Algorithms for Fat Objects: Decompositions and Applications

Christopher Miles Gray

Algorithms for Fat Objects: Decompositions and Applications

PROEFSCHRIFT

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. M.T. de Berg

Promotor: prof.dr. M.T. de Berg
                 faculteit Wiskunde & Informatics
                 Technische Universiteit Eindhoven

Kerncommissie:
prof.dr. B. Aronov (Polytechnic University)
prof.dr. P.K. Bose (Carleton University)
dr. B. Speckmann (Eindhoven University of Technology)
prof.dr. G. Woeginger (Eindhoven University of Technology)

Cover Design: Abby Normal
Printing: Eindhoven University Press

Contents

i

Preface

I can still remember the sequence of events that led me to write this thesis: I was sitting in my office at the University of British Columbia and chatting online with my friend Chris Wu[1]. He mentioned that he had heard of a Ph.D. position that was open at the Technical University of Eindhoven with Mark de Berg. I knew of Mark from the book he had written on computational geometry, but Eindhoven was new to me. Still, the position seemed like a good one, so I made up a CV and sent it along. I heard back fairly quickly that I had been accepted, and I made the decision to come to Eindhoven after a few days of thinking.

I have never regretted that decision. The people in Eindhoven have been extremely kind and it has been a wonderful environment in which to do research.

I started working right away on topics related to my thesis—a bit of a surprise after seeing the normal procedure at North American universities, which is to do a lot of reading for the first two years before deciding on a topic. Within the first few months, I had results that are included in this thesis.

Since then, in collaboration with many coauthors, I have been fortunate enough to have written quite a few papers that have been published in conferences and scientific journals. Many of the results from those papers are included in this thesis.

I must thank many people who have made my time in Eindhoven the enjoyable time that it has been. First, my advisor Mark de Berg. He has been a wonderful teacher. He has directed me to many good problems, and then has been extremely patient as he tries to help me write down a clear and understandable solution. My work has benefited greatly from our collaboration.

---

[1]Incidentally, Chris also convinced me to take my first course in computational geometry as well as to apply to UBC for the Master's program. He has had a strangely disproportionate influence on my life up to now.

Introduction

1.1 Computational geometry

Computational geometry is the branch of theoretical computer science that deals with algorithms and data structures for geometric objects. The most basic geometric objects include points, lines, polygons, and polyhedra. Computational geometry has applications in many areas of computer science, including computer graphics, robotics, and geographic information systems.

Perhaps a sample computational-geometry problem would help give a more clear view of what computational geometry is. The problem of finding the convex hull of a set of n input points is a convenient such example. The *convex hull* of a set of points is the convex polygon with the smallest area that contains all the points—see Figure 1.1(a). (A convex set S is one where any line segment between two points p and q in S is completely inside S).

A naïve algorithm for finding the convex hull, known as the *gift-wrapping algorithm* [17], is as follows. Find the lowest point p—we assume for simplicity that this is unique—of the input (this is guaranteed to be a vertex of the convex hull) and let ℓ be an imaginary horizontal ray starting at p, directed rightwards. Then find the point q—again, we assume that this is unique—where the angle between $\overline{pq}$ and ℓ is the smallest. Thus, conceptually, we rotate ℓ counterclockwise around p until we hit another point q—see Figure 1.1(b). Add the edge $\overline{pq}$ to the convex hull, let ℓ be the ray contained in $\overline{pq}$ that starts at q and let p point to q. Then repeat this procedure until p is the lowest point of the input again.
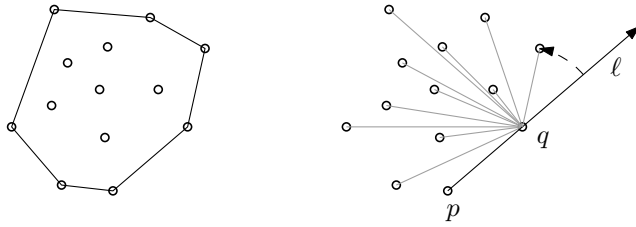
Figure 1.1 (a) A convex hull. (b) The gift-wrapping algorithm after one edge has been added.

This example shows how we can construct the convex hull as a sequence of edges that are themselves made out of pairs of input vertices. Since the algorithm looks at every vertex of the input for every convex-hull vertex that it finds, and since every vertex can be on the convex hull, the gift-wrapping algorithm is clearly a $\Theta(n^2)$ algorithm, meaning that its worst-case running time grows quadratically with its input size. Can we do better?

It turns out we can—there are algorithms that use more advanced techniques like divide-and-conquer or sorting that take $\Theta(n \log n)$ time [74]. If we disregard the constants hidden in the Θ-notation, this means that these more advanced algorithms would take about ten thousand steps versus about a million for the gift-wrapping algorithm on an input of one thousand points. There is a lower bound of $\Omega(n \log n)$ on finding the convex hull of n vertices, so we can not do any better than these more advanced algorithms in theory.

So why do we remember the gift-wrapping algorithm? Is it simply a relic that can be discarded? If one implements and runs the gift-wrapping algorithm and compares it head-to-head with a more advanced algorithm, a surprising event can occur. On some inputs, the gift-wrapping algorithm actually runs faster. How can this happen?

The problem was in our analysis of the gift-wrapping algorithm. It was not incorrect: in the worst case, the algorithm can take $\Omega(n^2)$ time. However, this worst case only happens if there are $\Omega(n)$ points on the convex hull. If there is only a constant number of points on the convex hull, then the algorithm runs in $O(n)$ time. This disparity leads us to look at the time complexity in terms of n and a different parameter h—the number of points on the convex hull. The time complexity of the gift-wrapping algorithm when using these parameters has been shown to be $\Theta(nh)$. It has been shown, in fact, that the expected number of points on the convex hull is $O(\log n)$ for points spread uniformly at random inside a convex polygon [45]. Hence, on such inputs the gift-wrapping algorithm has an expected running time of $O(n \log n)$.

1.2 Realistic input models

The previous example illustrates a problem with the worst-case analysis that we employ in theoretical computer science. That is, we concentrate (by definition) on the worst case that the input can take, no matter how unlikely it is.

A number of solutions to this problem have been proposed, including looking at the output complexity, as illustrated above, and looking at the expected complexity of the algorithms on random inputs. The solution that we explore in this thesis looks at the "geometric complexity" of the input.

(a)   (b)
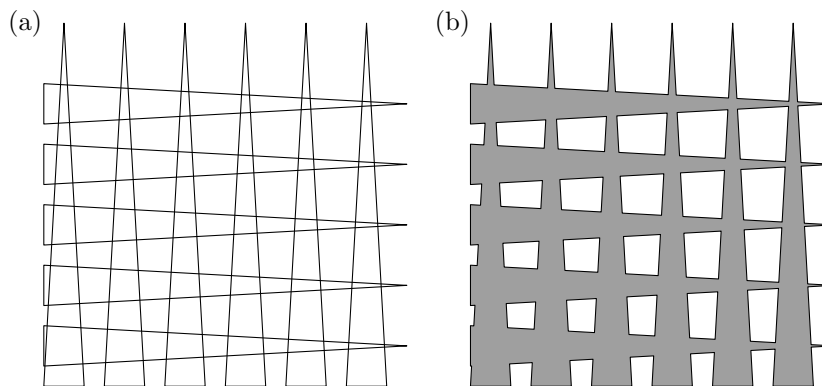
Figure 1.2   (a) n triangles. (b) Their union.

As an example, it is easy to see that n triangles in the plane can have a union with complexity $\Theta(n^2)$—see Figure 1.2. However, we can also see that these triangles must have an angle that is very small—in fact, to make the grid-like example of Figure 1.2, one needs angles whose size depends on $1/n$. Thus the larger n is, the smaller angles are needed. If we restrict the smallest angle of any triangle to be larger than a constant α, though, then it has been shown that the complexity of the union drops to $O(n \log \log n)$ (where the constant in the O-notation depends on α) [65]—see Figure 1.3.

In most realistic situations, the angles of input triangles do not depend on the size of the input. The model where the input triangles are required to have a constant minimum angle is an example of a *realistic input model* [41] (having to do with the *fatness* of the input).

There are two categories of realistic input models: those that make assumptions about the *shape* of the individual input objects and those that make assumptions about the *distribution* of the input objects. One example of a realistic input model which makes assumptions about the shape of the objects was just given—triangles are said to be α-*fat* if their minimum angle is bounded from below by a constant α. An example of a realistic input model that makes assumptions about the distribution of the input is the λ-*low-density* model. Here, we assume that the number of "large" objects in a "small" region is bounded by a

3
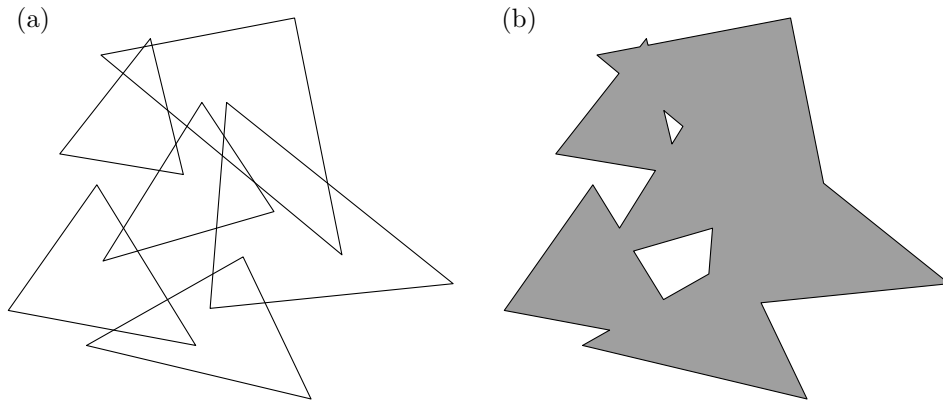
Figure 1.3 (a) n fat triangles. (b) Their union.

constant λ. More formally, if we let $\mathrm{size}(o)$ denote some measure of size of an object o, then a low-density scene is one where the number of objects o intersecting a region R where $\mathrm{size}(o) > \mathrm{size}(R)$ is at most some constant λ for all regions R.

It is often the case that realistic input models that make assumptions only on the distribution of the input are more general than those that make assumptions about the shape of the input. Our example realistic input models are a case in point: any scene consisting of disjoint fat triangles in $\mathbb{R}^2$ is low-density, but not all low-density scenes consist only of fat objects. A hierarchy of such relations has been previously given [41].

As suggested above, one primary motivation for using realistic input models is the notion that they do a better job at predicting the performance of algorithms in reality. Furthermore, algorithms for realistic input are often simpler than algorithms that must be tuned to arbitrary worst-case examples.

One caveat about working with realistic input models: we must be careful to show the dependence on the constants associated with the models that could be hidden in the O-notation in the analysis. This is because any object could be called α-fat and any collection of objects could be called λ-low density if α and λ are chosen suitably small (in the case of α) or large (in the case of λ). If, on the other hand, we show the dependence in the analysis, then it is clear that at some value of the constant the result becomes less useful.

1.2.1   Previous work

The past work on realistic input models has focused on four main areas: union complexity, motion planning, point location and range searching, and certain computer-graphics problems. More recently, there have been some new results related to realistic terrains.

Union complexity. The complexity of the union of a set of objects is a combinatorial property that is interesting from an algorithmic point of view because it influences the running times of some algorithms. One area where it is especially important is in robotics and motion planning. This is because the first step of the standard technique for determining whether a robot can move between two points is to shrink the robot down to a point, expanding the obstacles accordingly. The algorithm then determines whether there are any paths that can go from the starting point to the target. The computational complexity of this technique depends in large part on the complexity of the union of the expanded obstacles.

The union complexity of n fat triangles was first shown to be $O(n \log \log n)$ by Matoušek *et al.* [65] and the dependence on the fatness constant was later improved by Pach and Tardos [76]. In fact, since convex fat polygons of complexity m can be covered by $O(m)$ fat triangles (as we show in a later chapter), the same is true for this class of objects. Furthermore, under a different definition of fatness, Van Kreveld showed [98] that non-convex polygons have the same property.

For objects that are not convex and that can have curved edges, De Berg showed that the union complexity is also close to linear [30]. For locally-γ-fat objects (and thus (α, β)-covered objects)—defined in Section 1.4—whose curved edges can intersect at most s times, the union complexity is $O(\lambda_{s+2}(n) \log^2 n)$, where $\lambda_s(n)$ represents the length of an (n, s) Davenport-Schinzel sequence. Such a sequence has a length that is near-linear in n for any constant s [87].

It has recently been shown that the union of fat tetrahedra in $\mathbb{R}^3$ is $O(n^{2+\varepsilon})$ [51]. There has been some work done under other definitions of fatness as well: Aronov *et al.* [11] showed that the complexity of the union of so-called κ-round objects is $O(n^{2+\varepsilon})$ in three dimensions and $O(n^{3+\varepsilon})$ in four.

Robotics and motion planning. The application of realistic input models to motion planning has been quite successful. For example, when a robot has f degrees of freedom, the free space (that is, the set of places into which the robot can move without colliding with an obstacle) has complexity $\Theta(n^f)$. This implies that any exact solution to the motion planning problem has time complexity $\Omega(n^f)$. Currently, the algorithm with the best time complexity for motion planning has time complexity $O(n^f \log n)$ [16]. However, when the obstacles form a low-density scene and the robot is not much larger than the obstacles, the complexity of the free space is $O(n)$ [97]. This has enabled the development of motion-planning algorithms with running times that are nearly linear given these realistic input assumptions [96].

Point location and range searching. There has been some research into data structures for point location and range searching in a set S of disjoint fat objects. In the first problem, one wishes to find the specific object from S containing a query point. In the second, the problem is to find all objects from S intersecting a query range. That is, we wish to report

all objects that intersect some specific part of space. These two problems are related and often treated in tandem.

Point location has been well studied in two dimensions, while it remains essentially open in higher dimensions. A common data structure for point location in two dimensions, known as the trapezoidal map, is given in the book by De Berg *et al.* [42]. In arrangements of hyperplanes in d dimensions, Chazelle and Friedman give a data structure [23] that can answer a point-location query in $O(\log n)$ time using $O(n^d)$ space.

Range searching is another well-studied problem. For arbitrary input, the two best-known data structures are partition trees and cutting trees. Each has a trade-off: partition trees use linear space, but queries take $O(n^{1-1/d+\varepsilon})$ time [66], while cutting trees have $O(\log^d n)$ query time, but take $O(n^{d+\varepsilon})$ storage [20]. It is also possible to trade storage for query time by combining the two types of trees: for any $n \leq m \leq n^d$, there exists a data structure with $O(m^{1+\varepsilon})$ storage and $O(n^\varepsilon/m^{1+d})$ query time.

Overmars and Van der Stappen first showed [75] that point-location and range-searching queries can be handled efficiently when the input is fat. They presented a data structure that supports point-location and range-searching queries in $O(\log^{d-1} n)$ time that requires $O(n \log^{d-1} n)$ storage after $O(n \log^{d-1} n \log \log n)$ preprocessing. However, the range-searching portion of this result requires the range to be not too much larger than the objects being queried. Subsequently, the same bounds for query time and storage space were obtained for low-density input at the expense of a small increase in preprocessing time [85]. This was further improved by De Berg, who gave [29] a linear-sized data structure with logarithmic query time for *uncluttered* scenes (another realistic input model on the distribution of the input that generalizes low density).

Most recently, object BAR-trees were employed to perform approximate range queries on low-density input in approximately $O(\log n + k)$ time using linear space [40].

Computer graphics. Some of the problems related to computer graphics that have been studied in the context of realistic input models are hidden surface removal, ray shooting, and the computation of depth orders. We study these problems in later chapters and give detailed overviews of the related work in the next section.

Realistic terrains. A *polyhedral terrain* (also known as a triangulated irregular network) is a 2.5-dimensional representation of a portion of a surface. The most common surface that is represented by a terrain is the Earth. A terrain is modeled as a planar triangulation of a set of two-dimensional points. That is, it is a tiling of the convex hull of the points by triangles with the condition that every point is the vertex of at least one triangle. Each of the points has additional height information, and it is assumed that the elevation of any point inside a triangle t is given by interpolating the heights of the vertices of t.

Realistic terrains are a newer area of research related to realistic input models that are inspired by geographic information systems. Here, a few restrictions are placed on the terrain:

- The triangles of the terrain are fat.

- The triangles are not too steep.

- The triangles are all nearly the same size.

- The projection of the terrain onto the xy-plane is a rectangle that is nearly a square.

Certain properties of these terrains, such as the complexity of a geodesic bisector between two points, have been shown to be lower than in general terrains [71]. Also, some experiments have been done that show that these assumptions are in fact realistic [70]. In addition, there has been some work done on finding the watersheds of such terrains [32] and on computing the overlay of maps of such terrains in a manner that attempts to minimize the number of disk accesses [37].

1.3   Overview of this thesis

In the remainder of this chapter, we give a short outline of the chapters to follow. We also mention some of the relevant related work. We begin with two chapters related to the decomposition of fat polygons and polyhedra, which we follow with three chapters related to new algorithms for problems related to computer graphics.

Triangulating fat polygons.   In Chapter 2, we examine triangulation of a polygon—probably the most-used decomposition in computational geometry. We examine the problem in the context of fat objects. Connections between the running time of a triangulation algorithm and the shape complexity of the input polygon have been studied before. For example, it has been shown that monotone polygons [92], star-shaped polygons [84], and edge-visible polygons [93] can all be triangulated in linear time by fairly simple algorithms. Other measures of shape complexity studied include the number of reflex vertices [57] or the sinuosity [27] of the polygon.

We give a simple algorithm for computing the triangulation in time proportional to the complexity of the polygon and the number of guards that are necessary to "see" the entire boundary of the polygon. We also show that a certain type of fat polygons needs a constant number of guards—meaning that our algorithm is a linear-time algorithm for these polygons.

As of this writing, portions of Chapter 2 are to appear at the *20th Canadian Conference on Computational Geometry*.

Decomposing non-convex fat polyhedra.   In Chapter 3, we look at decompositions of non-convex fat polyhedra in three dimensions. Here, we attempt to find decompositions where the number of pieces is not too high. We show in a few cases that this can be done,

and we prove that it can not be done in most cases. This is, as far as we know, the first investigation of the possibilities of decomposition for the various types of fat polyhedra in three dimensions. In two dimensions, Van Kreveld showed [98] that non-convex polygons can be covered by fat triangles.

A preliminary version of Chapter 3 appeared at the *24th European Workshop on Computational Geometry*, and the full paper has been invited to the special issue of *Computational Geometry: Theory and Applications* that accompanies that workshop. As of this writing, the paper is to appear at the *16th European Symposium on Algorithms*.

Ray shooting and simplex range searching. In Chapter 4, we look at the problem of *ray-shooting* amidst fat objects from two perspectives. This is the problem of preprocessing data into a data structure that can answer which object is first hit by a query ray in a given direction from a given point. In the first part of the chapter we fix the direction, while in the second part of the chapter the direction is allowed to be arbitrary. We then conclude with a data structure that reports the objects intersected by a query simplex that works in a similar manner to the data structure for ray shooting in arbitrary directions.

Data structures for vertical ray-shooting queries among sets of arbitrary disjoint triangles in $\mathbb{R}^3$ have rather high storage requirements. When $O(\log n)$ query time is desired, the best-known data structure needs $O(n^2)$ space [28]. Space can be traded for query time: for any m satisfying $n \leq m \leq n^2$, a data structure can be constructed that uses $O(m^{1+\varepsilon})$ space and allows vertical-ray-shooting queries that take $O(n^{1+\varepsilon}/m^{1/2})$ time [28].

Given the prominence of the ray-shooting problem in computational geometry it is not surprising that ray shooting has already been studied from the perspective of realistic input models. In particular, the vertical-ray-shooting problem has been studied for fat convex polyhedra. For this case Katz [58] presented a data structure that uses $O(n \log^3 n)$ storage and has $O(\log^4 n)$ query time. Using the techniques of Efrat *et al.* [47] it is possible to improve the storage bound to $O(n \log^2 n)$ and the query time to $O(\log^3 n)$ [59]. Recently De Berg [31] presented a structure with $O(\log^2 n)$ query time; his structure uses $O(n \log^3 n(\log \log n)^2)$ storage.

Similarly, in the case of ray-shooting in arbitrary directions, the results achieved for non-fat objects require a lot of storage. If the input consists of n arbitrary triangles, the best known structures with $O(\log n)$ query time use $O(n^{4+\epsilon})$ storage [28, 78], whereas the best structures with near-linear storage have roughly $O(n^{3/4})$ query time [7]. More generally, for any m with $n < m < n^4$, one can obtain $O((n/m^{1/4}) \log n)$ query time using $O(m^{1+\varepsilon})$ storage [7]. Better results have been obtained for several special cases. When the set $\mathcal{P}$ is a collection of n axis-parallel boxes, one can achieve $O(\log n)$ query time with a structure using $O(n^{2+\varepsilon})$ storage [28]. Again, a trade-off between query time and storage is possible: with $O(m^{1+\varepsilon})$ storage, for any m with $n < m < n^2$, one can achieve $O((n/\sqrt{m}) \log n)$ query time. If $\mathcal{P}$ is a set of n balls, then it is possible to obtain $O(n^{2/3})$ query time with $O(n^{1+\varepsilon})$ storage [90], or $O(n^\varepsilon)$ query time with $O(n^{3+\varepsilon})$ storage [72].

When the input is fat, the results are somewhat better. For the case of *horizontal* fat

triangles, there is a structure that uses $O(n^{2+\varepsilon})$ storage and has $O(\log n)$ query time [28], but the restriction to horizontal triangles is quite severe. Another related result is by Mitchell *et al.* [69]. In their solution, the amount of storage depends on the so-called *simple-cover complexity* of the scene, and the query time depends on the simple-cover complexity of the query ray. Unfortunately the simple-cover complexity of the ray—and, hence, the worst-case query time—can be $\Theta(n)$ for fat objects. In fact, this can happen even when the input is a set of cubes. The first (and so far only, as far as we know) result that works for arbitrary rays and rather arbitrary fat objects was recently obtained by Sharir and Shaul [89]. They present a data structure for ray shooting in a collection of fat triangles that has $O(n^{2/3+\varepsilon})$ query time and uses $O(n^{1+\varepsilon})$ storage. Curiously, their method does not improve the known bounds at the other end of the query-time–storage spectrum, so for logarithmic-time queries the best known storage bound is still $O(n^{4+\varepsilon})$.

We present a new data structure for answering vertical ray-shooting queries as well as a data structure for answering ray-shooting queries for rays with arbitrary direction. Both structures improve the best known results on these problems. Finally, we use ideas from the second data structure to make a data structure for simplex range searching.

Portions of Chapter 4 appeared at the *22nd European Workshop on Computational Geometry*, where the full paper was also invited to the special issue of *Computational Geometry: Theory and Applications* that accompanies that workshop [9]. The paper also appeared at the *22nd Symposium on Computational Geometry* [8].

Depth order. Another problem that is studied in the field of computer graphics is the *depth-order* problem. We study it in Chapter 5 in the computational-geometry context. This is the problem of finding an ordering of the objects in the scene from "top" to "bottom", where one object is above the other if they share a point in the projection to the xy-plane and the first object has a higher z-value at that point.

The depth-order problem for arbitrary sets of triangles in 3-space does not seem to admit a near-linear solution; the best known algorithm runs in $O(n^{4/3+\varepsilon})$ time [39]. This has led researchers to also study this problem for fat objects. Agarwal *et al.* [5] gave an algorithm for computing the depth order of a set of triangles whose projections onto the xy-plane are fat; their algorithm runs in $O(n \log^5 n)$ time. However, their algorithm cannot detect cycles—when there are cycles it reports an incorrect order. A subsequent result by Katz [58] produced an algorithm that runs in $O(n \log^5 n)$ time and that can detect cycles. In this case, one of the restrictions placed on the input is that the overlap of the objects in the projection is not too small. Thus, the constant of proportionality depends on the minimum overlap of the projections of the objects that do overlap. If there is a pair of objects whose projections barely overlap, then the running time of the algorithm increases greatly. One advantage that this algorithm has is that it can deal with convex curved objects.

We give an algorithm for finding the depth order of a group of fat objects and an algorithm for verifying if a depth order of a group of fat objects is correct. The latter algorithm is useful because the former can return an incorrect order if the objects do not have a depth

order (this can happen if the above/below relationship has a cycle in it). The first algorithm improves on the results previously known for fat objects; the second is the first algorithm for verifying depth orders of fat objects.

Portions of Chapter 5 appeared at the *17th ACM-SIAM Symposium on Discrete Algorithms* [34]. The full version of the paper has appeared in the *SIAM Journal on Computing* [36].

Hidden-surface removal. The final problem that we study is the *hidden-surface removal* problem. In this problem, we wish to find and report the visible portions of a scene from a given viewpoint—this is called the *visibility map*. The main difficulty in this problem is to find an algorithm whose running time depends in part on the complexity of the output. For example, if all but one of the objects in the input scene are hidden behind one large object, then our algorithm should have a faster running time than if all of the objects are visible and have borders that overlap. We give such an algorithm—called an *output-sensitive* algorithm—in Chapter 6.

The first output-sensitive algorithms for computing visibility maps only worked for polygons parallel to the viewing plane or for the slightly more general case that a depth order on the objects exists and is given [15, 53, 54, 80, 81, 88]. Unfortunately a depth order need not exist since there can be cyclic overlap among the objects. De Berg and Overmars [38] (see also [28]) developed a method to obtain an output-sensitive algorithm that does not need a depth order. When applied to axis-parallel boxes (or, more generally, c-oriented polyhedra) it runs in $O((n + k) \log n)$ time [38] and when applied to arbitrary triangles it runs in $O(n^{1+\varepsilon} + n^{2/3+\varepsilon} k^{2/3})$ time [6]. Unfortunately, the running time for the algorithm when applied to arbitrary triangles is not near-linear in n—the complexity of the input—and k—the complexity of the output; for example, when $k = n$ the running time is $O(n^{4/3+\varepsilon})$. For general curved objects no output-sensitive algorithm is known, not even when a depth order exists and is given.

Hidden-surface removal has also been studied for fat objects: Katz *et al.* [60] gave an algorithm with running time $O((U(n) + k) \log^2 n)$, where $U(m)$ denotes the maximum complexity of the union of the projection onto the viewing plane of any subset of m objects. Since $U(m) = O(m \log \log m)$ for fat polyhedra [76] and $U(m) = O(\lambda_{s+2}(m) \log^2 m)$ for fat curved objects [30], their algorithm is near-linear in n and k. However, the algorithm only works if a depth order exists and is given.

We give an algorithm for hidden-surface removal that does not need a depth order and whose running time is still near-linear in n and k.

Portions of Chapter 6 appeared at the *10th International Workshop on Algorithms and Data Structures* [35] and the full paper was also invited to the special issue of *Computational Geometry: Theory and Applications* that accompanies that workshop.

Conclusions. We end the thesis with some conclusions and we state some open problems in Chapter 7.

1.4 Definitions and basic techniques

Many realistic input models (and measures of fatness) have been proposed. In the next few paragraphs, we define those that we use most in this thesis and discuss some techniques that we feel are important to know about when dealing with realistic input.

β-**fat objects.** The best-known and most widely used of the realistic input models is β-*fatness*. This is the model of fatness that we employ in this thesis, unless otherwise noted. It is defined as follows.

Definition 1.1 *Let β be a constant, with $0 < \beta \leq 1$. An object o in $\mathbb{R}^d$ is defined to be β-fat if, for any ball b whose center lies in o and that does not fully contain o, we have $vol(b \cap o) \geq \beta \cdot vol(b)$.*

There have been other definitions of fatness proposed (such as the one given in Section 1.2, restricting the minimum angle of triangles), but when the input is convex, they are all equivalent up to constant factors.

Locally-γ-fat objects. When the input is not convex, defining fatness in such a way that the objects satisfy the intuitive definition of fatness is trickier. Many of the results stated for fat objects break completely under Definition 1.1 when the input is not convex—for example, the union complexity of two non-convex β-fat objects can be $\Omega(n^2)$ as can be seen in Figure 1.4(b). (In fact, n constant-complexity non-convex β-fat objects can also have a union complexity of $\Omega(n^2)$, but the example is more complicated [30].) We use two definitions of fatness for non-convex objects in this thesis that satisfy the intuitive definition better than Definition 1.1. The first is of locally-γ-fat objects. See Figure 1.4(a).

(a) (b)
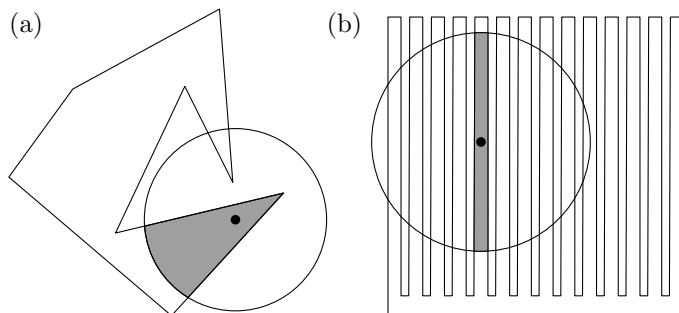
Figure 1.4 (a) A locally-fat polygon. Note that only the part of the intersection containing the center of the circle is counted. (b) An object that is approximately $(1/4)$-fat, but not locally-$(1/4)$-fat.

11

Definition 1.2 *For an object o and a ball b, define $b \sqcap o$ to be the connected component of $b \cap o$ that contains the center of b. Let γ be a constant, with $0 < \gamma \le 1$. An object o in $\mathbb{R}^d$ is defined to be* locally-γ-fat *if, for any ball b whose center lies in o and that does not fully contain o, we have $vol(b \sqcap o) \ge \gamma \cdot vol(b)$.*

(α, β)**-covered objects.** Definition 1.2 is a small modification of Definition 1.1—we simply replace $\cap$ by $\sqcap$. The second definition that we use shares less with Definition 1.1. It is illustrated in Figure 1.5.
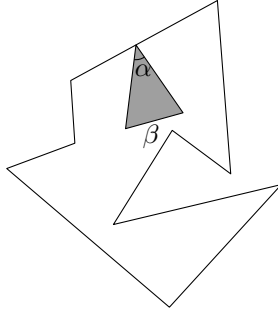
Figure 1.5 An (α, β)-covered polygon with diameter 1.

Definition 1.3 *Let P be a polyhedron in $\mathbb{R}^d$ and α and β be two constants with $0 < \alpha \le 1$ and $0 < \beta \le 1$. A* good simplex *is a simplex that has fatness α (using Definition 1.1) and has smallest edge-length $\beta \cdot diam(P)$. P is (α, β)-covered if every point p on the boundary of P admits a good simplex that has one vertex at p and stays completely inside P.*

Definition 1.3 is a generalization to higher dimensions of the (α, β)-covered polygons proposed by Efrat [46]. As observed by De Berg [30] when he introduced the class of locally-γ-fat polygons, the class of locally-γ-fat objects is strictly more general than the class of (α, β)-covered objects: any object that is (α, β)-covered for some constants α and β is also locally-γ-fat for some constant γ depending on α and β, but the reverse is not true.

Low-density scenes. Another realistic input model assumes that the input is *low density*. This means, essentially, that there can not be too many large objects intersecting a small space. The formal definition is given below. We define $\text{size}(o)$, the *size* of an object o, to be the radius of the smallest enclosing ball[1] of o.

Definition 1.4 *The* density *of a set S of objects is defined as the smallest number λ such that any ball b is intersected by at most λ objects $o \in S$ such that $\text{size}(o) \ge \text{size}(b)$.*

---

[1] It is also possible to use the diameter of o as the measure of its size. This leads to slightly different constants in the analysis, but has no asymptotic effect.
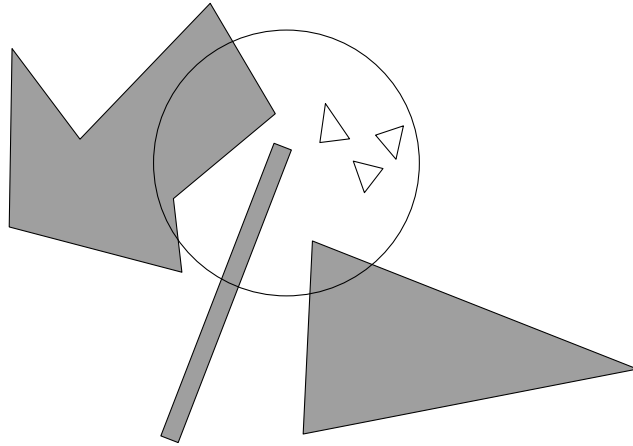
Figure 1.6 A low-density scene. Note that the small triangles are not counted, since they are not as large as the circle.

The following lemma relates the density of a set of disjoint objects to their fatness.

Lemma 1.5 (De Berg *et al.* [41]) *Any set of disjoint β-fat objects has density λ for some $\lambda = O(1/\beta)$.*

Below, we look at three techniques that are useful when dealing with realistic input.

Canonical directions. One simple but powerful tool often used when designing algorithms and combinatorial proofs for fat objects is a small set of canonical directions. It is difficult to define such a set in the absence of the context of a specific problem, so we first give an example.

We again restrict the input to triangles that have a minimum angle that is at least some constant α. Let $D = \{0, \alpha/2, \alpha, 3\alpha/2, \ldots\}$ be a set of directions with $|D| = \lceil 4\pi/\alpha \rceil$. Then at every vertex v of a triangle t, there must be at least one direction $\vec{d} \in D$ where a line segment placed at v in direction $\vec{d}$ stays in the interior t. It is important to note that the size of D is independent of the number of triangles in the input set; no matter how many triangles are input, if they are all α-fat, then $O(1/\alpha)$ directions suffice.

One application in which such a set of canonical directions is useful is the following. Let P be a set of n points. We wish to query a data structure on P with ranges that are fat triangles. The data structure should return all the points inside the range. This is known as *simplex range searching*. For arbitrarily skinny ranges, *cutting trees* have near-logarithmic query times and $O(n^{2+\varepsilon})$ storage requirements, and *partition trees* have near-linear storage requirements but have query times that are $O(n^{1/2+\varepsilon})$ [42].
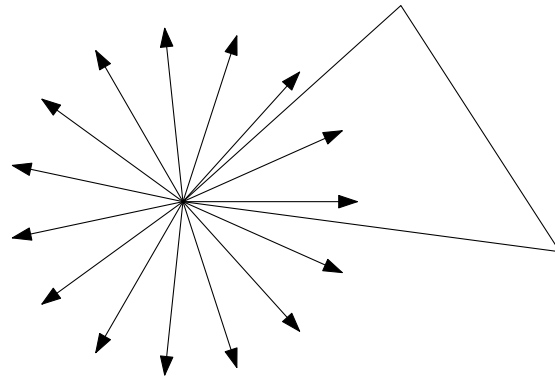
Figure 1.7 A set of canonical directions for $\alpha = 48°$.
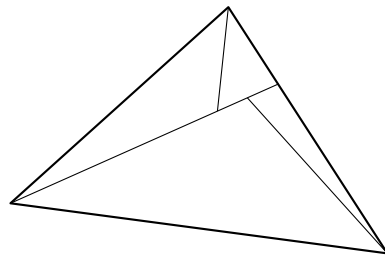
Figure 1.8 A fat triangle divided into triangles with two edges that have canonical directions. The canonical directions are those shown in Figure 1.7.

However, an α-fat triangle can be divided into four smaller triangles that each have two edges that have directions from D—see Figure 1.8. This allows us to design a more efficient data structure. Each range query with a triangle can be thought of as the intersection of three range queries with half-planes. When the direction of an edge of the triangle is known beforehand, such a half-plane query is simple: a balanced binary search tree will suffice.

Therefore, we can construct $O(1/\alpha^2)$ multi-level data structures—see [42] for a good introduction to multi-level data structures—with three levels. Each of the first two levels corresponds to a half-plane query data structure optimized for one of the canonical directions (that is, the balanced binary search tree). The final level of the data structure is a data structure by Chazelle *et al.* [26]. This is a slightly more complex data structure that uses $O(n)$ space and can answer half-space range queries in time $O(\log n + k)$, where k is the size of the output. Our data structure then has query time $O(\log^3 n + k)$, while using $O(n \log^2 n)$ space. In other words, its query time is approximately the same as the query time for cutting trees while its space requirement is about the same as that of partition trees.

In this example, the property that the canonical directions have is that a segment travelling in a direction from D away from a vertex stays inside the triangle. In later chapters, we use sets of canonical directions with more complicated properties, such as when we define towers in Chapter 3 and when we define witness edges in Chapter 5.

Guards. Another tool used when dealing with realistic input is a *guarding set*. The goal when creating a guarding set is to define a set of points that have the property that any *range* that does not contain one of the points must intersect a small number of the input objects. A range, in this context, is an element of a family R of shapes. An example family R could be the set of all squares in $\mathbb{R}^2$, and a range from that family would then be a specific square.

Given a set D of disjoint disks, we can build a guarding set by placing a guard at each corner of the axis-aligned bounding square of each disk in D as well as at the center of each disk in D. Using the same family of ranges R defined above—namely the set of all squares—any range r from R that contains no guard can intersect at most four disks from D. This property is useful for constructing data structures, such as binary space partitions, discussed below.
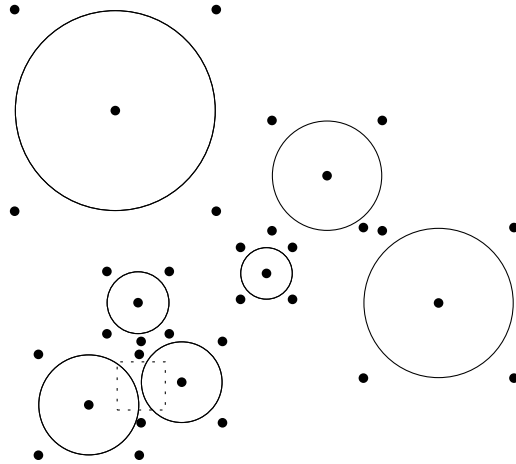
Figure 1.9 A set of circles with guards. No square can intersect more than four circles without containing a guard.

In contrast to the example above, where the size of the guarding set depends on n, in certain situations the size of the guarding set is a constant depending on the fatness constant of the input. In Chapter 4, for example, we create a constant-sized grid that guards against a family of ranges that consists of a subset of the input. However, in this thesis, guarding sets are most often used implicitly in the construction of binary space partitions, which we discuss next.

Binary space partitions.   Another technique used in computational geometry is the decomposition of space into cells. Generally the goal is to obtain a constant number of (fragments of) input objects in each cell. This is a widely used technique, and when the objects conform to a realistic input model, properties of the decompositions often improve.

One decomposition of space is known as the *binary space partition*, or *BSP*. BSPs are widely used in practice despite the fact that their use often does not lead to the best-known theoretical time bounds. However, their actual performance is often better than the theory predicts. One reason for this might be that the objects input to BSPs in practice tend to fit realistic input models.

The main idea behind a BSP is to recursively split space until the remaining subspaces each contain at most one fragment of an input object. This process can be modeled as a tree structure. A BSP is constructed as follows: first, a hyperplane (a line in two dimensions or a plane in three dimensions) h_1 splits space. Then two hyperplanes h_2 and h_3 split the parts of space on either side of h_1. Two hyperplanes then split the parts of space on either side of h_2 and two more hyperplanes split the parts of space on either side of h_3. This process continues until each part of space contains at most one piece of input.

The BSP tree structure is defined as follows: every fragment of an object is contained in some leaf. A node ν contains a splitting hyperplane h (the root node contains h_1) and has two BSP trees as children. The left child of ν is the BSP tree on the space above h and the right child of ν is the BSP tree on the space below h. See Figure 1.10.
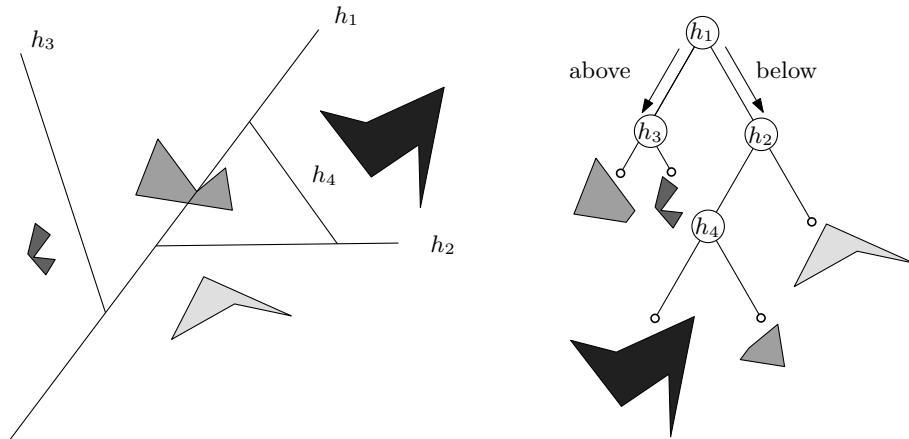
Figure 1.10   A BSP and its associated tree.

The *size* of a BSP is defined to be the number of fragments that are stored in the nodes of the BSP. It is generally desirable to have a BSP that has a small size. However, even for segments in $\mathbb{R}^2$, it is not always possible to obtain a linear-sized BSP, as Tóth has shown [91] that there are input configurations that imply a BSP of size $\Omega(n \log n / \log \log n)$. In $\mathbb{R}^3$, the situation is even worse: Paterson and Yao show [77] that binary space parti-

16

tions for disjoint triangles can be forced to have size $\Omega(n^2)$. This is too large for many applications, so BSPs were often ignored by the theoretical-computer-science community. However, when the input conforms to a realistic input model, the situation does not look so bad. First, De Berg designed [29] a BSP with linear size for low-density scenes. Then De Berg and Streppel designed [40] the *object BAR-tree*, which also has linear size for low-density scenes as well as a few other nice properties that we discuss below.

The object BAR-tree is an extension of the *balanced aspect-ratio tree*, or *BAR-tree*, introduced by Duncan *et al.* [44]. This is a BSP on points that has linear size (as do all BSPs on points, since points can not be split). The cells of the BAR-tree are fat, and the depth of a BAR-tree on n points is $O(\log n)$.

The object BAR-tree is constructed by surrounding each input object by a set of guards and then building a BAR-tree on the guards. As long as the input objects have low density, the tree has the same properties as a BAR-tree: linear size, fat cells, and logarithmic depth. These properties are quite useful, and we see examples of the use of the object BAR-tree in many chapters of this thesis.

Triangulating fat polygons

2.1 Introduction

Polyhedra and their planar equivalent, polygons, play an important role in many geometric problems. From an algorithmic point of view, however, general polygons and polyhedra are unwieldy to handle directly: many algorithms can only handle them when they are convex, preferably of constant complexity. Hence, there has been extensive research into decomposing polyhedra (or, more generally, arrangements of triangles) into tetrahedra and polygons into triangles or other constant-complexity convex pieces. The two main issues in developing decomposition algorithms are (i) to keep the number of pieces in the decomposition small, and (ii) to compute the decomposition quickly.

In the planar setting the number of pieces is, in fact, not an issue if the pieces should be triangles: any polygon admits a *triangulation*—that is, a partition of a polygon into triangles without adding extra vertices—and any triangulation of a simple polygon with n vertices has $n - 2$ triangles. Hence, research focused on developing fast triangulation algorithms, culminating in Chazelle's linear-time triangulation algorithm [19]. An extensive survey of algorithms for decomposing polygons and their applications is given by Keil [61].

In this chapter, we look at the problem in the planar context; we study the problem in $\mathbb{R}^3$ in the next. In particular, in this chapter we look at the triangulation problem with respect to fat objects. Polygon triangulation is a common preprocessing step in geometric algorithms.

It has long been known that linear-time polygon triangulation is possible but the algorithm by Chazelle [19] that achieves this is quite complicated. There are several implementable algorithms which triangulate polygons in near-linear time. For example, Kirkpatrick *et al.* [64] describe an $O(n \log \log n)$ algorithm and Seidel [86] presents a randomized algorithm which runs in $O(n \log^* n)$ expected time. However, it is a major open problem in computational geometry to present a linear-time implementable algorithm.

We study triangulation in the context of fat objects. Relationships between shape complexity and the number of steps necessary to triangulate polygons have been investigated before. For example, it has been shown that monotone polygons [92], star-shaped polygons [84], and edge-visible polygons [93] can all be triangulated in linear time by fairly simple algorithms. Other measures of shape complexity studied include the number of reflex vertices [57] or the sinuosity [27] of the polygon. However, no linear-time algorithm (except Chazelle's complicated general algorithm) is known for fat polygons, arguably the most popular shape-complexity model of the last decade. This is the goal of our work: to develop a simple linear-time algorithm for fat polygons.

We begin, after defining some terms and setting up some tools in Section 2.2, by showing that (α, β)-covered polygons can be "guarded" by a constant number, k, of points in Section 2.3. We call polygons that have this property k-*guardable*. In this context, a polygon is guarded by a set of points G if, for each point p on the boundary of the polygon, there is a line segment between p and one of the guards in G that is contained in P. Note that this is a different definition than the one we gave in Chapter 1 when discussing techniques for dealing with realistic input. We conclude in Section 2.4 by giving two algorithms for triangulating k-guardable polygons in $O(kn)$ time. If the link diameter of the input—see the next section for a formal definition—is d, then one of our algorithms takes $O(dn)$ time— a slightly stronger result. We also describe an algorithm which triangulates k-guardable polygons in $O(kn)$ time. That algorithm uses even easier subroutines than the other, but it requires the actual guards as input, which might be undesirable in certain situations.

As mentioned in Chapter 1, there are several algorithms and data structures for collections of realistic objects. For example, the problem of ray-shooting in an environment consisting of fat objects has been studied extensively [31, 58] (see also Chapter 4 of this thesis). However, there are few results concerning individual realistic objects. We hope that our results on triangulating realistic polygons will encourage further research in this direction.

2.2   Tools and definitions

Throughout this chapter let P be a simple polygon with n vertices. We assume that P has no vertical edges. If P has vertical edges, it is easy to rotate it by a small amount until the vertical edges are eliminated.

We denote the interior of P by $\mathrm{int}(P)$, the boundary of P by ∂P, and the *diameter* of P by $\mathrm{diam}(P)$. The boundary is considered part of the polygon, that is, $P = \mathrm{int}(P) \cup \partial P$. We say that a point p is *in P* if $p \in \mathrm{int}(P) \cup \partial P$.
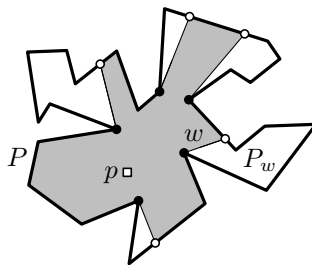
Figure 2.1 The visibility polygon $VP(p, P)$ is shaded. P_w is the pocket of w with respect to $VP(p, P)$.

The segment or edge between two points p and q is denoted by $\overline{pq}$. The same notation implies the direction from p to q if necessary. Two points p and q in P *see* each other if $\overline{pq} \cap P = \overline{pq}$. If p and q see each other, then we also say that p is *visible* from q and vice versa. We call a polygon P *k-guardable* if there exists a set G of k points in P called *guards* such that every point $p \in \partial P$ can see at least one point in G.

A *star-shaped* polygon is defined as a polygon that contains a set of points—the *kernel*— each of which can see the entire polygon. If there exists an edge $\overline{pq} \subset \partial P$ such that each point in P sees some point on $\overline{pq}$, then P is *weakly edge-visible*. The *visibility polygon* of a point $p \in P$ with respect to P, denoted by $VP(p, P)$ is the set of points in P that are visible from p. Visibility polygons are star-shaped and have complexity $O(n)$.
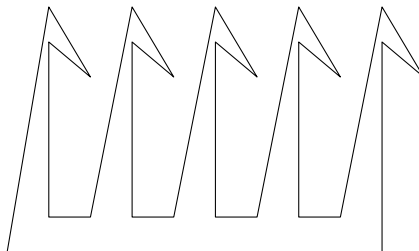
Figure 2.2 A polygon with low link diameter that needs $O(n)$ guards.

A concept related to visibility in a polygon P is the *link distance*, which we denote by $ld(p, q)$ for two points p and q in P. Consider a polygonal path π that connects p and q while staying in int(P). We say that π is a minimum link path if it has the fewest number of segments (links) among all such paths. The link distance of p and q is the number of links of a minimum link path between p and q. We define the *link diameter d* of P to be $\max_{p,q \in P} ld(p, q)$. The link diameter of a polygon may be much less than the number of guards required to see its boundary, and is upper bounded by the number of guards required to see the boundary. This can be seen in the so-called "comb" polygons—see Figure 2.2—that generally have a low link diameter but need a linear number of guards.

Let Q be a subpolygon of P (that is, a simple polygon that is a subset of P), where all vertices of Q are on ∂P. If all vertices of Q coincide with vertices of P, then we call Q a *pure subpolygon*. If ∂P intersects an edge w of ∂Q only at w's endpoints, then w is called a *window* of Q. Any window w separates P into two subpolygons. The one not containing Q is the *pocket* of w with respect to Q (see Figure 2.1). Any vertex added to the polygon (such as the endpoint of a window) is called a *Steiner point*.

Lemma 2.1 (El Gindy and Avis [48]) $VP(p, P)$ *can be computed in* $O(n)$ *time.*

This algorithm, while not trivial, is fairly simple. It involves a single scan of the polygon and a stack. See O'Rourke's book [73] for a good summary.

2.3 Guarding realistic polygons

In this section we discuss several realistic input models for polygons and their connection to k-guardable polygons. We first consider the so-called ε-*good* polygons introduced by Valtr [95]. An ε-good polygon P has the property that any point $p \in P$ can see a constant fraction ε of the area of P. Valtr showed that these polygons can be guarded by a constant number of guards. Hence ε-good polygons fall naturally in the class of k-guardable polygons. Kirkpatrick [63] achieved similar results for a related class of polygons, namely polygons P where any point $p \in P$ can see a constant fraction ε of the length of the boundary of P. These polygons can be guarded by a constant number of guards as well, and hence are k-guardable polygons.
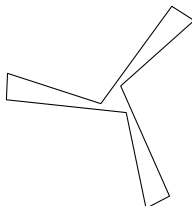
Figure 2.3 A polygon P that is (α, β)-covered but not ε-good. By scaling the length of the edges, the central point of P can be made to see an arbitrarily small fraction of the area of P.

We now turn our attention to fat polygons. In particular, we consider (α, β)-covered polygons—see Chapter 1 for the definition. It is easy to show that the classes of (α, β)-covered polygons and ε-good polygons are not equal—any convex polygon that is not fat is ε-good but not (α, β)-covered, and the polygon in Figure 2.3 is (α, β)-covered but not ε-good. In the remainder of this section we prove that (α, β)-covered polygons can also be guarded by a constant number of guards and hence are k-guardable polygons. In particular, we prove with a simple grid-based argument that we can guard the boundary of an (α, β)-covered polygon with $\lceil 32\pi/(\alpha\beta^2) \rceil$ guards.

Let P be an (α, β)-covered polygon with diameter 1 and let p be a point on ∂P. We construct a circle C of radius $\beta/2$ around p and place $\lceil 4\pi/\alpha \rceil$ guards evenly spaced on the boundary of C. Call this set of guards G_p. By construction, the triangle consisting of p and any two consecutive guards of G_p has an angle at p of $\alpha/2$. Hence any good triangle which is placed at p must contain at least one guard from G_p. Now consider the circle C' centered at p with radius $\beta/4$. We show in the lemma below that any good triangle placed at a point inside the circle C' must contain at least one guard from G_p.

Lemma 2.2 *Let T be a good triangle with a vertex inside the circle C'. Then T contains at least one guard from G_p.*

Proof. Let v be the vertex of T that lies inside C'. Since T is a good triangle, all of its edges have length at least β. Also, all of its angles are at least α. In particular, the angle that is at v is at least α. Since all angles in T are at least α, α is at most $\pi/3$.
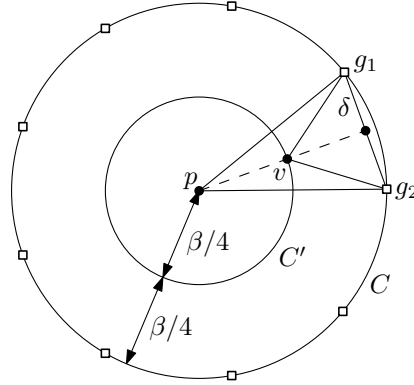
Figure 2.4 The guarding set G_p.

Let r be the ray that bisects the angle at v. Assume that T contains no guards from G_p. It is easy to see that we lose no generality by assuming that v is on the boundary of C'. Indeed, moving T towards the boundary of C' along r, no guards from G_p can enter T. We also lose no generality by assuming that r is orthogonal to C' at v and that it passes through the center point of the segment connecting two consecutive guards from G_p. We now show that even in this worst case, pictured in Figure 2.4, there must be a guard from G_p in T.

We show that there is a segment connecting two consecutive guards from G_p that is completely contained in T. Let g_1 and g_2 be the guards which have the property that r passes through the segment $\overline{g_1 g_2}$. We denote the length of the segment $\overline{g_1 g_2}$ by 2δ. Hence $\tan(\alpha/4) = 2\delta/\beta$. Let the angle $g_1 v g_2$ be denoted by 2θ. We have $\tan\theta = 4\delta/\beta$. Therefore, $\tan\theta = 2\tan(\alpha/4)$. Since $0 < \alpha/4 \le \pi/12 < \pi/4$, we have $0 < 2\tan(\alpha/4) < \tan(\alpha/2)$ (by double-angle identities for tan). This implies that $\tan\theta < \tan(\alpha/2)$ and

23

hence $2\theta < \alpha$. It follows that T must contain the segment $\overline{g_1g_2}$. Since we chose the worst possible T, any good triangle inside C' must contain at least one guard from G_p. □

This lemma almost immediately provides a guarding set for ∂P.

Theorem 2.3 *Let P be a simple (α, β)-covered polygon. The boundary of P can be guarded by $\lceil 4\pi/\alpha \rceil \lceil 2\sqrt{2}/\beta \rceil^2$ guards.*

Proof. Assume without loss of generality that the diameter of P is 1. Thus, P has a bounding square B with area 1. The circle C' in the guarding set G_p from Lemma 2.2 contains a square with area $\beta^2/8$. We cover B by $\lceil 2\sqrt{2}/\beta \rceil^2$ such squares that are each surrounded by a copy of G_p. Since every point of ∂P is contained in at least one such square, this must be a guarding set by Lemma 2.2. Since each copy of G_p contains $\lceil 4\pi/\alpha \rceil$ guards, we need at most $\lceil 4\pi/\alpha \rceil \lceil 2\sqrt{2}/\beta \rceil^2$ guards to guard ∂P. □

2.4 Triangulating k-guardable polygons

We present two algorithms that triangulate a k-guardable polygon. The first algorithm is slightly simpler, but it needs the set of guards as input. The second algorithm does not. The model under which the first algorithm operates, that is, that it needs the guards as input, may seem strange at first. However, given the results of the previous section for (α, β)-covered polygons, we can easily find a small guarding set in linear time for certain fat polygons.

2.4.1 Triangulating with a given set of guards

Let $G = \{g_1, \ldots, g_k\}$ be a given set of k guards in P that jointly see ∂P. In this section we describe a simple algorithm that triangulates P in $O(kn)$ time.

A *vertical decomposition* of P—also known as a trapezoidal decomposition of P, leading to the notation $\mathcal{T}(P)$—is obtained by adding a *vertical extension* to each vertex of P. A *vertical extension* of v, denoted vert(v), is the maximal vertical line segment which is contained in int(P) and intersects v. We sometimes refer to an *upward* (resp. *downward*) *vertical extension* of v. This is the (possibly empty) part of vert(v) that is above (resp. below) v.

Let g be a guard and w be a window of $VP(g, P)$. P_w denotes the pocket of w with respect to $VP(g, P)$. The *vertical projection onto* w is the ordered list of intersection points of w with the vertical extensions of the vertices of P_w (see Figure 2.5).

Our algorithm finds the vertical decomposition $\mathcal{T}(P)$ of P in $O(kn)$ time. In particular, we show how to compute all vertical extensions of $\mathcal{T}(P)$ that are contained in or cross the visibility polygon of a guard in $O(n)$ time. Since each vertex of P is seen by at least one guard, every vertical extension is computed by our algorithm. It is well known that finding a triangulation of a polygon P is simple given the vertical decomposition of P [27]. The
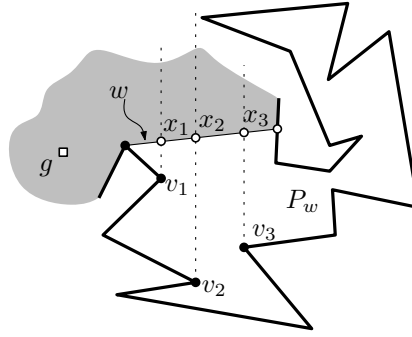
Figure 2.5 The vertical projection onto w is (x_1, x_2, x_3).

most complicated procedure used in our algorithm has the difficulty level of computing the visibility polygon of a point.

Below is a high-level description of our algorithm. The details of the various steps will be discussed later.

TRIANGULATEWITHGUARDS(P, G)

1   **for** each guard $g \in G$
2   **do** find the visibility polygon $VP_1(g, P)$.
3   **for** each window w in $VP_1(g, P)$
4   **do** compute the vertical projection onto w and add the resulting Steiner points to w.
       ▷ *After all windows of $VP_1(g, P)$ have been processed, we have a simple polygon $VP_2(g, P)$ that includes the points in the vertical projections as Steiner points on the windows.*
5   Compute the vertical decomposition of $VP_2(g, P)$. For every vertex v of $VP_2(g, P)$ that is not a Steiner point created in Step 2, add the vertical extension of v to $\partial VP_2(g, P)$, creating $VP_3(g, P)$.
       ▷ *We have now computed the restriction of $\mathcal{T}(P)$ to $VP(g, P)$. That is, every vertical extension that is part of $\mathcal{T}(VP_3(g, P))$ is contained in a vertical extension of $\mathcal{T}(P)$ and every vertical extension of $\mathcal{T}(P)$ that crosses $VP(g, P)$ is represented in $\mathcal{T}(VP_3(g, P))$ for some $g \in G$.*
6   For each vertex v of $VP_3(g, P)$, determine the endpoints of vert(v) on ∂P.

By Lemma 2.1, Step 2 takes $O(n)$ time. We now discuss the other steps of the algorithm.

Step 4: Computing a vertical projection onto a window. Without loss of generality, we assume that w is not vertical and that int($VP(g, P)$) is above w (see Figure 2.7). Let v be a vertex of P_w such that vert(v) intersects w. Furthermore, let z be a point at infinite distance vertically above some point on w. Observe that if we remove the parts of P above w so that z can see all of w, then z can see v. This implies that we should

25
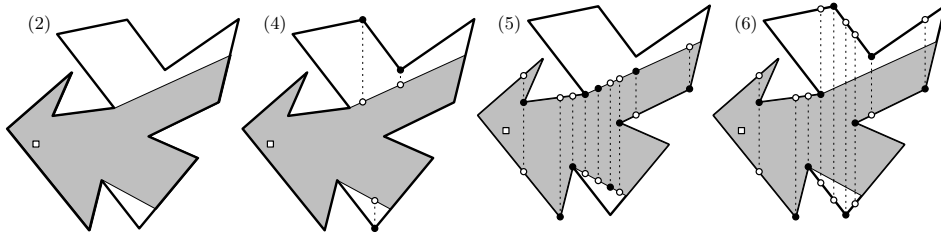
Figure 2.6 Sample execution of the algorithm. The box is the guard, unfilled circles are new Steiner points, and filled circles are points from which a vertical extension is computed.

remove all parts of P_w that are inside the "vertical slab" above w, so that vertices whose vertical extensions intersect w are precisely those that form the visibility polygon of z. The technique of computing a visibility polygon of a point at infinity was first used by Toussaint and El Gindy [94].
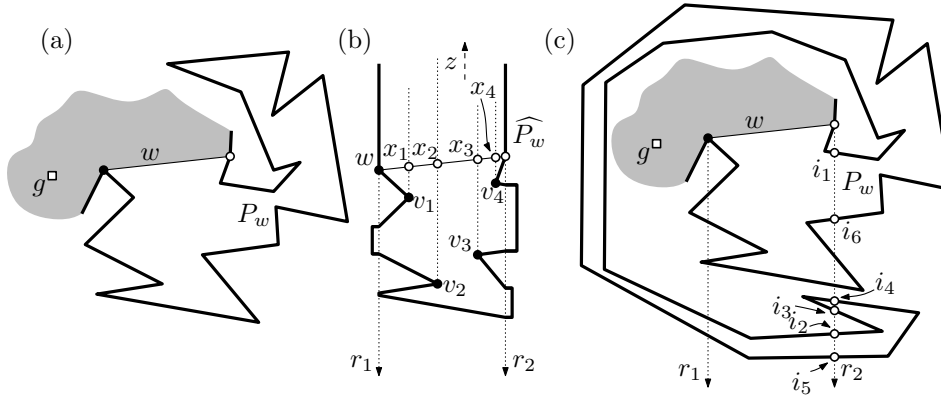
Figure 2.7 Computing a vertical projection. (a) A polygon that does not wrap around w. (b) Its vertical projection. (c) A polygon that wraps around w. The counter c_2 is incremented at i_1 and i_2, decremented at i_3, incremented again at i_4, and decremented two more times at i_5 and i_6, at which time it is once again 0.

We remove all the parts of P_w that are outside the vertical slab directly below w, as follows. Imagine shooting two rays downward from the start- and end-points of w. We call the rays r_1 and r_2. We keep two counters called c_1 and c_2 that are initialized to 0, and are associated to r_1 and r_2, respectively. Assume that r_1 is to the left of r_2. We begin scanning ∂P_w at one of the endpoints of w and proceed toward the other endpoint. If an edge of ∂P_w intersects r_1 from the right, we increment c_1 and proceed as follows until c_1 is again 0. We continue scanning ∂P_w, throwing away edges as we go. If an edge

intersects r_1 from the right, we increment c_1 and if an edge intersects r_1 from the left, we decrement c_1. When c_1 is 0, we connect the first and last intersections of ∂P_w by a segment. The procedure is essentially the same when an edge intersects r_2 except that we interchange "right" and "left". Note that if P_w winds around w many times, c_1 or c_2 might be much larger than 1. Finally, once ∂P_w has been traced back to w, we remove potential intersections between newly constructed line segments along r_1 by shifting them to the left by a small amount proportional to their length. We shift the new segments along r_2 to the right by a small amount proportional to their length. The simplicity of P implies that the new segments are either nested or disjoint, so we obtain a simple polygon that does not cross the vertical slab above w. Finally, we remove w and attach its endpoints to z, thus obtaining polygon $\widehat{P_w}$. The vertices of $VP(z, \widehat{P_w})$ are precisely those vertices of P_w whose vertical extensions intersect w and appear as output in sorted order.

Lemma 2.4 *The vertical projection onto w can be computed in $O(|P_w|)$ time.*

Proof. The algorithm described in the text consists of a scan of ∂P_w and a visibility polygon calculation, which has complexity $O(|P_w|)$. Therefore, it remains to show that a point x is added to w if and only if there is a corresponding vertex v_x in P_w whose vertical extension intersects w at x.

Suppose there is a vertex v_x whose vertical extension intersects w. Then v_x is visible from z, so v_x is included in $VP(z, \widehat{P_w})$ and thus x is added to w. On the other hand, suppose there is a point x added to w. This occurs if there is a vertex v_x which is visible to z through w. Since this is the case, the vertical extension of v_x intersects w. □

Step 5: Computing a vertical decomposition of a star-shaped polygon. Let S be a given star-shaped polygon and g be a point inside the kernel of S. We assume that the vertices of S are given in counterclockwise order around S. To simplify the algorithm, we describe only the computation of the upward vertical decomposition (that is, for each vertex v, we find the upper endpoint of $\text{vert}(v)$) of the part of S that is to the left of the vertical line through g. See Figure 2.8. We say that a vertex v *supports* a vertical line ℓ if the two edges adjacent to v are both on the same side of ℓ.

The algorithm for finding the upward vertical decomposition of S consists of a sequence of alternating leftward and rightward walks: a leftward walk which moves a pointer to a vertex which supports a vertical line (locally) outside S, and a rightward walk which adds vertical decomposition edges. The algorithm begins with the leftward walk which starts from the point directly above g. It ends when the rightward walk passes under g.

The leftward walk simply moves a pointer forward along ∂S until a vertex v_s which supports a vertical line outside S is encountered—so we concentrate on describing the rightward walk. The rightward walk begins with two pointers, p_u and p_d, which initially point to v_s, the last point encountered in the leftward walk. The pointers are moved simultaneously so that they always have the same x-coordinate, with p_d being moved forward along ∂S—that is, counterclockwise—while p_u is moved backward along ∂S (imagine sweeping rightward with a vertical line from v_s). If p_d encounters a vertex, then
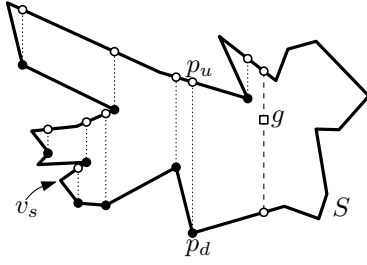
Figure 2.8 Upward vertical decomposition of the part of S to the left of the guard g.

a vertical decomposition edge is created between p_d and p_u. If p_u encounters a vertex v to which a vertical decomposition edge vert(v) is already attached (which implies that v supports a vertical line), then p_u moves to the top of vert(v) and continues from there. When p_d encounters a vertex v that supports a vertical line, the rightward walk ends and the leftward walk begins anew at v.

Lemma 2.5 *The vertical decomposition of a star-shaped polygon P is correctly computed by the above algorithm in $O(n)$ time.*

Proof. The algorithm outlined in the text maintains the following *extension invariant*: the correct upward vertical extension has been found for every vertex to which p_d has pointed. Initially, the invariant is trivially true.

By construction, p_d visits all vertices of S that are the endpoints of the edges of the upward vertical decomposition of S in counterclockwise order. Hence the algorithm constructs a vertical extension for each of these vertices. It remains to show that the upper endpoint of the vertical extension is correctly identified. Denote the current position of p_d by v_d. Again by construction, p_u lies vertically above v_d at position v_u. We need to show that $\overline{v_d v_u}$ is not intersected by an edge of S.

Consider the triangle $g v_d v_u$. Since g sees all of S, $\overline{g v_d}$ and $\overline{g v_u}$ can not be intersected by an edge of S. This implies that any edge e that intersects $g v_d v_u$ must intersect $\overline{v_d v_u}$. Furthermore, e must be an edge in the chain C_L, which is the chain from v_u to v_d in counterclockwise order. To show that no edge from C_L intersects $\overline{v_u v_d}$, we establish the *order invariant*: C_L is always to the left of $\overline{p_u p_d}$. The invariant is trivially true whenever p_u and p_d point to v_s, that is, whenever we begin a rightward walk. Suppose that the invariant has been true until step k and we will show that it is still true at step $k + 1$. Let C'_L be the chain from p_u to p_d at step k and C_L be the chain from p_u to p_d at step $k + 1$. There are three cases in step k: (a) p_d is pointing to a vertex of S, (b) p_u is pointing to a vertex of S without a vertical extension, or (c) p_u is pointing to a vertex v of S with a vertical extension. See Figure 2.9. In the first two cases, the invariant is maintained since C_L only differs from C'_L by two segments that by definition both lie to the left of $\overline{p_u p_d}$. Since the vertices in C_L come before v_d, the correct vertical extension of each vertex in

C_L has been computed by the assumption of the extension invariant. This implies that the order invariant is also maintained in the case where p_u is pointing to a vertex v of S with a vertical extension and is moved to the top of $\mathrm{vert}(v)$. This is because C'_L differs from C_L by a segment which is to the left of p_d and a chain which must be to the left of $\overline{p_u p_d}$ since $\mathrm{vert}(v)$ is a valid vertical extension.

Both p_d and p_u visit every vertex of S at most once, hence the running time is $O(n)$. □
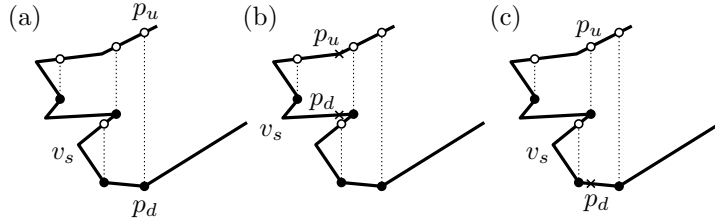
Figure 2.9 Establishing correctness of the order invariant: three cases.

Step 6: Computing the endpoints of vertical extensions. The final step of the algorithm is to find and connect the endpoints of the vertical extensions of every vertex of $VP_3(g, P)$. Let v be an arbitrary vertex of $VP_3(g, P)$. If both endpoints of $\mathrm{vert}(v)$ are on the boundary of $VP(g, P)$, we have already found and connected them in the previous step. Thus, let us assume that at least one of the endpoints of $\mathrm{vert}(v)$ is not on the boundary of $VP(g, P)$. That is, $\mathrm{vert}(v)$ intersects at least one window of $VP(g, P)$. Since we have already connected the endpoints of $\mathrm{vert}(v) \cap VP(g, P)$ in the previous step, it is sufficient to find the endpoints of $\mathrm{vert}(v)$ that are outside of $VP(g, P)$. Thus, it suffices to examine vertices that are Steiner points on windows.

Let $v_1, \ldots, v_j$ be vertices on window w, in sorted order. Again without loss of generality, we assume that $\mathrm{int}(VP(g, P))$ is above w. To find the endpoint of $\mathrm{vert}(v)$ that is below w for all $v \in \{v_1, \ldots, v_j\}$, we use the visibility polygon $VP(z, \widehat{P_w})$ computed in Step 4 of the algorithm. Note that the vertices of $VP(z, \widehat{P_w})$ as well $\{v_1, \ldots, v_j\}$ are sorted by x-coordinate. Thus we find the endpoints of $\{\mathrm{vert}(v) | v \in \{v_1, \ldots, v_j\}\}$ by simultaneously scanning in $VP(z, \widehat{P_w})$ and $\{v_1, \ldots, v_j\}$ (as though performing a merge operation in merge-sort). Since $\sum_w |P_w| \le n$ and the number of Steiner points added to windows is at most n, we find the endpoints of the vertical extensions of all Steiner points on windows in $O(n)$ time.

Since each guard is processed in linear time, we obtain the following.

Theorem 2.6 *The algorithm* TRIANGULATEWITHGUARDS *computes the vertical decomposition of an n-vertex k-guardable polygon in $O(kn)$ time, if the k guards are given.*

2.4.2 Triangulating without guards

In many situations where triangulation is desired, it may be unrealistic to expect a set of guards as part of the input. In this section we show how to triangulate a k-guardable polygon in $O(kn)$ time without knowing the guards. The most complicated procedure used in our algorithm is computing the visibility polygon from an edge in linear time [56]. This is, in fact, considerably more complicated than all the steps of the previous algorithm. We begin with some new notation and definitions.

The *edge-visibility polygon*, $EVP(e, P)$, of an edge e with respect to polygon P consists of all points in P that are visible from at least one point on e. We sometimes call $EVP(e, P)$ the *weak visibility polygon* of the edge e if the polygon is clear from the context. We define an *extended edge-visibility polygon* of e with respect to P, denoted by $EEVP(e, P)$, to be the smallest (in terms of the number of edges) pure subpolygon of P that contains $EVP(e, P)$. These concepts are illustrated in Figure 2.10.
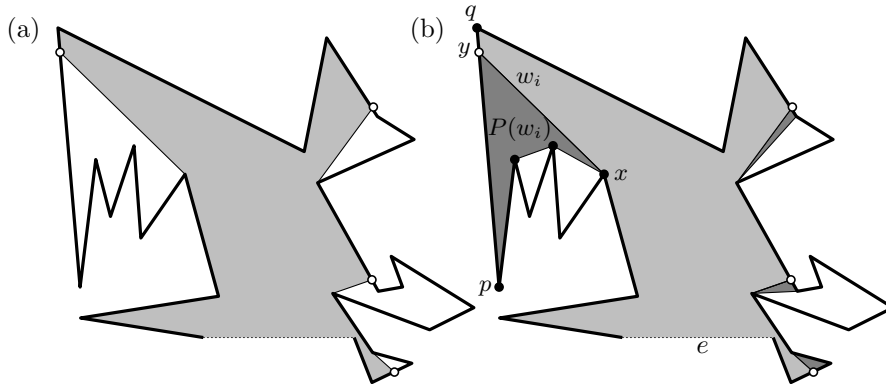
Figure 2.10   (a) The weak visibility polygon of the dotted edge. (b) The associated extended edge visible polygon. $EEVP(e, P)$ is the union of the light and dark gray regions.

The *geodesic* between two points in P is the shortest polygonal path connecting them that is contained in P. The vertices of a geodesic (except possibly the first and last) belong to ∂P. Below, we show that Melkman's algorithm [68] can find a specific type of geodesic related to finding the $EEVP$ of a polygon.

Lemma 2.7 *Let x be a vertex of polygon P and let y be a point on edge $\overline{vw} \in P$. If y sees x, then the geodesic between x and v: (a) is a convex chain and entirely visible from y, and (b) can be computed in $O(n)$ time.*

Proof. Property (a) holds trivially if x sees v. Consider the case where x does not see v. Then, the triangle (x, y, v), denoted by T, must contain at least one vertex of P in its

30
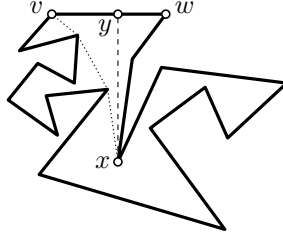
Figure 2.11 The geodesic from x to v.

interior. Let I be all the vertices of P inside T and let $CH(I)$ be the convex hull of I. The path $S = \partial CH(I) \setminus \overline{xv}$ is the geodesic from x to v. Any other path from x to v inside T can be shortened. Thus, property (a) holds.

To prove property (b), note that since the geodesic we seek is entirely visible from y by part (a) it is fully contained in $VP(y, P)$. We compute $VP(y, P)$ in linear time. Consider the polygonal chain from x to v along $\partial VP(y, P)$ that avoids y. By construction of $VP(y, P)$, the shortest path from x to v is part of the convex hull of this chain. Using Melkman's algorithm, we compute the convex hull of a simple polygonal chain in linear time. $\square$

Finally, a weakly edge-visible polygon can be triangulated using a very simple algorithm known as Graham's scan. The following lemma formalizes that.

Lemma 2.8 (Toussaint and Avis [93]) *Let P be a weakly edge-visible polygon. By performing Graham's scan on the points of P we can obtain a triangulation of P.*

We now show how to compute and triangulate the extended edge visibility polygon, which is the main subroutine of our algorithm.

Lemma 2.9 *$EEVP(e, P)$ can be computed and triangulated in $O(n)$ time.*

Proof. We begin by computing $EVP(e, P)$ in $O(n)$ time using the algorithm of Heffernan and Mitchell [56]. This yields a set of windows W and their associated pockets. For each window $w_i \in W$ that is not a diagonal of P, we do the following.

Let x be the endpoint of w_i closer to e, and let y be the endpoint farther from e. Then x is a vertex of P, and y is an interior point on some edge $\overline{pq}$ of P. Without loss of generality let p be the endpoint of $\overline{pq}$ that is inside the pocket of w_i, as illustrated in Figure 2.10 (b). Since x sees y, we can use Lemma 2.7(b) to compute the geodesic from x to p. Let $P(w_i)$ denote the polygon enclosed by the geodesic from x to p, $\overline{py}$ and w_i. It is simple to verify that the extended edge-visibility polygon is $EEVP(e, P) = EVP(e, P) \cup \left(\bigcup_{w_i \in W} P(w_i) \right)$.

By Lemma 2.7 (b), each pocket $P(w_i)$ can be computed in time linear in the size of the pocket of w_i. Since pockets are disjoint and can be processed in order, $\bigcup_{w_i \in W} P(w_i)$, and thus $EEVP(e, P)$, can be computed in $O(n)$ time.

We now proceed to triangulate $EEVP(g, P)$. Consider $P(w_i)$. Let T be the triangle determined by the points x, y and q. If e sees q, then q sees each vertex in $P(w_i) \cup T$ by Lemma 2.7 (a). Therefore, $P(w_i) \cup T$ is a weakly edge-visible pure subpolygon of P. By Lemma 2.8, we can triangulate $P(w_i) \cup T$ in $O(|P(w_i)|)$ time.

If e does not see q then $q \in P(w_j)$ for some $w_j \in W$ with $j \neq i$. Let Q be the quadrilateral determined by the endpoints of w_i and w_j. The polygon $Y = P(w_i) \cup P(w_j) \cup Q$ is a pure subpolygon of P and each of its vertices is visible from p or q, which means that Y is weakly edge-visible from $\overline{pq}$. This implies that Y can be triangulated using a simple method as before.

It is straightforward to verify that all of the pure subpolygons of $EEVP(e, P)$ triangulated thus far are pairwise non-overlapping. If T is the union of these subpolygons then the closure of $EEVP(e, P) \setminus T$ is a weakly edge-visible pure subpolygon of $EEVP(e, P)$ and thus can also be triangulated in linear time. This results in a triangulation of $EEVP(e, P)$, as required. $\qquad\square$

When $EEVP(e, P_i)$ is triangulated, diagonals of P that are on $\partial EEVP(e, P_i)$ become windows of new pockets. Each such window serves as the edge from which a new visibility polygon will be computed and triangulated, within its respective pocket. In this recursive manner we break pockets into smaller components until all of P is triangulated. The procedure, although straightforward, is outlined below in more detail. This is followed by the analysis of the time complexity, where we show that the recursion depth is of the order of the number of guards that suffice to guard ∂P.

We will maintain a queue $\mathcal{S}$ of non-overlapping polygons such that each $P_i \in \mathcal{S}$ has one edge w_i labelled as a window. Thus elements of $\mathcal{S}$ are pairs (P_i, w_i). We start with $\mathcal{S} := (P, w)$, where w is an arbitrary boundary edge of P. We process the elements of $\mathcal{S}$ in the order in which they were inserted. The main loop of our algorithm is as follows:

TRIANGULATEWITHOUTGUARDS(P)

1 $\mathcal{S} := (P, w)$ where w is an arbitrary edge of P
2 **while** $\mathcal{S} \neq \emptyset$
3   **do for** each $(w_i, P_i) \in \mathcal{S}$
4   **do** remove (w_i, P_i) from $\mathcal{S}$.
5   Compute and triangulate $EEVP(w_i, P_i)$.
6   Add the edges of the triangulation to P.
7   **for** each boundary edge w_j of $EEVP(w_i, P_i)$ that is a diagonal
                  of P.
8   **do** identify Q_j as the untriangulated portion of P whose
                      boundary is enclosed by w_j and ∂P.
9   Add every remaining untriangulated portion (w_j, Q_j) to $\mathcal{S}$.
10 **return** P.

Theorem 2.10 *The algorithm* TRIANGULATEWITHOUTGUARDS *triangulates an* n-*vertex* k-*guardable polygon in* $O(kn)$ *time.*

Proof. We first note that the $EEVP$s created by our algorithm define a tree structure T, as follows. At the root of T is $EEVP(w, P)$. For every window w_j of $EEVP(w_i, P_i)$, we have that $EEVP(w_j, P_j)$ is a child of $EEVP(w_i, P_i)$. The construction of the child nodes from their parents ensures that no $EEVP$ overlaps with any other and that the triangulation covers the entire polygon P.

We now show that T has at most $3k$ levels (a *level* is a set of nodes at the same distance from the root) which implies that the main loop of the algorithm performs at most $3k$ iterations. Let ℓ_i, ℓ_{i+1}, and ℓ_{i+2} be three successive levels of T, in which all the nodes in ℓ_{i+1} are descendants of the nodes in ℓ_i, and where all the nodes in ℓ_{i+2} are descendants of the nodes in ℓ_{i+1}. Further, let G be a point set of size k such that every point $p \in \partial P$ sees at least one guard of G. Assume, for the purpose of obtaining a contradiction, that there are no guards from G in the $EEVP$s corresponding to the nodes in levels ℓ_i, ℓ_{i+1}, or ℓ_{i+2}.

Let g be a guard which sees into a node n_i at level ℓ_i through window w_i. There are two cases: either g is at a higher level than ℓ_i or it is at a lower level. If g is in a higher level and is visible from a window of n_i, then g can be in only one level: ℓ_{i+1} (because ℓ_{i+1} contains the union of all the edge-visibility polygons of the windows of the nodes in ℓ_i). We have assumed that this can not happen. Otherwise, if g is in a lower level, g can not see into any level higher than ℓ_i, because w_i must be the window which created n_i.

The combination of these two facts implies that no guard from G can see into ℓ_{i+1}. This is a contradiction to G being a guarding set. Therefore, G must have at least one guard in ℓ_i, ℓ_{i+1}, or ℓ_{i+2}. This implies that there is at least one guard for every three levels, or at most three levels per guard.

Each level of the tree can be processed in $O(n)$ time by Lemma 2.9, since all nodes of a level are disjoint. Therefore, the algorithm terminates in $O(kn)$ time. □

As is apparent from the proof of Theorem 2.10, our algorithm runs in $O(tn)$ time, where t is the number of iterations of the while-loop. The above argument also implies a stronger result. The number of iterations, t, of the while loop is proportional to the link diameter, d, of the polygon, since any minimum link path between two points must have at least one bend for every three levels. This leads to the following corollary:

Corollary 2.11 *The algorithm* TRIANGULATEWITHOUTGUARDS *triangulates an* n-*vertex polygon with link diameter* d *in* $O(dn)$ *time.*

33

2.5 Conclusion

Several known classes of realistic polygons are in fact k-guardable. In particular, we have shown that the boundary of an (α, β)-covered polygon can be guarded by a constant—depending on α and β—number of guards, which implies that (α, β)-covered polygons are k-guardable. We also gave two simple algorithms that triangulate k-guardable polygons in linear time, if k is a constant. The first algorithm is slightly simpler, but does require the guards as input, while the second algorithm does not need the guards.
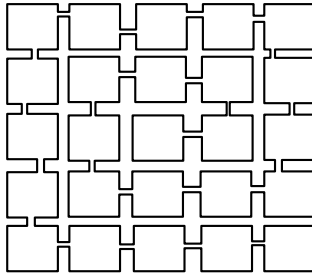
Figure 2.12   A locally-γ-fat polygon that requires $\Omega(n)$ guards.

Our work leaves some open problems. First, can the techniques shown here be used to design a triangulation algorithm which does not depend on the number of guards? Second, are there other problems that can be solved efficiently for k-guardable polygons? Finally, are there more general classes of polygons that can be triangulated in linear time with simple algorithms? For example, our approach does not work with locally-γ-fat polygons because they can require $\Omega(n)$ guards—see Figure 2.12. However, we believe it is likely that there is a simple triangulation algorithm that works for locally-γ-fat polygons.

CHAPTER 3

Decomposing non-convex fat polyhedra

3.1 Introduction

In the previous chapter, we studied triangulation—a common decomposition of polygons in the plane. We saw that every simple polygon of complexity n admits a partition into $n - 2$ triangles and that we do not need to add any extra vertices; every triangle edge is either a boundary edge or a diagonal.

For 3-dimensional polyhedra, however, the situation is much less rosy. First of all, not every non-convex polyhedron admits a tetrahedralization: there are polyhedra that cannot be decomposed into tetrahedra without using Steiner points, such as *Schönhardt's polyhedron* [83]. Moreover, deciding whether a polyhedron admits a tetrahedralization without Steiner points is NP-complete [82]. Thus we have to settle for decompositions using Steiner points. Chazelle [18] has shown that any polyhedron with n vertices can be decomposed into $O(n^2)$ tetrahedra, and that this is tight in the worst case: there are polyhedra with n vertices for which any decomposition uses $\Omega(n^2)$ tetrahedra. (In fact, the result is even stronger: any *convex decomposition*—a decomposition into convex pieces—uses $\Omega(n^2)$ pieces, even if one allows pieces of non-constant complexity.) Since the complexity of algorithms that need a decomposition depends on the number of pieces in the decomposition, this is rather disappointing. The polyhedron used in Chazelle's lower-bound example (known as Chazelle's polyhedron) is quite special, however, and one may hope that polyhedra arising in practical applications are easier to handle. This is the topic of this chapter: are there types of polyhedra that can be decomposed into fewer than a

35

quadratic number of pieces?

Erickson [50] has answered this question affirmatively for so-called *local polyhedra* (see below) by showing that any such 3-dimensional polyhedron P can be decomposed into $O(n \log n)$ tetrahedra and that this bound is tight. We consider *fat polyhedra*.

Types of fatness. Before we can state our results, we first need to give the definition of fatness that we use. When the input is convex, most of these definitions are equivalent up to constants. When the input is not convex, however, this is not the case: polyhedra that are fat under one definition may not be fat under a different definition. Therefore we use two different definitions from Chapter 1: locally-γ-fat polyhedra and (α, β)-covered polyhedra.

For comparison, let us also give the definition of a local polyhedron P [50]. To this end, define the *scale factor at a vertex* v of P as the ratio between the length of the longest edge incident to v and the minimum distance from v to any other vertex. The *local scale factor* of P is now the maximum scale factor at any vertex. The *global scale factor* of P is the ratio between the longest and shortest edge lengths of the whole polyhedron. Finally, P is called a *local polyhedron* if its local scale factor is a constant, while its global scale factor is polynomial in the number of vertices of P.

Our Results. First we study the decomposition of (α, β)-covered polyhedra and locally-γ-fat polyhedra into tetrahedra. By modifying Chazelle's polyhedron so that it becomes (α, β)-covered, we obtain the following negative result.

- There are (α, β)-covered (and, hence, locally fat) polyhedra with n vertices such that any decomposition into convex pieces uses $\Omega(n^2)$ pieces.

Next we restrict the class of fat polyhedra further by requiring that their faces should be convex and fat, when considered as planar polygons in the plane containing them. For this class of polyhedra we obtain a positive result.

- Any locally-fat polyhedron (and, hence, any (α, β)-covered polyhedron) with n vertices whose faces are convex and fat can be decomposed into $O(n)$ tetrahedra in $O(n \log n)$ time.

Several applications that need a decomposition or covering of a polyhedron into tetrahedra would profit if the tetrahedra were fat. In the plane any fat polygon can be covered[1] by $O(n)$ fat triangles, as shown by Van Kreveld [98] (for a slightly different definition of fatness). We show that a similar result is, unfortunately, not possible in 3-dimensional space.

---

[1]A *covering* of a set S is a set of subsets of S where every element of S is in at least one subset. This is as opposed to a *partition* of S which is a set of subsets of S where every element of S is in exactly one subset. A *decomposition* is a generic term that can refer to either a covering or a partition.

- There are (α, β)-covered (and, hence, locally-fat) polyhedra with n vertices and convex fat faces such that the number of tetrahedra in any covering that only uses fat tetrahedra cannot be bounded as a function of n.

For some applications—ray shooting is an example—we do not need a decomposition of the full interior of the given polyhedron P; instead it is sufficient to have a *boundary covering*, that is, a set of objects whose union is contained in P and that together cover the boundary of P. Interestingly, when we consider boundary coverings there is a distinction between (α, β)-covered polyhedra and locally-fat polyhedra:

- The boundary of any (α, β)-covered polyhedron P, can be covered by $O(n^2 \log n)$ fat convex constant-complexity polyhedra, and there are (α, β)-covered polyhedra that require $\Omega(n^2)$ convex pieces in any boundary covering. If the faces of the (α, β)-covered polyhedron are fat, convex and of approximately the same size, then the boundary can be covered with only $O(n)$ convex fat polyhedra. Furthermore, the worst-case number of convex pieces needed to cover the boundary of a locally-fat polyhedron cannot be bounded as a function of n.

Finally, we consider boundary coverings using so-called *towers* [9]—see Section 3.3 for a definition. Such coverings are useful for ray shooting.

Table 3.1 summarizes our results.

	decomposition of interior by		covering of boundary by	
	tetrahedra	fat tetrahedra	fat convex polyhedra	towers
general	$\Theta(n^2)$ [18]	$\times$	$\times$	unbounded
local	$\Theta(n \log n)$ [50]	$\times$	$\times$	unbounded
locally fat	$\Theta(n^2)$	unbounded	unbounded	unbounded
with fat faces	$\Theta(n)$	unbounded	unbounded	unbounded
(α, β)-covered	$\Theta(n^2)$	unbounded	$O(n^2 \log n), \Omega(n^2)$	$\Theta(1)$
with fat faces	$\Theta(n)$	unbounded	$O(n^2 \log n)$	$\Theta(1)$

Table 3.1 Overview of results on decomposing and covering polyhedra. An entry marked $\times$ means that the corresponding decomposition or covering is not always possible. (For example, since general polyhedra can have arbitrarily sharp vertices, they cannot always be decomposed into fat tetrahedra.)

Applications. As already mentioned, decomposing polyhedra into tetrahedra or other convex pieces is an important preprocessing step in many applications. Below we mention some of these applications, where our results help to get improved performance when the input polyhedra are fat.

Hachenberger [55] studied the computation of Minkowski sums of non-convex polyhedra. To obtain a robust and efficient algorithm for this problem, he first decomposes the poly-

hedra into convex pieces. Our results imply that this first step can be done such that the resulting number of pieces is $O(n)$ if the input polyhedra are locally fat with fat faces, while in general this number can be quadratic.

Another application is in computing depth orders. The best-known algorithm to compute a depth order for n tetrahedra runs in time $O(n^{4/3+\varepsilon})$ [28]. In Chapter 5 we show that for fat convex polyhedra of constant complexity, this can be improved to $O(n \log^3 n)$. Our results imply that any constant-complexity (α, β)-covered polyhedron can be decomposed into constant-complexity fat convex polyhedra. It can be shown that this is sufficient to be able to use the depth-order algorithm of Chapter 5. Similarly, our results imply that the results from Chapter 4 on vertical ray shooting in convex polyhedra extend to constant-complexity (α, β)-covered polyhedra. Finally, our results on boundary coverings with towers imply that we can use the method of Chapter 4 to answer ray-shooting queries in (α, β)-covered polyhedra in $O((n/\sqrt{m}) \log^2 n)$ time with a structure that uses $O(m^{1+\varepsilon})$ storage, for any $n \le m \le n^2$. This is in contrast to the best-known data structure for arbitrary polyhedra [28], which gives $O(n^{1+\varepsilon}/m^{1/4})$ query time with $O(m^{1+\varepsilon})$ storage for $n \le m \le n^4$.

3.2 Decomposing the interior

In this section we discuss decomposing the interior of fat non-convex objects into tetrahedra. We start with decompositions into arbitrary tetrahedra, and then we consider decompositions into fat tetrahedra.

3.2.1 Decompositions into arbitrary tetrahedra

The upper bound. Let P be a locally-γ-fat polyhedron in $\mathbb{R}^3$ whose faces, when viewed as polygons in the plane containing the face, are convex and β-fat. We will prove that P can be decomposed into $O(n)$ tetrahedra in $O(n \log n)$ time.

In our proof, we will need the concept of density. Recall from Chapter 1 that the *density* of a set S of objects is defined as the smallest number λ such that the following holds: any ball $B \subset \mathbb{R}^3$ is intersected by at most λ objects $o \in S$ such that $\text{size}(o) \ge \text{size}(B)$.

We also need the following technical lemma.

Lemma 3.1 *Let P be a convex β-fat polygon embedded in $\mathbb{R}^3$ where $diam(P) \ge 1$. Let C and C' be axis-aligned cubes centered at the same point. Let the side length of C be 1 and the side length of C' be $2\sqrt{3}/3$. If P intersects C, then $P' := P \cap C'$ is β'-fat for some $\beta' = \Omega(\beta)$.*

Proof. Since P must cross the region between C and C' to be different from P', $\text{size}(P') \ge (\sqrt{3}/3) - 1/2$. For the same reason and since P is fat, this implies that the area of P' is

at least $((2\sqrt{3} - 3)/12)^2\beta\pi$. Since the diameter of C' is 2, the diameter of P' is at most 2. Since P' is convex, its fatness is determined by a circle whose center is placed at one of the vertices that determines the diameter of P'. This implies that the fatness of P' is at least

$$\frac{\pi\left(\frac{2\sqrt{3}-3}{12}\right)^2\beta}{\pi 2^2} = \frac{7 - 4\sqrt{3}}{96}\beta\,.$$

$\square$

The following lemma shows that the faces of a locally-γ-fat polyhedron have low density if they are fat themselves.

Lemma 3.2 *Let F_P be the set of faces of a locally-γ-fat polyhedron P treated as polygons. If the faces of P are themselves β-fat and convex, then F_P has density $O(1/\gamma\beta^3)$.*

Proof. Without loss of generality, let S be a sphere with unit radius. We wish to show that the number of faces $f \in F_P$ with $\text{size}(f) \geq 1$ that intersect S is $O(1/\gamma\beta^3)$.

Partition the bounding cube of S into eight equal-sized cubes by bisecting it along each dimension. Consider one of the cubes: call it C. Also construct an axis-aligned cube C' that has side length $2\sqrt{3}/3$ and concentric with C. For all faces f intersecting C that have $\text{size}(f) \geq 1$, we define $f' := f \cap C'$. By Lemma 3.1, we know that f' is β'-fat for some $\beta' = \Omega(\beta)$.

Since f' is a fat convex polygon with a diameter of at least $2\sqrt{3}/3 - 1$, it must contain a circle c of radius $\rho = \beta'(2\sqrt{3}/3 - 1)/8$ [96]. For any such circle c, there is a face F of C' such that the projection of c onto F is an ellipse which has a minor axis with length at least $\rho/\sqrt{2}$.
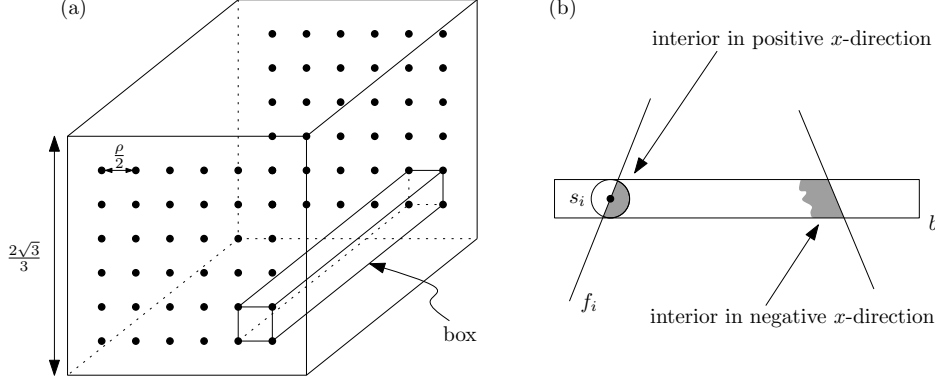
Figure 3.1 (a) A box. (b) A box b (side view) and the different types of faces assigned to it.

We make a grid on each face of C' where every grid cell has side length $\rho/2$. We call the rectangular prism between two grid cells on opposite faces of C' a *box*—see Figure 3.1(a).

39

Each face f' has an intersection with some box that is the entire cross-section of the box. We assign each face to such a box.

We now consider the set of faces that can be assigned to any one box b. There are two types of faces in this set—see Figure 3.1(b). For example, if b has its long edges parallel to the x axis, there are the faces that have the interior of P in the positive x direction and the faces that have the interior in the negative x direction. We consider one type of face at a time. For each face f_i, we place a sphere s_i with radius $\rho/4$ so that its center is on f_i and in the center of b (that is, the center is exactly between the long faces of b). Since P is locally-γ-fat,

$$vol(P \sqcap s_i) \geq \frac{\gamma 4\pi}{3} \left(\frac{\rho}{4}\right)^3 = \frac{\gamma \pi \rho^3}{48} \,.$$

Since we only consider one type of face, $(P \sqcap s_i) \cap (P \sqcap s_j) = \emptyset$ for any $s_j \neq s_i$. Therefore the number of faces of one type that can cross one box is $8\sqrt{3}/\gamma \pi \rho$. The number of faces that can cross one box is twice that. The number of boxes per direction is

$$\left(\frac{2\sqrt{3}/3}{\rho/2}\right)^2 = \frac{16}{3\rho^2}$$

and the number of directions is 3. Hence, the number of faces that can intersect S is at most

$$2 \cdot 3 \cdot \frac{8\sqrt{3}}{\gamma \pi \rho} \cdot \frac{16}{3\rho^2} = \frac{256\sqrt{3}}{\pi \gamma \rho^3} \,.$$

Since $\rho = \Omega(\beta)$, this is $O(1/\gamma \beta^3)$. □

Since the set F_P of faces of the polyhedron P has density $O(1/\gamma \beta^3) = O(1)$, there is a BSP for F_P of size $O(n)$ that can be computed in $O(n \log n)$ time [29]. The cells of the BSP are convex and contain at most one facet, so we can easily decompose all cells further into $O(n)$ tetrahedra in total.

Theorem 3.3 *Let γ and β be fixed constants. Any locally-γ-fat polyhedron with β-fat convex faces can be partitioned into $O(n)$ tetrahedra in $O(n \log n)$ time, where n is the number of vertices of the polyhedron.*

The lower bound. Next we show that the restriction that the faces of the polyhedron are fat is necessary, because there are fat polyhedra with non-fat faces that need a quadratic number of tetrahedra to be covered.

The polyhedron known as *Chazelle's polyhedron* [18]—see Figure 3.2(b)—is an important polyhedron used to construct lower-bound examples. We describe a slight modification of that polyhedron which makes it (α, β)-covered and retains the properties needed for the lower bound.

The essential property of Chazelle's polyhedron is that it contains a region sandwiched between a set L of line segments defined as follows. Fix a small positive constant $\varepsilon > 0$.

For an integer i with $1 \leq i \leq n$, define the line segment ℓ_i as

$$\ell_i := \{(x, y, z) : 0 \leq x \leq n + 1 \text{ and } y = i \text{ and } z = ix - \varepsilon\}$$

and the line segment ℓ_i' as

$$\ell_i' := \{(x, y, z) : x = i \text{ and } 0 \leq y \leq n + 1 \text{ and } z = iy\}.$$

Next define

$$L := \{\ell_i : 1 \leq i \leq n\} \cup \{\ell_i' : 1 \leq i \leq n\}.$$

The region $\Sigma := \{(x, y, z) : 1 \leq x, y \leq n \text{ and } xy - \varepsilon \leq z \leq xy\}$ between these segments has volume $\Theta(\varepsilon n^2)$. Chazelle showed that for any convex object o that does not intersect any of the segments in L we have $\mathrm{vol}(o \cap \Sigma) = O(\varepsilon)$. These two facts are enough to show that $\Omega(n^2)$ convex objects are required to cover any polyhedron that contains Σ but whose interior does not intersect the segments in L.

(a) (b)
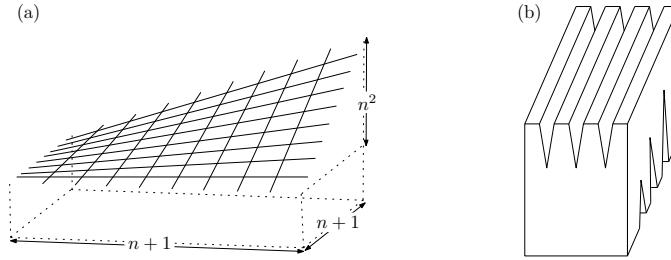
Figure 3.2 (a) The line segments used in the lower-bound construction (not to scale). (b) Chazelle's polyhedron before modification (also not to scale).

Chazelle turns the set of line segments into a polyhedron by putting a box around L, and making a slit into the box for each segment, as shown in Figure 3.2(b). The resulting polyhedron has each of the segments in L as one of its edges, and contains the sandwich region Σ. Hence, any convex decomposition or covering of its interior needs $\Omega(n^2)$ pieces.

Chazelle's polyhedron is not (α, β)-covered. We therefore modify it as follows. First of all, we make the outer box from which the polyhedron is formed a cube of size $6n^2 \times 6n^2 \times 3n^2$ centered at the origin. Second, we replace the slits by long triangular prisms—we will call the prisms *needles* from now on—sticking into the cube. Thus, for each segment in L, there is a needle that has an edge containing the segment. We do not completely pierce the cube with the needles, so that the resulting polyhedron, P, remains simple (that is, topologically equivalent to a sphere). Note that Σ is still contained in P, and that for each segment in L there is an edge containing it.

Next we argue that P is (α, β)-covered. First, consider a point $p \in \partial P$ on one of the needles. Assume without loss of generality that the needle is parallel to the xz-plane. If p is near one of the needles going into the other direction, then the situation is as in Figure 3.3. Note that the distance between consecutive needles of the same orientation—

41
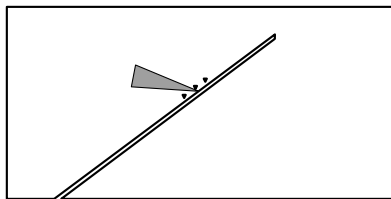
Figure 3.3 Cross-section of the polyhedron P shown with the cross-section of a good tetrahedron (shaded).

that is, the distance between the small triangles in Figure 3.3—is at least 1. Moreover, we can choose the distance ε between the needles of opposite orientation—that is, the distance between the small triangles and the long needle in Figure 3.3—as small as we like. The same is true for the "width" of the needles—that is, the size of the small triangles in the figure. Hence, we can make the construction such that we can always put a good (that is, large and fat) tetrahedron at p.

Next, consider a point $p \in \partial P$ that is near one of the places where a needle "enters" the cube. Note that the segments in L have slopes ranging from 1 to n, and that any needle passes near the center of the cube—this is true since the cube has size $6n^2 \times 6n^2 \times 3n^2$, while the segments in L all pass at a distance at most n from the cube's center. Hence, the needles will intersect the bottom facet of the cube, and they make an angle of at least $45°$ with the bottom facet. This implies that also for points p near the places where these needles enter the cube, we can place a good tetrahedron.

Finally, it is easy to see that for points p on a cube facet, and for points on a needle that are not close to a needle of opposite orientation, we can also put a good tetrahedron. We can conclude with the following theorem.

Theorem 3.4 *There are constants $\alpha > 0$ and $\beta > 0$, such that there are (α, β)-covered polyhedra for which any convex decomposition consists of $\Omega(n^2)$ convex pieces, where n is the number of vertices of the polyhedron.*

3.2.2 Decompositions and coverings with fat tetrahedra

When we attempt to partition non-convex polyhedra into fat tetrahedra, or other fat convex objects, the news is uniformly bad. That is, no matter which of the realistic input models we use (of those we are studying), the number of fat convex objects necessary to cover the polyhedron can be made arbitrarily high. For polyhedra without fatness restrictions, there are many examples which require an arbitrary number of fat convex objects for partitioning. In fact, for any constant $\beta > 0$ we can even construct a polyhedron that cannot be covered at all into β-fat convex objects—simply take a polyhedron that has a vertex whose solid angle is much smaller than β. It is also not hard to construct, for any

42

given $\beta > 0$, a local polyhedron that cannot be covered with β-fat convex objects. For instance, we can take a pyramid whose base is a unit square and whose top vertex is at distance $\varepsilon \ll \beta$ above the center of the base.

Next we show how to construct, for any given k, an (α, β)-covered polyhedron of constant complexity and with convex fat faces, which requires $\Omega(k)$ fat convex objects to cover it. First we observe that a rectangular box of size $1 \times (\beta/k) \times (\beta/k)$ requires $\Omega(k)$ β-fat convex objects to cover it. Now consider the (α, β)-covered polyhedron in Figure 3.4.
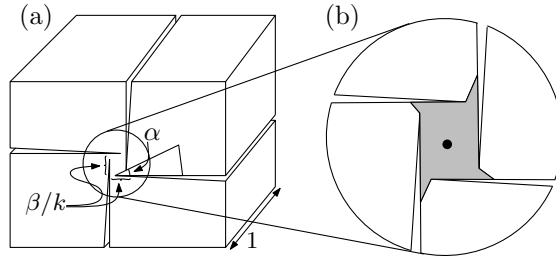
Figure 3.4 (a) An (α, β)-covered polyhedron with fat faces whose interior cannot be covered by a bounded number of fat tetrahedra. (b) The part of the polyhedron seen by a point in the center. Note that the polyhedron is constructed so that a good tetrahedron just fits at the points on the boundary inside the central "tube".

The essential feature of the construction in Figure 3.4 is that from any point p along the long axis of the tube, one cannot see much outside the tube. Thus any convex object inside P that contains p must stay mainly within the tube, and the tube basically acts as a rectangular box of size $1 \times (\beta/k) \times (\beta/k)$. Hence, $\Omega(k)$ β-fat tetrahedra are required in any convex covering of the polyhedron. We obtain the following result.

Theorem 3.5 *There are (α, β)-covered (and, hence, locally-fat) polyhedra with n vertices and convex fat faces, such that the number of objects used in any covering by fat convex objects cannot be bounded as a function of n. Furthermore, for any given $\beta > 0$ there are local polyhedra for which no convex covering with β-fat tetrahedra exists.*

3.3   Covering the boundary

In the previous section we have seen that the number of fat convex objects needed to cover the interior of a fat non-convex polyhedron P cannot be bounded as a function of n. In this section we show that we can do better if we only wish to cover the boundary of P. Unfortunately, this only holds when P is (α, β)-covered; when P is locally fat, we may still need an arbitrarily large number of fat convex objects to cover its boundary.

Recall that for each point p on the boundary of an (α, β)-covered polyhedron P, there is

a good tetrahedron $T_p \subset P$ with one vertex at p, that is, a tetrahedron that is α-fat and has diameter $\beta \cdot diam(P)$. We first observe that we can actually replace T_p by a canonical tetrahedron, as made precise in the following lemma.

Lemma 3.6 *Let P be an (α, β)-covered polyhedron. There exists a set $\mathcal{C}$ of $O(1/\alpha)$ canonical tetrahedra that are $\Omega(\alpha)$-fat and have diameter $\Omega(\beta \cdot diam(P))$ with the following property: for any point $p \in \partial P$, there is a translated copy T'_p of a canonical tetrahedron that is contained in P and has p as a vertex.*

Proof. Cover the boundary of the unit sphere S in a grid-like fashion by $O(1/\alpha)$ triangular surface patches, each of area roughly $c\alpha$, for a suitably small constant c as in Figure 3.5(a). For each triangular patch, define a canonical tetrahedron that has the origin as one of its vertices, and that has edges going through the vertices of the patch—see Figure 3.5(b). Scale the resulting set of tetrahedra appropriately, thus giving the set $\mathcal{C}$. Now consider a good tetrahedron p. Place (a suitably scaled copy) of the sphere S with its center at p. T_p will intersect S in a fat region R of area α. Hence, by choosing c appropriately we can ensure that R contains one of the triangular patches. This implies we can select a tetrahedron T'_p from $\mathcal{C}$ with the required properties. $\square$

Now we can prove that we can cover the boundary of an (α, β)-covered polyhedron with

(a) (b)
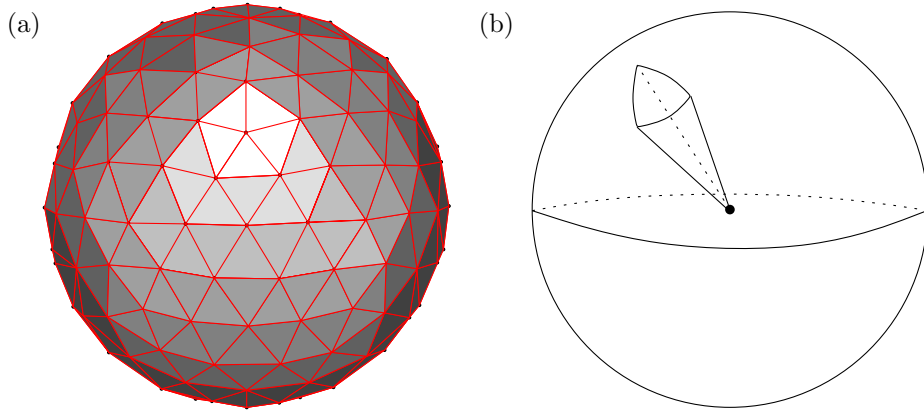
Figure 3.5 (a) A canonical tetrahedron defined by a triangular patch on a sphere. (b) A sphere with a triangular grid.

a bounded number of fat convex objects.

Theorem 3.7 *The boundary of an (α, β)-covered polyhedron with complexity n can be covered by $O(n^2 \log n)$ convex, fat, constant-complexity polyhedra.*

Proof. Let $\mathcal{C}$ be the set of canonical tetrahedra defined in Lemma 3.6. Fix a canonical tetrahedron $T \in \mathcal{C}$. Note that when we put a translated copy of T at some point $p \in \partial P$

according to Lemma 3.6, we always put the same vertex, v, at p. (Namely, the vertex coinciding with the origin before the translation.) For a face f of P, let $f(T) \subset f$ be the subset of points p on f such that we can place T with its designated vertex v at p in such a way that T is contained in P. The region $f(T)$ is polygonal. We triangulate $f(T)$, and for each triangle t in this triangulation, we define a convex polyhedron by taking the union of all the translated copies of T that have $v \in t$. By doing this for all faces f, we get a collection C_T of convex polyhedra that together cover $\bigcup_f f(T)$.

We claim that every convex object $o \in C_T$ is fat. This follows from the fact that T is fat and that T cannot be much smaller than t. Indeed, $diam(T) = \Omega(\beta \cdot diam(P)) = \Omega(\beta \cdot diam(t))$.

Next, we claim that $|C_T| = O(n^2 \log n)$. This follows directly from the fact that the complexity of $\bigcup_f f(T)$ is upper bounded by the complexity of the *free space* of T, when it is translated amidst the faces of P. Aronov and Sharir [12] showed that this free space has complexity $O(n^2 \log n)$.

Finally, we observe that $\bigcup_{T \in \mathcal{C}} \bigcup_f f(T) = \partial P$ by Lemma 3.6. In other words, the convex objects in the set $\bigcup_{T \in \mathcal{C}} C_T$ together cover the boundary of P. □

Theorem 3.7 implies that the boundary of a constant-complexity (α, β)-covered polyhedron P can be covered by a constant number of fat objects. Unfortunately, the number of convex objects used in the boundary covering grows quadratically in the complexity of P. If P has convex fat faces that are roughly the same size, then the number of convex fat objects required to cover the boundary reduces to linear.

Theorem 3.8 *Let P be an (α, β)-covered polyhedron with convex β'-fat faces. Further, let there be a constant c where, for any two faces f_1 and f_2 of P, $diam(f_1) \leq c \cdot diam(f_2)$. Then the boundary of P can be covered by $O(n)$ convex, fat, constant-complexity polyhedra.*

Proof. The proof is very similar to the proof of Theorem 3.7, with one simple change, namely that we shrink the canonical tetrahedra such that their diameter is roughly the same as the size of the faces. Note that the sets C_T still contain fat objects. It remains to argue that each set C_T has size $O(n)$.

To this end, recall from Lemma 3.2 that the set of faces of a fat polyhedron with fat faces has low density. Thus the free space of a canonical tetrahedron T amidst the faces of P is the free space of a translating tetrahedron T in a low density environment, whose obstacles (the faces) are not much smaller than T. Van der Stappen [96] has shown that such a free space has $O(n)$ complexity. □

We claim that any covering of the boundary of an (α, β)-covered polyhedron by fat convex objects requires $\Omega(n^2)$ pieces. To show this, we slightly modify our version of Chazelle's polyhedron from the previous section. In particular, we replace the edges of the needles that contain the segments in the set L by long and thin rectangular facets. The resulting polyhedron is still (α, β)-covered, and it requires $\Omega(n^2)$ fat convex polyhedra to cover the newly introduced facets.

Theorem 3.9 *There are constants $\alpha > 0$ and $\beta > 0$ such that there are (α, β)-covered polyhedra P for which any decomposition of ∂P into fat convex polyhedra requires $\Omega(n^2)$ pieces.*

The number of fat convex polyhedra necessary to cover the boundary of a polyhedron P that is not (α, β)-covered can not be bounded as a function of n. To see this, we make a simple modification to the polyhedron of Figure 3.4. We reduce the gaps that separate the interior "tube" from the rest of P to some arbitrarily small constant ε. This forces any fat convex polyhedron that covers the part of the boundary of the polyhedron inside the tube to be inside the tube. Now for any k, we can reduce the width and height of the tube until its boundary requires more than k fat convex polyhedra to be covered. This example remains locally fat with fat convex faces and it is a local polyhedron. Note that P is no longer (α, β)-covered: reducing the gaps that separate the tube from the rest of the polyhedron causes the points on the boundary inside the tube to no longer have a good tetrahedron.

Theorem 3.10 *For any given k, there exist locally-γ-fat polyhedra for some absolute constant γ with faces that are β fat for some absolute constant β which require at least k fat convex polyhedra to cover their boundaries. These polyhedra are also local polyhedra.*

3.3.1 Boundary covering by towers

In Chapter 4, we describe a data structure for ray shooting in a set S of fat polyhedra. This result uses a covering of the boundaries of the polyhedra in S by so-called *towers*. Here we first describe how to obtain such a covering for convex fat polyhedra. Following that, we extend the covering to (α, β)-covered polyhedra.

We first show that any β-fat convex object o admits two concentric cubes, one containing o and one contained in o, whose size ratio is bounded by a function of β only. For a cube C, define $\operatorname{size}(C)$ to be the edge length of C.

Lemma 3.11 *Let $\sigma := \lceil 54\sqrt{3}/\beta \rceil$. For any convex β-fat object o in $\mathbb{R}^3$, there exist concentric axis-aligned cubes $C^-(o)$ and $C^+(o)$ with $C^-(o) \subseteq o \subseteq C^+(o)$ such that*

$$\frac{\operatorname{size}(C^+(o))}{\operatorname{size}(C^-(o))} = \sigma .$$

Proof. Let $\rho = \rho(o)$ be the radius of the smallest enclosing ball of o. From the results in Section 3.2.1 of Van der Stappen's thesis [96], it follows that o contains an axis-aligned cube $C^-(o)$ with edge length $2\beta\rho/(27\sqrt{3})$. Let p be the center of such a cube. Observe that p is at distance at most ρ from the center of the minimum enclosing ball of o. Let $C^+(o)$ be the axis-aligned cube with edge length 4ρ and center p. Then $o \subseteq C^+(o)$ since the ball centered at p with radius 2ρ clearly contains o and $C^+(o)$ is a bounding box for

that ball. Therefore, we have

$$\frac{\text{size}(C^+(o))}{\text{size}(C^-(o))} = \sigma \,.$$

<div align="right">□</div>

Next we define the canonical directions that we will use in our decomposition. Let C^+ and C^- be two concentric axis-aligned cubes such that $\frac{\text{size}(C^+)}{\text{size}(C^-)} = \sigma$, where σ is defined as in Lemma 3.11; refer to Fig. 3.6(a). Since σ is an integer, we can partition each face
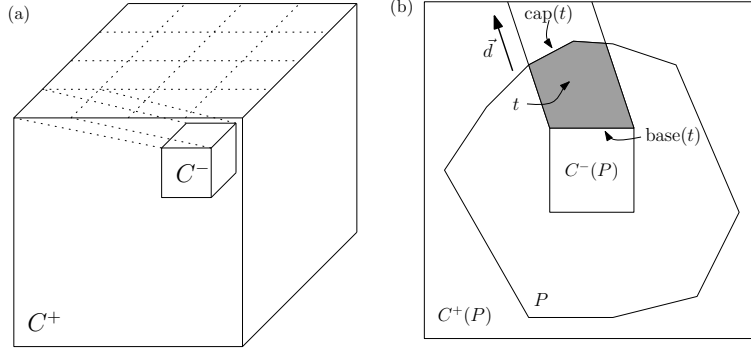
Figure 3.6 (a) Swept volume defining a tower. (b) Two-dimensional analogue of a tower t.

of C^+ into σ^2 squares of the same size as the facets of C^-. We use this to define a set $\mathcal{D}$ of $O(1/\beta^2)$ canonical directions, as follows. For each square s on the top facet of C^+, we add to $\mathcal{D}$ the direction in which the top facet of C^- must be translated to make it coincide with s. The remaining five facets of C^+ are treated similarly. The resulting set $\mathcal{D}$ of canonical directions[2] has size $6\sigma^2 = O(1/\beta^2)$.

Finally, we define the towers. A *tower in the direction* $\vec{d} \in \mathcal{D}$ is a convex polyhedron t with the following properties:

(i) One of the facets of t is an axis-parallel square; this facet is called the *base* of t, denoted by $\text{base}(t)$. We require that the orientation of the base—whether it is parallel to the xy-plane, to the xz-plane, or to the yz-plane—be uniquely determined by the direction $\vec{d}$. Hence, all towers in a given direction $\vec{d}$ have parallel bases.

(ii) The remaining facets of t form a terrain in direction $\vec{d}$, that is, any line parallel to $\vec{d}$ and intersecting the base intersects the remaining facets either in a single point or in a line segment. We call the union of these remaining facets, excluding facets parallel to $\vec{d}$, the *cap* of the tower, denoted $\text{cap}(t)$.

---

[2]In fact, some of the directions defined for, say, the top facet of C^+ are identical to a direction defined for a side facet. It will be convenient to treat these directions as different.

Let P be a β-fat convex polyhedron. The decomposition of P is performed in a manner similar to the way we constructed the canonical directions. Let $C^-(P)$ and $C^+(P)$ be cubes with the properties given in Lemma 3.11. Partition each facet of $C^+(P)$ into σ^2 equal-sized squares. For each such square s we construct a tower by sweeping s towards the corresponding facet of $C^-(P)$, and taking the intersection of the swept volume and the polyhedron P—see Fig. 3.6(b) for an illustration. This way we obtain for each polyhedron P one tower for each of the $|\mathcal{D}|$ canonical directions. We denote the set of towers constructed for P by $T(P)$. The union of the towers in $T(P)$ is contained in P; the boundary of this union consists of the boundaries of P and of $C^-(P)$.

Since $|\mathcal{D}|$ is $O(1/\beta^2)$, our construction leads to the following theorem:

Theorem 3.12 *The boundary of a β-fat convex polyhedron can be covered by $O(1/(\beta)^2)$ towers.*

The natural generalization of towers to non-convex polyhedra is to allow more than one set of towers to be present inside a polyhedron at a time. A single set of towers generated by one pair of cubes would then not need to cover the entire boundary of the polyhedron— see Figure 3.7. As long as all of the points of the boundary of the polyhedron are covered by some tower, the results from Chapter 4 hold. We could apply the results from the previous section to do this: cover the boundary of the (α, β)-covered polyhedron P by fat convex polyhedra and then cover the boundary of those polyhedra by towers using the method described above. Unfortunately, this is not very efficient, since the boundary covering presented in the previous section uses $O(n^2 \log n)$ convex polyhedra. Therefore we describe a direct method to cover the boundary by towers. Our method only uses a constant number of towers per polyhedron.
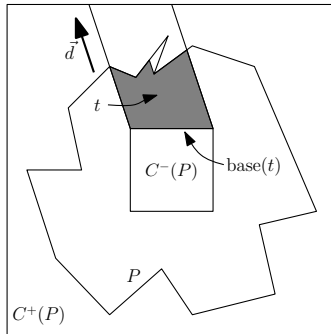

Figure 3.7 A tower in a non-convex polyhedron.

Next we explain how to get a set of towers covering the boundary of an (α, β)-covered polyhedron P. Recall that for every point $p \in \partial P$ there is a good tetrahedron T_p, that is, a tetrahedron that stays completely within P that is α-fat, has diameter $\beta \cdot diam(P)$, and has p as a vertex.

Lemma 3.13 *There is a set of $O(1/(\alpha\beta)^3)$ axis-aligned congruent cubes of edge length $\Omega(\alpha\beta \cdot diam(P))$ such that the good tetrahedron T_p of every point $p \in \partial P$ contains at least one such cube.*

Proof. Consider a good tetrahedron T_p. Since it is α-fat and has diameter $\beta \cdot diam(P)$, it contains a ball B_p of radius $\rho = \Omega(\alpha\beta \cdot diam(P))$. Halve the radius of this ball, while keeping its center at the same position, and let B_p^* denote the resulting ball. Let C be a bounding cube of P. If we put a sufficiently fine grid inside C, then B_p^* must contain at least one grid point. Since B_p^* has radius $\Omega(\alpha\beta \cdot diam(P))$, and C has edge length at most $diam(P)$, it suffices to put a grid with $O(1/(\alpha\beta)^3)$ grid points [33].

For each grid point q inside P, put a cube C_q centered at q with edge length $\rho/2$. If $q \in B_p^*$, then $C_q \subset B_p \subset T_p$. Since there is a grid point q inside every B_p^*, this implies we have a cube C_q inside every T_p. □

We now have a collection $\mathcal{C}$ of $O(1/(\alpha\beta)^3) = O(1)$ cubes of size $\Omega(\alpha\beta \cdot diam(P))$. Next we construct a cube C^+ of size $c \cdot diam(P)$ where c is a constant such that $P \subset C^+$ whenever the center of C^+ is in P. Finally, we construct towers for each cube $C^- \in \mathcal{C}$, by placing C^+ concentric with C^- and using the approach described above. Clearly, this gives us a set of $O(1)$ towers in total. Note that if $C^- \subset T_p$, then one of the towers created for C^- covers p.

Theorem 3.14 $O(1/(\alpha\beta)^5)$ *towers are sufficient to cover the boundary of an (α, β)-covered polyhedron.*

Proof. We already noted that the number of towers in our construction is $O(1)$. (More precisely, it is $O(1/(\alpha\beta)^5)$, since we have $O(1/(\alpha\beta)^3)$ cubes in $\mathcal{C}$, and for each cube we generate $O(1/(\alpha\beta)^2)$ towers.) Moreover, each point $p \in \partial P$ is covered, because $C^- \subset T_p$ for at least one $C^- \in \mathcal{C}$. □

Note that construction of the towers in Theorem 3.14 is very similar to the construction of the guarding set from Theorem 2.3. In fact, the existence of a guarding set (with the extra property that the guards are a sufficient distance from the boundary of the polyhedron) is a sufficient condition for the construction of a set of towers.

By slightly modifying the example from Theorem 3.10, we see that the number of towers necessary to cover the boundary of a polyhedron P that is not (α, β)-covered can not be bounded. Recall that we modified Figure 3.4 so that the "tube" in the middle of the polyhedron was very skinny and had arbitrarily small gaps to the rest of the polyhedron. If we further modify the polyhedron so that the tube does not have its long axis parallel to any of the directions from $\mathcal{D}$, then, given k, we can force ∂P to require more than k towers to be covered[3]. This polyhedron remains locally-fat with fat faces and local.

---

[3]One could, of course, "cheat" and define $\mathcal{D}$ so that it contained a direction parallel to the tube. However, as we have noted, $\mathcal{D}$ should not depend on any specific polyhedron, as this would render the decomposition useless when applied to multiple polyhedra.

Theorem 3.15 *For any given k, there exist locally-γ-fat polyhedra for some absolute constant γ with faces that are β-fat for some absolute constant β which require at least k towers to cover their boundaries. These polyhedra are also local polyhedra.*

3.4   Conclusion

We studied decompositions and boundary coverings of fat polyhedra. Our bounds on the number of objects needed in the decomposition (or covering) are tight, except for the bound on the number of convex fat polyhedra needed to cover the boundary of an (α, β)-covered object. In particular, there is still a large gap for the case that the facets of the polyhedron are also fat. It would be interesting to get tight bounds for this case.

Ray shooting and range searching

4.1 Introduction

The *ray-shooting problem* is to preprocess a set $\mathcal{P}$ of objects in $\mathbb{R}^d$ for the following queries: what is the first object (if any) in $\mathcal{P}$ hit by a query ray? Such queries form the basis of ray-tracing algorithms, they can be used to approximate form factors in radiosity methods, and they can be used for other visibility problems. Since ray shooting is an integral part of many graphics applications, it should not be surprising that it has received much attention, both in computer graphics and computational geometry. In fact, after the range-searching problem it is probably one of the most widely studied data-structuring questions in computational geometry. The survey by Pellegrini [79] and the book by De Berg [28] discuss several of the data structures that have been developed within computational geometry for the ray-shooting problem (although there is also much work that is not covered there, for example, research concerning ray shooting in two-dimensional scenes, or in d-dimensional space, for $d > 3$). In the discussion below, we will restrict our attention to results on ray shooting in $\mathbb{R}^3$.

In the first part of the discussion below, we examine ray shooting when the ray is restricted to travel in a single direction. We assume without loss of generality that this direction is parallel to the z-axis and thus call this type of problem *vertical ray shooting*. Afterwards we remove the restriction and give a data structure for answering ray-shooting queries where the query ray can have any direction.

Related work. Data structures for vertical ray-shooting queries among sets of arbitrary triangles in $\mathbb{R}^3$ have rather high storage requirements. When a $O(\log n)$ query time is desired, the best-known data structure needs $O(n^3)$ space [28]. Space can be traded for query time: for any m satisfying $n \leqslant m \leqslant n^3$, a data structure can be constructed that uses $O(m^{1+\varepsilon})$ space that allows vertical-ray-shooting queries that take $O(n^{1+\varepsilon}/m^{1/3})$ time [28].

Given the prominence of the ray-shooting problem in computational geometry it is not surprising that ray shooting has already been studied from the perspective of realistic input models. In particular, the vertical-ray-shooting problem has been studied for fat convex polyhedra. For this case Katz [58] presented a data structure that uses $O(n \log^3 n)$ storage and has $O(\log^4 n)$ query time. (In fact, Katz's solution works for polygons whose projections onto the xy-plane are fat, but it is not difficult to see that it works for fat 3D polytopes as well.) Using the techniques of Efrat *et al.* [47] it is possible to improve the storage bound to $O(n \log^2 n)$ and the query time to $O(\log^3 n)$ [59]. Recently De Berg [31] presented a structure with $O(\log^2 n)$ query time; his structure uses $O(n \log^3 n (\log \log n)^2)$ storage.

Similarly, in the case of ray-shooting in arbitrary directions, the results achieved for non-fat objects require a lot of storage. If the set $\mathcal{P}$ consists of n arbitrary triangles, the best known structures with $O(\log n)$ query time use $O(n^{4+\epsilon})$ storage [28, 78], whereas the best structures with near-linear storage have roughly $O(n^{3/4})$ query time [7]. More generally, for any m with $n < m < n^4$, one can obtain $O((n/m^{1/4}) \log n)$ query time using $O(m^{1+\varepsilon})$ storage [7]. Better results have been obtained for several special cases. When the set $\mathcal{P}$ is a collection of n axis-parallel boxes, one can achieve $O(\log n)$ query time with a structure using $O(n^{2+\varepsilon})$ storage [28]. Again, a trade-off between query time and storage is possible: with $O(m^{1+\varepsilon})$ storage, for any m with $n < m < n^2$, one can achieve $O((n/\sqrt{m}) \log n)$ query time. If $\mathcal{P}$ is a set of n balls, then it is possible to obtain $O(n^{2/3})$ query time with $O(n^{1+\varepsilon})$ storage [90], or $O(n^\varepsilon)$ query time with $O(n^{3+\varepsilon})$ storage [72].

Both axis-parallel boxes and balls are very special objects, and in most graphics applications the scene will not consist of such objects. The question thus becomes: is it possible to improve upon the ray-shooting bounds for classes of objects that are more general than axis-parallel boxes or spheres? This is the problem we tackle in this chapter. More precisely, we study the ray-shooting problem for convex polyhedra that are *fat*—see Chapter 1 for a formal definition.

For the case of *horizontal* fat triangles, there is a structure that uses $O(n^{2+\varepsilon})$ storage and has $O(\log n)$ query time [28], but the restriction to horizontal triangles is quite severe. Another related result is by Mitchell *et al.* [69]. In their solution, the amount of storage depends on the so-called *simple-cover complexity* of the scene, and the query time depends on the simple-cover complexity of the query ray. Unfortunately the simple-cover complexity of the ray—and, hence, the worst-case query time—can be $\Theta(n)$ for fat objects. In fact, this can happen even when the input is a set of cubes. The first (and so far only, as far as we know) result that works for arbitrary rays and rather arbitrary fat objects

was recently obtained by Sharir and Shaul [89]. They present a data structure for ray shooting in a collection of fat triangles that has $O(n^{2/3+\varepsilon})$ query time and uses $O(n^{1+\varepsilon})$ storage. Curiously, their method does not improve the known bounds at the other end of the query-time–storage spectrum, so for logarithmic-time queries the best known storage bound is still $O(n^{4+\varepsilon})$.

Our results for ray shooting.   First, we present a new data structure for vertical ray shooting in a collection of n convex constant-complexity fat polyhedra[1] in $\mathbb{R}^3$. Our data structure uses $O((1/\beta)n \log^2 n)$ storage and has $O((1/\beta^2) \log^2 n)$ query time. Compared to Katz's structure [59] it has a better query time (while the storage is the same) and compared to the De Berg's structure [31] it has a better storage bound (while keeping the same query time).

We then present a data structure for ray shooting with arbitrary rays in a collection $\mathcal{P}$ of (not necessarily disjoint) convex fat polyhedra with n vertices in total. Our structure requires $O(n^{2+\varepsilon})$ storage and has query time $O(\log^2 n)$. A trade-off between storage and query time is also possible: for any m with $n < m < n^2$, we can construct a structure that uses $O(m^{1+\varepsilon})$ storage and has $O((n/\sqrt{m}) \log^2 n)$ query time. Compared to the bounds obtained by Sharir and Shaul there are two differences: our query time for near-linear storage is $O(\sqrt{n} \log^2 n)$ while the query time of Sharir and Shaul is $O(n^{2/3+\varepsilon})$, and we get improved bounds at the other end of the spectrum while Sharir and Shaul will need $O(n^{4+\varepsilon})$ storage for $O(\log n)$ query time. Of course, the two settings are not the same: Sharir and Shaul consider fat triangles, whereas we consider fat polyhedra. Indeed, our solution makes crucial use of the fact that fat polyhedra have a relatively large volume. Note that neither setting implies the other: fat triangles need not form fat polyhedra, and fat polyhedra do not necessarily have fat facets. (For example, a polyhedral model of a cylinder is likely to contain long and thin facets.)

Results on range searching.   The intersection-searching problem is to preprocess a set of objects such that all objects intersecting a query range can be reported efficiently. If the objects are points, then the problem becomes the standard range-searching problem: report all points inside a query range. Range searching and intersection searching have been studied extensively—see for example the surveys by Agarwal [2] and Agarwal and Erickson [4]. For intersection-searching with a query simplex in a set of simplices in $\mathbb{R}^3$ one can, for any m with $n < m < n^4$, obtain $O((n/m^{1/4}) \log n + k)$ query time using $O(m^{1+\varepsilon})$ storage, where k is the number of reported simplices. Using our technique of covering with towers, we show that simplex-intersection queries can be answered more efficiently if the objects are fat convex polyhedra of constant complexity: for any m with $n < m < n^3$, we obtain $O((n/m^{1/3}) \log n + k)$ query time with a structure using $O(m^{1+\varepsilon})$ storage. This matches the best known bounds for simplex range searching in point sets in $\mathbb{R}^3$. So far, no general results were known for intersection searching among

---

[1]Though results are presented in terms of fat polyhedra, our results for vertical ray shooting also work without modification in the more general setting of objects that project to fat polygons.

fat polyhedra that were better than those for arbitrary polyhedra—there has been work on intersection searching in fat objects [29, 75, 85] but these results require the query range to be not too large compared to the input objects and they require the input objects to be disjoint.

4.2 Preliminaries

Basic properties of fat objects.   We need a result that will allow us to stab a set of relatively large fat objects that all intersect some region R using only a few points. Similar results have been proved earlier [33].

Lemma 4.1 *Let R be a bounded region in the plane, and let c be a constant that satisfies $0 < c \leq 1$. Then there is a collection Q of $O(1/(c\beta)^2)$ points with the following property: any β-fat object o with $\mathrm{size}(o) \geq c \cdot \mathrm{size}(R)$ that intersects R contains at least one point from Q.*

Proof.   Let U be a bounding square of R, and let $\widehat{U}$ be the concentric square twice the size of U. Consider a β-fat object o with $\mathrm{size}(o) \geq c \cdot \mathrm{size}(R)$ that intersects R. Then $area(o \cap \widehat{U}) \geq c'c\beta \cdot area(\widehat{U})$ for a suitable constant c' (*cf.* Van der Stappen's thesis [96], Theorem 2.9). Hence, a regular grid on $\widehat{U}$ with $\lceil M \rceil^2$ cells, where $M = 2/(c'c\beta)$, must have at least one grid point inside P, because the area of any convex object missing all grid points is less than $2 \cdot area(\widehat{U})/M$. $\square$

The following lemma was proved by Van Kreveld [98] for non-convex polygons. However, his definition of fatness is different from ours and is not obviously compatible. Therefore we have proved it independently using our definition. The proof is rather long, so it can be found in the appendix to this chapter. In the lemma below, an α-fat triangle refers to a triangle all of whose angles are at least α. (Such a triangle is α'-fat according to Definition 1.1 for some $\alpha' = \Omega(\alpha)$.)

Lemma 4.2 *Let P be a β-fat convex polygon with n vertices. There is a set T of α-fat triangles that cover P where $|T| = O(n)$ and $\alpha = \Theta(\beta)$.*

Ray shooting and parametric search.   Agarwal and Matoušek [6] described a technique that reduces the ray-shooting problem on a set $\mathcal{P}$ of objects to the segment-emptiness problem, *i.e.*, testing whether a query segment intersects any of the objects in $\mathcal{P}$. Since then their technique has been used in several papers dealing with ray shooting [72, 89, 90]. We will also use this technique.

Theorem 4.3 (Agarwal and Matoušek [6]) *Let $\mathcal{P}$ be a set of objects. Suppose that we have a data structure Σ supporting segment-emptiness queries with respect to $\mathcal{P}$, for*

arbitrary segments. Let A_p be a parallel algorithm for answering a segment-emptiness query, which uses at most p processors and runs in at most T_A parallel steps, and such that for a query segment ox, the computation of A_p uses the information about x only in deciding the signs of certain fixed-degree polynomials in the coordinates of x. Let B be another version of the segment-emptiness algorithm, which can report an object $P_i \in \mathcal{P}$ containing the endpoint of the query segment, provided that the segment is otherwise empty, and let T_B be the maximum running time of B. Then the ray-shooting problem for rays in R can be solved using the same data structure Σ, in time $O(pT_A + T_B T_A \log p)$.

Finally, we will need the following result.

Theorem 4.4 (Chazelle *et al.* [22]) *Let L be a set of n lines in 3-space. For any m with $n < m < n^2$, we can preprocess the set L using $O(m^{1+\varepsilon})$ time and storage so that we can detect in $O((n/\sqrt{m}) \log n)$ time whether a query line ℓ lies above all the lines in L.*

4.3 Vertical ray shooting

Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a collection of n constant-complexity convex β-fat polyhedra that we wish to preprocess for vertical ray shooting. We start by studying the simpler case where all the objects are intersected by a common vertical line. After that we will show how to use this structure to obtain an efficient solution to the general problem.

Agarwal *et al.* [5] already described a data structure for the case where all objects are intersected by a common vertical line and project to triangles. We observe that it is possible to apply fractional cascading to their structure to obtain the following result.

Lemma 4.5 *Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set of n disjoint convex constant-complexity β-fat polyhedra that are all stabbed by a vertical line ℓ and that all project to fat triangles. Then there is a data structure such that vertical ray shooting queries on $\mathcal{P}$ can be answered in $O(\log n)$ time. The structure uses $O((1/\beta)n \log n)$ storage and it can be built in $O((1/\beta)n \log n)$ time.*

Proof. As stated above, all we need to do is apply fractional cascading to the structure of Agarwal *et al.* [5]. For completeness, we briefly describe their solution and explain how to apply fractional cascading.

The structure is a balanced binary tree $\mathcal{T}$ with the objects in the leaves, sorted by their position along ℓ; the lowest object is in the leftmost leaf, the second lowest object in the next leaf, and so on. Since the objects are non-intersecting and convex, this ordering is well-defined.

For a node ν, let $\mathcal{P}(\nu)$ denote the set of objects stored in the leaves of the subtree rooted at ν. At each non-leaf node ν of $\mathcal{T}$, we store the union $U(\nu)$ of the vertical projections of the objects in $\mathcal{P}(\nu)$. We preprocess $U(\nu)$ for point-enclosure queries—that is, queries

that ask whether a point q in the xy-plane lies inside $U(\nu)$— as follows. Let p_ℓ be the point where ℓ intersects the xy-plane. Then all projections contain p_ℓ, and since they are convex $U(\nu)$ is star-shaped with p_ℓ in the kernel. Hence, if we partition the plane into cones by drawing half-lines from p_ℓ through all breakpoints on the boundary of $U(\nu)$, then a point-enclosure query can be answered in $O(1)$ time after we have determined in which cone the query point lies.

To perform a query with a vertical ray starting above all objects, we walk down the tree as follows. Suppose we reach a node ν. When the point p where the ray hits the xy-plane lies inside the union of the right child of ν we proceed to the right child, otherwise we proceed to the left child. The leaf we reach must store the first object hit (if any object is hit at all). When the starting point of the ray does not lie above all objects, things are more complicated. However, Agarwal *et al.* have shown that a query can still be answered by walking down the tree, although now up to four nodes per level may be visited. In any case, we visit $O(\log n)$ nodes in total, and at each node we have to do a point-enclosure query. As explained above, a point-enclosure query can be answered in $O(1)$ time after we have determined in which cone the query point lies. Finding the right cone can be done in $O(\log n)$ time by binary search, but this can be made faster: using fractional cascading [24, 25] finding the cones can be done in $O(1)$ time, except for the search at the root. Since the application of fractional cascading is completely standard in this setting we omit further details.

To build the structure, we sort the objects along ℓ in $O(n \log n)$ time, and then we construct the unions to be stored at each node in a bottom-up fashion. Hence, when we arrive at a node ν, we have to merge the two unions of the children of ν. Because the unions are star-shaped with respect to the same point, computing the union of these unions boils down to merging the two circularly sorted lists of breakpoints. Hence, this can be done in linear time. The total time to construct all unions is therefore equal to the total size of the data structure, which is $\sum_\nu O(|\mathcal{P}(\nu)|) = O(n \log n)$. Adding the additional pointers for the fractional cascading does not increase the preprocessing time or the amount of storage asymptotically. □

Now consider the general case, where the objects in $\mathcal{P}$ are not necessarily stabbed by a vertical line. We can cover each object by $O(1)$ subobjects whose projections are fat triangles using the technique of Lemma 4.2, so we can assume without loss of generality that all objects project to fat triangles. We shall make use of two-dimensional BAR-trees. Recall from Chapter 1 that *BAR-trees* (or *balanced aspect ratio trees*) are a special type of BSP trees for point sets. A BSP tree $\mathcal{T}$ for a set S of points contained in some bounding square σ is a recursive partitioning of σ by splitting lines, such that the final cells of the subdivision do not contain any points in their interior. Each node ν of $\mathcal{T}$ corresponds to a region $region(\nu) \subset \sigma$, which is defined recursively as follows. The region $region(root(\mathcal{T}))$ is the whole square σ. Furthermore, if the splitting line stored at a node ν is $\ell(\nu)$, then $region(leftchild(\nu)) = region(\nu) \cap \ell(\nu)^-$, where $\ell(\nu)^-$ is the half-plane below $\ell(\nu)$. Similarly, $region(rightchild(\nu)) = region(\nu) \cap \ell(\nu)^+$, where

$\ell(\nu)^+$ is the half-plane above $\ell(\nu)$.

The special properties of BAR-trees that are relevant for us are the following. First, a BAR-tree on a set S of points has depth $O(\log|S|)$ and size $O(|S|)$. Furthermore, the regions corresponding to a node in a BAR-tree have bounded aspect ratio, which implies they are c-fat for some constant c. It has been shown by De Berg and Streppel [40] that this implies the following.

Lemma 4.6 (De Berg and Streppel [40]) *Let o be a β-fat object. Then there is a set $G(o)$ of 12 points—we call these points* guards—*such that for any BAR-tree region R that intersects o but does not contain a guard from $G(o)$ in its interior we have* $\mathrm{size}(o) = \Omega(\mathrm{size}(R))$.

De Berg and Streppel [40] used this to design a so-called object BAR-tree: this is a BAR-tree that can be used for approximate range searching in a set of objects rather than in a point set. Our ray-shooting structure combines BAR-trees and the lemma above in a different way, as described next.

Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set of n constant-complexity β-fat polyhedra. Let $G_i = G(\mathrm{proj}(P_i))$ be a set of guards for the projection of P_i, as in Lemma 4.6. Our data structure for vertical ray shooting on $\mathcal{P}$ is defined as follows.

- The main tree $\mathcal{T}$ is a BAR-tree for the set $G = G_1 \cup \cdots \cup G_n$.

- Let ν be a node in $\mathcal{T}$. We say that an object P_i is *large* at ν if (i) $\mathrm{proj}(P_i)$ intersects $region(\nu)$, and (ii) $region(parent(\nu))$ contains a guard from G_i in its interior but $region(\nu)$ does not. Note that Lemma 4.6 implies that $\mathrm{size}(P_i) = \Omega(\mathrm{size}(region(\nu)))$ if P_i is large at ν. Let $\mathcal{P}(\nu) \subset \mathcal{P}$ be the subset of objects that are large at ν.

  Let $Q(\nu)$ be a set of points such that for any $P_i \in \mathcal{P}(\nu)$, there is a point $q \in Q(\nu)$ with $q \in \mathrm{proj}(P_i)$. By Lemma 4.1 there exists such a set $Q(\nu)$ of size $O(1/\beta^2)$. Assign each object $P_i \in \mathcal{P}(\nu)$ arbitrarily to one of the points $q \in Q(\nu)$ contained in its projection. Let $\mathcal{P}(q)$ denote the set of objects assigned to q. We store the set $\mathcal{P}(q)$ in a data structure $\mathcal{T}(q)$ for vertical ray shooting according to Lemma 4.5. Thus each node ν has $|Q(\nu)|$ associated structures.

Let's first see how to answer a vertical ray-shooting query with this structure.

Lemma 4.7 *A vertical ray-shooting query can be answered in $O((1/\beta^2)\log^2 n)$ time.*

Proof. Let p be the point where the line through the query ray intersects the xy-plane. Search with p down the tree $\mathcal{T}$. At every node ν on the search path, perform a query in the associated structure $\mathcal{T}(q)$ of each $q \in Q(\nu)$. A query thus takes $O(\log n \cdot \log n \cdot (1/\beta^2))$ time—that is, the length of every search path, times the query time in the associated data structures along the search path, times the size of $Q(\nu)$.

To prove the correctness, it suffices to argue that any object P_i whose projection contains p must be large at one of the nodes on the search path of p. To see this, we observe that $region(root(\mathcal{T}))$ contains all guards from G_i while the leaf regions do not contain any guards in their interior. It follows that when we follow the path of p, the object P_i must become large at some node. □

We can now prove our final result on vertical ray shooting.

Theorem 4.8 *Let $\mathcal{P}$ be a collection of n convex disjoint constant-complexity β-fat polyhedra in $\mathbb{R}^3$. Then there is a data structure such that vertical ray shooting queries on $\mathcal{P}$ can be answered in $O((1/\beta^2)\log^2 n)$ time. The structure uses $O((1/\beta)n\log^2 n)$ storage and it can be built in $O((1/\beta)n\log^2 n)$ time.*

Proof. The correctness of the query procedure and the query time have been shown in Lemma 4.7.

It remains to prove the bound on the construction time; the storage bound then follows trivially. Computing the guards for each object takes constant time per object, and constructing the BAR-tree takes $O(n\log n)$ time [44]. We claim that an object P_i is large at $O(\log n)$ nodes. Indeed, any guard is contained in the regions of the nodes on a single path down the tree, and an object can only be large at a node if the parent region contains one of its guards. Hence, $\sum_\nu |\mathcal{P}(\nu)| = O(n\log n)$. We can generate the sets $\mathcal{P}(\nu)$ in $O(n\log n)$ time by filtering the objects down the tree $\mathcal{T}$. The set $Q(\nu)$ can be constructed in $O(|Q(\nu)|)$ time, and associating the objects with the points in $Q(\nu)$ can be done in a brute-force way in $O(|Q(\nu)| \cdot |\mathcal{P}(\nu)|)$. Finally, constructing the associated structures of ν takes time

$$\sum_{q\in Q(\nu)} O((1/\beta)|\mathcal{P}(q)|\log|\mathcal{P}(q)|) \;\; = \;\; O((1/\beta)|\mathcal{P}(\nu)|\log|\mathcal{P}(\nu)|)$$

by Lemma 4.5. Hence, the overall construction time is

$$\sum_\nu O(|\mathcal{P}(\nu)| \cdot (|Q(\nu)| + (1/\beta)\log|\mathcal{P}(\nu)|))$$
$$= \;\; O((1/\beta)n\log^2 n + (1/\beta^2)n\log n)$$
$$= \;\; O((1/\beta)n\log^2 n).$$

□

4.4 A ray-shooting data structure for arbitrary directions

Let $\mathcal{P}$ be the set of either convex fat polyhedra or (α,β)-covered polyhedra that we wish to preprocess for ray-shooting queries with query rays that have arbitrary directions. We

use n to denote the total number of vertices of the polyhedra. Our global strategy is roughly as follows.

We first decompose of the boundary of each polyhedron into a constant number of towers. This is the same decomposition that we mentioned in Chapter 3. Recall that for a convex β-fat polyhedron P, we can cover the boundary P with $O(1/\beta^2)$ towers by Theorem 3.12. By Theorem 3.14, for an (α, β)-covered polyhedron P, we can cover the boundary of P with $O(1/(\alpha\beta)^5)$ towers, where the towers have $O(1/(\alpha\beta)^2)$ canonical directions. Next, we present a data structure to efficiently perform segment-emptiness queries on the towers. Using Agarwal and Matoušek's parametric-search technique mentioned above, we then convert this structure into a structure for ray shooting.

Testing for segment emptiness. Before we describe the data structure for segment-emptiness queries, we describe necessary and sufficient conditions for a segment to intersect a polyhedron P. We treat P as a solid, meaning that a segment s intersects P even if both endpoints of s are inside P.

In the lemma below and in the rest of the chapter, whenever we speak of "above" and "below" when referring to a specific tower, this is always with respect to the canonical direction $\vec{d}$ of that tower. More precisely, we say that an object o is *below* an object o' whenever there exists a directed line with orientation $\vec{d}$ that first intersects o and then o'. A point is inside a tower, for instance, if and only if it is above the base and below the cap. Finally, we use $\mathrm{proj}(o)$ to denote the projection of an object o in direction $\vec{d}$ onto a plane orthogonal to $\vec{d}$.

Lemma 4.9 *A segment $s = \overline{pq}$ intersects a polyhedron $P \in \mathcal{P}$ if and only if one of the following conditions holds:*

1. *p or q is inside P, or*

2. *there is a tower $t \in T(P)$ such that*

   (a) *$\overline{pq}$ intersects $\mathrm{base}(t)$, or*

   (b) *$\overline{pq}$ passes below an edge of $\mathrm{cap}(t)$ and above an edge of $\mathrm{base}(t)$, or*

   (c) *$\overline{pq}$ passes below an edge of $\mathrm{cap}(t)$ and p or q is above $\mathrm{base}(t)$.*

Proof. If one of the conditions is met, s clearly intersects P. Therefore, we will concentrate on the "only if" part of the proof. If s meets P but misses all $t \in T(P)$, condition 1 clearly holds. Suppose s intersects some $t \in T(P)$. Put $b := \mathrm{base}(t)$. Up to exchanging p and q, there are three possible scenarios for p and q with respect to t:

Case (i): $\mathrm{proj}(p)$ *is inside* $\mathrm{proj}(t)$ *but* $\mathrm{proj}(q)$ *is not.* Then p is either above t, below it, or inside it. If p is inside t, then condition 1 is satisfied. If p is below b, then there must be some other point on s that is above b, which implies that s must satisfy

59

condition 2a. Finally, if p is above b but not inside t, then condition 2c is satisfied by the property that t is a terrain and the fact that $\text{proj}(q)$ is not inside $\text{proj}(t)$.

Case (ii): both $\text{proj}(p)$ *and* $\text{proj}(q)$ *are inside* $\text{proj}(t)$. If either p or q are inside t, then condition 1 is satisfied. If p is below b and q is above it (or vice versa), then condition 2a is satisfied. If both p and q are below b, then s can not intersect t. If both p and q are above b but not inside t, then they must both be above $\text{cap}(t)$. Since s intersects t, this implies that condition 2c is satisfied.

Case (iii): neither $\text{proj}(p)$ *nor* $\text{proj}(q)$ *is inside* $\text{proj}(t)$. Now condition 2b must always be satisfied.

$\square$

Lemma 4.9 allows us to treat a segment-emptiness query as the disjunction of several different conditions and test separately for each of these conditions. Developing data structures for each of these conditions is relatively routine; they can be implemented using standard multi-level range-searching data structures. Below we provide some of the details.

Lemma 4.10 *Let $\mathcal{P}$ be a set of β-fat convex polyhedra in $\mathbb{R}^3$ of total complexity n. Given a query segment s, we can detect in $O((n/\beta^2 \sqrt{m}) \log n)$ time whether an endpoint of s is inside a polyhedron of $\mathcal{P}$ using a data structure that requires $O(m^{1+\varepsilon}/\beta^2)$ preprocessing time and storage, for any parameter m with $n < m < n^2$. If the polyhedra are disjoint, this can be improved to $O(n/\beta)$ storage and preprocessing time and $O((1/\beta) \log n)$ query time.*

If $\mathcal{P}$ is a set of n (α, β)-covered polyhedra in $\mathbb{R}^3$ with total complexity n, then the bounds are the same except for the dependence on the fatness constant: in all cases, $O(1/\beta^2)$ is replaced by $O(1/(\alpha\beta)^2)$.

Proof. When the objects in $\mathcal{P}$ may intersect, we treat the towers and the inner cubes of each object separately.

We preprocess the cubes into a three-level segment tree [42]. This tree uses $O(n \log^3 n)$ storage and allows us to check if any of the cubes contains a query point in $O(\log^3 n)$ time. By increasing the degree of the nodes in the segment tree to $O(n^\varepsilon)$, we can reduce the query time to $O(\log n)$ at the cost of using $O(n^{1+\varepsilon})$ storage.

The towers are handled as follows. Consider the collection of towers for a fixed canonical direction. Assume without loss of generality that the bases of the towers are parallel to the xy-plane. A tower t contains a query point q if and only if the following three conditions hold: the projection of q is contained in the projection of the base of t (here the projections are onto the xy-plane and in the canonical direction of the tower), q is above that base, and q is below the plane through the cap facet whose projection contains q. This means we can detect this using a multi-level tree: the first two levels are segment trees on the projections of the bases onto the xy-plane (these are used to select the bases whose projections contain the projection of q), the third level is a binary tree on height (used to

select those bases that are below q), the next three[2] levels are partition trees on the edges of the projected cap facets (to select the cap facets whose projections contain q), and the final level is a structure to test if q is above the upper envelope of the planes through the cap facets. As usual with this type of multi-level data structure, the performance is determined by the worst-case performance of any of the levels. Hence, we get the same bounds as in a two-dimensional partition tree, as stated in the lemma; the extra factor $O(1/\beta^2)$ for convex objects or $O(1/(\alpha\beta)^2)$ is because each of the canonical directions is treated separately.

When the objects in $\mathcal{P}$ are guaranteed not to intersect, we use the so-called *object BAR-tree* designed by De Berg and Streppel [40]. Recall from Chapter 1 that this is a BSP-tree with $O(n)$ nodes and depth $O(\log n)$, such that every leaf region intersects at most $O(1/\beta)$ objects. Therefore, assuming the polyhedra have constant complexity, we can test whether p is inside any of the polyhedra in $\mathcal{P}$ simply by finding the cell containing p in $O(\log n)$ time and then testing whether p is inside any of the polyhedra in the cell. If the polyhedra do not have constant complexity, we apply the Dobkin-Kirkpatrick hierarchy [43] to each polyhedron. In either case, the test takes $O(\log n)$ to determine which cell p is in and $O((1/\beta) \log n)$ to test if p is inside any of the $O(1/\beta)$ polyhedra meeting that cell. □

Lemma 4.11 *Let $\mathcal{P}$ be a set of convex β-fat polyhedra. Assuming there is no endpoint of query segment s inside any polyhedron in $\mathcal{P}$, we can detect whether s intersects any polyhedron in $\mathcal{P}$ using a data structure which requires $O(n^{2+\varepsilon}/\beta^2)$ storage and preprocessing time and has query time $O((\log n)/\beta^2)$. Furthermore, for any m with $n < m < n^2$, we can construct a structure that uses $O(m^{1+\varepsilon}/\beta^2)$ storage and preprocessing time and has $O((n/(\beta^2 \sqrt{m})) \log n)$ query time.*

If $\mathcal{P}$ is a set of (α, β)-covered polyhedra, the dependence on the fatness constant, which is $O(1/\beta^2)$ in the convex case, is replaced by $O(1/(\alpha\beta)^2)$.

Proof. There are three cases to consider, according to Lemma 4.9. We will design a different structure for each of them, and in each case we will need a separate structure for each of the $|\mathcal{D}|$ canonical tower directions. So we fix one of the canonical directions $\vec{d}$, and let $\mathcal{T} = \mathcal{T}(\vec{d})$ be the set of all towers of that direction. Without loss of generality, assume that the base of the towers in $\mathcal{T}$ is horizontal, *i.e.*, parallel to the xy-plane.

Condition 2a: s intersects $\mathrm{base}(t)$ for some tower $t \in \mathcal{T}$: Since $\mathrm{base}(t)$ is an axis-aligned rectangle, a segment s intersects $\mathrm{base}(t)$ if and only if $\ell(s)$, the line through s, intersects $\mathrm{base}(t)$ both in the projection onto the yz-plane and in the projection onto the xz-plane, and the endpoints of s lie on opposite sides of the plane through $\mathrm{base}(t)$. Hence, we can test whether there is an intersected base using a five-level tree: the first two levels are partition trees used to select the bases that intersect

---

[2]We assume without loss of generality that each cap facet is a triangle.
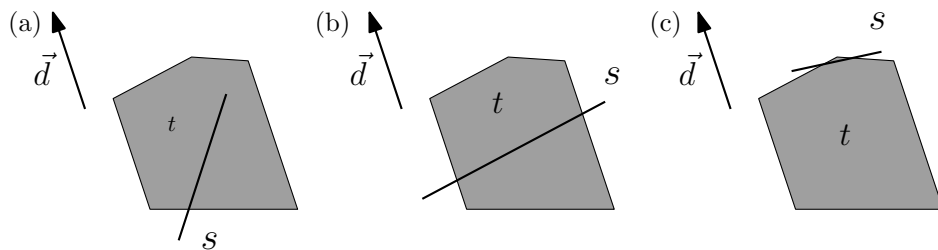
Figure 4.1 (a) Condition 2a. (b) Condition 2b. (c) Condition 2c.

$\ell(s)$ in the projection onto the xz-plane, the next two levels are partition trees used to select of these bases the ones that also intersect $\ell(s)$ in the projection onto the yz-plane, and the last level is a search tree on z-coordinate to test whether s has its endpoints on opposite sides of any of the selected bases.

Condition 2b: s passes above an edge of $\mathrm{base}(t)$ *and below an edge of* $\mathrm{cap}(t)$*, for some $t \in \mathcal{T}$:* This happens if and only if s intersects an edge of $\mathrm{base}(t)$ in the projection onto the xy-plane, is above that edge in the orthogonal projection onto the plane orthogonal to that edge, and is below some edge of $\mathrm{cap}(t)$. Therefore, we can also check condition 2b by using a multi-level structure based on partition trees: the first levels are used to select all towers having a base with an edge that intersects s in the projection onto the xy-plane, the next level is to restrict the selection to towers with base edges below s, the next levels are to select of those towers the cap edges intersecting s in the projection to the xy-plane. It remains to check whether any of the selected cap edges is above s. Since these edges all intersect s in the projection, we can treat them and s as full lines and use the structure from Theorem 4.4.

Condition 2c: s passes below an edge of $\mathrm{cap}(t)$ *and it has an endpoint above* $\mathrm{base}(t)$*, for some $t \in \mathcal{T}$:* We first select all towers having a base below one of the endpoints of s—this can be done using a two-level segment tree storing the projected bases and a binary search tree on the heights of the bases—then we select the towers with a cap edge that is intersected in the projection to the xy-plane, and finally we apply the structure of Theorem 4.4 again.

In all cases, we have described a multi-level data structure with a constant number of levels, where each level is either a two-dimensional partition tree, a segment tree, a binary tree, or (as the final level) the structure of Theorem 4.4. The bounds for such multi-level structure are determined by the worst level, which leads exactly to the bounds stated in the lemma. (To speed up the query from $O(\log^4 n)$ time to $O(\log n)$, when $O(n^{1+\varepsilon})$ storage is used, we employ the standard trick [28]: we choose the branching degree to be $O(n^\varepsilon)$, and we add a point-location structure to identify the correct child to which we must descend in $O(\log n)$ time.)

It is easily checked that, in the worst case, the total complexity of the towers of $\mathcal{T}(\vec{d})$

can be $\Theta(n)$, for each $\vec{d}$. Hence the above estimates for query time and preprocessing time and space must be multiplied by $1/\beta^2$ in the case of convex polyhedra or $1/(\alpha\beta)^2$ in the case of (α, β)-covered polyhedra to account for processing all canonical directions. $\square$

Putting it all together. In order to apply the parametric-search technique described in Theorem 4.3, we must describe parallel algorithms for querying the data structures presented in the previous section. For the object BAR-tree and the Dobkin-Kirkpatrick hierarchies for the polyhedra of non-constant complexity, the parallel query algorithm coincides with the sequential one. In the other structures, we can obtain $O(\log n)$ parallel query time using $O(n/\sqrt{m})$ processors: basically, whenever a search path splits we add another processor. Applying Theorem 4.3 now gives the final result.

Theorem 4.12 *Let $\mathcal{P}$ be a set of β-fat convex polyhedra in $\mathbb{R}^3$ of total complexity n. We can preprocess $\mathcal{P}$ using $O(n^{2+\varepsilon}/\beta^2)$ storage and preprocessing time, such that ray-shooting queries can be answered in $O((\log^2 n)/\beta^2)$ time. Moreover, for any m with $n < m < n^2$, we can construct a structure that uses $O(m^{1+\varepsilon}/\beta^2)$ preprocessing time and storage such that queries take $O((n/\beta^2\sqrt{m})\log^2 n)$ time.*

If $\mathcal{P}$ is a set of (α, β)-covered polyhedra in $\mathbb{R}^3$ with total complexity n, then the bounds are the same except for the dependence on the fatness constant: in all cases, $O(1/\beta^2)$ is replaced by $O(1/(\alpha\beta)^5)$.

Remark 4.13 This result is most likely optimal, up to an $O(n^\varepsilon)$ factor. Indeed, the ray-shooting problem for fat polyhedra in 3-space is at least as hard as the ray-shooting problem for squares in the plane. For the latter problem no better bounds are known. Moreover, Hopcroft's problem—deciding whether there is an incidence between given sets of n_p points and n_ℓ lines in the plane—can be solved by performing n_ℓ ray-shooting queries in the set of points (which can be considered degenerate squares). If we set the storage parameter m of our structure to $m = n_p + n_p^{2/3} n_\ell^{2/3}$, then our algorithm solves Hopcroft's problem in $O((n_p + n_p^{2/3} n_\ell^{2/3} + n_\ell)^{1+\varepsilon})$ time. In a restricted model of computation, the lower bound for Hopcroft's problem [49] is $\Omega(n_p \log n_\ell + n_p^{2/3} n_\ell^{2/3} + n_\ell \log n_p)$. Thus it is unlikely that better results than the trade-off bounds that we obtain are possible for ray shooting in a set of points in the plane. (This is not a formal statement, because of the restricted model of computation.) Hence, such improvements are also unlikely for ray shooting in a set of fat objects in 3-space.

4.5 Simplex Range Searching

The techniques described above can be adapted to the problem of simplex range searching. The task is to preprocess $\mathcal{P}$ to facilitate queries of the form: report, given a query simplex

Δ, all objects intersecting Δ. Unlike the previous sections of this chapter, we must assume that $\mathcal{P}$ contains constant-complexity polyhedra.

Lemma 4.14 *A tower t of polyhedron $P \in \mathcal{P}$ intersects a query simplex Δ if and only if one of the following conditions holds:*

   (i) $t \subset \Delta$, or

  (ii) an edge of Δ intersects t, or

 (iii) t properly intersects a facet f of Δ, that is, t intersects only the relative interior $\mathrm{int}(f)$ of f and not its boundary.

Proof. If one of the conditions is met, then t clearly intersects Δ. Suppose neither of the first two conditions are met, but t intersects Δ. Since t intersects Δ but is not contained in it, a facet f of Δ must intersect t; $\Delta \subset t$ is ruled out by condition (ii). Since none of the edges of Δ meet t, it must be the case that $t \cap f = t \cap \mathrm{int}(f)$, so the last condition holds, as claimed. □

Let $\mathcal{T} = \mathcal{T}(\vec{d})$ be the collection of all towers for the canonical direction $\vec{d}$. For each of the three conditions we will construct a data structure that can report all towers $t \in \mathcal{T}$ satisfying that condition.

To handle condition (i), we take a vertex of each tower in $\mathcal{T}$ and preprocess the resulting set of points for simplex range searching. Thus, for any m with $n < m < n^3$, we can obtain $O((n/m^{1/3}) \log n + k)$ query time using $O(m^{1+\varepsilon})$ preprocessing time and storage, where k is the number of reported towers [2].

Condition (ii) is handled as follows. Recall that in Section 4.4 we developed a structure for segment-emptiness queries in a set of towers. Now we need to report all towers intersecting a segment, instead of only testing if there is such a tower. The structures presented in Section 4.4 can also report all intersected towers, with one exception: in the multi-level structures for conditions 2b and 2c we used the structure of Theorem 4.4 as the final level. This structure was used to check whether any line from a given set of lines was above a query line. Now we need to report all such lines, which can be done using a structure for half-space range reporting in 5-dimensional (Plücker) space [2]. Note that since the objects in $\mathcal{P}$ are constant-complexity, each object is reported at most a constant number of times. We get a structure with $O(m^{1+\varepsilon})$ preprocessing time and storage such that queries take $O((n/\sqrt{m}) \log n + k)$ time.

To handle condition (iii) we proceed as follows. For each vertex v of the cap of a tower $t \in \mathcal{T}$ we define a segment s_v, which we call a *stick*, as follows. Let ℓ_v be the line through v in direction $\vec{d}$. Then $s_v := \ell_v \cap t$. Thus the stick s_v connects v to the point on $\mathrm{base}(t)$ that is below v.

Lemma 4.15 *If a tower $t \in \mathcal{T}$ properly intersects a facet f of the query simplex Δ, then f is intersected by an edge of $\text{base}(t)$, or by the stick s_v of a vertex v of $\text{cap}(t)$.*

Proof. Suppose t properly intersects f, but f does not intersect an edge of $\text{base}(t)$. Then $\text{base}(t)$ must be completely below the plane containing f. On the other hand, at least one cap vertex, v, must be above that plane, otherwise t would not intersect f. Hence, the stick s_v must intersect that plane. Since t properly intersects f, this implies that s_v must intersect f. $\qquad\qquad\square$

This lemma gives us an easy way to handle condition (iii): we need a structure so that we can find all sticks whose projection in direction $\vec{d}$ (note that this is a point) is contained in the projection of a facet f of the query simplex and whose endpoints are on opposite sides of the plane through f. This can again be done with a multi-level partition tree that uses $O(m^{1+\varepsilon}/\beta^2)$ preprocessing time and storage, and for which queries take $O((n/\beta^2\sqrt{m})\log n + k)$ time. A similar structure can be used to find the base edges intersecting f, since the base edges of the towers in $\mathcal{T}$ have only two distinct directions.

Since we have $O(1/\beta^2)$ different canonical directions in the case of convex polyhedra and $O(1/(\alpha\beta)^5)$ towers in the case of (α, β)-covered polyhedra, we get the following.

Theorem 4.16 *Let $\mathcal{P}$ be a set of n β-fat constant-complexity convex polyhedra in $\mathbb{R}^3$. For any m with $n < m < n^3$, we can preprocess $\mathcal{P}$ using $O(m^{1+\varepsilon}/\beta^2)$ time and storage, such that simplex range-searching queries can be answered in $O((n/\beta^2 m^{1/3})\log n + k/\beta^2)$ time, where k is the number of polyhedra reported.*

If $\mathcal{P}$ is a set of n constant-complexity (α, β)-covered polyhedra in $\mathbb{R}^3$, then the bounds are the same except for the dependence on the fatness constant: in all cases, $O(1/\beta^2)$ is replaced by $O(1/(\alpha\beta)^5)$.

4.6 Conclusion

In this chapter, we looked at two related problems: ray shooting and range searching. We studied ray shooting both for rays whose direction is fixed and for rays that can have an arbitrary direction. In the first case, we gave improved results for objects that are fat and convex, whereas in the second case we gave improved results for polyhedra that are (α, β)-covered. These results extended fairly directly into simplex range searching as well. The results on ray shooting in arbitrary directions and simplex range searching polyhedra show the utility of the decomposition into towers that we introduced in Chapter 3.

Appendix

This appendix contains a proof that was omitted.

Lemma 4.2. Let P be a β-fat convex polygon with n vertices. There is a set T of α-fat triangles that cover P where $|T| = O(n)$ and $\alpha \geq \beta/128$.

Proof. Recall that for triangles, we use the definition that the fatness is given by the smallest angle in the triangle.

Let S be the largest possible square contained in P. Any convex subset of P that contains all of S is at least β'-fat where $\beta' = \Theta(\beta)$ by Lemma 4.18 below.

We extend the edges of S until they intersect P and add vertices to P at the intersection points. We let P_a denote the part of P above the (extended) top edge of S, let P_b denote the part below the bottom edge of S, let P_c denote the part to the right of the right edge of S, and let P_d denote the part to the left of the left edge of S. We will show how to cover P_a. The three other parts of P are covered similarly, and S is covered with two triangles that each have a fatness of $45°$.
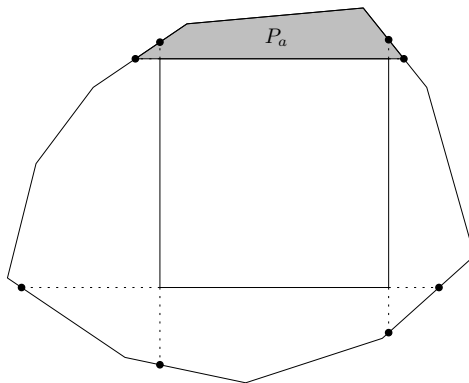
Figure 4.2 One of the subpolygons of P induced by S.

An ear of a polygon P consists of two consecutive edges of P that have the property that a straight edge connecting the first and last vertex of the edges stays completely inside the polygon. In a convex polygon, any two consecutive edges are ears.

We cover P_a by choosing an arbitrary ear from it (except any ear that also contains the top edge of S), covering it using Lemma 4.17 below, and then replacing P by P with that ear removed. Since no part of S is ever removed, P remains fat. Thus we keep removing ears from P_a until it exactly coincides with the extended edge of S.

Since we cover the ears that we remove using the procedure from Lemma 4.17, we add a constant number of triangles to T per vertex, implying that $|T| = O(n)$. The exact bound

on α is given by combining Lemmas 4.17 and 4.18. □

Lemma 4.17 *An ear of a β-fat polygon P can be covered with at most four α-fat triangles that all stay inside P where $\alpha := (\beta\pi)/16$.*

Proof. In a convex polygon, an ear is a triangle formed by three consecutive vertices. Consider the ear defined by vertices v_{i-1}, v_i, and v_{i+1}. Let ϕ_{i-1}, ϕ_i, and ϕ_{i+1} be the angles at the respective vertices—see Figure 4.3 (a). Because P is β-fat, we know that the angle between any two adjacent edges of P, and in particular the angle ϕ_i, is at least $\beta/(2\pi)$. There are three possibilities for the other two angles, ϕ_{i-1} and ϕ_{i+1}: either they are both at least α, they are both less than 2α, or one is larger than 2α and one is smaller than α. Note that these cases overlap.

Case (i): $\phi_{i-1} \geq \alpha$ and $\phi_{i+1} \geq \alpha$. In this case, the ear is trivial to cover: it is already an α-fat triangle that can be covered by a copy of itself.

Case (ii): $\phi_{i-1} < 2\alpha$ and $\phi_{i+1} < 2\alpha$. First, we add triangles to the edges $v_{i-1}v_i$ and v_iv_{i+1} where the angles of the edges of the triangles with respect to the boundary edges are at least 2α—these are the triangles with dotted edges in Figure 4.3 (a). These triangles must stay inside P as long as $\alpha \leq (\beta\pi)/16$ by Lemma 5.6, proved in the next chapter. However, it is clear that the non-boundary vertex of these triangles must be outside the ear that we are covering. Therefore, we can place a triangle at the middle vertex of the ear with two sides that correspond to the sides of the two triangles that we just added and whose third side is the edge of the ear that goes between these two edges. This triangle completes the covering of the ear.

(a)

(b)
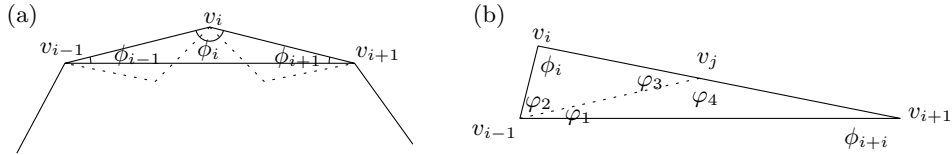
Figure 4.3 (a) Case (ii). (b) Case (iii).

Case (iii): $\phi_{i-1} > 2\alpha$ and $\phi_{i+1} < \alpha$ (or the symmetric case). See Figure 4.3 (b). In this case, we add an edge between the vertex that is at the large angle (v_{i-1}, without loss of generality) and the edge across from it, making vertex v_j. This splits ϕ_i into two angles φ_1 and φ_2. We place v_j such that φ_1 is exactly α. Thus, $\varphi_3 = \alpha + \phi_{i+1} > \alpha$. By assumption, $\varphi_2 > \alpha$. Thus, we can cover the triangle $v_{i-1}v_iv_j$ with a copy of itself. Triangle $v_{i-1}v_jv_{i+1}$ can be covered according to the procedure outlined for case (ii) above.

Note that in all cases, we have covered the ear with at most four α-fat triangles. □

Lemma 4.18 *Let P be a convex β-fat polygon in $\mathbb{R}^2$ and S be the largest square contained in P. Then any convex subset P' such that $S \subseteq P' \subseteq P$ is β'-fat where $\beta' \geq \beta/(8\pi)$.*

Proof. By the results of Section 3.2.1 of Van der Stappen's thesis [96], the side length of S is at least $\beta\rho/(2\sqrt{2})$, where ρ is the diameter of P.

Let $d = \overline{p_1 p_2}$ be the diameter of P'. Let $S' \subseteq S$ be the largest square contained in S that has an edge parallel to d. The side length of S' is at least $\sqrt{2}/2$ times the side length of S. Let p_3 and p_4 denote the midpoints of the sides of S' parallel to d—see Figure 4.4.
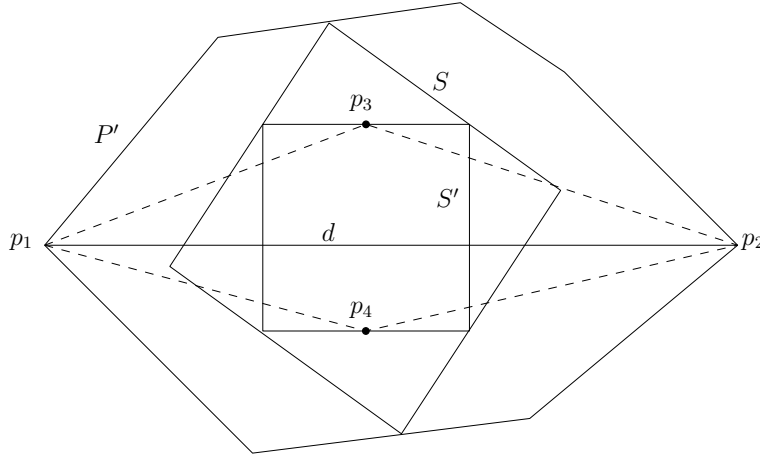
Figure 4.4 P' must be fat.

We will make two triangles: $p_1 p_3 p_4$ and $p_2 p_3 p_4$. By convexity, both of these triangles must be completely inside P'. The sum of the area of these triangles is not dependent on the placement of S'—it is always $d \cdot s/2$, where s is the side length of S'.

Since P' is convex, the fatness of P' is determined by a circle placed at p_1 with radius d [96]. The area of that circle is πd^2. Thus the fatness of P' is at least

$$\beta' = \frac{\frac{d \cdot s}{2}}{\pi d^2} = \frac{s}{2d\pi} \geq \frac{\beta\rho}{8d\pi} \geq \frac{\beta}{8\pi}$$

since $d \leq \rho$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

CHAPTER 5

Depth orders

5.1 Introduction

In this chapter we study another problem arising in computer graphics in the context of realistic input models; namely the depth-order problem for fat polytopes in $\mathbb{R}^3$.

Problem statement and previous results. Let $\mathcal{P}$ be a set of n disjoint objects in $\mathbb{R}^3$. The problem we study is the *depth-order problem*: compute a depth order for the set $\mathcal{P}$, that is, an ordering $P_1, \ldots, P_n$ of the objects in $\mathcal{P}$ such that if P_i is below P_j then $i < j$. Here we say that P_i is below P_j, denoted by $P_i \prec P_j$, if there are points $(x, y, z_i) \in P_i$ and $(x, y, z_j) \in P_j$ with $z_i < z_j$. In other words, a depth order is a linear extension of the $\prec$-relation. Since there can be cycles in the $\prec$-relation—we then say there is *cyclic overlap* among the objects—a depth order does not always exist. In that case the algorithm should report that there is cyclic overlap. Depth orders are useful in several applications. For example, they can be used to render scenes with the Painter's Algorithm [52] or to do hidden-surface removal with the algorithm of Katz *et al.* [60]. Depth orders also play a role in assembly planning [3].

The depth-order problem for arbitrary sets of triangles in 3-space does not seem to admit a near-linear solution; the best known algorithm runs in $O(n^{4/3+\varepsilon})$ time [39]. This has led researchers to also study this problem for fat objects. Agarwal *et al.* [5] gave an algorithm for computing the depth order of a set of triangles whose projections onto the

xy-plane are fat; their algorithm runs in $O(n \log^5 n)$ time. However, their algorithm cannot detect cycles—when there are cycles it reports an incorrect order. A subsequent result by Katz [58] produced an algorithm that runs in $O(n \log^5 n)$ time and that can detect cycles. In this case though, the constant of proportionality depends on the minimum overlap of the projections of the objects that do overlap. If there is a pair of objects whose projections barely overlap, then the running time of the algorithm increases greatly. One advantage that this algorithm has is that it can deal with convex curved objects.

Our results. We present an algorithm for computing a depth order on a collection of n convex constant-complexity fat polyhedra in $\mathbb{R}^3$. Our algorithm runs in $O((1/\beta^3)n \log^3 n)$ time, improving the result of Agarwal *et al.* [5] by two logarithmic factors. Like the algorithm of Agarwal *et al.*, our algorithm unfortunately does not detect cyclic overlap. Hence, we also study the problem of verifying a given depth order. We give an algorithm that checks in $O((1/\beta^2)n \log^3 n)$ time[1] whether a given ordering for a set of fat convex polyhedra is a valid depth order. This is the first result for this problem. Until now, the only algorithm for verifying a given depth order was an algorithm for arbitrary triangles [39], which runs in $O(n^{4/3+\varepsilon})$ time.

5.2 Preliminaries

In this section we introduce some basic definitions and terminology.

Define the *size* of an object o, denoted by $\text{size}(o)$ to be the radius of its smallest enclosing ball. Note that the size of a ball is simply its radius.

It is not hard to show that the projection of a fat object is also fat, as proved by De Berg [31] and made precise in the following lemma.

Lemma 5.1 (De Berg [31]) *If an object P is a β-fat object in three dimensions, then* $\text{proj}(P)$ *has fatness $\Omega(\beta)$ in two dimensions.*

We will also need the following lemma.

Lemma 5.2 *Let P_1 and P_2 be simple polygons. Let e_1 be an edge of P_1 and e_2 be an edge of P_2. If P_1 intersects P_2 so that there is no vertex of P_1 inside P_2 and no vertex of P_2 inside P_1, then there is an intersection of edges e_3 of P_1 and e_4 of P_2 such that $e_3 \neq e_1$ and $e_4 \neq e_2$.*

Proof. Let e of P_1 and e' of P_2 be edges that intersect. If $e \neq e_1$ and $e' \neq e_2$, then we are done. If $e \neq e_1$ and $e' = e_2$, then there must be an intersection between e and a different

---

[1]This is an improvement over the $O(n \log^4 n)$ bound that we had in the preliminary version of the paper [34], which was published in SODA 2006.

edge of P_2 (since there are no vertices of P_1 inside P_2) meaning that we have found an intersection between e and some edge $e'' \neq e_2$, and we are done. Similarly, we are done if $e = e_1$ and $e' \neq e_2$. Finally, suppose $e = e_1$ and $e' = e_2$. Since e_1 enters P_2, it must exit it, and that implies that there must be an intersection between e_1 and some edge $e'' \neq e_2$. This puts us in the previous case, so we are done. □

5.3 The size of the transitive reduction of depth-order graphs

Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set of disjoint objects in $\mathbb{R}^3$. Recall that we say that P_i is *below* P_j, denoted by $P_i \prec P_j$, if there are points $(x, y, z_i) \in P_i$ and $(x, y, z_j) \in P_j$ with $z_i < z_j$. We define the *depth-order graph of* $\mathcal{P}$ to be the graph $\mathcal{G}(\mathcal{P}) = (\mathcal{P}, E)$ where $(P_i, P_j) \in E$ iff $P_i \prec P_j$. Hence, a depth order for $\mathcal{P}$ corresponds to a topological order on $\mathcal{G}(\mathcal{P})$.

In general it is too costly to compute $\mathcal{G}(\mathcal{P})$ explicitly, since it can have $\Omega(n^2)$ arcs. When computing depth orders for segments in the plane, this can be circumvented by only looking at pairs of segments that "see" each other, that is, that can be connected vertically without crossing another segment. For objects in 3-space, however, the number of pairs that see each other can be quadratic, even when the objects are fat. In this section we therefore study the size of the transitive reduction of depth-order graphs, since the transitive reduction is the smallest subgraph that is sufficient to topologically sort a graph. The main result is that the number of arcs in the transitive reduction of the depth-order graph of a set of fat objects is linear. Then in the next section we will compute a superset of the arcs in the transitive reduction.

We define the *separation* of two nodes in the depth-order graph, denoted $sep(P_i, P_j)$ to be the length of the longest path from P_i to P_j. Notice that if the graph contains cycles, $sep(P_i, P_j)$ can be infinite. We define $\mathcal{G}^{(1)}(\mathcal{P}) = (\mathcal{P}, E^{(1)})$ to be the subgraph of the depth-order graph $\mathcal{G}(\mathcal{P})$ where $(P_i, P_j) \in E^{(1)}$ if and only if $sep(P_i, P_j) = 1$ in $\mathcal{G}(\mathcal{P})$.

Lemma 5.3 *If $\mathcal{G}(\mathcal{P})$ is acyclic, the transitive closure of $\mathcal{G}^{(1)}(\mathcal{P})$ is the transitive closure of $\mathcal{G}(\mathcal{P})$.*

Proof. We have to prove that there is a path $P_i \rightsquigarrow P_j$ in $\mathcal{G}(\mathcal{P})$ if and only if there is a path $P_i \rightsquigarrow P_j$ in $\mathcal{G}^{(1)}(\mathcal{P})$. The "if" part is obvious since $\mathcal{G}^{(1)}(\mathcal{P})$ is a subgraph of $\mathcal{G}(\mathcal{P})$. We prove the "only if" part by induction on $sep(P_i, P_j)$.

If $sep(P_i, P_j) = 1$, the arc (P_i, P_j) exists in $\mathcal{G}^{(1)}(\mathcal{P})$ by construction. Now assume there is a path in $\mathcal{G}^{(1)}(\mathcal{P})$ between all nodes with separation m. Take P_i and P_j in $\mathcal{G}(\mathcal{P})$ which have separation $m + 1$. Then there is a node x such that $sep(P_i, x) = 1$ and

71

$sep(x, P_j) = m$. By the induction hypothesis, we then have a path $P_i \rightarrow x \rightsquigarrow P_j$ in $\mathcal{G}^{(1)}(\mathcal{P})$. $\qquad\qquad\qquad\square$

For arbitrary triangles in 3-space, the number of arcs in $\mathcal{G}^{(1)}(\mathcal{P})$ can still be $\Theta(n^2)$. For some special classes of objects, however, the number of arcs is linear. For example, one can show that this number is linear for a set of disjoint polyhedra whose projections form a set of polygonal pseudodisks [42]. Here we concentrate on the case where the objects in the given set $\mathcal{P}$ project onto fat convex objects. We show that in this case the number of arcs is also linear. Since fat convex objects project to fat objects, showing this also shows that the number of arcs in $\mathcal{G}^{(1)}(\mathcal{P})$ is small if the input is a set of fat objects. We start with an auxiliary lemma

Lemma 5.4 *Let $P_i \in \mathcal{P}$ be an object and let $\mathcal{P}^+(i)$ be the subset of objects $P_j \in \mathcal{P}$ that are above P_i and where $sep(P_i, P_j) = 1$. Then the projections $\mathrm{proj}(P_j)$ of the objects $P_j \in \mathcal{P}^+(i)$ are pairwise disjoint.*

Proof. Suppose not. Then there are objects $P_j, P_k \in \mathcal{P}^+(i)$ such that $\mathrm{proj}(P_j) \cap \mathrm{proj}(P_k) \neq \emptyset$ and $sep(P_i, P_j) = 1$ and $sep(P_i, P_k) = 1$. Since $\mathrm{proj}(P_j)$ and $\mathrm{proj}(P_k)$ intersect, they must share at least one point, so there must be an arc between P_j and P_k in $\mathcal{G}(\mathcal{P})$. Therefore, either $sep(P_i, P_j) > 1$ or $sep(P_i, P_k) > 1$, either case being a contradiction. $\qquad\square$

Theorem 5.5 *Let $\mathcal{P}$ be a collection of n disjoint objects in $\mathbb{R}^3$ that project to convex β-fat objects. Then the number of edges in $\mathcal{G}^{(1)}(\mathcal{P})$ is $O(n/\beta)$.*

Proof. We will charge each arc in $\mathcal{G}^{(1)}(\mathcal{P})$ to an object, and then use a packing argument to show that the number of arcs in $\mathcal{G}^{(1)}(\mathcal{P})$ charged to each object is $O(1/\beta)$.

We project all objects onto the xy-plane, making them convex fat objects. In this setting, we say that one object is above another if the original objects satisfy this relationship.

Recall that for a planar object o, its size is defined as the radius of its smallest enclosing disk. Consider an arc (P_i, P_j) in $\mathcal{G}^{(1)}(\mathcal{P})$. We charge the arc to the smaller of the two objects. That is, we charge the arc to P_i if $\mathrm{size}(\mathrm{proj}(P_i)) < \mathrm{size}(\mathrm{proj}(P_j))$ and to P_j otherwise, breaking ties arbitrarily. We claim that any object is charged $O(1/\beta)$ arcs. To prove this, take an arbitrary object P_j such that (P_i, P_j) is charged to P_i. Let $\rho = \mathrm{size}(\mathrm{proj}(P_i))$. If there is an arc in $\mathcal{G}^{(1)}(\mathcal{P})$ between P_i and P_j, then $\mathrm{proj}(P_j)$ intersects $\mathrm{proj}(P_i)$. Let p be a point in this intersection. Then a circle centered at p with radius ρ is centered in $\mathrm{proj}(P_j)$ and does not fully enclose $\mathrm{proj}(P_j)$, or else $\mathrm{proj}(P_j)$ would have a smallest enclosing circle that is smaller or equal to that of $\mathrm{proj}(P_i)$. Thus, this circle contains at least $\beta\pi\rho^2$ units of area of $\mathrm{proj}(P_j)$ by the definition of fatness. Also, this circle is completely enclosed in a circle of radius 2ρ centered at the center of the smallest enclosing disk of $\mathrm{proj}(P_i)$. This is illustrated in Figure 5.1.

Since all objects $\mathrm{proj}(P_j)$ where P_j is above P_i and $sep(P_i, P_j) = 1$ must be disjoint by Lemma 5.4, and because each must have at least $\beta\pi\rho^2$ units of area inside a disk that
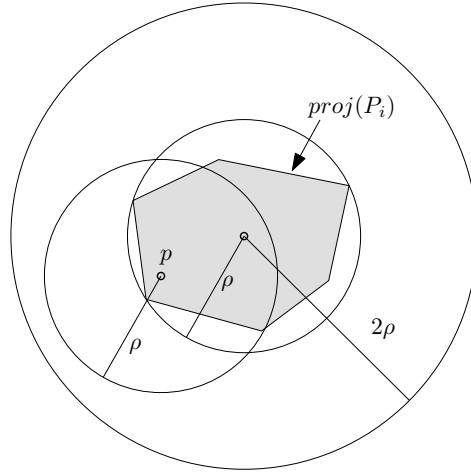
Figure 5.1 Illustration of the packing argument.

has $4\pi\rho^2$ units of area, there can only be $4/\beta$ edges of $\mathcal{G}^{(1)}(\mathcal{P})$ charged to P_i. We must double this number to account for objects P_j below P_i such that (P_j, P_i) is charged to P_i. Therefore, we get an upper bound on the number of arcs charged to P_i of $8/\beta$. Finally, since there are n objects, $\mathcal{G}^{(1)}(\mathcal{P})$ can have at most $8n/\beta$ edges, which is $O(n/\beta)$. ☐

5.4 Computing depth orders

We now present the algorithm for finding the depth order of a set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of n disjoint β-fat convex polyhedra. In contrast to Theorem 5.5, we require the complexity of the projection of each object to be constant.

Witness edges. One of the basic steps that we need to perform repeatedly in our algorithm will be to find polyhedra that are above a query polyhedron. To facilitate this, we will add so-called *witness edges* inside the projection of each P_i. They are defined as follows.

Let β' be defined so that each member of $\{\text{proj}(P_i) \mid P_i \in \mathcal{P}\}$ is β'-fat. By Lemma 5.1 we know that $\beta' = \Omega(\beta)$. Also let $\mathcal{C} = \{0, \alpha, 2\alpha, \ldots, c\alpha\}$ where $\alpha = (\beta'\pi)/8$ and $c = \lfloor 2\pi/\alpha \rfloor$. We call the directions in $\mathcal{C}$ *canonical directions* as in Chapter 1. We require the witness edges to have the following properties. Let W_i and W_j be the sets of witness edges constructed for P_i and P_j respectively.

(i) Each witness edge has one of the canonical directions.

(ii) For any pair of polyhedra P_i and P_j, we have that $\mathrm{proj}(P_i)$ intersects $\mathrm{proj}(P_j)$ if and only if at least one of the following is true:

- A vertex of $\mathrm{proj}(P_i)$ is inside $\mathrm{proj}(P_j)$, or a vertex of $\mathrm{proj}(P_j)$ is inside $\mathrm{proj}(P_i)$.
- A witness edge in W_i crosses a witness edge in W_j.

The construction of the set W_i of witness edges for P_i is done as follows. For each edge $e = vw$ of $\mathrm{proj}(P_i)$ we add to W_i two witness edges e' and e'' that are incident to v and w, respectively, extend into the interior of P_i, and form a triangle with e. The directions of the witnesses are chosen from the canonical directions, such that the interior angles that e' and e'' make with e are minimal—see Figure 5.2. We claim that if we add
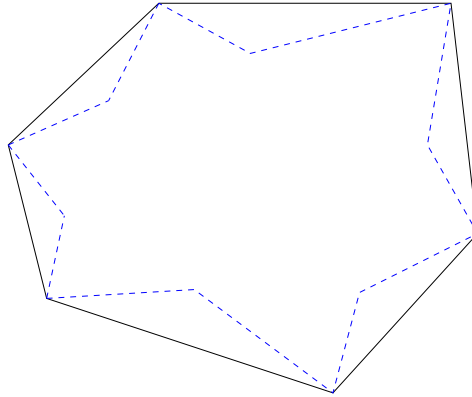

Figure 5.2 A projection of a polyhedron with witness edges added

the witness edges in this manner, they have the required properties. The first property holds by construction, so it remains to prove the second property. We first argue that the witness edges lie completely inside $\mathrm{proj}(P_i)$, which implies that the "if"-part of the second property holds.

Lemma 5.6 *The witness edges in W_i lie completely inside $\mathrm{proj}(P_i)$.*

Proof. Let e be the edge for which we are adding witness edges. Let p be the midpoint of e and consider the circle C with center p and diameter equal to the length of e. Suppose an edge of $\mathrm{proj}(P_i)$ intersects the triangle formed by e, e', and e''. Note that this region must be inside the isosceles triangle with angles α and base e—the lighter region in Figure 5.3 by the minimal-angle condition which implies that the angles that e makes with e' and e'' are at most α. Then, by convexity of $\mathrm{proj}(P_i)$, we know that $\mathrm{proj}(P_i) \cap C$ must be completely inside the union of the triangular wedges in Figure 5.3. These wedges have area at most $\beta'\pi|e|^2/8$ inside C. Hence, $area(\mathrm{proj}(P_i) \cap C) < \beta'\pi|e|^2/4$, contradicting
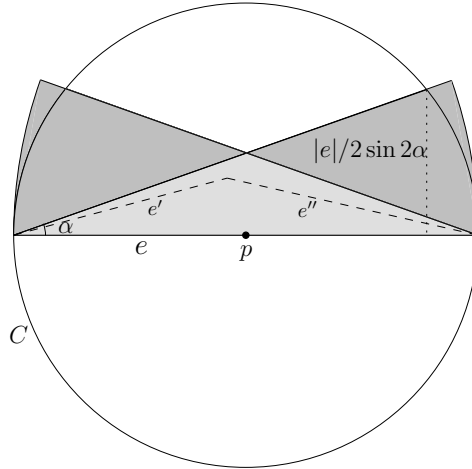
Figure 5.3 No edge of the polygon may enter the light gray region.

our assumption that $\mathrm{proj}(P_i)$ is β'-fat. □

The following lemma, which follows directly from Lemma 5.2, finishes the proof that the witness edges have the required properties.

Lemma 5.7 *If* $\mathrm{proj}(P_i)$ *intersects* $\mathrm{proj}(P_j)$ *and* $\mathrm{proj}(P_i)$ *does not contain a vertex of* $\mathrm{proj}(P_j)$ *or vice versa, then a witness edge from* P_i *intersects a witness edge from* P_j.

The algorithm. The general idea of our algorithm is as follows. By Lemma 5.3 it is sufficient to find all pairs of objects P_i, P_j of separation 1 in the depth-order graph. Such a pair of objects must, of course, intersect in the projection. Thus ideally we would like to find among all pairs P_i, P_j whose projections intersect the ones of separation 1. Our algorithm does not quite achieve this—it will find more pairs—but the number of extra pairs we find will be small. Lemma 5.7 suggests that the task of finding the intersecting pairs of projections can be broken into two parts: finding pairs for which there is a vertex of the projection of one polyhedron inside the projection of another, and finding crossing pairs of witness edges.

Below we give a more detailed description of the algorithm. The algorithm will find a set A of arcs—a superset of the arcs (P_i, P_j) for objects of separation 1—and then topologically sort the graph $\mathcal{G}^* = (\mathcal{P}, A)$. Initially A is empty.

1. For every vertex v of each object $P_i \in \mathcal{P}$, find the objects $P^b(v)$ and $P^a(v)$ that are directly below and above v, respectively, and add the arcs $(P^b(v), P_i)$ and

$(P_i, P^a(v))$ to A.

2. Sort the objects by decreasing size so that $\text{size}(\text{proj}(P_1)) \geq \cdots \geq \text{size}(\text{proj}(P_n))$, and define $\mathcal{S}_i = \{P_1, \ldots, P_i\}$.

3. For every witness edge e associated with each P_i, find a set $\mathcal{P}(e)$ consisting of objects $P_j \in \mathcal{S}_{i-1}$ with the following properties:

   (P1) Each $P_j \in \mathcal{P}(e)$ has a witness edge that intersects e.

   (P2) Each $P_j \in \mathcal{P}(e)$ is above P_i.

   (P3) Each $P_j \in \mathcal{S}_{i-1}$ with $sep(P_i, P_j) = 1$ that satisfies (P1) and (P2) is a member of $\mathcal{P}(e)$.

   For each P_i, add the set of arcs $\{(P_i, P_j) : P_j \in \mathcal{P}(e) \text{ and } e \text{ is a witness edge of } P_i\}$ to A.

4. Repeat step 3 with "below" substituted for "above" and the directions of the arcs added reversed.

5. Topologically sort the graph $\mathcal{G}^* = (\mathcal{P}, A)$ and report the order.

Lemma 5.8 *The order reported by the algorithm is a valid depth order for $\mathcal{P}$, if a depth order exists.*

Proof. Assume a depth order exists for $\mathcal{P}$. It follows directly from the construction that every arc added to the set A is also an arc in the depth-order graph $\mathcal{G}(\mathcal{P})$. It remains to argue that A is a superset of the set of arcs in the graph $\mathcal{G}^{(1)}(\mathcal{P})$.

Consider an arc (P_i, P_j) in $\mathcal{G}^{(1)}(\mathcal{P})$. If there is a vertex of $\text{proj}(P_i)$ in $\text{proj}(P_j)$ (or vice versa) then, because $sep(P_i, P_j) = 1$, that vertex is directly below P_j (resp. above P_i). Hence, the arc is found in Step 1. By Lemma 5.7, the remaining case is that a witness edge of $\text{proj}(P_i)$ intersects a witness edge from $\text{proj}(P_j)$. Without loss of generality, assume P_i is smaller than P_j. Hence, $P_j \in \mathcal{S}_{i-1}$. Since (P_i, P_j) is an arc in $\mathcal{G}^{(1)}(\mathcal{P})$, $sep(P_i, P_j) = 1$. By condition (P3), the arc will be found in Step 3 or 4, depending on whether P_j is above or below P_i. $\square$

Step 1 can be carried out efficiently using the vertical ray-shooting data structure presented in Chapter 4. Hence, it remains to describe Step 3 in more detail. This step will be performed as follows. We will treat each $P_2, \ldots, P_n$ in order. When we have to handle P_i, we will make sure we have a data structure available that we can query with each witness edge e of P_i and that will then report the set $\mathcal{P}(e)$. After having queried with all witness edges of P_i, we insert P_i into the data structure and proceed with P_{i+1}. Next we describe this data structure.

The witness-edge-intersection data structure. Consider the set of all witness edges of the objects in $\mathcal{P}_{i-1}$. These witness edges have canonical directions, so we can partition them into $|\mathcal{C}|$ subsets depending on their directions. The query segment e has one of the canonical directions as well. Hence, we construct for each subset $|\mathcal{C}|$ different data structures, one for each query direction. We now describe the structure for one such subset, let's call it W, and a fixed query direction.

Assume without loss of generality that the witness edges in W are all horizontal, and that the query edge e is vertical. The structure is a multi-level data structure defined as follows.

- The top level of the data structure is a segment tree $\mathcal{T}$ on the projections of the edges in W onto the x-axis. Note that each node ν in $\mathcal{T}$ corresponds to a vertical slab in the plane.

- Let $W(\nu)$ denote the edges in W whose projection is in the canonical subset of ν. Such an edge crosses the slab of ν but not the slab of the parent of ν. We store the edges in $W(\nu)$ in a balanced binary tree $\mathcal{T}(\nu)$, ordered according to their y-coordinates. We call this the "slab tree". So far our structure is just a standard two-level tree to perform intersection queries with vertical segments in a set of horizontal segment in the plane [42].

- Let μ be a node in $\mathcal{T}(\nu)$. Let $\mathcal{P}(\mu)$ denote the subset of objects that have a witness edge in the subtree rooted at μ. The node μ represents a rectangular[2] region $R(\mu)$ that is bounded by two slab boundaries and the topmost and bottommost edge stored in the subtree rooted at μ. We associate with μ a reduced subset $\overline{\mathcal{P}(\mu)} \subset \mathcal{P}(\mu)$ of the objects, in the following way: $P_j \in \overline{\mathcal{P}(\mu)}$ iff $P_j \in \mathcal{P}(\mu)$ and $\text{size}(\text{proj}(P_j)) \geq \text{size}(R(\mu))/2\sqrt{2}$.

  By Lemma 4.1 we can find a set $Q(\mu)$ consisting of $O(1/\beta^2)$ points such the projection of any object $P_j \in \overline{\mathcal{P}(\mu)}$ is stabbed. We arbitrarily assign each $P_j \in \overline{\mathcal{P}(\mu)}$ to one of the points q it contains, and we associate a balanced binary search tree $\mathcal{T}(q)$ with each point q on the associated objects, where the sorting order is defined by the height of the objects along the vertical line through q.

This finishes the description of the data structure. Next we describe the algorithms to query the structure and to insert an object.

Lemma 5.9 *With the structure described above, we can find the set $\mathcal{P}(e)$ referred to in Step 3 of the depth-order algorithm in $O((1/\beta^3)\log^3 n)$ time. Furthermore, the set $\mathcal{P}(e)$ contains $O((1/\beta^3)\log^2 n)$ objects.*

Proof. Recall that we actually have to query $|\mathcal{C}| = O(1/\beta)$ different versions of the structure. We focus on the time spent in one of these structures.

---

[2]This is only true because we assumed the edges in W are horizontal and the query edge is vertical. In general, μ will represent a parallelogram, but this does not influence the arguments.

To perform a query with a witness edge e belonging to an object P_i, we search with e in the first two levels of the tree in the standard way. This gives us $O(\log^2 n)$ nodes μ whose subtrees contain exactly those edges that intersect e. At each node μ, we use the trees $\mathcal{T}(q)$ for $q \in Q(\mu)$ to find the lowest object that is above P_i. We can search in $\mathcal{T}(q)$ since P_i is known to intersect all objects in $\mathcal{P}(q)$ in the projection. Hence, at μ, we find $|Q(\mu)|$ objects in $O(|Q(\mu)|\log n)$ time in total. The query time and the bound on the size of $\mathcal{P}(e)$ follow.

It remains to argue that the reported set has the required properties. Properties (P1) and (P2) follow immediately from the definition of the data structure and query algorithm. Furthermore, when we query a tree $\mathcal{T}(q)$ we can indeed restrict our attention to the lowest object that is above P_i, because the other objects P_j will either be below P_i or have $sep(P_i, P_j) > 1$. Hence, to prove (P3) it is sufficient to argue that any P_j satisfying (P1) and (P2) and with $sep(P_i, P_j) = 1$ will be a member of one of the sets $\overline{\mathcal{P}(\mu)}$. We know that the object will be a member of $\mathcal{P}(\mu)$ for a visited node μ.

Suppose for a contradiction that $P_j \notin \overline{\mathcal{P}(\mu)}$. This means we must have $\mathrm{size}(\mathrm{proj}(P_j)) < \mathrm{size}(R(\mu))/2\sqrt{2}$. This can only happen when $\mathrm{size}(\mathrm{proj}(P_j))$ is less than $d/2$, where d is the distance between the top and bottom edge of $R(\mu)$, because P_j crosses the slab of which $R(\mu)$ is a part. On the other hand, when we reach a node μ in the slab tree by querying with a witness edge e of P_i, we have $\mathrm{size}(\mathrm{proj}(P_i)) \geq \mathrm{length}(e)/2 \geq d/2$. This contradicts that when we query with a witness edge e of P_i, all objects P_j in the data structure have $\mathrm{size}(\mathrm{proj}(P_j)) \geq \mathrm{size}(\mathrm{proj}(P_i))$. $\qquad\square$

Lemma 5.10 *An object P_i can be inserted into the structure in $O((1/\beta)\log^2 n(\log n + 1/\beta^2))$ time.*

Proof. Each of the $O(1)$ witness edges of P_i has to be inserted into $|\mathcal{C}| = O(1/\beta)$ structures. To insert a witness edge, we first find each node μ in a slab tree whose canonical subset contains the witness edge. We test if $\mathrm{size}(P_j) \geq \mathrm{size}(R(\mu))/2$ and, if so, find a point $q \in Q(\mu)$ that is contained in $\mathrm{proj}(P_i)$ and insert P_i into the tree $\mathcal{T}(q)$. This takes $O(\log^2 n(\log n + 1/\beta^2))$ time per structure, so $O((1/\beta)\log^2 n(\log n + 1/\beta^2))$ time in total. $\qquad\square$

From the two lemmas above, we see that Steps 3 and 4 of the depth-order algorithm can be performed in $O((1/\beta^3)n\log^3 n)$ time in total. We get the following theorem.

Theorem 5.11 *Let $\mathcal{P}$ be a collection of n disjoint constant-complexity β-fat convex polyhedra in $\mathbb{R}^3$. Then we can compute a depth order for $\mathcal{P}$ in time $O((1/\beta^3)n\log^3 n)$, if it exists.*

5.5   Verifying depth orders

In order for our algorithm to be complete, it should output the correct depth order if one exists, but it should also not output an incorrect depth order if no depth order exists. Unfortunately the algorithm of the previous section does not necessarily detect cycles in the $\prec$-relation. Hence, we present an algorithm for checking whether a given order is correct.

We use the general approach by De Berg *et al.* [39] for verifying depth orders. Let $L = P_1, \ldots, P_n$ be the given order. We define $L_1 = \{P_1, \ldots, P_{\lfloor n/2 \rfloor}\}$ and $L_2 = \{P_{\lfloor n/2 \rfloor + 1}, \ldots, P_n\}$. We first check if any object in L_2 is above any object in L_1. Clearly, if the answer is "yes" then the given ordering is not valid. Otherwise, we verify the lists L_1 and L_2 recursively. If $T(\beta, n)$ is the amount of time to check the objects in L_1 against those in L_2, then the overall algorithm takes $O(T(\beta, n) \log n)$ time. We shall see that $T(\beta, n) = O((1/\beta^2)n \log^2 n)$, so the algorithm for verifying the depth order takes $O((1/\beta^2)n \log^3 n)$ time. Next we describe how to check the objects in L_1 against those in L_2.

First we introduce a new type of witness edge. The difference with the witness edges in Section 5.4 is that the new witness edges will have canonical directions in 3D, rather than in the projection. To achieve this we again use towers. Recall the following lemma from Chapter 3.

Lemma 3.11 *Let $\sigma := \lceil 54\sqrt{3}/\beta \rceil$. For any convex β-fat object o in $\mathbb{R}^3$, there exist concentric axis-aligned cubes $C^-(o)$ and $C^+(o)$ with $C^-(o) \subseteq o \subseteq C^+(o)$ such that*

$$\frac{\text{size}(C^+(o))}{\text{size}(C^-(o))} = \sigma \,.$$

Assume we are given $C^-(o)$ and $C^+(o)$ for object o. We partition each face of $C^+(o)$ into squares with side length equal to the side length of $C^-(o)$. For each facet f of $C^-(o)$ and each square on the corresponding facet of $C^+(o)$, we sweep f so that it coincides with the square—see Figure 5.4(a). The sweeping directions form the set of canonical directions. There are at most σ^2 different directions that a facet of $C^-(o)$ can be swept in, so we have $O(1/\beta^2)$ canonical directions. We denote an arbitrary member of this set of directions by $\vec{d}$. Note that the set of canonical directions thus obtained does not depend on o, only on the value σ, which is specified by the fatness factor β. This is precisely the construction of towers for convex objects we gave in Chapter 3.

For each P_i we construct a set W_i of witness edges, as follows. First, we add the edges of $C^-(P_i)$ to W_i. Second, for each silhouette vertex v of P_i—a silhouette vertex is a vertex whose projection is a boundary vertex of the projection of P_i—we add an edge e_v that connects v to one of the facets of $C^-(P_i)$. This edge is allowed to be in any of the canonical directions as long as it reaches a facet of $C^-(P_i)$. We can be certain that at least one direction works for v since there must be at least one pair consisting of a facet f
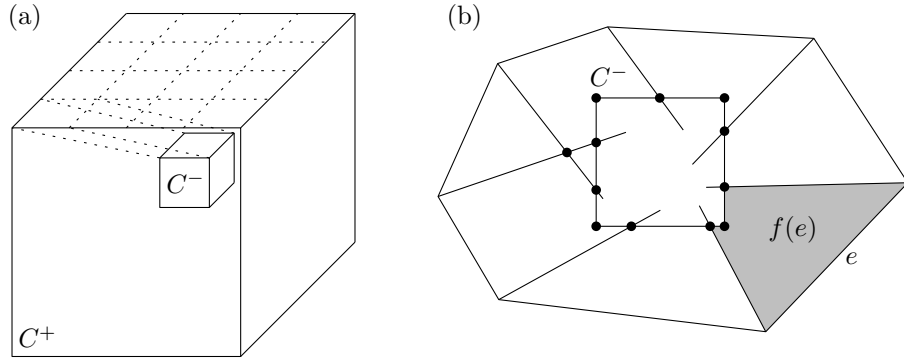
Figure 5.4 (a) One of the canonical directions. (b) Projection of the new witness edges and witness vertices.

of $C^-(P_i)$ and a square on a facet of $C^+(P_i)$ such that v is hit when sweeping f to that square.

We also add some vertices to P_i that we call *witness vertices*, as follows—see Figure 5.4(b). For each witness edge e, we add the intersection point between $proj(e)$ and $\partial\, proj(C^-(P_i))$, lifted back to e, to the set of witness vertices for P_i. Moreover, if the projected witness edges of two consecutive silhouette vertices intersect, then we lift the intersection points to one of the two intersecting witness edges (choosing arbitrarily), and make the resulting point a witness vertex. Finally, we add the vertices of $C^-(P_i)$ to the set of witness vertices.

Lemma 5.12 *The witness edges satisfy the properties that*

(i) *Each witness edge has one of the canonical directions.*

(ii) *For any pair of polyhedra P_i and P_j, $\mathrm{proj}(P_i)$ intersects $\mathrm{proj}(P_j)$ if and only if at least one of the following is true:*

- *A projected witness or silhouette vertex of P_i is inside $\mathrm{proj}(P_j)$, or a projected witness or silhouette vertex of P_j is inside $\mathrm{proj}(P_i)$.*

- *A projected witness edge in W_i crosses a projected witness edge in W_j.*

Proof. Property (i) is satisfied by construction. Also, if one of the two conditions in property (ii) is satisfied, then the projections of P_i and P_j must intersect since they share a point. Therefore, we will concentrate on proving that a projected witness edge in W_i crosses a projected witness edge in W_j assuming that $\mathrm{proj}(P_i) \cap \mathrm{proj}(P_j) \neq \emptyset$, and that no projected witness or silhouette vertex of P_i is contained in $\mathrm{proj}(P_j)$ (or vice versa).

Since $\mathrm{proj}(P_i)$ intersects $\mathrm{proj}(P_j)$ and no projected silhouette vertex of one is inside the projection of the other, there must be silhouette edges of $\mathrm{proj}(P_i)$ and $\mathrm{proj}(P_j)$ that cross. Take one such pair of edges and call them e_i and e_j. Consider the arrangement induced by the projections of the silhouette edges and the witness edges of P_i, and let $f(e_i)$ denote the (bounded) face in this arrangement with e_i on its boundary—see Figure 5.4(b). Define $f(e_j)$ similarly for the arrangement induced by the projections of the silhouette edges and the witness edges of P_j. By Lemma 5.2, there must be an intersection between a pair of edges from $f(e_i)$ and $f(e_j)$, neither of which is $\mathrm{proj}(e_i)$ or $\mathrm{proj}(e_j)$. Hence, there must be an intersection between two projected witness edges. $\qquad\square$

It follows from Lemma 5.12 that there is an object from L_1 below an object from L_2 when at least one of the following conditions holds for some pair P_i, P_j with $P_i \in L_1$ and $P_j \in L_2$: a witness or silhouette vertex of P_i is below P_j, or a witness or silhouette vertex of P_j is above P_i, or a witness edge of P_i is below a witness edge of P_j. To test for the first condition, we construct a data structure for vertical ray shooting on the objects in L_2 and query it with upward rays from the witness and silhouette vertices of the objects in L_1. The second condition can be tested similarly, namely by constructing a data structure for vertical ray shooting on the objects in L_1 and query it with downward rays from the witness and silhouette vertices of the objects in L_2. By Theorem 4.8 and the fact that the total number of witness and silhouette vertices is $O(n)$, this will take $O((1/\beta^2)\log^2 n)$ in total. To test the third condition we proceed as follows. Let $W(L_1)$ and $W(L_2)$ denote the set of all witness edges defined for the objects in L_1 and L_2, respectively. We will preprocess $W(L_2)$ into a data structure for querying with witness edges from $W(L_1)$, according to the following lemma.

Lemma 5.13 *We can preprocess the set $W(L_2)$ in $O((1/\beta^2)n\log n)$ time into a data structure of size $O((1/\beta^2)n\log n)$ such that, for any witness edge $e \in W(L_1)$, we can determine in $O((1/\beta^2)\log^2 n)$ time whether any witness edge from $W(L_2)$ is above e.*

Proof. Let $W_{\vec{d}}(L_2) \subset W(L_2)$ denote the subset of witness edges of canonical direction $\vec{d}$. Note that

$$\sum_{\vec{d}} |W_{\vec{d}}(L_2)| = |W(L_2)| = O(n).$$

Define $W_{\vec{d}}(L_1)$ similarly. For each pair of directions $\vec{d}_1, \vec{d}_2$ we build a data structure on $W_{\vec{d}_1}(L_2)$ for querying with edges from $W_{\vec{d}_2}(L_1)$. (In fact, the structure can be queried with any segment with direction $\vec{d}_2$.) Assume without loss of generality that $\vec{d}_1$ is parallel to the x-axis and $\vec{d}_2$ is parallel to the y-axis. The structure is a multi-level data structure similar to the structure of Section 5.4. The first two levels are exactly the same as for the structure in Section 5.4: the first level is a segment tree on the x-ranges of the witness edges, and the second level is a balanced binary search tree on their y-values (in Section 5.4 this was called the slab tree). For each canonical subset of a node in the slab tree, we store the witness edge with the highest y-coordinate. Note that the witness edge with the highest y-coordinate is a single edge since the witness edges in the canonical subset all have the same direction and the query edge will have a fixed direction as well.

81

A query with a witness edge $e \in W_{\vec{d}_2}(L_1)$ can be answered in $O(\log^2 n)$ time, as follows: query with the x-coordinate of e in the segment tree, for each node ν on the path query with the y-range of e in the associated slab tree $\mathcal{T}(\nu)$, and for each selected node μ in $\mathcal{T}(\nu)$ check if the witness stored there is above e.

When we are querying with an edge e, we actually have to query in the sets $W_{\vec{d}}(L_2)$ for each canonical direction $\vec{d}$. Since there are $O(1/\beta^2)$ canonical directions this means that the total query time is $O((1/\beta^2) \log^2 n)$.

If we let $s := |W_{\vec{d}_1}(L_2)|$, building the structure on $W_{\vec{d}_1}(L_2)$ for a given query direction $\vec{d}_2$ can be done in $O(s \log s)$ time. This is because the associated structures of the segment tree (the slab trees) can be built in linear time if we pre-sort the witness edges on y-cooordinate. After that we compute the highest edge for each node in a slab tree in a bottom up fashion—the highest edge for a node is the higher of the highest edges of its two children—in linear time. Hence, the overall preprocessing time is the same as the amount of storage, which is $O(s \log s)$. Overall, the preprocessing is therefore

$$\sum_{\vec{d}_1, \vec{d}_2} O(|W_{\vec{d}_1}(L_2)| \log |W_{\vec{d}_1}(L_2)|)$$
$$= O(1/\beta^2) \cdot \sum_{\vec{d}_1} O(|W_{\vec{d}_1}(L_2)| \log |W_{\vec{d}_1}(L_2)|)$$
$$= O((1/\beta^2)n \log n).$$

$\square$

Putting everything together, we get the following theorem.

Theorem 5.14 *We can verify whether a given order on a set of n disjoint convex constant-complexity β-fat polyhedra in $\mathbb{R}^3$ is a valid depth order in $O((1/\beta^2)n \log^3 n)$ time.*

5.6 Conclusion

We have presented new and improved solutions to two problems on fat convex polyhedra in 3-space: computing depth orders, and verifying depth orders. One open problem is to see if the results can be extended to fat non-convex polyhedra, or fat curved objects.

Our algorithm for verifying depth orders uses a collection of witness edges that have canonical directions in 3D and allow us to capture (together with a certain set of points in the objects) the above-below relation between the objects. It would be interesting to investigate if these witness edges can be useful for other problems on convex fat objects as well.

CHAPTER 6

Visibility maps

6.1   Introduction

Hidden-surface removal is an important and well-studied computational-geometry problem with obvious applications in computer graphics. The problem is to find those portions of objects in a scene that are visible from a given viewpoint. There are two main approaches to the hidden-surface removal problem: the *image-space approach* and the *object-space approach*. In the former, one calculates the visible object for each pixel of the image; the well known Z-buffer algorithm is the standard example of this. In the latter, one computes the so-called *visibility map* of the scene, which gives an exact description of the visible part of each object; this is the approach taken in computational geometry.

Formally, the visibility map of a set $\mathcal{P}$ of objects in $\mathbb{R}^3$ with respect to a viewpoint p is defined as the subdivision of the viewing plane into maximal regions such that in each region a single object in $\mathcal{P}$ is visible from p, or no object is visible. We will assume in this chapter, as is usual, that the objects are disjoint. The visibility map of a set of n constant-complexity objects can be computed in $O(n^2)$ time [67]. Since the (combinatorial) complexity of the visibility map can be $\Omega(n^2)$—a set of n long and thin triangles that form a grid-like pattern when projected on the viewing plane is an example—this is optimal in the worst case. In most cases, however, the complexity of the visibility map is much smaller than quadratic. Therefore the main challenge in the design of algorithms for computing visibility maps has been to obtain *output-sensitive* algorithms: algorithms whose running time depends not only on the complexity of the input, n, but also on the
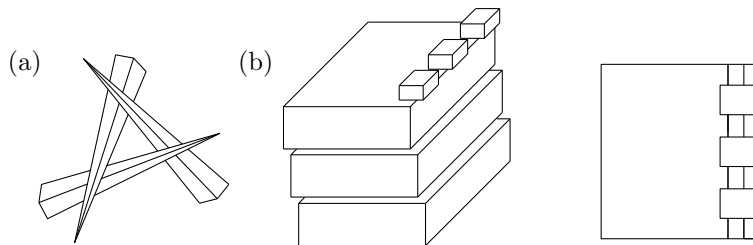
Figure 6.1 (a) The visibility map of a scene with cyclic overlap. (b) The visibility map of fat boxes can have quadratic complexity. Left: the scene. Right: the visibility map for $p = (0, 0, \infty)$.

complexity of the output (that is, the visibility map), k. Ideally the running time should be near-linear in n and k.

The first output-sensitive algorithms for computing visibility maps only worked for polygons parallel to the viewing plane or for the slightly more general case that a depth order on the objects exists and is given [15, 53, 54, 80, 81, 88]. Unfortunately a depth order need not exist since there can be cyclic overlap among the objects[1]—see Figure 6.1 (a). De Berg and Overmars [38] (see also [28]) developed a method to obtain an output-sensitive algorithm that does not need a depth order. When applied to axis-parallel boxes (or, more generally, c-oriented polyhedra) it runs in $O((n + k) \log n)$ time [38] and when applied to arbitrary triangles it runs in $O(n^{1+\varepsilon} + n^{2/3+\varepsilon} k^{2/3})$ time [6]. Unfortunately, the running time for the algorithm when applied to arbitrary triangles is not near-linear in n and k; for example, when $k = n$ the running time is $O(n^{4/3+\varepsilon})$. For general curved objects no output-sensitive algorithm is known,[2] not even when a depth order exists and is given.

In this chapter we study the hidden-surface removal problem for so-called *fat objects*—see Chapter 1 for a definition of fatness. As illustrated in Figure 6.1(b), the complexity of the visibility map of fat objects can still be $\Theta(n^2)$, so also here the main challenge is to obtain an output-sensitive algorithm. Since hidden-surface removal has been widely studied in computational geometry, it is not surprising that it has also been studied for fat objects: Katz *et al.* [60] gave an algorithm with running time $O((U(n)+k) \log^2 n)$, where $U(m)$ denotes the maximum complexity of the union of the projection onto the viewing plane of any subset of m objects. Since $U(m) = O(m \log \log m)$ for fat polyhedra [76] and $U(m) = O(\lambda_{s+2}(m) \log^2 m)$ for fat curved objects [30], their algorithm is near-

---

[1]One might be tempted to try to cut the input objects until they have a depth order. This is probably not such a good idea because $\Omega(n^{3/2})$ cuts are required for some examples [21]. Also, it has recently been shown [10] that minimizing the number of cuts that removes a depth order is NP-complete.

[2]Some of the algorithms can be generalized to curved objects using standard techniques. The resulting algorithms are not very efficient, however, and typically have running time close to quadratic even when the visibility map has linear complexity.

linear in n and k. (Here $\lambda_{s+2}(n)$ is the maximum length of an $(n, s + 2)$ Davenport-Schinzel sequence; $\lambda_{s+2}(n)$ is almost linear in n, for any constant s.) However, the algorithm only works if a depth order exists and is given. This leads to the main question we wish to answer: is it possible to obtain an output-sensitive hidden-surface removal algorithm for fat objects that is near-linear in n and k and does not need a depth order on the objects? We answer this question affirmatively by giving an algorithm with running time $O((n + k) \operatorname{polylog} n)$ for fat convex objects of constant complexity. More precisely, the running time is $O((n \log n (\log \log n)^2 + k) \log^3 n)$ when the objects are polyhedra, and it is $O((n \log^{5+\varepsilon} n + k) \log^3 n)$ when the objects are curved.

The only previously known method for output-sensitive hidden-surface removal that can handle objects without depth order [28, 38] needs an auxiliary data structure for ray shooting in so-called *curtains*—these are semi-infinite surfaces, extending downward from the edges of the input objects—and it appears to be difficult to profit from the fact that the objects are fat when implementing this data structure. This also explains why there is currently no efficient output-sensitive algorithm for hidden-surface removal in curved objects: there are no efficient data structures known for ray shooting (with curved rays, in this case) in curved curtains. Although our algorithm borrows some ideas from this method—we describe the necessary preliminaries in Section 6.2—we therefore proceed differently. Instead of building a data structure for ray shooting in curtains in 3D, we project the rays and the objects onto planes "in between" the objects and the rays. Then ray shooting boils down to tracing the rays on these planes similar to the line-segment-intersection algorithm of Bentley and Ottmann [14]. To make this work, we need a collection of planes such that for every ray and object one of the planes separates them. For this we use a binary space partition on the objects. Section 6.3 describes all of this in detail. We conclude the chapter in Section 6.4 by mentioning some open problems.

6.2 Preliminaries

Visibility maps. Next we define some notation and terminology relating to visibility maps. We assume from now on that we are looking at the scene from above with the viewpoint at $z = \infty$; hence, we are dealing with a parallel view. As already mentioned, the visibility map $\mathcal{M}(\mathcal{P})$ of a set $\mathcal{P}$ of objects is the subdivision of the viewing plane into maximal regions such that in each region a single object in $\mathcal{P}$ is visible from the viewpoint p, or no object is visible. We assume without loss of generality that the viewing plane is the xy-plane.

Consider an object $o \in \mathcal{P}$. We denote the (orthogonal) projection of o onto the viewing plane by $\operatorname{proj}(o)$. Since o is convex, the boundary of $\operatorname{proj}(o)$ consists of the projection of all points of vertical tangency of o. Let $\sigma(o)$ denote the curve[3] on the boundary of o that projects onto the boundary of $\operatorname{proj}(o)$. Note that if o is polyhedral, $\sigma(o)$ consists of

---

[3] For simplicity of presentation we assume o does not have any vertical facets, so that $\sigma(o)$ is uniquely defined. It is easy to adapt the definitions to the general case.
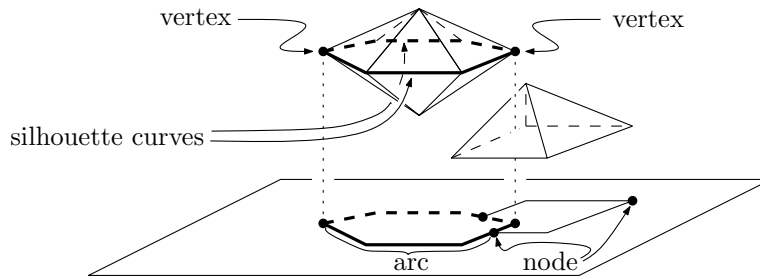
Figure 6.2 A scene consisting of two polyhedral objects, and their visibility map. For one of the objects, its silhouette curves and vertices are indicated in bold. One arc and two nodes of the visibility map are indicated explicitly, but in total the visibility map has six arcs and five nodes.

certain edges of o. We cut $\sigma(o)$ into two pieces at the points of minimum and maximum x-coordinate; we can assume without loss of generality that these points are unique. We call these pieces *silhouette curves*. Note that for polyhedral objects a silhouette curve consists, in general, of multiple edges of the object—see Figure 6.2. The endpoints of the silhouette curves are called *vertices*.

$\mathcal{M}(\mathcal{P})$ is a plane graph whose *nodes* are intersection points of projected silhouette curves and whose *arcs* are portions of projected silhouette curves. Arcs of the visibility map will be denoted by a, and silhouette curves by e. The curve whose projection contains the arc a is denoted $e(a)$. Note that a single silhouette curve can induce more than one arc, so for two arcs a, a' we can have $e(a) = e(a')$. It will be convenient to also consider the projections of visible endpoints of silhouette curves (that is, visible vertices) as nodes, as indicated in Figure 6.2. Since we cut $\sigma(o)$ into two pieces when it changes direction with respect to the x-axis, the arcs of $\mathcal{M}(\mathcal{P})$ are x-monotone.

Curtains. For a curve e in $\mathbb{R}^3$ define the *curtain* of e, denoted $\text{curt}(e)$, as the ruled surface constructed by taking a vertical ray pointing downward and moving its starting point from one end of e to the other. Thus, if e is a segment then $\text{curt}(e)$ is an infinite polygon defined by e and two unbounded edges, each parallel to the z-axis. For a set E of curves we let $\text{curt}(E) := \{\text{curt}(e) \mid e \in E\}$.

Computing visibility maps. Our algorithm is based on the existing output-sensitive hidden-surface removal algorithm from [28]. Hence, we give a brief overview of this algorithm.

The algorithm is a plane-sweep algorithm. It sweeps over the viewing plane from left to right, detecting the arcs of the visibility map along the way. There are two types of event points: projections of object vertices (these are known in advance), and nodes of the visibility map (these will be computed as the sweep progresses).

86
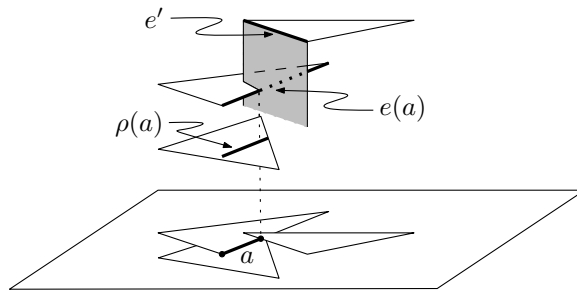
Figure 6.3 The endpoint of arc a is the intersection of $\text{proj}(e(a))$ and $\text{proj}(e')$, and it corresponds to the ray along $e(a)$ hitting $\text{curt}(e')$. (Note that the objects pictured here are not fat, but could be the top surfaces of fat polyhedra. We draw the objects in this way to ease visualization.)

When the projection of an object vertex v is reached by the sweep line, the algorithm checks whether v is visible. This is done by shooting a ray from v vertically upward. The vertex v is visible if and only if no object is hit by the ray. (Thus the algorithm needs a supporting data structure that can answer vertical ray shooting queries such as the one in Chapter 4.) If v is visible, its projection is a node of the visibility map. This node will then be treated as an event for the sweep, as described next.

When a node of the visibility map is reached by the sweep line, the algorithm proceeds as follows. First the arcs ending at that node—this information can easily be maintained—are reported. Next it is determined whether any new arcs start at the node, that is, whether any arcs have the node as their left endpoint. This can be decided based on the two silhouette curves defining the node. For each new arc a, its right endpoint is computed and inserted into the event queue.

It remains to explain how to compute the right endpoint of a given arc a of the visibility map. An arc a can end for two reasons. One is that the silhouette curve $e(a)$ defining a ends. The other is that $\text{proj}(e(a))$ intersects some other projected silhouette curve $\text{proj}(e')$ such that either $e(a)$ becomes invisible or e' becomes visible—see Figure 6.3. This can be detected by a ray shooting in a set of curtains, as described next. When $e(a)$ becomes invisible because it disappears below some object o, then the ray along $e(a)$ must hit the curtain hanging from one of o's silhouette curves. When some other silhouette curve e' becomes visible, something similar holds. To this end, we define a ray[4] $\rho(a)$ for an arc a of the visibility map as follows. Let q be the point on $e(a)$ projecting onto the left endpoint of a. Project the portion of $e(a)$ to the right of q onto the object $o(q)$ immediately below q. (If there is no such object, we project onto a plane below all objects.) This gives us a ray on the surface of $o(q)$ whose projection contains a. It can be argued [28] that the point where $\rho(a)$ hits $\text{curt}(e')$ corresponds to the point where the silhouette curve e' becomes visible. Since any curtain hit by the ray along $e(a)$ is also hit by $\rho(a)$—after

---

[4]Note that in case of curved objects, the ray will be curved.

all, $\rho(a)$ is below $e(a)$—we can detect events where $e(a)$ becomes invisible by shooting along $\rho(a)$ as well.

The next lemma summarizes the discussion above.

Lemma 6.1 (De Berg [28]) *Let E be the set of silhouette curves of the objects in $\mathcal{P}$. The right endpoint of an arc a of $\mathcal{M}(\mathcal{P})$ is the leftmost of the following event points:*

- *The projection of the right endpoint of $e(a)$.*

- *The projection of the first intersection of $\rho(a)$ with a curtain in $\mathrm{curt}(E)$.*

6.3   The algorithm

As mentioned in the introduction to this chapter, it seems hard to implement a structure for ray shooting in curtains that profits from the fact that the objects are fat. Therefore we use the following idea.

Consider a collection of curtains hanging from the silhouette curves of some set of objects that are all above a plane h. Now suppose we want to do ray shooting in those curtains with a query ray $\rho(a)$ that lies below h. Then we can project all objects and the ray onto h, and shoot with the projected ray in the union of the projected objects; the point where the ray first hits a curtain then corresponds to the point where the projected ray hits the union. This is true because in our application the ray will always be visible, so the projected ray cannot start inside the union. Unfortunately two-dimensional ray shooting is still too costly. If, however, we have to answer many queries, then we can project all of them onto h, and perform a sweep to detect when they intersect the union. Of course there will not be a plane h that separates all objects from all rays. Therefore we construct a binary space partition (a *BSP*, for short) on the objects. This will give us a collection of $O(\log n)$ planes that together separate any ray from all the objects. The ray will then be traced on each of these planes. Below, we make this idea more precise.

We start by describing the BSP in Section 6.3.1, then discuss in Section 6.3.2 the correspondence between ray shooting in curtains and tracing rays on a suitable set of planes, and finally we give the details of the algorithm in Section 6.3.3.

6.3.1   The data structure

Recall that a *balanced aspect ratio tree* (or *BAR-tree* for short) is a special type of BSP for storing points. The variant known as the *object BAR-tree* [40] stores objects rather than points and has proved especially useful in designing data structures for fat objects. It has been used as a basis for vertical ray shooting in Chapter 4 and [31] as well as approximate range searching and nearest neighbor searching [40].

We denote the region associated with a node ν in the object BAR-tree for $\mathcal{P}$ by $region(\nu)$, and we let $\mathcal{P}_\nu$ denote the set of all objects $o \in \mathcal{P}$ intersecting $region(\nu)$, clipped to $region(\nu)$. The following lemma states the properties of the object BAR-tree we will need.

Lemma 6.2 (De Berg and Streppel [40]) *Let $\mathcal{P}$ be a set of n β-fat disjoint convex objects in $\mathbb{R}^d$. An object BAR-tree on $\mathcal{P}$ is a BSP tree $\mathcal{T}$ for $\mathcal{P}$ with the following properties:*
(i) the tree has $O(n)$ leaves and each leaf region intersects $O(1/\beta)$ objects from $\mathcal{P}$;
(ii) the depth of the tree is $O(\log n)$;
(iii) for each node ν, $region(\nu)$ has constant complexity and fatness.

De Berg [31] has shown how to augment an object BAR-tree $\mathcal{T}$ with secondary structures, so that vertical ray shooting can be performed efficiently. The augmentation is as follows.

- For each leaf node μ of $\mathcal{T}$, we store the set $\mathcal{P}_\mu$ in a list $\mathcal{L}_\mu$.

- For an internal node ν, let h_ν denote the splitting plane stored at ν.

    - If h_ν is vertical, then we store the set $\{h_\nu \cap o : o \in \mathcal{P}_\nu\}$—that is, the cross-sections of the polyhedra in $\mathcal{P}_\nu$ with h_v—in a structure $\mathcal{T}_\nu$, which is an optimal point-location structure [62] on the trapezoidal map defined by $h_\nu \cap \mathcal{P}_\nu$.

    - If h_ν is not vertical, then ν has two associated data structures, $\mathcal{T}_\nu^+$ and $\mathcal{T}_\nu^-$, defined as follows.
    Let $\mathcal{P}_\nu^+$ denote the set of object parts from $\mathcal{P}_\nu$ lying above h_ν. Thus $\mathcal{P}_\nu^+ = \mathcal{P}_\mu$, where μ is the child of ν corresponding to the region above h_ν. Let $\mathrm{proj}(\mathcal{P}_\nu^+)$ denote the set of vertical projections of the objects in $\mathcal{P}_\nu^+$ onto h_ν. Then $\mathcal{T}_\nu^+$ is an optimal point-location structure for $U(\mathrm{proj}(\mathcal{P}_\nu^+))$, the union of $\mathrm{proj}(\mathcal{P}_\nu^+)$. In our application, we not only store the point-location structure for $U(\mathrm{proj}(\mathcal{P}_\nu^+))$, but also an explicit list of all union edges.
    The associated structure $\mathcal{T}_\nu^-$ is defined similarly, but this time for the object parts below h_ν.

Lemma 6.3 (De Berg [31]) *The augmented object-BAR-tree data structure above requires $O((\frac{1}{\beta^5} \log^2 \frac{1}{\beta})n \log^3 n (\log \log n)^2)$ storage and $O((\frac{1}{\beta^5} \log^2 \frac{1}{\beta})n \log^4 n (\log \log n)^2)$ preprocessing time for convex β-fat polyhedral objects, and $O(\frac{1}{\beta^{14}} n \log^{7+\varepsilon} n)$ storage and $O(\frac{1}{\beta^{14}} n \log^{8+\varepsilon} n)$ preprocessing time for convex β-fat curved objects. With this structure, we can answer vertical ray-shooting queries in $O(\log^2 n + 1/\beta)$ time.*

Recall that we want to use the structure not only to answer vertical ray-shooting queries in the given set of objects, we also want to use it for ray shooting in the curtains hanging from the objects' silhouette curves. The idea is as follows. Suppose that the query ray ρ is located inside the region of some leaf μ. Then any object above ρ (except for the $O(1/\beta)$

89

objects stored at μ) will be separated from ρ by some of the splitting planes stored at nodes on the path to μ. Hence, the ray shooting query can be answered by tracing ρ in the unions stored at these nodes.

There is one problem with this approach, however. The query rays are along the projections of (parts of) silhouette curves onto the object immediately below them. These objects and, hence, the query rays can be cut into many pieces by the BSP.[5] At the points where a ray is cut into pieces, it moves to a different leaf region. Then we would have to trace the ray on a different set of planes, because the path from the root changes—something we cannot afford.

To avoid this problem we proceed as follows. Let $\partial_{\text{top}}(o)$ denote the top surface of an object o, that is, the part of the boundary of o visible from above. For each object $o \in \mathcal{P}$ we will store the union of the projection of a certain subset $\mathcal{P}(o) \subset \mathcal{P}$ onto $\partial_{\text{top}}(o)$. The subset $\mathcal{P}(o)$ is defined as follows. Call an object o *large* at a node ν of $\mathcal{T}$ if o intersects $region(\nu)$ and the following two conditions are met: (i) $\text{size}(o) < \text{size}(region(\text{parent}(\nu)))$ and (ii) either $\text{size}(o) \geq \text{size}(region(\nu))$ or ν is a leaf. Now we define

$$\mathcal{P}(o) := \{ \, o' \in \mathcal{P} : \text{there is a node } \nu \text{ such that } o \text{ is large at } \nu,$$
$$o' \text{ intersects } region(\nu), \text{ and } o' \text{ is above } o \, \}$$

Finally, we also store the union of the projections of all the objects in $\mathcal{P}$ onto the xy-plane. (The xy-plane can be seen as a dummy object added below the whole scene, which is large at the root of $\mathcal{T}$.)

Next we analyze the cost of the additional information. We need the following lemma.

Lemma 6.4 *Any object $o \in \mathcal{P}$ is large at $O(\log n)$ nodes, and at any node ν there are $O(1/\beta)$ large objects.*

Proof. By Lemma 6.2(iii) we know that every cell of $\mathcal{T}$ is $O(1)$-fat. Lemma 1.5 then implies that any collection of disjoint cells has density $O(1)$. Therefore, since the cells at any level of the BAR-tree are disjoint, the number of nodes ν in any level of the BAR-tree intersecting some $o \in \mathcal{P}$ with $\text{size}(region(\nu)) \geq \text{size}(o)$ is $O(1)$. An object o can only be large at the node ν if $\text{size}(region(\text{parent}(\nu))) \geq \text{size}(o)$. Thus, the number of cells per level at which o can be large is $O(1)$. Finally we know that $\mathcal{T}$ has $O(\log n)$ levels by Lemma 6.2, proving the first part of the lemma.

To prove the second part of the lemma, we note that a set of disjoint β-fat objects has density $O(1/\beta)$ by Lemma 1.5. This immediately proves that there are only $O(1/\beta)$ large objects at any internal node. For leaf nodes this follows from Lemma 6.2(i). $\square$

Using Lemma 6.4 we can prove a bound on the total size of all sets $\mathcal{P}(o)$.

---

[5]The fact that the objects may be cut into many pieces also prevents us from applying the following simple strategy: compute the object BAR-tree, use it to find a depth order on the resulting set of pieces, and apply the algorithm of Katz *et al.* [60]. The problem is that the visibility map of the pieces may be much more complex than the visibility map of the original objects.

Lemma 6.5 $\sum_o |\mathcal{P}(o)| = O((1/\beta) \cdot n \log n)$.

Proof. We have

$$
\begin{aligned}
\sum_o |\mathcal{P}(o)| \;&\leq\; \sum_\nu \{(\text{\# large objects at } \nu) \cdot (\text{\# objects intersecting } region(\nu))\} \\
&\leq\; O(1/\beta) \cdot \sum_\nu |\mathcal{P}_\nu| \;\leq\; O((1/\beta) \cdot n \log n),
\end{aligned}
$$

where the last inequality follows from [31]. □

Together with the known bounds on the union of fat objects [30, 76] this is easily seen to imply that the total amount of storage and preprocessing time needed to construct the unions of the projections of $\mathcal{P}(o)$ onto the top surfaces $\partial_{\text{top}}(o)$ is upper bounded by the bounds in Lemma 6.3.

6.3.2 Tracing an arc

Recall that the right endpoint of an arc a can be found by shooting with $\rho(a)$ in $curt(E)$. Next we explain how to find the right endpoint of a using the unions stored in $\mathcal{T}$ and the unions on the objects' top surfaces. The key is to find a collection of $O(\log n)$ unions such that the first point where $\rho(a)$ hits a curtain corresponds to the first point where one of the unions is hit.

To this end we first define for a node ν a collection $S^+(\nu)$ of $O(\log n)$ splitting planes:

$$S^+(\nu) := \{\text{ splitting planes } h_{\nu'} : \nu' \text{ is an ancestor of } \nu \text{ and } region(\nu) \text{ is below } h_{\nu'} \}.$$

Let $e(a)$ be the silhouette curve defining an arc a, and let $p \in e(a)$ be the point projecting onto the left endpoint of a. Recall that $\rho(a)$ is a ray on the top surface of the object o directly below p. We denote the projection of p onto o by $\tilde{p}$. The first curtain hit by $\rho(a)$ can now be found using the following lemma.

Lemma 6.6 *Let $\rho(a)$ be a ray on $\partial_{\text{top}}(o)$ and let $\tilde{p}$ be the starting point of $\rho(a)$. Let ν be the node in $\mathcal{T}$ such that $\tilde{p} \in region(\nu)$ and o is large at ν. Then the first curtain from $curt(E)$ inside $region(\nu)$ hit by $\rho(a)$ corresponds to the first of the following events:*

(i) *$\rho(a)$ hits the union of the projection of the objects in $\mathcal{P}(o)$ onto $\partial_{\text{top}}(o)$;*

(ii) *the projection of $\rho(a)$ onto $h_{\nu'}$ hits the union stored on $h_{\nu'}$, for some $\nu' \in S^+(\nu)$.*

Proof. Note that the node ν referred to in the lemma is unique and must exist, since we consider the xy-plane to be a dummy object below the whole scene.

Let $\tilde{q}$ be the first point where $\rho(a)$ intersects a curtain in $curt(E)$, let e be the silhouette curve defining the curtain, and let $q \in e$ be the point directly above $\tilde{q}$. If $q \in region(\nu)$ then the object containing the silhouette curve e is a member of $\mathcal{P}(o)$ and we are in case (i).
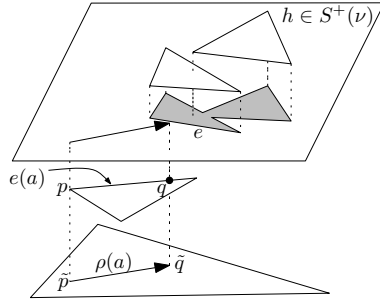
Figure 6.4 $\rho(a)$ hits a curtain in $\mathrm{curt}(E)$ at point q when its projection intersects a silhouette curve of a union stored at $S^+(\nu)$.

Otherwise there is a splitting plane $h_{\nu'}$ stored at some ancestor ν' of ν with q above $h_{\nu'}$ and $\tilde{q}$ below $h_{\nu'}$. Then the relevant portion of e must be part of the union stored at the first such node ν' (as seen from the root of $\mathcal{T}$). See Figure 6.4.

Conversely, since all the unions considered are generated by (portions of) objects above o, we know that $\rho(a)$ cannot hit such a union before it hits a curtain. $\square$

6.3.3 Details of the algorithm

We now describe our algorithm for computing the visibility map of a set $\mathcal{P} = \{o_1, \ldots, o_n\}$ of convex, disjoint, constant-complexity, β-fat objects. The algorithm is a space-sweep algorithm that moves a sweep plane h parallel to the yz-plane from left to right through space. The space sweep induces a plane sweep for each of the unions stored in $\mathcal{T}$. Thus, instead of thinking about the algorithm as a 3D sweep, one may also think about it as a number of coordinated 2D sweeps. That is, while we sweep $\mathbb{R}^3$ with h, we also sweep each (non-vertical) splitting plane h_ν with the line $h \cap h_\nu$. Such a 2D sweep is performed to detect intersections of the union on h_ν with certain rays (projected onto h_ν). The same holds for the unions stored for each object: while we sweep $\mathbb{R}^3$ with h, we sweep the top surface $\partial_{\mathrm{top}}(o)$ of each object o with the curve $h \cap \partial_{\mathrm{top}}(o)$. Finally, the sweep of h induces a sweep on the viewing plane. As in the algorithm from [28], the visibility map will be computed as we go, so that at the end of the sweep the entire visibility map has been computed.

The space-sweep algorithm is supported by the following data structures:

- There is a global event queue Q, where the priority of an event is its x-coordinate. Initially, all vertices of the objects (that is, all endpoints of silhouette curves) are placed into Q. In addition, all vertices of any of the unions stored in $\mathcal{T}$ are placed into Q. During the sweep, new event points will be inserted into Q, for example

endpoints of arcs of the visibility map. It is also possible that events will be removed before they are handled.

- For every splitting plane h_ν (and the top surface of every object o) we maintain a balanced binary tree, which we will call the *intersection-detection data structure*. This tree will store the edges of the union on the splitting plane (resp. $\partial_{\text{top}}(o)$) that intersect the sweep line $h \cap h_\nu$ (resp. $h \cap \partial_{\text{top}}(o)$) as well as the rays traced on it that intersect the sweep line; the edges and rays are stored in order of their intersection with the sweep line. Thus we are essentially running the standard line-segment intersection algorithm of Bentley and Ottmann [14] on the union edges and rays.

Next we discuss the events that can take place, and how they are handled. The first two events are essentially subroutines that we use in the other events.

(i) *The sweep reaches the left endpoint of an arc a.*
Let $e(a)$ be the silhouette curve defining a, and let $p \in e(a)$ be the point whose projection is the left endpoint of a. Let o be the first object that a vertical ray downward from p hits, and let $\tilde{p} \in o$ be the point where o is hit. Finally, let ν be the node where o is large such that $\tilde{p} \in region(\nu)$. Determine $S^+(\nu)$, and insert the portion of $e(a)$ starting at p into each of the intersection-detection data structures associated with the splitting planes in $S^+(\nu)$. (More precisely, the projection of the silhouette curve onto the plane is added.) Also add the projection of the silhouette curve onto $\partial_{\text{top}}(o)$ to the intersection-detection structure for o. Determine any new events using these data structures in the standard way (that is, by checking new pairs of adjacent elements); add any new events to Q. Finally, add the following three events to Q: the right endpoint of $e(a)$, the (first) intersection of $\rho(a)$ with the boundary of $region(\nu)$, and the (first) intersection of $\rho(a)$ with the silhouette of o.

(ii) *The sweep reaches the right endpoint of an arc a.*
Determine ν and o as above. Remove a from all intersection-detection data structures in $S^+(\nu)$ and the intersection-detection data structure associated with o. Remove all events associated with a from Q. Check for new events in each of the intersection-detection data structures; add any new events to Q. Output a as an arc of $\mathcal{M}$. (Note that the right endpoint of an arc may be the left endpoint of one or two other arcs; in this case the left endpoints will be separate events, which are handled according to case (i).)

(iii) *The sweep reaches the left vertex v of a silhouette curve.*
(In other words, we reach the leftmost point of an object $o \in \mathcal{P}$.) Determine if v is visible by shooting a ray vertically up from it. If v is visible, two arcs start at the projection of v onto the viewing plane. Run the actions from case (i) for each of these arcs.

(iv) *The sweep reaches the right vertex v of a silhouette curve it is currently tracing defining an arc currently intersected by the sweep line.*
Run the actions from case (ii) for the arc ending at the projection of v.

93
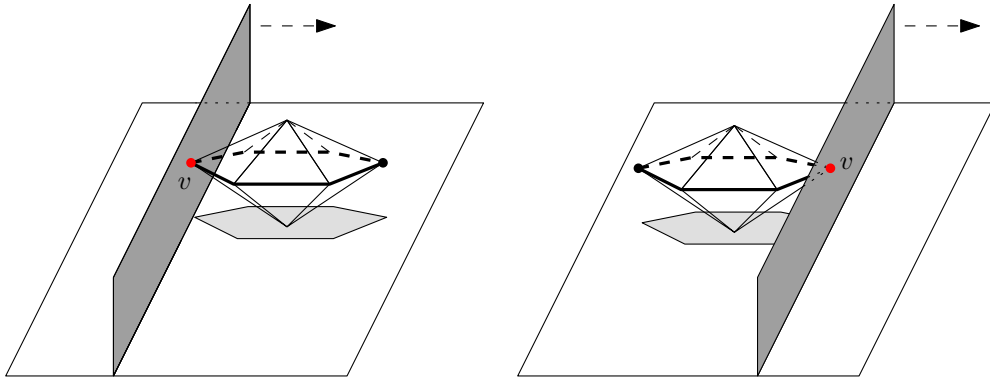
Figure 6.5 Cases (iii) and (iv)
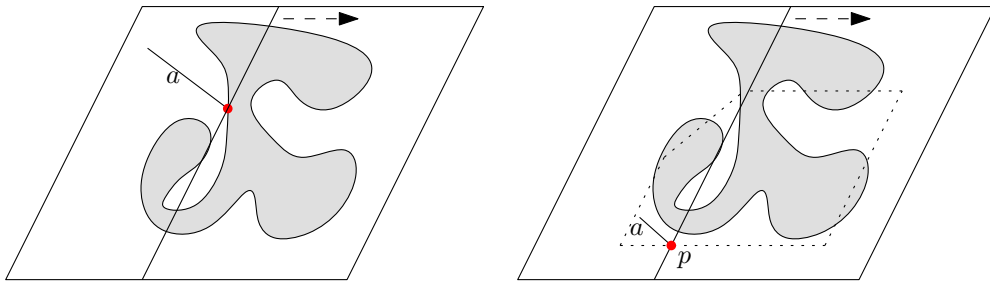
Figure 6.6 Cases (v) and (vi)

(v) *The sweep reaches the intersection point of the union boundary on some splitting plane (or top surface of an object) and an arc a traced on the plane (or top surface).* This case corresponds to a hitting a curtain in $\mathrm{curt}(E)$. Now the arc a ends. Run the actions from case (ii) for a. One or two new arcs may start at this point, at most one along the silhouette curve $e(a)$, and one along the silhouette curve corresponding to the curtain that is hit. Run the action from case (i) for the new arc(s).

(vi) *The sweep reaches a point p where the projection of a currently visible silhouette curve onto the object o below hits the boundary of a cell ν where o is large.* Let a be the arc defined by the silhouette curve. Remove a from all the intersection-detection data structures in $S^+(\nu)$ and all events associated with a from Q. Run the action for case (i) for the continuation of a. (The only thing that happens here is that the set $S^+(\cdot)$ changes, because the ray that we are tracing moves out of a cell where the object o on which the ray is traced is large.)

(vii) *The sweep reaches the point where the object o immediately below a currently visible silhouette curve changes.*

94
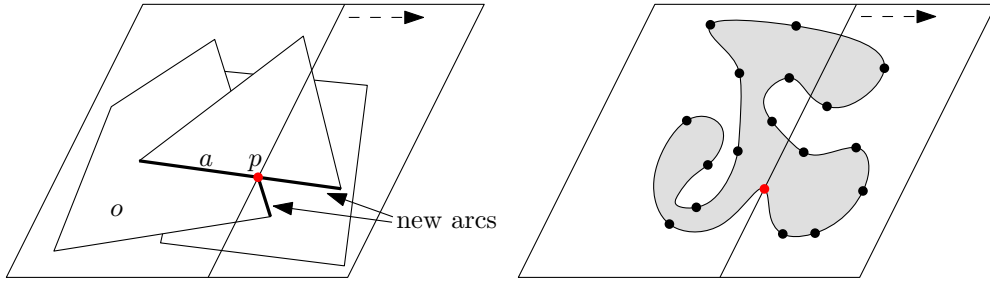
Figure 6.7 Cases (vii) and (viii)

This can be detected because the visible silhouette curve is traced on $\partial_{\text{top}}(o)$, and therefore we also know where it reaches the boundary of the top surface. Note that the projection of the point p where the curve reaches the boundary of the top surface is the right endpoint of an arc a. Run the actions from case (ii) for a. At most two new arcs start at p, one that is the continuation of a, and one that is along a silhouette curve of o (which became visible or stops being visible). Run the actions for case (i) on these curve(s).

(viii) *The sweep reaches a point on a splitting plane (or top surface of an object), where a union edge starts or ends.*
In this case we only have to update the relevant intersection-detection data structure, check for new events in the intersection-detection data structures, and add any new events to Q.

Lemma 6.7 *The number of events of type (i)–(vii) is $O(n + k \log n)$, where k is the complexity of $\mathcal{M}$, and the number of events of type (viii) is $O((\frac{1}{\beta^5} \log^2 \frac{1}{\beta})n \log^3 n (\log \log n)^2)$ for fat polyhedra and $O(\frac{1}{\beta^{14}} n \log^{7+\varepsilon} n)$ for fat curved objects.*

Proof. Clearly, the number of events of types (i), (ii), (iv), (v), and (vii) is $O(k)$, since they can be charged to a vertex of $\mathcal{M}$. The number of events of type (iii) is $O(n)$. It remains to bound the number of events of type (vi). Consider the portion of a silhouette curve $e(a)$ defining some arc a. This portion has a unique object o immediately below it. Since o is large at $O(\log n)$ cells by Lemma 6.4 and the projection of $e(a)$ onto o can leave any cell only a constant number of times, we can conclude that there are only $O(\log n)$ type (vi) events for any arc a, this giving $O(k \log n)$ such events in total.

The bound on the number of events of type (viii) follows from Lemma 6.3. $\qquad \square$

Lemma 6.8 *The time taken for each event of type (i)–(vii) is $O(\log^2 n)$, and the time taken for each event of type (viii) is $O(\log n)$.*

Proof. In all event types, we may need to perform several actions: vertical ray shooting, updating intersection-detection data structures, determining a set $S^+(\nu)$, and updating Q.

By Lemma 6.3, the time taken for the vertical ray shooting is $O(\log^2 n)$. Each event needs to do only a constant number of ray shooting queries, so this is $O(\log^2 n)$ in total. The intersection-detection data structures are balanced binary trees, so updates take $O(\log n)$ time. At each event we have to update $O(\log n)$ intersection-detection data structures, so the total time taken for updating is $O(\log^2 n)$. Determining new events in the intersection-detection data structures takes $O(1)$ per data structure, so the total amount of time taken for events of type (iii) is $O(\log^2 n)$. Determining a set $S^+(\nu)$ can be done in $O(\log n)$ time by searching in $\mathcal{T}$. At each event we may have to remove $O(\log n)$ event points from Q, each removal taking $O(\log n)$ time. Hence, all events of type (i)–(vii) can be handled in $O(\log^2 n)$ time, as claimed.

The events of type (viii) require only $O(\log n)$ time, since they only involve a constant number of operations on a single intersection-detection data structure. $\square$

The correctness of the algorithm follows from Lemmas 6.1 and 6.6 as well as the correctness of the algorithm in [28]. We conclude with the following theorem.

Theorem 6.9 *The visibility map of a set of n disjoint constant-complexity convex β-fat polyhedra in $\mathbb{R}^3$ can be computed in time $O((\frac{1}{\beta^5} \log^2 \frac{1}{\beta})(n \log n (\log \log n)^2 + k) \log^3 n)$, where k is the complexity of the visibility map. When the objects are curved (and disjoint, constant-complexity, convex, and β-fat) the visibility map can be computed in time $O(\frac{1}{\beta^{14}}(n \log^{5+\varepsilon} n + k) \log^3 n)$.*

6.4   Conclusion

We presented the first algorithm to compute the visibility map of a set of fat convex objects that does not need a depth order and that runs in $O((n+k) \text{ polylog } n)$ time.

One obvious open problem is to further reduce the running time, either by getting rid of some logarithmic factors or by reducing the dependency on the fatness factor β (which is currently quite bad). A second open problem is to extend the results to non-convex objects. Finally, it would be very interesting to come up with an approach that works for low-density scenes, and not just for fat objects. The main problem here is that the union complexity of the projection of a low density scene can be $\Omega(n^2)$, so the approach would need to use a different data structure than the one presented in this chapter.

Concluding remarks

In this thesis, we have looked at some computational-geometry problems in the context of fat objects. We first studied decompositions in two and three dimensions. We then gave algorithms and data structures related to three different problems inspired by computer graphics: ray shooting, depth orders, and hidden-surface removal.

We introduced the technique of decomposing objects into towers in Chapter 3. We showed its utility in ray shooting, range searching, and in verifying depth orders. We believe that this technique has potential for use in other situations as well. Also, given that any (α, β)-covered polyhedron can have its boundary covered by $O(1)$ towers, it seems likely that any algorithm that operates on towers can be extended to (α, β)-covered polyhedra without any extra asymptotic cost. This provides extra incentive to work with towers, since most algorithms for fat objects only apply to objects that are also convex—often an unreasonable extra restriction.

Other techniques that could potentially be useful in the future are the witness edges from Chapter 5 that give us an easy test of the above/below relation for fat polyhedra and the simulation of a space sweep by plane sweeps that we employed in Chapter 6. Moreover, we believe that the technique of designing algorithms for polygons that have a small set of guards, such as in Chapter 2, could be interesting on its own.

We conclude by stating some problems that have arisen from this work that would be exciting to see solved.

Vertical ray shooting. Our first open problem concerns vertical ray-shooting in non-convex objects. We would like to have a data structure that has properties (in terms of query time and space complexity) similar to those in Section 4.3. This would greatly improve the algorithms that we have for hidden-surface removal and depth-order computation in the context of non-convex objects. Our current solution relies on covering the boundaries of the input by convex fat objects. At the moment we can only do this for constant-complexity (α, β)-covered polyhedra. We think that it should be possible to perform these queries in a more general input model. There could be two ways of achieving this. One possibility is that we could improve the results of Chapter 3. Another possibility is that a different algorithm could be devised that operates directly on non-convex objects.

Kinetic data structures. All of the problems that we have studied in this thesis have been for objects that are stationary. In many applications, such as in video games and movies, the objects in the scene move. One way of dealing with such motion is known as a kinetic data structure [13]. A strategy that is perhaps more commonly used, known as time-slicing, is to recompute everything in regular increments (such as for every new frame). In contrast, a kinetic data structure attempts to update only when a change is required.

As an example, one popular application for kinetic data structures is collision detection. A kinetic data structure for collision detection keeps a set of *certificates* that the objects in the scene have not collided as well as a queue of times when the certificates could potentially fail (the objects could change course, for example). When a time in the queue is reached, the data structure is updated. Kinetic data structures have been studied in the context of realistic input models before—a kinetic data structure for collision detection amongst fat objects in $\mathbb{R}^3$ has recently been proposed [1].

We would like to know whether it is possible to create an efficient kinetic data structure for realistic input in any of the problems we studied related to computer graphics. Clearly the problems would need to be changed slightly in order to make sense in a kinetic context: a kinetic data structure for the visibility-map problem, for example, would need to be updated only when the visibility map changes combinatorially.

We feel the problem that has the most potential in this regard is that of finding the depth order of a set of objects. This is because of the result in Section 5.3. Since, as we showed in that section, the size of the transitive reduction of the depth-order graph is not too large, it might be possible to compute the graph and only change it when the comparability (in the above/below relation) of a pair of objects changes.

Dynamic data structures. Related to the question of whether a kinetic data structure can be built, we also wonder whether dynamic data structures can be built for any of the computer-graphics problems that we studied. A dynamic data structure would need to support insertion and deletion operations. In some cases a dynamic data structure can be easily turned into a kinetic data structure. The method for doing this is to update the data structure as needed by deleting and reinserting affected objects.

Practicality. Finally, it remains to be seen which of the algorithms and data structures presented here are of practical interest. With the possible exception of the data structure for performing ray-shooting queries in arbitrary directions from Chapter 4 (which uses parametric search), all the data structures that we present are certainly implementable. We would like to see experiments comparing these algorithms with the current state of the art.

Such experiments might give extra insight into how realistic our realistic input models actually are. For example, some of the algorithms that we present in this thesis have time complexities with rather high dependencies on the fatness constant—$O(1/\beta^{14})$ in one case. In some cases, the dependence on the fatness constant is inherent in the algorithm: making a data structure for every pair of canonical directions in our depth-order algorithm, for example, can not be helped. In other cases, the dependence on the fatness constant is partially an artifact of a proof. Performing experiments is one way to evaluate whether these proofs should be targeted for improvement.

Index

References

[1] Mohammad Ali Abam, Mark de Berg, Sheung-Hung Poon, and Bettina Speckmann. Kinetic collision detection for convex fat objects. In Yossi Azar and Thomas Erlebach, editors, *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 4–15. Springer, 2006.

[2] Pankaj K. Agarwal. Range searching. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, 1997.

[3] Pankaj K. Agarwal, Mark de Berg, Dan Halperin, and Micha Sharir. Efficient generation of k-directional assembly sequences. In *SODA '96: Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 122–131, Atlanta, Georgia, 28–30 January 1996.

[4] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. In Bernard Chazelle, Jacob E. Goodman, and Richard Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 23, pages 1–56. American Mathematical Society, 1998.

[5] Pankaj K. Agarwal, Matthew J. Katz, and Matthew Sharir. Computing depth orders for fat objects and related problems. *Computational Geometry: Theory and Applications*, 5:187–206, 1995.

[6] Pankaj K. Agarwal and Jiří Matoušek. Ray shooting and parametric search. *SIAM Journal on Computing*, 22(4):794–806, 1993.

[7] Pankaj K. Agarwal and Jiří Matoušek. On range-searching with semi-algebraic sets. *Discrete and Computational Geometry*, 11:393–418, 1993.

[8] Boris Aronov, Mark de Berg, and Chris Gray. Ray shooting and intersection searching amidst fat convex polyhedra in 3-space. In *SCG '06: Proceedings of the Twenty-Second Annual Symposium on Computational geometry*, pages 88–94, New York, NY, USA, 2006. ACM Press.

[9] Boris Aronov, Mark de Berg, and Chris Gray. Ray shooting and intersection searching amidst fat convex polyhedra in 3-space. *Computational Geometry: Theory and Applications*, 41:68–76, 2008.

[10] Boris Aronov, Mark de Berg, Chris Gray, and Elena Mumford. Cutting cycles of rods in space: hardness and approximation. In *SODA '08: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1241–1248, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[11] Boris Aronov, Alon Efrat, Vladlen Koltun, and Micha Sharir. On the union of κ-round objects in three and four dimensions. *Discrete and Computational Geometry*, 36:511–526, 2006.

[12] Boris Aronov and Micha Sharir. On translational motion planning of a convex polyhedron in 3-space. *SIAM Journal on Computing*, 26(6):1785–1803, 1997.

[13] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, April 1999.

[14] Jon L. Bentley and Thomas A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, 28(9):643–647, 1979.

[15] Marshall Bern. Hidden surface removal for rectangles. *Journal of Computer and System Sciences*, 40(1):49–69, February 1990.

[16] John F. Canny. *The complexity of robot motion planning*. PhD thesis, Massachussetts Institute of Technology, 1988.

[17] Donald R. Chand and Sham S. Kapur. An algorithm for convex polytopes. *Journal of the ACM*, 17(1):78–86, January 1970.

[18] Bernard Chazelle. Convex partitions of polyhedra: A lower bound and worst-case optimal algorithm. *SIAM Journal on Computing*, 13(3):488–507, August 1984.

[19] Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6(5):485–524, 1991.

[20] Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete and Computational Geometry*, 9:145–158, 1993.

[21] Bernard Chazelle, Herbert Edelsbrunner, Leonidas J. Guibas, Richard Pollack, Raimund Seidel, Micha Sharir, and Jack Snoeyink. Counting and cutting cycles of lines and rods in space. *Computational Geometry: Theory and Applications*, 1(6):305–323, 1992.

[22] Bernard Chazelle, Herbert Edelsbrunner, Leonidas J. Guibas, Micha Sharir, and Jorge Stolfi. Lines in space: Combinatorics and algorithms. *Algorithmica*, 15(5):428–447, 1996.

[23] Bernard Chazelle and Joel Friedman. Point location among hyperplanes and unidirectional ray-shooting. *Computational Geometry: Theory and Applications*, 4:53–62, 1994.

[24] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[25] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1(2):163–191, 1986.

[26] Bernard Chazelle, Leonidas J. Guibas, and Der-Tsai Lee. The power of geometric duality. *BIT*, 25(1):76–90, 1985.

[27] Bernard Chazelle and Janet Incerpi. Triangulation and shape-complexity. *ACM Transactions on Graphics*, 3(2):135–152, 1984.

[28] Mark de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*. Springer-Verlag New York, LNCS 703, 1993.

[29] Mark de Berg. Linear size binary space partitions for uncluttered scenes. *Algorithmica*, 28:353–366, 2000.

[30] Mark de Berg. Improved bounds on the union complexity of fat objects. In Ramaswamy Ramanujam and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings*, volume 3821 of *Lecture Notes in Computer Science*, pages 116–127. Springer, 2005.

[31] Mark de Berg. Vertical ray shooting for fat objects. In *SCG '05: Proceedings of the Twenty-First Annual Symposium on Computational geometry*, pages 288–295, 2005.

[32] Mark de Berg, Otfried Cheong, Herman J. Haverkort, Jung Gun Lim, and Laura Toma. I/O-efficient flow modeling on fat terrains. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *Algorithms and Data Structures, 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007, Proceedings*, volume 4619 of *Lecture Notes in Computer Science*, pages 239–250. Springer, 2007.

[33] Mark de Berg, Haggai David, Matthew J. Katz, Mark Overmars, A. Frank van der Stappen, and Jules Vleugels. Guarding scenes against invasive hypercubes. *Computational Geometry: Theory and Applications*, 26:99–117, 2003.

[34] Mark de Berg and Chris Gray. Vertical ray shooting and computing depth orders for fat objects. In *SODA '06: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 494–503, 2006.

[35] Mark de Berg and Chris Gray. Computing the visibility map of fat objects. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *Algorithms and Data Structures, 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007, Proceedings*, volume 4619 of *Lecture Notes in Computer Science*, pages 251–262. Springer, 2007.

[36] Mark de Berg and Chris Gray. Vertical ray shooting and computing depth orders for fat objects. *SIAM Journal on Computing*, 38(1):257–275, 2008.

[37] Mark de Berg, Herman J. Haverkort, Shripad Thite, and Laura Toma. I/O-efficient map overlay and point location in low-density subdivisions. In Takeshi Tokuyama, editor, *Algorithms and Computation, 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007, Proceedings*, volume 4835 of *Lecture Notes in Computer Science*, pages 500–511. Springer, 2007.

[38] Mark de Berg and Marc H. Overmars. Hidden-surface removal for c-oriented polyhedra. *Computational Geometry: Theory and Applications*, 1:247–268, 1992.

[39] Mark de Berg, Mark Overmars, and Otfried Schwarzkopf. Computing and verifying depth orders. *SIAM Journal on Computing*, 23(2):437–446, April 1994.

[40] Mark de Berg and Micha Streppel. Approximate range searching using binary space partitions. In *Proc. 24th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 110–121, 2004.

[41] Mark de Berg, A. Frank van der Stappen, Jules Vleugels, and Matthew J. Katz. Realistic input models for geometric algorithms. *Algorithmica*, 34(1):81–97, 2002.

[42] Mark de Berg, Mark van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry Algorithms and Applications*. Springer-Verlag, Berlin Heidelberg, 3 edition, 2008.

[43] David P. Dobkin and David G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoretical Computer Science*, 27(3):241–253, 1983.

[44] Christian A. Duncan, Michael T. Goodrich, and Stephen G. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octtrees. In *SODA '99: Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 300–309, 1999.

[45] Rex A. Dwyer. On the convex hull of random points in a polytope. *Journal of Applied Probability*, 25(4):688–699, 1988.

[46] Alon Efrat. The complexity of the union of (α, β)-covered objects. *SIAM Journal on Computing*, 34(4):775–787, 2005.

[47] Alon Efrat, Matthew J. Katz, Franck Nielsen, and Micha Sharir. Dynamic data structures for fat objects and their applications. *Computational Geometry: Theory and Applications*, 15:215–227, 2000.

[48] Hossam A. El Gindy and David Avis. A linear algorithm for computing the visibility polygon from a point. *Journal of Algorithms*, 2:186–197, 1981.

[49] Jeff Erickson. New lower bounds for Hopcroft's problem. *Discrete and Computational Geometry*, 16:389–418, 1996.

[50] Jeff Erickson. Local polyhedra and geometric graphs. *Computational Geometry: Theory and Applications*, 31:101–125, 2005.

[51] Esther Ezra and Micha Sharir. Almost tight bound for the union of fat tetrahedra in three dimensions. In *FOCS '07: Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, pages 525–535, Washington, DC, USA, 2007. IEEE Computer Society.

[52] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990.

[53] Michael T. Goodrich, Mikhail J. Atallah, and Mark H. Overmars. An input-size/output-size trade-off in the time-complexity of rectilinear hidden surface removal. In M. S. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming, ICALP'90 (Warwick University, England, July 16-20, 1990)*, volume 443 of *LNCS*, pages 689–702. Springer-Verlag, Berlin-Heidelberg-New York-London-Paris-Tokyo-Hong Kong, 1990.

[54] Ralf Hartmut Güting and Thomas Ottmann. New algorithms for special cases of the hidden line elimination problem. *Computer Vision, Graphics, and Image Processing*, 40(2):188–204, November 1987.

[55] Peter Hachenberger. Exact minkowksi sums of polyhedra and exact and efficient decomposition of polyhedra in convex pieces. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, volume 4698 of *Lecture Notes in Computer Science*, pages 669–680. Springer, 2007.

[56] Paul J. Heffernan and Joseph S. B. Mitchell. Structured visibility profiles with applications to problems in simple polygons (extended abstract). In *SCG '90: Proceedings of the Sixth Annual Symposium on Computational geometry*, pages 53–62, 1990.

[57] Stefan Hertel and Kurt Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th Conf. Foundations of Computation Theory*, pages 207–218. Springer-Verlag, LNCS 158, 1983.

[58] Matthew J. Katz. 3-d vertical ray shooting and 2-d point enclosure, range searching, and arc shooting amidst convex fat objects. *Computational Geometry: Theory and Applications*, 8:299–316, 1997.

[59] Matthew J. Katz. personal communication, 2005.

[60] Matthew J. Katz, Marc Overmars, and Micha Sharir. Efficient hidden surface removal for objects with small union size. *Computational Geometry: Theory and Applications*, 2:223–234, 1992.

[61] J. Mark Keil. Polygon decomposition. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 11, pages 491–518. Elsevier, 2000.

[62] David Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12:28–35, 1983.

[63] David Kirkpatrick. Guarding galleries with no nooks. In *Proceedings of the 12th Canadian Conference on Computational Geometry (CCCG'00)*, pages 43–46, 2000.

[64] David G. Kirkpatrick, Maria M. Klawe, and Robert Endre Tarjan. Polygon triangulation in $O(nloglogn)$ time with simple data structures. *Discrete and Computational Geometry*, 7:329–346, 1992.

[65] Jiří Matoušek, János Pach, Micha Sharir, Shmuel Sifrony, and Emo Welzl. Fat triangles determine linearly many holes. *SIAM Journal on Computing*, 23(1):154–169, February 1994.

[66] Jiří Matoušek. Efficient partition trees. In *SCG '91: Proceedings of the Seventh Annual Symposium on Computational geometry*, pages 1–9, 1991.

[67] Michael McKenna. Worst-case optimal hidden surface removal. *ACM Transactions on Graphics*, 6:19–28, 1987.

[68] Avraham A. Melkman. On-line construction of the convex hull of a simple polyline. *Information Processing Letters*, 25(1):11–12, April 1987.

[69] Joseph S. B. Mitchell, David M. Mount, and Subhash Suri. Query-sensitive ray shooting. *International Journal of Computational Geometry and Applications*, 7(4):317–347, 1997.

[70] Esther Moet. *Computation and complexity of visibility in geometric environments*. PhD thesis, Department of Computer Science, Utrecht University, 2008.

[71] Esther Moet, Marc van Kreveld, and A. Frank van der Stappen. On realistic terrains. In *SCG '06: Proceedings of the Twenty-Second Annual Symposium on Computational geometry*, pages 177–186, New York, NY, USA, 2006. ACM.

[72] Shai Mohaban and Micha Sharir. Ray shooting amidst spheres in three dimensions and related problems. *SIAM Journal on Computing*, 26(3):654–674, 1997.

[73] Joseph O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, New York, NY, 1987.

[74] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1994.

[75] Marc H. Overmars and A. F. van der Stappen. Range searching and point location among fat objects. In *Proc. 2nd European Symposium on Algorithms*, pages 240–253. Springer Verlag, LNCS 885, 1994.

[76] János Pach and Gábor Tardos. On the boundary complexity of the union of fat triangles. *SIAM Journal on Computing*, 31(6):1745–1760, 2002.

[77] Mike Paterson and F. Frances Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete and Computational Geometry*, 5:485–503, 1990.

[78] Marco Pellegrini. Ray shooting on triangles in 3-space. *Algorithmica*, 9:471–494, 1993.

[79] Marco Pellegrini. Ray shooting and lines in space. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 599–614. CRC Press, Boca Raton-New York, 1997.

[80] Franco P. Preparata, Jeffrey Scott Vitter, and Mariette Yvinec. Computation of the axial view of a set of isothetic parallelepipeds. *ACM Transactions on Graphics*, 9(3):278–300, July 1990.

[81] John H. Reif and Sandeep Sen. An efficient output-sensitive hidden-surface removal algorithm and its parallelization. In *SCG '88: Proceedings of the Fourth Annual Symposium on Computational geometry*, pages 193–200, June 1988.

[82] Jim Ruppert and Raimund Seidel. On the difficulty of triangulating three-dimensional nonconvex polyhedra. *Discrete and Computational Geometry*, 7:227–253, 1992.

[83] E. Schönhardt. Über die Zerlegung von Dreieckspolyedern in Tetraeder. *Mathematische Annalen*, 98:309–312, 1928.

[84] Anneke A. Schoone and Jan van Leeuwen. Triangulating a starshaped polygon. Technical Report RUU-CS-80-03, Institute of Information and Computing Sciences, Utrecht University, 1980.

[85] Ottfried Schwarzkopf and Jules Vleugels. Range searching in low-density environments. *Information Processing Letters*, 60:121–127, 1996.

[86] Raimund Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1:51–64, 1991.

[87] Micha Sharir and Pankaj K. Agarwal. *Davenport-Schinzel sequences and their geometric applications*. Cambridge University Press, New York, NY, USA, 1996.

[88] Micha Sharir and Marc H.Overmars. A simple method for output-sensitive hidden surface removal. *ACM Transactions on Graphics*, 11:1–11, 1992.

[89] Micha Sharir and Hayim Shaul. Ray shooting and stone throwing. In *Proc. 11th European Symposium on Algorithms*, pages 470–481. Springer-Verlag, LNCS 2832, 2003.

[90] Micha Sharir and Hayim Shaul. Ray shooting amid balls, farthest point from a line, and range emptiness queries. In *SODA '05: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 525–534, 2005.

[91] Csaba D. Tóth. A note on binary plane partitions. In *SCG '01: Proceedings of the Seventeenth Annual Symposium on Computational geometry*, pages 151–156, New York, NY, USA, 2001. ACM.

[92] Godfried T. Toussaint. A new linear algorithm for triangulating monotone polygons. *Pattern Recognition Letters*, 2:155–158, 1984.

[93] Godfried T. Toussaint and David Avis. On a convex hull algorithm for polygons and its application to triangulation problems. *Pattern Recognition*, 15(1):23–29, 1982.

[94] Godfried T. Toussaint and Hossam El Gindy. A counterexample to an algorithm for computing monotone hulls of simple polygons. *Pattern Recognition Letters*, 1:219–222, 1983.

[95] Pavel Valtr. Guarding galleries where no point sees a small area. *Israel Journal of Mathematics*, 104:1–16, 1998.

[96] A. Frank van der Stappen. *Motion Planning Amidst Fat Obstacles*. PhD thesis, Dept. of Computer Science, Utrecht University, 1994.

[97] A. Frank van der Stappen, Dan Halperin, and Mark. H. Overmars. The complexity of the free space for a robot moving amidst fat obstacles. *Computational Geometry: Theory and Applications*, 3:353–373, 1993.

[98] Marc van Kreveld. On fat partitioning, fat covering and the union size of polygons. *Computational Geometry: Theory and Applications*, 9(4):197–210, 1998.

Algorithms for Fat Objects:
Decompositions and Applications

Computational geometry is the branch of theoretical computer science that deals with algorithms and data structures for geometric objects. The most basic geometric objects include points, lines, polygons, and polyhedra. Computational geometry has applications in many areas of computer science, including computer graphics, robotics, and geographic information systems.

In many computational-geometry problems, the theoretical worst case is achieved by input that is in some way "unrealistic". This causes situations where the theoretical running time is not a good predictor of the running time in practice. In addition, algorithms must also be designed with the worst-case examples in mind, which causes them to be needlessly complicated. In recent years, *realistic input models* have been proposed in an attempt to deal with this problem. The usual form such solutions take is to limit some geometric property of the input to a constant.

We examine a specific realistic input model in this thesis: the model where objects are restricted to be *fat*. Intuitively, objects that are more like a ball are more fat, and objects that are more like a long pole are less fat. We look at fat objects in the context of five different problems—two related to decompositions of input objects and three problems suggested by computer graphics.

Decompositions of geometric objects are important because they are often used as a preliminary step in other algorithms, since many algorithms can only handle geometric objects that are convex and preferably of low complexity. The two main issues in developing decomposition algorithms are to keep the number of pieces produced by the decomposition small and to compute the decomposition quickly. The main question we address is

the following: is it possible to obtain better decompositions for fat objects than for general objects, and/or is it possible to obtain decompositions quickly? These questions are also interesting because most research into fat objects has concerned objects that are convex.

We begin by *triangulating* fat polygons. The problem of triangulating polygons—that is, partitioning them into triangles without adding any vertices—has been solved already, but the only linear-time algorithm is so complicated that it has never been implemented. We propose two algorithms for triangulating fat polygons in linear time that are much simpler. They make use of the observation that a small set of guards placed at points inside a (certain type of) fat polygon is sufficient to see the boundary of such a polygon.

We then look at decompositions of fat polyhedra in three dimensions. We show that polyhedra can be decomposed into a linear number of convex pieces if certain fatness restrictions are met. We also show that if these restrictions are not met, a quadratic number of pieces may be needed. We also show that if we wish the output to be fat and convex, the restrictions must be much tighter.

We then study three computational-geometry problems inspired by computer graphics.

First, we study *ray-shooting* amidst fat objects from two perspectives. This is the problem of preprocessing data into a data structure that can answer which object is first hit by a query ray in a given direction from a given point. We present a new data structure for answering vertical ray-shooting queries—that is, queries where the ray's direction is fixed—as well as a data structure for answering ray-shooting queries for rays with arbitrary direction. Both structures improve the best known results on these problems.

Another problem that is studied in the field of computer graphics is the *depth-order* problem. We study it in the context of computational geometry. This is the problem of finding an ordering of the objects in the scene from "top" to "bottom", where one object is above the other if they share a point in the projection to the xy-plane and the first object has a higher z-value at that point. We give an algorithm for finding the depth order of a group of fat objects and an algorithm for verifying if a depth order of a group of fat objects is correct. The latter algorithm is useful because the former can return an incorrect order if the objects do not have a depth order (this can happen if the above/below relationship has a cycle in it). The first algorithm improves on the results previously known for fat objects; the second is the first algorithm for verifying depth orders of fat objects.

The final problem that we study is the *hidden-surface removal* problem. In this problem, we wish to find and report the visible portions of a scene from a given viewpoint—this is called the *visibility map*. The main difficulty in this problem is to find an algorithm whose running time depends in part on the complexity of the output. For example, if all but one of the objects in the input scene are hidden behind one large object, then our algorithm should have a faster running time than if all of the objects are visible and have borders that overlap. We give such an algorithm that improves on the running time of previous algorithms for fat objects. Furthermore, our algorithm is able to handle curved objects and situations where the objects do not have a depth order—two features missing from most other algorithms that perform hidden surface removal.

110

Curriculum Vitae

Chris Gray was born on the November 11, 1980 in Flint, Michigan, USA. He graduated from Bishop Carroll High School in Calgary, Alberta, Canada in 1998. He received his Bachelor of Science in Mathematics and Computer Science from McGill University in Montréal in 2002. He completed his Master of Science degree in Computer Science at the University of British Columbia in Vancouver in 2004. Since September 2004, he has been a Ph.D. student within the Computer Science department of the Technische Universiteit Eindhoven (TU/e).

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition*

and construction. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*.

Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support- *. Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to*

Developing Future-Proof System Architectures. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering,

Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty

of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance*

Analysis of Data-Independent Stream Processing Systems. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathemat-

ics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19