Mark de Berg

Ccc1-703

Ray Shooting, Depth Orders and Hidden Surface Removal

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, FRG

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Author

Mark de Berg
Department of Computer Science, Utrecht University
Padualaan 14, NL-3508 TB Utrecht, The Netherlands

CR Subject Classification (1991): I.3.5-8, I.4.8

ISBN 3-540-57020-9 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-57020-9 Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera ready by author
45/3140-543210 - Printed on acid-free paper

Preface

Computational geometry is the part of theoretical computer science that concerns itself with geometric objects; it tries to design efficient algorithms for problems involving points, lines, polygons, and so on. The field has gained popularity very rapidly during the last decade. This is partly due to the many application areas of computational geometry and partly due to the beauty of the field itself. I hope that the reader will experience some of this beauty when reading this book. Initially, most research was directed towards problems in two-dimensional space, and good solutions to many of these problems have been found. In recent years the field has shifted its attention to problems in three- (and higher) dimensional space. This shift of attention is also reflected in this book: almost everything that is studied is three-dimensional.

This book focuses on three problems in computational geometry that arise in computer graphics. (A more ample discussion of the relation of these problems to computer graphics can be found in Chapter 1.) The first problem is the ray shooting problem: preprocess a set of polyhedra into a data structure such that the first polyhedron that is hit by a query ray can be determined quickly. The second problem is that of computing depth orders: we want to sort a set of polyhedra such that if one polyhedron is (partially) obscured by another polyhedron then it come first in the order. The third problem that we study is the hidden surface removal problem: given a set of polyhedra and a view point, compute which parts of the polyhedra are visible from the view point. These are not only three nice problems that arise naturally in one of the application areas of computational geometry; they involve issues that are fundamental to three-dimensional computational geometry and are, hence, also of considerable theoretical interest. The book also contains a large introductory part, which discusses the techniques that will be used to tackle the three problems stated above. This part should be interesting not only to those who want to read the rest of the book but miss the necessary background, but to anyone who wants to know more about some (recent) techniques in computational geometry.

This book is a revised version of my Ph.D. thesis, which was the result of the research I did at the department of computer science of Utrecht University (the Netherlands) in the period from August 1988 to March 1992. The research was supported by the Dutch Organization for Scientific Research (N.W.O.), and partially by the ESPRIT Basic Research Action No. 3075 (project ALCOM). There are many people without whom this book would not have been what it is now. First of all, there is Mark Overmars, who introduced me to the fascinating field of computational geometry: I could not have wished myself a more stimulating supervisor for my Ph.D. research. Secondly, there is Marc van Kreveld, my roommate during all those years:

they would certainly have been less pleasant without the many discussions we had on various subjects (including computational geometry). Practically everything in this book has profited from their comments and suggestions. Let me also mention the other people that contributed to the research that is reported in this book: Hazel Everett, Danny Halperin, Otfried Schwarzkopf, Jack Snoeyink and Hubert Wagener. Finally, I thank Jan van Leeuwen, Kees van Overveld, Micha Sharir and Emo Welzl for being in the reading committee of my thesis.

Utrecht, April 1993 Mark de Berg

Contents

Part A

Introduction

Chapter 1

Computational Geometry and Computer Graphics

This book studies three problems in computational geometry that arise in computer graphics. Computational geometry is the part of theoretical computer science that concerns itself with geometric objects; it tries to design efficient algorithms for problems involving points, lines, polyhedra, et cetera, in one-, two- and more-dimensional space. A paper [114] by Michael Shamos in 1975 marks the beginning of computational geometry as a separate area in algorithms research. Since then computational geometry has attracted many researchers and it has made tremendous progress. By now it has established itself as an important field within theoretical computer science: hundreds of people are working in the field, thousands of papers have been published, every major conference or journal on theoretical computer science has papers on it, and there are even several journals and conferences which are devoted solely to computational geometry.

What is it that makes computational geometry so appealing? The reason for this is twofold. First of all, most problems have a simple and intuitive definition and are easily visualized, which makes it easy to interest people in them. In the second place, the applications of computational geometry are numerous, in particular in areas like computer graphics, robotics, geographical information systems (GIS), databases, and VLSI design. Computational geometry offers the right abstraction to study the problems that arise in these areas. Let us give a few prime examples of these applications.

Consider a database that stores information about, say, the age and number of children of people. A typical query in this database is of the form: "Which persons are between 20 and 65 years of age, and have between 13 and 18 children?" If we represent each person by a point in a two-dimensional space, where the first coordinate of the point is the age of the person and the second coordinate is the number of children, then the query asks for all points in the rectangle $[20:65] \times [13:18]$. See Figure 1.1. This problem is called the *orthogonal range searching problem* in computational geometry, and it has been studied extensively.

A second example can be found in what are called *motion planning problems* in computational geometry. Who did not have the frustrating experience of a sofa that

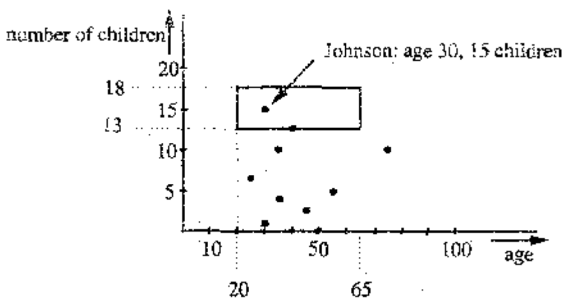

Figure 1.1: A query in a database.

refuses to leave through the same door that it was brought in through? Planning the motion of an object from a starting position to a goal position is an essential problem in robotics. Motion planning problems have been studied in great detail in computational geometry, leading to papers with intriguing titles like "How to Move a Chair through a Door"[126] and "On the Piano Movers' Problem, I: The Case of a Two-Dimensional Rigid Polygonal Body Moving amidst Polygonal Barriers"[113].

A third important application area is computer graphics. Since the problems that are studied in this book originate in this area, we discuss it in more detail.

Computer graphics concerns itself with the display of visual information on the screen of a computer terminal. We can roughly subdivide computer graphics into two subareas. One considers hardware aspects, and the other studies algorithmic aspects. It is the latter area where computational geometry comes into play and which has our interest. There is an enormous amount of literature on computer graphics, and we will not even try to give a survey. We just mention two good textbooks, by Foley et al. [58] and by Watt [120], which the interested reader can consult.

The most fundamental algorithmic problems arise when one wants to display the view of a three-dimensional scene. As an example, consider an architect who is designing a building. It would be useful for her to see the building before it is actually constructed. This can be accomplished using computer graphics. The architect tells the system exactly where walls, doors, windows, tables, chairs and other objects are located, what their shape is, and so on, and the system calculates what the building looks like for an observer standing, say, at the main entrance. The system might even compute what the building looks like from the inside, giving the architect the possibility to walk through the building, so to speak. Thus the system has to compute the view of a collection of objects in three-dimensional space as seen from a certain view point. A simple illustration is given in Figure 1.2. The problem of determining which parts of each object are visible and which parts are hidden is called the *hidden surface removal problem*.

To understand the different approaches to the hidden surface removal problem, one has to know how a single, completely visible, object is displayed onto the screen. A computer screen consists of many small dots (typically about one million) called

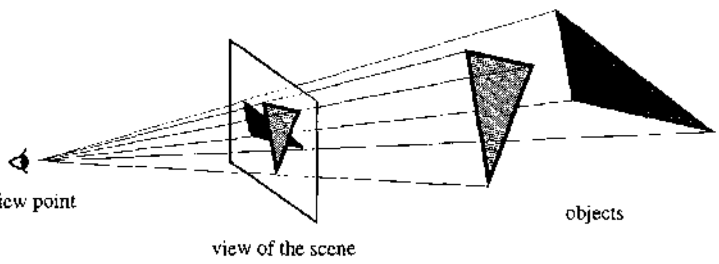

Figure 1.2: Viewing two triangles in three-dimensional space.

pixels. To display an object it is projected onto the screen with the view point as the center of projection. The projected image of the object covers a number of pixels on the screen. These pixels get the color of the object, and the remaining pixels get the background color. A realistic scene consists of many objects, and a pixel can be covered by several of them. It is for the computer to decide which object is visible at each pixel, so that it can give the pixels the right color. Hidden surface removal algorithms are used to compute which object is visible at which place. They are usually classified into *image space algorithms*, which work with the projected images of the objects, and *object space algorithms*, which work with the objects themselves.

Image space algorithms typically look for every pixel on the screen at all the objects whose projected image covers the pixel; the one that is closest to the view point is displayed. A notable example of such a method is the *depth-buffer algorithm*, also called *z-buffer algorithm*. This algorithm works as follows. It processes each object in turn, maintaining a buffer that stores for every pixel the (depth of the) currently visible object. To process a new object, one tests for every pixel that lies in the projected image of the object whether the new object is closer to the view point (at that pixel) than the currently visible object. If this is the case, then the buffer is updated. The main advantage of this method is that it is easily implemented in hardware. Therefore the method is—despite its brute-force approach—fast in practice and often used.

An interesting variation on this method is the *depth sorting algorithm*. This method eliminates the test between the processed object and the currently visible object in the depth-buffer algorithm. Moreover, the algorithm does not need a buffer for maintaining the z-values for each pixel. It works in two phases. In the first phase of the algorithm the objects are sorted according to their distance from the view point: if an object A is (partially) obscured by object B then A comes before B in the ordering. Such an order on the objects is called a depth order. Note that a depth order on the objects does not always exist, since there can be *cyclic overlap* among the objects. See, for example, Figure 1.3. In such cases the cycles have to be broken by cutting the objects into smaller pieces. This first phase works in object space. Efficient algorithms for computing depth orders are presented in Part C of this book. The second phase is similar to the depth buffer method, i.e. the objects are drawn

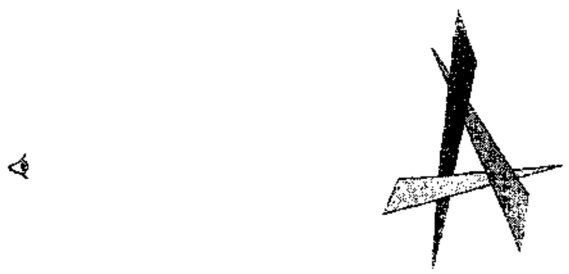

Figure 1.3: Three triangles with cyclic overlap.

one by one onto the screen. Because the order in which the objects are processed is a depth order—that is, objects in the back are processed earlier than objects in the front—none of the objects already processed can obscure a new object. Hence, all pixels in the projected image should get the color of the new object and no depth comparison is needed. Note that the second phase of the depth order algorithm is similar to the way an artist makes a painting: first the background colors are painted and later the objects are painted 'on top of this'. Thus the algorithm is sometimes called the *painter's algorithm.*

Another way of computing which object is visible at a certain pixel is to 'shoot a ray from the view point through the pixel'. The first object that is hit is the one that is visible at that pixel. See Figure 1.4. Algorithms following this approach are

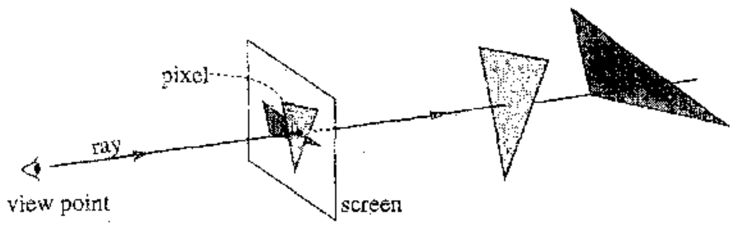

Figure 1.4: Hidden surface removal using ray tracing.

called *ray tracing algorithms.* Glassner [61] discusses this technique in great detail. To speed up the tracing process, the objects are stored in a data structure that allows one to determine the first object that is hit without looking at all the objects. The problem of designing efficient data structures for this task is called the *ray shooting problem* in computational geometry. It is the topic of Part B of this book.

Obtaining realistic pictures of a scene not only involves computing which parts of the objects can be seen from the view point, but also computing shading information about the light intensity that reflects from an object. The ray tracing algorithm can be extended to provide this information. The idea of the method is as follows. By shooting a so-called *primary ray* from the view point through each pixel, one can determine which object, let's call it A, is visible at that pixel. Let p be the point where the primary ray through the pixel hits A. To compute the shading information we shoot extra rays, called a *shadow feelers*, from point p into the direction of each light source in the scene. If a shadow feeler does not hit any object before it reaches the light source, then we know that p is lit by the light source. Otherwise p will be in shadow. It is, however, still possible that p is lit indirectly by the light source, by reflections via other objects. Such indirect illumination effects can also be computed, by shooting even more rays. It is beyond the scope of this book to discuss the ray tracing method any further, but it should be clear that the method requires many rays to be traced. Hence, it is extremely important to be able to compute the first object hit by a ray efficiently. The design of data structures for this task is the topic of Part B of this book.

The pixels on the screen can be grouped into regions where the same object is visible. The image space methods discussed above compute the view of a scene pixel by pixel. This means that the 'structure' of the view is lost. Object space algorithms compute a combinatorial representation of the structure of the view. Let us be more precise about this. The view of a scene is a subdivision of the viewing plane into maximal connected regions in each of which (some portion of) a single object can be seen, or no object is seen. In Figure 1.4, for example, this subdivision consists of four regions; the light triangle is visible in one of them, the dark triangle is visible in two other regions, and no object is seen in the fourth, white region. Object space algorithms compute this subdivision—which is called the *visibility map* of the given set of objects—as a collection of (polygonal) faces. In other words, object space algorithms compute exactly which parts of each object are visible. After that the visible parts can be projected and displayed without difficulty. Part D is devoted to the computation of these maps.

In general, object space hidden surface removal methods tend to be slower than image space methods such as the z-buffer algorithm, because they cannot be implemented in hardware very well. However, object space algorithms have certain advantages over image space methods. Suppose that we want to display the hidden lines in a scene dashed, instead of making them invisible. Object space algorithms compute exactly which parts of each line are visible and which parts are not, thus making it easy to display the invisible part dashed. Image space algorithms do not provide the information that is necessary to achieve this. Another weak point of image space algorithms comes up when one wants to print the view of a scene on paper, instead of displaying it on the screen of a computer terminal. When hidden surface removal has been done in image space, the only thing we can do is to plot every pixel separately. But this method fails to take advantage of the fact that the resolution (that is, the number of pixels per square inch) of modern laser printers is much higher than the resolution of computer screens. If hidden surface removal has been performed in object space then the visibility map can be processed directly, resulting in a picture of higher quality. A third advantage of object space algorithms is that they can be used to compute shadows in a scene. This follows from the fact that a point on an object is lit by a light source if and only if the ray from the light source to that point does not intersect any other object on its way. In other words, the part of a scene that is lit by a light source is exactly the same as what can be seen from the light source. Thus we can use an object space hidden surface removal algorithm to compute which parts of the objects are lit by the light source. Image space solutions such as the z-buffer method do not have this possibility: we can apply a z-buffer-like algorithm to compute the 'view' from the light source, but the problem is that the pixels with respect to the viewpoint do not coincide with the 'pixels' with respect to the light source. Notice that the shadows in a scene do not change when the view point moves. (Although what is visible of a shadow can change, of course.) Hence, the combinatorial representation of the shadow—which is exactly what an object-space hidden surface removal algorithm can compute—can be used for every view point. If shadow calculation is performed using the ray tracing algorithm described above, then everything has to be computed anew for each view point.

Before we give an overview of the contents of this book, let us spend a few words on the status of our work. This is a book about problems in computational geometry that arise in computer graphics, not a book about computer graphics. Hence, we give a theoretical treatment of these problems: we derive exact time bounds for our algorithms that will show that our algorithms perform better (in theory) than previously known algorithms. On the other hand, we have not implemented any of our methods. Thus the applicability in practice has not been established yet. Indeed, some of our data structures probably will not perform well in practice if they are implemented exactly as described here. Clearly, this is a topic of future study. In any case, a theoretical study such as undertaken here provides us with a deeper understanding of the important issues involved in the problems, which can be useful to obtain good practical solutions.

The book consists of four parts.

Part A, of which this introduction is the first chapter, introduces the reader to the problems that are studied and to the techniques that we use to solve them. Actually, this part might be interesting not only to people who want to read this book, but to anyone who wants to know more about some of the (recent) techniques in computational geometry.

Part B concerns itself with the ray shooting problem: Preprocess a set of polyhedra in three-dimensional space into a data structure, such that the first polyhedron that is hit by a query ray can be computed efficiently. We develop new, efficient data structures for various settings of this problem. In particular, we distinguish the case where the origin of the rays is fixed, the case where the direction of the rays is fixed, and the general case where there are no restrictions on the ray. For each setting, we study a number of different scenes: scenes consisting of axis-parallel polyhedra (whose edges are parallel to the x-axis, the y-axis, or the z-axis), scenes consisting of c-oriented polyhedra (whose edges are parallel to c different axes), and scenes consisting of arbitrary polyhedra.

In Part C we study the problem of computing depth orders: Given a set of polygons, sort them according to their distance to the view point. We start by studying the problem in two-dimensional space. This is useful, because many three-dimensional scenes—in particular certain types of landscapes, which are called *polyhedral terrains*—can be treated as being two-dimensional. We present a data structure such that a depth order for a given view point can be calculated efficiently. We also study three-dimensional scenes; here we present the first algorithm to compute a depth order for a set of polygons in three-dimensional space that achieves a subquadratic running time.

Finally, in Part D we present an object space algorithm for hidden surface removal in general polyhedral scenes. The running time of our algorithm is *output-sensitive*, that is, the running time decreases as the complexity of the visibility map decreases. Previously, such algorithms were known only for special cases, where the objects in the scene satisfy the—not very realistic—constraint that a depth order on the objects exists and is known.