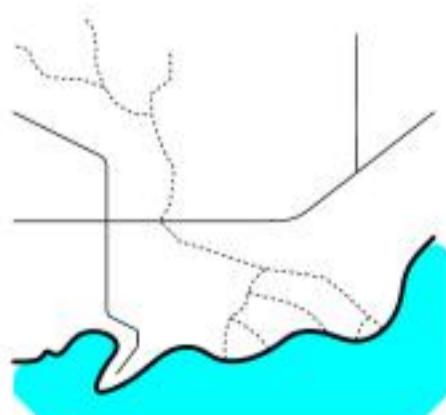
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



and so on. The theme of a layer can also be more abstract. For instance, there could be a layer for the population density, for average precipitation, habitat of the grizzly bear, or for vegetation. The type of geometric information stored in a layer can be very different: the layer for a road map could store the roads as collections of line segments (or curves, perhaps), the layer for cities could contain points labeled with city names, and the layer for vegetation could store a subdivision of the map into regions labeled with the type of vegetation.

Users of a geographic information system can select one of the thematic maps for display. To find a small town you would select the layer storing cities, and you would not be distracted by information such as the names of rivers and lakes. After you have spotted the town, you probably want to know how to get there. To this end geographic information systems allow users to view an *overlay* of several maps—see Figure 2.1. Using an overlay of the road map and the map storing cities you can now figure out how to get to the town. When two or more thematic map layers are shown together, intersections in the overlay are positions of special interest. For example, when viewing the overlay of the layer for the roads and the layer for the rivers, it would be useful if the intersections were clearly marked. In this example the two maps are basically networks, and the intersections are points. In other cases one is interested in the intersection of complete regions. For instance, geographers studying the climate could be interested in finding regions where there is pine forest and the annual precipitation is between 1000 mm and 1500 mm. These regions are the intersections of the regions labeled “pine forest” in the vegetation map and the regions labeled “1000–1500” in the precipitation map.

2.1 Line Segment Intersection



We first study the simplest form of the map overlay problem, where the two map layers are networks represented as collections of line segments. For example, a map layer storing roads, railroads, or rivers at a small scale. Note that curves can be approximated by a number of small segments. At first we won't be interested in the regions induced by these line segments. Later we shall look at the more complex situation where the maps are not just networks, but subdivisions of the plane into regions that have an explicit meaning. To solve the network overlay problem we first have to state it in a geometric setting. For the overlay of two networks the geometric situation is the following: given two sets of line segments, compute all intersections between a segment from one set and a segment from the other. This problem specification is not quite precise enough yet, as we didn't define when two segments intersect. In particular, do two segments intersect when an endpoint of one of them lies on the other? In other words, we have to specify whether the input segments are open or closed. To make this decision we should go back to the application, the network overlay problem. Roads in a road map and rivers in a river map are represented by chains of segments, so a crossing of a road and a river corresponds to the interior of one chain intersecting the interior of another chain. This does not mean that

there is an intersection between the interior of two segments: the intersection point could happen to coincide with an endpoint of a segment of a chain. In fact, this situation is not uncommon because windy rivers are represented by many small segments and coordinates of endpoints may have been rounded when maps are digitized. We conclude that we should define the segments to be closed, so that an endpoint of one segment lying on another segment counts as an intersection.

To simplify the description somewhat we shall put the segments from the two sets into one set, and compute all intersections among the segments in that set. This way we certainly find all the intersections we want. We may also find intersections between segments from the same set. Actually, we certainly will, because in our application the segments from one set form a number of chains, and we count coinciding endpoints as intersections. These other intersections can be filtered out afterwards by simply checking for each reported intersection whether the two segments involved belong to the same set. So our problem specification is as follows: given a set S of n closed segments in the plane, report all intersection points among the segments in S .

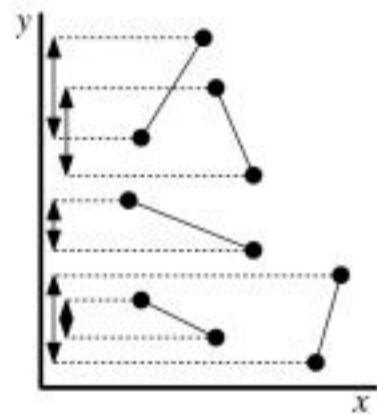
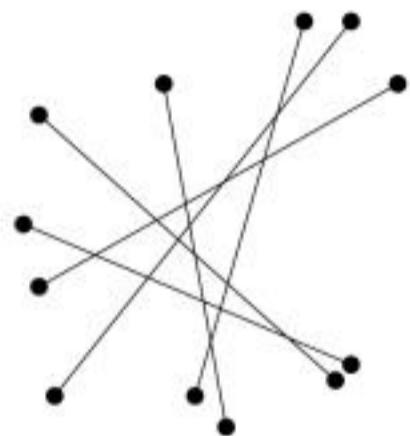
This doesn't seem like a challenging problem: we can simply take each pair of segments, compute whether they intersect, and, if so, report their intersection point. This brute-force algorithm clearly requires $O(n^2)$ time. In a sense this is optimal: when each pair of segments intersects any algorithm must take $\Omega(n^2)$ time, because it has to report all intersections. A similar example can be given when the overlay of two networks is considered. In practical situations, however, most segments intersect no or only a few other segments, so the total number of intersection points is much smaller than quadratic. It would be nice to have an algorithm that is faster in such situations. In other words, we want an algorithm whose running time depends not only on the number of segments in the input, but also on the number of intersection points. Such an algorithm is called an *output-sensitive algorithm*: the running time of the algorithm is sensitive to the size of the output. We could also call such an algorithm *intersection-sensitive*, since the number of intersections is what determines the size of the output.

How can we avoid testing all pairs of segments for intersection? Here we must make use of the geometry of the situation: segments that are close together are candidates for intersection, unlike segments that are far apart. Below we shall see how we can use this observation to obtain an output-sensitive algorithm for the line segment intersection problem.

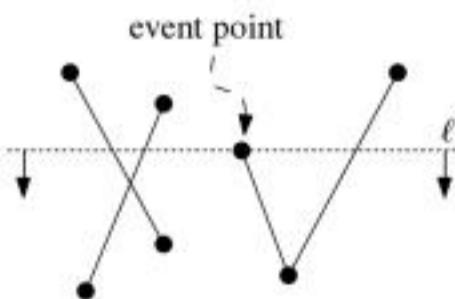
Let $S := \{s_1, s_2, \dots, s_n\}$ be the set of segments for which we want to compute all intersections. We want to avoid testing pairs of segments that are far apart. But how can we do this? Let's first try to rule out an easy case. Define the y -interval of a segment to be its orthogonal projection onto the y -axis. When the y -intervals of a pair of segments do not overlap—we could say that they are far apart in the y -direction—then they cannot intersect. Hence, we only need to test pairs of segments whose y -intervals overlap, that is, pairs for which there exists a horizontal line that intersects both segments. To find these pairs we imagine sweeping a line ℓ downwards over the plane, starting from a position above all

Section 2.1

LINE SEGMENT INTERSECTION

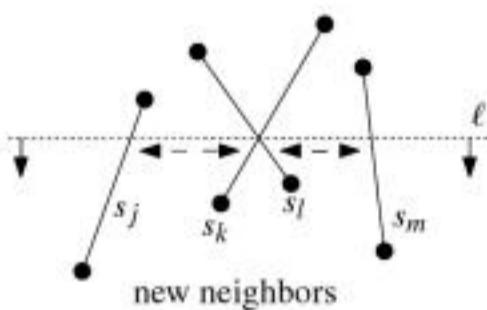


segments. While we sweep the imaginary line, we keep track of all segments intersecting it—the details of this will be explained later—so that we can find the pairs we need.



This type of algorithm is called a *plane sweep algorithm* and the line ℓ is called the *sweep line*. The *status* of the sweep line is the set of segments intersecting it. The status changes while the sweep line moves downwards, but not continuously. Only at particular points is an update of the status required. We call these points the *event points* of the plane sweep algorithm. In this algorithm the event points are the endpoints of the segments.

The moments at which the sweep line reaches an event point are the only moments when the algorithm actually does something: it updates the status of the sweep line and performs some intersection tests. In particular, if the event point is the upper endpoint of a segment, then a new segment starts intersecting the sweep line and must be added to the status. This segment is tested for intersection against the ones already intersecting the sweep line. If the event point is a lower endpoint, a segment stops intersecting the sweep line and must be deleted from the status. This way we only test pairs of segments for which there is a horizontal line that intersects both segments. Unfortunately, this is not enough: there are still situations where we test a quadratic number of pairs, whereas there is only a small number of intersection points. A simple example is a set of vertical segments that all intersect the x -axis. So the algorithm is not output-sensitive. The problem is that two segments that intersect the sweep line can still be far apart in the horizontal direction.



Let's order the segments from left to right as they intersect the sweep line, to include the idea of being close in the horizontal direction. We shall only test segments when they are adjacent in the horizontal ordering. This means that we only test any new segment against two segments, namely, the ones immediately left and right of the upper endpoint. Later, when the sweep line has moved downwards to another position, a segment can become adjacent to other segments against which it will be tested. Our new strategy should be reflected in the status of our algorithm: the status now corresponds to the *ordered* sequence of segments intersecting the sweep line. The new status not only changes at endpoints of segments; it also changes at intersection points, where the order of the intersected segments changes. When this happens we must test the two segments that change position against their new neighbors. This is a new type of event point.

Before trying to turn these ideas into an efficient algorithm, we should convince ourselves that the approach is correct. We have reduced the number of pairs to be tested, but do we still find all intersections? In other words, if two segments s_i and s_j intersect, is there always a position of the sweep line ℓ where s_i and s_j are adjacent along ℓ ? Let's first ignore some nasty cases: assume that no segment is horizontal, that any two segments intersect in at most one point—they do not overlap—, and that no three segments meet in a common point. Later we shall see that these cases are easy to handle, but for now it is convenient to forget about them. The intersections where an endpoint of a segment lies on another segment can easily be detected when the sweep line

reaches the endpoint. So the only question is whether intersections between the interiors of segments are always detected.

Lemma 2.1 Let s_i and s_j be two non-horizontal segments whose interiors intersect in a single point p , and assume there is no third segment passing through p . Then there is an event point above p where s_i and s_j become adjacent and are tested for intersection.

Proof. Let ℓ be a horizontal line slightly above p . If ℓ is close enough to p then s_i and s_j must be adjacent along ℓ . (To be precise, we should take ℓ such that there is no event point on ℓ , nor in between ℓ and the horizontal line through p .) In other words, there is a position of the sweep line where s_i and s_j are adjacent. On the other hand, s_i and s_j are not yet adjacent when the algorithm starts, because the sweep line starts above all line segments and the status is empty. Hence, there must be an event point q where s_i and s_j become adjacent and are tested for intersection. \square

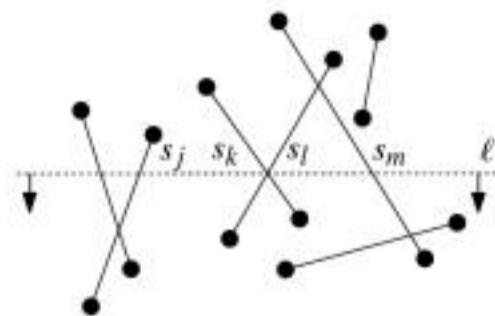
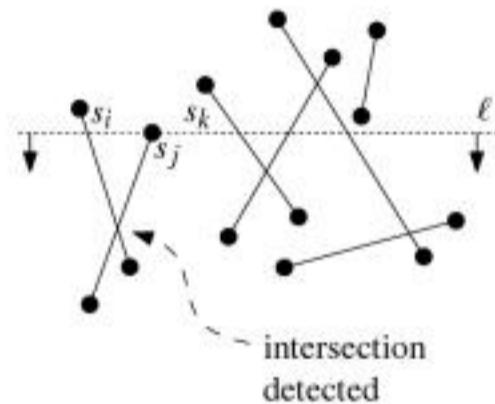
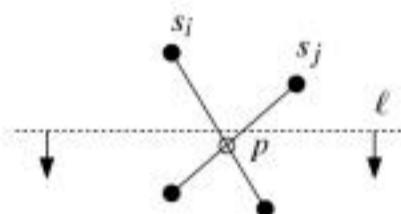
So our approach is correct, at least when we forget about the nasty cases mentioned earlier. Now we can proceed with the development of the plane sweep algorithm. Let's briefly recap the overall approach. We imagine moving a horizontal sweep line ℓ downwards over the plane. The sweep line halts at certain event points; in our case these are the endpoints of the segments, which we know beforehand, and the intersection points, which are computed on the fly. While the sweep line moves we maintain the ordered sequence of segments intersected by it. When the sweep line halts at an event point the sequence of segments changes and, depending on the type of event point, we have to take several actions to update the status and detect intersections.

When the event point is the upper endpoint of a segment, there is a new segment intersecting the sweep line. This segment must be tested for intersection against its two neighbors along the sweep line. Only intersections below the sweep line are important; the ones above the sweep line have been detected already. For example, if segments s_i and s_k are adjacent on the sweep line, and a new upper endpoint of a segment s_j appears in between, then we have to test s_j for intersection with s_i and s_k . If we find an intersection below the sweep line, we have found a new event point. After the upper endpoint is handled we continue to the next event point.

When the event point is an intersection, the two segments that intersect change their order. Each of them gets (at most) one new neighbor against which it is tested for intersection. Again, only intersections below the sweep line are still interesting. Suppose that four segments s_j , s_k , s_l , and s_m appear in this order on the sweep line when the intersection point of s_k and s_l is reached. Then s_k and s_l switch position and we must test s_l and s_j for intersection below the sweep line, and also s_k and s_m . The new intersections that we find are, of course, also event points for the algorithm. Note, however, that it is possible that these events have already been detected earlier, namely if a pair becoming adjacent has been adjacent before.

Section 2.1

LINE SEGMENT INTERSECTION





When the event point is the lower endpoint of a segment, its two neighbors now become adjacent and must be tested for intersection. If they intersect below the sweep line, then their intersection point is an event point. (Again, this event could have been detected already.) Assume three segments s_k , s_l , and s_m appear in this order on the sweep line when the lower endpoint of s_l is encountered. Then s_k and s_m will become adjacent and we test them for intersection.

After we have swept the whole plane—more precisely, after we have treated the last event point—we have computed all intersection points. This is guaranteed by the following invariant, which holds at any time during the plane sweep: all intersection points above the sweep line have been computed correctly.

After this sketch of the algorithm, it's time to go into more detail. It's also time to look at the degenerate cases that can arise, like three or more segments meeting in a point. We should first specify what we expect from the algorithm in these cases. We could require the algorithm to simply report each intersection point once, but it seems more useful if it reports for each intersection point a list of segments that pass through it or have it as an endpoint. There is another special case for which we should define the required output more carefully, namely that of two partially overlapping segments, but for simplicity we shall ignore this case in the rest of this section.

We start by describing the data structures the algorithm uses.

First of all we need a data structure—called the *event queue*—that stores the events. We denote the event queue by Ω . We need an operation that removes the next event that will occur from Ω , and returns it so that it can be treated. This event is the highest event below the sweep line. If two event points have the same y -coordinate, then the one with smaller x -coordinate will be returned. In other words, event points on the same horizontal line are treated from left to right. This implies that we should consider the left endpoint of a horizontal segment to be its upper endpoint, and its right endpoint to be its lower endpoint. You can also think about our convention as follows: instead of having a horizontal sweep line, imagine it is sloping just a tiny bit upward. As a result the sweep line reaches the left endpoint of a horizontal segment just before reaching the right endpoint. The event queue must allow insertions, because new events will be computed on the fly. Notice that two event points can coincide. For example, the upper endpoints of two distinct segments may coincide. It is convenient to treat this as one event point. Hence, an insertion must be able to check whether an event is already present in Ω .

We implement the event queue as follows. Define an order \prec on the event points that represents the order in which they will be handled. Hence, if p and q are two event points then we have $p \prec q$ if and only if $p_y > q_y$ holds or $p_y = q_y$ and $p_x < q_x$ holds. We store the event points in a balanced binary search tree, ordered according to \prec . With each event point p in Ω we will store the segments starting at p , that is, the segments whose upper endpoint is p . This information will be needed to handle the event. Both operations—fetching the next event and inserting an event—take $O(\log m)$ time, where m is the number of events



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The procedures for finding the new intersections are easy: they simply test two segments for intersection. The only thing we need to be careful about is, when we find an intersection, whether this intersection has already been handled earlier or not. When there are no horizontal segments, then the intersection has not been handled yet when the intersection point lies below the sweep line. But how should we deal with horizontal segments? Recall our convention that events with the same y -coordinate are treated from left to right. This implies that we are still interested in intersection points lying to the right of the current event point. Hence, the procedure FINDNEWEVENT is defined as follows.

FINDNEWEVENT(s_l, s_r, p)

1. **if** s_l and s_r intersect below the sweep line, or on it and to the right of the current event point p , and the intersection is not yet present as an event in Ω
2. **then** Insert the intersection point as an event into Ω .

What about the correctness of our algorithm? It is clear that FINDINTERSECTIONS only reports true intersection points, but does it find all of them? The next lemma states that this is indeed the case.

Lemma 2.2 *Algorithm FINDINTERSECTIONS computes all intersection points and the segments that contain it correctly.*

Proof. Recall that the priority of an event is given by its y -coordinate, and that when two events have the same y -coordinate the one with smaller x -coordinate is given higher priority. We shall prove the lemma by induction on the priority of the event points.

Let p be an intersection point and assume that all intersection points q with a higher priority have been computed correctly. We shall prove that p and the segments that contain p are computed correctly. Let $U(p)$ be the set of segments that have p as their upper endpoint (or, for horizontal segments, their left endpoint), let $L(p)$ be the set of segments having p as their lower endpoint (or, for horizontal segments, their right endpoint), and let $C(p)$ be the set of segments having p in their interior.

First, assume that p is an endpoint of one or more of the segments. In that case p is stored in the event queue Ω at the start of the algorithm. The segments from $U(p)$ are stored with p , so they will be found. The segments from $L(p)$ and $C(p)$ are stored in \mathcal{T} when p is handled, so they will be found in line 2 of HANDLEEVENTPOINT. Hence, p and all the segments involved are determined correctly when p is an endpoint of one or more of the segments.

Now assume that p is not an endpoint of a segment. All we need to show is that p will be inserted into Ω at some moment. Note that all segments that are involved have p in their interior. Order these segments by angle around p , and let s_i and s_j be two neighboring segments. Following the proof of Lemma 2.1 we see that there is an event point with a higher priority than p such that s_i and s_j become adjacent when p is passed. In Lemma 2.1 we assumed for simplicity that s_i and s_j are non-horizontal, but it is straightforward to adapt the proof for

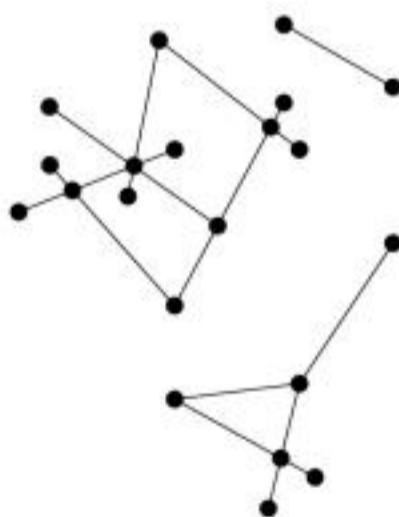
horizontal segments. By induction, the event point q was handled correctly, which means that p is detected and stored into Ω . \square

So we have a correct algorithm. But did we succeed in developing an output-sensitive algorithm? The answer is yes: the running time of the algorithm is $O((n+k)\log n)$, where k is the size of the output. The following lemma states an even stronger result: the running time is $O((n+I)\log n)$, where I is the number of intersections. This is stronger, because for one intersection point the output can consist of a large number of segments, namely in the case where many segments intersect in a common point.

Lemma 2.3 *The running time of Algorithm FINDINTERSECTIONS for a set S of n line segments in the plane is $O(n \log n + I \log n)$, where I is the number of intersection points of segments in S .*

Proof. The algorithm starts by constructing the event queue on the segment endpoints. Because we implemented the event queue as a balanced binary search tree, this takes $O(n \log n)$ time. Initializing the status structure takes constant time. Then the plane sweep starts and all the events are handled. To handle an event we perform three operations on the event queue Ω : the event itself is deleted from Ω in line 4 of FINDINTERSECTIONS, and there can be one or two calls to FINDNEWEVENT, which may cause at most two new events to be inserted into Ω . Deletions and insertions on Ω take $O(\log n)$ time each. We also perform operations—insertions, deletions, and neighbor finding—on the status structure T , which take $O(\log n)$ time each. The number of operations is linear in the number $m(p) := \text{card}(L(p) \cup U(p) \cup C(p))$ of segments that are involved in the event. If we denote the sum of all $m(p)$, over all event points p , by m , the running time of the algorithm is $O(m \log n)$.

It is clear that $m = O(n+k)$, where k is the size of the output; after all, whenever $m(p) > 1$ we report all segments involved in the event, and the only events involving one segment are the endpoints of segments. But we want to prove that $m = O(n+I)$, where I is the number of intersection points. To show this, we will interpret the set of segments as a planar graph embedded in the plane. (If you are not familiar with planar graph terminology, you should read the first paragraphs of Section 2.2 first.) Its vertices are the endpoints of segments and intersection points of segments, and its edges are the pieces of the segments connecting vertices. Consider an event point p . It is a vertex of the graph, and $m(p)$ is bounded by the degree of the vertex. Consequently, m is bounded by the sum of the degrees of all vertices of our graph. Every edge of the graph contributes one to the degree of exactly two vertices (its endpoints), so m is bounded by $2n_e$, where n_e is the number of edges of the graph. Let's bound n_e in terms of n and I . By definition, n_v , the number of vertices, is at most $2n+I$. It is well known that in planar graphs $n_e = O(n_v)$, which proves our claim. But, for completeness, let us give the argument here. Every face of the planar graph is bounded by at least three edges—provided that there are at least three segments—and an edge can bound at most two different faces. Therefore n_f , the number of faces, is at most $2n_e/3$. We now use *Euler's formula*, which states that for any planar graph with n_v vertices, n_e edges, and n_f faces, the



following relation holds:

$$n_v - n_e + n_f \geq 2.$$

Equality holds if and only if the graph is connected. Plugging the bounds on n_v and n_f into this formula, we get

$$2 \leq (2n+I) - n_e + \frac{2n_e}{3} = (2n+I) - n_e/3.$$

So $n_e \leq 6n + 3I - 6$, and $m \leq 12n + 6I - 12$, and the bound on the running time follows. \square

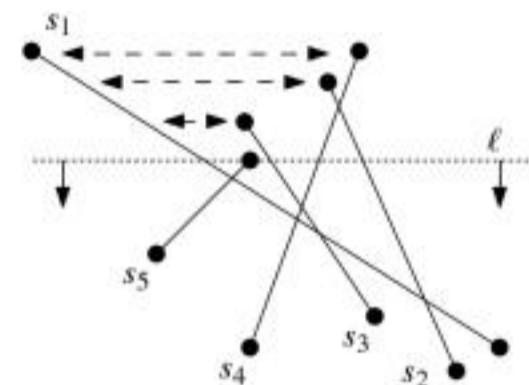
We still have to analyze the other complexity aspect, the amount of storage used by the algorithm. The tree T stores a segment at most once, so it uses $O(n)$ storage. The size of Ω can be larger, however. The algorithm inserts intersection points in Ω when they are detected and it removes them when they are handled. When it takes a long time before intersections are handled, it could happen that Ω gets very large. Of course its size is always bounded by $O(n+I)$, but it would be better if the working storage were always linear.

There is a relatively simple way to achieve this: only store intersection points of pairs of segments that are currently adjacent on the sweep line. The algorithm given above also stores intersection points of segments that have been horizontally adjacent, but aren't anymore. By storing only intersections among adjacent segments, the number of event points in Ω is never more than linear. The modification required in the algorithm is that the intersection point of two segments must be deleted when they stop being adjacent. These segments must become adjacent again before the intersection point is reached, so the intersection point will still be reported correctly. The total time taken by the algorithm remains $O(n \log n + I \log n)$. We obtain the following theorem:

Theorem 2.4 Let S be a set of n line segments in the plane. All intersection points in S , with for each intersection point the segments involved in it, can be reported in $O(n \log n + I \log n)$ time and $O(n)$ space, where I is the number of intersection points.

Section 2.2

THE DOUBLY-CONNECTED EDGE LIST

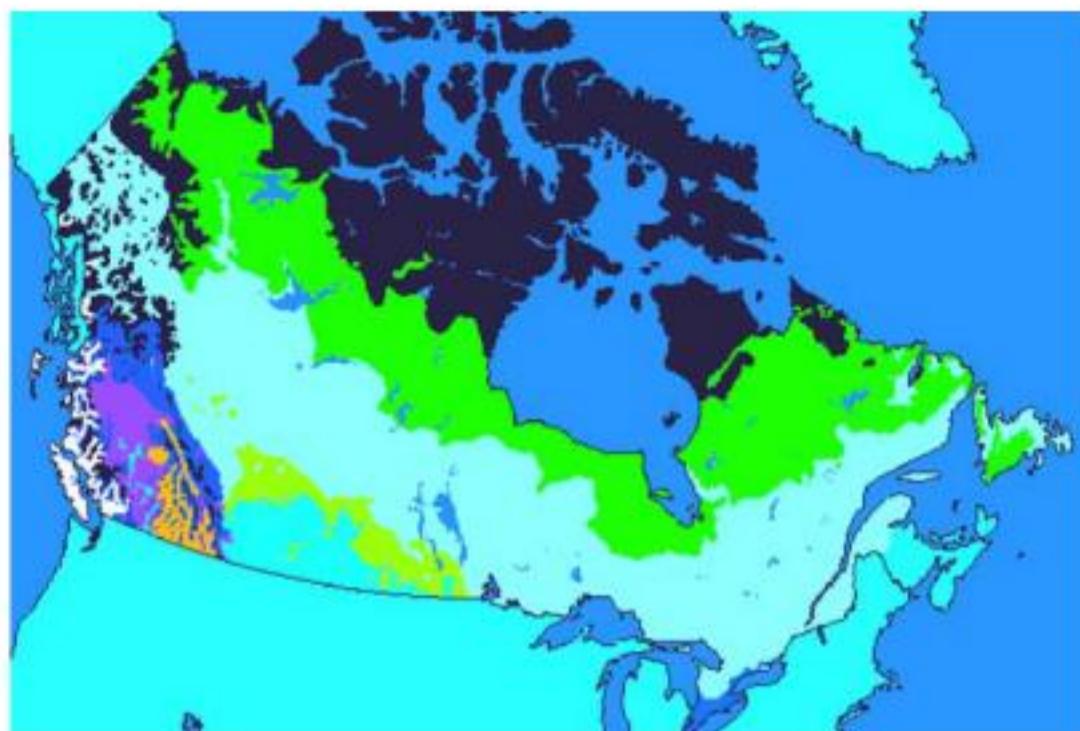


2.2 The Doubly-Connected Edge List

We have solved the easiest case of the map overlay problem, where the two maps are networks represented as collections of line segments. In general, maps have a more complicated structure: they are subdivisions of the plane into labeled regions. A thematic map of forests in Canada, for instance, would be a subdivision of Canada into regions with labels such as “pine”, “deciduous”, “birch”, and “mixed”.

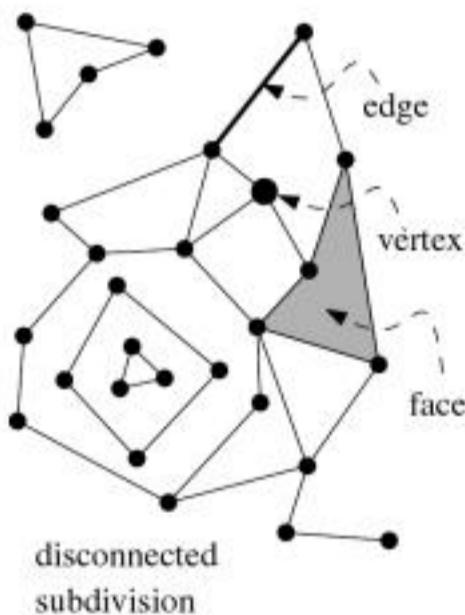
Before we can give an algorithm for computing the overlay of two subdivisions, we must develop a suitable representation for a subdivision. Storing a subdivision as a collection of line segments is not such a good idea. Operations like reporting the boundary of a region would be rather complicated. It is better

Figure 2.3
Types of forest in Canada



to incorporate structural, topological information: which segments bound a given region, which regions are adjacent, and so on.

The maps we consider are *planar subdivisions* induced by planar embeddings of graphs. Such a subdivision is *connected* if the underlying graph is connected. The embedding of a node of the graph is called a *vertex*, and the embedding of an arc is called an *edge*. We only consider embeddings where every edge is a straight line segment. In principle, edges in a subdivision need not be straight. A subdivision need not even be a planar embedding of a graph, as it may have unbounded edges. In this section, however, we don't consider such more general subdivisions. We consider an edge to be open, that is, its endpoints—which are vertices of the subdivision—are not part of it. A *face* of the subdivision is a maximal connected subset of the plane that doesn't contain a point on an edge or a vertex. Thus a face is an open polygonal region whose boundary is formed by edges and vertices from the subdivision. The *complexity* of a subdivision is defined as the sum of the number of vertices, the number of edges, and the number of faces it consists of. If a vertex is the endpoint of an edge, then we say that the vertex and the edge are *incident*. Similarly, a face and an edge on its boundary are incident, and a face and a vertex of its boundary are incident.



What should we require from a representation of a subdivision? An operation one could ask for is to determine the face containing a given point. This is definitely useful in some applications—indeed, in a later chapter we shall design a data structure for this—but it is a bit too much to ask from a basic representation. The things we can ask for should be more local. For example, it is reasonable to require that we can walk around the boundary of a given face, or that we can access one face from an adjacent one if we are given a common edge. Another operation that could be useful is to visit all the edges around a given vertex. The representation that we shall discuss supports these operations. It is called the *doubly-connected edge list*.

A *doubly-connected edge list* contains a record for each face, edge, and vertex

of the subdivision. Besides the geometric and topological information—to be described shortly—each record may also store additional information. For instance, if the subdivision represents a thematic map for vegetation, the doubly-connected edge list would store in each face record the type of vegetation of the corresponding region. The additional information is also called *attribute information*. The geometric and topological information stored in the doubly-connected edge list should enable us to perform the basic operations mentioned earlier. To be able to walk around a face in counterclockwise order we store a pointer from each edge to the next. It can also come in handy to walk around a face the other way, so we also store a pointer to the previous edge. An edge usually bounds two faces, so we need two pairs of pointers for it. It is convenient to view the different sides of an edge as two distinct *half-edges*, so that we have a unique next half-edge and previous half-edge for every half-edge. This also means that a half-edge bounds only one face. The two half-edges we get for a given edge are called *twins*. Defining the next half-edge of a given half-edge with respect to a counterclockwise traversal of a face induces an orientation on each half-edge: it is oriented such that the face that it bounds lies to its left for an observer walking along the edge. Because half-edges are oriented we can speak of the *origin* and the *destination* of a half-edge. If a half-edge \vec{e} has v as its origin and w as its destination, then its twin $\text{Twin}(\vec{e})$ has w as its origin and v as its destination. To reach the boundary of a face we just need to store one pointer in the face record to an arbitrary half-edge bounding the face. Starting from that half-edge, we can step from each half-edge to the next and walk around the face.

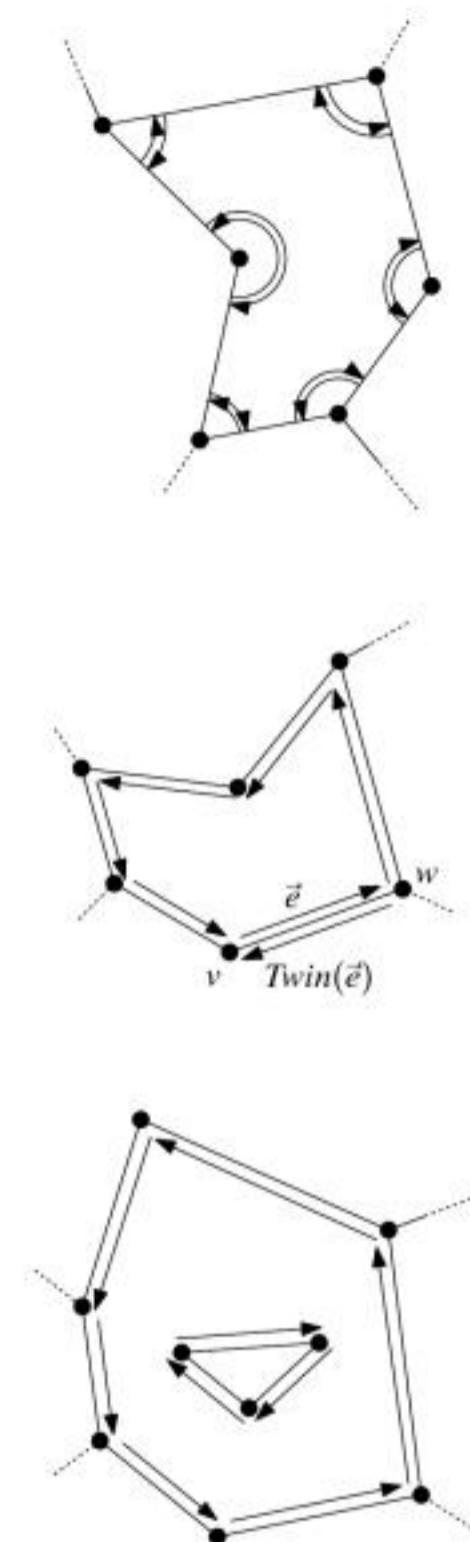
What we just said does not quite hold for the boundaries of holes in a face: if they are traversed in counterclockwise order then the face lies to the right. It will be convenient to orient half-edges such that their face always lies to the same side, so we change the direction of traversal for the boundary of a hole to clockwise. Now a face always lies to the left of any half-edge on its boundary. Another consequence is that twin half-edges always have opposite orientations. The presence of holes in a face also means that one pointer from the face to an arbitrary half-edge on its boundary is not enough to visit the whole boundary: we need a pointer to a half-edge in every boundary component. If a face has isolated vertices that don't have any incident edge, we can store pointers to them as well. For simplicity we'll ignore this case.

Let's summarize. The doubly-connected edge list consists of three collections of records: one for the vertices, one for the faces, and one for the half-edges. These records store the following geometric and topological information:

- The vertex record of a vertex v stores the coordinates of v in a field called *Coordinates*(v). It also stores a pointer *IncidentEdge*(v) to an arbitrary half-edge that has v as its origin.
- The face record of a face f stores a pointer *OuterComponent*(f) to some half-edge on its outer boundary. For the unbounded face this pointer is **nil**. It also stores a list *InnerComponents*(f), which contains for each hole in the face a pointer to some half-edge on the boundary of the hole.

Section 2.2

THE DOUBLY-CONNECTED EDGE LIST





You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

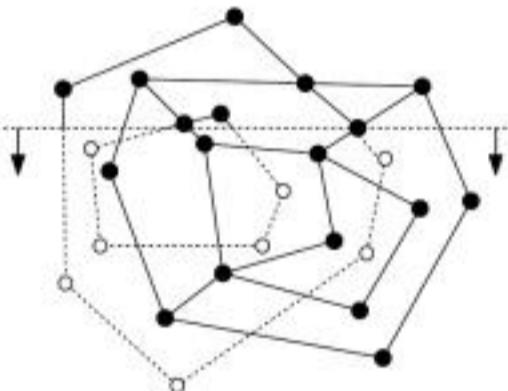
hold for the half-edge records in the doubly-connected edge list that correspond to the pieces? If the orientation of a half-edge would change, we would still have to change the information in these records. Fortunately, this is not the case. The half-edges are oriented such that the face that they bound lies to the left; the shape of the face may change in the overlay, but it will remain to the same side of the half-edge. Hence, we can re-use half-edge records corresponding to edges that are not intersected by edges from the other map. Stated differently, the only half-edge records in the doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1, \mathcal{S}_2)$ that we cannot borrow from \mathcal{S}_1 or \mathcal{S}_2 are the ones that are incident to an intersection between edges from different maps.

This suggests the following approach. First, copy the doubly-connected edge lists of \mathcal{S}_1 and \mathcal{S}_2 into one new doubly-connected edge list. The new doubly-connected edge list is not a valid doubly-connected edge list, of course, in the sense that it does not yet represent a planar subdivision. This is the task of the overlay algorithm: it must transform the doubly-connected edge list into a valid doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1, \mathcal{S}_2)$ by computing the intersections between the two networks of edges, and linking together the appropriate parts of the two doubly-connected edge lists.

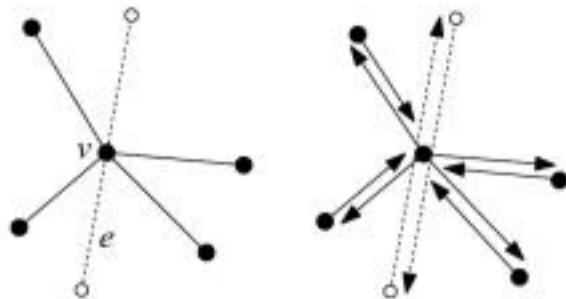
We did not talk about the new face records yet. The information for these records is more difficult to compute, so we leave this for later. We first describe in a little more detail how the vertex and half-edge records of the doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1, \mathcal{S}_2)$ are computed.

Our algorithm is based on the plane sweep algorithm of Section 2.1 for computing the intersections in a set of line segments. We run this algorithm on the set of segments that is the union of the sets of edges of the two subdivisions \mathcal{S}_1 and \mathcal{S}_2 . Here we consider the edges to be closed. Recall that the algorithm is supported by two data structures: an event queue \mathcal{Q} , which stores the event points, and the status structure \mathcal{T} , which is a balanced binary search tree storing the segments intersecting the sweep line, ordered from left to right. We now also maintain a doubly-connected edge list \mathcal{D} . Initially, \mathcal{D} contains a copy of the doubly-connected edge list for \mathcal{S}_1 and a copy of the doubly-connected edge list for \mathcal{S}_2 . During the plane sweep we shall transform \mathcal{D} to a correct doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1, \mathcal{S}_2)$. That is to say, as far as the vertex and half-edge records are concerned; the face information will be computed later. We keep cross pointers between the edges in the status structure \mathcal{T} and the half-edge records in \mathcal{D} that correspond to them. This way we can access the part of \mathcal{D} that needs to be changed when we encounter an intersection point. The invariant that we maintain is that at any time during the sweep, the part of the overlay above the sweep line has been computed correctly.

Now, let's consider what we must do when we reach an event point. First of all, we update \mathcal{T} and \mathcal{Q} as in the line segment intersection algorithm. If the event involves only edges from one of the two subdivisions, this is all; the event point is a vertex that can be re-used. If the event involves edges from both subdivisions, we must make local changes to \mathcal{D} to link the doubly-connected edge lists of the two original subdivisions at the intersection point. This is tedious but not difficult.



the geometric situation and the two doubly-connected edge lists before handling the intersection



the doubly-connected edge list after handling the intersection

Section 2.3

COMPUTING THE OVERLAY OF TWO SUBDIVISIONS

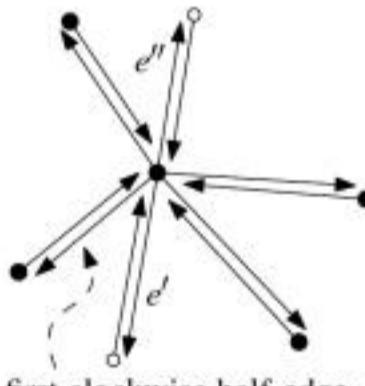
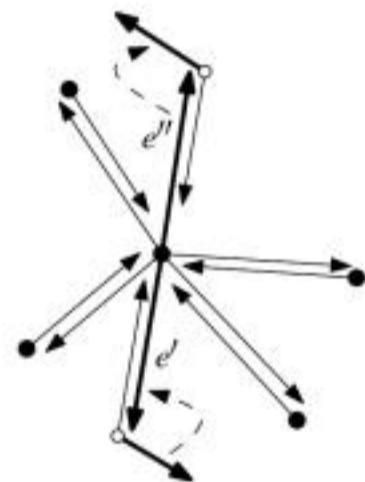
Figure 2.5

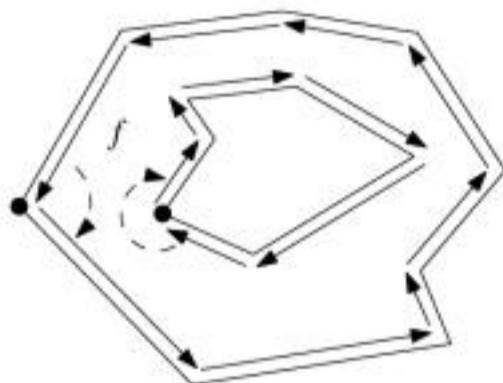
An edge of one subdivision passing through a vertex of the other

We describe the details for one of the possible cases, namely when an edge e of S_1 passes through a vertex v of S_2 , see Figure 2.5. The edge e must be replaced by two edges denoted e' and e'' . In the doubly-connected edge list, the two half-edges for e must become four. We create two new half-edge records, both with v as the origin. The two existing half-edges for e keep the endpoints of e as their origin, as shown in Figure 2.5. Then we pair up the existing half-edges with the new half-edges by setting their *Twin()* pointers. So e' is represented by one new and one existing half-edge, and the same holds for e'' . Now we must set a number of *Prev()* and *Next()* pointers. We first deal with the situation around the endpoints of e ; later we'll worry about the situation around v . The *Next()* pointers of the two new half-edges each copy the *Next()* pointer of the old half-edge that is not its twin. The half-edges to which these pointers point must also update their *Prev()* pointer and set it to the new half-edges. The correctness of this step can be verified best by looking at a figure.

It remains to correct the situation around vertex v . We must set the *Next()* and *Prev()* pointers of the four half-edges representing e' and e'' , and of the four half-edges incident from S_2 to v . We locate these four half-edges from S_2 by testing where e' and e'' should be in the cyclic order of the edges around vertex v . There are four pairs of half-edges that become linked by a *Next()* pointer from the one and a *Prev()* pointer from the other. Consider the half-edge for e' that has v as its destination. It must be linked to the first half-edge, seen clockwise from e' , with v as its origin. The half-edge for e' with v as its origin must be linked to the first counterclockwise half-edge with v as its destination. The same statements hold for e'' .

Most of the steps in the description above take only constant time. Only locating where e' and e'' appear in the cyclic order around v may take longer: it will take time linear in the degree of v . The other cases that can arise—crossings of two edges from different maps, and coinciding vertices—are not more difficult than the case we just discussed. These cases also take time $O(m)$, where m is the number of edges incident to the event point. This means that updating \mathcal{D} does not increase the running time of the line segment intersection algorithm asymptotically. Notice that every intersection that we find is a vertex of the overlay. It follows that the vertex records and the half-edge records of the doubly-connected edge list for $\mathcal{O}(S_1, S_2)$ can be computed in $O(n \log n + k \log n)$ time, where n denotes the sum of the complexities of S_1 and S_2 , and k is the complexity of the overlay.





After the fields involving vertex and half-edge records have been set, it remains to compute the information about the faces of $\mathcal{O}(\mathcal{S}_1, \mathcal{S}_2)$. More precisely, we have to create a face record for each face f in $\mathcal{O}(\mathcal{S}_1, \mathcal{S}_2)$, we have to make $\text{OuterComponent}(f)$ point to a half-edge on the outer boundary of f , and we have to make a list $\text{InnerComponents}(f)$ of pointers to half-edges on the boundaries of the holes inside f . Furthermore, we must set the $\text{IncidentFace}()$ fields of the half-edges on the boundary of f so that they point to the face record of f . Finally, each of the new faces must be labeled with the names of the faces in the old subdivisions that contain it.

How many face records will there be? Well, except for the unbounded face, every face has a unique outer boundary, so the number of face records we have to create is equal to the number of outer boundaries plus one. From the part of the doubly-connected edge list we have constructed so far we can easily extract all boundary cycles. But how do we know whether a cycle is an outer boundary or the boundary of a hole in a face? This can be decided by looking at the leftmost vertex v of the cycle, or, in case of ties, at the lowest of the leftmost ones. Recall that half-edges are directed in such a way that their incident face locally lies to the left. Consider the two half-edges of the cycle that are incident to v . Because we know that the incident face lies to the left, we can compute the angle these two half-edges make inside the incident face. If this angle is smaller than 180° then the cycle is an outer boundary, and otherwise it is the boundary of a hole. This property holds for the leftmost vertex of a cycle, but not necessarily for other vertices of that cycle.

To decide which boundary cycles bound the same face we construct a graph \mathcal{G} . For every boundary cycle—inner and outer—there is a node in \mathcal{G} . There is also one node for the imaginary outer boundary of the unbounded face. There is an arc between two cycles if and only if one of the cycles is the boundary of a hole and the other cycle has a half-edge immediately to the left of the leftmost vertex of that hole cycle. If there is no half-edge to the left of the leftmost vertex of a cycle, then the node representing the cycle is linked to the node of the unbounded face. Figure 2.6 gives an example. The dotted segments in the figure indicate the linking of the hole cycles to other cycles. The graph corresponding to the subdivision is also shown in the figure. The hole cycles are shown as single circles, and the outer boundary cycles are shown as double circles. Observe that C_3 and C_6 are in the same connected component as C_2 . This indicates that C_3 and C_6 are hole cycles in the face whose outer boundary is C_2 . If there is only one hole in a face f , then the graph \mathcal{G} links the boundary cycle of the hole to the outer boundary of f . In general this need not be the case: a hole can also be linked to another hole, as you can see in Figure 2.6. This hole, which lies in the same face f , may be linked to the outer boundary of f , or it may be linked to yet another hole. But eventually we must end up linking a hole to the outer boundary, as the next lemma shows.

Lemma 2.5 *Each connected component of the graph \mathcal{G} corresponds exactly to the set of cycles incident to one face.*

Proof. Consider a cycle C bounding a hole in a face f . Because f lies locally to the left of the leftmost vertex of C , C must be linked to another cycle that also

Section 2.3

COMPUTING THE OVERLAY OF TWO SUBDIVISIONS

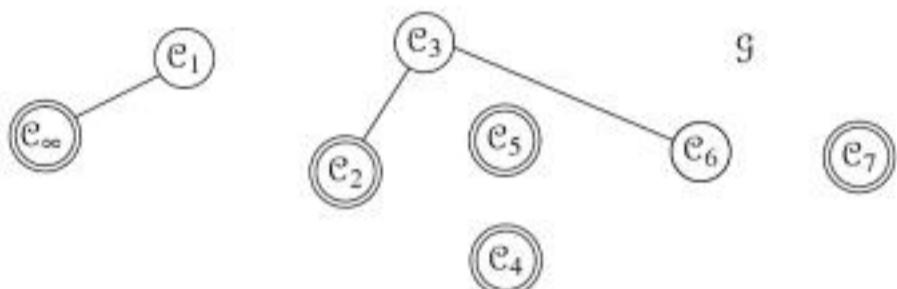
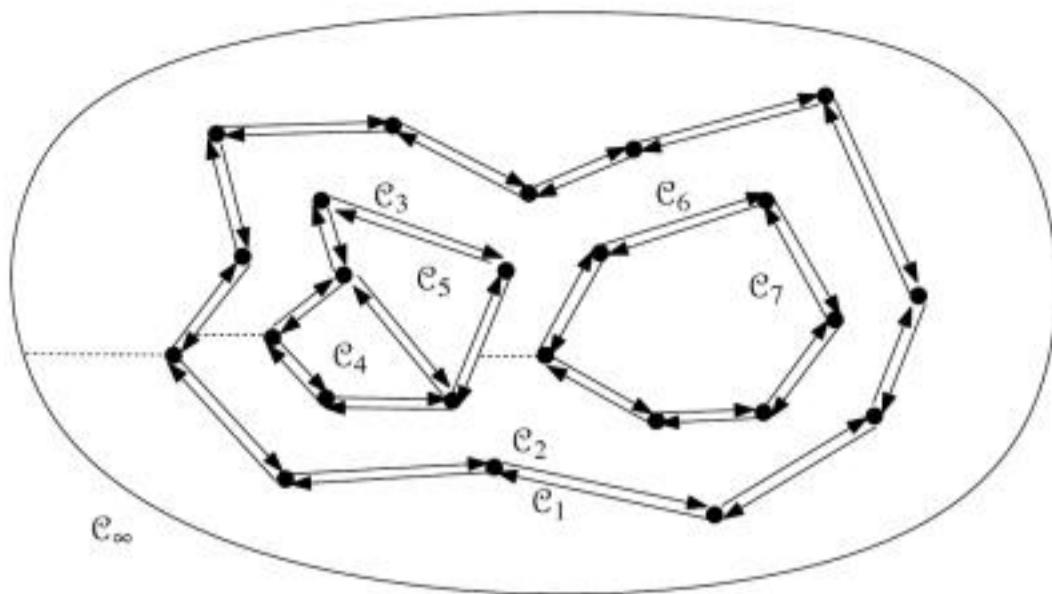


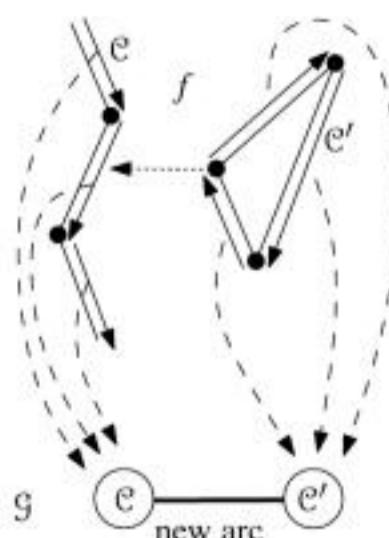
Figure 2.6

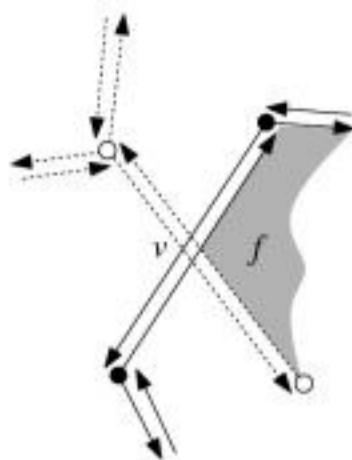
A subdivision and the corresponding graph \mathcal{G}

bounds f . It follows that cycles in the same connected component of \mathcal{G} bound the same face.

To finish the proof, we show that every cycle bounding a hole in f is in the same connected component as the outer boundary of f . Suppose there is a cycle for which this is not the case. Let \mathcal{C} be the leftmost such cycle, that is, the one whose the leftmost vertex is leftmost. By definition there is an arc between the \mathcal{C} and another cycle \mathcal{C}' that lies partly to the left of the leftmost vertex of \mathcal{C} . Hence, \mathcal{C}' is in the same connected component as \mathcal{C} , which is not the component of the outer boundary of f . This contradicts the definition of \mathcal{C} . \square

Lemma 2.5 shows that once we have the graph \mathcal{G} , we can create a face record for every component. Then we can set the *IncidentFace()* pointers of the half-edges that bound each face f , and we can construct the list *InnerComponents*(f) and the set *OuterComponent*(f). How can we construct \mathcal{G} ? Recall that in the plane sweep algorithm for line segment intersection we always looked for the segments immediately to the left of an event point. (They had to be tested for intersection against the leftmost edge through the event point.) Hence, the information we need to construct \mathcal{G} is determined during the plane sweep. So, to construct \mathcal{G} , we first make a node for every cycle. To find the arcs of \mathcal{G} , we consider the leftmost vertex v of every cycle bounding a hole. If \vec{e} is the half-edge immediately left of v , then we add an arc between the two nodes in \mathcal{G} representing the cycle containing \vec{e} and the hole cycle of which v is the leftmost vertex. To find these nodes in \mathcal{G} efficiently we need pointers from every half-edge record to the node in \mathcal{G} representing the cycle it is in. So the face information of the doubly-connected edge list can be set in $O(n+k)$ additional time, after the plane sweep.





One thing remains: each face f in the overlay must be labeled with the names of the faces in the old subdivisions that contained it. To find these faces, consider an arbitrary vertex v of f . If v is the intersection of an edge e_1 from S_1 and an edge e_2 from S_2 then we can decide which faces of S_1 and S_2 contain f by looking at the $\text{IncidentFace}()$ pointer of the appropriate half-edges corresponding to e_1 and e_2 . If v is not an intersection but a vertex of, say, S_1 , then we only know the face of S_1 containing f . To find the face of S_2 containing f , we have to do some more work: we have to determine the face of S_2 that contains v . In other words, if we knew for each vertex of S_1 in which face of S_2 it lay, and vice versa, then we could label the faces of $\mathcal{O}(S_1, S_2)$ correctly. How can we compute this information? The solution is to apply the paradigm that has been introduced in this chapter, plane sweep, once more. However, we won't explain this final step here. It is a good exercise to test your understanding of the plane sweep approach to design the algorithm yourself. (In fact, it is not necessary to compute this information in a separate plane sweep. It can also be done in the sweep that computes the intersections.)

Putting everything together we get the following algorithm.

Algorithm MAPOVERLAY(S_1, S_2)

Input. Two planar subdivisions S_1 and S_2 stored in doubly-connected edge lists.

Output. The overlay of S_1 and S_2 stored in a doubly-connected edge list \mathcal{D} .

1. Copy the doubly-connected edge lists for S_1 and S_2 to a new doubly-connected edge list \mathcal{D} .
2. Compute all intersections between edges from S_1 and S_2 with the plane sweep algorithm of Section 2.1. In addition to the actions on \mathcal{T} and \mathcal{Q} required at the event points, do the following:
 - Update \mathcal{D} as explained above if the event involves edges of both S_1 and S_2 . (This was explained for the case where an edge of S_1 passes through a vertex of S_2 .)
 - Store the half-edge immediately to the left of the event point at the vertex in \mathcal{D} representing it.
3. (* Now \mathcal{D} is the doubly-connected edge list for $\mathcal{O}(S_1, S_2)$, except that the information about the faces has not been computed yet. *)
4. Determine the boundary cycles in $\mathcal{O}(S_1, S_2)$ by traversing \mathcal{D} .
5. Construct the graph \mathcal{G} whose nodes correspond to boundary cycles and whose arcs connect each hole cycle to the cycle to the left of its leftmost vertex, and compute its connected components. (The information to determine the arcs of \mathcal{G} has been computed in line 2, second item.)
6. **for** each connected component in \mathcal{G}
7. **do** Let \mathcal{C} be the unique outer boundary cycle in the component and let f denote the face bounded by the cycle. Create a face record for f , set $\text{OuterComponent}(f)$ to some half-edge of \mathcal{C} , and construct the list $\text{InnerComponents}(f)$ consisting of pointers to one half-edge in each hole cycle in the component. Let the $\text{IncidentFace}()$ pointers of all half-edges in the cycles point to the face record of f .



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

and $O(n)$ storage by Mairson and Stolfi [262] before the general problem was solved optimally. Other optimal red-blue intersection algorithms were given by Chazelle et al. [101] and by Palazzi and Snoeyink [315]. If the two sets of segments form connected subdivisions then the situation is even better: in this case the overlay can be computed in $O(n + k)$ time, as has been shown by Finke and Hinrichs [176]. Their result generalizes and improves previous results on map overlay by Nievergelt and Preparata [293], Guibas and Seidel [200], and Mairson and Stolfi [262].

The line segment intersection counting problem is to determine the number of intersection points in a set of n line segments. Since the output is a single integer, a term with k in the time bound no longer refers to the output size (which is constant), but only to the number of intersections. Algorithms that do not depend on the number of intersections take $O(n^{4/3} \log^c n)$ time, for some small constant c [4, 95]; a running time close to $O(n \log n)$ is not known to exist.

Plane sweep is one of the most important paradigms for designing geometric algorithms. The first algorithms in computational geometry based on this paradigm are by Shamos and Hoey [351], Lee and Preparata [250], and Bentley and Ottmann [47]. Plane sweep algorithms are especially suited for finding intersections in sets of objects, but they can also be used for solving many other problems. In Chapter 3 plane sweep solves part of the polygon triangulation problem, and in Chapter 7 we will see a plane sweep algorithm to compute the so-called Voronoi diagram of a set of points. The algorithm presented in the current chapter sweeps a horizontal line downwards over the plane. For some problems it is more convenient to sweep the plane in another way. For instance, we can sweep the plane with a rotating line—see Chapter 15 for an example—or with a pseudo-line (a line that need not be straight, but otherwise behaves more or less as a line) [159]. The plane sweep technique can also be used in higher dimensions: here we sweep the space with a hyperplane [213, 311, 324]. Such algorithms are called space sweep algorithms.

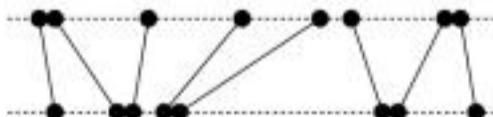
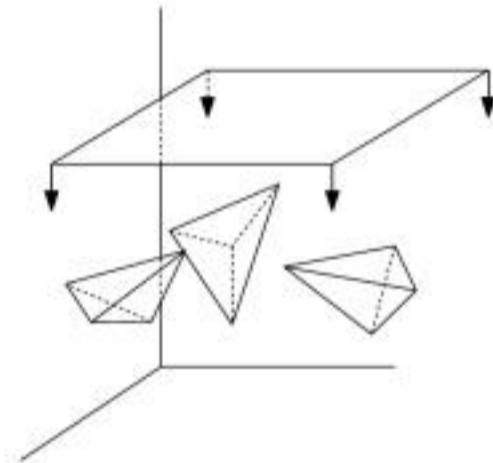
In this chapter we described a data structure for storing subdivisions: the doubly-connected edge list. This structure, or in fact a variant of it, was described by Muller and Preparata [286]. There are also other data structures for storing subdivisions, such as the winged edge structure by Baumgart [40] and the quad edge structure by Guibas and Stolfi [202]. The difference between all these structures is small. They all have more or less the same functionality, but some save a few bytes of storage per edge.

Section 2.6

EXERCISES

2.6 Exercises

- 2.1 Let S be a set of n disjoint line segments whose upper endpoints lie on the line $y = 1$ and whose lower endpoints lie on the line $y = 0$. These segments partition the horizontal strip $[-\infty : \infty] \times [0 : 1]$ into $n + 1$ regions. Give an $O(n \log n)$ time algorithm to build a binary search tree on the segments



in S such that the region containing a query point can be determined in $O(\log n)$ time. Also describe the query algorithm in detail.

- 2.2 The intersection detection problem for a set S of n line segments is to determine whether there exists a pair of segments in S that intersect. Give a plane sweep algorithm that solves the intersection detection problem in $O(n \log n)$ time.
- 2.3 Change the code of Algorithm FINDINTERSECTIONS (and of the procedures that it calls) such that the working storage is $O(n)$ instead of $O(n + k)$.
- 2.4 Let S be a set of n line segments in the plane that may (partly) overlap each other. For example, S could contain the segments $(0, 0)(1, 0)$ and $(-1, 0)(2, 0)$. We want to compute all intersections in S . More precisely, we want to compute each proper intersection of two segments in S (that is, each intersection of two non-parallel segments) and for each endpoint of a segment all segments containing the point. Adapt algorithm FINDINTERSECTIONS to this end.
- 2.5 Which of the following equalities are always true?

$$\begin{aligned} \text{Twin}(\text{Twin}(\vec{e})) &= \vec{e} \\ \text{Next}(\text{Prev}(\vec{e})) &= \vec{e} \\ \text{Twin}(\text{Prev}(\text{Twin}(\vec{e}))) &= \text{Next}(\vec{e}) \\ \text{IncidentFace}(\vec{e}) &= \text{IncidentFace}(\text{Next}(\vec{e})) \end{aligned}$$

- 2.6 Give an example of a doubly-connected edge list where for an edge e the faces $\text{IncidentFace}(\vec{e})$ and $\text{IncidentFace}(\text{Twin}(\vec{e}))$ are the same.
- 2.7 Given a doubly-connected edge list representation of a subdivision where $\text{Twin}(\vec{e}) = \text{Next}(\vec{e})$ holds for every half-edge \vec{e} , how many faces can the subdivision have at most?
- 2.8 Give pseudocode for an algorithm that lists all vertices adjacent to a given vertex v in a doubly-connected edge list. Also, give pseudocode for an algorithm that lists all edges that bound a face in a not necessarily connected subdivision.
- 2.9 Suppose that a doubly-connected edge list of a connected subdivision is given. Give pseudocode for an algorithm that lists all faces with vertices that appear on the outer boundary.
- 2.10 Let S be a subdivision of complexity n , and let P be a set of m points. Give a plane sweep algorithm that computes for every point in P in which face of S it is contained. Show that your algorithm runs in $O((n + m) \log(n + m))$ time.
- 2.11 Let S be a set of n circles in the plane. Describe a plane sweep algorithm to compute all intersection points between the circles. (Because we deal

with circles, not discs, two circles do not intersect if one lies entirely inside the other.) Your algorithm should run in $O((n+k)\log n)$ time, where k is the number of intersection points.

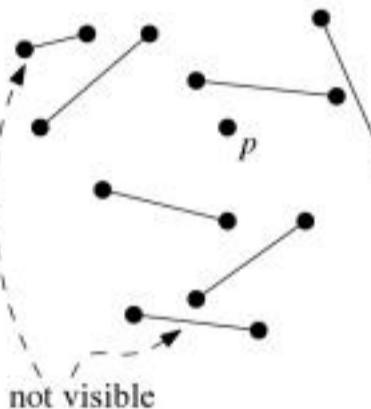
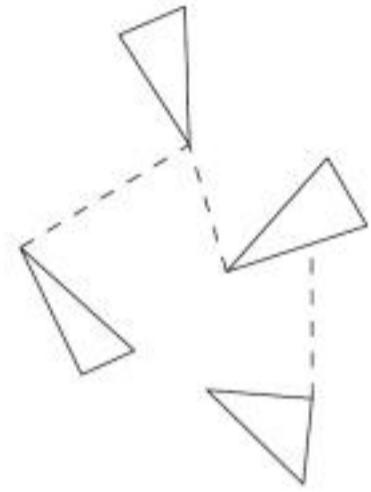
- 2.12 Let S be a set of n triangles in the plane. The boundaries of the triangles are disjoint, but it is possible that a triangle lies completely inside another triangle. Let P be a set of n points in the plane. Give an $O(n \log n)$ algorithm that reports each point in P lying outside all triangles.
- 2.13* Let S be a set of n disjoint triangles in the plane. We want to find a set of $n - 1$ segments with the following properties:

- Each segment connects a point on the boundary of one triangle to a point on the boundary of another triangle.
- The interiors of the segments are pairwise disjoint and they are disjoint from the triangles.
- Together they connect all triangles to each other, that is, by walking along the segments and the triangle boundaries it must be possible to walk from a triangle to any other triangle.

Section 2.6

EXERCISES

- Develop a plane sweep algorithm for this problem that runs in $O(n \log n)$ time. State the events and the data structures that you use explicitly, and describe the cases that arise and the actions required for each of them. Also state the sweep invariant.
- 2.14 Let S be a set of n disjoint line segments in the plane, and let p be a point not on any of the line segments of S . We wish to determine all line segments of S that p can see, that is, all line segments of S that contain some point q so that the open segment \overline{pq} doesn't intersect any line segment of S . Give an $O(n \log n)$ time algorithm for this problem that uses a rotating half-line with its endpoint at p .



3 Polygon Triangulation

Guarding an Art Gallery

Works of famous painters are not only popular among art lovers, but also among criminals. They are very valuable, easy to transport, and apparently not so difficult to sell. Art galleries therefore have to guard their collections carefully.

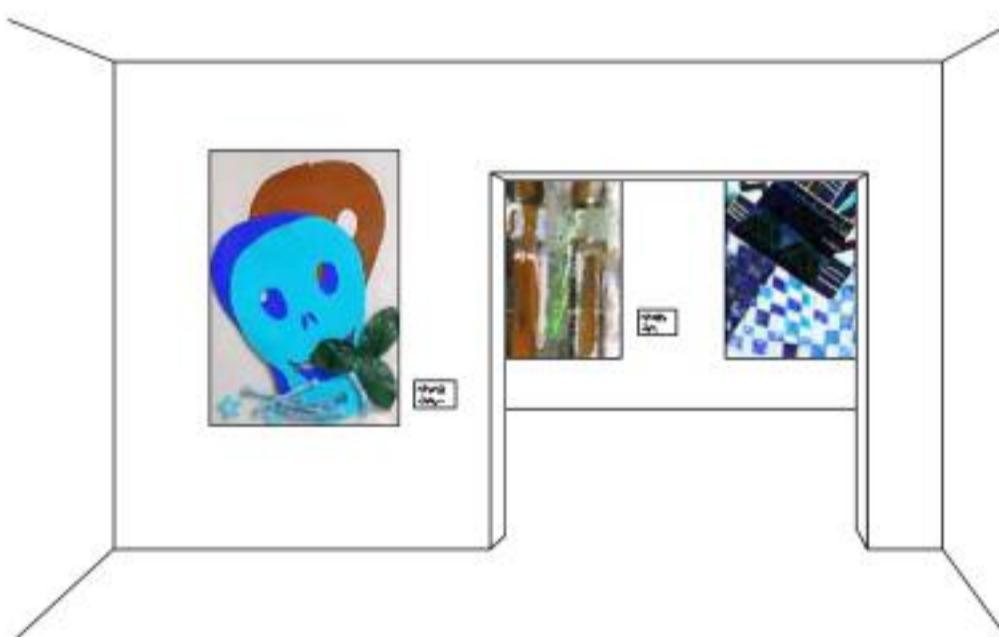
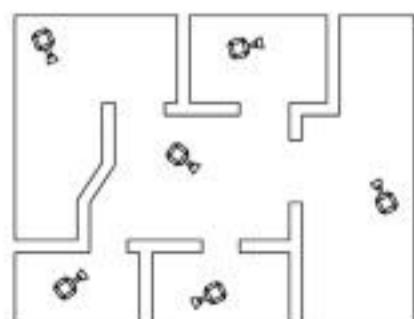
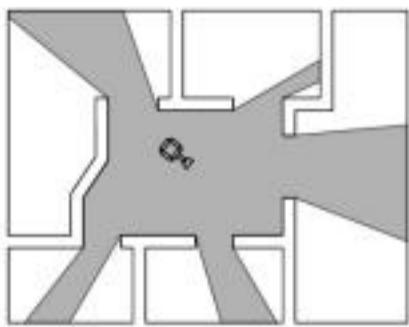


Figure 3.1
An art gallery

During the day the attendants can keep a look-out, but at night this has to be done by video cameras. These cameras are usually hung from the ceiling and they rotate about a vertical axis. The images from the cameras are sent to TV screens in the office of the night watch. Because it is easier to keep an eye on few TV screens rather than on many, the number of cameras should be as small as possible. An additional advantage of a small number of cameras is that the cost of the security system will be lower. On the other hand we cannot have too few cameras, because every part of the gallery must be visible to at least one of them. So we should place the cameras at strategic positions, such that each of them guards a large part of the gallery. This gives rise to what is usually referred to as the *Art Gallery Problem*: how many cameras do we need to guard a given gallery and how do we decide where to place them?





3.1 Guarding and Triangulations

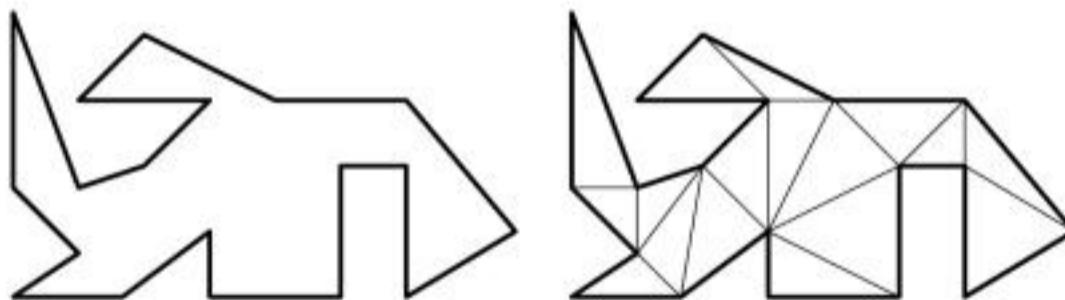
If we want to define the art gallery problem more precisely, we should first formalize the notion of gallery. A gallery is, of course, a 3-dimensional space, but a floor plan gives us enough information to place the cameras. Therefore we model a gallery as a polygonal region in the plane. We further restrict ourselves to regions that are *simple polygons*, that is, regions enclosed by a single closed polygonal chain that does not intersect itself. Thus we do not allow regions with holes. A camera position in the gallery corresponds to a point in the polygon. A camera sees those points in the polygon to which it can be connected with an open segment that lies in the interior of the polygon.

How many cameras do we need to guard a simple polygon? This clearly depends on the polygon at hand: the more complex the polygon, the more cameras are required. We shall therefore express the bound on the number of cameras needed in terms of n , the number of vertices of the polygon. But even when two polygons have the same number of vertices, one can be easier to guard than the other. A convex polygon, for example, can always be guarded with one camera. To be on the safe side we shall look at the worst-case scenario, that is, we shall give a bound that is good for any simple polygon with n vertices. (It would be nice if we could find the minimum number of cameras for the specific polygon we are given, not just a worst-case bound. Unfortunately, the problem of finding the minimum number of cameras for a given polygon is NP-hard.)

Let \mathcal{P} be a simple polygon with n vertices. Because \mathcal{P} may be a complicated shape, it seems difficult to say anything about the number of cameras we need to guard \mathcal{P} . Hence, we first decompose \mathcal{P} into pieces that are easy to guard, namely triangles. We do this by drawing *diagonals* between pairs of vertices.

Figure 3.2

A simple polygon and a possible triangulation of it



A diagonal is an open line segment that connects two vertices of \mathcal{P} and lies in the interior of \mathcal{P} . A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals is called a *triangulation* of the polygon—see Figure 3.2. (We require that the set of non-intersecting diagonals be maximal to ensure that no triangle has a polygon vertex in the interior of one of its edges. This could happen if the polygon has three consecutive collinear vertices.) Triangulations are usually not unique; the polygon in Figure 3.2, for example, can be triangulated in many different ways. We can guard \mathcal{P} by placing a camera in every triangle of a triangulation $\mathcal{T}_{\mathcal{P}}$ of \mathcal{P} . But does a triangulation always exist? And how many triangles can there be in a triangulation? The following theorem answers these questions.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Take the plane spanned by the vectors (we assume both vectors are rooted at the origin); the angle of the vectors is the smaller of the two angles measured in this plane. Now \hat{f} blocks any translation in a direction making an angle of less than 90° with $\vec{\eta}(f)$, the outward normal of f . So a necessary condition on \vec{d} is that it makes an angle of at least 90° with the outward normal of every ordinary facet of \mathcal{P} . The next lemma shows that this condition is also sufficient.

Lemma 4.1 *The polyhedron \mathcal{P} can be removed from its mold by a translation in direction \vec{d} if and only if \vec{d} makes an angle of at least 90° with the outward normal of all ordinary facets of \mathcal{P} .*

Proof. The “only if” part is easy: if \vec{d} made an angle less than 90° with some outward normal $\vec{\eta}(f)$, then any point q in the interior of f collides with the mold when translated in direction \vec{d} .

To prove the “if” part, suppose that at some moment \mathcal{P} collides with the mold when translated in direction \vec{d} . We have to show that there must be an outward normal making an angle of less than 90° with \vec{d} . Let p be a point of \mathcal{P} that collides with a facet \hat{f} of the mold. This means that p is about to move into the interior of the mold, so $\vec{\eta}(\hat{f})$, the outward normal of \hat{f} , must make an angle greater than 90° with \vec{d} . But then \vec{d} makes an angle less than 90° with the outward normal of the ordinary facet f of \mathcal{P} that corresponds to \hat{f} . \square

Lemma 4.1 has an interesting consequence: if \mathcal{P} can be removed by a sequence of small translations, then it can be removed by a single translation. So allowing for more than one translation does not help in removing the object from its mold.

We are left with the task of finding a direction \vec{d} that makes an angle of at least 90° with the outward normal of each ordinary facet of \mathcal{P} . A direction in 3-dimensional space can be represented by a vector rooted at the origin. We already know that we can restrict our attention to directions with a positive z -component. We can represent all such directions as points in the plane $z = 1$, where the point $(x, y, 1)$ represents the direction of the vector $(x, y, 1)$. This way every point in the plane $z = 1$ represents a unique direction, and every direction with a positive z -value is represented by a unique point in that plane.

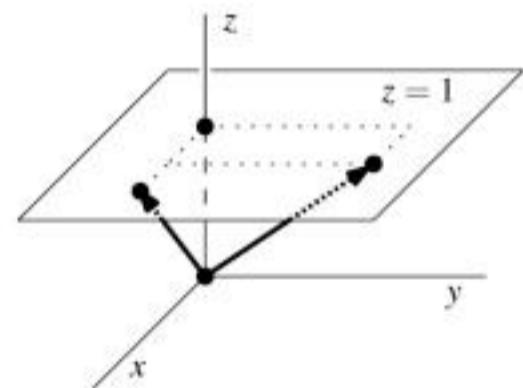
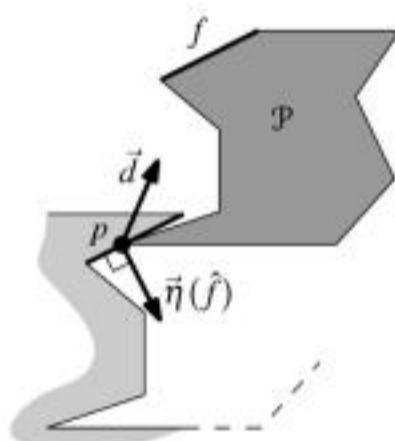
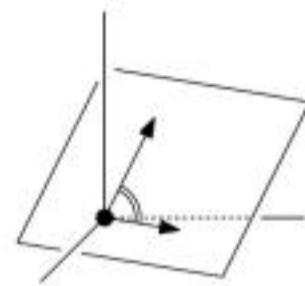
Lemma 4.1 gives necessary and sufficient conditions on the removal direction \vec{d} . How do these conditions translate into our plane of directions? Let $\vec{\eta} = (\vec{\eta}_x, \vec{\eta}_y, \vec{\eta}_z)$ be the outward normal of an ordinary facet. The direction $\vec{d} = (d_x, d_y, 1)$ makes an angle at least 90° with $\vec{\eta}$ if and only if the dot product of \vec{d} and $\vec{\eta}$ is non-positive. Hence, an ordinary facet induces a constraint of the form

$$\vec{\eta}_x d_x + \vec{\eta}_y d_y + \vec{\eta}_z \leq 0.$$

This inequality describes a half-plane on the plane $z = 1$, that is, the area left or the area right of a line on the plane. (This last statement is not true for horizontal facets, which have $\vec{\eta}_x = \vec{\eta}_y = 0$. In this case the constraint is either impossible to satisfy or always satisfied, which is easy to test.) Hence, every non-horizontal facet of \mathcal{P} defines a closed half-plane on the plane $z = 1$, and any point in the

Section 4.1

THE GEOMETRY OF CASTING





common intersection of these half-planes corresponds to a direction in which \mathcal{P} can be removed. The common intersection of these half-planes may be empty; in this case \mathcal{P} cannot be removed from the given mold.

We have transformed our manufacturing problem to a purely geometric problem in the plane: given a set of half-planes, find a point in their common intersection or decide that the common intersection is empty. If the polyhedron to be manufactured has n facets, then the planar problem has at most $n - 1$ half-planes (the top facet does not induce a half-plane). In the next sections we will see that the planar problem just stated can be solved in expected linear time—see Section 4.4, where also the meaning of “expected” is explained.

Recall that the geometric problem corresponds to testing whether \mathcal{P} can be removed from a given mold. If this is impossible, there can still be other molds, corresponding to different choices of the top facet, from which \mathcal{P} is removable. In order to test whether \mathcal{P} is castable, we try all its facets as top facets. This leads to the following result.

Theorem 4.2 *Let \mathcal{P} be a polyhedron with n facets. In $O(n^2)$ expected time and using $O(n)$ storage it can be decided whether \mathcal{P} is castable. Moreover, if \mathcal{P} is castable, a mold and a valid direction for removing \mathcal{P} from it can be computed in the same amount of time.*

4.2 Half-Plane Intersection

Let $H = \{h_1, h_2, \dots, h_n\}$ be a set of linear constraints in two variables, that is, constraints of the form

$$a_i x + b_i y \leq c_i,$$

where a_i , b_i , and c_i are constants such that at least one of a_i and b_i is non-zero. Geometrically, we can interpret such a constraint as a closed half-plane in \mathbb{R}^2 , bounded by the line $a_i x + b_i y = c_i$. The problem we consider in this section is to find the set of all points $(x, y) \in \mathbb{R}^2$ that satisfy all n constraints at the same time. In other words, we want to find all the points lying in the common intersection of the half-planes in H . (In the previous section we reduced the casting problem to finding *some* point in the intersection of a set of half-planes. The problem we study now is more general.)

The shape of the intersection of a set of half-planes is easy to determine: a half-plane is convex, and the intersection of convex sets is again a convex set, so the intersection of a set of half-planes is a convex region in the plane. Every point on the intersection boundary must lie on the bounding line of some half-plane. Hence, the boundary of the region consists of edges contained in these bounding lines. Since the intersection is convex, every bounding line can contribute at most one edge. It follows that the intersection of n half-planes is a convex polygonal region bounded by at most n edges. Figure 4.2 shows a few examples of intersections of half-planes. To which side of its bounding



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



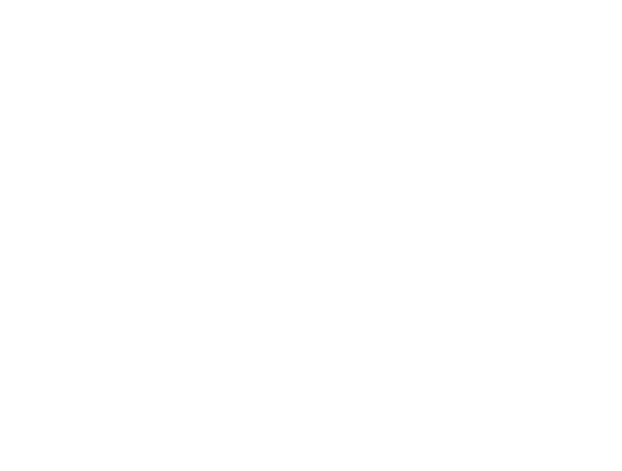
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



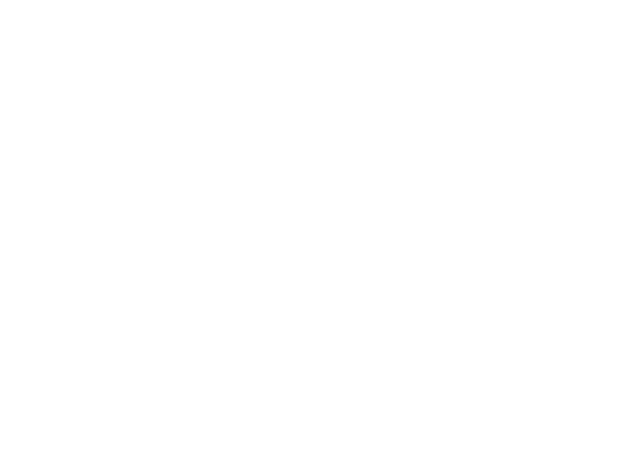
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



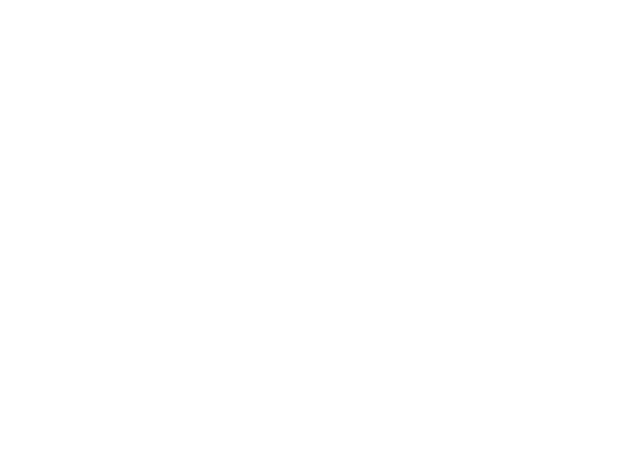
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



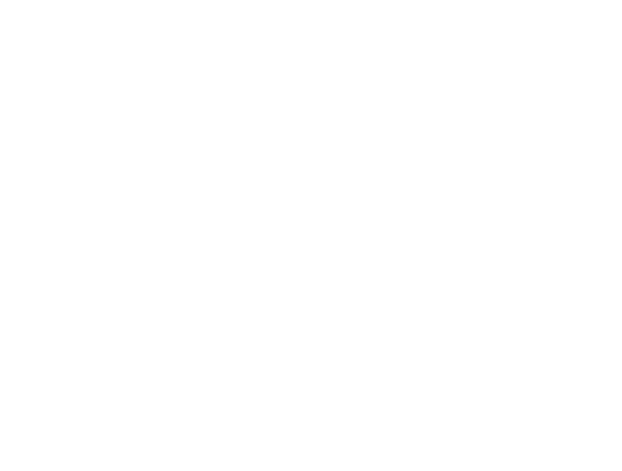
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

INDEX

- forbidden configuration space, 285, 330
FORBIDDENSPACE, 298
Fortune's algorithm, 151
fractional cascading, 109, 112, 143, 221
free configuration space, 285, 324, 330
free path, 305
free space, 285, 324, 330
 representation of, 287
 trapezoidal map of, 287, 324
free split, 265
- Gabriel graph, 215, 217
general position, 9, 124
GENERATEMESH, 316
genus, 245
geographic information systems, 1, 11, 15
geometric graph, 215
geometric modeling, 15
GIS, 11
Graham's scan, 13
graph
 Gabriel, 215, 217
 geometric, 215
 relative neighborhood, 215, 217
 visibility, 323
grid, 116
guard
 for low-density scene, 272
- half-edge, 31
 destination of, 31
 origin of, 31
 record of, 32
half-plane discrepancy, 175
half-planes
 common intersection of, 66, 89
HANDLECIRCLEEVENT, 158
HANDLEENDVERTEX, 53
HANDLEEVENTPOINT, 26
HANDLEMERGEVERTEX, 54
HANDLEREGULARVERTEX, 54
HANDLESITEEVENT, 158
HANDLESPLITVERTEX, 53
HANDLESTARTVERTEX, 53
harmonic number, 135
heap, 227
Helly-type theorem, 90
hidden surface removal, 259
higher-dimensional linear programming, 82
- higher-order Voronoi diagram, 169
horizon, 247
hull
 convex, 2, 89, 193, 243
 lower, 6, 254
 upper, 6, 253
- illegal edge, 194
implicit point location, 144
incidence preserving, 178
incident, 30
infeasible linear program, 71
infeasible point, 71
inner vertex, 325
INSERTSEGMENTTREE, 234
interpolation
 data-independent, 214
 linear, 191
INTERSECTHALFPLANES, 67
intersection
 of half-planes, 66, 89
 of line segments, 19
 of polygons, 39
intersection-sensitive algorithm, 21
interval
 elementary, 232
interval tree, 220, 222, 237
inversion, 186
isolated vertex, 31
iterated logarithm, 60
- jaggies, 174
Jarvis's march, 13
joint
 of robot, 283
 prismatic, 283
 revolute, 283
- k*-level
 in arrangement, 187
k-set, 187
kd-tree, 100, 116
- L_1 -metric, 168
 L_2 -metric, 148, 169, 332
 L_p -metric, 168
layer, 19, 335
layered range tree, 113
legal triangulation, 195
LEGALIZEEDGE, 201
LEGALTRIANGULATION, 195



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- S**earchKDTree, 103
 second-level tree, 106, 344
 see, 325
 segment
 axis-parallel, 220
 segment tree, 231, 233, 237
SELECTBELOWPAIR, 350
SELECTBELOWPOINT, 348
SELECTINHALFPLANE, 339
SELECTINTSEGMENTS, 344
 semi-free path, 305
 shear transformation, 137
 shortest path, 323
 for polygonal robot, 330
 in graph, 331
SHORTESTPATH, 326
 side
 in trapezoidal map, 125
 simple arrangement, 180
 simple polygon, 46
 simplex algorithm, 72, 90
 simplex range query, 352
 lower bound, 352
 simplicial partition, 337
 simplicial polytope, 245
 single cell, 304
 site, 147
 line segment, 160
 site event, 153, 161
 skeleton
 of polygon, 169
 slab, 122, 235
SLOWCONVEXHULL, 3
 smallest enclosing ball, 90
 smallest enclosing disc, 86
 smallest enclosing ellipse, 90
 smallest-width annulus, 163
 solution
 feasible, 71
 split
 free, 265
 split vertex, 50
 stabber, 189, 190
 stabbing counting query, 237
 stabbing number
 of polygon, 61
 stabbing query, 237
 star-shaped polygon, 93, 145
 start vertex, 50
 status
 of sweep line, 22, 25, 52, 155, 328
 status structure, 25, 52, 155, 328
 Steiner point, 214, 309
 Steiner triangulation, 309
 strictly monotone polygon, 55
 structural change, 205, 210, 250
 subdivision, 30, 121
 complexity of, 30
 connected, 30
 maximal planar, 193
 quadtree, 310
 representation of, 29
 subdivision overlay, 33
 sum
 Minkowski, 291
 of two points, 291
 vector, 291
 supersampling, 174
 sweep algorithm, 22, 34, 51, 68, 151, 328
 rotational, 328
 sweep line, 22, 51, 151, 160
 symbolic perturbation, 9, 14
 tail estimate, 140
 terrain, 191
 domain of, 191
 polyhedral, 192
 tetrahedralization
 of polytope, 60
 Thales' Theorem, 194
 thematic map layer, 19, 335
 thematic map overlay, 20
 top facet, 64
 trading area, 147
 transform
 duality, 177
 inversion, 186
 shear, 137
 translating polygon
 configuration space of, 291
 configuration-space obstacle for, 291
 transversal, 189, 190
 trapezoidal decomposition, 124
 trapezoidal map, 122, 124, 287, 324
 complexity of, 127
 computation, 128
 of the free space, 287, 324
 pseudo code, 129



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.