Efficient Hidden-Surface Removal in Theory and in Practice

by

T. M. Murali

B. Tech., Indian Institute of Technology, Madras, 1991

Sc. M., Brown University, 1993

Thesis

Submitted in partial fulfillment of the requirements for the

Degree of Doctor of Philosophy in the Department of Computer Science

at Brown University.

May 1999

This dissertation by T. M. Murali
is accepted in its present form
by the Department of Computer Science
as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ______________ ______________________________

Jeffrey Scott Vitter

Recommended to the Graduate Council

Date ______________ ______________________________

Pankaj K. Agarwal

Date ______________ ______________________________

John F. Hughes

Approved by the Graduate Council

Date ______________ ______________________________

Vita

I was born on August 4, 1969 in Madras, India. I studied at St. Xavier's School in Bokaro Steel City, Bihar till 1985 and then moved to Vidya Mandir in Madras for the last two years of my school education. In 1987, I joined the Indian Institute of Technology in Madras. Four years later, I had the nickname T_{MAX} and a B. Tech degree in Computer Science and Engineering. I joined the Ph.D. programme in the Department of Computer Science at Brown University in Providence, RI, in 1991, and received an Sc. M. degree in 1993. Since July 1993, I have been a visiting scholar at the Department of Computer Science at Duke University in Durham, NC.

Acknowledgments

I have been very fortunate to have Jeff Vitter as my advisor. I am very grateful to him for inviting me in 1993 to Duke to do my Ph. D. research with him. Working with him has made my graduate life a wonderful experience. I have benefited greatly from his advice and support. He taught me the value of research that is both theoretically appealing and practically useful. I hope I always strive to achieve this balance between theory and practice.

Moving to Duke to work with Jeff also brought me into contact with Pankaj Agarwal. Pankaj has been a co-author on most of my publications. I have learnt a lot about computational geometry from him. He has readily shared his keen insight into geometric issues with me. He has always been a fount of ideas and open problems. His enthusiasm and dynamism have always motivated me. I have thoroughly enjoyed the many courses he taught at Duke. His influence on my research will be lasting.

I thank John "Spike" Hughes for being on my thesis committee. The probing questions he has asked whenever I talked to him have egged me to improve the results in this thesis and to evaluate their usefulness in the broader context surrounding them.

This dissertation owes a lot to my wife, Padma Rajagopalan. I thank her for cheering me when bugs were plentiful, for listening patiently while I explained my half-baked ideas over the phone, for her unflagging enthusiasm and support, and above all, for providing the spark of inspiration whenever I needed it the most.

I am indebted to my parents Raji and T. S. Maheswaran for their constant encouragement and love. I have reached this stage only because of their wholehearted support for all my intellectual endeavours and their belief in my abilities. I thank my sister Nikila for her love, and my grandparents Kamala and K. R. Balasubramaniam, and Gnanambal and S. Krishnamurthy for all they have done for me during the years

I studied at Madras and in the years since.

During my graduate life, many friends have made life and work a pleasure: Lars Arge, Rakesh Barve, Jeff Erickson, Eddie Grove, Gopal Kidao, P. Krishnan, Kasturi Varadarajan, and Darren Vengroff at Duke; and Swarup Acharya, Anand Bodapati, Andrea Pietracaprina, Viswanath Ramachandran, R. Ravi, Ranjani Saigal, Bharathi Subramanian, and Jayashree Subrahmonia at Brown. I especially value my interactions with Kasturi and the innumerable hours we have spent discussing research and geometry over coffee. I would also like to thank him for his comments on Chapter 3 of this thesis. Owen Astrachan and Robert Duvall patiently answered all the programming-related questions I asked them.

I learnt a lot from my summer internships at SGI and Bell Laboratories. I thank Michael Jones for making me a part of the Performer team at SGI in 1995, and Tom Funkhouser for his guidance and collaboration at Bell Labs in 1996.

Doing my research at Duke while I was a student at Brown posed a series of administrative and legal questions. I would like to thank the Departments of Computer Science and the Graduate Schools at both universities for making this dual existence easy for me. In particular, I thank Mary Andrade, Dawn Nicholaus, and Michael Diffily at Brown, and Cathie Caimano, Beatrice Chestnutt, Angela Davis-Kincy, Anna Drozdowski, Tina Gaither, and Denita Thomas at Duke. I am also extremely grateful to John Eng-Wong and Marilyn von Kriegenbergh at the Foreign Students Office at Brown for navigating me effortlessly through the maze of INS regulations; I have found their help and advice invaluable.

I would like to thank Andy van Dam and the Computer Graphics group at Brown and Henry Fuch and the Computer Graphics group at University of North Carolina for letting me use their video network to broadcast my thesis proposal. (I gave the talk at Brown, where Spike was also present, while Jeff and Pankaj watched the talk from UNC.) In particular, I am grateful to David Harrison and Mark Oribello for ensuring that the video transmission went without a hitch.

Finally, I would like to thank K. Q. Brown for his beautiful Ph. D. thesis “Geometric Transforms for Fast Geometric Algorithms.” In my third year at the Indian Institute of Technology at Madras, I read his thesis and wrote a term paper on it. I have been in love with computational geometry since then!

Credits

The research described in this dissertation has been conducted with many colleagues. I thank them for allowing me to include these results here. When I visited Bell Laboratories in the summer of 1996, Tom Funkhouser and I developed the algorithm for geometric data repair presented in Chapter 2. All the other algorithms described were designed while I was a visiting scholar at Duke University: the object complexity hidden-surface removal algorithm in Chapter 3 is joint work with Pankaj Agarwal, Pavan Desikan, and Jeff Vitter; Pankaj Agarwal, Eddie Grove, Jeff Vitter, and I developed the technique for constructing a BSP for fat rectangles described in Chapter 4); I implemented the algorithm for constructing a BSP for orthogonal rectangles with Pankaj Agarwal and Jeff Vitter (see Chapter 5); Pankaj Agarwal, Leo Guibas, Jeff Vitter, and I collaborated on the algorithms presented in Chapter 6 for constructing a BSP for triangles in $\mathbb{R}^3$.

I would like to thank Seth Teller for providing me with the Soda Hall data set created at the Department of Computer Science, University of California at Berkeley, and the Walkthrough Project, Department of Computer Science, University of North Carolina at Chapel Hill for providing me with the data sets for Sitterson Hall, the Orange United Methodist Church Fellowship Hall, and the Sitterson Hall Lobby.

I gratefully acknowledge the financial support provided to me by National Science Foundation research grants CCR–9007851 and CCR–9522047, by Army Research Office grant DAAL03–91–G–0035, and by Army Research Office MURI grant DAAH04–96–1–0013.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

How to render a set of opaque or partially transparent objects in $\mathbb{R}^3$ quickly and in a visually realistic way is a fundamental problem in computer graphics [44, 100]. A central component of rendering is *hidden-surface removal:* given a set of objects (some of which may be moving continuously), a continuously-moving viewpoint, and an image plane, compute the scene visible from the viewpoint as projected onto the image plane. Informally, for each point p in the image plane, we would like to compute the first object that is intersected by a ray originating at the viewpoint and passing through p.

Typical applications of this problem are architectural walkthroughs, flight simulation, terrain fly-overs, video games, and CAD design. In such applications, the visible scene must be computed and rendered as many as 60 times a second in order to achieve realism. The fundamental difficulty in the problem arises from the fact that such "frame rates" need to be achieved for input data sets that may contain hundreds of millions or even billions of objects. Further, the motion of the viewpoint and the objects is often user-determined and unpredictable. Given these challenges, our aim in this dissertation is to develop theoretically and practically efficient algorithms for hidden-surface removal.

In this chapter, we give a brief overview of the hidden-surface removal techniques developed in computer graphics and in computational geometry, and compare and contrast these two classes of algorithms. Next, we discuss how these known algorithms motivate the framework in which we propose to solve the hidden-surface removal problem. Finally, we highlight the contributions of this dissertation.

1.1 Perspective on Previous Work

The hidden-surface removal problem has been studied extensively in both the computer graphics [44] and the computational geometry communities [39]. We now briefly survey the vast literature on this problem, in order to put the developments of the last 30 years into perspective. In this section, our goal is to highlight the similarities and the differences between the algorithms developed in the computer graphics and the computational geometry literature.

Back-face culling and view-frustum culling are classical approaches to hidden-surface removal. In back-face culling, any triangle whose normal is directed away from the viewpoint is deemed to be invisible and is not rendered. View-frustum culling is based on the fact that the visibility is usually restricted to a frustum of directions around the viewpoint. As the name indicates, view-frustum culling renders only those triangles that intersect this frustum of directions. Excellent surveys of early techniques for hidden-surface removal are provided by Sutherland et al. [100] and Foley et al. [44, Chapter 15]. These approaches are divided into two prototypical classes: *object-space* techniques work at the precision with which each object is defined and compute the visibility of each object, while *image-space* algorithms work at the precision of the display device and determine visibility information at each pixel of the display.

In the last 10 years, many sophisticated algorithms for hidden-surface removal of large data sets have been developed in the computer graphics community. These algorithms are based on the principle of *occlusion culling*: do not render objects that are not visible from the viewpoint. Techniques developed by Airey [8], Teller [101], and Luebke and Georges [65] are effective for architectural environments. Coorg and Teller [34] and Hudson et al. [58] develop object-space techniques for occlusion culling in general polygonal models. Greene et al. [53] and Zhang et al. [108] have developed algorithms that combine object-space and image-space components. We survey these approaches in more detail in Chapter 3. There has been a lot of research on techniques orthogonal to occlusion culling that render objects that are far away from the viewpoint in simplified form or by replacing them with textures or images. These techniques do not solve the hidden-surface removal problem directly, but can be used to supplement algorithms for hidden-surface removal.

Many algorithms have been proposed in the computational geometry literature to solve the hidden-surface removal problem. These algorithms typically compute the *visibility map*, which is a partition of the image plane into maximally-connected regions with the property that in each region, at most one triangle is visible. Worst-case optimal algorithms are presented by Dévai [38] and McKenna [67]. The running time of more recent algorithms depends on n, the size of the input as well as k, the *scene complexity* or the number of edges in the visibility map. Many of these algorithms are designed to work for specific classes of inputs: rectangles and orthogonal objects [17, 50, 86], polyhedral terrains [59, 85, 87], and objects whose unions have small size [59]. de Berg et al. [36] present a general algorithm for hidden-surface removal that constructs the visibility map for n triangles in $\mathbb{R}^3$ in $O(n^{1+\varepsilon}\sqrt{k})$ time, for any fixed $\varepsilon > 0$. Agarwal and Matoušek [5] improve this time to $O(n^{2/3+\varepsilon}k^{2/3} + n^{1+\varepsilon})$, which is the best-known so far.

Examining the various algorithms mentioned above, we observe some fundamental differences between those used in computer graphics and those developed in computational geometry: (i) performance in theory vs. performance in practice, (ii) the granularity at which visibility is resolved and (iii) assumptions made on the motion of the viewpoint and input objects. Let us look at these three issues in some more detail.

Theory vs. Practice Computer graphics algorithms perform well in practice for many classes of inputs since they are typically developed to optimise performance for the "average" input. However, a fundamental drawback of most of these algorithms is that their theoretical performance is not analysed. Teller [101] analyses only some components of his algorithms while Chamberlain et al. [26] prove bounds on the performance of their algorithms under rather strong assumptions on the input, assumptions which seem to be valid only for a small class of input data sets.

On the other hand, algorithms used in computational geometry are proved to be efficient even in the worst case. Unfortunately, they are often very complicated (since they are designed to overcome all pathological inputs) and are difficult to implement as a result. Even when they are implemented, large constant factors in the running time, which are hidden in the theoretical analysis by the big-oh notation, tend to make these algorithms impractical.

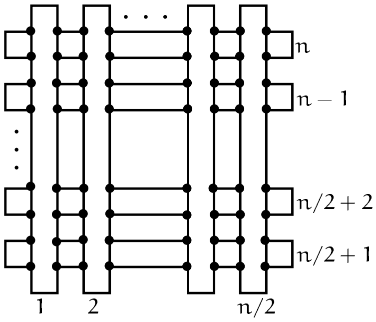


Figure 1.1: A set of n rectangles whose visibility map has $\Omega(n^2)$ size.

Visibility Granularity As we mentioned earlier, hidden-surface removal algorithms in computational geometry compute the visibility map. Informally, the visibility map encodes exactly which portions of each input object is visible. A major drawback of this approach is that the size of the visibility map can be $\Omega(n^2)$ for n objects, e.g., when the viewpoint is at $z = +\infty$ and the scene contains $n/2$ thin rectangles parallel to the x-axis lying directly above $n/2$ thin rectangles parallel to the y-axis (see Figure 1.1). As a result, any algorithm that solves the hidden-surface removal problem by computing the visibility map takes quadratic time in the worst case to compute and render the visible scene.

In marked contrast, most computer graphics algorithms only compute which input objects are visible (an object is deemed to be visible if at least one point on the object is visible). These algorithms then feed the visible objects to graphics hardware (such as the z-buffer, which we examine in the next section) to compute the final scene.

Assumptions on Motion Most computational geometry algorithms for hidden-surface removal have been developed for the static setting, when the viewpoint and input objects are not moving. It appears that the only way we can extend any of them to the case when the viewpoint or objects in the input are moving is to periodically execute the static algorithm from scratch. Clearly, this approach is wasteful since it does not exploit the continuity of the motion of the viewpoint and input objects. In the worst-case, we might recompute the entire visibility map only to discover that it has not changed "combinatorially" (we will define this notion precisely in Chapter 3) since the previous invocation of the algorithm. A few algorithms for dealing with moving viewpoints have been developed [18, 64, 71]. However, all these algorithms assume that the trajectory of the viewpoint is either restricted (along a straight line, for example) and completely known *a priori*. Both assumptions are invalid in the vast majority of practical situations, when viewpoint and object motions are user-defined and not fully predictable. Pellegrini presents a related algorithm for constructing a data structure that encodes all possible visibility maps compactly so that the visibility map for a query viewpoint can be computed efficiently [83]. However, this algorithm improves on the fastest static algorithm (by Agarwal and Matoušek [5]) only when the visibility map has small size.

Some computer graphics algorithms take advantage of *temporal coherence*, the phenomenon in which the set of visible triangles changes gradually as the viewpoint and input objects move continuously. The basic idea these algorithms use is to sample time discretely and to update the set of visible objects at the end of each time step or *frame*. For example, some algorithms that store the input objects in a data structure approximate the continuous motion of each object by deleting and reinserting it (or a suitably chosen bounding volume) in a new position at the beginning of each frame [31, 76, 97, 103]. Such approaches suffer from the fundamental problem that it is very difficult to know how to choose the correct interval size: if the interval is too small, then the data structure and the set of visible objects does not in fact change, and the deletion/re-insertion is just wasted computation; if it is too big, then important intermediate changes to the data structure can be missed. Other algorithms for hidden-surface removal [33, 53, 108] exploit temporal coherence to speed up only some steps of the algorithms; they execute other, possibly time-consuming steps of the algorithm from scratch at the beginning of each frame.

To summarise, there are three basic differences between hidden-surface removal algorithms in computer graphics and computational geometry:

1. Computer graphics algorithms work efficiently in practice but their theoretical performance is not well understood. Techniques described in the computational geometry literature are analysed theoretically in the worst-case but they have not been used in practice since they are complicated.

2. Computer graphics algorithms compute just which set of objects are visible and rely on graphics hardware such as the z-buffer to render the final scene. Computational geometry techniques compute exactly which portion of each object is visible, and have worst-case running times that are quadratic in the number of input objects.

3. While computer graphics techniques try to exploit the continuity of viewpoint and object motion, they do so only indirectly since they approximate continuous motion by a discrete sampling of time. Algorithms developed in computational geometry are developed either for static inputs or make very strong assumptions about the object motions.

1.2 Goals of this Thesis

In this thesis, our goal is to bridge the gap between hidden-surface removal in theory and in practice. One of our primary goals is to design algorithms that are theoretically efficient. At the same time, we would like our algorithms to be simple, implementable, and efficient in practice too. By developing algorithms in realistic and appropriate models of computation, we hope to ensure that our algorithms have all these desirable properties. In the rest of this section, we describe the three major themes we use to develop theoretically- and practically-efficient algorithms for hidden-surface removal. These themes are motivated by the different advantages and drawbacks of known algorithms for hidden-surface removal that we noted earlier.

Geometric complexity Practically all algorithms in computational geometry are analysed only in terms of the *combinatorial complexity* (or size) of the input. As we have noted already, such a model requires that algorithms be efficient for *all* possible inputs, a requirement that often results in the development of algorithms that are complicated and unimplementable, for all practical purposes. In contrast, we propose to analyse algorithms in the framework of *geometric complexity* (in addition to combinatorial complexity); we use geometric complexity as a catch-all that captures typical characteristics of input models such as aspect ratio [104], clutter [35], low density [90], and depth complexity [101]. For example, we say that a set of rectangles has low geometric complexity if the aspect ratio of most rectangles in the set is bounded by a small constant. Similar ideas have been studied by de Berg et al. [37]. The use of geometric complexity is motivated by the observation that algorithms in computer graphics are fast in practice since they implicitly (or even explicitly) tune their performance to certain geometric features of the input, which allows the development of simple solutions to the hidden-surface removal problem. We aim to develop provably-efficient algorithms that take advantage of the low geometric complexity of typical input models. As we demonstrate later in our thesis, since our algorithms exploit such special (yet common) inputs, they are simple, easy to implement, and efficient in practice too. It is important to stress that our algorithms will perform correctly for all inputs (irrespective of the geometric complexity of the input); it is only the *analysis* and *performance* of the algorithms that will depend on the geometric complexity of the input.

Object complexity As we noted earlier, computational geometry algorithms for solving the hidden-surface removal problem compute the visibility map. In practice, computing the entire visibility map is not necessary. Since the image plane is an abstraction of a finite-resolution screen, it is enough to compute for each pixel on the image plane, which object is visible at that pixel. Suppose the input is a set of polygons. The popular z-buffer algorithm [25, 44] is a simple technique that computes this information by sequentially processing the input polygons. For each pixel, the algorithm stores the colour and the distance from the viewpoint of the closest already-processed polygon that covers that pixel. For a new polygon, the algorithm examines each pixel covered by the polygon, and updates the information for that pixel if the distance of the new polygon from the viewpoint is less than the distance stored with the pixel. The running time of the z-buffer algorithm is $O(n + a)$, where a is the total number of pixels contained in the projections of the polygons in the input onto the image plane [49]. In the complex and massive data sets we are interested in performing hidden-surface removal on, most object projections occupy a very small area. Hence, it is reasonable to approximate the running time of the z-buffer algorithm by the $O(n)$ term.

While the z-buffer can be implemented very efficiently in hardware (for example, SGI's InfiniteReality system [68] can render 7 million triangles a second), current data sets containing millions of objects are simply too large to be processed at the rate of 60 times a second even by state-of-the-art hardware z-buffers. Furthermore, recall that computational geometry algorithms for hidden-surface removal can spend $\Omega(n^2)$ time to compute the visibility map; in such cases, it is likely that the z-buffer can process the original n polygons much faster than it can process the $\Omega(n^2)$ faces of the visibility map. Thus, the $O(n)$ running time of the z-buffer algorithm indicates that the visibility map is an inappropriate model in which to develop hidden-surface removal algorithms. Note that the number of visible triangles is a lower bound on the running time of the z-buffer algorithm, since the z-buffer must process all visible triangles in order to render the scene correctly.

Motivated by these observations, we propose the *object complexity* model, in which we measure the size of a scene in terms of the size of the input and number of visible objects. Object complexity captures the cost of rendering objects using the z-buffer more accurately than scene complexity does. In this model, we develop algorithms that compute the set of objects visible from the viewpoint and then use hardware z-buffers to render these objects.

Kinetic data structures As we have seen, known hidden-surface removal algorithms exploit continuous motion only indirectly since they approximate it by sampling time discretely. We propose to use the notion of kinetic data structures pioneered by Basch et al. [13]. In this paradigm, an algorithm maintains a discrete attribute of a set of moving objects (for example, the set of objects that are visible from a viewpoint) by animating a proof of correctness through time. The proof consists of a set of elementary geometric conditions called *certificates*, which are based on the tests performed by the algorithm. Certificates that will fail in the future due to the motion of the objects are stored in an event queue, ordered by time of failure. When a certificate fails, the algorithm performs the operations necessary to update the proof and the attribute being maintained. Thus, a kinetic data structure explicitly exploits the continuous motion of the objects so that its sampling of time is not fixed but is determined by the instants at which the certificates fail.

Our Thesis

To summarise the above discussion, we aim to develop hidden-surface removal algorithms

1. in the geometric complexity model, so as to exploit the geometric properties and structure of the input objects and to remove the traditional focus on worst-case analysis of algorithms,

2. that compute the set of visible objects and render them using the z-buffer, so that the running time of the algorithms depends on the object complexity of the input, and

3. that use the kinetic data structure paradigm to exploit the continuity of viewpoint and object motion comprehensively.

It is our thesis that developing algorithms for hidden-surface removal under these three criteria will enable us to perform hidden-surface removal efficiently both in theory and in practice.

1.3 Contributions of this Thesis

In this section, we discuss the significance of the results developed in this thesis and provide a road-map of how the rest of the thesis is organised. Throughout the thesis, we assume that our input consists of polyhedral and polygonal objects. In fact, we assume without loss of generality that our input is a set of triangles in $\mathbb{R}^3$. We first describe an algorithm for automatically removing topological and geometric errors from the input. Next, we present our algorithm for hidden-surface removal. The efficiency of both techniques depends on an underlying spatial partition called the binary space partition (BSP). Finally, we discuss our algorithms for constructing BSPs efficiently.

Most commonly available models contain geometric and topological flaws, such as missing triangles and cracks. Intuitively, "clean" models have continuous boundaries with consistently-oriented triangles; such models enable the unambiguous classification of a point as lying inside the model, outside the model, or on the surface of the model. Many algorithms for hidden-surface removal do not perform well for models that contain such errors. For example, an algorithm might expend a lot of time in trying to resolve visibility through small cracks in the model. Another example is back-face culling. Recall that back-face culling declares any triangle that is oriented away from the viewpoint as being invisible. To work correctly, back-face culling requires that all triangles in the model be consistently oriented (in a sense we will define precisely in Chapter 2).

In Chapter 2, we describe a technique we have developed for geometric data repair that automatically corrects such errors and constructs a model that is guaranteed to be consistent. Our algorithm partitions $\mathbb{R}^3$ into regions and uses the novel idea of using region adjacencies to determine "how solid" a region is. We then use the solid regions to construct an error-free representation of the input data. We have implemented and tested our algorithm on various data sets. Our experiments demonstrate that unlike previously described approaches, our method gives excellent results on many real models in $\mathbb{R}^3$ containing intersecting, overlapping, and unconnected triangles and is effective for a large class of input models.

In Chapter 3, we present a new object-complexity algorithm for hidden-surface removal of massive models. We compute a hierarchical spatial decomposition, and classify cells of this decomposition as visible or invisible by intersecting the cells with the union of the shadows cast by a few carefully-chosen occluders. Our technique has several new and attractive features: we use the binary space partition (BSP) to unify occluder selection, visibility maintenance, and mechanisms for frame-rate control; we maintain the union of the shadows as a set of rays in $\mathbb{R}^3$; we can compute the set of visible cells exactly, and can also detect when a cell is occluded by multiple, disconnected triangles; we explicitly exploit the continuity of the motion of the viewpoint and the objects in the input by using kinetic data structures [13]: based on the current motion of the viewpoint and the triangles, we predict the instants when the union of the shadows and the set of visible cells change, and efficiently update the union of shadows and the set of visible cells at each such instant.

Both the model repair and hidden-surface removal algorithms use the BSP as an underlying spatial data structure. Informally, a BSP $\mathcal{B}$ for a set of triangles in $\mathbb{R}^3$ is a binary tree, where each node v is associated with a convex polytope $\mathcal{R}_v$. The regions associated with the children of v are obtained by splitting $\mathcal{R}_v$ with a plane. The polytopes associated with the leaves of the tree form a convex decomposition of $\mathbb{R}^3$, and the interior of the polytope associated with a leaf does not intersect any triangle. The faces of the decomposition induced by the leaves intersect the triangles and divide them into sub-polygons; these sub-polygons are stored at appropriate nodes of the BSP. The size of the BSP is the sum of the number of nodes in it and the total number of faces of all dimensions stored at its nodes.

The BSP was introduced by Fuchs et al. [47] (based on preliminary work by Schumacker et al. [89]) to implement the "painter's algorithm" for hidden-surface removal, which draws the triangles to be displayed on the screen in a back-to-front order (in which no triangle is occluded by any triangle earlier in the order). In general, it is not possible to find a back-to-front order from a given viewpoint for an arbitrary set of triangles. By fragmenting the triangles, the BSP ensures that a back-to-front order from *any* viewpoint can be determined for the fragments. BSPs have subsequently proven to be versatile, with applications in many problems apart from hidden-surface removal [8, 101]—global illumination [23], shadow generation [29, 30], solid modelling [75, 78, 102], ray tracing [74], robotics [12], and approximation algorithms for network design [66] and surface simplification [7].

The efficiency of our model repair and hidden-surface removal algorithms algorithms as well as the applications mentioned above inherently depends on the size and height of the BSP. In Chapters 4–6, we describe our algorithms for constructing BSPs of small size. We show that BSPs of near-linear size can be constructed for orthogonal rectangles with low geometric complexity, where geometric complexity is measured in terms of the aspect ratio of the rectangles (see Chapter 4 for details). Our implementation demonstrates that the algorithm indeed constructs BSPs of linear size in practice on "real" models. We present these implementation results in Chapter 5, where we demonstrate that our algorithm performs better in practice than most algorithms presented in the literature. We also present two algorithms for constructing BSPs for a set of triangles in $\mathbb{R}^3$. One of these algorithms is the first-known algorithm whose running time is close-to-optimal in the worst case. The other algorithm constructs BSPs of near-linear size and polylogarithmic depth for triangles that form a “near”-terrain (we defer a definition of this aspect of geometric complexity to Chapter 6). Finally, we present the first-known provably-efficient algorithm for maintaining the BSP for a set of moving segments in the plane.

In the remaining chapters, we describe each of these techniques in full detail. Our model repair and hidden-surface removal algorithms in combination with our efficient algorithms for constructing BSPs provide a powerful set of techniques for solving the hidden-surface removal problem. In Chapter 7, we examine these issues once again and propose open problems motivated by our results. We also discuss our framework of geometric complexity, kinetic data structures, and object complexity. We point out that the framework is very general and suggest other problem domains in which it can be put to good use.

1.4 Geometric Preliminaries

Before we proceed to the rest of the thesis, we formaly define the BSP. We assume that our input is a set S of n triangles. A *binary space partition* $\mathcal{B}$ for a set S of triangles with pairwise-disjoint interiors in $\mathbb{R}^3$ is a tree defined as follows: Each node v in $\mathcal{B}$ is associated with a polytope $\mathcal{R}_v$ and the set of triangles $S_v = \{s \cap \mathcal{R}_v \mid s \in S\}$ that intersect $\mathcal{R}_v$. The polytope associated with the root is $\mathbb{R}^3$ itself. If S_v is empty, then node v is a leaf of $\mathcal{B}$. Otherwise, we partition $\mathcal{R}_v$ into two convex polytopes by a *cutting plane* H_v. We refer to the polygon $H_v \cap \mathcal{R}_v$ as the *cut* made at v. At v, we store the equation of H_v and the set $\{s \mid s \subseteq H_v, s \in S_v\}$, the subset of triangles in S_v that lie in H_v. If H_v^+ denotes the positive halfspace and H_v^- the negative halfspace bounded by H_v, the polytopes associated with the left and right children of v are $\mathcal{R}_v \cap H_v^-$ and $\mathcal{R}_v \cap H_v^+$, respectively. The left subtree of v is a BSP for the set of triangles $S_v^- = \{s \cap H_v^- \mid s \in S_v\}$ and the right subtree of v is a BSP for the set of triangles $S_v^+ = \{s \cap H_v^+ \mid s \in S_v\}$. The size of $\mathcal{B}$ is the sum of the number of nodes in $\mathcal{B}$ and the total number of triangles stored at all the nodes in $\mathcal{B}$.

Chapter 2

Input Model Repair

In this chapter, we present an algorithm for the automatic correction of polygonal models that contain topological and geometric error. We start by giving some definitions to help us formalise the problem. We say that a set S of triangles in $\mathbb{R}^3$ is *consistent* if the union of the triangles is a closed 2-manifold [72] in which each triangle is oriented with its normal pointing away from the interior of the volume enclosed by the manifold. We say that a consistent set S of triangles is a *correct representation* of a polyhedral solid object in $\mathbb{R}^3$ if the manifold formed by the triangles in S is identical to the boundary of the solid object.

Correct representations of three-dimensional objects are useful and even required in hidden-surface removal. As we pointed out in Chapter 1.3, the simple and popular technique of back-face culling performs incorrectly when the input triangles are inconsistently oriented. In the case of occlusion-culling algorithms, cracks in the input model or triangles missing from S might mislead such algorithms into declaring actually-invisible portions of the input as visible, thus generating incorrect views of the scene. Further, the performance of current rendering graphics hardware is optimised when the triangles to be drawn are presented to the hardware in the form of triangle strips [43, 44]; consistent representations aid in partitioning the input into triangle strips, since the triangles in such a representation form a continuous surface with no cracks, intersections, or missing triangles.

Consistent and correct geometric models are also useful in interactive collision-detection, where some algorithms first process "free" space, i.e., the complement of the union of all the obstacles in the environment [55]. Such algorithms require a correct representation of the boundary of the obstacles so that they can effectively construct the free space. Similarly, algorithms for lighting simulation process meshes constructed on the boundaries of the objects being lit or analysed [15]. If the boundaries have cracks, the mesh is malformed, causing errors and artifacts like spurious shadows in the result. "Bad" meshes can also produce errors in finite element analysis. Further, basic CAD/CAM operations like computing the mass or volume of an object, solid modelling techniques such as Constructive Solid Geometry that perform set operations on solid objects [75, 78, 102], and rapid prototyping [94], which is used to manufacture objects from CAD designs, need models with continuous and closed boundaries, with no cracks or improper intersections. Finally, systems that design and optimise wireless communication systems for a closed environment like a building require descriptions of the boundaries of the obstacles in the building [46, 60].

Unfortunately, most commonly available models of solid objects, whether created by hand or by using automatic tools, contain geometric and topological flaws. Typical errors are:

- wrongly-oriented triangles,
- intersecting or overlapping triangles,
- missing triangles,
- cracks (edges that should be connected have distinct endpoints, faces that should be adjacent have separated edges), and
- T-junctions (the endpoint of one edge lies in the interior of another edge).

For example, in Figure 2.1, there is a crack between segments a and b; segments c and d intersect; and the right endpoint of segment e lies in the interior of d, forming a T-junction. Such errors may be caused by operational mistakes made by the person creating the model, may creep in when converting from one file format to another, or may occur because a particular modeller does not support some features (e.g., a snap grid).

Motivated by the above applications and model imperfections, we consider the following *solid reconstruction* problem:

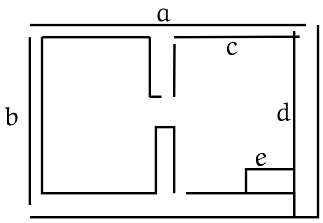


Figure 2.1: A non-manifold model.

> *From an arbitrary set S of triangles in $\mathbb{R}^3$, reconstruct a correct representation of the solid objects modelled by the triangles in S.*

We describe an automatic technique to solve the solid reconstruction problem that works well on many realistic models and is guaranteed to output a consistent set of triangles. The remainder of this chapter is organised as follows: In the next section, we discuss previous algorithms to solve the above problem. In Section 2.2, we describe our solid reconstruction algorithm in detail. Section 2.3 contains experimental results and a brief discussion of the advantages and limitations of our approach. In Section 2.4, we conclude by discussing extensions to our algorithm and posing some open problems.

2.1 Previous Work

It has been noted in the literature that there are currently no robust techniques to solve the solid reconstruction problem [51, 70]. Previous approaches can be divided into two categories: boundary-based approaches and solid-based approaches.

Boundary-based techniques determine how the input triangles mesh together to form the boundaries of the objects modelled by them. Typically, these algorithms merge vertices and edges that are "close" or zip together the boundaries of two faces by merging pairs of "nearby" vertices, where "close" and "nearby" are defined in terms of a pre-specified tolerance. Some boundary-based methods assume that either all the input triangles are consistently oriented or that the orientation of a triangle can be determined from the order of the vertices on its boundary [15, 21]. Such an assumption is often invalid since many data sets contain inconsistently oriented triangles. Other algorithms require (a lot of) user intervention [41, 61], are inherently two-dimensional [60, 63] or are limited to removing parts of zero-volume (like internal walls) from CAD models [20]. Bøhn and Wozny [21] fill cracks or holes by adding triangles; their method can potentially add a lot of triangles to the model. However, the most common deficiency of many of the previous techniques is that they use model-relative tolerances to "fill over" cracks and generate connectivity information about the model [15, 22, 94]. Determining the right tolerance for a given model is a difficult task, probably requiring input from the user. Moreover, such approaches do not work well when the size of some error in the input is larger than the smallest feature in the model. In this case, no suitable tolerance can be chosen that both fills the cracks and preserves small features.

Solid-based algorithms partition $\mathbb{R}^3$ into regions and determine which regions are solid. Thibault and Naylor [102] classify a region as solid when there is at least one input triangle lying on the region's boundary whose normal is directed away from the interior of the region, while Teller [101] declares a region to be solid only if a majority of the triangles lying on the region's boundary have such normals. Both techniques assume that the orientations of the input triangles are correct. As we have pointed out earlier, this assumption is unwarranted for many data sets. Note that both algorithms were developed as a means to represent polyhedra; the authors did not set out to explicitly solve the solid reconstruction problem.

In the computational geometry and solid modelling communities, there has been a lot of work on the related problem of robust geometric computing [45, 56, 57, 91, 95, 99, 105, 107]. These techniques are not applicable to our problem since they attempt to avoid errors caused by *numerical* imprecision and cannot clean-up already incorrect data.

2.2 Our Approach

We have adopted a novel solid-based approach that uses region adjacency relationships to compute which regions are solid and constructs a consistent set of triangles from the solid regions. In contrast to previous boundary-based approaches that attempt to stitch and orient boundary triangles directly, we first focus on classifying *spatial* regions as solid or not and then derive a boundary representation from the solid regions. Also, in contrast to previous solid-based approaches that determine whether regions are solid or not based only on local input triangle orientations, we execute a *global* algorithm that focuses on the opacities of boundaries between regions. As a result, unlike previous approaches, our algorithm is: 1) effective for models containing intersecting, overlapping, and unconnected triangles, 2) independent of input triangle orientations, and 3) guaranteed to output a consistent set of triangles.

Our algorithm proceeds in three phases (as show in Figure 2.2): (a) spatial subdivision, (b) determination of solid regions, and (c) model output.

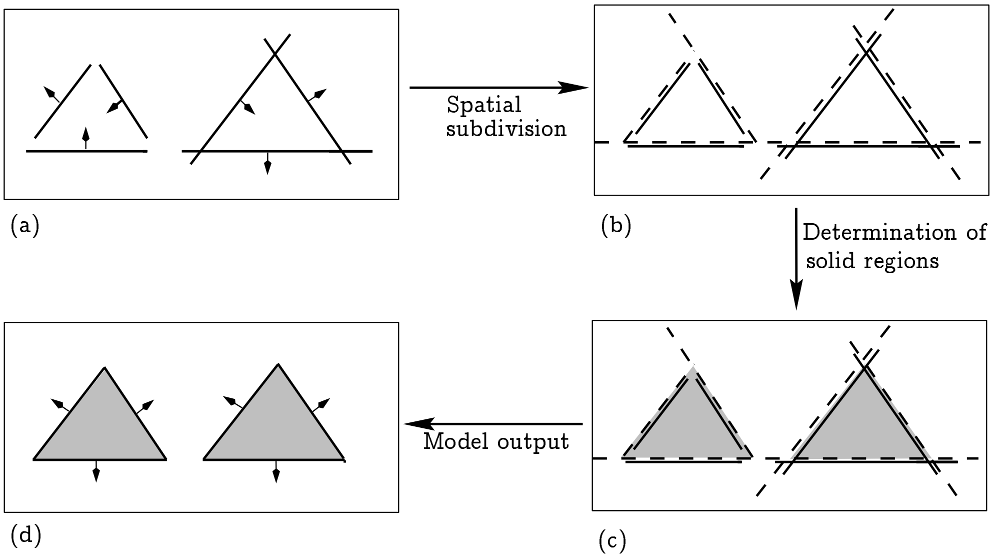


Figure 2.2: (a) Input model with incorrect orientations (b) Subdivision of space (to aid clarity, edges of the subdivision are drawn slightly shifted from the triangle edges) (c) Solid regions (shaded) (d) Output model with correct triangle orientations.

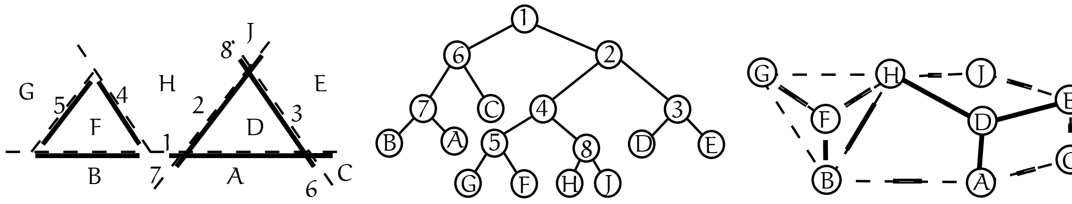
2.2.1 Spatial subdivision

During the spatial subdivision phase, we partition $\mathbb{R}^3$ into a set $\mathcal{C}$ of polytopes with the property that the triangles in S do not intersect the interiors of the cells in $\mathcal{C}$. We then build a graph that explicitly represents the adjacencies between the cells in $\mathcal{C}$. We represent each cell in $\mathcal{C}$ by a node in the graph. If two cells in $\mathcal{C}$ are adjacent, i.e., they share a planar boundary, we add a link between the two corresponding nodes in the graph. Note that any partition of $\mathbb{R}^3$ (e.g., a tetrahedral decomposition) will satisfy our purposes, as long as the triangles in S are contained in the faces of the cells in $\mathcal{C}$. In our implementation, the cells in $\mathcal{C}$ correspond to the polytopes associated with the leaves of a BSP $\mathcal{B}$ for S.

We now outline our algorithm for constructing $\mathcal{B}$. Let P be the set of planes supporting the triangles in S and let $k = |P|$; $k < n$ if S contains coplanar triangles. With each plane $\pi \in P$, we associate the total area of the triangles in S that are contained in π, and sort the planes in P in non-increasing order of area. Let $\pi_i, 1 \leq i \leq k$ be the ith plane in the sorted order. We construct BSP in k stages by adding the planes in P one-by-one in sorted order. Let $\mathcal{B}_i$ denote the top subtree of $\mathcal{B}$ constructed after i stages; $\mathcal{B}_0$ consists of a single node corresponding to $\mathbb{R}^3$ and $\mathcal{B}_k$ is $\mathcal{B}$. In the ith stage, we use π_i to split all leaves $v \in \mathcal{B}_{i-1}$ such that a triangle in S that is contained in π_i intersects $\mathcal{R}_v$. Similar procedures are described in Section 5.1 and in Teller's thesis [101].

We dove-tail the construction of the cell adjacency graph with the construction of $\mathcal{B}$. Each node in the graph represents the polytope $\mathcal{R}_v$ associated with a leaf $v \in \mathcal{B}$. Each link represents the convex polygon corresponding to the face common to the polytopes corresponding to two leaves of $\mathcal{B}$; we augment each link with lists of triangles describing the link's opaque portions (those contained in some triangle in S) and the link's transparent portions (those not contained in any triangle in S). During the construction of $\mathcal{B}$, if π_i partitions $\mathcal{R}_v$ for a leaf $v \in \mathcal{B}_i$, we delete the node in the graph corresponding to v, create new nodes in the graph corresponding to the children of v, and update the links of v's neighbours to reflect the new adjacencies. For each updated link, we perform two more operations: (i) we subtract from the transparent part of the link any triangle $s \in S$ that is contained in π_i and (ii) we add s to the link's opaque part.

Figure 2.3 shows an example spatial subdivision (in the plane). The input "triangles" are shown in Figure 2.3(a) as thick line segments. The leaf nodes of $\mathcal{B}$ (regions labelled with letters) are constructed using splitting "planes" (dashed lines labelled with numbers) that support input "triangles," as shown in Figure 2.3(b). Finally, the cell adjacency graph for this example is shown in Figure 2.3(c) with the opacity of each link indicated by its line style (solid lines represent the opaque parts of a link, while dashed lines represent the transparent parts).



(a) Example input model with spatial subdivision.
(b) Binary space partition
(c) Cell adjacency graph

Figure 2.3: Example of a spatial subdivision.

2.2.2 Determination of solid regions

During the solid determination phase, we compute whether each cell in $\mathcal{C}$ is solid or not, based on the properties of its links and neighbours. This approach is motivated by the following observations:

1. if two adjacent cells share a mostly transparent boundary, it is likely that they are both solid or both non-solid,

2. if two adjacent cells share a mostly opaque boundary, it is likely that one is solid and the other is non-solid, and

3. unbounded cells (like cell E in Figure 2.3(a)) are not solid.

We quantify "how solid" each cell $C_i \in \mathcal{C}$ is by its *solidity*, $s_i \in [-1, 1]$. We use $s_i = 1$ to denote that C_i is solid (i.e., contained in the interior of a solid object), and $s_i = -1$ to denote that C_i is non-solid (i.e., lies in the exterior of all solid objects). An s_i value between -1 and 1 indicates that we are not entirely sure whether C_i is solid or not.

Our solid determination algorithm proceeds as follows. First, we assign a solidity value of -1 to all unbounded cells since they are in the exterior of all solid objects. We then compute the solidity s_i of each bounded cell C_i based on the solidities of its neighbour cells and the opacities of its links. Formally, let $a_{i,j}$, $o_{i,j}$, and $t_{i,j}$ represent the total area, opaque area, and transparent area, respectively, of the link $L_{i,j}$ between cells C_i and C_j. Note that $a_{i,j} = o_{i,j} = t_{i,j} = 0$ if and only if C_i and C_j are not adjacent (do not share a planar boundary); otherwise, $a_{i,j} > 0$ and $o_{i,j}, t_{i,j} \geq 0$. Let the total area of the boundary of C_i be denoted by $A_i = \sum_j a_{i,j}$, where C_j ranges over all neighbours of C_i. Now, we can write an expression for the solidity of each bounded cell C_i in terms of the solidities of every other cell C_j:

$$s_i = \frac{\sum_j (t_{i,j} - o_{i,j})\, s_j}{A_i} \tag{2.2.1}$$

This formulation for computing cell solidities matches our intuition. We would like the contribution of cell C_j to s_i to be proportional to $a_{i,j}$, the area of the link $L_{i,j}$. When the link $L_{i,j}$ between two cells C_i and C_j is entirely transparent ($t_{i,j} = a_{i,j}$ and $o_{i,j} = 0$), we expect the solidities s_i and s_j to be close to each other; hence, we scale s_j by $a_{i,j}$ to calculate the contribution of C_j to s_i. On the other hand, when $L_{i,j}$ is entirely opaque ($o_{i,j} = a_{i,j}$ and $t_{i,j} = 0$), we expect s_i and s_j to have "opposite" values; hence, we scale s_j by $-a_{i,j}$ to calculate the contribution of C_j to s_i. Finally, when $L_{i,j}$ is partially opaque ($0 < t_{i,j} < a_{i,j}$ and $0 < o_{i,j} < a_{i,j}$) the contribution from C_j to s_i is a linear interpolation between these two extremes. We divide the total contribution to s_i by A_i to normalise the value of s_i between -1 and 1.

If there are m bounded cells in $\mathcal{C}$, (2.2.1) leads to a linear system of m equations, $Mx = b$, where

- $x \in \mathbb{R}^m$ is a vector of the (unknown) solidities of the bounded cells,
- $b \in \mathbb{R}^m$ is a vector representing the contributions from the (known) solidities of the unbounded cells ($b_i = \sum_k (o_{i,k} - t_{i,k})$, where k ranges over all unbounded neighbours of C_i),
- and M is an $m \times m$ matrix with the following properties (here i and j are integers with $1 \leq i, j \leq m$ and $i \neq j$):
  1. Each diagonal element is positive, i.e., $M_{i,i} = A_i > 0$.
  2. $M_{i,j} = o_{i,j} - t_{i,j}$. Thus, $M_{i,j} > 0$ indicates that $L_{i,j}$ is mostly opaque, and $M_{i,j} < 0$ indicates that $L_{i,j}$ is mostly transparent.
  3. M is symmetric, i.e., $M_{i,j} = M_{j,i}$.

4. M has weak diagonal dominance, i.e., $\sum_{j,j\neq i}|M_{i,j}| \leq |M_{i,i}|$, for $1 \leq i \leq m$ and for some $1 \leq k \leq m$, $\sum_{j,j\neq k}|M_{k,j}| < |M_{k,k}|$ (for example, if C_k has an unbounded neighbour or if C_k has at least one link that is not fully opaque or fully transparent).

We now prove that M has an inverse.

Lemma 2.2.1 *If* M *is a matrix with weak diagonal dominance, i.e.,*

$$\sum_{j,j\neq i}|M_{i,j}| \leq |M_{i,i}|,$$

for all i *and there is a* k *such that*

$$\sum_{j,j\neq k}|M_{k,j}| < |M_{k,k}|, \tag{2.2.2}$$

then M^{-1} *exists.*

Proof: Assume that M has no inverse. Then the determinant $\det M = 0$. As a result, there exists a non-zero vector u such that $Mu = 0$ [106, Theorem 1.4]. Let i be the index such that $|u_i| = \max_j |u_j|$. Note that $|u_i| > 0$ since u is not a zero vector. Now, $Mu = 0$ implies that

$$M_{ii}u_i = -\sum_{j,j\neq i} M_{ij}u_j.$$

Using the diagonal dominance of M and taking absolute values in the above equation, we have

$$\begin{aligned}\left(\sum_{j,j\neq i}|M_{ij}|\right)|u_i| \leq |M_{ii}||u_i| &\leq \sum_{j,j\neq i}|M_{ij}||u_j| \\ &\leq \left(\sum_{j,j\neq i}|M_{ij}|\right)|u_i|,\end{aligned}$$

since $|u_i| = \max_j |u_j|$. As a result, all the inequalities above turn into equalities. Hence,

$$\sum_{j,j\neq i}|M_{ij}||u_j| = \left(\sum_{j,j\neq i}|M_{ij}|\right)|u_i|.$$

But $|u_i| = \max_j |u_j|$. Therefore,

$$|u_1| = |u_2| = \cdots = |u_n|. \tag{2.2.3}$$

Solving for u_k in $Mu = 0$, where k is the row of M that causes M to be weakly diagonally dominant, we have

$$M_{kk}u_k = -\sum_{j,j\neq k} M_{kj}u_j.$$

Taking absolute values in the above equation and using (2.2.2), we have

$$\begin{aligned}\left(\sum_{j,j\neq k} |M_{kj}|\right)|u_k| < |M_{kk}||u_k| &\leq \sum_{j,j\neq k} |M_{kj}||u_j| \\ &= \left(\sum_{j,j\neq k} |M_{kj}|\right)|u_k|, \qquad \text{by (2.2.3).}\end{aligned}$$

Therefore, $|u_k| < |u_k|$, which is a contradiction. Hence $\det M \neq 0$ and M has an inverse. □

This lemma implies that the linear system $Mx = b$ has a unique solution. It is not difficult to show that the elements of x have values between -1 and 1. We can solve the linear system to obtain the cell solidities by computing $x = M^{-1}b$. However, inverting M takes $O(m^2)$ time, which can be prohibitively costly if $\mathcal{B}$ has many leaves, as is likely to be the case if there are many triangles in S. In such cases, we take advantage of the fact that most leaves in $\mathcal{B}$ are likely to have a small number of neighbours. Therefore, M is sparse, and we can use an iterative procedure to solve the linear system efficiently [106].

In our implementation, we use Gauss-Seidel iterations. Each iteration takes time proportional to the number of links in the adjacency graph. We set the initial values of the solidities of the bounded cells of the subdivision to be 0. We terminate the iterations when the change in the solidity of each cell is less than some small pre-specified tolerance.

After solving the linear system of equations, we classify each cell as solid or not by looking at the sign of its solidity. A cell whose solidity is positive is determined to be solid, whereas a cell whose solidity is negative or zero is determined to be non-solid.

2.2.3 Model output

Finally, in the model output phase, we output consistent descriptions of the objects represented by the input triangles. To generate a consistent representation, we simply output a list of triangles describing all links in the adjacency graph that represent the boundaries between cells that are solid (cells with positive solidity) and cells that are not solid (cells with zero or negative solidity). We consistently orient all output triangles away from solid cells (see Figure 2.2(d)).

Another format we support is a solid-based representation that represents the solid objects by explicitly encoding $\mathcal{B}$ as a hierarchical set of cutting planes augmented by a solidity value for each leaf cell.

2.3 Results and Discussion

We have implemented our solid reconstruction algorithm and run it on a number of data sets. When the input model is a manifold triangulated surface, the boundary representation output by our algorithm is identical to the input model, as desired. In this case, all cells lie entirely in the interior or exterior of the modelled objects, and the solidity computed by our algorithm is exactly 1 for every cell in the interior of the solid object and exactly -1 for every cell in the exterior of all solid objects. This follows from the fact that each link is either fully opaque or fully transparent. Note that unlike previous boundary-based approaches, our algorithm additionally outputs a representation of the modelled object as the union of a set of convex polyhedra.

In many complex cases, when the input model contains errors, our algorithm is able to fix errors automatically and output consistent solid and boundary representations, even in cases where previous approaches are unsuccessful. For instance, consider the three-dimensional model of a coffee mug shown in Figure 2.4(a) on page 28. In this example, triangles are oriented incorrectly (back-facing triangles are drawn in black); the handle is modelled by several improperly intersecting and disconnected hollow cylinders (note the gaps along the top silhouette edge of the handle); and the triangles at both ends of the handle intersect the side of the cup.

All previous approaches known to us fail for this simple example. Boundary-based approaches that traverse the surface of the object [61] fail in the areas where triangles are unconnected (along the handle). Proximity-based approaches [15, 22, 94] that merge features within some tolerance of each other do not work as no suitable tolerance can be chosen for the entire model because the size of the largest error (a crack between triangles on the handle) is larger than the size of the smallest feature (a bevel on the top of the lip). Finally, solid-based approaches [101, 102] that decide whether each cell is solid or not based on the orientations of the input triangles along the cell's boundaries fail because the input has many wrongly-oriented triangles.

Our algorithm is able to fix the errors in this example and output a correct and consistent model (see Figures 2.4(b)– 2.4(d)). Figure 2.4(b) shows outlines of the cells constructed during the spatial subdivision phase, with each cell labelled by its solidity computed during the solid determination phase. In addition, each cell C_i is outlined with a colour that depends on the value of its solidity s_i (the colour ranges from red when $s_i = 1$ to green when $s_i = -1$). Figure 2.4(c) shows triangles computed during the model output phase; each triangle is adjacent to one solid and one non-solid cell, has a normal pointing away from the solid cell, and is drawn with the colour of the solid cell it is adjacent to. Finally, Figure 2.4(d) shows the boundary representation output by our algorithm. The reconstructed model is correct: the cracks in the handle have been filled in; intersections in the handle have been made explicit; and all triangles have been oriented correctly. This experiment was run on an SGI Indigo2 with a 200MHz R4400 processor. The model of the mug contained 121 triangles. The BSP we constructed for the mug contained 359 leaves. We needed 61 Gauss-Seidel iterations to determine the solidities. The experiment took 9.77 seconds to run, with 35% of the time spent on calculating solidities.

Figures 2.5–2.7 show results derived from experiments with larger models from a variety of applications. The images on the left side of the last page show the input models, while the images on the right show different visualisations of the cell solidities computed for these models with our algorithm. In all these cases, we were able to construct correct and consistent solid and boundary representations. The three sets of images demonstrate the importance of appropriate visualisation techniques for viewing the solid cells of the BSP. For instance, the text strings drawn in Figure 2.5(b) would be overlapping if used in Figure 2.6(b) and 2.7(b). Similarly, the opaque boundaries drawn in Figure 2.6(b) would be inappropriate for use in Figure 2.7(b) as the outermost solid cells (ceilings and floors) would mask the interior non-solid cells. In complex cases, such as Figure 2.7(b), we simply represent cell solidities by coloured dots drawn at the cells' centroids. In all cases, solidity information is drawn with a representative colour linearly interpolated between red (for $s_i = 1$) and green (for $s_i = -1$). We use a three-dimensional viewing program to allow the user to select different visualisation options interactively.

We now present an analysis of the running time of our algorithm for the above models. In the table below, each row corresponds to the model whose name is specified in the first column. The second column displays the number of triangles in the model. The third column contains the number of cells in the spatial subdivision constructed for that model. The fourth and fifth columns specify the number of Gauss-Seidel iterations needed for convergence and the total running time for the model in seconds, respectively. These experiments were run on SGI Indigo2 with a 200MHz R4400 processor. For the mug, about 35% of the total time was spent on calculating solidities, while for the other models, this time ranged from 10-15%.

Model	#polys	#cells	#iter.	time
Mug	121	359	61	9.77
Clutch	420	159	39	8.64
Phone	1228	819	82	78.64
Building	1687	1956	92	240.34

Table 2.1: Performance of the model repair algorithm

Our algorithm has several advantages since we use a *global* approach to classify *regions of space* rather than just considering local boundary relationships or feature proximities. First, our algorithm is effective for models containing intersecting, overlapping, or unconnected triangles for which it is difficult to traverse boundaries. Second, the output of our algorithm does not depend on the initial orientations of input triangles. Third, the boundary output by our algorithm is *always* guaranteed to be consistent (although it may not be a correct representation of the modelled object) since it is derived directly from the solid cells of the partition. Finally, we are able to output a solid representation of the model as well as a boundary representation, which may be critical to many applications.

However, our approach does have limitations. Its success depends on the spatial subdivision constructed. As a result, missing triangles may lead to the creation of cells that do not correctly model the shape of the solid object. Another limitation of our technique is that it is based on the assumption that input triangles separate solid and non-solid regions. Therefore, if the input model contains two solid objects that are intersecting or separated by a triangle in the input model (e.g., a mouse on a table), the solidities for the cells along the solid-on-solid boundary are driven by each other to values lower than 1. Intuitively, the triangle separating the solid objects is an "extra" triangle.

Fortunately, in many cases, cells with intermediate solidity values (i.e., values close to 0) identify parts of the model containing topological errors and inconsistencies like missing and extra triangles. This feature is useful for verifying model consistency and localising model inaccuracies. For example, in Figure 2.4(b) and 2.4(c), cells are red (solidity close to 1) in areas where the input model has no errors (on the left side of the cup) and cells are yellow-ish (solidity close to 0) in areas where there are intersecting or unconnected triangles in the input model. This example demonstrates an important feature of our algorithm: it not only helps fix up errors, but also identifies where they are.

2.4 Conclusions

We have described an algorithm that reconstructs consistent solid and boundary representations of triangles from error-ridden polygonal data. The algorithm partitions $\mathbb{R}^3$ into a set of cells, computes the solidity of each cell based on cell adjacency relationships, and utilises computed cell solidities to construct a consistent output representation. In contrast to previously described approaches, our method gives excellent results on many real models in $\mathbb{R}^3$ containing intersecting, overlapping, and unconnected polygons.

An important issue that we have not considered in this chapter is how "far away" the manifold we output is from the original set of solid objects. It would be very interesting to examine questions of the following kind: let S be a set of triangles obtained by randomly perturbing (by a small amount) the triangles forming a closed 2-manifold $\mathcal{M}$. Let $\mathcal{M}'$ be the manifold formed by the set of triangles output by our algorithm when it is given S as input. What is the volume of the symmetric difference of $\mathcal{M}$ and $\mathcal{M}'$ as a fraction of the volume of $\mathcal{M}$? We could use other measures of closeness between $\mathcal{M}$ and $\mathcal{M}'$, for example, the number of components in their symmetric difference. We expect answers to such questions to exploit interesting properties of the BSP.

As regards algorithmic extensions to our work, the limitations of our approach that we discussed in the last section suggest two directions for future research. One approach is to determine the location of missing/extra triangles as $\mathcal{B}$ is being constructed. Recall that we construct $\mathcal{B}$ in stages. After the ith stage, we can compute the solidities of the leaves of $\mathcal{B}_i$. If the solidity of such a leaf ν is close to 0 and there is a plane π (not necessarily supporting a triangle in S) that splits $\mathcal{R}_\nu$ so that at least one of the two new nodes has solidity close to -1 or 1, then it is likely that π contains a missing triangle.

Another avenue for future work is to utilise final cell solidity values to recognise missing/extra triangles. A possible approach is to first define a metric that measures the "goodness" of the solution. For example, the metric could penalise solutions in which two cells with markedly different (resp., similar) solidities share a mostly transparent (resp., opaque) boundary. Once a suitable metric is chosen, we can associate weights with the transparent and opaque areas of each link and use the metric to drive a simulated annealing or optimisation process that searches for that set of weights that maximises the goodness of the solution.

Note that the two ideas outlined above are not restricted to BSPs and can be generalised to any spatial decomposition. It will also be interesting to combine our solid-based algorithm with the boundary-based techniques we discussed in Section 2.1. We expect these extensions to allow our algorithm to adapt dynamically when the model has missing triangles and solid-on-solid regions, and thus generate correct solutions for a wider class of input models.

Chapter 3

Object Complexity Hidden-Surface Removal

We present our object-complexity algorithm for hidden-surface removal of a set S of n triangles in $\mathbb{R}^3$ in this chapter. For the sake of completeness, we describe the motivation for the object complexity model once again. Traditionally, computational geometry algorithms for hidden-surface removal compute the *visibility map,* which is a partition of the image plane into maximally-connected regions, such that in each region, at most one triangle in S is visible. In the worst case, the size of the visibility map (called the *scene complexity*) can be $\Omega(n^2)$, where n is the number of triangles in the input, e.g., when the viewpoint is at $z = +\infty$ and the scene contains $n/2$ thin rectangles parallel to the x-axis lying directly above $n/2$ thin rectangles parallel to the y-axis. See Figure 3.1. As a result, any algorithm that computes the visibility map takes $\Omega(n^2)$ time in the worst case.

On the other hand, we saw in Chapter 1.2 that the simple and popular z-buffer algorithm solves the hidden-surface removal problem in $O(n + a)$ time, where a is the total area of the projections of the triangles in S onto the image plane. For the massive data sets that we are interested in (such as models of aircraft and submarines), the running time of the z-buffer algorithm can be approximated as $O(n)$; these models can have hundreds of millions of triangles, and it is reasonable to assume that the projection of most visible triangles in these models has small area. The popularity of the z-buffer algorithm stems from its simplicity and the fact that it can be implemented

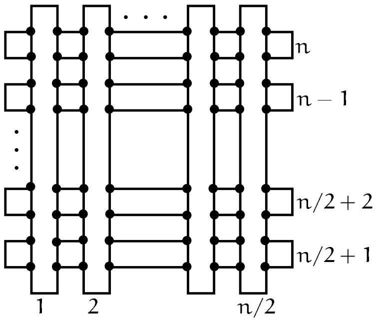


Figure 3.1: A set of n objects whose object complexity is $O(n)$ and scene complexity is $\Omega(n^2)$.

very efficiently in hardware. However, even state-of-the-art z-buffers incorporated in graphics systems like SGI's InfiniteReality [68], which can render 7 million triangles a second, are not fast enough to do hidden-surface removal on massive data sets at interactive rates of 60 scenes a second. Note that the z-buffer must process all visible triangles in order to render the visible scene correctly. Hence, the number of visible triangles is a lower bound on the running time of the z-buffer algorithm.

Motivated by these observations, we propose the *object complexity* model, in which we measure the size of a scene by the number of triangles visible in it. Given a viewpoint, an object-complexity algorithm for hidden-surface removal determines the set of triangles that are visible from the viewpoint and renders the visible set using the z-buffer. The object complexity of a scene is always less than n, the number of triangles in S, and can be much less than the scene complexity, which can be $\Omega(n^2)$ in the worst case. Algorithms that compute the visibility map can be used trivially to determine visible objects by outputting all the objects that contain edges in the visibility map. However, in the worst case, this might entail spending $\Omega(n^2)$ time to output $O(n)$ distinct objects.

In the computer graphics community, a commonly used model that is similar to object complexity is the *conservative visibility* model proposed by Teller [101]. In this model, an algorithm computes a superset of the set of visible triangles and renders this superset using the z-buffer. The critical observation used in this approach is that computing a superset is enough to guarantee that the rendered scene is correct.

The challenge in developing such algorithms lies in ensuring that the superset is not much larger than the actual set of visible triangles. This technique is also known as *occlusion* or *visibility culling*, since triangles that are occluded by other triangles are not rendered in this approach.

In the next section, we survey known occlusion-culling algorithms. We highlight the features of our algorithm in Section 3.2 before proceeding to describe it in detail in Section 3.3. In Section 3.3.6, we present some possible extensions to the algorithm. Finally, we discuss some open problems suggested by our algorithm Section 3.4.

3.1 Occlusion Culling Algorithms

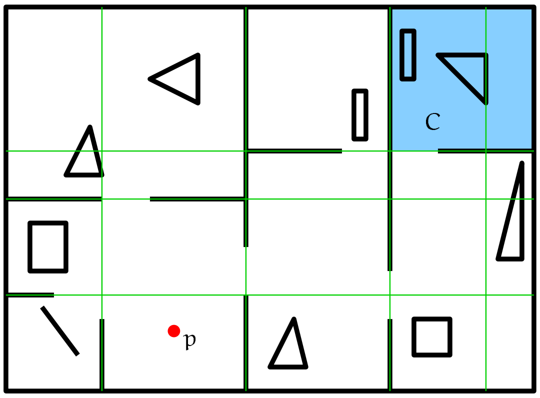


Figure 3.2: Cell C is not visible from p, so the segments intersecting it are not rendered.

Most occlusion-culling algorithms work on the same general principle. They first partition $\mathbb{R}^3$ into a (hierarchical) set of convex cells $\mathcal{C}$. Typically, these cells form a uniform grid, octree, or BSP. The crucial observation that these algorithms use is that if a cell $C \in \mathcal{C}$ is not visible, then *all* the triangles in S that are contained in the interior of C are also invisible. See Figure 3.2. Thus the visibility status of all these triangles is resolved just by testing the visibility of C. The algorithms render only those triangles in S that intersect visible cells. The success of these algorithms critically depends on how efficiently they determine if a cell $C \in \mathcal{C}$ is visible. To do so, the algorithms use the "shadows" cast by the triangles in S. A cell C is deemed to be invisible if it is contained in the union of the shadows of the triangles. Occlusion-culling algorithms that have been presented in the literature primarily differ in how they represent and compute the union of these shadows. We first examine these differences. Then we discuss how these algorithms handle moving triangles. Finally, we discuss how the algorithms exploit the continuity of motion of the viewpoint and the input triangles.

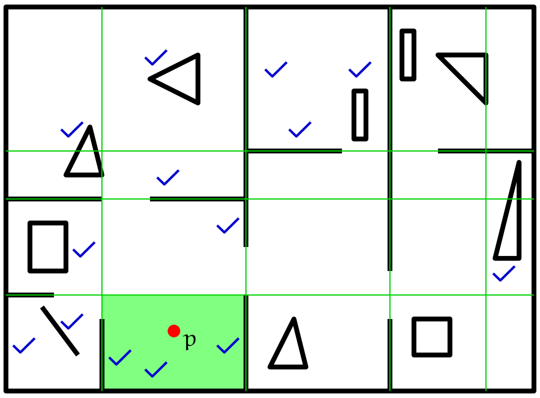
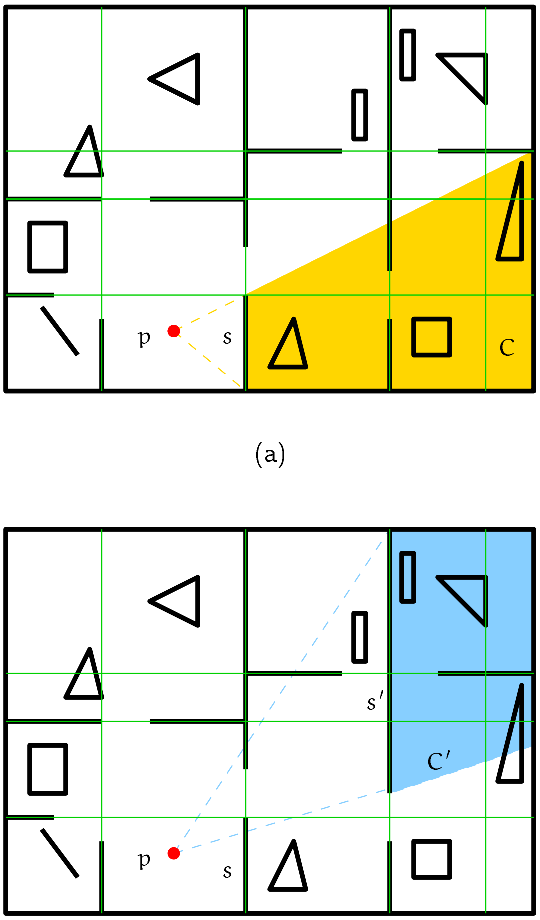


Figure 3.3: Marked segments lie in the potentially-visible set of the cell containing p.

In a preprocessing phase, for each cell $C \in \mathcal{C}$, Airey [8] and Teller [101] compute the *potentially-visible set* for C, which is the set of triangles that are visible from any point in C. See Figure 3.3. Teller first constructs transparent portals between the cells. To compute the potentially-visible set for a cell C, he traverses the cells in $\mathcal{C}$ starting at C and uses linear programming to maintain lines of sight between C and other cells. If there is a line of sight between C and another cell D, the triangles intersecting D are added to C's potentially-visible set. Given a viewpoint, his algorithm locates the cell containing it, and uses lines of sight from the viewpoint to cull the potentially-visible set for the cell in order to generate a set of triangles for rendering. Luebke and Georges [65] describe a modification of Teller's algorithm that requires very little pre-processing but is likely to produce larger sets of triangles for rendering than Teller's algorithm. Airey's and Teller's techniques use extensive preprocessing time and memory. While all these algorithms work very well for architectural environments, they do not seem suitable for other classes of data sets, such as CAD models, terrains, and urban landscapes, that do not lend themselves to being decomposed into cells and portals.



(a)

(b)

Figure 3.4: (a) Cell C is occluded by the shadow of segment s. (b) Cell C' is jointly occluded by the shadows of segments s and s' but not by any single shadow.

Algorithms have been developed for more general scenes by Coorg and Teller [33, 34] and Hudson et al. [58]. These approaches manipulate the three-dimensional shadows cast by visible triangles. For each triangle (or set of connected triangles with a convex silhouette) in S, these techniques intersect each cell $C \in \mathcal{C}$ with the triangle's shadow to determine whether C is visible. See Figure 3.4(a). Both Coorg and Teller [34] and Hudson et al. [58] use the interesting idea of dynamically picking a subset of the input triangles (called *occluders*) and processing the shadows of only the occluders. The intuition behind this method is that in densely-occluded scenes, it is likely that most invisible triangles are occluded by a small subset of the triangles. Since computing this subset is exactly the same as the hidden-surface removal problem itself, the authors present heuristics to estimate good occluders. A related algorithm of Naylor [77] processes the triangles in front-to-back order and uses a BSP in the image plane to represent the projected shadows. The disadvantage with all these algorithms is that they do not handle the case when a cell is invisible because it is occluded by multiple, disconnected triangles. See Figure 3.4(b).

In order to handle occlusion caused by disconnected triangles, Greene et al. [53] represent the union of the shadows by projecting each triangle onto the image plane, and constructing a quadtree for the set of projected triangles. Each leaf of the quadtree corresponds to a pixel in the z-buffer; the leaf stores the distance value associated with that pixel. The distance stored with each interior node in the quadtree is the maximum of the distances stored at the node's children. Greene et al. call such a quadtree a *z-pyramid*. In order to determine if a cell is visible, they use the z-pyramid to check whether the projection of each face of the cell onto the image plane is visible. While their algorithm is elegant, fast computation of the z-pyramid is unfortunately not supported by current graphics hardware. Greene [52] incorporates this technique into a scheme for fast *polygon tiling* (the problem of determining which pixels in the image plane are covered by visible portions of a set of polygons). The main contribution of this work seems to be the polygon tiling technique itself; if the tiling algorithm is implemented efficiently in hardware, it could be used to speed up other occlusion-culling algorithms.

Zhang et al. [108] develop a related approach, where the quadtree has no distance information stored in it. Instead each node stores an opacity value that is average of the opacities stored at the node's children; the opacity represents the fraction of the node's region that is covered by projected triangles. They call such a quadtree a *hierarchical occlusion map*. Construction of the occlusion map is supported by current graphics hardware. Since the occlusion map does not store any distance information, they use a separate *depth-estimation buffer* to store information about the distance of the triangles from the viewpoint. Zhang et al. use both the occlusion map and the depth-estimation buffer to determine if a triangle is visible.

While these two techniques are general and applicable to arbitrary polygonal models, they may not perform occlusion culling effectively for all models, since they use a discrete and approximate representation of the union of the shadows. If the quadtree-based approximations are not close to the actual union of shadows, these algorithms might render many invisible triangles. Moreover, in the case of the algorithm developed by Zhang et al., the fact that the distance information and the image-plane coverage information are stored in different data structures seems to limit the effectiveness of the visibility test.

Let us now examine how different authors handle moving triangles. Greene [52]'s polygon-tiling technique requires that the input triangles be traversed in front-to-back order. He suggests storing triangles belonging to each moving object in data structures similar to octrees and BSPs, and merges these data structures to determine the front-to-back order of the triangles. When the front-to-back order is not easily determined, he outlines a "lazy z-buffering" approach to resolve visibility. Zhang et al. simply check the visibility of each moving object in each frame. Sudarsky and Gotsman [97, 98] extend Naylor's technique [77] to handle moving objects. They use the interesting idea of enclosing each moving object in a *temporal bounding volume* based on known estimates on the objects's velocity. The store the temporal bounding volume for each moving object in a BSP. Their algorithm attempts to process a moving object only when it becomes visible or when its temporal bounding volume becomes invalid. Their method is the closest in spirit to the notion of kinetic data structures that we use in our algorithm for hidden-surface removal, which we discuss in Section 3.3. However, they still merge BSPs for static objects and visible moving objects in each frame, an operation that could be costly.

Finally, we turn our attention to how these algorithms exploit temporal coherence.

Recall that temporal coherence is the phenomenon where the set of visible triangles changes gradually as the viewpoint and triangles move. As we noted in Chapter 1.1, most algorithms approximate time by a set of evenly-spaced intervals called frames. At the end of each frame, Coorg and Teller [34] store visibility information for each visible cell, and check if this information is still valid at the end of the next frame. While this procedure performs better than recalculating the information for each cell (whether the cell is visible or not) at every frame, their method can update visibility information for many cells whose visibility status does not change. During each frame, Greene et al. [53] keep track of which cells in $\mathcal{C}$ are visible and use the visible cells to initialise the z-pyramid at the beginning of the next frame. Then they traverse the cells in $\mathcal{C}$ to determine those cells whose visibility status has changed. Zhang et al. [108] use a feedback mechanism to improve occluder selection if their algorithm detects that the occluders picked for the current frame do not occlude many triangles. However, in each frame, they construct the occlusion map and the depth-estimation buffer from scratch and check each triangle in the input for visibility.

One lacuna these methods exhibit is that they make use of temporal coherence to speed up only certain portions of the algorithm. Other, possibly time-consuming steps have to be executed from scratch at every frame. A more serious disadvantage is (as we pointed out in Chapter 1.1) that these algorithms exploit the continuity of viewpoint and object motion only indirectly since they approximate continuous motion by a discrete sampling of time.

The ultimate aim of occlusion culling is to reduce the number of triangles rendered, while ensuring that all visible triangles are rendered. *Simplification* is an orthogonal approach that also tries to reduce the number of triangles rendered but might not render all visible triangles. In this approach, if a cell C projects onto a small area on the image plane, the set of triangles that intersect it are rendered in a simplified form. Erickson surveys some of the literature on simplification [42]. Funkhouser and Sequin [49] present a framework for combining occlusion culling with simplification to achieve bounded frame rates. Recent techniques have used images and textures to represent simplified versions of triangles [10, 93, 88]. Aliaga et al. [9] present a rendering system that incorporates occlusion culling, simplification, and image-based techniques.

3.2 Features of Our Algorithm

In this section, we discuss the features of our algorithm for the hidden-surface removal of polygonal models. Like other occlusion-culling algorithms, we (i) compute a hierarchical spatial decomposition, (ii) carefully select a small subset of occluders, (iii) process the union of the shadows cast by the occluders to classify cells of the decomposition as visible or invisible, and (iv) render triangles in S that intersect visible cells. However, our technique has several new and attractive features, which we summarise below:

- We represent the union of the shadows of the occluders as a set of rays in $\mathbb{R}^3$; these rays correspond to the edges of the union of the shadows. As a result, we can compute the set of visible cells exactly, and can also detect when a cell is occluded by multiple, disconnected triangles. Previous algorithms for occlusion culling do not possess both these properties. Note that the union of shadows of the occluders is essentialy the same as the visibility map of the occluders. Nevertheless, our approach is more powerful and effective than previous algorithms that compute the visibility map for two reasons. Firstly, the occluders we pick are a small subset of the input triangles. Hence, the visibility map of these occluders is much smaller than the number of triangles in S even in the worst case. Secondly, we pick occluders that are likely to obscure most invisible triangles and we store non-occluders in a hierarchical spatial partition (the BSP), so we can determine that large subsets of the triangles in S are invisible with a small number of visibility tests.

- We explicitly exploit the continuity of the motion of the viewpoint and the triangles by using kinetic data structures [13]: based on the current motion of the viewpoint and the triangles, we predict the instants when the combinatorial structure of the set of edges of the union of the shadows (we define the combinatorial structure later) and the set of visible cells change, and develop techniques to efficiently update the union of the shadows and the set of visible cells at each such event. As a result, unlike the techniques we surveyed in the previous section, we avoid recomputing our data structures from scratch or traversing each cell in the partition at regular intervals.

- We use the binary space partition (BSP) as a unifying data structure for occluder selection and for maintaining visibility information. We also use the BSP in a natural way to balance the time spent by the algorithm in internal processing with the time spent by the graphics system in rendering triangles input to it by the algorithm. This mechanism provides a means for controlling the *frame rate* (the number of scenes drawn every second) achieved by our algorithm. In future work, we plan to investigate how we can exploit the properties of the BSP to perform view-dependent simplification in addition to occlusion culling and frame-rate control. As Aliaga et al. [9] point out, when rendering large models, it is crucial to construct a single data structure that supports all the techniques used to accelerate rendering.

- If we pick *all* the triangles in S as occluders, our algorithm computes the set of visible triangles *exactly* and maintains this set efficiently as the viewpoint and triangles move. Thus, our algorithm provides a means for comparing different occlusion-culling techniques: the measure of an algorithm's performance is the ratio of the size of the superset of visible triangles output by it and the size of visible set computed by our algorithm. Previously, algorithms were compared based on the fraction of input triangles they culled away, a number that may have no relation to the number of triangles that are actually visible.

- Our algorithm can be easily modified to compute the *change* in the set of visible triangles. This property is very useful when our algorithm is running on a server, and is connected by a slow network to a client that is displaying the scene. In such situations, our algorithm delivers only updates to the visible set over the network.

- One of the primary reasons we use the BSP as our primary data structure is that it supports *ray-shooting* queries: *given a query ray ρ, determine the first triangle in S that ρ intersects.* The simple observation we use is that such a triangle must intersect a leaf node of the BSP that is also intersected by the ray. Indeed, this observation is the basis of many well-known spatial partitions such as octrees that are used for answering ray-shooting queries (see the survey by

Arvo and Kirk [11]). As a result, our algorithm can be modified to use any of these ray-shooting data structures.

3.3 Our Algorithm

In this section, we describe our object complexity algorithm for hidden-surface removal in detail. In the framework of kinetic data structures that we develop our algorithm, we assume that we know the instantaneous laws of motion of the viewpoint and the triangles in S. These laws can be changed at anytime by notifying the kinetic data structure. As we have pointed out earlier, we do not update our data structures at fixed intervals. Instead, we do so at instants when certain elementary geometric conditions called *certificates* fail. An example of a certificate we use is the fact that a ray originating at the viewpoint and passing through the vertex of a triangle in S intersects the interior of a different triangle in S. See Figure 3.5 for an example.

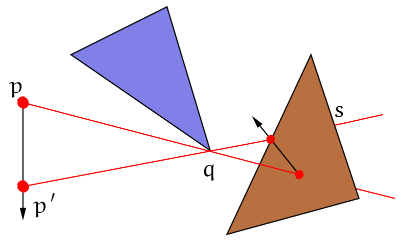


Figure 3.5: Example of a certificate used by our occlusion culling algorithm: the ray originating at p and passing through q intersects the triangle s. When p moves to p', the ray stops intersecting s.

We parametrise the motion of the viewpoint and the triangles in S by time and use t to denote time. For a given time t, we use t^- and t^+ to denote the instants $t - \varepsilon$ and $t + \varepsilon$, respectively, where $\varepsilon > 0$ is a suitably small constant.

In our model of motion, we assume that the trajectory of the viewpoint is specified as $p(t) = (x(t), y(t), z(t))$, where $x(t), y(t)$, and $z(t)$ are continuous (usually algebraic) functions of time t. Similarly, we assume that the trajectory of a triangle $s \in S$ is specified by the trajectory of a fixed point in s (if s is moving rigidly) or by the trajectories of the vertices of s (if s is not moving rigidly). The certificates we use will be algebraic functions (linear or quadratic) of the positions of the viewpoint and a constant number of vertices of triangles in S. Therefore, we assume that we can compute the failure time of a certificate and compare the failure times of two certificates in constant time, under standard models of computation.

3.3.1 Overview of the algorithm

To aid the description of our algorithm, we start with some definitions. Given a viewpoint p, the *shadow* σ_s^p of a triangle s is

$$\sigma_s^p = \{q \mid \text{ the segment } pq \text{ intersects } s\},$$

the set of points q such that the segment joining p and q intersects s; σ_s^p is an unbounded convex polyhedron. See Figure 3.6(a). Given a set of triangles $O \subseteq S$, we use Σ_O^p to denote $\bigcup_{s \in O} \sigma_s^p$, the union of the shadows of the triangles in O; Σ_O^p is an unbounded polyhedron. See Figure 3.6(b). We say that a point $q \in \mathbb{R}^3$ is *visible* if the segment pq does not intersect any triangle in O. Let $\mathcal{C}$ be a partition of $\mathbb{R}^3$ into convex polyhedra; we will refer to the polyhedra in $\mathcal{C}$ as *cells*. We say that a cell $C \in \mathcal{C}$ is *visible* if C intersects $\mathbb{R}^3 - \Sigma_O^p$, i.e., if a point $q \in C$ is visible. See Figure 3.6(c). We use $\mathcal{V}_O^p \subseteq \mathcal{C}$ to denote the subset of cells in $\mathcal{C}$ that are visible from p. We will often use the notation σ_s, Σ_O, and $\mathcal{V}_O$ since the viewpoint will be clear from the context.

Armed with these definitions, we first give a high-level description of our algorithm and then explain how we implement each of the steps. We now make two assumptions to help us describe our key ideas clearly: that the triangles in S are static and that the set O does not change as the viewpoint moves. We relax these assumptions in later sections.

In a pre-processing phase, we execute the following steps:

1. We pick a set O of occluders (say, by including the walls, ceilings, and floors in a building in O).

2. We construct a BSP $\mathcal{B}$ for O.

3. We set $\mathcal{C}$ to be the set of polytopes associated with the leaves of $\mathcal{B}$. For each cell $C \in C$, we compute S_C, the set of triangles in S that intersect C.

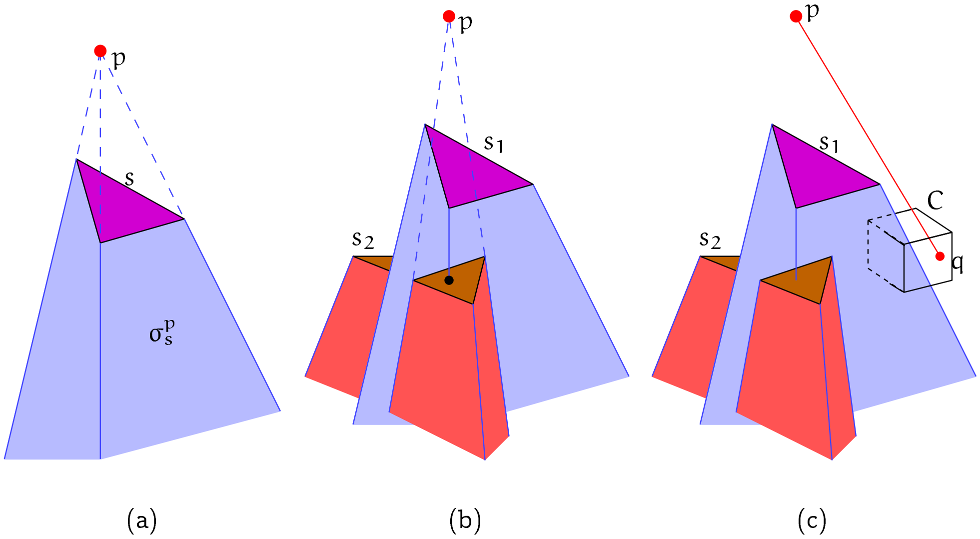


Figure 3.6: (a) The shadow of triangle s. (b) The union of shadows of triangles s_1 and s_2. (c) A visible cell C (point q in C is visible).

Recall that the definition of the BSP (which we gave on 12) implies that no triangle in O intersects the interior of a cell in $\mathcal{C}$. We will use this property crucially in Section 3.3.4.

As the viewpoint moves, the union of shadows $\Sigma_O^{p(t)}$ and $\mathcal{E}_O^{p(t)}$, the set of edges of $\Sigma_O^{p(t)}$, changes continuously. However, the combinatorial structure of $\mathcal{E}_O^{p(t)}$ (we will define this notion precisely in the next section) changes only at discrete instants. Our kinetic algorithm and data structures have the property that for any value of t, we maintain the set $\mathcal{E}_O^{p(t)}$ of edges of the union of shadows and the set $\mathcal{V}_O^{p(t)}$ of visible cells correctly. We store the events when $\mathcal{E}_O^{p(t)}$ and $\mathcal{V}_O^{p(t)}$ change in a priority queue, keyed by the time they occur. We repeatedly remove the next event from the queue, and update $\mathcal{E}_O^{p(t)}$ and $\mathcal{V}_O^{p(t)}$, as required. In order to actually render the visible scene, we output the set $\{S_C \mid C \in \mathcal{V}_O^p(t)\}$ of triangles intersecting visible cells to the graphics system at periodic intervals (say, every $1/60$ seconds). This completes the high level description of the algorithm.

As we will see, our algorithm rests on the efficient implementation of what we call ray-dragging and segment-dragging queries. Since we will use these queries often in the next two sections, we define these queries now. The *ray-dragging* query can be defined as follows (see Figure 3.7(a)):

> *Given the ray $\rho(t)$ originating at the point $p(t)$ and passing through a given edge of a triangle $s_1 \in O$ and a given edge of a triangle $s_2 \in O$, and given the point $q(0)$ at which $\rho(0)$ first intersects the interior of a triangle $\tau \in O$, find the smallest positive value t' such that the segment $p(t')q(t')$ intersects the boundary of a triangle $s \in O - \{s_1, s_2\}$ and return t' and s.*

If $\rho(0)$ does not intersect any triangle, then $q(t)$ is the point at infinity along $\rho(t)$; otherwise, $q(t)$ is the intersection of $\rho(t)$ and τ. The triangles s_1 and s_2 specified in the query may be the same triangle. Note that s might be the triangle τ itself.

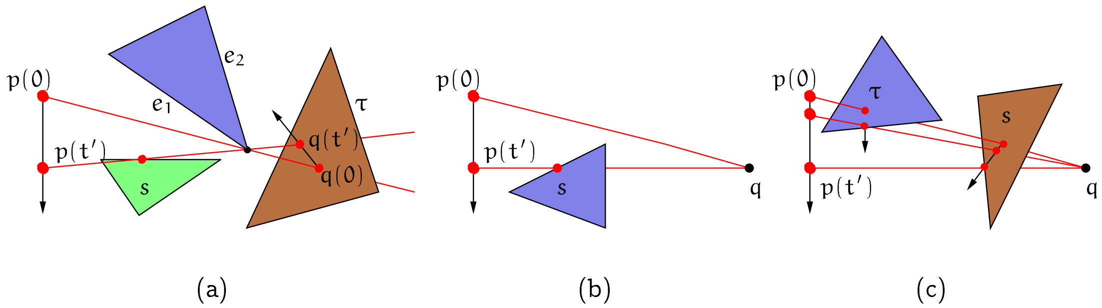


Figure 3.7: (a) A ray-dragging query. At t', the ray defined by edges e_1 and e_2 intersects triangle s. (b) A segment-dragging query of the first type. At t', the segment $p(t')q$ starts intersecting the triangle s. (c) A segment-dragging query of the second type. The segment $p(0)q$ intersects the triangle τ. At t' the segment $p(t')q$ stops intersecting the triangle s. Note that s, the triangle returned, is different from τ.

The *segment-dragging query* can be similarly defined (it has two versions): *Given a point $p(t)$ and a static point q,*

1. *if the segment $p(0)q$ does not intersect the interior of any triangle in O, find the smallest positive value t' such that the segment $p(t')q$ intersects the boundary of a triangle $s \in O$ and return t' and s. See Figure 3.7(b).*
2. *given the first triangle $\tau \in O$ whose interior the segment $p(0)q$ intersects, find the smallest positive value t' such that the segment $p(t')q$*

does not intersect the boundary of any triangle in O *and return* t' *and the last triangle* s *intersected by* $p(t)q$*. See Figure 3.7(c).*

In the second case, it is possible that s and τ may be different triangles, as in Figure 3.7(c).

To complete the description of the algorithm, in the next two sections we describe how we maintain the sets $\mathcal{E}_O^{p(t)}$ and $\mathcal{V}_O^{p(t)}$. For each set, we characterise the instants when the set changes, the certificates we maintain to detect these changes, how we use ray-dragging and segment-dragging queries to compute when the certificates expire, and how we process each event. In Section 3.3.4, we describe our technique for answering ray- and segment-dragging queries. We show how to dynamically modify the set O as the viewpoints move in Section 3.3.5. In Section 3.3.6, we describe how to handle moving triangles and present some extensions to our algorithm.

3.3.2 Maintaining edges of the union of shadows

In this section, we describe how we maintain $\mathcal{E}_O$, the edges of the union Σ_O of shadows. Note that some edges of Σ_O are portions of edges of triangles in O. We *do not* include these edges in $\mathcal{E}_O$. We first describe the combinatorial structure of the edges in $\mathcal{E}_O$. Each edge e of $\mathcal{E}_O$ is contained in a ray ρ_e originating at the viewpoint; ρ_e intersects two edges e_1 and e_2 of triangles in O. The edge e has two vertices; the vertex farther from the viewpoint is either the vertex at infinity along ρ_e or the intersection of ρ_e with the interior of a triangle τ_e in O; τ_e is called the *stopper* of e. The other endpoint q of e can be of two types:

1. If e_1 and e_2 are edges of the same triangle $s \in O$, q is the vertex of s that e_1 and e_2 are incident on; we call e a *primary* edge and say that q *defines* e. A primary edge is illustrated in Figure 3.8(a).

2. If e_1 and e_2 are edges of different triangles in O, assume without loss of generality that e_2 is farther away from the viewpoint than e_1; q is the intersection of ρ_e and e_2. We call e a *derived* edge and say that the pair (e_1, e_2) *defines* e. Figure 3.8(b) displays a derived edge.

If e is a primary edge defined by a vertex q, we say that ρ_e is a *primary* ray *defined* by q. We use similar terminology for ρ_e if e is a derived edge. The *combinatorial representation* of an edge $e \in \mathcal{E}_O$ is the vertex or pair of edges that define it and the stopper τ_e. In order to detect and process changes to $\mathcal{E}_O$, for each edge in $\mathcal{E}_O$, we store its combinatorial representation.

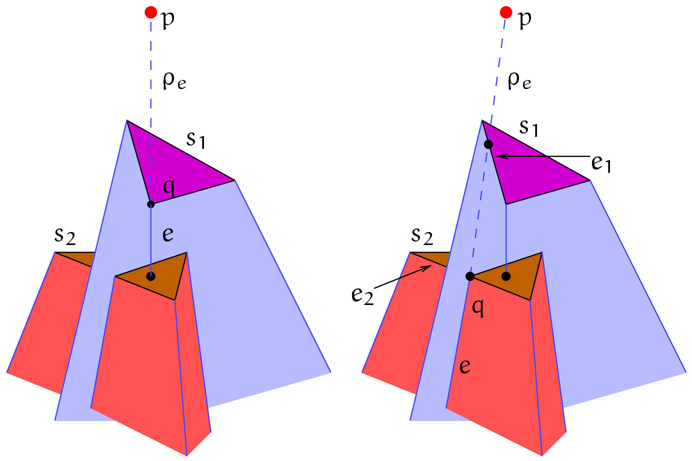


Figure 3.8: (a) A primary ray e defined by the vertex q of triangle s_1. The triangle s_2 is the stopper of e. (b) A derived edge e defined by edge e_1 of triangle s_1 and edge e_2 of triangle s_2.

We first state a lemma that characterises the conditions that determine the combinatorial structure of an edge of $\mathcal{E}_O$. The proof of this lemma is easy to derive and we omit it.

Lemma 3.3.1 *Let e be an edge of $\mathcal{E}_O^p$. If e is a primary edge, let q be the triangle vertex defining it. If e is a derived edge defined by a pair of triangle edges e_1 and e_2, where e_1 is closer than e_2 to p, let $q = \rho_e \cap e_2$. The following two conditions are true:*

1. *The segment pq does not intersect the interior of any triangle in O.*

2. *If the stopper τ_e exists, let q' be the intersection of ρ_e and τ_e; otherwise,*

q' is the point at infinity along ρ_e. The segment qq' does not intersect the interior of any triangle in $O - \{\tau_e\}$.

It is also not difficult to see that these conditions are both necessary and sufficient to define all the set of edges of $\mathcal{E}_O$.

This lemma suggests that for each edge $e \in \mathcal{E}_O$, we store the following certificates:

Stopper certificate: ρ_e intersects τ_e (at the point q', which is the point at infinity along ρ_e if τ_e does not exist).

Validity certificate: the segment pq' does not intersect the interior of any triangle in $O - \{\tau_e\}$.

In addition, we will find it useful to maintain the following *invisibility certificates*. We define $\mathcal{I}_O$ to be the set of invisible vertices of triangles in O. For each such vertex q, we maintain an invisibility certificate, which guarantees that q is contained in Σ_O. We use τ_q to denote the first triangle in O whose interior the segment pq intersects. A natural question that arises at this point is whether we should also maintain such invisibility certificates for all pairs of triangle edges that do not define edges of Σ_O. As we will see, we do not need such certificates because new derived edges appear only when one of the above certificates becomes invalid.

As the viewpoint moves, these certificates become invalid if and only if one of the following two conditions is true:

Triangle vertex event: A vertex q of a triangle $s \in O$ intersects the boundary of Σ_O, or

Triangle edge event: An edge e of a triangle $s \in O$ intersects a ray $\rho_{e'}$, where e' is an edge of Σ_O.

We can compute these events by asking appropriate queries:

1. For each edge $e \in \mathcal{E}_O$, we use a ray-dragging query to compute the first time ρ_e intersects an edge of a triangle in O.

2. For each triangle vertex $q \in \mathcal{I}_O$, we use a segment-dragging query of the second type to compute the first time that the segment pq does not intersect the interior of any triangle in O.

Next, we discuss how to process triangle vertex and triangle edge events. We describe the procedure in detail only when the event corresponds to a vertex or edge of s leaving the interior of Σ_O. The procedure for the case when a vertex or edge of s enters the interior of Σ_O is symmetrical.

Triangle vertex events Suppose a vertex q of s intersects a face of Σ_O that is defined by an edge e' of a triangle in O. Let e_1 and e_2 be the edges incident on q. We delete q from $\mathcal{T}_O$, and add the primary ray defined by q and the derived rays defined by (e', e_1) and (e', e_2) to $\mathcal{E}_O$.

Triangle edge events Suppose edge e of a triangle $s \in O$ intersects the ray $\rho_{e'}$, for an edge $e' \in \mathcal{E}_O$. If e' is a primary edge defined by a vertex q of a triangle $s' \in O$, let e_1 and e_2 be the edges of s' that are incident on q. Otherwise, let e_1 and e_2 be the pair of edges defining e. We have three cases to consider:

1. If e' is a primary edge, note that q is closer to p than the point $q' = \rho_{e'} \cap e$. We set $\tau_{e'} = s$. We delete the derived edges defined by (e_1, e) and (e_2, e) from $\mathcal{E}_O$.

2. If e' is a derived edge and e is farther from the viewpoint than both e_1 and e_2, we set $\tau_{e'} = s$ add the derived edges (e_1, e) and (e_2, e) to $\mathcal{E}_O$.

3. Otherwise, we delete the derived edge (e_1, e_2) from $\mathcal{E}_O$ and add the derived edge (e_2, e) to $\mathcal{E}_O$.

Finally, for each edge e we add to $\mathcal{E}_O$, we compute the next triangle vertex/edge event it will be involved in by making a ray-dragging query using e and store the event in the event queue.

Remark: We can modify this algorithm to maintain which triangles in O are visible. For each triangle $s \in O$, we keep track of which edges of $\mathcal{E}_O$ are (partially) defined by the vertices and edges of s, and for which edges of $\mathcal{E}_O$, s is the stopper. From this information, we can deduce whether s is visible or not.

We now analyse the time taken to process each event. Clearly, the update procedures take constant time and add a constant number of edges to $\mathcal{E}_O$ and a constant number of vertices to $\mathcal{T}_O$. Adding an edge involves computing its stopper (by invoking a ray-shooting query) and asking a ray-dragging query for the edge. Similarly, adding a vertex to $\mathcal{T}_O$ involves a ray-shooting query and a segment-dragging query. Further, each time we process an event, the set $\mathcal{E}_O$ changes. As a result, the following theorem is immediate:

Theorem 3.3.1 *Let O be a set of n triangles in $\mathbb{R}^3$ and let $p(t)$ be a point moving along a continuous trajectory. At every instant that the set of edges of $\Sigma_O^{p(t)}$ changes, we can update it in $O(k(n))$ time, where $k(n)$ is the time taken to answer a ray-shooting, ray-dragging, or segment-dragging query for a set of n triangles in $\mathbb{R}^3$. If the set of edges of $\Sigma_O^{p(t)}$ changes m times during the motion, the total time we spend in maintaining it is $O(mk(n))$.*

3.3.3 Maintaining visible cells

In this section, we describe our technique for maintaining the set $\mathcal{V}_O$ of visible cells. We first prove a lemma that characterises when a cell $C \in \mathcal{C}$ is visible.

Lemma 3.3.2 *A cell $C \in \mathcal{C}$ is visible from a point p if and only if at least one of the following conditions is true:*

1. *A vertex q of C is visible, i.e., the segment pq does not intersect any triangle in O.*
2. *An edge e of C intersects the boundary of Σ_O.*
3. *An edge of Σ_O intersects the interior of C.*

Proof: It is clear that C is visible if any of the conditions in the lemma is true. Suppose C is visible but that each vertex and edge of C is contained in Σ_O. Let r be a point in C that is visible and let q be a vertex of C. Since q is contained in Σ_O, the segment rq intersects a face f of Σ_O, say at a point r'. Since each edge of C is contained in Σ_O, f does not intersect any edge of C. However, the point r' lies inside both f and C. Therefore, an edge on the boundary of f must intersect the interior of C, which completes the proof. $\square$

Thus, for each cell $C \in \mathcal{C}$, we maintain the following certificates that attest to its visibility:

1. *vertex certificates*: the set of visible vertices of C,
2. *edge certificates*: the set of edges of C that intersect the boundary of Σ_O, and
3. *shadow certificates*: the set of edges of Σ_O intersect the interior of C.

For each edge certificate e, we also maintain the sequence F_O^e of faces of Σ_O that e intersects. If C has no such certificates, it is not visible. As the viewpoint moves, the certificates become invalid (possibly changing C's visibility status) if and only if one of the following two conditions is true (see Figure 3.9):

Cell vertex event: A vertex q of C intersects the boundary of Σ_O, or

Cell edge event: An edge e of C intersects an edge of Σ_O.

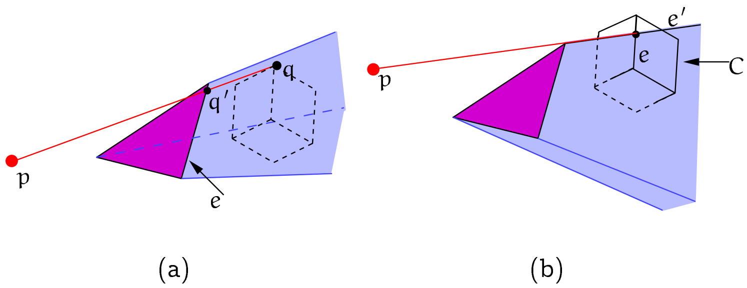


Figure 3.9: (a) Cell vertex event: the segment joining p and vertex q of cell C intersects the edge e at q'. (b) Cell edge event: edge e of the cell intersects edge $e' \in \mathcal{E}_O$.

Note that these conditions are essentially identical to the conditions that characterise when triangle vertex and triangle edge events happen. We define $\mathcal{Q}_\mathcal{C}$ to be the set of vertices of the cells in $\mathcal{C}$. We compute cell vertex and cell edge events as follows:

1. For every vertex $q \in \mathcal{Q}_\mathcal{C}$, we use a segment-dragging query with the segment pq to compute when q will intersect the boundary of Σ_O.
2. For every shadow certificate e of a cell $C \in \mathcal{C}$, we use a ray-dragging query to compute when ρ_e will intersect an edge of C.

We now discuss how we process these events. We describe the procedure in detail only when the event corresponds to a vertex or edge of C leaving the interior of Σ_O. The procedure for the case when a vertex or edge of C enters the interior of Σ_O is symmetrical. Let t be the time the event happens.

Cell vertex events Let vertex q of cell C intersect a face f of Σ_O.We add q to the list of vertex certificates of C. For each edge e of C that is incident on v, we insert f at the beginning of F^e_O. If F^e_O was empty at t^-, we add e to the list of edge certificates of C. If C had no certificates at t^-, we add C to $\mathcal{V}_O$.

Cell edge events Let edge e of cell C intersect an edge e' of Σ_O and let e' be incident on faces f_1 and f_2 of Σ_O. Recall that e' is a shadow certificate for C. Therefore, e' intersects the interior of C at t^-. We have two cases:

1. If e' is a primary edge, we remove f_1 and f_2 from F^e_O. If F^e_O is now empty, we remove e from the list of edge certificates of C.

2. If e' is a derived edge, we add f_1 and f_2 to F^e_O. If F^e_O was empty at t^-, we add e to the list of edge certificates of C.

We delete e' from the list of shadow certificates of C. Note that C is visible at t^+. For each cell C' incident on e, we add e' to the list of shadow certificates of C'. If C' had no certificates at t^-, we add C' to $\mathcal{V}_O$.

In addition, for each newly added vertex and shadow certificate, we compute when it expires by asking the appropriate segment- or ray-dragging queries and store the resulting events in the event queue.

3.3.4 Ray-dragging queries

In this section, we describe how we implement ray-dragging queries. These techniques can be modified to answer both types of segment-dragging queries. The details are straightforward and we leave them to the reader. We recapitulate the definition of the ray-dragging query to refresh the memory of the reader. The *ray-dragging* query can be defined as follows (see Figure 3.10):

Given the ray $\rho(t)$ originating at the point $p(t)$ and passing through a given edge of a triangle $s_1 \in \mathcal{O}$ and a given edge of a triangle $s_2 \in \mathcal{O}$ and given the point $q(0)$ at which $\rho(0)$ first intersects the interior of a triangle $\tau_\rho \in \mathcal{O}$, find the smallest positive value t' such that the segment $p(t')q(t')$ intersects the boundary of a triangle $s \in \mathcal{O} - \{s_1, s_2\}$ and return t' and s.

If $\rho(0)$ does not intersect any triangle, then $q(t)$ is the point at infinity along $\rho(t)$; otherwise, $q(t)$ is the intersection of $\rho(t)$ and τ_ρ. The triangles s_1 and s_2 specified in the query may be the same triangle. Note that s might be the triangle τ_ρ itself. As $p(t)$ moves continuously, $\rho(t)$ sweeps out a surface in $\mathbb{R}^3$. In particular, if $p(t)$ moves along a straight line, then the swept surface is a plane if s_1 and s_2 are the same triangle and a quadric if they are not [96].

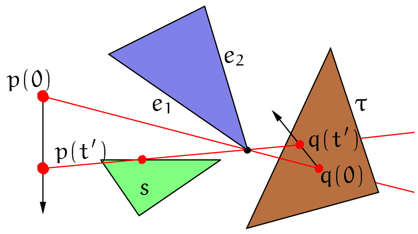


Figure 3.10: A ray-dragging query. At t', the ray defined by edges e_1 and e_2 intersects triangle s.

Note that such queries can be answered in sub-linear time by known range-searching data structures [1, 6]. Unfortunately, these techniques are complicated and difficult to implement. Our kinetic data structure for ray-dragging queries is simple and can be implemented easily. While we have been unable to prove any non-trivial bounds on its performance, there is evidence in the computer graphics literature that our algorithm is likely to perform efficiently in practice [11, 74].

Let $F_\rho(t)$ be the sequence of faces of cells in $\mathcal{C}$ that the initial segment $p(t)q(t)$ of the ray $\rho(t)$ intersects. In the rest of the section, we use just ρ to denote $\rho(t)$, and similarly remove the parametrisation in terms of time from all moving objects. We will find the following lemma useful in developing our technique:

Lemma 3.3.3 *If the initial segment pq of the ray ρ intersects the boundary of a* *triangle* s *in* O*, then there is a face* f *in* F_ρ *such that* $s \cap f \neq \emptyset$*.*

Proof: Each cell in $\mathcal{C}$ is the polytope $\mathcal{R}_\nu$, for some leaf $\nu \in \mathcal{B}$, a BSP for O. This implies that s does not intersect the interior of any cell in $\mathcal{C}$. Therefore, the intersection of ρ and s is contained in a face f for some cell in $\mathcal{C}$. Clearly, f is an element of F_ρ. □

For each face $f \in F_\rho$, let s_f be the first triangle contained in f that ρ intersects. The above lemma implies that the we can answer the ray-dragging query by maintaining the sequence of intersected faces F_ρ and by computing s_f for each face $f \in F_\rho$. For each face $f \in F_\rho$, let p_f denote intersection of ρ and f. We maintain the following certificates for ρ:

Stopper certificate: the triangle τ_ρ, if it exists.

Face certificates: the set of faces F_ρ and a certificate for each face $f \in F_\rho$ that p_f does not lie in the interior of any triangle in O that intersects the interior of f.

These certificates become invalid if and only if one of the following conditions is true:

Exit stopper event: If τ_ρ exists, ρ intersects the boundary of τ_ρ.

Face event: For some face $f \in F_\rho$, p_f intersects the boundary of f.

Enter stopper event: For some face $f \in F_\rho$, p_f intersects the boundary of a triangle $s \in O$ that intersects the interior of f.

We can compute the exit stopper event and each face event in constant time, and each enter stopper event in time proportional to the number of triangles intersecting a face. We process each event as follows (let t be the time the event occurs and let $f \in F_\rho$ be the face involved in the event):

Face event: Let e be the edge of f that the ray intersects. We delete f from F_ρ. For each face f' of a cell in $\mathcal{C}$ that is incident on e, we add f' to F_ρ if $\rho(t^+)$ intersects f' and compute face and enter stopper events for f'.

Enter stopper event: Let $s \in O$ be the triangle that ρ intersects. We set $\tau_\rho = s$. We delete all faces in F_ρ that are farther away from p than f. We also remove all events associated with these faces that are stored in the event queue. Finally, we return the current time t and the triangle s as the answer to the ray-dragging query.

Exit stopper event: Let $s' \in \mathcal{O}$ be the current value of τ_ρ. We perform a ray-shooting query with the portion ρ' of ρ beyond τ_ρ. Let s be the first triangle in $\mathcal{O}$ whose interior ρ' intersects. We append the faces intersected by ρ' to F_ρ and set $\tau_\rho = s$. For each inserted face, we compute face and enter stopper events. We also compute an exit stopper event for s. Finally, we return the current time t and the triangle s' as the answer to the ray-dragging query.

3.3.5 Selecting occluders dynamically

One of the assumptions we made in Section 3.3.1 was that we pick the set $\mathcal{O}$ of occluders in a pre-processing phase. Such a strategy is successful in architectural environments, where most invisible triangles are occluded by the walls, floors, and ceilings in the model. For other classes of models, such as those arising in CAD design, urban landscapes, or terrains, this technique is not tenable since different viewpoints have different sets of good occluders. In this section, we show how we can pick occluders dynamically as the viewpoint moves, thus removing one of the assumptions we made in Section 3.3.1. The intuition behind picking occluders dynamically is that we should select those triangles whose projections onto the image plane have large size. For example, Coorg and Teller [34] and Hudson et al. [58] estimate the solid angle subtended by a triangle at the viewpoint and select triangles with large solid angles as occluders.

The intuition behind our method is that good occluders are likely to be close to the viewpoint. Since the set of cells $\mathcal{C}$ used by our algorithm corresponds to the leaves of a BSP for $\mathcal{O}$, the set of occluders, the partition of $\mathbb{R}^3$ induced by $\mathcal{C}$ is likely to be "fine" near the viewpoint and progressively "coarser" as we move away from the viewpoint. Informally, the density of the cells in $\mathcal{C}$ (the number of cells intersecting a fixed volume) decreases with increasing distance from the viewpoint. Moreover, as the viewpoint moves, the sets $\mathcal{O}$ and $\mathcal{C}$ gradually change, thus changing the induced partition of $\mathbb{R}^3$.

Our novel technique for occluder selection is motivated by these observations. It is based on the fact that we use a BSP as the underlying data structure to maintain visibility information. In a pre-processing phase, we construct a BSP $\mathcal{B}$ for *all* the triangles in S (in Section 3.3.1, we constructed a BSP only for the set O of occluders). As the viewpoint moves, we maintain a set $\mathcal{M}$ of marked nodes in $\mathcal{B}$ with the property that the set of polytopes corresponding to the nodes in $\mathcal{M}$ form a decomposition of $\mathbb{R}^3$. The following conditions are enough to ensure this property:

1. For each leaf $v \in \mathcal{B}$, we mark an ancestor of v.

2. If we mark a node $v \in \mathcal{B}$, we do not mark any ancestor of v.

Given the set $\mathcal{M}$, we define the set $\mathcal{C}$ of cells to be

$$\mathcal{C} = \{\mathcal{R}_v \mid v \in \mathcal{M}\},$$

the set of polytopes corresponding to the marked nodes and the set O of occluders to be

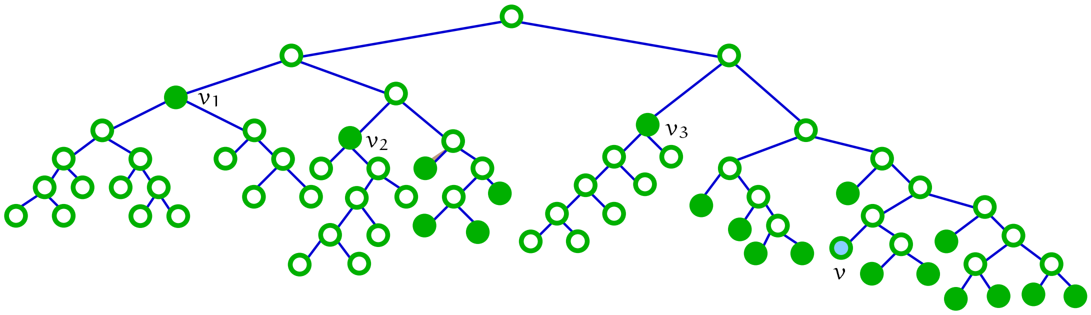
$$O = \{s' \mid s' = s \cap \partial C \neq \emptyset, s \in S, C \in \mathcal{C}\},$$

the portions of triangles in S that are contained in the boundaries of the cells in $\mathcal{C}$. An illustration of marked nodes and the cells and occluders they define is given in Figure 3.11. We apply our hidden-surface removal algorithm on O and $\mathcal{C}$, as just defined.

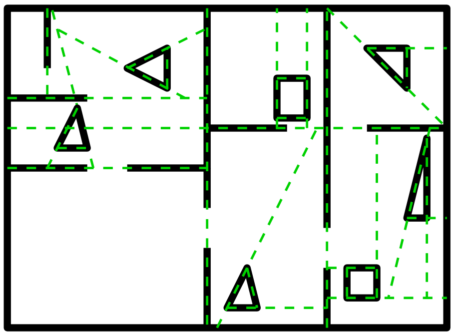
We now describe our technique for marking nodes and for maintaining $\mathcal{M}$ and O. For a node $v \in \mathcal{B}$, let d_v be the closest distance of $\mathcal{R}_v$ from the viewpoint p; we define d_v to be 0 if p is contained in $\mathcal{R}_v$. Let h_v denote the height of v, i.e., the length of the path from the root of $\mathcal{B}$ to v. We define the *measure* μ_v of v to be $\mu_v = h_v d_v$. We mark the nodes in $\mathcal{B}$ according to the following rule:

($*$) For every leaf $v \in \mathcal{B}$, mark the ancestor w of v such that $\mu_w \geq \kappa > 0$, where κ is a specified parameter (it can be input by the user). If there are many ancestors satisfying this inequality, we mark the one with least height. If there is no such ancestor, we mark v.

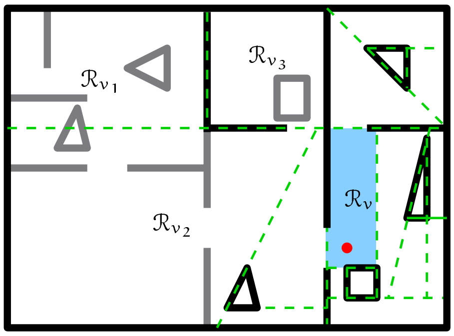
It is easy to verify that the nodes in $\mathcal{M}$ form a partition of $\mathbb{R}^3$. The condition we use to mark nodes corresponds to our earlier intution. Note that if the viewpoint p is contained in $\mathcal{R}_v$, where v is a leaf of $\mathcal{B}$, then we mark v. If a node v is close to the



(a)



(b)



(c)

Figure 3.11: (a) BSP for the entire input with marked nodes. (b) Decomposition induced by leaves of the BSP. Dashed lines are edges of the decomposition. (c) Decomposition induced by marked nodes. Occluders are drawn in black and non-occluders in grey. The viewpoint is contained in $\mathcal{R}_v$.

viewpoint (d_ν is small), we mark it only if its height is large (i.e., if ν is close to a leaf). If a node ν is far away from the viewpoint (d_ν is large), we mark it when its height is small (i.e., if ν is close to the root).

The set $\mathcal{M}$ changes as the viewpoint moves. We use properties of the function μ_ν to set up a kinetic data structure for maintaining $\mathcal{M}$. If ν is a node in $\mathcal{B}$ and ν_1 and ν_2 are its two children, we have $d_\nu = \min\{d_{\nu_1}, d_{\nu_2}\}$, which implies that $\mu_\nu < \mu_{\nu_1}, \mu_{\nu_2}$. Thus the measure of the nodes along any root-to-leaf path in $\mathcal{B}$ increases monotonically. Condition $(*)$ implies that the set $\mathcal{M}$ remains unchanged as long as for every node $\nu \in \mathcal{M}$, $\mu_\nu > \kappa$ and $\mu_{p(\nu)} < \kappa$, where $p(\nu)$ is the parent of ν. Therefore, the set $\mathcal{M}$ changes if and only if one of the following two conditions is true:

1. For a node $\nu \in \mathcal{M}, \mu = \kappa$.
2. For the parent $p(\nu)$ of a node in $\mathcal{M}$, $\mu_{p(\nu)} = \kappa$.

Thus, in order to detect when $\mathcal{M}$ changes, for each node $\nu \in \mathcal{M}$, we compute when μ_ν or $\mu_{p(\nu)}$ becomes equal to κ.

To process such an event, we either add a node to or delete a node from $\mathcal{M}$. We also have to update the sets $\mathcal{C}$, O, and $\mathcal{E}_O$. We now explain in detail how we perform these modifications. Let t be the time the event happens. Let ν be the node to be added to or deleted from $\mathcal{M}$ and let w and z be the children of ν. Recall that H_ν is the cutting plane at ν; let O_ν be the set of triangles that are contained in $H_\nu \cap \mathcal{R}_\nu$ (the portion of H_ν inside $\mathcal{R}_\nu$). At time t, we have $\mu_\nu = \kappa$. Clearly, $\mu_w, \mu_z > \kappa$ at all instants in the interval $[t^-, t^+]$.

Deleting a node Since $\mu_\nu < \kappa$ at t^+, we have to mark w and z at t^+.

1. We update $\mathcal{M}$ by deleting ν and adding w and z. We similarly update $\mathcal{C}$.
2. For each shadow certificate e of $\mathcal{R}_\nu$, we execute the following steps:
   (a) If the ray ρ_e intersects only $\mathcal{R}_w$ (resp., only $\mathcal{R}_z$), we add e as a shadow certificate of $\mathcal{R}_w$ (resp., $\mathcal{R}_z$).
   (b) Otherwise, assume without loss of generality that ρ_e intersects $\mathcal{R}_w$ first. We add e as a shadow certificate of $\mathcal{R}_w$. We also check if ρ_e intersects a

triangle $s \in O_v$. If it does not, we add e as a shadow certificate of $\mathcal{R}_z$. If it does, we handle ρ_e in the next step.

3. We add each triangle $s \in O_v$ to O, as follows (let $O' = O \cup \{s\}$):

Update $\mathcal{E}_O$, $\mathcal{I}_O$, **and** $\mathcal{Q}_\mathcal{C}$: For each edge $e \in \mathcal{E}_O$ such that the ray ρ_e supporting e intersects $s \cap H_v$,

(a) If e is a primary edge defined by a vertex q of a triangle in O, we check if s is closer to the viewpoint than q is. If it is, we delete e from $\mathcal{E}_O$ and add q to $\mathcal{I}_{O'}$ setting $\tau_q = s$. Otherwise, we set the stopper $\tau_e = s$.

(b) If e is a derived edge defined by two edges e_1 and e_2, we set $\tau_e = s$ if s is farther away from the viewpoint than both e_1 and e_2. We also delete all faces in F_{ρ_e} (the list of faces of cells in $\mathcal{C}$ intersected by ρ_e) that lie beyond s. Otherwise, we delete e from $\mathcal{E}_O$.

For each vertex $q \in \mathcal{I}_O \cup \mathcal{Q}_\mathcal{C}$ such that pq intersects $s \cap H_v$ before τ_q, we set $\tau_q = s$. If $q \in \mathcal{Q}_\mathcal{C}$ is a vertex of cell $C \in \mathcal{C}$ and q is visible, we remove q as a vertex certificate of C.

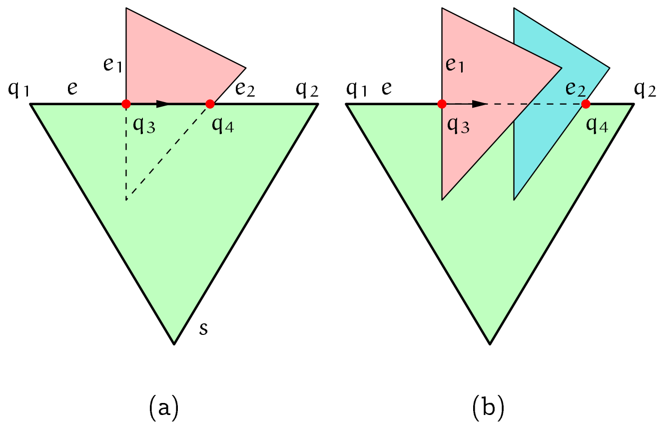


Figure 3.12: Discovering new derived edges defined by edge e of triangle s (drawn with thick edges): The answer to the first ray-dragging query is point q_3 and edge e_1. The answer to the second query is point q_4 and edge e_2. (The viewpoint is at $z = +\infty$ and the image plane is the xy-plane.) (a) Edge e_1 is behind edge e. The second query is a ray-dragging query. (b) Edge e_1 is in front of edge e. The second query is a segment-dragging query.

Compute edges of $\mathcal{E}_{O'}$ defined by s: For each vertex q of s, we check if it is visible. If it is, we add the primary edge defined by q to $\mathcal{E}_{O'}$. Otherwise, we add q to $\mathcal{I}_{O'}$. For each edge $e = q_1q_2$ of s, we discover derived edges of $\mathcal{E}_{O'}$ defined by e (and other triangle edges) by asking repeated ray- and segment-dragging queries as follows: Suppose q_1 is visible from p. Let $q(u)$ be a point parametrised by u such that $q(0) = q_1$, $q(1) = q_2$, and $q(u)$ lies on e for $0 \leq u \leq 1$. Let $r(u)$ be a ray passing through $q(u)$ with origin at p. We invoke a ray-dragging query to compute the first value $u', 0 < u' < 1$ such that $r(u')$ intersects the boundary of a triangle in O; let e' the intersected edge. There are two cases to consider:

(a) If $r(u')$ intersects e before it intersects e', we add the derived edge (e, e') to $\mathcal{E}_{O'}$, and find the next derived edge defined by e by asking another ray-dragging query, this time starting at $q(u')$. See Figure 3.12(a).

(b) Otherwise, we add the derived edge (e', e) to $\mathcal{E}_{O'}$. Note that $q(u' + \varepsilon)$ is not visible from p, for any infinitesimally small value of ε. See Figure 3.12(b). To discover the next derived edge defined by e, we reparametrise $q(u)$ so that $q(0) = r(u') \cap e$ and compute the first value $u', 0 < u' < 1$ such that $q(u')$ is visible from p. We compute u' by asking a segment-dragging query of the second type.

In this manner, by asking a series of ray-dragging queries and segment-dragging queries, we move $q(u)$ along e until we reach q_2 and discover all the derived edges of $\mathcal{E}_{O'}$ defined by e and other triangle edges. If q_1 is not visible from p, we start the process with a segment-dragging query of the second type.

4. We compute visibility certificates for $\mathcal{R}_w$ (the steps are similar for $\mathcal{R}_z$) as follows:

Vertex certificates: For each vertex q of $\mathcal{R}_w$, we check if it is visible. If it is, we add q as a vertex certificate for $\mathcal{R}_w$.

Edge certificates: To compute edge certificates for each edge e of $\mathcal{R}_w$, we construct the sequence $F^e_{O'}$ of faces of $\Sigma_{O'}$ that e intersects. We can do so by modifying the procedure described above for computing the derived edges defined by a triangle edge. The primary difference is that we now ask a series of segment-dragging queries of both types to compute the edge certificates. The other details are easy to work out.

Shadow certificates: We have created shadow certificates for $\mathcal{R}_w$ in Steps 2 and Step 3.

Adding a node Since we did not mark v at t^-, we must have marked both w and z at t^-.

1. We update $\mathcal{M}$ by deleting w and z and adding v. We similarly update $\mathcal{C}$ by deleting $\mathcal{R}_w$ and $\mathcal{R}_z$ and adding $\mathcal{R}_v$. If $\mathcal{R}_w$ or $\mathcal{R}_z$ is visible (i.e., it is in $\mathcal{V}_O$), we delete it from $\mathcal{V}_O$ and include $\mathcal{R}_v$ in $\mathcal{V}_O$.

2. For each triangle $s \in O_v$, we delete s from O and perform the following operations (let $O' = O \setminus \{s\}$):

   Extend edges of $\mathcal{E}_O$, $\mathcal{I}_O$, and $\mathcal{Q}_\mathcal{C}$: For each edge $e \in \mathcal{E}_O$ such that $\tau_e = s$, we shoot ρ_e, the ray containing e beyond s, and update τ_e. For each vertex $q \in \mathcal{I}_O \cup \mathcal{Q}_\mathcal{C}$ such that $\tau_e = s$, we check if pq intersects any triangle $s \in O$. If it does, we set $\tau_e = s'$. Otherwise,

   (a) If $q \in \mathcal{I}_O$, we delete q from $\mathcal{I}_O$ and add a primary ray defined by q to $\mathcal{E}_O$.

   (b) If q is a vertex of a cell $C \in \mathcal{C}$, we add q as a vertex certificate of C, and add C to $\mathcal{V}_O$ if C has no visibility certificates at t^-.

   Discover new edges of $\mathcal{E}_{O'}$: Let Φ_s be the set of all rays originating at p and intersecting s; Φ_s is an infinite polyhedron that contains σ_s, the shadow of s. The deletion of s causes new edges to appear in $\mathcal{E}_{O'}$. For each such edge e, ρ_e is contained in Φ_s. We find these edges by using a technique first described by Overmars and Sharir [79] and later used by other authors. It is easy to prove that one of the new edges is either

   (a) a primary edge defined by a vertex q of a triangle in O' such that s occludes q (i.e., $q \in \mathcal{I}_O$ and $\tau_{pq} = s$) at t^-, or

(b) a derived edge defined by an edge e of a triangle in O' (and an edge of another triangle in O') such that there is a derived edge (e, e_1) or (e_1, e) in $\mathcal{E}_O$, where e_1 is an edge of s.

See Figure 3.13 for an illustration of these types of vertices.

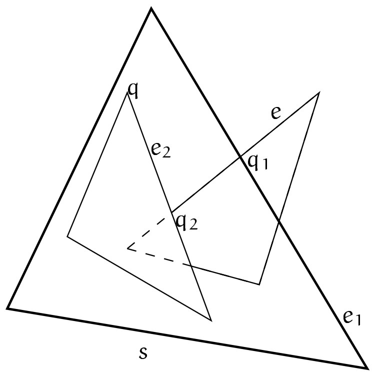


Figure 3.13: Types of new edges in $\mathcal{E}_{O'}$ that appear when s is deleted (the viewpoint is at $z = +\infty$ and the image plane is the xy-plane). The primary edge $q \in \mathcal{E}_{O'}$ is occluded by s at t^-. The derived edge $q_2 \in \mathcal{E}_{O'}$ is defined by edges e and e_2, and e (and edge e_1) define the derived $q_1 \in \mathcal{E}_O$.

Note that we have already computed all such invisible vertices in Step 2. From each such invisible vertex and each edge e of a triangle in O' such that e and an edge of s define a derived edge in $\mathcal{E}_O$, we perform a series of ray-dragging queries as described below. In general, we have an edge e of a triangle in O', a point q on e, and a direction along e such that if we traverse e from q along this direction, we follow a visible portion of e inside Φ_s (visibility is with respect to the triangles in O'). Let $q(u)$ be a point paramterised by u, such that $q(0) = q$ and $q(u)$ moves along e in the given direction as u increases. Let $r(u)$ be a ray with origin at the viewpoint p and passing through $q(u)$. We invoke a ray-dragging procedure with $r(u)$ to discover one of the following types of points (see Figure 3.14):

(a) a point z on e such that there is a derived edge e_1 defined by (e', e) in $\mathcal{E}_O$, where e' is an edge of s and $z = \rho_{e_1} \cap e$.

(b) a vertex z of e,

(c) a point z on e at which e is hidden by a closer triangle edge e', or

(d) a point z on e at which e hides an edge e' of another triangle lying farther away from p than z.

In the first two cases, we stop. In each of the last two cases, we find a new edge of $\mathcal{E}_{O'}$, and we repeat the ray-dragging query starting at z and moving along e' in the direction that makes it visible (this direction is easy to compute). In this manner, we find all the new edges in $\mathcal{E}_{O'}$ created by the deletion of s.

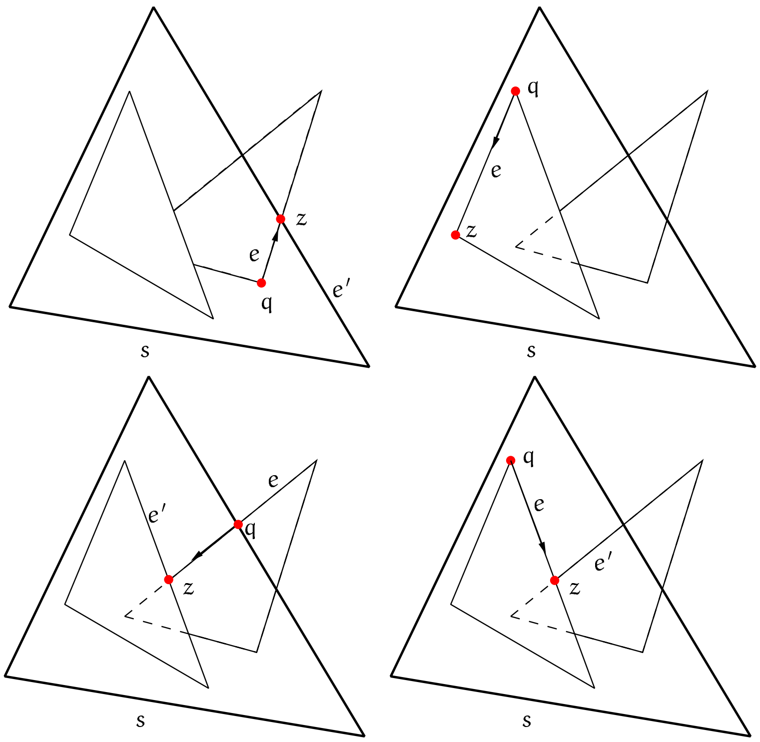


Figure 3.14: Different cases that arise when discovering new edges of $\mathcal{E}_{O'}$. The viewpoint is at $z = +\infty$ and the image plane is the xy-plane.

3. We compute the visibility certificates for $\mathcal{R}_v$ as follows:

Vertex certificates: For each vertex q of $\mathcal{R}_v$, we add q as a vertex certificate of $\mathcal{R}_v$ if q is visible. Note that we update the visibility status of q to reflect the deletion of the triangles in O_v in Step 2.

Edge certificates: We compute edge certificates as outlined under the corresponding step for adding a node to $\mathcal{M}$.

Shadow certificates: For each edge $e \in \mathcal{E}_O$ that intersects the interior of $\mathcal{R}_v$, we add e as a shadow certificate. Note that we update the set of such edges in Step 2.

3.3.6 Extensions

In this section, we show how to modify the algorithm described so far to handle moving triangles and discuss some extensions to our algorithm. We handle a moving triangle $s \in S$ in two different ways:

1. If $s \notin O$ (this may happen if s is small or far away from the viewpoint), we use its motion to track the cells in $\mathcal{C}$ that s intersects. The details of the kinetic data structure used to maintain the cells that s intersects are easy to develop. We render s if it intersects a cell in $\mathcal{V}_O$.

2. If $s \in O$ (this may happen in an architectural environment if s is in the same room as the viewpoint), we have to modify the BSP $\mathcal{B}$ as s moves. To this end, we use the recently-developed technique of Agarwal et al. [2] for efficiently maintaining BSPs for moving triangles in $\mathbb{R}^3$. Their algorithm is an extension to $\mathbb{R}^3$ of a kinetic data structure we present in Chapter 6.3 for maintaining a BSP for moving segments in the plane.

We now discuss how we can incorporate frame-rate control into our algorithm. Recall that in our algorithm for marking nodes (see Section 3.3.5), we use a parameter κ to decide which nodes of $\mathcal{B}$ to mark. If κ is sufficiently small, we mark nodes near the root of $\mathcal{B}$. As a result, we select a very small subset of the triangles in S as occluders. Therefore, the time taken by our algorithm to process its internal data structures is very small, but our occlusion culling is not very efficient, and we output most triangles in S to the graphics system, which consumes a lot of time to render these triangles. On the other hand, if κ is sufficiently large, we mark most of the leaves in $\mathcal{B}$, thus picking most triangles in S as occluders. Our kinetic data structure is more time-consuming but we send very few invisible triangles to the graphics system. Between these two extremes, there is a median that balances the time spent by our algorithm in internal processing and the time spent by the graphics system in rendering.

In future work, we plan to examine various schemes for choosing the value of κ dynamically to achieve this balance. One possibility is to modify κ based on the relation between internal processing time and rendering time. We are also considering more sophisticated techniques that estimate whether it is better to keep a node in $\mathcal{B}$ marked or to unmark it and mark its two children instead. If we mark the children, we increase the size of $\mathcal{O}$ and the processing time but we may also decrease the number of triangles sent to the graphics system, thus reducing the rendering time.

Our eventual goal is to merge these ideas for frame-rate control with simplification and image-based rendering. We expect to exploit some of the unique properties of the BSP to successfully integrate all these techniques.

3.4 Conclusions

In this chapter, we have presented an object complexity algorithm for hidden-surface removal that computes the exact set of visible triangles and renders them using the z-buffer. The algorithm can be tuned to compute a small superset of the visible triangles. While the algorithm is based on well-known principles (if a cell is invisible, then all triangles intersecting the cell are also invisible), we use several novel algorithmic ideas to perform hidden-surface removal efficiently:

- maintaining the union of the triangle shadows exactly,
- using kinetic data structures to efficiently maintain the set of visible cells, and
- employing the BSP as an underlying data structure for supporting all our operations.

We are currently implementing this algorithm and using BSPs constructed by techniques to be presented in the rest of this thesis.

We now discuss some of the important questions that are suggested by our algorithm. A major drawback of our algorithm is that the motion of the viewpoint is involved in every certificate used in the algorithm. Therefore, when the viewpoint's motion changes, we have to recalculate *every* certificate, which is very costly. In fact, this point brings out a drawback of kinetic data structures: they are not ideal for processing moving objects whose motions are not independent. One promising avenue that we are studying in order to solve this problem is to avoid calculating the expiry time of a certificate exactly; the farther the expiration is in the future, the less accurately we compute it. As a result, when the viewpoint's motion changes, we will expend our effort in recalculating expiration times for a small number of certificates. Interesting issues arise now regarding what laws the viewpoint motion should follow so that we can still ensure the correctness of our algorithm.

Our algorithm also raises several other theoretical issues:

1. Under what conditions can we guarantee a bound on the size of the superset of visible triangles output by our algorithm?

2. Can we prove bounds on the running time of the ray-dragging procedure? This question is related to the number of cells of $\mathcal{B}$ stabbed by a moving ray or segment. It is possible that examining this problem under a suitable model of geometric complexity will yield practically-useful solutions.

3. It will also be interesting to extend our algorithm to the situation when the triangles in S reside on disk and we need to minimise the number of input/output operations to disk. One possible solution is to have two occlusion culling algorithms running in parallel. One algorithm runs on the triangles in memory and sends visible triangles to the z-buffer. The other algorithm processes the triangles on disk and stores visible triangles in memory. Thus the second algorithm ensures that all visible triangles are in memory at any given time.

We expect our algorithm to perform efficiently in practice, especially when we incorporate our ideas for frame-rate control and simplification. The answers to the open problems we have posed are likely to further strengthen our techniques and make them applicable to a large class of inputs.

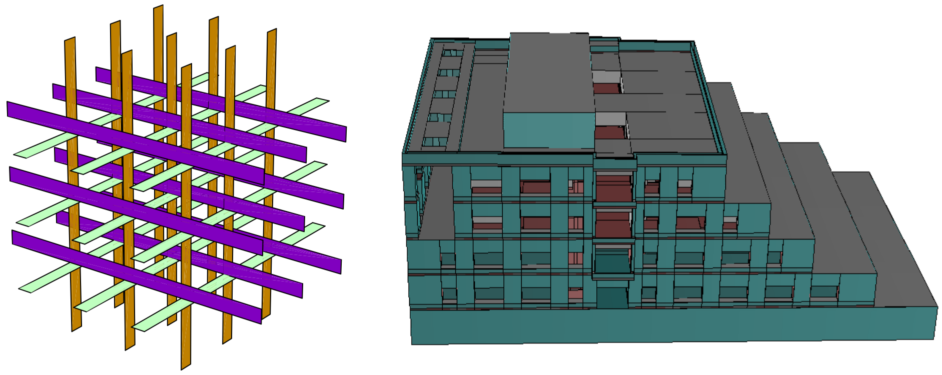
Chapter 4

Binary Space Partitions for Fat Rectangles

In the last two chapters, we presented our algorithms for model repair and hidden-surface removal. Since both algorithms use the BSP as an underlying data structure, their running time inherently depends on properties of the BSP such as size and height. In fact, the same is true of algorithms that use the BSP in a variety of different applications: hidden-surface removal itself [8, 101], global illumination [23], shadow generation [29, 30], solid modelling [75, 78, 102], ray tracing [74], robotics [12], and approximation algorithms for network design [66] and surface simplification [7].

As a result, there has been a lot of effort to construct BSPs of small size. While several simple heuristics have been developed for constructing BSPs of reasonable sizes [8, 24, 47, 73, 101, 102], provable bounds were first obtained by Paterson and Yao. They show that a BSP of size $O(n \log n)$ can be constructed for n disjoint segments in $\mathbb{R}^2$; they also show that a BSP of size $O(n^2)$ can be constructed for n disjoint triangles in $\mathbb{R}^3$, which is optimal in the worst case [81]. But in graphics-related applications, many common environments like buildings are composed largely of orthogonal rectangles, and non-orthogonal triangles are approximated by their orthogonal bounding boxes [44]. Paterson and Yao [82] prove that a BSP of size $O(n)$ exists for n non-intersecting, orthogonal segments in $\mathbb{R}^2$ and of size $O(n\sqrt{n})$ for n non-intersecting, orthogonal rectangles in $\mathbb{R}^3$. Both bounds are optimal in the worst case.

In all known lower bound examples of orthogonal rectangles in $\mathbb{R}^3$ requiring BSPs



(a) (b)

Figure 4.1: (a) Lower bound construction for orthogonal rectangles. (b) Model of Soda Hall—85% of the rectangles have aspect ratio at most 25.

of size $\Omega(n\sqrt{n})$, most of the rectangles are "thin." For example, Paterson and Yao's lower bound proof uses a configuration of $\Theta(n)$ orthogonal rectangles, arranged in a $\sqrt{n} \times \sqrt{n} \times \sqrt{n}$ grid, for which any BSP has size $\Omega(n\sqrt{n})$ (see Figure 4.1(a)). All rectangles in their construction have aspect ratio $\Omega(\sqrt{n})$. Such configurations of thin rectangles rarely occur in practice. Many real databases consist mainly of "fat" rectangles, i.e., the aspect ratios of these rectangles are bounded by a constant. An examination of four data sets containing a few thousand rectangles—the Sitterson Hall, the Orange United Methodist Church Fellowship Hall, and the Sitterson Hall Lobby databases from the University of North Carolina at Chapel Hill and the model of Soda Hall from the University of California at Berkeley (shown in Figure 4.1(b))—shows that most of the rectangles in these models have aspect ratio less than 30.

It is natural to ask whether BSPs of near-linear size can be constructed if most of the rectangles are "fat." We call a rectangle *fat* if its aspect ratio (the ratio of the longer side to the shorter side) is bounded by a fixed constant; for specificity, we use $\alpha \geq 1$ to denote this constant. A rectangle is said to be *thin* if its aspect ratio is greater than α. In this paper, we consider the following problem:

Given a set S of n non-intersecting, orthogonal, two-dimensional rectangles *in $\mathbb{R}^3$, of which m are thin and the remaining $n-m$ are fat, construct a BSP for S.*

We first show how to construct a BSP of size $n2^{O(\sqrt{\log n})}$ for n fat rectangles in $\mathbb{R}^3$ (i.e., when $m=0$). We then show that if $m>0$, a BSP of size $n\sqrt{m}2^{O(\sqrt{\log n})}$ can be built. We also prove a lower bound of $\Omega(n\sqrt{m})$ on the size of such a BSP.

We finally prove two important extensions to these results. If p of the n input objects are non-orthogonal, we show that an $np2^{O(\sqrt{\log n})}$-size BSP exists. Unlike in the case of orthogonal objects, fatness does not help in reducing the worst-case size of BSPs for non-orthogonal objects. In particular, we prove that there exists a set of n fat triangles in $\mathbb{R}^3$ for which any BSP has $\Omega(n^2)$ size. However, non-orthogonal objects can be approximated by orthogonal bounding boxes. The resulting bounding boxes might intersect each other. Motivated by this observation, we also consider the problem in which n fat rectangles contain k intersecting pairs of rectangles, and we show that we can construct a BSP of size $(n+k)\sqrt{k}2^{O(\sqrt{\log n})}$. We also prove a lower bound of $\Omega(n+k\sqrt{k})$ on the size of such a BSP.

In all cases, the constant of proportionality in the big-oh terms is linear in $\log \alpha$, where α is the maximum aspect ratio of the fat rectangles. Our algorithms to construct these BSPs run in time proportional to the size of the BSPs they build. In Chapter 5, we describe experiments that demonstrate that our algorithms work well in practice and construct BSPs of near-linear size when most of the rectangles are fat, and perform better than most known algorithms for constructing BSPs for orthogonal rectangles.

As far as we are aware, ours is the first work to consider BSPs for the practical and common case of (two-dimensional) fat polygons in $\mathbb{R}^3$. de Berg considers a weaker model, the case of (three-dimensional) fat polyhedra in $\mathbb{R}^3$ (a polyhedron is said to be *fat* if its volume is at least a constant fraction of the volume of the smallest sphere enclosing it), although his results extend to higher dimensions [35].

One of the main ingredients of our algorithm is the construction of an $O(n \log n)$-size BSP for a set of n fat rectangles that are "long" with respect to a box B, i.e., none of the vertices of the rectangles lie in the interior of B. We present this algorithm in Section 4.2. To prove this result, we crucially use the fatness of the rectangles. In Section 4.3, we use this procedure to construct a BSP of size $O(n^{4/3})$ for fat rectangles.

The algorithm repeatedly applies cuts that bisect the set of vertices of rectangles in S until all sub-problems have long rectangles and the total size of the sub-problems is $O(n^{4/3})$, at which point we can invoke the algorithm for long rectangles. We improve the size of the BSP to $n2^{O(\sqrt{\log n})}$ by simultaneously simulating the algorithm for long rectangles and partitioning the vertices of rectangles in S in a clever manner; we describe the algorithm in Section 4.4 and analyse it in Section 4.5. Finally, we extend this result in Section 4.6 to construct BSPs in cases when some triangles in the input are (i) thin, (ii) non-orthogonal, or (iii) intersecting. We conclude in Section 4.7 with some open problems.

4.1 Geometric Preliminaries

Before we describe our algorithm, we recapitulate the definition of the BSP and give some other preliminary definitions. In our case, S is a set of orthogonal rectangles with pairwise-disjoint interiors in $\mathbb{R}^3$. A *binary space partition* $\mathcal{B}$ for S is a tree defined as follows: Each node v in $\mathcal{B}$ is associated with a polytope $\mathcal{R}_v$ and the set of rectangles $S_v = \{s \cap \mathcal{R}_v \mid s \in S\}$ that intersect $\mathcal{R}_v$. The polytope associated with the root is $\mathbb{R}^3$ itself. If S_v is empty, then node v is a leaf of $\mathcal{B}$. Otherwise, we partition $\mathcal{R}_v$ into two convex polytopes by a *cutting plane* H_v. We refer to the polygon $H_v \cap \mathcal{R}_v$ as the *cut* made at v. At v, we store the equation of H_v and the set $\{s \mid s \subseteq H_v, s \in S_v\}$, the subset of rectangles in S_v that lie in H_v. If H_v^+ denotes the positive halfspace and H_v^- the negative halfspace bounded by H_v, the polytopes associated with the left and right children of v are $\mathcal{R}_v \cap H_v^-$ and $\mathcal{R}_v \cap H_v^+$, respectively. The left subtree of v is a BSP for the set of rectangles $S_v^- = \{s \cap H_v^- \mid s \in S_v\}$ and the right subtree of v is a BSP for the set of rectangles $S_v^+ = \{s \cap H_v^+ \mid s \in S_v\}$. The size of $\mathcal{B}$ is the sum of the number of nodes in $\mathcal{B}$ and the total number of rectangles stored at all the nodes in $\mathcal{B}$.

In our algorithms, each cutting plane will be orthogonal. Therefore, the region $\mathcal{R}_v$ associated with each node v in $\mathcal{B}$ is a *box* (rectangular parallelepiped). We say that a rectangle r is *long* with respect to a box B if none of the vertices of r lie in the interior of B. Otherwise, r is said to be *short*. See Figure 4.2. A long rectangle is said to be *free* if none of its edges lies in the interior of B; otherwise it is *non-free*. A *free cut* is a cutting plane that does not cross any rectangle in S and that either divides S into

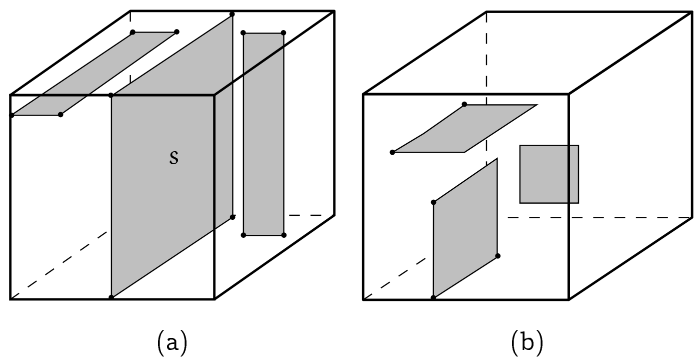


Figure 4.2: (a) Long and (b) short rectangles. Heavy dots indicate the vertices of these rectangles that lie on the boundary of the box. Rectangle s is a free rectangle.

two non-empty sets or contains a rectangle in S. Note that the plane containing a free rectangle is a free cut.

We will often focus on a box B and construct a BSP for the rectangles intersecting it. Given a set of rectangles R, let

$$R_B = \{s \cap B \mid s \in R\}$$

be the set of rectangles obtained by clipping the rectangles in R within B. For a set of points P, let P_B be the subset of P lying in the interior of B.

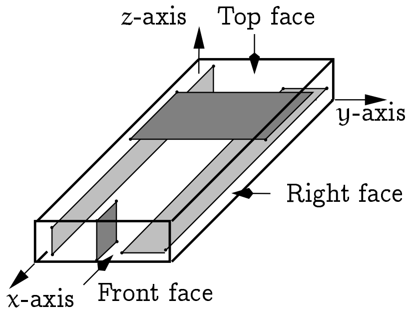


Figure 4.3: Different classes of rectangles.

A box B has six faces—*top, bottom, front, back, right,* and *left*, as shown in Figure 4.3. We assume, without loss of generality, that the back, bottom, left corner of B is the origin (i.e., the back face of B lies on the yz-plane). A rectangle s that is long with respect to B belongs to the *top class* if two parallel edges of $s \cap B$ are contained in the top and bottom faces of B. We similarly define the *front* and *right* classes. A long rectangle belongs to at least one of these three classes; a non-free rectangle belongs to a unique class. See Figure 4.3 for examples of rectangles belonging to different classes.

Although a BSP is a tree, we will often just discuss how to partition the box represented by a node into two boxes. We will not explicitly detail the associated construction of the actual tree itself, since the construction is straightforward once we specify the cutting plane. Sometimes, we will abuse notation and use B to also refer to the corresponding node in the BSP.

In the rest of this section, we assume that the vertices of the rectangles in S are sorted by x-, y-, and z-coordinate, and that for each axis, the rectangles perpendicular to that axis are sorted by intercept. The cost of this sort will not affect the asymptotic running times of our algorithms.

We now state two preliminary lemmas that we will use below. The first lemma characterises a set of rectangles that are long with respect to a box and belong to one class. The second lemma applies to two classes of long rectangles.

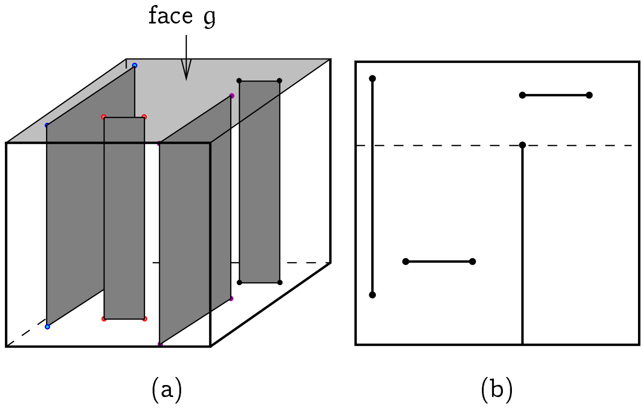


Figure 4.4: (a) Long rectangles in the top class. (b) Projections of the rectangles in (a) onto the top face g; heavy dots indicate the vertices of these rectangles that lie in the interior of g. The dashed line is the cut satisfying (4.4.1).

Lemma 4.1.1 *Let C be a box, P a set of points in the interior of C, R a set of rectangles long with respect to C, and $w \geq 1$ a real number. If the rectangles in*

R_C *belong to one class, then the following two conditions hold (see Figure 4.4):*

(i) There exists a face g *of the box* C *that contains exactly one of the edges of each rectangle in* R_C*. Let* V *be the set of those vertices of the rectangles in* R_C *that lie in the interior of* g*.*

(ii) We can find a plane that partitions C *into two boxes* C_1 *and* C_2 *so that for* $i = 1, 2$*,*

$$|V \cap C_i| + w|P_{C_i}| \leq \frac{|V| + w|P|}{2}. \tag{4.4.1}$$

If the rectangles in R_C *and the points in* P *are sorted along each of the three axes, the partitioning plane can be computed in* $O(|R_C| + |P|)$ *time.*

Proof: (i) follows from the definition of a class. To prove part (ii) of the lemma, let P^* be the set of projections of the points in P onto g. Assume g is the top face of C. If we associate a weight of 1 with each point in V and a weight w with each point in P^*, the total weight of the points in $V \cup P^*$ is $|V| + w|P|$. By sweeping g, we can find in $O(|V| + |P|)$ time a line ℓ lying in g and parallel to the x-axis that contains a point in $V \cup P^*$ and that divides $V \cup P^*$ into two sets, each with weight at most $(|V| + w|P|)/2$. We split C into two boxes C_1 and C_2 by drawing the plane containing ℓ that is orthogonal to g. By construction, C_1 and C_2 satisfy (4.4.1). The time bound follows easily. □

Lemma 4.1.2 *Let* C *be a box,* P *a set of points in the interior of* C*,* R *a set of rectangles long with respect to* C*, and* $w \geq 1$ *a real number. If the rectangles in* R_C *belong to two classes, then one of the following two conditions holds (see Figure 4.5):*

(i) We can find one free cut that partitions C *into two boxes* C_1 *and* C_2 *so that*

$$|R_{C_i}| + w|P_{C_i}| \leq \frac{2\,(|R_C| + w|P|)}{3} \tag{4.4.2}$$

for $i = 1, 2$*.*

(ii) We can find two parallel free cuts that divide C *into three boxes* C_1, C_2*, and* C_3 *such that all rectangles in* R_{C_2} *belong to the same class and such that*

$$|R_{C_2}| + w|P_{C_2}| \geq \frac{|R_C| + w|P|}{3}. \tag{4.4.3}$$

If the rectangles in R_C and the points in P are sorted along each of the three axes, these free cuts can be computed in $O(|R_C| + |P|)$ time.

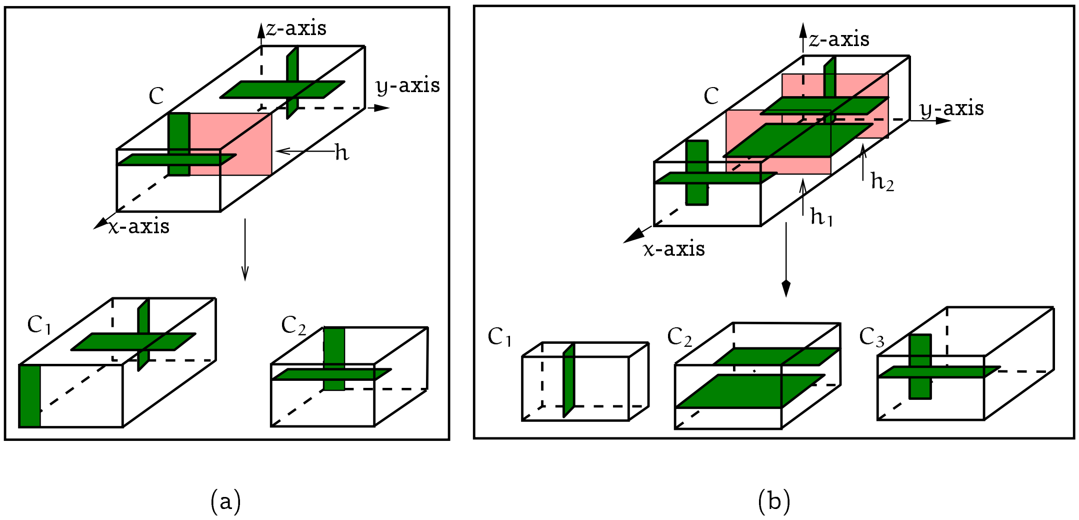


Figure 4.5: (a) Free cut h partitions box C into two boxes C_1 and C_2. (b) Two parallel free cuts h_1 and h_2 partition C into three boxes C_1, C_2, and C_3.

Proof: We assume without loss of generality that the rectangles in R_C belong to the top and right classes. Let $\bar{r}$ denote the projection of a rectangle $r \in R_C$ onto the x-axis: $\bar{r}$ is either a point or an interval. Similarly, let $\bar{p}$ denote the projection of a point $p \in P$ onto the x-axis. Set

$$U = \left\{ \left(\bigcup_{r \in R_C} \bar{r} \right) \cup \left(\bigcup_{p \in P} \bar{p} \right) \right\}.$$

U is a set of disjoint intervals, some of which may be single points. Let r_1 (resp., r_2) be a rectangle in R_C belonging to the top (resp., right) class. Since the rectangles in R_C are disjoint, it is easily seen that $\bar{r_1}$ and $\bar{r_2}$ are also disjoint. Hence, each connected component of U contains the projections of rectangles belonging to at most one class. For any connected component I of U define,

$$\mu(I) = |\{r \in R_C \mid \bar{r} \subseteq I\}| + w|\{p \in P \mid \bar{p} \subseteq I\}|.$$

Set $W = |R_C| + w|P|$. If U contains a connected component $I = [\beta, \gamma]$ with $\mu(I) > W/3$, then the two free cuts are $x = \beta$ and $x = \gamma$. The cuts partition the box C into three boxes C_1, C_2, and C_3, where C_2 denotes the middle box. By construction, all rectangles in R_{C_2} belong to at most one class. Hence, condition (ii) holds.[1]

If there is no such connected component of U, then let $I = [\beta, \gamma]$ be the leftmost connected component of U with $\sum_{I' \leq I} \mu(I') > W/3$ (we say that $I' \leq I$ if I' lies to the left of I). Since $\mu(I) \leq W/3$ and $\sum_{I' < I} \mu(I') < W/3$,

$$\sum_{I' \leq I} \mu(I') \leq 2W/3.$$

We partition C into two boxes C_1 and C_2 using the cut $x = \gamma$. In this case, condition (i) holds.

If the rectangles in R_C and the points in P are sorted along the x-axis, it is clear that the components in U can be computed and sorted in $O(|R_C| + |P|)$ time. The free cut(s) used to partition C can be found in the same time by sweeping the components of U. □

4.2 BSPs for Long Fat Rectangles

Assume that all the rectangles in S are long with respect to a box B. In this section, we show how to build a BSP for S_B, the set of rectangles clipped within B. This algorithm will form the basis of our algorithm for the general case, when S_B contains both long and short rectangles. In general, S_B can have all three classes of rectangles. We first exploit the fatness of the rectangles in S to prove that whenever all three classes are present in S_B, a small number of cuts can divide B into boxes each of which has only two classes of rectangles. Then we describe an algorithm that constructs a BSP when all the rectangles belong to only two classes.

4.2.1 Reducing three classes to two classes

Assume, without loss of generality, that the longest edge of B is parallel to the x-axis. The rectangles in S_B that belong to the front class can be partitioned into two subsets: the set R of rectangles that are vertical (and parallel to the right face of B) and the set T of rectangles that are horizontal (and parallel to the top face of B). See Figure 4.6(a).

[1] If $\beta = \gamma$, i.e. I is a point, then C_2 is regarded as a degenerate box. If I is the first (resp., last) connected component of U, then C_1 (resp., C_3) may be a degenerate box.

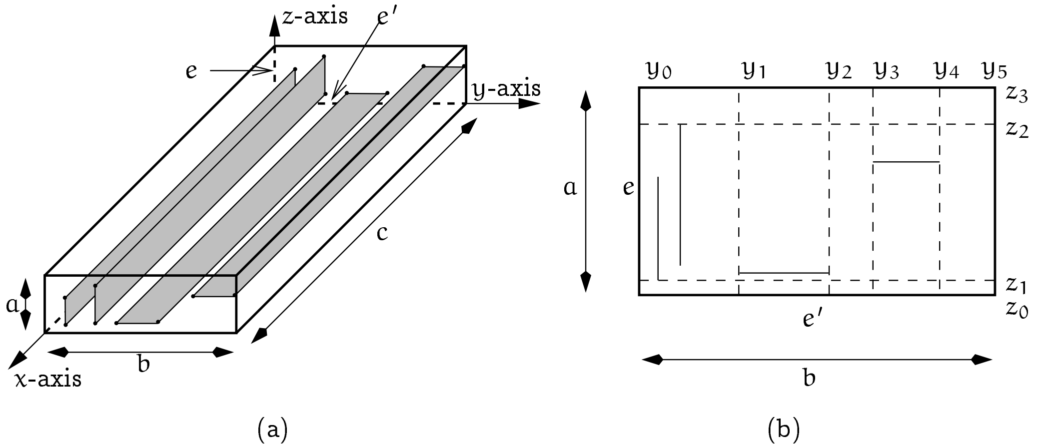


Figure 4.6: (a) Rectangles belonging to the sets R and T. (b) The back face of B; dashed lines are intersections of the back face with the α-cuts.

Let e be the edge of B that lies on the z-axis and let e' be the edge of B that lies on the y-axis. The intersection of each rectangle in R with the back face of B is a segment parallel to the z-axis. Let $\bar{r}$ denote the projection of such a segment r onto the z-axis, and let $\bar{R} = \{\bar{r} \mid r \in R\}$. Let $z_1 < z_2 < \cdots < z_{k-1}$ be the endpoints of intervals in $\bar{R}$ that lie in the interior of e but not in the interior of any interval of $\bar{R}$. Note that $k-1$ may be less than $2|R|$, as in Figure 4.6(b), if some of the projected segments overlap. If no two intervals in $\bar{R}$ share an endpoint, then $\{z_1, z_2, \ldots, z_{k-1}\}$ is the set of vertices of the union of the intervals in $\bar{R}$; otherwise, $\{z_1, z_2, \ldots, z_{k-1}\}$ includes endpoints common to two intervals in $\bar{R}$ and not lying in the interior of any other interval in $\bar{R}$. Similarly, for each rectangle t in the set T, let $\bar{t}$ be the projection of t onto the y-axis, and let $\bar{T} = \{\bar{t} \mid t \in T\}$. Let $y_1 < y_2 < \cdots < y_{l-1}$ be the endpoints of intervals in $\bar{T}$ that lie in the interior of e' but not in the interior of any interval of $\bar{T}$.

We divide B into kl boxes by drawing the planes $z = z_i$ for $1 \le i < k$ and the planes $y = y_j$ for $1 \le j < l$. See Figure 4.6(b). This decomposition of B into kl boxes can easily be constructed in a tree-like fashion by performing $(k-1)(l-1)$ cuts. We refer to these cuts as *α-cuts*. If any resulting box has a free rectangle, we divide that box into two boxes by applying the free cut along the free rectangle. Let $\mathcal{C}$ be the set of boxes into which B is partitioned in this manner. We can prove the following lemma about the decomposition of B into $\mathcal{C}$.

Lemma 4.2.1 *The set $\mathcal{C}$ of boxes formed by the above process satisfies the following properties:*

(*i*) *Each box C in $\mathcal{C}$ has only two classes of rectangles;*

(*ii*) *there are at most $26\lfloor\alpha\rfloor^2 n$ boxes in $\mathcal{C}$; and*

(*iii*) $\sum_{C\in\mathcal{C}} |S_C| \leq 16\lfloor\alpha\rfloor n$.

Proof: Let z_0 and z_k, where $z_0 < z_k$, be the endpoints of e, the edge of the box B that lies on the z-axis. Similarly, define y_0 and y_l, where $y_0 < y_l$, to be the endpoints of the edge of B that lies on the y-axis.

(i) Let C be a box in $\mathcal{C}$. If C does not contain a rectangle from $T \cup R$, the claim is obvious since the rectangles in T and R together constitute the front class. Suppose C contains rectangles from the set R. Rectangles in R belong to the front class and are parallel to the right face of B. We claim that C cannot have any rectangles from the right class. Indeed, consider an edge of C parallel to the z-axis. The endpoints of this edge have z-coordinates z_i and z_{i+1}, for some $0 \leq i < k$. Since C contains a rectangle from R, by construction, the interval $z_i z_{i+1}$ must be covered by projections of rectangles in R (onto the z-axis). If C also contains a rectangle r belonging to the right class, then let $z_i < z < z_{i+1}$ be the z-coordinate of a point in $r \cap C$. Let r' be a rectangle in R whose projection on the z-axis contains z. Since both r and r' are long with respect to C, r and r' intersect, which contradicts the fact that the rectangles in S are non-intersecting. A similar proof shows that if C contains rectangles from T, then C does not contain any rectangle in the top class.

(ii) We first show that both k and l are at most $2\lfloor\alpha\rfloor + 3$. Let a (respectively, b, c) denote the length of the edges of B parallel to the z-axis (respectively, y-axis, x-axis). By assumption, $a, b \leq c$. Let $r \in R$ be a rectangle with dimensions β and γ, where $\beta \leq \gamma$. Consider $\bar{r}$, the projection of r onto the z-axis. Suppose that $\bar{r} \subseteq z_i z_{i+1}$, for some $0 < i < k-1$, i.e., $\bar{r}$ lies in the interior of the edge e of B lying on the z-axis. Since r is a rectangle in the front class and is parallel to the right face of B, we

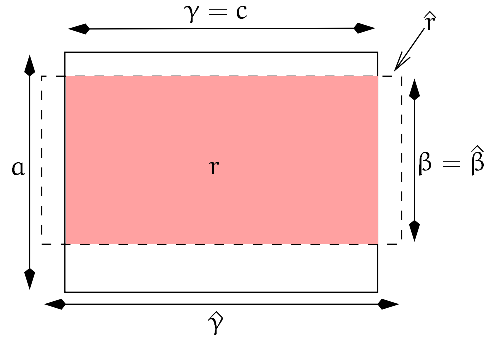


Figure 4.7: Projections of $\hat{r}$ (the dashed rectangle), $r = \hat{r} \cap B$ (the shaded rectangle), and the right face of B onto the zx-plane.

have $\beta \le a \le c = \gamma$. If $\hat{r}$, the rectangle supporting r in the set $S_,$ has dimensions $\hat{\beta}$ and $\hat{\gamma}$, where $\hat{\beta} \le \hat{\gamma} \le \alpha\hat{\beta}$, we have $\beta = \hat{\beta}$ (since $\bar{r} \subseteq \mathrm{int}(e)$) and $\gamma \le \hat{\gamma}$. (If i is 0 or $k-1$, we cannot claim that $\beta = \hat{\beta}$; in these cases, it is possible that $\beta \ll \hat{\beta}$.) See Figure 4.7. Thus, we obtain

$$a \le c = \gamma \le \hat{\gamma} \le \alpha\hat{\beta} = \alpha\beta.$$

It follows that the length of the interval $\bar{r}$, and hence the length of $z_i z_{i+1}$, is at least a/α. Since every alternate interval $z_i z_{i+1}, 0 < i < k-1$ contains the projection of at least one rectangle of R, $k \le 2\lfloor\alpha\rfloor + 3$. In a similar manner, $l \le 2\lfloor\alpha\rfloor + 3$.

Hence, the planes $z = z_i, 1 \le i \le k-1$ and the planes $y = y_j, 1 \le j \le l-1$ partition B into at most $kl \le (2\lfloor\alpha\rfloor + 3)^2$ boxes. Each such box C can contain at most n rectangles. Hence, at most n free cuts can be made inside C. The free cuts can divide C into at most $n+1$ boxes. This implies that the set $\mathcal{C}$ has at most $kl(n+1) \le 26\lfloor\alpha\rfloor^2 n$ boxes.

(iii) Each rectangle r in S_B is cut into at most kl pieces. The edges of these pieces form an arrangement on r. Each face of the arrangement is one of the at most kl rectangles that r is partitioned into. Only $2(k+l-2)$ faces of the arrangement have an edge on the boundary of r. All other faces can be used as free cuts. Hence, after all possible free cuts are made in the boxes into which B is divided by the $(k-1)(l-1)$ cuts, only $2(k+l-2)$ pieces of each rectangle in S_B survive. This proves that $\sum_{C \in \mathcal{C}} |S_C| \le 16\lfloor\alpha\rfloor n$. $\square$

Remarks: The only place in the whole algorithm where we use the fatness of the rectangles in S is in the proof of Lemma 4.2.1. If the rectangles in S are thin, then Lemma 4.2.1(ii) is not true; both k and l can be $\Omega(n)$.

If S_B contains a short rectangle, the α-cuts partition the short rectangle into a constant number of pieces. Hence, Lemma 4.2.1(ii) and (iii) hold even when S_B contains short rectangles.

4.2.2 BSPs for two classes of long rectangles

Let C be one of the boxes into which B is partitioned by the α-cuts. We now present an algorithm for constructing a BSP for the set of clipped rectangles S_C, which has only two classes of long rectangles. We recursively apply the following steps to each of the boxes produced by the algorithm until no box contains a rectangle.

1. If S_C has a free rectangle, we use the free cut containing that rectangle to split C into two boxes.

2. If S_C has two classes of rectangles, we use Lemma 4.1.2 (with $R = S$ and $P = \emptyset$) to split C into at most three boxes, using at most two parallel free cuts.

3. If S_C has only one class of rectangles, we split C into two by a plane, using Lemma 4.1.1 (with $R = S$ and $P = \emptyset$).

We first analyse the algorithm for two classes of long rectangles. The BSP produced has the following structure: If Step 3 is executed at a node v, then Step 2 is not invoked at any descendant of v. Note that the cutting planes used in Step 1 and 2 do not intersect any rectangle of S_C, so only the cuts made in Step 3 increase the number of rectangles. Hence, repeated execution of Steps 1 or 2 on S_C constructs a top subtree $\mathcal{T}_C$ of the BSP with $O(|S_C|)$ nodes such that each leaf in $\mathcal{T}_C$ has only one class of rectangles and the total number of rectangles in all the leaves is at most $|S_C|$. The operations at a box D in $\mathcal{T}_C$ involve determining the cuts to be made at D, partitioning D according to these cuts, identifying the resulting free rectangles and applying free cuts containing them, and partitioning the rectangles in S_D into the new boxes. Since we assume that we have sorted the vertices of the rectangles in S at the very beginning, Lemmas 4.1.1

and 4.1.2 imply that the cuts to be made at D can be determined in $O(|S_D|)$ time. The free cuts resulting after partitioning D according to these cuts are parallel to each other. Hence, all free cuts can be applied in $O(|S_D|)$ time. Further, the number of rectangles at each child of D is at most $2|S_D|/3$. Hence, $\mathcal{T}_C$ can be constructed in $O(|S_C| \log |S_C|)$ time. At each leaf ν of the tree $\mathcal{T}_C$, recursive invocations of Steps 1 and 3 build a BSP of size $O(|S_\nu| \log |S_\nu|)$ in $O(|S_\nu| \log |S_\nu|)$ time (see [81] for details). Since $\sum_\nu S_\nu \leq |S_C|$, where the sum is taken over all leaves ν of $\mathcal{T}_C$, the total size of the BSP constructed inside C is $O(|S_C| \log |S_C|)$. It also follows that the BSP inside C can be constructed in $O(|S_C| \log |S_C|)$ time.

We now analyse the overall algorithm for long rectangles. The algorithm first applies the α-cuts to the rectangles in S_B, as described in Section 4.2.1. Consider the set of boxes $\mathcal{C}$ produced by the α-cuts. Each of the boxes in $\mathcal{C}$ contains only two classes of rectangles (by Lemma 4.2.1(i)). In view of the above discussion, for each box $C \in \mathcal{C}$, we can construct a BSP for S_C of size $O(|S_C| \log |S_C|)$ in time $O(|S_C| \log |S_C|)$. Lemma 4.2.1(ii) and 4.2.1(iii) imply that the total size of the BSP is $O(n) + \sum_{C \in \mathcal{C}} O(|S_C| \log |S_C|) = O(n \log n)$. The time spent in building the BSP is also $O(n \log n)$. We can now state the following theorem:

Theorem 4.2.2 *Let S be a set of n fat rectangles and B a box so that all rectangles in S are long with respect to B. Then an $O(n \log n)$-size BSP for the clipped rectangles S_B can be constructed in $O(n \log n)$ time. The constants of proportionality in the big-oh terms are linear in α^2, where α is the maximum aspect ratio of the rectangles in S.*

Remark: We can show that the height of the BSP constructed by the above logarithm is $O(\log n)$. We can also modify our algorithm to construct a BSP of size $O(n)$ for n long rectangles as follows: If a box C has two classes of long rectangles, we apply Step 1 or 2 of the previous algorithm. If the rectangles in C belong to one class, we use the algorithm of Paterson and Yao for constructing BSPs for orthogonal segments in the plane [82] to construct a BSP of linear size for S_C. However, the height of the BSP can now be $\Omega(n)$ in the worst case. We will not need this improved result, except in Section 4.3.

4.3 BSPs of Size $O(n^{4/3})$

In this section, we present a simple algorithm that constructs a BSP of size $O(n^{4/3})$ for n fat rectangles. We need a definition before describing the algorithm. A *bisecting cut* is an orthogonal cut that partitions those vertices of the rectangles in S that lie in the interior of B into two halves, i.e., if k vertices lie in the interior of B, then a bisecting cut divides B into two boxes, each containing at most $\lfloor k/2 \rfloor$ vertices in its interior.

The algorithm proceeds in phases. A *phase* is a sequence of three bisecting cuts, with exactly one cut perpendicular to each of the three orthogonal directions. After each phase, if a box contains a free rectangle, we use the corresponding free cut to further divide the box into two. We begin the first phase with a box enclosing all the rectangles, with at most $4n$ vertices in its interior, and continue executing phases of bisecting cuts until each box has no vertex in its interior. At this point, each box contains only long rectangles. We then invoke the algorithm described in the remark following Theorem 4.2.2 to construct a BSP in each of these boxes.

The crux of the analysis of the size of the BSP produced by this algorithm is counting the number of pieces into which one rectangle is split when subjected to a specified number of phases. To this effect, we use the following result of Paterson and Yao [82]. We include the proof here for the sake of completeness.

Lemma 4.3.1 (Paterson-Yao) *A rectangle that has been subjected to d phases of cuts (with all available free cuts used at the end of each phase) is divided into $O(2^d)$ rectangles.*

Proof: A phase of cuts divides a rectangle r into at most four sub-rectangles r_i, $1 \le i \le 4$. See Figure 4.8. Suppose g edges of r, for $g \le 4$, lie on the boundary of the rectangle in S that supports r; each such edge lies in the interior of B. If $g = 0$, then r is a free rectangle. Let g_i be the corresponding number for r_i. If $\chi(g, d)$ is the number of rectangles into which r is divided by d phases of cuts, we have the following recurrence for $\chi(g, d)$, where $g, d > 0$:

$$\chi(g, d) \le \sum_{i=1}^{4} \chi(g_i, d-1), \qquad \text{where } \sum_{i=1}^{4} g_i \le 2g,$$

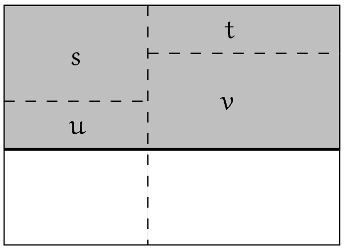


Figure 4.8: The shaded region is a rectangle r with one edge lying in the interior of B (r is shown projected onto a face of B that r is parallel to). A phase divides r into four pieces s, t, u, and v; s and t are free rectangles.

because each edge of r is split into at most two edges by a phase of cuts (by the cutting plane orthogonal to the edge). We see that $\chi(g, 0) = 1$ and $\chi(0, d) = 1$, because a free rectangle at the beginning of a phase will be removed by a free cut. It is easily checked that the solution to this recurrence is $3g(2^d - 1) + 1$. $\square$

We can now bound the size of the BSP constructed by the algorithm.

Theorem 4.3.2 *A BSP of size $O(n^{4/3})$ can be constructed for n fat orthogonal rectangles in $\mathbb{R}^3$. The constant of proportionality in the big-oh term is linear in α^2, where α is the maximum aspect ratio of the input rectangles.*

Proof: If a box B has k vertices in its interior, one phase of cuts partitions B into eight boxes each of which has at most $\lfloor k/8 \rfloor$ vertices in its interior. Since we start with n rectangles that have at most $4n$ vertices, the number of phases executed by the above algorithm is at most $\lceil (\log n)/3 + 2/3 \rceil$. Lemma 4.3.1 now implies that the total number of rectangles formed once all the phases are executed is $O(n 2^{(\log n)/3}) = O(n^{4/3})$. At this stage, all boxes have only long rectangles. Hence, Theorem 4.2.2 and the remark at the end of Section 4.2 imply that we can construct a linear-size BSP in each of these boxes, which increases the total size of the BSP only by a constant factor. This proves the theorem. $\square$

4.4 An Improved Algorithm

We now describe our main algorithm for constructing a BSP for a set S of n fat non-intersecting rectangles, in which we simultaneously simulate the algorithm for long fat rectangles presented in Section 4.2 and partition the vertices of the rectangles in S. The algorithm proceeds in rounds. Each round simulates a few steps of the algorithm for long rectangles and partitions the vertices of the rectangles in S into a small number of sets of approximately equal size. At the beginning of the ith round, for $i > 0$, the algorithm has a top subtree $\mathcal{B}_i$ of the BSP for S. Let Q_i be the set of boxes associated with the leaves of $\mathcal{B}_i$ containing at least one rectangle. The initial tree $\mathcal{B}_1$ consists of one node and Q_1 consists of one box that contains all the input rectangles. Our algorithm maintains the invariant that for each box $B \in Q_i$, all long rectangles in S_B are non-free. If Q_i is empty, we are done. Otherwise, in the ith round, for each box $B \in Q_i$, we construct a top subtree $\mathcal{T}_B$ of the BSP for the set S_B and attach it to the corresponding leaf of $\mathcal{B}_i$. This gives us the new top subtree $\mathcal{B}_{i+1}$. Thus, it suffices to describe how to build the tree $\mathcal{T}_B$ on a box B during a round.

The algorithm proceeds in rounds. Each round simulates a few steps of the algorithm for long rectangles and partitions the vertices of the rectangles in S into a small number of sets of approximately equal size. At the beginning of the ith round, for $i > 0$, the algorithm has a top subtree $\mathcal{B}_i$ of the BSP for S. Let Q_i be the set of boxes associated with the leaves of $\mathcal{B}_i$ containing at least one rectangle. The initial tree $\mathcal{B}_1$ consists of one node and Q_1 consists of one box that contains all the input rectangles. Our algorithm maintains the invariant that for each box $B \in Q_i$, all long rectangles in S_B are non-free. If Q_i is empty, we are done. Otherwise, in the ith round, for each box $B \in Q_i$, we construct a top subtree $\mathcal{T}_B$ of the BSP for the set S_B and attach it to the corresponding leaf of $\mathcal{B}_i$. This gives us the new top subtree $\mathcal{B}_{i+1}$. Thus, it suffices to describe how to build the tree $\mathcal{T}_B$ on a box B during a round.

Let $F \subseteq S_B$ be the set of rectangles that are long with respect to B. Set $f = |F|$, and let k be the number of vertices of rectangles in S_B that lie in the interior of B (note that each such vertex is a vertex of an original rectangle in the input set S). By assumption, all rectangles in F are non-free. We choose a parameter a, which remains fixed throughout the round. We pick

$$a = 2^{\sqrt{\log(f+k)}}$$

to optimise the size of the BSP that the algorithm creates. We now describe the ith round in detail. See Figure 4.9 for an outline of $\mathcal{B}$'s structure.

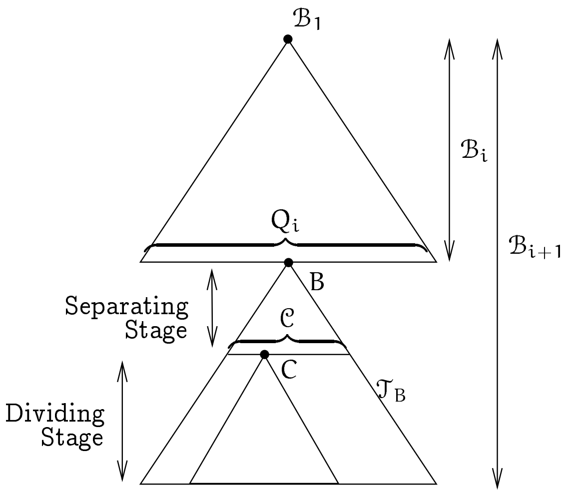


Figure 4.9: Overall structure of $\mathcal{B}$.

If $k = 0$ (i.e., if all rectangles in S_B are long), we use Theorem 4.2.2 to construct a BSP for S_B. Otherwise, we perform a sequence of cuts in two stages that partition B as follows:

Separating Stage: We apply the α-cuts, as described in Section 4.2. We make these cuts with respect to the rectangles in F, i.e., we consider only those rectangles of S_B that are long with respect to B. Let $\mathcal{C}$ be the set of boxes into which B is partitioned by the α-cuts.

Dividing Stage: We refine each box C in $\mathcal{C}$ by applying cuts similar to the ones made in Section 4.2.2, as described below. Let k_C denote the number of vertices of rectangles in S_C that lie in the interior of C. Recall that F_C is the set of rectangles in F that are clipped within C. We recursively invoke the dividing stage until $|F_C| + 2ak_C \leq (f + ak)/a$ and S_C does not contain any free rectangles.

1. If S_C has any free rectangle, we use the free cut containing that rectangle to split C into two boxes.
2. If the rectangles in F_C belong to two classes, let P_C denote the set of vertices of the rectangles in S_C that lie in the interior of C. We apply at most two parallel free cuts that satisfy Lemma 4.1.2, with $R = F, P = P_C$, and $w = 2a$.
3. If the rectangles in F_C belong to just one class, we apply one cut using Lemma 4.1.1, with $R = F, P = P_C$, and $w = 2a$.

The cuts introduced during the dividing stage can be made in a tree-like fashion. At the end of the dividing stage, we have a set of boxes so that for each box D in this set, S_D does not contain any free rectangle and $|F_D| + ak_D \leq (f + ak)/a$. Note that as we apply cuts in C and in the resulting boxes, rectangles that are short with respect to C may become long with respect to the new boxes. We ignore these new long rectangles until the next round, unless they induce a free cut.

4.5 Analysis of the Algorithm

We now analyze the size of the BSP constructed by the algorithm and the time complexity of the algorithm. In a round, the algorithm constructs a top subtree $\mathcal{T}_B$ of the BSP for the set of clipped rectangles S_B. Recall that F is the set of rectangles that are long with respect to B, $f = |F|$, and k is the number of vertices of rectangles in S_B that lie in the interior of B. For a node C in $\mathcal{T}_B$, recall that k_C denotes the number of vertices of rectangles in S_C that lie in the interior of C.

We now define some more notation that we need for the analysis. For a node C in $\mathcal{T}_B$, let $\mathcal{T}_C$ be the subtree of $\mathcal{T}_B$ rooted at C, L_C be the set of leaves in $\mathcal{T}_C$, ϕ_C be the number of long rectangles in F_C (recall that F_C is the set of rectangles in F that intersect C and are clipped within C), and n_C be the number of long rectangles in $S_C \setminus F_C$ (recall that a rectangle $S_C \setminus F_C$ is a portion of a rectangle in S_B that is short with respect to B). For a box D corresponding to a leaf of $\mathcal{T}_B$, let f_D be the number of long rectangles in S_D. Note that f_D counts both the "old" long rectangles in F_D (pieces of rectangles that were long with respect to B) and the "new" long rectangles in $S_D \setminus F_D$ (pieces of rectangles that were short with respect to B, but became long with respect to D due to the cuts made during the round); $f_D = \phi_D + n_D$.

In a round, the separating stage first splits B into a set of boxes $\mathcal{C}$. For each box $C \in \mathcal{C}$, F_C has only two classes of long rectangles. The algorithm then executes the dividing stage on each such box C. As in the case of the algorithm for long rectangles (see Section 4.2), the subtree constructed in C has the following property: if Step 3 is executed at a node v, then Step 2 is not executed at any descendent of v. Let D be a node in $\mathcal{T}_C$ where Step 3 is invoked. In Lemma 4.5.1, we prove a bound on the total number of long rectangles at each leaf of $\mathcal{T}_D$. We bound the number of long rectangles at the leaves of $\mathcal{T}_C$ in Lemmas 4.5.2 and 4.5.3. In Lemma 4.5.4, we prove a bound on the size of the tree $\mathcal{T}_B$. Finally, we use these lemmas to establish bounds on the size of the BSP constructed by our algorithm and the running time of our algorithm in Theorem 4.5.5.

Lemma 4.5.1 *For a box D associated with a leaf of $\mathcal{T}_B$,*

$$f_D + 2ak_D \leq \frac{f + 2ak}{a}.$$

Proof: We know that n_D is at most k (since a rectangle in $S_D \setminus F_D$ must be a piece of a rectangle short with respect to B, and there are at most k short rectangles in B). Since $f_D + 2ak_D \leq \phi_D + 2ak_D + n_D$ and $\phi_D + 2ak_D \leq (f + ak)/a$ (by construction), the lemma follows. □

Lemma 4.5.2 *Let C be a box associated with a node in $\mathcal{T}_B$. If all rectangles in F_C belong to one class, then*

$$\sum_{D \in L_C} f_D \leq 2\phi_C + 2n_C \max\left\{\frac{2(\phi_C + ak_C)}{\mu}, 1\right\} + 4k_C\left(\frac{\phi_C + ak_C}{\mu}\right),$$

where $\mu = (f + ak)/a$.

Proof: Assume, without loss of generality, that all rectangles in F_C belong to the top class, and let g be the top face of C. By Lemma 4.1.1(i), g contains an edge of every rectangle in F_C. Let ρ_C be the number of vertices of the non-free long rectangles in F_C that lie in the interior of g; obviously, $\phi_C \leq \rho_C \leq 2\phi_C$. Set

$$\Phi(\rho_C, n_C, k_C) = \max \sum_{D \in L_C} f_D,$$

where the maximum is taken over all boxes C and over all sets S of rectangles with ρ_C vertices of rectangles in F_C lying in the interior of the top face of C, n_C long rectangles in $S_C \setminus F_C$, and k_C vertices in the interior of C. We claim that

$$\Phi\left(\rho_C, n_C, k_C\right) \leq \rho_C + 2n_C \max\left\{\frac{\rho_C + 2ak_C}{\mu}, 1\right\} + 2k_C\left(\frac{\rho_C + 2ak_C}{\mu}\right), \tag{4.4.4}$$

which implies the lemma, because $\rho_C \leq 2\phi_C$.

Note that if S_C contains $m \geq 1$ free rectangles, we apply the free cuts containing these rectangles to partition C (by repeatedly invoking Step 1 of the dividing stage) until the resulting boxes do not contain any free rectangle. The free cuts partition C into a set $\mathcal{E}$ of $m+1$ boxes. Since we have created the boxes in $\mathcal{E}$ using free cuts,

$$\rho_E + 2ak_E \leq \rho_C + 2ak_C, \qquad \text{for any box } E \text{ in } \mathcal{E}, \tag{4.4.5}$$

$$\sum_{E\in\mathcal{E}} n_E \leq n_C, \qquad \sum_{E\in\mathcal{E}} k_E \leq k_C, \qquad \sum_{E\in\mathcal{E}} \rho_E \leq \rho_C. \tag{4.4.6}$$

These inequalities imply that if (4.4.4) holds for each box in $\mathcal{E}$, then (4.4.4) holds for C as well. Therefore, we prove (4.4.4) for all boxes C such that F_C contains only one class of rectangles and S_C does not contain any free rectangle. We proceed by induction on $\rho_C + 2ak_C$.

Base Case: $0 \leq \rho_C + 2ak_C \leq \mu$. Since $0 \leq \rho_C + 2ak_C \leq \mu$ and S_C does not contain any free rectangle, C is a leaf of $\mathcal{T}_B$, i.e., $L_C = \{C\}$. We have

$$\Phi\left(\rho_C, n_C, k_C\right) = \sum_{D\in L_C} f_D = f_C = \phi_C + n_C \leq \rho_C + n_C, \tag{4.4.7}$$

which implies (4.4.4).

Induction step: $\rho_C + 2ak_C > \mu$. In this case, C is split into two sub-boxes C_1 and C_2 by a cutting plane h. Since $\sum_{D\in L_C} f_D = \sum_{D\in L_{C_1}} f_D + \sum_{D\in L_{C_2}} f_D$,

$$\Phi\left(\rho_C, n_C, k_C\right) = \Phi\left(\rho_{C_1}, n_{C_1}, k_{C_1}\right) + \Phi\left(\rho_{C_2}, n_{C_2}, k_{C_2}\right),$$

where $k_{C_1} + k_{C_2} \leq k_C$ and $\rho_{C_1} + \rho_{C_2} \leq \rho_C$.

Note that h does not contain a free rectangle. For $i = 1, 2$, each long rectangle in $S_{C_i} \setminus F_{C_i}$ is contained either in a long rectangle in $S_C \setminus F_C$ or in a short rectangle in S_C. Since h intersects each rectangle in S_C at most once and a short rectangle intersected by h is divided into one short and one long rectangle,

$$n_{C_1} + n_{C_2} \leq 2n_C + k_C. \tag{4.4.8}$$

By Lemma 4.1.1(ii), we have

$$\rho_{C_i} + 2ak_{C_i} \leq \frac{\rho_C + 2ak_C}{2}, \qquad \text{for } i = 1, 2. \tag{4.4.9}$$

Let $\mathcal{E}_1$ (respectively, $\mathcal{E}_2$) be the set of boxes obtained by applying all the free cuts in S_{C_1} (respectively, S_{C_2}) in Step 1 of the dividing stage. Clearly,

$$\Phi(\rho_C, n_C, k_C) = \sum_{E \in \mathcal{E}_1} \Phi(\rho_E, n_E, k_E) + \sum_{E \in \mathcal{E}_2} \Phi(\rho_E, n_E, k_E).$$

We consider two cases.

Case (i): $\mu < \rho_C + 2ak_C \leq 2\mu$. For each $i = 1, 2$

$$\rho_{C_i} + 2ak_{C_i} \leq \frac{\rho_C + 2ak_C}{2} \leq \mu.$$

As a result, all boxes in $\mathcal{E}_1$ and $\mathcal{E}_2$ are leaves of $\mathcal{T}_B$. Using (4.4.5), and (4.4.7), we obtain

$$\begin{aligned}\Phi(\rho_C, n_C, k_C) &\leq \sum_{E \in \mathcal{E}_1} f_E + \sum_{E \in \mathcal{E}_2} f_E \leq \sum_{E \in \mathcal{E}_1} (\rho_E + n_E) + \sum_{E \in \mathcal{E}_2} (\rho_E + n_E) \\ &\leq \rho_{C_1} + n_{C_1} + \rho_{C_2} + n_{C_2}.\end{aligned}$$

It now follows from (4.4.8) that

$$\Phi(\rho_C, n_C, k_C) \leq \rho_C + 2n_C + k_C, \tag{4.4.10}$$

which implies (4.4.4), because $\rho_C + 2ak_C > \mu$.

Case (ii): $\rho_C + 2ak_C > 2\mu$. For any box E in $\mathcal{E}_1 \cup \mathcal{E}_2$, by (4.4.5) and (4.4.9),

$$\max\left\{\frac{\rho_E + 2ak_E}{\mu}, 1\right\} \leq \max\left\{\frac{\rho_C + 2ak_C}{2\mu}, 1\right\} = \frac{\rho_C + 2ak_C}{2\mu}.$$

By (4.4.5) and the induction hypothesis,

$$\begin{aligned}\Phi(\rho_C, n_C, k_C) &\leq \sum_{E \in \mathcal{E}_1} \left(\rho_E + 2n_E \left(\frac{\rho_C + 2ak_C}{2\mu}\right) + 2k_E \left(\frac{\rho_C + 2ak_C}{2\mu}\right)\right) \\ &\quad + \sum_{E \in \mathcal{E}_2} \left(\rho_E + 2n_E \left(\frac{\rho_C + 2ak_C}{2\mu}\right) + 2k_E \left(\frac{\rho_C + 2ak_C}{2\mu}\right)\right) \\ &\leq (\rho_{C_1} + \rho_{C_2}) + 2(n_{C_1} + n_{C_2}) \left(\frac{\rho_C + 2ak_C}{2\mu}\right) \\ &\quad + 2(k_{C_1} + k_{C_2}) \left(\frac{\rho_C + 2ak_C}{2\mu}\right)\end{aligned}$$

Using (4.4.8), we obtain

$$\begin{aligned}\Phi(\rho_C, n_C, k_C) &\leq \rho_C + 2(2n_C + k_C) \left(\frac{\rho_C + 2ak_C}{2\mu}\right) + 2k_C \left(\frac{\rho_C + 2ak_C}{2\mu}\right) \\ &= \rho_C + 2n_C \left(\frac{\rho_C + 2ak_C}{\mu}\right) + 2k_C \left(\frac{\rho_C + 2ak_C}{\mu}\right),\end{aligned}$$

which implies (4.4.4). □

Lemma 4.5.3 *Let* C *be a box associated with a node in* $\mathcal{T}_B$. *If all rectangles in* F_C *belong to two classes, then*

$$\sum_{D \in L_C} f_D \leq 2\phi_C + 5n_C \max\left\{\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3, 1\right\} + 6k_C \left(\frac{\phi_C + 2ak_C}{\mu}\right)^3,$$

where $\mu = (f + ak)/a$.

Proof: The proof of this lemma is long and tedious. We first give some intuition behind the proof. If $\phi_C + 2ak_C$ is the *weight* of box C, the cuts we make inside C in Step 2 of the dividing stage split it into at most three boxes. For each box C' that C is split into, either the number of classes in $F_{C'}$ is 1 or the weight of C' is two-thirds the weight of C. Since we split C into at most three boxes, the total number of long rectangles intersecting these boxes is roughly three times the number of rectangles in S_C. Therefore, the total number of long rectangles intersecting the boxes in L_C is roughly a factor of

$$3^{\log_{3/2}\left(\frac{\phi_C + 2ak_C}{\mu}\right)} = O\left(\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right)$$

more the number of rectangles in S_C.

We now give the formal proof of the lemma. Let

$$\Psi(\phi_C, n_C, k_C) = \max \sum_{D \in L_C} f_D,$$

where the maximum is taken over all boxes C and over all sets S of rectangles with ϕ_C long rectangles in F_C, n_C long rectangles in $S_C \setminus F_C$, and k_C vertices in the interior of C. The rectangles in F_C belong to at most two classes. We claim that

$$\Psi(\phi_C, n_C, k_C) \le 2\phi_C + 5n_C \max\left\{\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3, 1\right\} + 6k_C\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 \tag{4.4.11}$$

which proves the lemma.

If S_C contains $m \ge 1$ free rectangles, we apply the free cuts containing these rectangles to partition C (by repeatedly invoking Step 1 of the dividing stage) until the resulting boxes do not contain any free rectangles. Let $\mathcal{E}$ be the set of boxes into which C is so partitioned. Then

$$\phi_E + 2ak_E \le \phi_C + 2ak_C, \qquad \text{for any box E in } \mathcal{E}, \tag{4.4.12}$$

$$\sum_{E \in \mathcal{E}} \phi_E \le \phi_C \qquad \sum_{E \in \mathcal{E}} n_E \le n_C \qquad \sum_{E \in \mathcal{E}} k_E \le k_C. \tag{4.4.13}$$

These inequalities imply that if (4.4.11) holds for each box in $\mathcal{E}$, then (4.4.11) holds for C as well. Therefore, we prove (4.4.11) for all boxes C such that F_C contains at most two classes of rectangles and S_C does not contain any free rectangle. We proceed by induction on $\phi_C + 2ak_C$.

Base case: $\phi_C + 2ak_C \le \mu$. Since $\phi_C + 2ak_C \le \mu$ and S_C does not contain any free rectangles, C is a leaf of $\mathcal{T}_B$. We have

$$\Psi(\phi_C, n_C, k_C) = \sum_{D \in L_C} f_D = f_C = \phi_C + n_C, \tag{4.4.14}$$

which implies (4.4.11).

Induction step: $\phi_C + 2ak_C > \mu$. The cuts made in Step 2 of the dividing stage fall into one of two categories (see Lemma 4.1.2). Note that none of these cuts contains a free rectangle.

Case (i): We divide C into two boxes C_1 and C_2 using a plane h that does not cross any rectangle in F_C. As a result,

$$\phi_{C_1} + \phi_{C_2} \le \phi_C \qquad \text{and} \qquad k_{C_1} + k_{C_2} \le k_C.$$

Since h intersects each rectangle in S_C at most once, (4.4.8) holds in this case too. i.e., Lemma 4.1.2 implies that

$$n_{C_1} + n_{C_2} \le 2n_C + k_C. \tag{4.4.15}$$

Lemma 4.1.2 implies that

$$\phi_{C_i} + 2ak_{C_i} \le \frac{2(\phi_C + 2ak_C)}{3} \qquad \text{for } i = 1, 2. \tag{4.4.16}$$

Let $\mathcal{E}_1$ (resp., $\mathcal{E}_2$) be the set of boxes obtained by applying all the free cuts in S_{C_1} (resp., S_{C_2}) in Step 1 of the dividing stage. Clearly,

$$\Psi(\phi_C, n_C, k_C) \le \sum_{E \in \mathcal{E}_1} \Psi(\phi_E, n_E, k_E) + \sum_{E \in \mathcal{E}_2} \Psi(\phi_E, n_E, k_E).$$

We consider two cases.

(a) $\mu \le \phi_C + 2ak_C \le 3\mu/2$. In this case, by (4.4.12) and (4.4.16),

$$\phi_E + 2ak_E \le \frac{2(\phi_C + 2ak_C)}{3} \le \mu,$$

for each box E in $\mathcal{E}_1$ and $\mathcal{E}_2$. Since E does not contain a free rectangle, E is a leaf of $\mathcal{T}_B$. Using (4.4.15), (4.4.13) and (4.4.14), we obtain

$$\begin{aligned}
\Psi(\phi_C, n_C, k_C) &\le \sum_{E \in \mathcal{E}_1} f_E + \sum_{E \in \mathcal{E}_2} f_E \\
&\le \sum_{E \in \mathcal{E}_1} (\phi_E + n_E) + \sum_{E \in \mathcal{E}_2} (\phi_E + n_E) \\
&\le \phi_{C_1} + n_{C_1} + \phi_{C_2} + n_{C_2} \\
&\le \phi_C + 2n_C + k_C,
\end{aligned}$$

which implies (4.4.11), because $\phi_C + 2ak_C > \mu$.

(b) $\phi_C + 2ak_C > 3\mu/2$. For a box E in $\mathcal{E}_1 \cup \mathcal{E}_2$, by (4.4.12) and (4.4.16), we have

$$\max\left\{\left(\frac{\phi_E + 2ak_E}{\mu}\right)^3, 1\right\} \le \max\left\{\left(\frac{2}{3}\left(\frac{\phi_C + 2ak_C}{\mu}\right)\right)^3, 1\right\} = \frac{8}{27}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3.$$

By the induction hypothesis, and by using (4.4.13) and (4.4.15),

$$\begin{aligned}
\Psi(\phi_C, n_C, k_C) &\le \sum_{E\in\mathcal{E}_1} \left(2\phi_E + 5n_E\left(\frac{8}{27}\left(\frac{\phi_C+2ak_C}{\mu}\right)^3\right)\right.\\
&\qquad \left. +6k_E\left(\frac{8}{27}\left(\frac{\phi_C+2ak_C}{\mu}\right)^3\right)\right)\\
&\quad + \sum_{E\in\mathcal{E}_2} \left(2\phi_E + 5n_E\left(\frac{8}{27}\left(\frac{\phi_C+2ak_C}{\mu}\right)^3\right)\right.\\
&\qquad \left. +6k_E\left(\frac{8}{27}\left(\frac{\phi_C+2ak_C}{\mu}\right)^3\right)\right)\\
&\le 2\left(\phi_{C_1}+\phi_{C_2}\right) + 5\left(n_{C_1}+n_{C_2}\right)\left(\frac{8}{27}\left(\frac{\phi_C+2ak_C}{\mu}\right)^3\right)\\
&\qquad + 6\left(k_{C_1}+k_{C_2}\right)\left(\frac{8}{27}\left(\frac{\phi_C+2ak_C}{\mu}\right)^3\right)\\
&\le 2\phi_C + 5\cdot\frac{8}{27}\left(2n_C+k_C\right)\left(\frac{\phi_C+2ak_C}{\mu}\right)^3 + 6\cdot\frac{8}{27}k_C\left(\frac{\phi_C+2ak_C}{\mu}\right)^3\\
&\le 2\phi_C + 5n_C\left(\frac{\phi_C+2ak_C}{\mu}\right)^3 + 6k_C\left(\frac{\phi_C+2ak_C}{\mu}\right)^3,
\end{aligned}$$

which implies (4.4.11).

Case (ii): We find two parallel planes h_1 and h_2 that divide C into three boxes C_1, C_2, and C_3 (in this order) so that all rectangles in F_{C_2} belong to one class. Lemma 4.1.2 implies that the rectangles in F_C are partitioned among C_1, C_2, and C_3. Moreover, h_1 and h_2 partition the vertices in the interior of C. Thus,

$$\phi_{C_1}+\phi_{C_2}+\phi_{C_3} \le \phi_C \qquad \text{and} \qquad k_{C_1}+k_{C_2}+k_{C_3} \le k_C.$$

The planes h_1 and h_2 can intersect the rectangles in $S_C \setminus F_C$. Each long rectangle in $S_C \setminus F_C$ is partitioned into at most three long rectangles. Each short rectangle in S_C is partitioned into at most three rectangles; if two of these rectangles are long, then one of the long rectangles must be in S_{C_2}, since C_2 is sandwiched between C_1 and C_3 (see the proof of Lemma 4.1.2). In other words,

$$n_{C_1}+n_{C_3} \le 2n_C + k_C \qquad \text{and} \qquad n_{C_2} \le n_C + k_C. \tag{4.4.17}$$

Lemma 4.1.2 also implies that

$$\phi_{C_i} + 2ak_{C_i} \le \frac{2(\phi_C + 2ak_C)}{3}, \qquad \text{for } i = 1, 3, \text{ and} \qquad \phi_{C_2} + 2ak_{C_2} \le \phi_C + 2ak_C. \tag{4.4.18}$$

Let $\mathcal{E}_i, 1 \le i \le 3$ be the set of boxes obtained by applying all the free cuts in S_{C_i} in Step 1 of the dividing stage. Note that for each box $E \in \mathcal{E}_2$, F_E contains only one class of rectangles. It is clear that

$$\Psi(\phi_C, n_C, k_C) \le \sum_{E\in\mathcal{E}_1} \Psi(\phi_E, n_E, k_E) + \sum_{E\in\mathcal{E}_2} \Phi(2\phi_E, n_E, k_E) + \sum_{E\in\mathcal{E}_3} \Psi(\phi_E, n_E, k_E),$$

where $\Phi(\,)$ is as defined in the proof of Lemma 4.5.2. We again consider two cases.

(a) $\mu < \phi_C + 2ak_C \le 3\mu/2$. By (4.4.18), each box in $\mathcal{E}_1$ and $\mathcal{E}_3$ is a leaf of $\mathcal{T}_B$. Therefore, by (4.4.14),

$$\begin{aligned}
\Psi(\phi_C, n_C, k_C) &\le \sum_{E\in\mathcal{E}_1} f_E + \sum_{E\in\mathcal{E}_2} \Phi\left(2\phi_E, n_E, k_E\right) + \sum_{E\in\mathcal{E}_3} f_E \\
&\le \sum_{E\in\mathcal{E}_1} (\phi_E + n_E) + \sum_{E\in\mathcal{E}_3} (\phi_E + n_E) + \sum_{E\in\mathcal{E}_2} \Phi\left(2\phi_E, n_E, k_E\right) \\
&\le (\phi_{C_1} + \phi_{C_3}) + (n_{C_1} + n_{C_3}) + \sum_{E\in\mathcal{E}_2} \Phi\left(2\phi_E, n_E, k_E\right).
\end{aligned}$$

For each box $E \in \mathcal{E}_2$, $\phi_E + 2ak_E \le 3\mu/2$. Hence, by (4.4.10), we have

$$\Phi\left(2\phi_E, n_E, k_E\right) \le 2\phi_E + 2n_E + k_E.$$

As a result,

$$\begin{aligned}
\Psi(\phi_C, n_C, k_C) &\le (\phi_{C_1} + \phi_{C_3}) + (n_{C_1} + n_{C_3}) + \sum_{E\in\mathcal{E}_2} (2\phi_E + 2n_E + k_E) \\
&\le (\phi_{C_1} + \phi_{C_3}) + (n_{C_1} + n_{C_3}) + (2\phi_{C_2} + 2n_{C_2} + k_{C_2}) \\
&\le 2\phi_C + 4n_C + 4k_C,
\end{aligned}$$

where the last inequality follows from (4.4.17). This inequality implies (4.4.11).

(b) $\phi_C + 2ak_C > 3\mu/2$. For a box $E \in \mathcal{E}_1 \cup \mathcal{E}_3$,

$$\max\left\{\left(\frac{\phi_E + 2ak_E}{\mu}\right)^3, 1\right\} \le \max\left\{\left(\frac{2}{3}\left(\frac{\phi_C + 2ak_C}{\mu}\right)\right)^3, 1\right\} = \frac{8}{27}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3.$$

Similarly, for a box $E \in \mathcal{E}_2$,

$$\max\left\{2\left(\frac{\phi_E + ak_E}{\mu}\right), 1\right\} \leq 2\left(\frac{\phi_C + 2ak_C}{\mu}\right).$$

By the induction hypothesis and (4.4.4),

$$\begin{aligned}
\Psi(\phi_C, n_C, k_C) \leq & \sum_{E\in\mathcal{E}_1}\left(2\phi_E + 5\cdot\frac{8}{27}n_E\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 + 6\cdot\frac{8}{27}k_E\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right) \\
& + \sum_{E\in\mathcal{E}_2}\left(2\phi_E + 4n_E\left(\frac{\phi_C + 2ak_C}{\mu}\right) + 4k_E\left(\frac{\phi_C + 2ak_C}{\mu}\right)\right) \\
& + \sum_{E\in\mathcal{E}_3}\left(2\phi_E + 5\cdot\frac{8}{27}n_E\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 + 6\cdot\frac{8}{27}k_E\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right).
\end{aligned}$$

Since $\phi_C + 2ak_C > 3\mu/2$,

$$\frac{\phi_C + 2ak_C}{\mu} \leq \frac{4}{9}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3.$$

Therefore, using (4.4.17), we have

$$\begin{aligned}
\Psi(\phi_C, n_C, k_C) \leq & \, 2(\phi_{C_1} + \phi_{C_2} + \phi_{C_3}) + 5\cdot\frac{8}{27}(n_{C_1} + n_{C_3})\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 \\
& + \frac{16}{9}n_{C_2}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 + 6\cdot\frac{8}{27}(k_{C_1} + k_{C_3})\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 \\
& + \frac{16}{9}k_{C_2}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 \\
\leq & \, 2\phi_C + 5n_C\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 + 6k_C\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3,
\end{aligned}$$

which implies (4.4.11).

□

We can use the previous three lemmas to prove the following lemma about the tree constructed during a round, which is crucial to the analysis of the algorithm:

Lemma 4.5.4 *The tree $\mathcal{T}_B$ constructed on box B in a round has the following properties:*

(i) $\sum_{D\in L_B} k_D \leq k$,

(ii) $\sum_{D \in L_B} f_D = O(f + a^3 k)$, *and*

(iii) $|\mathcal{T}_B| = O((f + a^3 k) \log a)$.

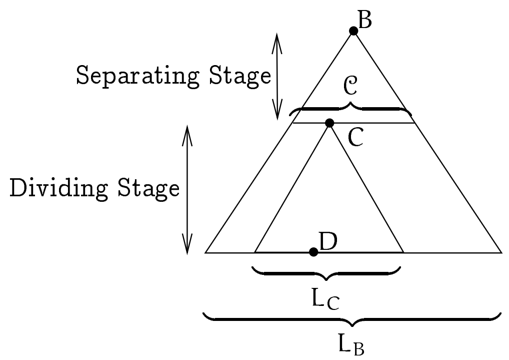


Figure 4.10: The tree $\mathcal{T}_B$ constructed in a round.

Proof: The bound on $\sum_{D \in L_B} k_D$ follows, since each vertex in the interior of S_B lies in the interior of at most one box of L_B. Next, we will use Lemma 4.5.3 to prove a bound on $\sum_{D \in L_B} f_D$.

Let $\mathcal{C}$ be the set of boxes into which B is partitioned by the separating stage. See Figure 4.10. Obviously, $\sum_{D \in L_B} f_D = \sum_{C \in \mathcal{C}} \sum_{D \in L_C} f_D$. For each box $C \in \mathcal{C}$, Lemma 4.2.1(i) implies that all rectangles in F_C belong to at most two classes. Hence, by Lemma 4.5.3,

$$\sum_{D \in L_B} f_D \leq \sum_{C \in \mathcal{C}} O\left(\phi_C + (n_C + k_C)\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right).$$

Arguing as in the proof of Lemma 4.2.1(iii), we can show that $\sum_{C \in \mathcal{C}} \phi_C = O(f)$ and that $\sum_{C \in \mathcal{C}} n_C = O(k)$. We also know that $\sum_{C \in \mathcal{C}} k_C \leq k$ and $\mu = (f + ak)/a$. Therefore,

$$\sum_{D \in L_B} f_D = O\left(f + k\left(\frac{f + 2ak}{\mu}\right)^3\right) = O\left(f + a^3 k\right).$$

We now sketch the proof that $|\mathcal{T}_B| = O((f + a^3 k) \log a)$. Following the same argument as in the proof of Lemma 4.2.1(ii), we can show that the separating stage creates a tree with $O(f+k)$ nodes. We now count the number of nodes in $\mathcal{T}_B$ that are created by the dividing stage. Let $D \in \mathcal{T}_B$ be such a node. If D is partitioned by a cut containing a free rectangle (i.e., Step 1 of the dividing stage is invoked at D), we charge D to its nearest ancestor $C \in \mathcal{T}_B$ such that C is not partitioned by a free cut (i.e., Step 2 or 3 of the dividing stage is executed at C). Otherwise, we charge a cost of 1 to D itself. Let C be a node in $\mathcal{T}_B$ that is not partitioned by a free cut. Since a free rectangle can be created only by partitioning a long rectangle, the cut used to partition C creates $O(\phi_C + n_C)$ free rectangles, which implies that C is charged $O(\phi_C + n_C + 1)$ times by the above argument. Hence,

$$|\mathcal{T}_B| = O\left(\sum_C (\phi_C + n_C + 1)\right),$$

where C ranges over all nodes in $\mathcal{T}_B$ where Step 2 or 3 of the dividing stage is executed. By following an inductive argument similar to the ones used to prove Lemmas 4.5.2 and 4.5.3, we can show that $|\mathcal{T}_B| = O((f + a^3 k)\log a)$. Informally, the charging scheme compresses $\mathcal{T}_B$ by collapsing all nodes that are split by free cuts. Lemma 4.1.1 and 4.1.2 imply that the height of the compressed tree is $O(\log a)$. We can show that $\sum_C (\phi_C + n_C + 1)$ is roughly the product of the height of the tree and $\sum_D f_D$, the total number of long rectangles intersecting the leaves of the tree.

□

We now present our main result regarding the performance of our improved algorithm:

Theorem 4.5.5 *Given a set S of n rectangles in $\mathbb{R}^3$ such that the aspect ratio of each rectangle in S is bounded by a constant $\alpha \geq 1$, we can construct a BSP of size $n2^{O(\sqrt{\log n})}$ for S in time $n2^{O(\sqrt{\log n})}$. The constants of proportionality in the big-oh terms are linear in $\log \alpha$.*

Proof: We first bound the size of the BSP constructed by the algorithm. Let $S(f, k)$ denote the maximum size of the BSP produced by the algorithm for a box B that contains f long rectangles and k vertices in its interior. If $k = 0$, Theorem 4.2.2 implies that $S(f, k) = O(f \log f)$. For $k > 0$, by Lemma 4.5.4(iii), we construct the subtree $\mathcal{T}_B$ on B of size $O((f + a^3 k)\log a)$ in one round, and recursively construct subtrees for each box in the set of leaves L_B. Therefore, there exist constants $c_1, c_2, c_3 > 0$ so that the size $S(f,k)$ satisfies the following recurrence:

$$S(f,k) \le \begin{cases} c_1 f \log f & \text{for} \quad k = 0, \\ \\ \sum_{D \in L_B} S(f_D, k_D) + c_2(f + a^3 k)\log a & \text{for} \quad k > 0, \end{cases} \tag{4.4.19}$$

where

$$f_D + 2ak_D \le \frac{f + 2ak}{a} \tag{4.4.20}$$

for every box D in L_B (by Lemma 4.5.1), and

$$\sum_D k_D \le k, \qquad \sum_D f_D \le c_3(f + a^3 k) \tag{4.4.21}$$

(by Lemma 4.5.4(i) and 4.5.4(ii)). We now prove that the solution to the above recurrence is

$$S(f,k) = f 2^{r\sqrt{\log(f+2ak)}} + k 2^{s\sqrt{\log(f+2ak)}},$$

where r and s are constants linear in $\log \alpha$, with $s \ge 2r$. It is easy to show that the solution is correct for $k = 0$, since $c_1 f \log f \le f 2^{r\sqrt{\log f}}$ for $r \ge \log c_1$. For $k \ge 0$, we prove the claim by induction on $f + 2ak$.

Base case: $f + 2ak = 5$. In this case, it is easily seen that $f = k = 1$. Then $a = 2^{\sqrt{\log(f+k)}} = 2$. The set S_B consists of one long rectangle and one short rectangle. In this case, it can be checked (by using Lemma 4.1.1) that our algorithm constructs a BSP of size at most 10. Hence, $S(1,1) \le 10 \le 2^{r\sqrt{\log 5}} + 2^{s\sqrt{\log 5}}$, provided that $r, s \ge 3$.

Induction step: $f + 2ak > 5$. Since $f_D + 2ak_D < f + 2ak$, using the induction hypothesis, we obtain

$$S(f,k) \le \sum_D f_D 2^{r\sqrt{\log(f_D + 2ak_D)}} + \sum_D k_D 2^{s\sqrt{\log(f_D+2ak_D)}} + c_2(f \log a + a^3 k)$$

Using (4.4.20) and (4.4.21), we have

$$\begin{aligned} S(f,k) &\le c_3\left(f + a^3 k\right) 2^{r\sqrt{\log((f+2ak)/a)}} + k2^{s\sqrt{\log((f+2ak)/a)}} + c_2\left(f \log a + a^3 k\right) \\ &\le \left(c_3 + \frac{c_2 \log a}{2^{r\sqrt{\log((f+2ak)/a)}}}\right) f 2^{r\sqrt{\log((f+2ak)/a)}} \\ &\quad + \left(c_3 a^3 2^{(r-s)\sqrt{\log((f+2ak)/a)}} + 1 + \frac{c_2 a^3}{2^{s\sqrt{\log((f+2ak)/a)}}}\right) k 2^{s\sqrt{\log((f+2ak)/a)}}. \end{aligned}$$

If $r \geq 9$, then it can be checked that $2^{r\sqrt{\log((f+2ak)/a)}} \geq a^3$. Since $a = 2^{\sqrt{\log(f+k)}}$, by choosing $s \geq 2r$, we obtain

$$\begin{aligned} S(f,k) &\leq (c_2+c_3)f2^{r\sqrt{\log(f+2ak)-\sqrt{\log(f+k)}}} + (1+c_2+c_3)k2^{s\sqrt{\log(f+2ak)-\sqrt{\log(f+k)}}} \\ &= f2^{r\sqrt{\log(f+2ak)-\sqrt{\log(f+k)}}+\log(c_2+c_3)} + k2^{s\sqrt{\log(f+2ak)-\sqrt{\log(f+k)}}+\log(1+c_2+c_3)}. \end{aligned}$$

If $r > 4\log(c_2+c_3)$, we have

$$\begin{aligned} r\sqrt{\log(f+2ak)-\sqrt{\log(f+k)}}+\log(c_2+c_3) &\leq r\left(\sqrt{\log(f+2ak)-\sqrt{\log(f+k)}}+\frac{1}{4}\right) \\ &\leq r\sqrt{\log(f+2ak)}. \end{aligned}$$

Similarly, since $s \geq 2r$,

$$s\sqrt{\log(f+2ak)-\sqrt{\log(f+k)}}+\log(1+c_2+c_3) \leq s\sqrt{\log(f+2ak)}.$$

As a result, we obtain

$$S(f,k) \leq f2^{r\sqrt{\log(f+2ak)}} + k2^{s\sqrt{\log(f+2ak)}}, \tag{4.4.22}$$

as desired, provided that $r \geq \max\{\log c_1, 4\log(c_2+c_3), 9\}$ and $s \geq 2r$. Lemma 4.2.1 implies $c_2 \leq 26\lfloor\alpha\rfloor^2$ and $c_3 \leq 16\lfloor\alpha\rfloor$. As a result, both r and s are linear in $\log\alpha$. Since $f \leq n$ and $k \leq 4n$ at the beginning of the first round, we get from (4.4.22) the bound $n2^{O(\sqrt{\log n})}$ on the size of the BSP constructed by the algorithm.

We now bound the running time of our algorithm. Recall that we initially sorted the vertices of the rectangles in S by x-, y-, and z-coordinates. Suppose S_B does not contain a free rectangle. By Lemmas 4.1.1 and 4.1.2, the cuts to be made at B can be determined in $O(|S_B|)$ time. Suppose C is a box obtained by partitioning B according to these cuts; we can easily obtain the sorted order of the vertices of the rectangles in S_C from the sorted order in B. Let $\mathcal{E}$ be the set of boxes obtained by splitting C using all the free rectangles in S_C. Since the free rectangles in S_C are parallel to each other, we can partition the rectangles in S_C among the boxes in $\mathcal{E}$ in $O(|S_C|)$ time. Therefore, we can construct the tree representing the partition of B into the set of boxes E in $O(|S_B|)$ time. Hence, we obtain the same $n2^{O(\sqrt{\log n})}$ bound for the running time of the algorithm.

□

Remark: We can modify our algorithm to prove that the height of the BSP constructed is $O(\log n)$: if S_B contains free rectangles, we assign appropriate weights to the free rectangles, and partition B using the weighted median of the free rectangles. Paterson and Yao [81] use a similar idea to bound the height of the BSP they construct for segments in the plane.

4.6 Extensions

In this section, we extend the algorithm of Section 4.4 to the following three cases: (i) some of the input rectangles are thin, (ii) some of the input polygons are triangles, and (iii) some of the (fat) input rectangles intersect.

4.6.1 Fat and thin rectangles

Let us assume that the input $S = F \cup T$ has n rectangles, consisting of $m \geq 1$ thin rectangles in T and $n - m$ fat rectangles in F. We first describe our algorithm and then construct a set of rectangles for which any BSP has size $\Omega(n\sqrt{m})$. The algorithm we use now is very similar to the algorithm for fat rectangles. Given a box B, let f be the number of long rectangles in F_B, k the number of vertices of rectangles in F_B that lie in the interior of B, and t the number of rectangles in T_B. We fix the parameter $a = 2^{\sqrt{\log(f+k)}}$ and perform the following steps:

1. If S_B contains a free rectangle, we use the corresponding free cut to split B into two boxes.

2. If $k = t = 0$, we use the algorithm for long rectangles to construct a BSP for the set of clipped rectangles S_B.

3. If $t \geq (f+k)$, we use the algorithm by Paterson and Yao for orthogonal rectangles in $\mathbb{R}^3$ to construct a BSP for S_B [82].

4. If $(f + k) > t$, we perform one round of the algorithm described in Section 4.4, with the difference that we also use thin rectangles to make free cuts.

This algorithm is recursively invoked on all the resulting sub-boxes. Let $S(f, k, t)$ be the maximum size of the BSP produced by this algorithm for a box B with k

vertices in its interior, f long rectangles in F_B, and t thin rectangles in T_B. Note that in Section 4.5, during the analysis of a round, we did not use the fact that the short rectangles were fat. As a result, Lemmas 4.5.2 and 4.5.3 hold for Step 4 above with $n_C + k_C + t_C$ replacing the term $n_C + k_C$. We can similarly extend Lemma 4.5.4 to obtain the following recurrence for $S(f, k, t)$. Here D ranges over all the boxes that B is divided into by a round of cuts, as described above in Step 4.

$$S(f,k,t) = \begin{cases} O(f \log f), & \text{for} \quad k = t = 0, \\ O(t\sqrt{t}), & \text{for} \quad t \geq f + k, \\ \sum_D S(f_D, k_D, t_D) + O(f \log a + a^3 k + a^3 t), & \text{for} \quad f + k > t, \end{cases}$$

where $\sum_D k_D \leq k$, $f_D + 2ak_D \leq (f + 2ak)/a$, $\sum_D f_D = O(f + a^3 k)$, and $\sum_D t_D = O(a^3 t)$.

We can analyse this recurrence as in Section 4.5 and show that its solution is

$$S(f,k,t) = (f+k)\sqrt{t}2^{O(\sqrt{\log(f+k)})},$$

where the constant of proportionality is linear in $\log \alpha$ with $q \geq 2p$. The following theorem is immediate.

Theorem 4.6.1 *Let S be a set of n rectangles in $\mathbb{R}^3$, of which $m \geq 1$ are thin. A BSP of size $n\sqrt{m}2^{O(\sqrt{\log n})}$ for S can be constructed in $n\sqrt{m}2^{O(\sqrt{\log n})}$ time. The constants of proportionality in the big-oh terms are linear in $\log \alpha$, where α is the maximum aspect ratio of the fat rectangles.*

We now show that Theorem 4.6.1 is near-optimal by constructing a set of n rectangles of which m are thin for which any BSP has size $\Omega(n\sqrt{m})$. Recall that there exists a set of m thin rectangles in $\mathbb{R}^3$ for which any BSP has size $\Omega(m\sqrt{m})$ [82]. To complete the proof of the lower bound, we now exhibit a set $S = T \cup F$ of n rectangles, where T is a set of m thin rectangles and F is a set of $n - m$ fat rectangles, for which any BSP has size $\Omega((n-m)\sqrt{m})$. We first describe the rectangles in T and then the rectangles in F. We assume without loss of generality that m is an even perfect square and that

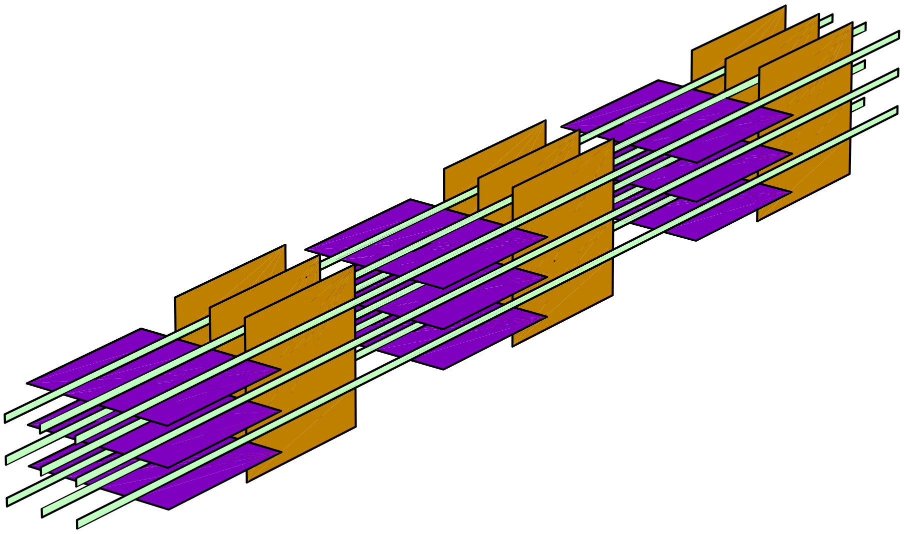


Figure 4.11: Lower bound construction for thin and fat rectangles.

n is a multiple of m. Consider a unit lattice of size $\sqrt{m} \times \sqrt{m}$ in the xz-plane. The points of the lattice have coordinates $(i, 0, j)$, where $0 \leq i, j < \sqrt{m}$. On each such lattice point, we erect a segment of length $n - m + 5(n-m)/(2\sqrt{m}) + 1$ (perpendicular to the xz-plane) in the $+y$-direction. These m segments constitute T. We now construct $(n-m)/(2\sqrt{m})$ sets, each consisting of $2\sqrt{m}$ squares with side length $\sqrt{m}+2$ as follows (we specify each square by giving the coordinates of two opposite corners of the square): Each set consists of $\sqrt{m}$ squares parallel to the xy-plane and $\sqrt{m}$ squares parallel to the yz-plane. Let $\delta = 2\sqrt{m} + 5$. For $0 \leq k < (n-m)/(2\sqrt{m})$, the kth set contains the set of squares

$$\{[(-1, \delta k + 1 - \varepsilon, j - \varepsilon), (\sqrt{m}+1, \delta k + \sqrt{m} + 3 - \varepsilon, j - \varepsilon)] \mid 0 \leq j < \sqrt{m}\}$$

parallel to the xy-plane and the set of squares

$$\{[(i - \varepsilon, \delta k + \sqrt{m} + 3 + \varepsilon, -1), (i - \varepsilon, \delta(k+1) + \varepsilon, \sqrt{m}+1)] \mid 0 \leq i < \sqrt{m}\}$$

parallel to the yz-plane, for some sufficiently small $\varepsilon > 0$. Note that in a set, each square parallel to the xy-plane is at a distance of 2ε from each square parallel to the

yz-plane. Further, for any square, the closest square in a different set is at a distance of $1 - 2\varepsilon$ and the closest thin rectangle is at a distance of ε.

We now prove that any BSP $\mathcal{B}$ for S has size $\Omega((n-m)\sqrt{m})$. Recall that we have defined the size of $\mathcal{B}$ to be the sum of the number of nodes in $\mathcal{B}$ and the total number of faces of all dimensions of the rectangles stored at the nodes in $\mathcal{B}$. Thus, it suffices to show that the number of proper intersection points between the cutting planes in $\mathcal{B}$ and edges of the rectangles in S (i.e., the number of points at which a cutting plane crosses an edge) is $\Omega((n-m)\sqrt{m})$.

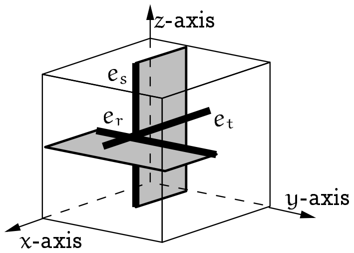


Figure 4.12: A cube ψ and the rectangles in S it intersects. The edges e_r, e_s and e_t are drawn in bold.

Consider a cube of side 4ε centered at each of the following $(n-m)\sqrt{m}/2$ points:

$$\{(i, \delta k + \sqrt{m} + 3, j) \mid 0 \leq i, j < \sqrt{m}, 0 \leq k < (n-m)/(2\sqrt{m})\}.$$

These cubes do not intersect each other if ε is chosen small enough. Each such cube ψ intersects a thin rectangle t of T and two squares r and s of F (one of the squares, say r, is parallel to the xy-plane and the other, s, is parallel to the yz-plane). Define e_r to be the edge of the rectangle $r \cap \psi$ that lies in the interior of ψ. Define e_s and e_t similarly (since t is a segment, e_t is just the intersection of t with ψ). See Figure 4.12. We claim that at least one of e_r, e_s, and e_t must be crossed by a cutting plane of $\mathcal{B}$, which implies that the cutting planes in $\mathcal{B}$ and the edges of the rectangles in S cross at $\Omega((n-m)\sqrt{m})$ points. Recall that no rectangle in S intersects the interior of the region associated with a leaf of $\mathcal{B}$. But e_r, e_s, and e_t are contained in the region corresponding to the root of $\mathcal{B}$. Hence, there must be a node $v \in \mathcal{B}$ such that at least one of e_r, e_s, and e_t is not contained in the interior of $\mathcal{R}_v$ and v has least height among all such nodes. Assume without loss of generality that e_r is not contained in the interior of $\mathcal{R}_v$. Let u be v's parent. If the cutting plane H_u at u contains e_r, H_u must intersect either e_s or e_t, proving the claim. Therefore, e_r does not intersect $\mathcal{R}_v$. Now, if H_u contains e_s, it must intersect e_t (since it does not intersect e_r), which also proves the claim. Therefore, H_u separates e_r and e_s. Since we can bring e_r and e_s arbitrarily close to each other, H_u is nearly parallel to the y-axis. Therefore, H_u intersects e_t, thus completing the proof.

4.6.2 Fat rectangles and triangles

Suppose that $p \geq 1$ polygons in the input S are (non-orthogonal) triangles and that the rest are fat rectangles. To construct a BSP for S, we use non-orthogonal cutting planes; hence, each region is a convex polytope and the intersection of a triangle with a region is a polygon, possibly with more than three edges. We can extend the algorithm of Section 4.6.1 to this case as follows: In 1, we check whether we can make free cuts through the non-orthogonal polygons too. In Step 3, if the number of triangles at a node is greater than the number of fat rectangles, we use the algorithm of Agarwal et al. for triangles in $\mathbb{R}^3$ to construct a BSP of size quadratic in the number of triangles in near-quadratic time [4]. Proceeding as in the previous section, we can prove the following theorem:

Theorem 4.6.2 *A BSP of size $np2^{O(\sqrt{\log n})}$ can be constructed in $np^2 2^{O(\sqrt{\log n})}$ time for n polygons in $\mathbb{R}^3$, of which $p \geq 1$ are non-orthogonal and the rest are fat rectangles. The constants of proportionality in the big-oh terms are linear in $\log \alpha$, where α is the maximum aspect ratio of the fat rectangles.*

We next show that unlike the case of rectangles, the fatness assumption does not help for triangles. More specifically, we construct a set of n triangles, each having bounded aspect ratio, for which any BSP has size $\Omega(n^2)$.

Our construction is similar to that of Paterson and Yao for proving a quadratic lower bound on the size of a BSP for (thin) triangles in $\mathbb{R}^3$ [81][2] Consider two families of segments lying inside the box $[0, n+1] \times [0, n+1] \times [0, n^2+1]$: the segments belonging to one family lie along the lines $\{x = i, z = iy - \varepsilon/2 \mid 1 \leq i \leq n\}$, where $0 < \varepsilon \leq 1/n^2$; these lines lie in the surface $z = xy - \varepsilon/2$. The segments in the

[2]Chazelle [27] uses a similar construction to prove a lower bound on the size of convex decompositions of polyhedra in $\mathbb{R}^3$.

second family are orthogonal to the ones in the first family and lie along the lines $\{y = i, z = ix + \varepsilon/2 \mid 1 \leq i \leq n\}$; the lines lie in the surface $z = xy + \varepsilon/2$. Figure 4.13 displays these lines. For each segment lying in the surface $z = xy - \varepsilon/2$, we construct an equilateral triangle with the segment as base and whose apex lies below the surface. Similarly, for each segment lying in the surface $z = xy + \varepsilon/2$, we construct an equilateral triangle with the segment as base and whose apex lies above the surface. The set S consists of these n non-intersecting fat triangles. Consider the "square" formed by two consecutive segments in the top family of segments and two consecutive segments in the bottom family. We can show that any BSP $\mathcal{B}$ for S must cut one of these four segments, which implies a lower bound of $\Omega(n^2)$ on the size of $\mathcal{B}$.

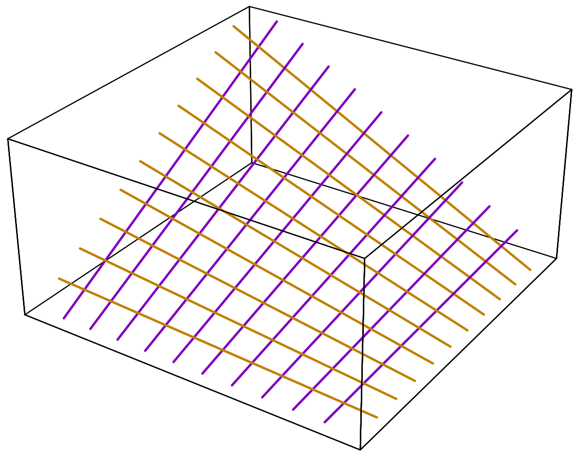


Figure 4.13: The edges of the notches lying on the hyperbolic paraboloids $z = xy - \varepsilon/2$ and $z = xy + \varepsilon/2$.

4.6.3 Intersecting fat rectangles

We now consider the case in which the n fat rectangles contain $k \leq \binom{n}{2}$ intersecting pairs. We construct the BSP using the following rather naive approach. For each intersecting pair of rectangles, we partition one of the rectangles in the pair into a constant number of rectangles such that the "smaller" rectangles do not intersect the other rectangle in the pair. This process creates a total of $n + O(k)$ rectangles. Some or all of the "new" $O(k)$ rectangles may be thin. We then use the algorithm of Section 4.6.1

to construct a BSP for the rectangles. The theorem below follows.

Theorem 4.6.3 *A BSP of size* $(n+k)\sqrt{k}2^{O(\sqrt{\log n})}$ *can be constructed in* $(n+k)\sqrt{k}2^{O(\sqrt{\log n})}$ *time for* n *rectangles in* $\mathbb{R}^3$*, which have* k *intersecting pairs of rectangles. The constants of proportionality in the big-oh terms are linear in* $\log \alpha$*, where* α *is the maximum aspect ratio of the fat rectangles.*

We can construct a set S of $O(n)$ rectangles containing k intersecting pairs of rectangles so that any BSP for S has size $\Omega(n + k\sqrt{k})$. Consider a set of $\sqrt{3k} = O(n)$ squares divided into three families of size $\sqrt{k/3}$ each as follows: the squares parallel to the xy-plane belong to the set

$$\{[(0,0,i),(n,n,i)] \mid 0 \leq \sqrt{k/3}\}.$$

The squares parallel to the yz- and xz-planes are defined analogously. Each square in a family intersects all the squares in the other two families. The total number of intersecting pairs of rectangles is k. Hence, the total number of vertices in the arrangement formed by the squares is $\Omega(n + k\sqrt{k})$. By an argument similar to that used to prove the lower bound in Section 4.6.1, we can show that any BSP for such a set of squares must contain $\Omega(n + k\sqrt{k})$ leaves. Hence, any BSP for these squares has size $\Omega(n + k\sqrt{k})$. Note that our upper bound is near-optimal when k is small.

4.7 Conclusions

Since worst-case complexities of BSPs are very high ($\Omega(n^{3/2})$ for n orthogonal rectangles in $\mathbb{R}^3$) and all known examples that achieve the worst case use configurations of thin rectangles that rarely occur in practice, we have made the natural assumption that rectangles are fat and have shown that this model of geometric complexity allows smaller worst-case size of BSPs. We believe that the $2^{\sqrt{\log n}}$ factor in our results is an artifact of our analysis; it should be possible to replace it by a polylogarithmic factor.

We showed in Section 4.6.2 that quadratic lower bounds hold on the size of BSPs for fat triangles. The edges of the triangles used in all such constructions have a linear number of distinct orientations. This observation leads to the question of whether sub-quadratic-size BSPs can be constructed for non-orthogonal objects if their edges have a small number of distinct orientations.

In all lower-bound constructions we have seen, the size of the BSP is roughly the same as number of points at which the rectangles or triangles in the constructions come very close to each other. An interesting open problem is to define a measure of geometric complexity that avoids this kind of uncommon worst-case behaviour and examine BSP size in this model.

Chapter 5

Binary Space Partitions for Rectangles in Practice

In the last chapter, we presented and analysed an algorithm for constructing BSPs for a set of fat and thin rectangles in $\mathbb{R}^3$. In this chapter, we describe our implementation of this algorithm, which we call Rounds, and study its performance on "real" data sets. In our implementation, we do not differentiate between fat and thin rectangles but do take advantage of rectangles with small aspect ratio, when such rectangles are part of the input. Our experiments show that Rounds is indeed practical: it constructs BSPs of near-linear size on real data sets (the size varies between 1.5 and 1.8 times the number of input rectangles).

While implementing our algorithm, we found that no systematic comparison of existing algorithms to construct BSPs has been performed. Therefore, we have implemented many other algorithms described in the literature and conducted a methodical study of the empirical performance of these algorithms and our algorithm. Instead of just implementing these algorithms as they appear in the literature, we have modified them to improve their performance. To compare the different algorithms, we measure the size of the BSP each algorithm constructs and the time spent in answering various queries like point location and ray shooting. Our algorithm performs better than not only theoretical algorithms like Paterson and Yao's algorithm [82] but also most other techniques described in the literature [8, 47, 102]. The only algorithm that performs better than our algorithm on some data sets is Teller's algorithm [101]; even in these cases, our algorithm has certain advantages in terms of the trade-off between the size of the BSP and query times (see Section 5.2).

To compare the performance of the different algorithms, we measure the size of the BSP each algorithm constructs and the time spent in answering various queries. The size measures the storage needed for the BSP and the time taken to compute a back-to-front order using the BSP (when the BSP is used in the painter's algorithm, for example). We use queries that are typically made in many BSP-based algorithms [8, 48, 101]:

1. *point location*: determine the leaf of the BSP that contains a query point. Point location is a fundamental geometric query and is the basis of many algorithms for answering other queries.

2. *ray shooting*: determine the first rectangle intersected by a query ray. Ray shooting is a very useful query in visibility problems, since it can be used to determine which object is visible along a given direction. In fact, this query forms the backbone of the hidden-surface removal algorithm we presented in Chapter 3.

It is worthwhile to point out the problem of robustness that bedevils geometric computing in general does not affect us, since we are dealing with orthogonal rectangles; all calculations (coordinate comparisons, rectangle intersections, etc.) can be carried out using only the coordinates of the vertices of the input rectangles. A related issue is that our algorithm *must* be able to handle degeneracy. In fact, the algorithm should take advantage of co-planar rectangles to reduce the size of the BSP and should not resort to standard perturbation techniques to remove degeneracies.

The rest of the chapter is organised as follows: We describe the other algorithms and heuristics that we have implemented in Section 5.1. In Section 5.2, we present the results of our experiments. We conclude in Section 5.3.

5.1 Other Algorithms

In this section we discuss our implementation of some heuristics described in the literature for constructing BSPs. Note that some of the heuristics discussed below were originally developed to construct BSPs for arbitrarily-oriented polygons in $\mathbb{R}^3$. All the algorithms work on the same basic principle: examine all the planes containing the rectangles in S_B and determine how "good" each plane is. Split B using the "best" plane and recurse. Our implementation refines the original descriptions of these heuristics in two respects: (i) At a node B, we first check whether S_B (recall that S_B is the set of rectangles in S clipped within B) contains a free rectangle; if it does, we apply the free cut containing that rectangle.[1] (ii) If there is more than one "best" plane, we choose one of the planes based on some local criteria. To complete the description of each heuristic, it suffices to describe how the heuristic measures how "good" a candidate plane is.

For a plane π, let f_π denote the number of rectangles in S_B intersected by π, f_π^+ the number of rectangles in S_B completely lying in the positive halfspace defined by π, and f_π^- the number of rectangles in S_B lying completely in the negative halfspace defined by π. We define the *occlusion factor* α_π to be the ratio of the total area of the rectangles in S_B lying in π to the area of π (when π is clipped within B), the *balance* β_π to be the ratio $\min\{f_\pi^+, f_\pi^-\}/\max\{f_\pi^+, f_\pi^-\}$ between the number of polygons that lie completely in each halfspace defined by π, and σ_π to be the *split factor* of π, which is the fraction of rectangles that π intersects, i.e., $\sigma_\pi = f_\pi/|S_B|$. We now discuss how each algorithm measures how good a plane is.

ThibaultNaylor: Thibault and Naylor [102] present three different heuristics (w is a positive weight that can be changed to tune the performance of the heuristics):

1. Pick a plane the minimises the function $|f_\pi^+ - f_\pi^-| + wf_\pi$. This measure tries to balance the number of rectangles on each side of π so that the height of the BSP is small and also tries to minimise the number of rectangles intersected by π.
2. Maximise the measure $f_\pi^+ \cdot f_\pi^- - wf_\pi$. This measure is very similar to the previous one, except that much more weight is given to constructing a balanced BSP.
3. Maximise the function $f_\pi^+ - wf_\pi$. Thibault and Naylor state in their paper

[1] Only Paterson and Yao's algorithm [82] originally incorporated the notion of free cuts.

that this measure is motivated by applications in Constructive Solid Geometry (see their paper for more details). This heuristic performed poorly in our experiments. We will not discuss it further.

In our experiments, we use $w = 8$, as suggested by Thibault and Naylor [102].

Airey: In his thesis, Airey [8] proposes to maximise a measure function that is a linear combination of a plane's occlusion factor, its balance, and its split factor:

$$0.5\alpha_\pi + 0.3\beta_\pi + 0.2\sigma_\pi.$$

Teller: Let $0 \le \tau \le 1$ be a real number. Teller [101] chooses the plane with the maximum occlusion factor α_π, provided $\alpha_\pi \ge \tau$. If there is no such plane, he chooses the plane with the minimum value of f_π. The intuition behind this algorithm is that planes that are "well-covered" are unlikely to intersect many rectangles. Further, if the data set contains many coplanar rectangles, planes containing such rectangles are likely to be used as cutting planes near the root of the BSP, thus constructing a BSP with a small number of nodes. We use the value $\tau = 0.5$ (as suggested by Teller [101]) in our implementation. If S has many coplanar polygons, Teller's algorithm is likely to construct a BSP with a small number of nodes. However, queries like ray shooting might be costly since such queries involve processing the (large number of) rectangles stored with each bisecting plane.

PatersonYao: We have implemented a refined version of the algorithm of Paterson and Yao. For a box B, let s_x (resp., s_y, s_z) denote the number of edges of the rectangles in S_B that lie in the interior of B and are parallel to the x-axis (resp., y-axis, z-axis). We define the measure of B to be $\mu(B) = s_x s_y s_z$. We make a cut that is perpendicular to the smallest set of segments and divides B into two boxes, each with measure at most $\mu(B)/4$. (Paterson and Yao prove that given any axis, we can find such a cut perpendicular to that axis [82].) We can show that this algorithm also produces BSPs of size $O(n\sqrt{n})$ for n rectangles, just like Paterson and Yao's original algorithm [82].

Our implementations of these algorithms are efficient in terms of running time because we exploit the fact that we are constructing BSPs for orthogonal rectangles.

After an initial sort, we determine the cut at any node in time linear in the number of the rectangles intersecting that node. If we were processing arbitrarily-oriented objects, computing a cut can take time quadratic in the number of objects.

Note that the algorithms we have implemented form a representative sample of the BSP algorithms proposed in the literature. We expect the performance of other known techniques to be similar to the ones we have implemented. For example, Naylor has proposed a technique that uses estimates of the costs incurred when the BSP is used to answer standard queries to control the construction of the BSP [73]. While his idea is new, the measure functions he uses to choose cutting planes are very similar to the ones used above. Cassen et al. [24] use genetic algorithms to construct BSPs. We have not compared our algorithms to theirs since they report that their algorithm takes hours to run even for data sets containing just a few hundred polygons.

5.2 Experimental Results

We have implemented the above algorithms and run them on the following data sets containing orthogonal rectangles:[2]

1. the FIFTH floor of Soda Hall containing 1677 rectangles,
2. the ENTIRE Soda Hall model with 8690 rectangles,
3. the Orange United Methodist CHURCH Fellowship Hall with 29988 rectangles,
4. the Sitterson Hall LOBBY with 12207 rectangles, and
5. SITTerson Hall containing 6002 rectangles.

We present three sets of results. These experiments were run on a Sun SPARCstation 5 running SunOS 5.5.1 with 64MB of RAM. For each set, we first discuss the experimental set-up and then present the performance of our algorithms.

[2]We discarded all non-orthogonal polygons from these data sets. The number of such polygons was very small.

5.2.1 Size of the BSP

Recall that in Chapter 4, we defined the size of a BSP to be the sum of the the number of nodes in the BSP and the total number of rectangles in S stored at all the nodes of the BSP. In our experiments, we use a slightly different and more realistic measure: we define the size of the BSP to be the sum of the number of *interior* nodes in the BSP and the total number of rectangles stored at all the nodes in the BSP. We use the number of interior nodes rather than the number of nodes since all the information about a leaf is captured by its parent and the cutting plane at the parent. Further, in the case of rectangles, the total number of faces of all dimensions stored at the nodes of the BSP is at most nine times the total number of rectangles stored at the nodes of the BSP. Note that the total number of rectangles stored in the BSP is the sum of the number of rectangles in the input and the number of times the input rectangles are fragmented by the BSP. Table 5.1 displays the size of the BSP and the total number of times the rectangles are fragmented by the cuts made by the BSP.

Data set	FIFTH	ENTIRE	CHURCH	LOBBY	SITT.
#rectangles	1677	8690	29988	12207	6002
Size of the BSP					
Rounds	**2744**	**14707**	45427	22225	9060
Teller	2931	14950	**33518**	**13911**	**7340**
PatersonYao	3310	22468	56868	30712	20600
Airey	3585	24683	41270	21753	19841
ThibaultNaylor1	6092	32929	65313	25051	10836
ThibaultNaylor2	3235	20089	58175	23159	12192
Number of Fragments					
Rounds	**113**	**741**	**838**	**475**	312
Teller	301	1458	873	514	**153**
PatersonYao	449	5545	12517	9642	6428
Airey	675	7001	5494	5350	8307
ThibaultNaylor1	1868	10580	13797	3441	1324
ThibaultNaylor2	262	2859	6905	1760	1601

Table 5.1: BSP sizes and number of fragments

Examining Table 5.1, we note that, in general, the number of fragments and size of the BSP scale well with the size of the data set. For the Soda Hall data sets (FIFTH and ENTIRE), algorithm Rounds creates the smallest number of fragments and constructs the smallest BSP. For the other three sets, algorithm Teller performs best in terms of BSP size. However, there are some peculiarities in the table. For example, for the CHURCH data set, algorithm Rounds creates a smaller number of fragments than algorithm Teller but constructs a larger BSP. We believe that this difference is explained by the fact that the 29998 rectangles in the CHURCH model lie in a total of only 859 distinct planes. Visualisation of the CHURCH model makes it clear that it contains many rectangles that lie in a small number of planes, each with large area. Since algorithm Teller makes cuts based on how much of a plane's area is covered by rectangles, it is reasonable to expect that the algorithm will "place" a lot of rectangles in cuts made close to the root of the BSP, thus leading to a BSP with a small number of nodes.

We further examined the issue of how well the performance of the algorithms scaled with the size of the data by running the algorithms on increasingly larger subsets of the data sets. We also "created" new large data sets by making translated and rotated copies of the original data sets. In Figure 5.1, we display the results of this experiment for the ENTIRE Soda Hall and the SITTERSON models. In these graphs, we do not display the curve for algorithm ThibaultNaylor1 since its performance is always worse than the performance of algorithm ThibaultNaylor2. The graphs show that the size of the BSP constructed by most algorithms increases linearly with the size of the data. The performance of algorithms Rounds and Teller is nearly identical for the Entire data set. However, algorithm Teller constructs a smaller BSP than algorithm Rounds for the Sitterson data set.

The time taken to construct the BSPs also scaled well with the size of the data sets. Algorithm Rounds took 11 seconds to construct a BSP for the FIFTH floor of Soda Hall and about 4.5 minutes for the church data set. Typically, algorithm PatersonYao took about 15% less time than algorithm Rounds while the heuristics (algorithms Airey, ThibaultNaylor, and Teller) took 2-4 times as much time as algorithm Rounds to construct a BSP. While the difference in time is negligible for small data sets, it can be considerable for large data sets. For example, for the data set obtained by placing 9 copies of the Sitterson model in a 3×3 array, algorithm Rounds took 13 minutes to construct a BSP while algorithm Teller took 51 minutes. The results for single copies of each data set are summarised in the following table.

	Fifth	Entire	Church	Lobby	Sitt.
#rectangles	1677	8690	29988	12207	6002
Rounds	11	58	273	111	57
Teller	16	130	1177	237	101
PatersonYao	9	55	238	82	59
Airey	30	232	1099	337	317
ThibaultNaylor1	28	272	2840	306	149
ThibaultNaylor2	13	97	1722	176	81

Table 5.2: Time (in seconds) taken to construct the BSP

5.2.2 Point location

In the point location query, we are given a set of random points and are asked to locate the leaf of the BSP that contains each point. We create the queries by generating random points that lie in the box B containing all the rectangles in S. We answer such queries by traversing the path from the root of the BSP that leads to the leaf that contains the query point.

We summarise the results for point location in Table 5.3. These results were highly correlated to the average heights of the trees. For example, algorithm Rounds constructed BSPs of average height between 11 and 16. The average cost of locating a random point in the BSP constructed by algorithm Rounds ranged between 10 and 15. It is worth pointing out that Algorithm ThibaultNaylor1 constructed BSPs about twice as deep as algorithm ThibaultNaylor2, bearing out our earlier intuition.

5.2.3 Ray shooting

Given a ray ρ oriented in a random direction, we are required to determine the first rectangle in S that is intersected by ρ or report that there is no such rectangle. We first locate the leaf ν containing the origin of ρ. Then we trace ρ through the leaves of the BSP as follows: we determine the point p where ρ intersects the boundary of ν. If p lies inside a rectangle in S (such a rectangle must be stored with the bisecting

	FIFTH	ENTIRE	CHURCH	LOBBY	SITT.
#rectangles	1677	8690	29988	12207	6002
Rounds	10.77	13.48	14.34	12.49	12.37
Teller	11.51	13.75	10.19	7.29	15.30
PatersonYao	10.81	13.12	12.98	11.32	13.23
Airey	10.52	13.62	12.87	8.23	14.15
ThibaultNaylor1	18.42	22.13	18.68	20.66	21.17
ThibaultNaylor2	11.38	14.19	14.50	13.75	13.08

Table 5.3: Random point location costs

plane of an ancestor of v or lie on the boundary of B), we report the rectangle as the answer to the query. Otherwise, we locate the other node w whose boundary p lies on by answering a point location query for p. If there is no such node, then p lies on the boundary of B and ρ does not intersect any rectangle. Otherwise, we continue tracing ρ at w. There are two components to the cost of answering the query with ρ: the number of nodes visited and the number of rectangles checked. We report the two factors separately below in Table 5.4. The actual cost of a ray shooting query is a linear combination of these two components; the exact form of the linear combination depends on the implementation.

There is an interesting tradeoff between these two costs. This tradeoff is most sharply noticeable for the CHURCH data set. Note that the average number of nodes visited to answer ray shooting queries in the BSP constructed by algorithm Teller is about a third the number visited in the BSP built by algorithm Rounds but the number of rectangles checked in the BSP constructed by algorithm Teller is about 10 times higher! This apparent discrepancy actually ties in with our earlier conclusion that algorithm Teller is able to construct a BSP with a small number of nodes for the CHURCH model because the rectangles in this model lie on a small number of distinct planes, as noted earlier. As a result, a ray shooting query does not visit too many nodes. However, whenever a point is checked to see if it lies in an input rectangle, a large number of rectangles are checked because each plane contains a large number of rectangles. This cost can be brought down by using an efficient data structure for range searching among rectangles. However, this change will increase the size of the

Data set	FIFTH	ENTIRE	CHURCH	LOBBY	SITT.
#rectangles	1677	8690	29988	12207	6002
#nodes visited					
Rounds	44.71	12.57	326.40	89.68	55.92
Teller	17.19	13.74	96.64	13.04	37.30
PatersonYao	40.06	11.85	531.47	49.83	83.12
Airey	24.02	13.31	170.26	10.59	129.99
ThibaultNaylor1	44.10	31.34	256.81	102.61	69.14
ThibaultNaylor2	44.56	14.20	298.54	78.40	59.85
#rectangles checked					
Rounds	5.78	3.03	49.60	2.02	19.24
Teller	12.08	11.02	4828.28	20.47	44.18
PatersonYao	4.05	5.71	5461.23	84.24	114.09
Airey	5.51	4.10	4757.91	11.94	27.55
ThibaultNaylor1	4.60	14.63	20.77	2.19	38.56
ThibaultNaylor2	5.14	7.59	28.54	2.67	7.33

Table 5.4: Ray shooting costs

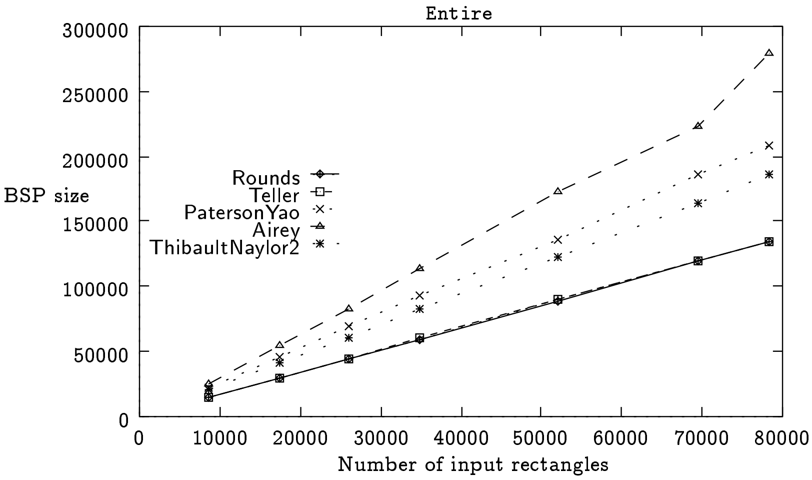
BSP itself. Determining the right combination needs further investigation.

5.3 Conclusions

Our experiments indicate that algorithms Rounds and Teller are the best techniques for constructing BSPs for orthogonal rectangles in $\mathbb{R}^3$. Algorithm Teller is best for applications like painter's algorithm in which the entire BSP is traversed. On the other hand, for queries such as ray shooting, it might be advisable to use algorithm Rounds since the size of the BSP constructed by this algorithm is not much more than the size for algorithm Teller but the query costs are better. It is possible that a better solution might be a combination of both algorithms. For example, one possibility is that we take rectangle areas into account while making cuts in the dividing stage of algorithm Rounds.

Note that we can change the performance of algorithms Teller, Airey and Thibault-Naylor by changing the internal weights used by these algorithms. It is possible to use a technique like simulated annealing to determine the best set of weights. However, it is likely that the time taken by such procedures will be prohibitive.

Clearly, there is a tradeoff between the amount of time spent on constructing the BSP and the size of the resulting BSP. Our experience suggests that while algorithm Teller constructs the smallest BSPs, algorithm Rounds will build BSPs that are not much larger, is likely to be fast in terms of execution, and will build compact BSPs that answer queries efficiently.



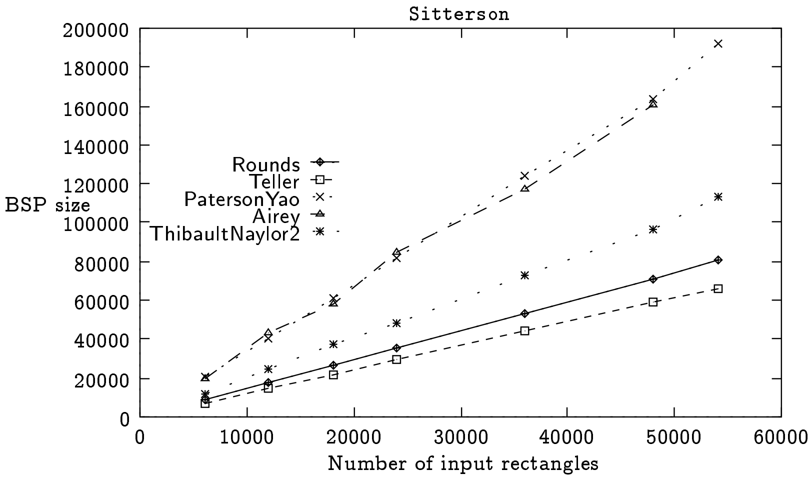


Figure 5.1: Graphs displaying BSP size vs. #rectangles in S.

Chapter 6

Binary Space Partitions for Triangles

In this chapter, we study the problem of computing a BSP for a set S of n triangles with pairwise-disjoint interiors in $\mathbb{R}^3$. Paterson and Yao [81] describe a randomised incremental algorithm that constructs a BSP of size $O(n^2)$ in expected time $O(n^3)$. They also show how to derandomise their algorithm without affecting its asymptotic running time. Their bound on the size of the BSP is optimal in the worst-case. It has been an open problem whether a BSP for n triangles in $\mathbb{R}^3$ can be constructed in near-quadratic time. Sub-quadratic bounds are known for special cases: Paterson and Yao's algorithm for orthogonal rectangles [82], our results for fat orthogonal rectangles (see Chapter 4), and de Berg's result for fat polyhedra [35]. However, none of these approaches lead to a near-quadratic algorithm for triangles in $\mathbb{R}^3$. The bottleneck in analyzing the expected running time of the Paterson-Yao algorithm is that no nontrivial bound is known on the number of vertices in the convex subdivision of $\mathbb{R}^3$ induced by the BSP constructed by the algorithm. Known techniques for analyzing randomised algorithms, such as the Clarkson-Shor framework [32] or backwards analysis [92], cannot be used to obtain a near-quadratic bound on this quantity, since the BSP constructed by the algorithm depends on the order in which triangles are processed.

In Section 6.1, we present a randomised algorithm that constructs a BSP for S of size $O(n^2)$ in $O(n^2 \log^2 n)$ expected time. Our algorithm is a variant of the Paterson-Yao algorithm. We construct the BSP for S in such a way that there is a close relationship between the BSP and the planar arrangement of lines supporting the edges of the xy-projections of the triangles in S. We use results on ε-nets [54] and on arrangements of lines [40] to bound the expected number of vertices in the convex subdivision of $\mathbb{R}^3$ induced by the BSP and the expected running time of the algorithm.

In Section 6.2, we consider the problem of whether we can use a suitable measure of geometric complexity to construct a BSP of size $o(n^2)$ for a set S of n triangles with pairwise-disjoint interiors in $\mathbb{R}^3$ (of course, the size of the BSP will depend on the measure of geometric complexity and will be $\Omega(n^2)$ in the worst-case). The construction that achieves the $\Omega(n^2)$ lower bound (we have already seen it in Chapter 4.6.2) uses two families of segments lying inside the box $[0, n+1] \times [0, n+1] \times [0, n^2+1]$: the segments belonging to one family lie along the lines $\{x = i, z = iy - \varepsilon/2 \mid 1 \le i \le n\}$, where $0 < \varepsilon \le 1/n^2$; these lines lie in the surface $z = xy - \varepsilon/2$. The segments in the second family are orthogonal to the ones in the first family and lie along the lines $\{y = i, z = ix + \varepsilon/2 \mid 1 \le i \le n\}$; the lines lie in the surface $z = xy + \varepsilon/2$. Intuitively, we can show that any BSP for this set of segments has $\Omega(n^2)$ size because these n segments come very "close" to each other at $\Omega(n^2)$ points. In particular, the xy-projections of the segments contain $\Omega(n^2)$ intersection points. Such configurations are very uncommon in practice. For example, urban landscapes and terrains have very few pairs of triangles with intersecting xy-projections.

Motivated by these observations, we measure the geometric complexity of a set S of triangles with pairwise-disjoint interiors in $\mathbb{R}^3$ by the number of intersections between the xy-projections of the triangles in S. If the number of intersections is k, we present a deterministic algorithm for constructing a BSP for S of size $O((n+k)\log^2 n)$ in time $O((n+k)\log^3 n)$; if $k \ll n^2$, the deterministic algorithm constructs a BSP whose size is much smaller than the BSPs constructed by Paterson and Yao's and our randomised algorithms. Another nice property of our deterministic algorithm is that the height of the BSP it constructs is $O(\log^3 n)$, which is useful for ray-shooting queries, for example. It was an open problem whether BSPs of near-quadratic size and poly-logarithmic height could be constructed for n triangles in $\mathbb{R}^3$. The height of the BSP constructed by the randomized algorithms (both ours and Paterson and Yao's) can be

$\Omega(n)$, e.g., if S is the set of faces of a convex polytope.

Finally, we study the problem of maintaining the BSP for a set of moving segments in the plane in Section 6.3. The problem can be formulated as follows: Let S be a set of n interior-disjoint segments in the plane, each moving along a continuous path. We want to maintain the BSP for S as the segments in S move. We assume that the segments move in such a way that they never intersect, except possibly at their endpoints. Most of the work to date deals with constructing a BSP for a set of "static" segments, which do not move. Paterson and Yao propose a randomized algorithm that constructs a BSP of $O(n \log n)$ size for a set of n segments in the plane [81]. They also propose a deterministic algorithm, based on a divide-and-conquer approach, that constructs a BSP of size $O(n \log n)$ in $O(n \log n)$ time [81]. Both of these algorithms are not "robust," in the sense that a small motion of one of the segments may cause many changes in the tree, or may cause non-local changes. Therefore, they are ill-suited for maintaining a BSP for a set of moving segments.

There have been a few attempts to update BSPs when the objects defining them move. Naylor describes a method to implement dynamic changes in a BSP, where the static objects are represented by a balanced BSP (computed in a preprocessing stage), and then the moving objects are inserted at each time step into the static tree [76]. Using the same assumption that moving objects are known *a priori*, Torres proposes the augmentation of BSPs with additional separating planes, which may localise the updates needed for deletion and re-insertion of moving objects in a BSP [103]. This approach tries to exploit the spatial coherence of the dynamic changes in the tree by introducing additional cutting planes. Chrysanthou suggests a more general approach, which does not make any distinction between static and moving objects [31]. By keeping additional information about topological adjacencies in the tree, the algorithm performs insertions and deletions of a node in a more localised way. But all these prior efforts boil down to deleting moving objects from their earlier positions and re-inserting them in their current positions after some time interval has elapsed. Such approaches suffer from the fundamental problem that it is very difficult to know how to choose the correct time interval size: if the interval is too small, then the BSP does not in fact change combinatorially, and the deletion/re-insertion is just wasted computation; if it is too big, then important intermediate changes to the BSP can be missed, which affect applications that use the tree.

Our algorithm, instead, treats the BSP as a *kinetic data structure*, a paradigm defined by Basch et al. [13]. We view the equations of the cuts made at the nodes of the BSP and the edges and faces of the subdivision induced by the BSP as functions of time. The cuts and the edges and faces of the subdivision change continuously with time. However, "combinatorial" changes in the BSP and in the subdivision (we precisely define this notion later) occur only at certain times. We explicitly take advantage of the continuity of the motion of the objects involved so as to generate updates to the BSP only when actual events cause the BSP to change combinatorially.

In Section 6.3, we describe a randomised kinetic algorithm for maintaining a BSP for moving segments in the plane, whose interiors remain disjoint at all the times during the motion. We also assume that the motions are *oblivious* to the random bits used by the algorithm; our algorithm chooses a random permutation of the segments at the beginning of time, and we assume that this permutation remains random irrespective of the segment motions. Following Basch et al. [13], we assume that each moving segment has a posted flight plan that gives full or partial information about the segment's current motion. Whenever a flight plan changes (possibly due to an external agent), our algorithm is notified and it updates a global event queue to reflect the change. We first derive a randomised algorithm for computing a BSP for a set of static segments, which combines ideas from Paterson and Yao's randomised and deterministic algorithms, but is also robust, in the sense described earlier. The "combinatorial structure" of the BSP constructed by this algorithm changes only when the x-coordinates of a pair of segment endpoints, among a certain subset of $O(n)$ pairs, become equal. We show that under the above assumption on the segment motions, the BSP can be updated in $O(\log n)$ expected time at each such event. We also show that if k of the segments of S move along "pseudo-algebraic" paths, and the remaining segments of S are stationary, then the expected number of changes in the BSP is $O(kn \log n)$. As far as we know, this is the first nontrivial algorithm for maintaining a BSP for moving segments in the plane.

A feature of the BSPs constructed by our algorithm is that the region $\mathcal{R}_v$ associated with each node v is a "cylindrical" cell in the sense that $\mathcal{R}_v$ may contain top and bottom faces that are contained in objects belonging to S, but all other faces are vertical (i.e., faces parallel to the z-axis). In particular, for each node v in the BSP constructed by the deterministic algorithm for triangles in $\mathbb{R}^3$, $\mathcal{R}_v$ has four vertical faces and two of these faces are perpendicular to the x-axis.

Before proceeding further, we recapitulate the definition of the BSP. A *binary space partition* $\mathcal{B}$ for a set S of triangles with pairwise-disjoint interiors in $\mathbb{R}^3$ is a tree defined as follows: Each node v in $\mathcal{B}$ is associated with a polytope $\mathcal{R}_v$ and the set of triangles $S_v = \{s \cap \mathcal{R}_v \mid s \in S\}$ that intersect $\mathcal{R}_v$. The polytope associated with the root is $\mathbb{R}^3$ itself. If S_v is empty, then node v is a leaf of $\mathcal{B}$· Otherwise, we partition $\mathcal{R}_v$ into two convex polytopes by a *cutting plane* H_v. We refer to the polygon $H_v \cap \mathcal{R}_v$ as the *cut* made at v. At v, we store the equation of H_v and the set $\{s \mid s \subseteq H_v, s \in S_v\}$, the subset of triangles in S_v that lie in H_v. If H_v^+ denotes the positive halfspace and H_v^- the negative halfspace bounded by H_v, the polytopes associated with the left and right children of v are $\mathcal{R}_v \cap H_v^-$ and $\mathcal{R}_v \cap H_v^+$, respectively. The left subtree of v is a BSP for the set of triangles $S_v^- = \{s \cap H_v^- \mid s \in S_v\}$ and the right subtree of v is a BSP for the set of triangles $S_v^+ = \{s \cap H_v^+ \mid s \in S_v\}$. The size of $\mathcal{B}$ is the sum of the number of nodes in $\mathcal{B}$ and the total number of triangles stored at all the nodes in $\mathcal{B}$. Note that we can modify this definition easily if S is a set of n segments with pairwise-disjoint interiors in the plane.

At a node v of $\mathcal{B}$, the cutting plane H_v may not intersect any triangles in S_v and (i) may support a triangle $s \in S_v$ such that the portion of H_v that lies in the interior of $\mathcal{R}_v$ is contained in s, i.e., $H_v \cap \mathcal{R}_v = s$ or (ii) partition S_v into two non-empty (and disjoint) sets. We refer to such a cutting plane H_v as a *free* cut; the triangle s is said to be free with respect to $\mathcal{R}_v$. Free cuts play a critical role in preventing excessive fragmentation of the triangle in S, and thus in keeping the size of $\mathcal{B}$ small.

6.1 BSPs for Triangles: A Randomised Algorithm

In this section we describe a randomized algorithm for constructing a BSP $\mathcal{B}$ of expected size $O(n^2)$ for a set of n triangles with pairwise-disjoint interiors in $\mathbb{R}^3$. The expected running time of the algorithm is $O(n^2 \log^2 n)$. We describe the algorithm in Section 6.1.1 and analyze its performance in Section 6.1.2.

6.1.1 Our algorithm

We start with some definitions. For an object s in $\mathbb{R}^3$, let s^* denote the xy-projection of s. Let E be the set of edges of the triangles in S, and let E^* denote the set $\{e^* \mid e \in E\}$. Let $\mathcal{L}$ be the set of lines in the xy-plane supporting the edges in E^*. We choose a random permutation $\langle \ell_1, \ell_2, \ldots, \ell_{3n} \rangle$ of $\mathcal{L}$, and add the lines one by one in this order to compute $\mathcal{B}$. Let $\mathcal{L}^i = \{\ell_1, \ell_2, \ldots \ell_i\}$. The algorithm works in $3n$ stages. In the ith stage, we add ℓ_i and construct a top subtree $\mathcal{B}^i$ of $\mathcal{B}$ by refining the leaves of $\mathcal{B}^{i-1}$; $\mathcal{B}^0$ consists of one node (corresponding to $\mathbb{R}^3$) and $\mathcal{B}^{3n}$ is $\mathcal{B}$. As usual, we associate a convex polytope $\mathcal{R}_v$ with each node v of $\mathcal{B}^{i-1}$. If v is a leaf of $\mathcal{B}^{i-1}$ and no triangle in S intersects the interior of $\mathcal{R}_v$, i.e., $S_v = \emptyset$, then v is a leaf of $\mathcal{B}$ and we do not refine it further. Otherwise, we partition $\mathcal{R}_v$ into two cells; these two cells are leaves of $\mathcal{B}^{i+1}$.

Before describing the ith stage of the algorithm in detail, we explain the structure of $\mathcal{B}^i$. We need a few definitions first. At a node v of $\mathcal{B}$, the cutting plane H_v may support a triangle $s \in S$ such that $H_v \cap \mathcal{R}_v \subseteq s$, i.e., the portion of H_v that lies in the interior of $\mathcal{R}_v$ is contained in s. Such a cutting plane is referred to as a *free cut* and s is called a *free triangle*. We say that a leaf v of $\mathcal{B}^i$ (or the cell $\mathcal{R}_v$) is *active* if a triangle in S intersects the interior of $\mathcal{R}_v$ (i.e. $S_v \neq \emptyset$); similarly, we say that a face f in the line arrangement $\mathcal{A}(\mathcal{L}^i)$ is *active* if a segment in E^* intersects the interior of f. For each active leaf v in $\mathcal{B}^i$, the algorithm ensures that $\mathcal{R}_v$ satisfies the following properties:

(P1) If a triangle $s \in S$ intersects the interior of $\mathcal{R}_v$, then the boundary of s also intersects the interior of $\mathcal{R}_v$.

(P2) The cell $\mathcal{R}_v$ is a vertical section of the cylinder $\{(p, z) \mid p \in f, z \in \mathbb{R}\}$, for exactly one active face f of $\mathcal{A}(\mathcal{L}^i)$; the vertical section may be truncated by triangles of S at the top and bottom. See Figure 6.1.

In order to execute each stage efficiently, we maintain the following additional information:

(i) For each active cell $\Delta \in \mathcal{B}^i$, we store the subset $S_\Delta \subseteq S$ of triangles that intersect the interior of Δ.

(ii) We maintain the arrangement $\mathcal{A}(\mathcal{L}^i)$ as a planar graph [40]. For each active face $f \in \mathcal{A}(\mathcal{L}^i)$, we maintain the set $\boldsymbol{\Delta}(f)$ of those active cells in $\mathcal{B}^i$ that lie

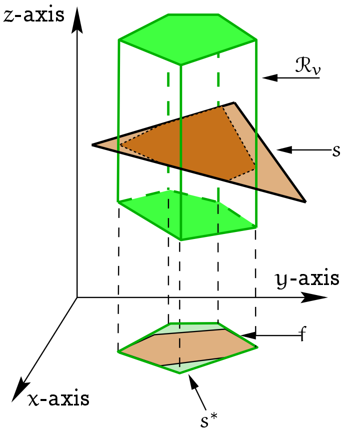


Figure 6.1: An active face f and an active cell $\mathcal{R}_v \in \boldsymbol{\Delta}(f)$ that is a vertical section of the cylinder erected on f. The boundary of triangle s intersects $\mathcal{R}_v$ and the boundary of s^* intersects f.

inside the cylinder $\{(p, z) \mid p \in f, z \in \mathbb{R}\}$. Note that by Properties (P1) and (P2), a face $f \in \mathcal{A}(\mathcal{L}^i)$ is active if and only if $\boldsymbol{\Delta}(f) \neq \emptyset$.

We now describe the ith stage in detail. In this stage, we make a vertical cut along the vertical plane H_i supporting ℓ_i, followed by a number of free cuts as follows: Let H_i^+ (resp., H_i^-) be the positive (resp., negative) halfspace supported by H_i.

1. We trace ℓ_i through the faces of $\mathcal{A}(\mathcal{L}^{i-1})$. For each face $f \in \mathcal{A}(\mathcal{L}^{i-1})$ intersected by ℓ_i, we use ℓ_i to split f into two faces f^+ and f^-. See Figure 6.2(a). Next, we partition each active cell $\Delta \in \boldsymbol{\Delta}(f)$ into two cells $\Delta^+ = \Delta \cap H_i^+$ and $\Delta^- = \Delta \cap H_i^-$ (see Figure 6.2(b)) and execute the following two steps on Δ:

2. We compute the set $S_{\Delta^+} \subseteq S_\Delta$ of triangles that intersect the interior of Δ^+. We also compute the set $F_{\Delta^+} \subseteq S_{\Delta^+}$ of triangles whose boundaries do not cross Δ^+. Similarly, we compute the sets S_{Δ^-} and F_{Δ^-} for Δ^-.

3. We split Δ^+ into a set Ψ of $|F_{\Delta^+}| + 1$ cells by making free cuts along each of the triangles in F_{Δ^+}. The cells in Ψ can be ordered by z-coordinate. Since the triangles in S are pairwise-disjoint, each triangle $s \in S_{\Delta^+} \setminus F_{\Delta^+}$ intersects a unique cell $\Delta' \in \Psi$. We compute Δ' by performing a binary search, and add s to $S_{\Delta'}$.

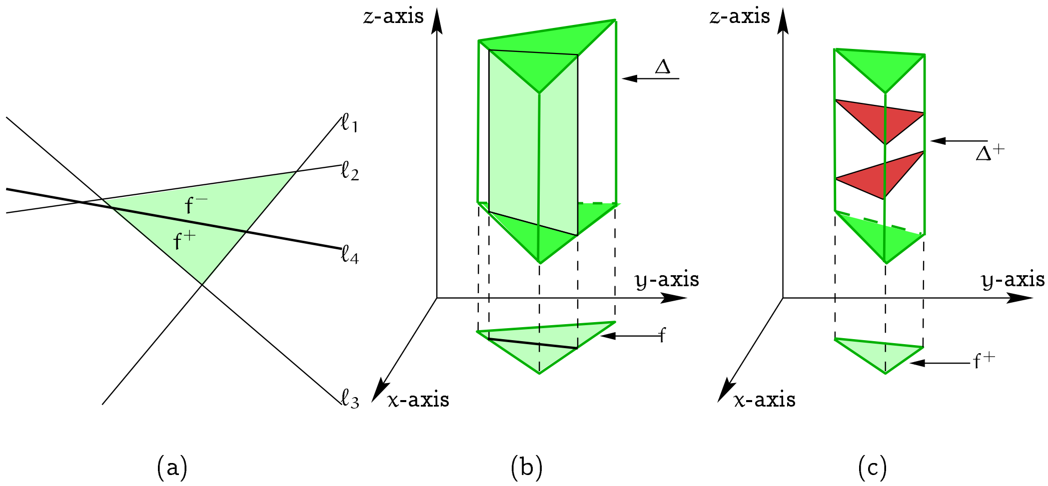


Figure 6.2: (a) Tracing ℓ_4 (the thick line) through the faces of $\mathcal{A}(\mathcal{L}^3)$. The face f is shaded. (b) Splitting cell $\Delta \in \mathbf{\Delta}(f)$ by the vertical plane containing ℓ_4. (c) The free cuts in F_{Δ^+} are ordered by z-coordinate.

For each cell $\Delta' \in \Psi$, we add Δ' to the set $\mathbf{\Delta}(f^+)$ if $S_{\Delta'} \neq \emptyset$. Next, we repeat the same procedure for Δ^-.

Whenever we split a three-dimensional cell into two, we add two children to the corresponding node in $\mathcal{B}^{i-1}$ and store the necessary information with the newly created nodes. The resulting tree is $\mathcal{B}^i$. The cuts made in Step 3 ensure that $\mathcal{B}^i$ satisfies property (P1). $\mathcal{B}^i$ satisfies property (P2) since the cuts made in Step 1 are vertical. Note that a triangle $s \in S$ does not intersect the interior of any cell after the three lines supporting the edges of s^* have been processed.

Remark: The free cuts made in Step 3 are crucial in keeping the size of the BSP quadratic. Instead, if we simply erect vertical planes as we do in the algorithm, and make cuts along a triangle $s \in S$ only when all three lines supporting the xy-projections of the edges of s have been added, then there are instances of input triangles for which our algorithm will construct a BSP of $\Omega(n^3)$ size regardless of the initial random permutation.

6.1.2 Analysis of the algorithm

We first bound the expected size of $\mathcal{B}$. A similar proof is used by Paterson and Yao [81] to analyse their randomised algorithm for constructing BSPs for triangles in $\mathbb{R}^3$. The cuts made by the algorithm partition each triangle in S into a number of sub-polygons; each such sub-polygon is contained in the cutting plane of some node in $\mathcal{B}$ and is stored at that node. Let $\nu(S)$ be the total number of polygons stored at the nodes of $\mathcal{B}$. The following lemma bounds the size of $\mathcal{B}$ in terms of $\nu(S)$.

Lemma 6.1.1 *The size of* $\mathcal{B}$ *is at most* $17\nu(S)$.

Proof: Recall that the size of $\mathcal{B}$ is defined to be the sum of the number of nodes in $\mathcal{B}$ and the total number of triangles stored at all the nodes in $\mathcal{B}$. To bound the sizeof $\mathcal{B}$, we count the number of nodes in $\mathcal{B}$ and then add $\nu(S)$. Let $\nu(E)$ be the number of segments into which the edges of the triangles in S are partitioned by the cuts in $\mathcal{B}$.

We first bound the number of leaves in $\mathcal{B}$ in terms of $\nu(S)$ and $\nu(E)$. Let v be the parent of a leaf in $\mathcal{B}$. Either a free cut or a vertical cut through an edge of a triangle in S is made at v. This implies that the number of leaves in $\mathcal{B}$ is at most $2(\nu(S)+\nu(E))$. To bound this quantity, for each triangle $s \in S$, consider the arrangement $\mathcal{A}_s$ on s formed by the intersection of s and the cutting planes in $\mathcal{B}$. Let e_s be the number of edges in $\mathcal{A}_s$ that are portions of the edges of s and let f_s be the number of faces in $\mathcal{A}_s$. Since at most three edges of the boundary of a face in $\mathcal{A}_s$ are also portions of the edges of s, we have $e_s \leq 3f_s$. Summing over all triangles $s \in S$, we have $\nu(E) \leq 3\nu(S)$. Hence, the number of leaves in $\mathcal{B}$ is at most $8\nu(S)$, which implies that the number of nodes in $\mathcal{B}$ is at most $16\nu(S)$, thus proving the lemma. □

Thus, it suffices to bound the expectation $E[\nu(S)]$ to bound the expected size of $\mathcal{B}$. To that end, we count the expected number of new sub-polygons created in the ith stage, and sum the result over all stages. We bound the number ν_s^i of new sub-polygons into which a triangle $s \in S$ is partitioned by the cuts made in the ith stage, and sum the resulting bound over all triangles in S. Note that the vertical cuts made in the ith stage are contained in the vertical plane H_i containing ℓ_i.

Fix a triangle $s \in S$. For $1 \leq k \leq i$, let $\lambda_k = H_k \cap s$ and let Λ be the set of resulting segments. Note that the endpoints of each λ_k lie on the boundary of s. To

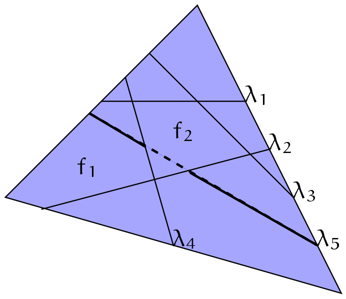


Figure 6.3: The arrangement $\mathcal{A}(\{\lambda_1, \lambda_2, \lambda_3, \lambda_4\})$ on triangle s (the shaded triangle). The face f_1 is a boundary face and the face f_2 is an interior face. The segment λ_5 does not partition f_2.

calculate $E\left[\nu_s^i\right]$, consider the segment arrangement $\mathcal{A}(\Lambda)$ on s. We call a face of $\mathcal{A}(\Lambda)$ a *boundary* face if it is adjacent to an edge of s; otherwise, it is an *interior* face. See Figure 6.3. Recall that for a leaf $\nu \in \mathcal{B}^{i-1}$, we partition the cell $\mathcal{R}_\nu$ only if $\mathcal{R}_\nu$ is active. Property (P1) implies that the cuts made in the $(i-1)$th stage do not cross the interior of any interior face of $\mathcal{A}(\Lambda)$, since such a face cannot intersect the interior of any active cell $\mathcal{R}_\nu$. Hence, ν_s^i is the number of boundary faces of $\mathcal{A}(\Lambda)$ that are intersected by λ_i. (If property (P1) did not hold, ν_s^i would be *all* the regions of $\mathcal{A}(\Lambda^i)$ that are intersected by λ_i.)

For $1 \leq k \leq i$, let $\mu(\Lambda, k)$ denote the number of boundary faces in the arrangement $\mathcal{A}(\Lambda \setminus \{\lambda_k\})$ that are intersected by λ_k. Observe that the sum $\sum_{1 \leq k \leq i} \mu(\Lambda, k)$ equals the total number of edges bounding the boundary faces of $\mathcal{A}(\Lambda)$. By a result of Bern et al. [19], the total number of edges of the boundary faces of $\mathcal{A}(\Lambda)$ is $O(i)$. Hence,

$$\sum_{1 \leq k \leq i} \mu(\Lambda, k) = O(i).$$

Since ℓ_i is chosen randomly from the set $\mathcal{L}^i$, λ_i can be any of the lines $\lambda_1, \lambda_2, \ldots, \lambda_i$ with equal probability. Therefore,

$$E\left[\nu_s^i\right] = \frac{1}{i} \sum_{1 \leq k \leq i} \mu(\Lambda, k) = O(1).$$

Hence, the total number of pieces created in the ith stage is $O(n)$. Summing over i, we find that the total number of sub-polygons into which the triangles in S are partitioned into over the course of the entire algorithm is $O(n^2)$. The following lemma is immediate:

Lemma 6.1.2 *The expected size of the BSP constructed by the algorithm is* $O(n^2)$.

Remark: Since each cell $\mathcal{R}_v \in \mathcal{B}$ is cylindrical, each vertex p of $\mathcal{R}_v$ is contained in one of the triangles s that contains the non-vertical faces of $\mathcal{R}_v$. In fact, p is a vertex of the arrangement $\mathcal{A}(\{\lambda_1, \lambda_2, \ldots, \lambda_{3n}\})$. Thus, the above argument implies that the expected value of the total number of vertices of the nodes of $\mathcal{B}$ is also $O(n^2)$. However, the height of $\mathcal{B}$ can be $\Omega(n)$, e.g., if the triangles in S form a convex polytope.

Before analysing the running time of the algorithm, we establish a relation between the projected edges intersecting an active face $f \in \mathcal{A}(\mathcal{L}^{i-1})$ and the triangles intersecting the cells in $\boldsymbol{\Delta}(f)$. For such an active face f, let k_f be the number of projected edges in E^* that intersect the interior of f. By Property (P1), if a triangle $s \in S$ intersects the interior of a cell $\Delta \in \boldsymbol{\Delta}(f)$, i.e., $s \in S_\Delta$, then the boundary of s also intersects the interior of Δ. Therefore, an edge of s^* intersects the interior of f. Since s intersects the interior of only one cell in $\boldsymbol{\Delta}(f)$, we obtain

$$\sum_{\Delta \in \boldsymbol{\Delta}(f)} |S_\Delta| \leq k_f. \tag{6.6.1}$$

We now analyze the expected running time of the algorithm. We count the time spent during the ith stage in inserting the line ℓ_i and then add this time over all stages of the algorithm. The zone theoreom [28, 40] implies that in Step 1 of the algorithm, we spend $O(i)$ time in tracing ℓ_i through $\mathcal{A}(\mathcal{L}^{i-1})$. While processing an active face f of $\mathcal{A}(\mathcal{L}^{i-1})$ that intersects ℓ_i, for each cell $\Delta \in \boldsymbol{\Delta}(f)$, we spend $O(1)$ time in Step 1 and $O(|S_\Delta|)$ time in Step 2. In Step 3, for each triangle $s \in S_{\Delta^+} \setminus F_{\Delta^+}$, we spend $O(\log |F_{\Delta^+}|)$ time in the binary search used to find the cell in the set Ψ that intersects s. Hence, the total time spent in Step 3 for the face f is $O(|S_\Delta| \log |S_\Delta|)$. Thus, (6.6.1) implies that the total time spent in processing f is

$$\sum_{\Delta \in \boldsymbol{\Delta}(f)} O(|S_\Delta| \log |S_\Delta|) = O(k_f \log k_f).$$

Let Z be the set of all active faces of $\mathcal{A}(\mathcal{L}^{i-1})$ that are intersected by ℓ_i. The total time spent in the ith stage is

$$\sum_{f \in Z} O(k_f \log k_f).$$

We now bound this sum. If we denote the number of vertices on the boundary of a face f by $|f|$, then by the result of Bern et al. [19], we have $\sum_{f\in Z} |f| = O(i)$. Consider the vertical decomposition $\mathcal{A}^{\|}(\mathcal{L}^{i-1})$ of $\mathcal{A}(\mathcal{L}^{i-1})$. Each face $f \in \mathcal{A}(\mathcal{L}^{i-1})$ is decomposed into $O(|f|)$ trapezoids in $\mathcal{A}^{\|}(\mathcal{L}^{i-1})$. By standard random-sampling arguments, the expected number of edges in E^* that intersect the interior of any such trapezoid is $O((n \log i)/i)$. This implies that for a face $f \in \mathcal{A}(\mathcal{L}^{i-1})$, the expected value of k_f is $O(|f|(n \log i)/i)$. Hence, the expected time spent in the ith stage is

$$\sum_{f \in Z} O(k_f \log k_f) = \sum_{f \in Z} O\left(|f| \left(\frac{n \log i}{i}\right) \log n\right) = O(n \log^2 n),$$

which implies the following theorem:

Theorem 6.1.1 *Let S be a set of n non-intersecting triangles in $\mathbb{R}^3$. We can compute a BSP for S of expected size $O(n^2)$ in expected time $O(n^2 \log^2 n)$.*

6.2 BSPs for Triangles: A Deterministic Algorithm

In this section, we describe a deterministic algorithm for computing a BSP for a set S of n triangles in $\mathbb{R}^3$. As in the previous section, let E denote the set of edges of triangles in S, and let $E^* = \{e^* \mid e \in E\}$ be the set of xy-projections of the edges in E. Let k be the number of intersections between the edges in E^*. Our algorithm constructs a BSP $\mathcal{B}$ of size $O((n+k)\log^2 n)$ in $O((n+k)\log^3 n)$ time.

As in the previous section, each node v of $\mathcal{B}$ is associated with a cylindrical cell $\mathcal{R}_v$, but the top and bottom faces of $\mathcal{R}_v$ are now trapezoids. Let $\mathcal{R}_v^*$ denote the xy-projection of the top (or bottom) face of $\mathcal{R}_v$; two of the edges of $\mathcal{R}_v^*$ are parallel to the y-axis. Before presenting our algorithm, we give some definitions.

Let E_v^* be the set of segments in E^* that intersect the interior of $\mathcal{R}_v^*$ and are clipped within $\mathcal{R}_v^*$. A segment $\gamma \in E_v^*$ is called *anchored* if its endpoints lie on the two parallel edges of $\mathcal{R}_v^*$ and if it does not intersect any other segment of E_v^*. Figure 6.4 shows an illustration. The anchored segments in E_v^* can be linearly ordered by y-coordinate

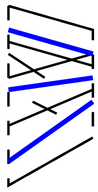


Figure 6.4: Anchored segments in E_v^* (these segments are drawn thick).

(since they are disjoint). Let A_v be the set of anchored segments in E_v^* and let P_v be the set of intersection points between the segments of E_v^*.

Let $F_v \subseteq S_v$ the set of all free triangles in S_v. Recall that a triangle $s \in S_v$ is free with respect to $\mathcal{R}_v$ if no edge of s intersects the interior of $\mathcal{R}_v$; s partitions $\mathcal{R}_v$ into two cylindrical cells. Since $\mathcal{R}_v$ is a cylindrical cell, the triangles in F_v can be sorted by their z-coordinates.

6.2.1 Our algorithm

Our algorithm constructs $\mathcal{B}$ in a top-down fashion by maintaining a top subtree of $\mathcal{B}$. A leaf v of the subtree is *active* if $S_v \neq \emptyset$. Note that v is active even if S_v contains free triangles; in Section 6.1, if a leaf v is active, S_v does contain any free triangles. We store the set of all active leaves of the current subtree in a list. For each active leaf v, we maintain the sets P_v, A_v, F_v, and S_v. Note that we can easily compute the set E_v^* from S_v. Before we begin constructing $\mathcal{B}$, we compute all k intersection points of E^* in $O((n+k)\log n)$ time using the Bentley-Ottman sweep-line algorithm [16, 84].

At each step of the algorithm, we choose an active leaf v, compute the cutting plane H_v, and use H_v to split $\mathcal{R}_v$ into two cells $\mathcal{R}_w$ and $\mathcal{R}_z$. If S_w (resp., S_z) is nonempty, we mark w (resp., z) as being active. Before describing how we compute H_v, we specify how we determine the sets P_w, F_w, S_w, and A_w (the procedure is symmetric for z):

P_w: Let $p \in P_v$ be the intersection point of $e_1{}^*$ and $e_2{}^*$, where e_1 and e_2 are triangle edges; $p \in P_w$ if both e_1 and e_2 intersect $\mathcal{R}_w$ and p is contained in $\mathcal{R}_w^*$.

F_w: Let s be a triangle in S_v. If s intersects $\mathcal{R}_w$ but none of the edges of s intersects the interior of $\mathcal{R}_w$, then $s \in F_w$.

S_w: Let s be a triangle in S_v. If an edge of s intersects the interior of $\mathcal{R}_w$, then $s \in S_w$.

A_w: Let $e^* \in E_v^*$, where e is an edge of a triangle in S_v. There are two cases to consider: (i) If $e^* \in A_v$ and e intersects $\mathcal{R}_w$, then $e^* \in A_w$. (ii) If $e^* \notin A_v$, e intersects $\mathcal{R}_w$, and $e^* \cap P_w = \emptyset$, then $e^* \in A_w$. To detect the second case, for each edge $e^* \in E_v^*$, we store the set of points $e^* \cap P_v$ that are formed by the intersection of e^* and other segments in E_v^*.

It is clear that these sets can be computed for both w and z in $O(|P_v| + |F_v| + |S_v| + |A_v|)$ time.

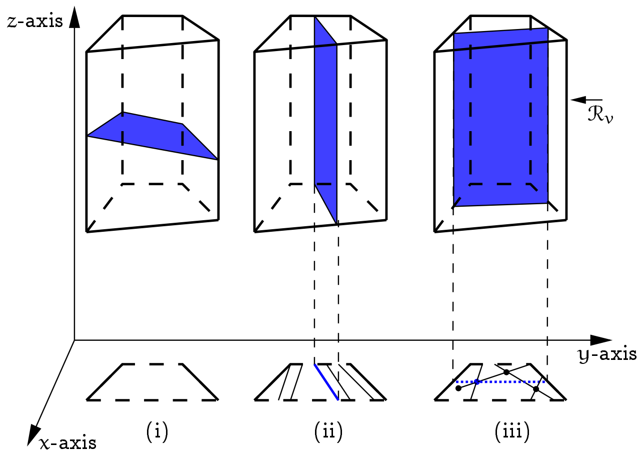


Figure 6.5: Cuts made in the deterministic algorithm: (i) Free cut, (ii) Cut parallel to the z-axis through an anchored segment, and (iii) Cut parallel to the yz-plane through a vertex of $\mathcal{A}(E_v^*)$.

Our algorithm uses three kinds of cuts: A *point* cut is a plane perpendicular to the x-axis passing through a vertex of the arrangement $\mathcal{A}(E_v^*)$, an *edge* cut is a vertical plane erected on an anchored segment in A_v, and a *face* cut is a plane containing a triangle in S (all face cuts will be free cuts). See Figure 6.5. We choose the cutting plane H_v as follows:

1. $F_v \neq \emptyset$: Recall that the triangles in F_v are totally ordered by z-coordinate. We set H_v to be the face cut containing the median triangle s of F_v. See Figure 6.5(i). Since s does not intersect any triangle of S_v, each triangle of $S_v \setminus \{s\}$ belongs to

either S_w or S_z. The free triangles in F_v and the anchored segments in A_v can be similarly partitioned.

2. $F_v = \emptyset$ and $A_v \neq \emptyset$: Recall that the anchored segments are totally ordered by y-coordinate. We set H_v to be the edge cut induced by the median anchored segment in A_v. See Figure 6.5(ii). Note that the anchored segments of A_v can be partitioned between A_w and A_z since do not intersect H_v.

3. $F_v = \emptyset, A_v = \emptyset$: We set H_v to be the point cut through the vertex in P_v with the median y-coordinate. See Figure 6.5(iii).

6.2.2 Analysis of our algorithm

We now analyse the algorithm. We first bound the size of $\mathcal{B}$, then the running time of the algorithm, and finally the height of $\mathcal{B}$.

Let v be a node in $\mathcal{B}$ such that P_v contains p intersection points, A_v contains a anchored segments, and F_v contains f free triangles; clearly S_v contains at most $O(p + a + f)$ triangles. Let $\mathcal{B}_v$ denote the subtree of $\mathcal{B}$ rooted at v. Set

$$S(p, a, f) = \max_{v} |\mathcal{B}_v|,$$

where the maximum is taken over all nodes v with $|P_v| = p, |A_v| = a$, and $|F_v| = f$. We bound $S(p, a, f)$ by setting up a recurrence for it. Suppose H_v, the cutting plane at v, partitions $\mathcal{R}_v$ into two cells $\mathcal{R}_w$ and $\mathcal{R}_w$. Let $p_w = |P_w|, a_w = |A_w|$, and $f_w = |F_w|$; define p_z, a_z, and f_z similarly. Note that P_w and P_z are disjoint subsets of P_v; therefore, $p_w + p_z \leq p$.[1] We consider three cases:

$f \neq 0$: H_v is a free cut containing the median free triangle in F_v. Since H_v is a free cut, it does not intersect any triangles in S_v. Hence, we have $a_w + a_z = a$, and $f_w, f_z \leq f/2$.

$f = 0, a \neq 0$: H_v is an edge cut erected on the median anchored segment in A_v; therefore $a_v, a_z \leq a/2$. Since H_v is an edge cut, it may intersect triangles in S_v, creating free triangles in F_w and F_z. However, there are $O(p + a)$ triangles in S_v; therefore, $f_w + f_z = O(p + a)$.

[1] It is possible that a point $p \in P_v$ is not an element of either P_w or P_z. This happens, for example, when p is the intersection of two projected triangle edges $e_1{}^*$ and $e_2{}^*$ but only e_1 intersects $\mathcal{R}_w$.

$f = 0, a = 0$: H_v is a point cut defined by the vertex in P_v with median y-coordinate, which implies that $p_w, p_z \leq p/2$. Since H_v may intersect triangles in S_v, both $\mathcal{R}_w$ and $\mathcal{R}_z$ can contain anchored segments and free triangles. Since there are $O(p)$ edges in E_v^* and $O(p)$ triangles in S_v, we have $a_w, a_z, f_w, f_z = O(p)$.

In the first case, the size of node v is 2, since the free triangle inducing H_v is stored at v. In the other two cases, the size of node v is 1. The above discussion implies that we can write the following recurrence for $S(p, a, f)$:

$$S(p, a, f) = S(p_w, a_w, f_w) + S(p_z, a_z, f_z) + O(1), \tag{6.6.2}$$

where $p_w + p_z \leq p$, and

1. $a_w + a_z = a$, and $f_w, f_z \leq f/2$, if $f \neq 0$;
2. $a_v, a_z \leq a/2$, and $f_w + f_z = O(p + a)$, if $f = 0, a \neq 0$;
3. $p_w, p_z \leq p/2$, and $a_w, a_z, f_w, f_z = O(p)$, if $f = 0, a = 0$.

Using mathematical induction, we can prove that the solution to this recurrence is

$$S(p, a, f) = O(p \log^2 p + (p + a) \log a + f).$$

Since the root node v of $\mathcal{B}$ has $n + k$ intersection points, no anchored segments, and no free triangles, we obtain the following lemma:

Lemma 6.2.1 *The size of* $\mathcal{B}$ *is* $O((n + k) \log^2 n)$.

We now analyse the running time of the algorithm. As we have noted earlier, at each node v, we can choose H_v and perform the operations to split v in $O(p + a + f)$ time. If $T(p, a, f)$ denotes the maximum time taken by our algorithm to construct the subtree of $\mathcal{B}$ rooted at a node v with $|P_v| = p, |A_v| = a$, and $|F_v| = f$ (the maximum is taken over all such nodes v), we have

$$T(p, a, f) = T(p_w, a_w, f_w) + T(p_z, a_z, f_z) + O(p + a + f), \tag{6.6.3}$$

where $p_w, a_w, f_w, p_z, a_z, f_z$ satisfy the same conditions as in (6.6.2). Using mathematical induction, we can prove that the solution to the above recurrence is

$$T(p, a, f) = O(p \log^3 p + (p + a) \log(p + a) \log a + (p + a + f) \log f).$$

Thus, we obtain the following lemma:

Lemma 6.2.2 *The time taken by our algorithm to construct* $\mathcal{B}$ *is* $O((n+k)\log^3 n)$.

We complete the analysis by bounding the height of $\mathcal{B}$. Since we always make the median point cut, along any root-to-leaf path π in $\mathcal{B}$, there are $O(\log n)$ nodes where a point cut is made. Let u be a node in π where a point cut is made and let v be the next node in π where a point cut is made. Since we always make the median edge cut, there are $O(\log n)$ nodes in π between u and w where an edge cut is made. Similarly, if u' is a node in π where an edge cut is made and w' is the next node in π where an edge cut is made, there are $O(\log n)$ nodes in π between u' and w' (a free cut is made at all such nodes). Hence, the length of π is $O(\log^3 n)$, which implies that the height of $\mathcal{B}$ is $O(\log^3 n)$.

Lemma 6.2.3 *The height of* $\mathcal{B}$ *is* $O(\log^3 n)$.

Combining the last three lemmas, we state the main result of this section:

Theorem 6.2.1 *Let* S *be a set of* n *triangles in* $\mathbb{R}^3$*, and let* k *be the number of intersection points of the* xy*-projections of the edges of* S*. We can compute a BSP of size* $O((n+k)\log^2 n)$ *and height* $O(\log^3 n)$ *for* S *in* $O((n+k)\log^3 n)$ *time.*

6.3 Kinetic Algorithm for Segments

In this section, we develop a technique for maintaining a BSP for a set S of n non-intersecting segments in the plane. We first describe a randomised algorithm for computing a BSP $\mathcal{B}$ for S assuming that the segments in S are stationary, and then explain how to maintain $\mathcal{B}$ as each segment in S moves along a continuous path, under the assumption that the segments remain non-intersecting during the motion.

6.3.1 The static algorithm

Our algorithm makes two types of cuts: a *point* cut is a vertical cut through an endpoint of a segment and an *edge* cut is a cut along a segment. Edge cuts are always contained totally within input segments; therefore, they do not cross any other input segment. For each node $v \in \mathcal{B}$, the corresponding polygon $\mathcal{R}_v$ is a trapezoid; the left and right boundaries of the trapezoid are bounded by point cuts, and the top and bottom boundaries are bounded by edge cuts.

We now describe our static algorithm. We start by choosing a random permutation $\langle s_1, s_2, \ldots, s_n \rangle$ of S. We say that s_i has a *higher priority* than s_j if $i < j$. We add the segments in decreasing order of priority and maintain a BSP for the segments added so far. Let $S^i = \{s_1, s_2, \ldots, s_i\}$ be the set of the first i segments in the permutation. Our algorithm works in n stages. At the beginning of the ith stage, where $i > 0$, we have a BSP $\mathcal{B}^{i-1}$ for S^{i-1}; $\mathcal{B}^0$ consists of a single node v, where $\mathcal{R}_v$ is the entire plane. In the ith stage, we add s_i and compute a BSP $\mathcal{B}^i$ for S^i as follows:

1. Suppose p and q are the left and right endpoints of s_i, respectively. Let v be the leaf of $\mathcal{B}^{i-1}$ such that $\mathcal{R}_v$ contains p. We partition $\mathcal{R}_v$ into two trapezoids $\mathcal{R}_v^-$ and $\mathcal{R}_v^+$ using a point cut defined by p, where $\mathcal{R}_v^-$ lies to the left of the cut. We create two children w and z of v, with w being the left child of v. We set $\mathcal{R}_w = \mathcal{R}_v^-$ and $\mathcal{R}_z = \mathcal{R}_v^+$ and store p at v. We then perform a similar step for q.

2. For each trapezoid $\mathcal{R}_x$ that intersects s_i, we store s_i at x, and split $\mathcal{R}_x$ into two trapezoids by making an edge cut along s_i. We again create two children w and z of v, with w being the left child. We set $\mathcal{R}_w$ to be the sub-trapezoid of $\mathcal{R}_x$ lying below the cut and $\mathcal{R}_z$ to be the sub-trapezoid of $\mathcal{R}_x$ lying above the cut.

The resulting tree is the BSP $\mathcal{B}^i$ for S^i. See Figure 6.6 for an example of constructing $\mathcal{B}^i$ from $\mathcal{B}^{i-1}$.

This completes the description of our algorithm. Note that once we fix the permutation, the algorithm is deterministic and constructs a unique BSP. In order to analyse the algorithm, we need a few definitions. The vertical segment drawn upwards (resp., downwards) from an endpoint p is referred to as the *upper* (resp., *lower*) *thread* of p. The segment containing the other endpoint of a thread is called the *stopper* of that thread. Note that the priority of the stopper of a thread of p is higher than that of the segment containing p. We can prove the following lemma about each thread:

Lemma 6.3.1 *Let p be an endpoint of a segment in S. The expected number of segments crossed by each of p's threads is $O(\log n)$.*

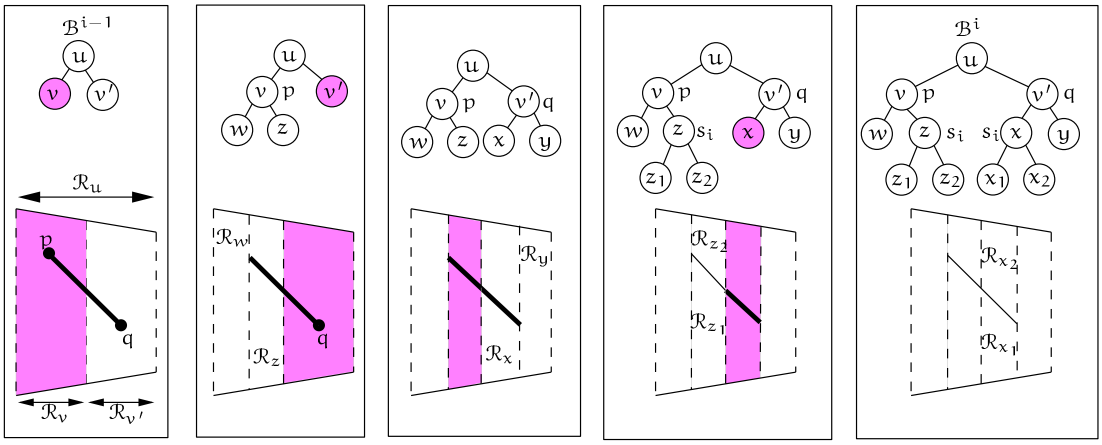


Figure 6.6: The BSP $\mathcal{B}^{i-1}$, the sequence of cuts made in the ith stage, and the BSP $\mathcal{B}^i$. At each step, the shaded trapezoids are split. Portions of s_i that lie in the interior of a trapezoid are drawn using thick lines. The label next to a node signifies the cut made at that node.

Proof: Let $\sigma_1, \sigma_2, \ldots$ be the sequence of segments in S that intersect the top thread ρ of p, sorted in increasing order of the y-coordinates of their intersection with ρ; clearly, there are at most n segments in this sequence. The segment σ_i is crossed by ρ if and only if s has greater priority than all the segments $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$. Since $\mathcal{B}$ is constructed by inserting the segments of S in random order, the probability that ρ crosses σ_i is $1/(i+1)$. Therefore the expected number of segments crossing ρ is at most $\sum_{i=1}^{n} 1/(i+1) = O(\log n)$. We can similarly show that the expected number of segments crossing p's lower thread is $O(\log n)$. □

We can use the above lemma to bound the size and height of $\mathcal{B}$.

Theorem 6.3.1 *The expected size of the BSP constructed by the above algorithm is* $O(n \log n)$, *and the height of the BSP is* $O(\log n)$ *with high probability.*

Proof: In order to bound the size of $\mathcal{B}$, the BSP constructed by the algorithm, it is enough to count the total number of cuts made in $\mathcal{B}$, since a cut is made at each interior node of $\mathcal{B}$. There are at most $2n$ point cuts made in $\mathcal{B}$. If an edge cut e is made at a node v, we charge e to the right endpoint of e. Suppose s is the segment in S containing e. The right endpoint of e is either the right endpoint of s or the intersection point of s with a thread of a segment whose priority is higher than s. In this way, we charge each endpoint and the intersection point of a segment and a thread at most once. As a result, Lemma 6.3.1 implies that the expected total number of edge cuts is $O(n \log n)$, which proves that the expected size of $\mathcal{B}$ is $O(n \log n)$.

To bound $\mathcal{B}$'s height, we first bound the *depth* of an arbitrary point p in the plane, i.e., the number of nodes in the path from the root of $\mathcal{B}$ to the leaf $\nu \in \mathcal{B}$ such that $\mathcal{R}_\nu$ contains p. We bound the number of nodes on this path that are split by edge cuts and point cuts separately.

Let $\sigma_1, \sigma_2, \ldots$ be the ordered sequence of segments in S intersected by a vertical ray starting at p and pointing in the $(+y)$-direction. An ancestor of ν is split by an edge cut through σ_i if and only if σ_i has higher priority than $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$. This event happens with probability $1/i$. Hence, the expected value of X, the number of ancestors of ν that are split by edge cuts, is $H_n = O(\log n)$. We can actually prove that this bound on X holds with high probability. Since X is the sum of independent 0-1 random variables, using Chernoff's bound [69, p. 68], we have that for any constant $\alpha \geq 1$,

$$\Pr[X > \alpha H_n] \leq \left(\frac{e^{\alpha-1}}{\alpha^\alpha}\right)^{H_n} = O(n^{-\alpha \ln \alpha + \alpha - 1}).$$

In particular for any constant $c \geq 3$, we can choose α so that $\Pr[X > \alpha H_n] < 1/n^c$.

We can similarly prove that the number of ancestors of ν that are split by point cuts is $O(\log n)$ with high probability. The segments in S and the vertical lines passing through every segment endpoint decompose the plane into $O(n^2)$ trapezoids. Any two points in one of these trapezoids will be contained in the same leaf of any BSP that our algorithm constructs, independent of the permutation we choose at the beginning of the algorithm. Hence, the height of BSP is the maximum depth of $O(n^2)$ points, one in each such trapezoid. Since the depth of each point is $O(\log n)$ with probability $1 - 1/n^c$, where $c \geq 3$, the height of $\mathcal{B}$ is also $O(\log n)$ with probability $1 - 1/n^{c-2}$. This completes the proof of the lemma. $\square$

6.3.2 The kinetic algorithm

We now describe how to maintain the static BSP as the segments in S move continuously, under the assumption that their interiors remain pairwise disjoint throughout the motion. We parameterise the motion of the segments by time and use t to denote time. For a given time instant t, we will use t^- and t^+ to denote the time instants $t-\varepsilon$ and $t+\varepsilon$, respectively, where $\varepsilon > 0$ is a sufficiently small constant.

Let $s_i \in S$ be a segment with endpoints p and q. We assume that the position of p at time t is $p(t) = (x_p(t), y_p(t))$, where $x_p(t)$ and $y_p(t)$ are continuous functions of time; $q(t)$ is specified similarly. The position of s_i at time t is $s_i(t) = (p(t), q(t))$; if s_i is moving rigidly, then the equations for its endpoints are not independent. Our algorithm and the analysis work even if s_i's endpoints move independently. Let $S(t)$ denote the set S at time t. We assume that we choose a random permutation π of S once in the very beginning (at $t = 0$), and that π does not change with time. Let $\mathcal{B}(t)$ denote the BSP of $S(t)$ constructed by the static algorithm, using π as the permutation to decide the priority of the segments. We describe an algorithm that updates the BSP under the following assumption:

> ($\star$) *There is no correlation between the motion of the segments in S and their priorities. Therefore, the chosen permutation π always behaves like a random permutation, and Lemma 6.3.1 and Theorem 6.3.1 hold at all times.*

We first give an important definition. The *combinatorial structure* of $\mathcal{B}$ is a binary tree, each of whose nodes v is associated with the set of segments S_v. We will use the combinatorial structure of the BSP crucially in our algorithm.

Critical events

As the segments in S move continuously, the equations of the cuts associated with the nodes of $\mathcal{B}$ also change. At the same time, the edges and vertices of the trapezoids in the subdivision of the plane induced by $\mathcal{B}$ also move. However, the combinatorial structure of $\mathcal{B}$ changes only when the set S_v changes for some node $v \in \mathcal{B}$. Since the segments in S are interior-disjoint and they move continuously, the set S_v changes only when the endpoint of a segment in S_ν lies on the left or right edge of $\mathcal{R}_\nu$. See Figure 6.7 for an example of such an event. We formalise this idea in the following lemma, which is not difficult to prove:

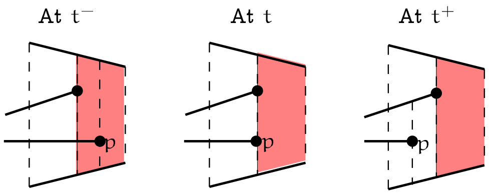


Figure 6.7: Endpoint p lies on the left edge of $\mathcal{R}_\nu$ (the shaded trapezoid) at t. The set S_ν changes at time instant t.

Lemma 6.3.2 *For any time instant* t, $\mathcal{B}(t^-)$ *and* $\mathcal{B}(t^+)$ *have different combinatorial structures if and only if there exists a* $j > 0$ *such that either* s_j *rotates through a vertical line at time* t *or there is a leaf* $\nu \in \mathcal{B}^{j-1}(t^-)$ *such that an endpoint of* s_j *lies on the left or right edge of* $\mathcal{R}_\nu$ *at time* t.

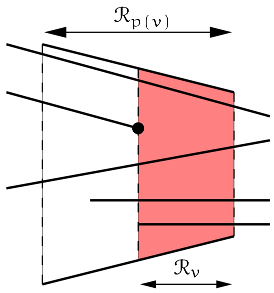


Figure 6.8: The shaded trapezoid $\mathcal{R}_\nu$ is transient.

The above lemma implies that the combinatorial structure of $\mathcal{B}(t)$ changes if and only if for a node $\nu \in \mathcal{B}(t)$, $\mathcal{R}_\nu$ shrinks to a vertical segment; we refer to these instants of time as *critical events*. This observation motivates us to call a node ν in $\mathcal{B}(t)$ *transient* if $\mathcal{R}_\nu$ does not contain any endpoint in its interior and a point cut is made at the parent $p(\nu)$ of ν; we call $\mathcal{R}_\nu$ a *transient trapezoid*. See Figure 6.8. Note that only edge cuts are made at ν and its descendants. The following corollary to Lemma 6.3.2 is immediate:

Lemma 6.3.3 *For any time instant* t*,* $\mathcal{B}(t^-)$ *and* $\mathcal{B}(t^+)$ *have different combinatorial structures if and only if there exists a transient node* ν *in* $\mathcal{B}(t^-)$ *so that* $\mathcal{R}_\nu$ *becomes a vertical segment at time* t*.*

Transient nodes have some useful properties that are summarised in the following lemma:

Lemma 6.3.4 *At any instant* t*, the set of transient nodes in* $\mathcal{B}(t)$ *have the following properties. Let* ν *be a transient node in* $\mathcal{B}(t)$*.*

1. *No proper ancestor of* ν *is transient.*
2. *Only edge cuts are made at the descendants of* ν *(including* ν *itself). The left (resp., right) edge of the trapezoid associated with each descendant of* ν *is a portion of the left (resp., right) edge of* $\mathcal{R}_\nu$*.*
3. *The expected number of descendants of* ν *is* $O(\log n)$*.*
4. *The number of transient nodes in* $\mathcal{B}(t)$ *is at most* $4n$*.*

Proof: Let p be the endpoint of a segment in S through which the point cut at $p(\nu)$ is made.

1. No proper ancestor w of ν can be transient since $\mathcal{R}_w$ contains p.
2. Since $\mathcal{R}_\nu$ does not contain any endpoints, only edge cuts are made at all the descendants of ν. Each segment that intersects $\mathcal{R}_\nu$ crosses the left and right boundaries of ν. Hence, the left (resp., right) edge of the trapezoids associated with each descendant of ν is a portion of the left (resp., right) edge of $\mathcal{R}_\nu$.
3. Each segment that induces an edge cut made at a descendant of ν intersects one of p's threads. Hence, by Lemma 6.3.1, the expected number of descendants of ν is $O(\log n)$.
4. A point cut is made at the parent of each transient node. There are $2n$ nodes in $\mathcal{B}$ that are split by point cuts; each such node has at most two children.

□

Intuitively, transient nodes are the highest nodes in $\mathcal{B}(t)$ that can shrink to a vertical segment, causing a change in the combinatorial structure of $\mathcal{B}(t)$. If a trapezoid contains an endpoint in its interior, it cannot shrink to a segment; and if an edge cut is made at the parent $p(\nu)$ of a node ν and $\mathcal{R}_{p(\nu)}$ does not contain an endpoint, then $\mathcal{R}_{p(\nu)}$ also shrinks to a segment whenever $\mathcal{R}_\nu$ shrinks to a segment. Hence, it suffices to keep track of transient nodes to determine all the instants when the combinatorial structure of $\mathcal{B}(t)$ changes. In the rest of the section, we present our kinetic algorithm motivated by this observation.

Our algorithm

For a node ν in $\mathcal{B}$, let λ_ν (resp., ρ_ν) denote the endpoint of a segment in S that induces the point cut containing the left (resp., right) edge of $\mathcal{R}_\nu$. To detect critical events, we maintain the set

$$\Gamma(t) = \{(\lambda_\nu, \rho_\nu) \mid \nu \text{ is a transient node at time } t\}$$

of endpoint pairs inducing the point cuts that bound the left and right edges of each transient node; Lemma 6.3.4 implies that $|\Gamma(t)| = O(n)$. The elements of $\Gamma(t)$ are certificates that prove that the combinatorial structure of $\mathcal{B}(t)$ is valid. For each pair (λ_ν, ρ_ν) in $\Gamma(t)$, we use the known flight paths of λ_ν and ρ_ν to compute the time at which the x-coordinates of λ_ν and ρ_ν coincide, and store these time values in a global priority queue. In order to expedite the updating of $\mathcal{B}$ at each critical event, we store some additional information with the nodes in $\mathcal{B}$ and the segments in S:

1. At each node ν of $\mathcal{B}$, we store the number c_ν of segment endpoints lying in the interior of $\mathcal{R}_\nu$ (c_ν helps us to determine the new transient trapezoids at an event).

2. For each endpoint p of a segment in S, we maintain the list T_p (resp., B_p) of segments that the upper (resp., lower) thread of p crosses, sorted in the $(+y)$-direction (resp., $(-y)$-direction). As the segments move, we will use these lists to update the stoppers of the threads issuing from the segment endpoints.

We first construct $\mathcal{B}(0)$ using the static algorithm presented in Section 6.3.1. Next, we compute the set $\Gamma(0)$ and insert the corresponding critical events in the priority queue. Then we repeatedly remove the next event from the priority queue and update $\mathcal{B}$, Γ, and the priority queue as required. In the rest of the section, we will prove that if the combinatorial structure of $\mathcal{B}$ changes at time t, then we can obtain $\mathcal{B}(t^+)$ from $\mathcal{B}(t^-)$ in $O(\log n)$ expected time. We will also show that at each event point, the expected time to update the global event queue is $O(\log n)$.

We now describe the procedure for updating the tree at each critical event. Recall that at each such instant t, (i) there is a segment $s_j \in S$ such that either s_j becomes vertical or (ii) there is a leaf $w \in \mathcal{B}^{j-1}(t^-)$ such that an endpoint p of s_j lies on the left or right edge of $\mathcal{R}_w$. If s_j is vertical, ν is a transient node in $\mathcal{B}(t^-)$ with the property that the left and right edges of $\mathcal{R}_\nu$ are contained in point cuts induced by the endpoints of s_j. In the second case, at time t^-, a point cut made through p divides $\mathcal{R}_w$ into two trapezoids. One of these trapezoids is $\mathcal{R}_\nu$, which is transient at time t^-. Let $\mathcal{B}^- = \mathcal{B}(t^-)$ and $\mathcal{B}^+ = \mathcal{B}(t^+)$. For a node $z \in \mathcal{B}^-$, let $\mathcal{B}^-_z$ denote the subtree of $\mathcal{B}^-$ rooted at z; define $\mathcal{B}^+_z$ similarly. There are two cases to consider.

Case (i): λ_ν and ρ_ν are both endpoints of s_j. The segment s_j is vertical at time t. See Figure 6.9. Let u be ν's grandparent in $\mathcal{B}^-$; the trapezoid $\mathcal{R}_u$ contains s_j. Suppose ν is in the right subtree of u in $\mathcal{B}^-$. Let ν_1 be the left child of u and let ν_2 be the sibling of ν in $\mathcal{B}^-$. To obtain $\mathcal{B}^+$, we create a new node ν_1' and attach it as the left child of u. The left and right child of ν_1' are ν_1 and ν, respectively. We delete the right child of u and replace it with ν_2.

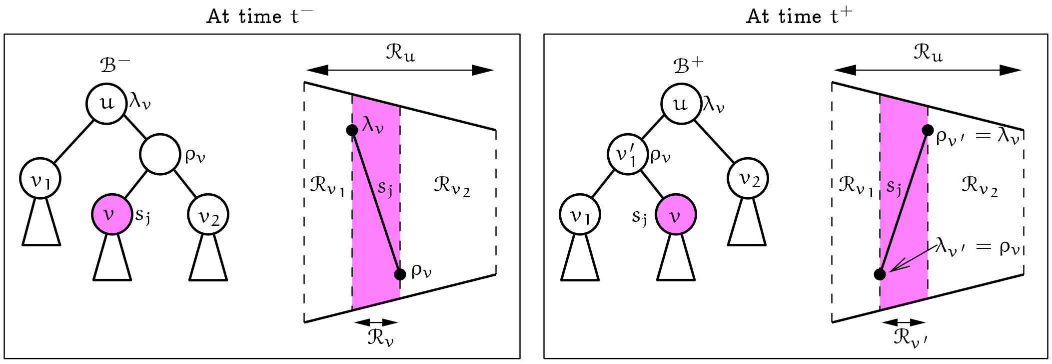


Figure 6.9: The case when λ_ν and ρ_ν belong to the same segment.

Case (ii): λ_v and ρ_v are endpoints of different segments. Assume that $\rho_v = p$ (resp., $\lambda_v = q$) is the right endpoint of the segment s_i (resp., s_j), that s_i lies above s_j, that the priority of s_i is higher than that of s_j (i.e., $i < j$), and that the x-coordinate of q is less than the x-coordinate of p at t^-; see Figure 6.10. We now describe how we update $\mathcal{B}(t)$ for this case. We will show later how to relax these assumptions. Let u and w be the leaves of $\mathcal{B}^{i-1}(t^-)$ and $\mathcal{B}^{j-1}(t^-)$, respectively, at which the point cuts through p and q, respectively, are made. Then, by our assumptions about the event, v is the right child of w, and w lies in the left subtree of u. Let u_L be the left child of u, and let w_L be the left child of w. Let x be the leaf of $\mathcal{B}^{j-1}(t^-)$ that contains q at time t^+; x lies in the right subtree of u. Since the combinatorial structures of $\mathcal{B}^{j-1}(t^-)$ and $\mathcal{B}^{j-1}(t^+)$ are identical, x is a leaf of $\mathcal{B}^{j-1}(t^+)$ too. Let $s_k \in S$ be the segment containing the top edge of $\mathcal{R}_x$; clearly, $k \geq i$. At time t^+, as q leaves the trapezoid $\mathcal{R}_w$ and enters $\mathcal{R}_x$, $\mathcal{R}_{w_L}$ expands to $\mathcal{R}_w$, $\mathcal{R}_v$ disappears, and $\mathcal{R}_x$ is split at t^+ into two trapezoids: a new trapezoid $\mathcal{R}_{v'}$ and the portion of $\mathcal{R}_x$ lying to the right of the cut through q. At time t^-, $\mathcal{R}_w$ is split by a point cut through q and $\mathcal{R}_{w_L}$ is split by an edge cut along s_j, while at time t^+, $\mathcal{R}_w$ is split by an edge cut along s_j. Therefore $\mathcal{B}^+_w$ is the same as $\mathcal{B}^-_{w_L}$. To obtain $\mathcal{B}^+$, we execute the following steps:

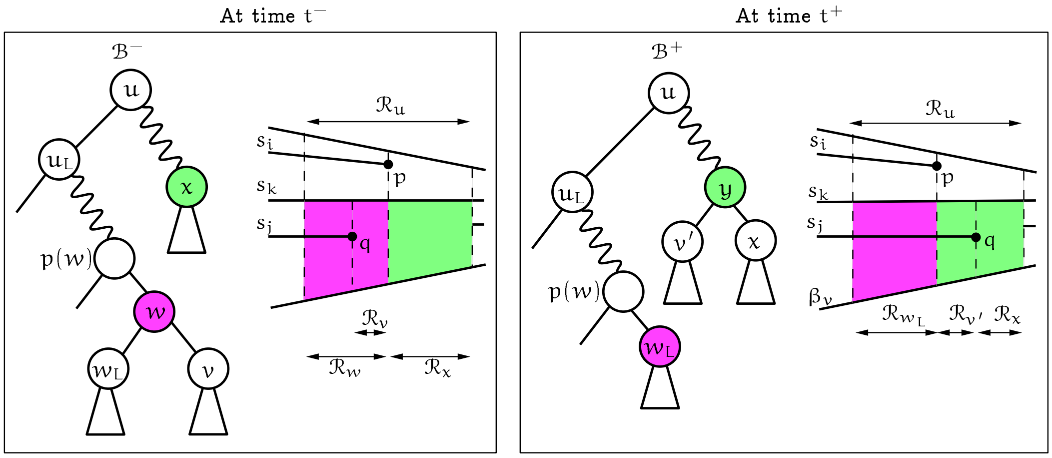


Figure 6.10: The case when λ_v and ρ_v are endpoints of different segments. Arrows mark the horizontal extents of the trapezoids.

1. We search in the right subtree of u to locate the leaf x of $\mathcal{B}^{j-1}(t^-)$ such that $\mathcal{R}_x$

contains q at time t^+.

2. We delete the node w from $\mathcal{B}^-$, and if w was a left (resp., right) child of its parent $p(w)$, we make w_L the new left (resp., right) child of $p(w)$.

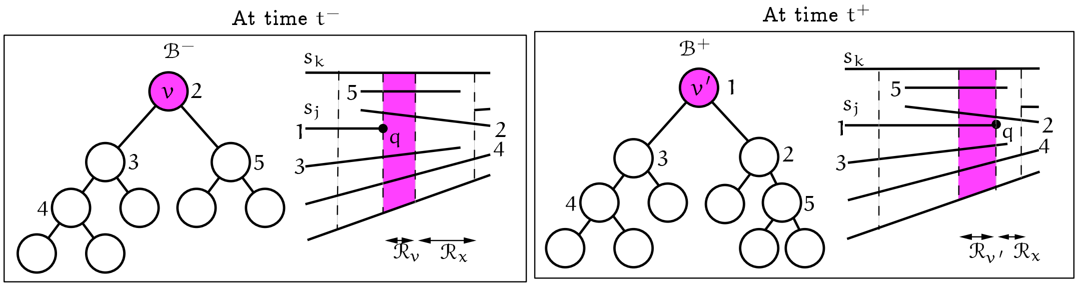


Figure 6.11: The edge cuts made in $\mathcal{B}_v^-$ and $\mathcal{B}_{v'}^+$. Segments crossing $\mathcal{R}_v$ and $\mathcal{R}_{v'}$ are labelled with their priorities. The label next to a node is the priority of the segment containing the edge cut made at that node.

3. We construct the subtree $\mathcal{B}_{v'}^+$ by determining the set C of segments that intersect $\mathcal{R}_{v'}$ (at time t^+) and by making edge cuts through the segments in C in decreasing order of priority. There are two cases to consider:

   (a) s_k contains the top edge of $\mathcal{R}_v$: See Figure 6.11. The set C consists of s_j and the set of segments intersecting $\mathcal{R}_v$ (at time t^-). We find these segments by traversing all the nodes of $\mathcal{B}_v^-$.

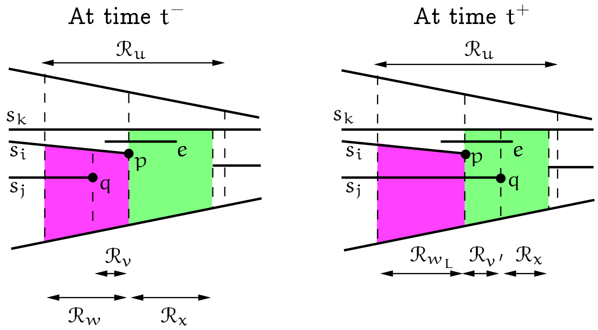


Figure 6.12: The case when the top edges of $\mathcal{R}_w$ and $\mathcal{R}_x$ are contained in different segments. The segment e is intersected by the top thread of q at t^+ but not at t^-.

   (b) s_i contains the top edge of v: See Figure 6.12. We set s_k to be the stopper of the upper thread of q. As in the previous case, we include s_j and the segments inducing the edge cuts made in $\mathcal{B}_v^-$ in C. In addition, C contains all segments that appear before s_k in the upper thread of p. We determine these segments by traversing T_p, the list of segments that cross the upper thread of p. Note that these segments also cross the upper thread of q at t^+.

Finally, we insert s_j into B_p, the list of segments in S crossed by the lower thread of p, and update T_q.

4. We attach $\mathcal{B}_{v'}^+$ to a descendant of $p(x)$, the parent of x in $\mathcal{B}^-$, as follows: We create a node y and associate the point cut through q with it. The left and right subtrees of y in $\mathcal{B}^+$ are $\mathcal{B}_{v'}^+$ and $\mathcal{B}_x^-$, respectively. If x is the left (resp., right) child of $p(x)$, then we add y as the new left (resp., right) child of $p(x)$.

5. We update the set $\Gamma(t^+)$. For a node z, the number c_z of endpoints lying in the interior of $\mathcal{R}_z$, changes only if z lies along the paths in $\mathcal{B}^+$ from u to the nodes $p(w)$ and y. For such a node z, if $c_z = 0$ at t^+ and if $\mathcal{R}_{p(z)}$ is split by a point cut, we add (λ_z, ρ_z) to the list $\Gamma(t^+)$. On the other hand, if $c_z \neq 0$ at t^+ but z is transient at t^- (z must be an ancestor of x in $\mathcal{B}^-$), we delete (λ_v, ρ_z) from $\Gamma(t^+)$. We also update the priority queue to reflect the changes to $\Gamma(t^+)$.

Other cases: We now show how we relax the assumptions we made earlier about the relative positions of s_i and s_j and their priorities. For each case, we show how a simple transformation reduces it to one of the earlier cases.

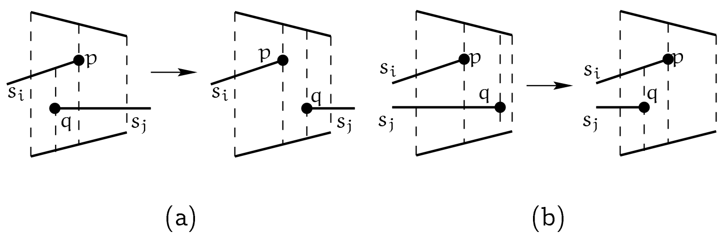


Figure 6.13: Some other cases that arise when different segments interact in a critical event.

1. If q is the left endpoint of s_j, the update procedure is the same, except that at time t^+ we do not make an edge cut through s_j in $\mathcal{B}^+_{v'}$. See Figure 6.13(a).

2. If the x-coordinate of q is greater than the x-coordinate of p at time t^-, we "go back in time." Figure 6.13(b). The node x is again the leaf of $\mathcal{B}^{j-1}(t^-)$ such that $\mathcal{R}_x$ contains q at t^+. We can reconstruct $\mathcal{B}^+_{v'}$ as before: if the same segment contains the top edge of $\mathcal{R}_v$ and $\mathcal{R}_x$, the same set of segments intersects $\mathcal{B}^-_v$ and $\mathcal{B}^+_{v'}$ (except that s_j does not intersect $\mathcal{B}^+_{v'}$); otherwise, among the segments that intersect $\mathcal{B}^-_v$, only segments with priority at most i intersect $\mathcal{B}^+_{v'}$. In the second case, we also update T_q accordingly. The other changes to $\mathcal{B}$ are similar to the cases we have handled; the details are not difficult to work out.

3. If p is the left endpoint of s_i, we reflect S about the y-axis.

4. If s_i lies below s_j, we reflect S about the x-axis.

5. If the priority of s_i is less than the priority of s_j, we swap the roles of s_i and s_j.

This completes the description of our procedure for processing critical events. We now analyse the running time of the update procedure. Assumption $(\star)$ implies that Lemma 6.3.1 and Theorem 6.3.1 hold at times t^-, t, and t^+. We spend $O(\log n)$ time in Step 1, since we traverse a path in $\mathcal{B}$ to find the node x. It is clear that Step 2 takes $O(1)$ time. In Step 3, we find the segments crossing $\mathcal{R}_{v'}$ and construct $\mathcal{B}^+_{v'}$ in $O(\log n)$ expected time, since the expected size of $\mathcal{B}^-_v$ is $O(\log n)$ (by Lemma 6.3.4) and the expected number of segments in T_p is $O(\log n)$ (by Lemma 6.3.1). It is clear that Step 4 takes $O(1)$ time. Finally, in Step 5, we process $O(\log n)$ nodes lying in two paths in the tree. By Lemma 6.3.4, each of the two paths contains at most one transient node. Hence, we insert or delete at most two events from the priority queue, which implies that Step 5 takes $O(\log n)$ time. We thus obtain the main result of this section:

Theorem 6.3.2 *At each critical event, we can update $\mathcal{B}(t)$ in $O(\log n)$ expected time.*

Note that this theorem makes our BSP a kinetic data structure that is responsive, efficient, local, and compact, in the sense defined by Basch et al. [13].

We say that the trajectories followed by a set of segments are *pseudo-algebraic* if the segments move so that each pair of endpoints exchanges y-order only $O(1)$ times. A special case of pseudo-algebraic trajectories is when the trajectories of all the endpoints are constant-degree polynomials. If the trajectories of k of the segments in S are pseudo-algebraic and the remaining segments are stationary, then the total number of event points is $O(kn)$. We spend $O(\log n)$ expected time to maintain $\mathcal{B}(t)$ at each event point. Hence, we obtain the following corollary to Theorem 6.3.2:

Corollary 6.3.3 *Let S be a set of n segments in the plane, and let $G \subseteq S$ be a set of k segments. Suppose each segment of G moves along a pseudo-algebraic trajectory and the remaining segments of S are stationary, the total expected time spent in maintaining $\mathcal{B}$ is $O(kn \log n)$.*

6.4 Conclusions

We have presented algorithms for constructing BSPs for triangles in $\mathbb{R}^3$. The randomised algorithm constructs a BSP of optimal size runs in near-optimal time in the worst case. The deterministic algorithm is near-optimal in the worst-case. However, we have shown that it is likely to construct BSPs of near-linear size for inputs that actually arise in practice.

An attractive feature of the BSP constructed by the deterministic algorithm is that for each node ν in the BSP, $\mathcal{R}_\nu$ is a cylindrical cell with four vertical faces and two faces contained in triangles in S. In current work, we are using this property to robustly implement our deterministic BSP algorithm. We reduce all computations needed to construct the BSP to comparisons between xy-projections of the triangles in S or between the z-coordinates of the triangles in S. Another advantage of the constant size of the BSP cells is that any operation on a node ν (e.g., checking if a point lies in the interior of $\mathcal{R}_\nu$ or intersecting a ray with $\mathcal{R}_\nu$) takes only $O(1)$ time. In BSPs constructed by other known algorithms [8, 101, 102, 73], the number of faces of $\mathcal{R}_\nu$ can be as large as the height of ν in the BSP; thus these operation take time proportional to the height of ν.

We have also presented an efficient algorithm to maintain the BSP of moving segments in the plane. Currently, we do not know any non-trivial lower bounds for this problem. Recently, Agarwal et al. [2] have extended our result and developed an algorithm to maintain BSPs for moving triangles in $\mathbb{R}^3$. Note that such BSPs can be used in our hidden-surface removal algorithm when occluders are moving (see Chapter 3.3.6).

There are many very interesting open questions regarding BSPs. First of all, our model of geometric complexity (the number of intersections between xy-projections of triangle edges) is ideal for terrains and urban landscapes but might not be very good for data sets in other domains (CAD design, for example). Proving near-linear bounds on BSP size in new (and more general) models of geometric complexity will be very useful. Secondly, all our algorithms for triangles in $\mathbb{R}^3$ construct BSPs of $\Omega(n^2)$ size even if an $O(n)$ size BSP exists. This raises the question of constructing a BSP of optimal or near-optimal size for triangles in $\mathbb{R}^3$.

Chapter 7

Conclusions

In this dissertation, we have studied the fundamental computer graphics problem of hidden-surface removal. There is a vast gap between the techniques developed for this problem in the computer graphics and computational geometry communities. A careful examination of the differences between these two classes of algorithms motivated us to study hidden-surface removal in the framework of

- *geometric complexity*, where we analyse algorithms in terms of the geometric structure present in the input, and develop techniques that are provably efficient for data sets that typically arise in practice,
- *object complexity*, a model inspired by the performance characteristics of current graphics hardware, in which we develop hidden-surface removal algorithms that simply determine which objects are visible rather than compute exactly which portions of each object are visible, and
- *kinetic data structures*, a mechanism for efficiently processing continuously-moving objects that avoids the pitfalls of approximating continuous motion by a uniform discretisation of time.

We first described a technique we have developed for geometric data repair for automatic correction of geometric and topological flaws in the input. Our algorithm partitioned $\mathbb{R}^3$ into regions and used the novel idea of using region adjacencies to determine "how solid" a region is. Our experiments demonstrated that unlike previously described approaches, our method is effective for a large class of input models.

Next, we presented a new object-complexity algorithm for the hidden-surface removal of massive models. The basic principles we used in the algorithm were to compute a hierarchical spatial decomposition, and classify cells of this decomposition as visible or invisible by intersecting the cells with the union of the shadows cast by a few carefully-chosen occluders. Our technique had several new and attractive features: we maintained the union of the shadows as a set of rays in $\mathbb{R}^3$; we were able to compute the set of visible cells exactly, and could also detect when a cell was occluded by multiple, disconnected triangles; we explicitly exploited the continuity of the motion of the viewpoint and the objects in the input by using kinetic data structures [13]; we used the binary space partition (BSP) to unify occluder selection, visibility maintenance, and mechanisms for frame-rate control.

Both the model repair and hidden-surface removal algorithms used the BSP as an underlying spatial data structure, and their running time depended on the size of the BSP. Inspired by this fact, we described several algorithms for constructing BSPs of small size. We showed that BSPs of near-linear size can be constructed for orthogonal rectangles with low geometric complexity, where geometric complexity is measured in terms of the aspect ratio of the rectangles. Our implementation demonstrated that our algorithm indeed constructs BSPs of linear size in practice on "real" models and performs better in practice than most algorithms presented in the literature. We also presented two algorithms for constructing BSPs for a set of triangles in $\mathbb{R}^3$. One of these algorithms was the first-known algorithm with near-optimal running time in the worst case. The other algorithm constructed BSPs of near-linear size and polylogarithmic depth for triangles that form a "near"-terrain. Finally, we presented the first-known provably-efficient algorithm for maintaining the BSP for a set of moving segments in the plane.

Our model repair and occlusion-culling algorithms in combination with our efficient algorithms for constructing BSPs provide a powerful set of techniques for solving the hidden-surface removal problem. The success of our approach is a direct result of our exploiting the three themes that form the core of our research: geometric complexity, object complexity, and kinetic data structures.

We now discuss the wider applicability of this framework and give evidence to suggest that these three notions are powerful and can be used to develop theoretically-and practically-efficient algorithms for problems in many other domains.

While the object complexity model is specific to the hidden-surface removal problem, the principle that motivated it is more widely applicable: develop algorithms that exploit the properties of the hardware available on current and future machines. Keeping algorithm creation in tune with developments in hardware also raises the intriguing question of influencing hardware design to support certain classes of algorithms. In the context of hidden-surface removal itself, algorithms that fully leverage the rapid developments in parallel graphics hardware and PC-based graphics accelerators are likely to find wide acceptance. Another problem that will benefit from a close connection between algorithms and hardware is volume rendering. So far, there has not been much theoretical work on efficient algorithms for volume rendering.

Geometric complexity is useful in any scenario where worst-case inputs do not usually arise in practice. For example, other authors have used similar ideas to develop efficient algorithms for motion planning [104] and point location and range searching [80]. The challenge in using geometric complexity effectively is in coming up with a measure that accurately captures the geometric characteristics of typical inputs. We believe this framework will be useful for other problems in computer graphics like global illumination and volume rendering, as well as in domains such as computer vision and multi-media indexing.

Apart from computer graphics, kinetic data structures are applicable in any domain with moving objects, such as robotics and computer vision. In fact, in addition to the kinetic data structures we have mentioned earlier in the thesis, there has already been a lot of research on using kinetic data structures to solve various geometric problems [3, 13, 14, 62]. Kinetic data structures promise to significantly impact any problem areas that deal with moving objects or continuous change.

In this thesis, we have applied the themes of object complexity, geometric complexity, and kinetic data structures to develop a set of efficient algorithms for hidden-surface removal. These three powerful paradigms hold the promise of stimulating the development of theoretically interesting and practically useful algorithms for problems in computer graphics, robotics, computer vision, and other fundamental areas of computer science.

Bibliography

[1] P. K. Agarwal and J. Erickson, Geometric range searching and its relatives, in: *Advances in Discrete and Computational Geometry* (J. E. G. B. Chazelle and R. Pollack, eds.), AMS Press, Providence, RI, 1998.

[2] P. K. Agarwal, J. Erickson, and L. J. Guibas, Kinetic binary space partitions for intersecting segments and disjoint triangles, *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998, pp. 107–116.

[3] P. K. Agarwal, L. J. Guibas, J. Hershberger, and E. Veach, Maintaining the extent of a moving point set, *Proc. 5th Workshop Algorithms Data Struct.*, *Lecture Notes Comput. Sci.*, Vol. 1272, Springer-Verlag, 1997, pp. 31–44.

[4] P. K. Agarwal, L. J. Guibas, T. M. Murali, and J. S. Vitter, Cylindrical static and kinetic binary space partitions, *Proc. 13th ACM Sympos. Comput. Geom.*, 1997, pp. 39–48.

[5] P. K. Agarwal and J. Matoušek, Ray shooting and parametric search, *SIAM J. Comput.*, 22 (1993), 794–806.

[6] P. K. Agarwal and J. Matoušek, On range searching with semialgebraic sets, *Discrete Comput. Geom.*, 11 (1994), 393–418.

[7] P. K. Agarwal and S. Suri, Surface approximation and geometric partitions, *SIAM Journal on Computing*, 27 (1998), 1016–1035.

[8] J. M. Airey, *Increasing Update Rates in the Building Walkthrough System with Automatic Model-space Subdivision and Potentially Visible Set Calculations*, Ph.D. Thesis, Dept. of Computer Science, University of North Carolina, Chapel Hill, 1990.

[9] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha, A framework for the real-time walkthrough of massive models, Tech. Rep. TR98-013, Department of Computer Science, University of North Carolina at Chapel Hill, 1998.

[10] D. G. Aliaga and A. A. Lastra, Architectural walkthroughs using portal textures, *Proc. IEEE Visualization '97*, IEEE, November 1997, pp. 355–362.

[11] J. Avro and D. Kirk, A survey of ray tracing acceleration techniques, in: *An Introduction to Ray Tracing* (A. Glassner, ed.), Academic Press, San Diego, CA, 1989, pp. 201–262.

[12] C. Ballieux, Motion planning using binary space partitions, Tech. Rep. inf/src/93-25, Utrecht University, 1993.

[13] J. Basch, L. J. Guibas, and J. Hershberger, Data structures for mobile data, *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, 1997, pp. 747–756.

[14] J. Basch, L. J. Guibas, and L. Zhang, Proximity problems on moving points, *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, 1997, pp. 344–351.

[15] D. R. Baum, S. Mann, K. P. Smith, and J. M. Winget, Making radiosity usable: Automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions, *Proc. SIGGRAPH 91*, *Comput. Graph.*, Vol. 25, ACM SIGGRAPH, 1991, pp. 51–60.

[16] J. L. Bentley and T. A. Ottmann, Algorithms for reporting and counting geometric intersections, *IEEE Trans. Comput.*, C-28 (1979), 643–647.

[17] M. Bern, Hidden surface removal for rectangles, *J. Comput. Syst. Sci.*, 40 (1990), 49–69.

[18] M. Bern, D. Dobkin, D. Eppstein, and R. Grossman, Visibility with a moving point of view, *Algorithmica*, 11 (1994), 360–378.

[19] M. Bern, D. Eppstein, P. Plassman, and F. Yao, Horizon theorems for lines and polygons, in: *Discrete and Computational Geometry: Papers from the DIMACS Special Year* (J. Goodman, R. Pollack, and W. Steiger, eds.), *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 6, American Mathematical Society, Providence, RI, 1991, pp. 45–66.

[20] J. H. Bøhn, Removing zero-volume parts from CAD models for layered manufacturing, *IEEE Computer Graphics and Applications*, 15 (1995), 27–34.

[21] J. H. Bøhn and M. J. Wozny, A topology-based approach for shell closure, in: *Geometric Modeling for Product Realization* (P. R. Wilson, M. J. Wozny, and M. J. Pratt, eds.), North-Holland, Amsterdam, 1993, pp. 297–319.

[22] G. Butlin and C. Stops, CAD data repair, *Proceedings of the 5th International Meshing Roundtable*, October 1996. See also `http://www.fegs.co.uk/CADfix.html`.

[23] A. T. Campbell, *Modeling Global Diffuse Illumination for Image Synthesis*, Ph.D. Thesis, Dept. of Computer Sciences, University of Texas, Austin, 1991.

[24] T. Cassen, K. R. Subramanian, and Z. Michalewicz, Near-optimal construction of partitioning trees by evolutionary techniques, *Proc. of Graphics Interface '95*, 1995, pp. 263–271.

[25] E. Catmull, A subdivision algorithm for computer display of curved surfaces, Tech. Rep. UTEC-CSC-74-133, Dept. Comput. Sci., Univ. Utah, Salt Lake City, 1974.

[26] B. Chamberlain, T. DeRose, D. Lischinski, D. Salesin, and J. Snyder, Fast rendering of complex environments using a spatial hierarchy, *Proc. Graphics Interface '96*, 1996, pp. 132–14.

[27] B. Chazelle, Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm, *SIAM J. Comput.*, 13 (1984), 488–507.

[28] B. Chazelle, L. J. Guibas, and D. T. Lee, The power of geometric duality, *BIT*, 25 (1985), 76–90.

[29] N. Chin and S. Feiner, Near real-time shadow generation using BSP trees, *Proc. SIGGRAPH 89*, *Comput. Graph.*, Vol. 23, ACM SIGGRAPH, 1989, pp. 99–106.

[30] N. Chin and S. Feiner, Fast object-precision shadow generation for areal light sources using BSP trees, *Proc. 1992 Sympos. Interactive 3D Graphics*, 1992, pp. 21–30.

[31] Y. Chrysanthou, *Shadow Computation for 3D Interaction and Animation*, Ph.D. Thesis, Queen Mary and Westfield College, University of London, 1996.

[32] K. L. Clarkson and P. W. Shor, Applications of random sampling in computational geometry, II, *Discrete Comput. Geom.*, 4 (1989), 387–421.

[33] S. Coorg and S. Teller, Temporally coherent conservative visibility, *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, 1996, pp. 78–87.

[34] S. Coorg and S. Teller, Real-time occlusion culling for models with large occluders, *Proc. 1997 Sympos. Interactive 3D Graphics*, 1997, pp. 83–90.

[35] M. de Berg, Linear size binary space partitions for fat objects, *Proc. 3rd Annu. European Sympos. Algorithms*, *Lecture Notes Comput. Sci.*, Vol. 979, Springer-Verlag, 1995, pp. 252–263.

[36] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld, Efficient ray shooting and hidden surface removal, *algo*, 12 (1994), 30–53.

[37] M. de Berg, M. J. Katz, A. F. van der Stappen, and J. Vleugels, Realistic input models for geometric algorithms, *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, 1997, pp. 294–303.

[38] F. Dévai, Quadratic bounds for hidden line elimination, *Proc. 2nd Annu. ACM Sympos. Comput. Geom.*, 1986, pp. 269–275.

[39] S. E. Dorward, A survey of object-space hidden surface removal, *Internat. J. Comput. Geom. Appl.*, 4 (1994), 325–362.

[40] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, West Germany, 1987.

[41] C. Erikson, Error correction of a large architectural model: the Henderson County Courthouse, Tech. Rep. TR95-013, Dept. of Computer Science, University of North Carolina at Chapel Hill, 1995.

[42] C. Erikson, Polygonal simplification: an overview, Tech. Rep. TR96-016, Dept. of Computer Science, University of North Carolina at Chapel Hill, 1996.

[43] F. Evans, S. S. Skiena, and A. Varshney, Optimizing triangle strips for fast rendering, *IEEE Visualization '96 Proceedings*, October 1996, pp. 319–326.

[44] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, MA, 1990.

[45] S. Fortune and C. J. Van Wyk, Efficient exact arithmetic for computational geometry, *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, 1993, pp. 163–172.

[46] S. J. Fortune, D. M. Gay, B. W. Kernighan, O. Landron, R. A. Valenzuela, and M. H. Wright, WISE design of indoor wireless systems: practical computation and optimization, *IEEE Computational Science and Engineering*, Vol. 2, Spring 1995, pp. 58–68.

[47] H. Fuchs, Z. M. Kedem, and B. Naylor, On visible surface generation by a priori tree structures, *Proc. SIGGRAPH 80*, *Comput. Graph.*, Vol. 14, ACM SIGGRAPH, 1980, pp. 124–133.

[48] T. A. Funkhouser, RING: A client-server system for multi-user virtual environments, *Proc. 1995 Sympos. Interactive 3D Graphics*, 1995, pp. 85–92.

[49] T. A. Funkhouser and C. H. Séquin, Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments, *Proc. SIGGRAPH 93*, in *Comput. Graph.* Proceedings, Annual Conference Series, ACM SIGGRAPH, 1993, pp. 247–254.

[50] M. T. Goodrich, M. J. Atallah, and M. H. Overmars, Output-sensitive methods for rectilinear hidden surface removal, *Inform. Comput.*, 107 (1993), 1–24.

[51] S. Gottschalk, M. C. Lin, and D. Manocha, OBBtree: A hierarchical structure for rapid interference detection, *Proc. SIGGRAPH 96*, In *Comput. Graph.* Proceedings, Annual Conference Series, ACM SIGGRAPH, 1996, pp. 171–180.

[52] N. Greene, Hierarchical polygon tiling with coverage masks, *Proc. SIGGRAPH 96*, In *Comput. Graph. Proceedings*, Annual Conference Series, ACM SIGGRAPH, 1996, pp. 65–74.

[53] N. Greene, M. Kass, and G. Miller, Hierarchical Z-buffer visibility, *Proc. SIGGRAPH 93*, In *Comput. Graph.* Proceedings, Annual Conference Series, ACM SIGGRAPH, 1993, pp. 231–238.

[54] D. Haussler and E. Welzl, Epsilon-nets and simplex range queries, *Discrete Comput. Geom.*, 2 (1987), 127–151.

[55] M. Held, J. T. Klosowski, and J. S. B. Mitchell, Evaluation of collision detection methods for virtual reality fly-throughs, *Proc. 7th Canad. Conf. Comput. Geom.*, 1995, pp. 205–210.

[56] C. Hoffmann, *Geometric and Solid Modeling*, Morgan-Kaufmann, San Mateo, CA, 1989.

[57] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick, Towards implementing robust geometric computations, *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, 1988, pp. 106–117.

[58] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang, Accelerated occlusion culling using shadow frusta, *Proc. 13th ACM Sympos. Comput. Geom.*, 1997, pp. 1–10.

[59] M. J. Katz, M. H. Overmars, and M. Sharir, Efficient hidden surface removal for objects with small union size, *Comput. Geom. Theory Appl.*, 2 (1992), 223–234.

[60] B. W. Kernighan and C. J. V. Wyk, Extracting geometrical information from architectural drawings, *Proc. Workshop on Applied Computational Geometry*, May 1996, pp. 82–87.

[61] D. Khorramabadi, A walk through the planned CS building, Tech. Rep. UCB/CSD 91/652, Computer Science Dept., University of California at Berkeley, 1991.

[62] D. Kim, L. J. Guibas, and S. Shin, Fast collision detection among multiple moving spheres, *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, 1997, pp. 373–375.

[63] R. Laurini and F. Milleret-Raffort, Topological reorganization of inconsistent geographical databases: a step towards their certification, *Computer and Graphics*, 18 (1994), 803–813.

[64] H.-P. Lenhof and M. Smid, Maintaining the visibility map of spheres while moving the viewpoint on a circle at infinity, *Algorithmica*, 13 (1995), 301–312.

[65] D. Luebke and C. Georges, Portals and mirrors: Simple, fast evaluation of potentially visible sets, *1995 Symposium on Interactive 3D Graphics*, ACM SIGGRAPH, 1995, pp. 105–106.

[66] C. Mata and J. S. B. Mitchell, Approximation algorithms for geometric tour and network design problems, *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995, pp. 360–369.

[67] M. McKenna, Worst-case optimal hidden-surface removal, *ACM Trans. Graph.*, 6 (1987), 19–28.

[68] J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal, InfiniteReality: A real-time graphics system, *Proc. SIGGRAPH 97*, In *Comput. Graph.* Proceedings, Annual Conference Series, ACM SIGGRAPH, August 1997, pp. 293–302.

[69] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, New York, NY, 1995.

[70] E. Mücke. Comments on the Computational Geometry Impact Task Force Report. At `http://www.cs.duke.edu/~jeffe/compgeom/files/mucke.html`, June 1996.

[71] K. Mulmuley, Hidden surface removal with respect to a moving point, *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, 1991, pp. 512–522.

[72] J. R. Munkres, *Topology: A first course*, Prentice Hall, Englewood Cliffs, NJ, 1975.

[73] B. Naylor, Constructing good partitioning trees, *Proc. Graphics Interface '93*, 1993, pp. 181–191.

[74] B. Naylor and W. Thibault, Application of BSP trees to ray-tracing and CSG evaluation, Technical Report GIT-ICS 86/03, Georgia Institute of Tech., School of Information and Computer Science, February 1986.

[75] B. F. Naylor, SCULPT: an interactive solid modeling tool, *Proc. Graphics Interface '90*, 1990, pp. 138–148.

[76] B. F. Naylor, Interactive solid geometry via partitioning trees, *Proc. Graphics Interface '92*, 1992, pp. 11–18.

[77] B. F. Naylor, Partitioning tree image representation and generation from 3D geometric models, *Proc. Graphics Interface '92*, 1992, pp. 201–212.

[78] B. F. Naylor, J. Amanatides, and W. C. Thibault, Merging BSP trees yields polyhedral set operations, *Proc. SIGGRAPH 90*, *Comput. Graph.*, Vol. 24, ACM SIGGRAPH, 1990, pp. 115–124.

[79] M. Overmars and M. Sharir, Output-sensitive hidden surface removal, *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, 1989, pp. 598–603.

[80] M. H. Overmars and A. F. van der Stappen, Range searching and point location among fat objects, *J. Algorithms*, 21 (1996), 629–656.

[81] M. S. Paterson and F. F. Yao, Efficient binary space partitions for hidden-surface removal and solid modeling, *Discrete Comput. Geom.*, 5 (1990), 485–503.

[82] M. S. Paterson and F. F. Yao, Optimal binary space partitions for orthogonal objects, *J. Algorithms*, 13 (1992), 99–113.

[83] M. Pellegrini, Repetitive hidden surface removal for polyhedra, *J. Algorithms*, 21 (1996), 80–101.

[84] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, NY, 1985.

[85] F. P. Preparata and J. S. Vitter, A simplified technique for hidden-line elimination in terrains, *Internat. J. Comput. Geom. Appl.*, 3 (1993), 167–181.

[86] F. P. Preparata, J. S. Vitter, and M. Yvinec, Output-sensitive generation of the perspective view of isothetic parallelepipeds, *Algorithmica*, 8 (1992), 257–283.

[87] J. H. Reif and S. Sen, An efficient output-sensitive hidden-surface removal algorithms and its parallelization, *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, 1988, pp. 193–200.

[88] G. Schaufler and W. Sturzlinger, A three-dimensional image cache for virtual reality, *Computer Graphics Forum*, 15 (1996), C227–C235, C471–C472.

[89] R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp, Study for applying computer-generated images to visual simulation, Tech. Rep. AFHRL–TR–69–14, U.S. Air Force Human Resources Laboratory, 1969.

[90] O. Schwarzkopf and J. Vleugels, Range searching in low-density environments, *Inform. Process. Lett.*, 60 (1996), 121–127.

[91] M. Segal, Using tolerances to guarantee valid polyhedral modeling results, *Proc. SIGGRAPH 90*, *Comput. Graph.*, Vol. 24, ACM SIGGRAPH, 1990, pp. 105–114.

[92] R. Seidel, Backwards analysis of randomized geometric algorithms, in: *New Trends in Discrete and Computational Geometry* (J. Pach, ed.), Springer-Verlag, Heidelberg, 1993, pp. 37–68.

[93] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder, Hierarchical image caching for accelerated walkthroughs of complex environments, *Proc. SIGGRAPH 96*, In *Comput. Graph.* Proceedings, Annual Conference Series, ACM SIGGRAPH, Addison Wesley, 1996, pp. 75–82.

[94] X. Sheng and I. R. Meier, Generating topological structures for surface models, *IEEE Computer Graphics and Applications*, 15 (1995), 35–41.

[95] J. R. Shewchuk, Robust adaptive floating-point geometric predicates, *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, 1996, pp. 141–150.

[96] D. M. Y. Sommerville, *Analytical Geometry in Three Dimensions*, Cambridge University Press, Cambridge, 1951.

[97] O. Sudarsky and C. Gotsman, Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality, *Computer Graphics Forum*, 15 (1996), C249–C258.

[98] O. Sudarsky and C. Gotsman, Output-sensitive rendering and communication in dynamic virtual environments, *Proceedings of the Symposium on Virtual Reality Software and Technology*, September 1997.

[99] K. Sugihara and M. Iri, A solid modelling system free from topological inconsistency, *J. Inform. Proc.*, 12 (1989), 380–393.

[100] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden-surface algorithms, *ACM Comput. Surv.*, 6 (1974), 1–55.

[101] S. J. Teller, *Visibility Computations in Densely Occluded Polyhedral Environments*, Ph.D. Thesis, Dept. of Computer Science, University of California, Berkeley, 1992.

[102] W. C. Thibault and B. F. Naylor, Set operations on polyhedra using binary space partitioning trees, *Proc. SIGGRAPH 87*, *Comput. Graph.*, Vol. 21, ACM SIGGRAPH, 1987, pp. 153–162.

[103] E. Torres, Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes, *Eurographics '90*, North-Holland, 1990, pp. 507–518.

[104] A. F. van der Stappen, *Motion Planning amidst Fat Obstacles*, Ph.D. Dissertation, Dept. Comput. Sci., Utrecht Univ., Utrecht, Netherlands, 1994.

[105] C. Yap, Towards exact geometric computation, *Comput. Geom. Theory Appl.*, 7 (1997), 3–23.

[106] D. M. Young, *Iterative solution of large linear systems*, Academic Press, New York, NY, USA, 1971.

[107] J. Yu, *Exact Arithmetic Solid Modeling*, Ph.D. Thesis, Purdue University, CS Dept., West Lafayette, IN 47907, USA, December 1991.

[108] H. Zhang, D. Manocha, T. Hudson, and K. Hoff, Visibility culling using hierarchical occlusion maps, *Proc. SIGGRAPH 97*, In *Comput. Graph. Proceedings*, Annual Conference Series, ACM SIGGRAPH, August 1997, pp. 77–88.