TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER
FACULTY OF EXACT SCIENCES
SCHOOL OF COMPUTER SCIENCE

Improved Output-Sensitive Construction of Vertical Decompositions of Triangles in Three-Dimensional Space

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree in the School of Computer Science, Tel-Aviv University

by

Hayim Shaul

The research work for this thesis has been carried out at Tel-Aviv University
under the supervision of Prof. Dan Halperin

August 2001

Acknowledgments

I would like to thank my advisor Prof. Dan Halperin, for directing me and giving me a helping hand, and for his clarifying comments and reformulations.

I would like to thank my beloved Mom and Dad, for their support.

Abstract

An arrangement of a set S of triangles in space is the division of space into maximal connected regions, each being the intersection of some subset of S. Arrangements have been studied extensively in computational geometry, both combinatorially and algorithmically. Arrangements are useful for solving many "real world" problems. For example, the problem of translational motion planning for a robot in 3-space (under the assumptions, which we shall soon relax, that the shape of the robot and the obstacles consist of triangles). The configuration space of the problem, that is, the space of all possible positions for the robot, can be represented as an arrangement of triangles. The arrangement can be viewed as a graph $G = (V, E)$, where each node corresponds to a three-dimensional cell and two nodes are connected if the robot can pass from one cell to another without passing through other cells or triangles. The problem becomes finding the cell that contains the starting point and the cell that contains the destination point and then finding a path in the graph between the two corresponding nodes. Alas, planning motion inside a cell might be very complicated since the cell itself might be complex. The arrangement can be refined by adding planar facets. The refined arrangement is called a decomposition, and in the case where we add vertical facets it is called a vertical decomposition. If cells in the decomposition are simple enough the motion planning problems becomes a simple path finding problem on a graph.

Arrangements and decompositions of triangles in three-dimensional space have drawn special attention, since surfaces can be approximated with polygonal surfaces which in turn are triangulated. Thus we can relax the assumption we have made above about the shape of the robot and the obstacles.

This work deals with vertical decompositions of n triangles in three-dimensional space. We describe a new decomposition scheme for triangles in three-dimensional space which in most cases produces less cells than the standard vertical decomposition. A cell in this decomposition might have complexity $\Theta(n)$, however each of the cells is convex. We show by experiments that in practice our new scheme is more favorable than the standard vertical decomposition.

We give a deterministic output-sensitive algorithm for computing the standard decomposition that runs in $O(n \log^2 n + V \log n)$, where V is the complexity of the decomposition. This is a significant improvement over the best previously known algorithm whose running time is $O(n^2 \log n + V \log n)$.

We also give a deterministic output-sensitive algorithm for computing the new decomposition we describe that runs in $O(n \log^2 n + V \log n)$, where V is the complexity of the decomposition.

We implemented the two algorithms and ran them on a series of scenes. We ran programs that use decompositions for their calculations and tried them on the output of the two decompositions. Our results show that programs usually benefit from the new decomposition we propose here. Our algorithms make use of a dynamic point location data structure. By using a trivial point location in restricted number of places, the algorithms become very simple and effective (in particular they perform only a space sweep, and all the computations are done in two dimensions on the sweep plane).

We also show how to extend these algorithms to the case of a vertical decomposition of polyhedral surfaces. We implemented these extensions as well.

Contents

1 Introduction **7**

2 Preliminaries **15**
2.1 Two Dimensions 15
2.2 Three Dimensions 16
2.2.1 General Position 17
2.2.2 Vertical Walls 17

3 Vertical Decompositions **19**
3.1 Preliminaries 19
3.2 The Full Decomposition 20
3.3 The Partial Decomposition 22

4 Vertical Decomposition Algorithms **27**
4.1 Output Representation 27
4.2 The Partial Decomposition 28
4.3 The Full Decomposition 36

5 Implementation Details **41**
5.1 Exact Computation 41
5.1.1 The Depth of the Algorithms 42
5.2 General Position and Perturbation 44
5.3 Output Representation 44
5.4 Program Checking 45
5.4.1 Cell Check 45
5.4.2 Neighbor Check 45

6 Results **47**
6.1 Complexity of Decompositions 47
6.2 Calculating Volumes 55
6.3 Finding a Free Path 59

7 Arrangement of Polyhedral Surfaces **61**
7.1 Additional Degeneracies 62
7.2 Events During the Sweep 62

8 Conclusions **65**

A Detecting Events **67**

B Handling Events **73**

C Detecting Polyhedral Surfaces Events **79**

D Handling Polyhedral Surfaces Events **83**

Chapter 1

Introduction

The study of arrangements of curves or surfaces is an important area of research in computational geometry. This is due to the fact that many geometric problems in diverse areas can be reduced to problems involving arrangements.

A collection S of n surfaces decomposes the Euclidean space $\mathbb{R}^d$ into open *cells* of dimension d (also called *d-faces*) and into relatively open *faces* of dimension k, for $0 \leq k < d$. Each face or cell is a maximal connected portion of the intersection of some subset S_0 of S which does not meet any other surface in $S \backslash S_0$. The resulting collection of cells form a partition of $\mathbb{R}^d$ known as the *arrangement* $A(S)$ of S. The *combinatorial complexity* of the arrangement is defined to be the total number of cells of all dimensions of $A(S)$. The simplest example of an arrangement is when S is a set of lines in the plane; such an arrangement is shown in Figure 1.1. For additional basic terminology related to arrangements, the reader is referred to [4, 23, 32, 33, 49]

The surfaces in S are usually assumed to have *constant description complexity*, such as the graphs of (possibly partially defined) low-degree algebraic functions, whose domains of definition are semi-algebraic sets defined by a constant number of polynomial equalities and inequalities of constant maximum degree.

Arrangements of triangles in $\mathbb{R}^3$ have drawn special attention, since surfaces can be approximated with polygonal surfaces which in turn are triangulated. This thesis deals with arrangements of triangles in $\mathbb{R}^3$.

A classical example of a problem that reduces to the study of arrangements is *the motion planning problem* for a robot. In this example, the robot

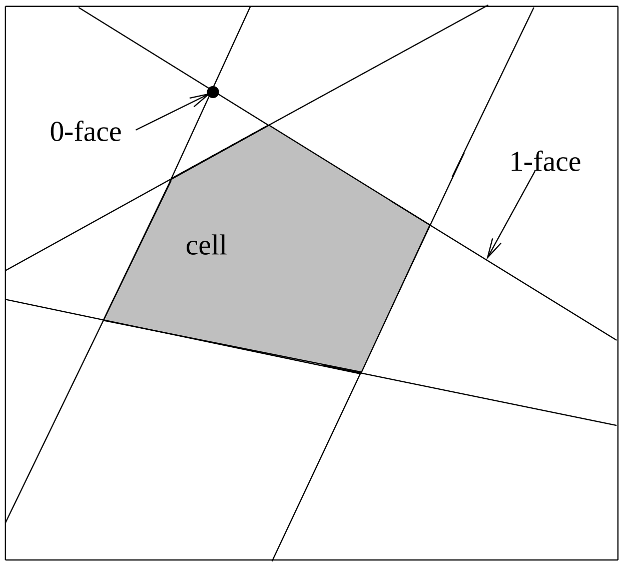

Figure 1.1: An arrangement of lines in the plane

is moving among obstacles. For simplicity assume that the robot is a point in $\mathbb{R}^3$ and the obstacles are triangles. The robot can move freely in space, but it cannot cross through the triangles. The robot is initially positioned in some location in space and it is given a target which it should reach. One way to find a collision free path for the robot, from the initial location to the target, is to decompose the space into cells, and construct a graph $G = (V, E)$ where each node represents a cell, and two nodes are connected with an edge if the robot can move between the cells they represent without crossing another cell, or a triangle. Motion planning becomes a simple path finding on a graph, from the node corresponding to the cell that contains the source point, to the node that contains the target point. In the study of robot motion planning, G is called a *Connectivity Graph* as it captures the connectivity of the free space.

Notice, however, that we still need to plan how the robot moves inside cells. These cells can be very complex and planning a motion inside them might not be trivial at all. For most algorithmic uses a raw arrangement is an unwieldy structure. The difficulty is that cells in an arrangement can have very complex topologies, so navigating around them is difficult. What we often want is a further refinement of the cells into convex pieces that are each homeomorphic to a ball. Ideally, the number of cells after the refinement should be proportional to the overall complexity of the arrangement. For arrangements of hyperplanes the well-known bottom vertex triangulation [19] meets this criterion. For more general arrangements such refined decompositions are more difficult to find.

The simplest way to decompose such an arrangement is to compute the bottom vertex triangulation of the arrangement of the planes supporting the triangles. The resulting decomposition has size $\Theta(n^3)$, which is optimal in the worst case. In many applications, however, the actual complexity of the arrangement of triangles is much smaller. So the challenge is to obtain a decomposition whose size is sensitive to the complexity of the arrangement of the triangles.

Such a complexity-sensitive decomposition was given by Aronov and Sharir [5]: their Slicing Theorem states that an arrangement of n triangles in space can be decomposed into $O(n^2\alpha(n) + K)$ tetrahedra, where K is the complexity of the arrangement. This result is close to optimal: $\Omega(K)$ is clearly a lower bound on any decomposition, and Chazelle [11] shows that there are arrangements of complexity $O(n)$ such that any decomposition into convex cells has size $\Omega(n^2)$. (The triangles in Chazelle's example form the boundary of a simple polytope.) The Slicing Theorem obtains a decomposition by adding vertical walls for each of the triangle boundary edges, one after the other. The wall of an edge e is obtained by "flooding" the zone of e in an arrangement on the vertical plane $H(e)$ containing e; this arrangement is defined by intersections of $H(e)$ with the triangles and added walls. After adding the walls one is left with convex cells that can easily be decomposed into tetrahedra. The Slicing Theorem decomposition has the unpleasant characteristic that it depends on the order in which triangle boundary edges are treated. Thus the tetrahedra in the decomposition are not defined "locally", and it is not canonical in the sense of Chazelle and Friedman [13]. This means that the decomposition is not very well suited for randomized incremental algorithms. It also makes it difficult to compute the decomposition efficiently.

A decomposition which does not have this problem is the following [18, 21, 40, 41]. This decomposition is also obtained by erecting vertical walls. This time the wall for edge e simply consists of those points in $H(e)$ that can be connected to e with a vertical segment that does not cross any of the triangles in T. Secondly, walls are erected from the intersection edges between pairs of triangles to produce a finer decomposition. Observe that the wall erected from an edge is not obstructed by other walls, so the decomposition does not depend on the order in which the edges are treated. We call this decomposition the *vertical decomposition* for T. Note that cells in the vertical decomposition need not be convex; in fact, they need not even be simply connected. However, the decomposition can easily be refined into a subdivision in which each cell is convex.

For a long time the best known bound on the maximum combinatorial complexity of the vertical decomposition of an arrangement of n "well-behaved" algebraic surfaces (see, e.g., [49] for a precise definition) in $\mathbb{R}^d$ for fixed d was $O(n^{2d-3}\beta(n))$ [12], where $\beta(n)$ is a slowly growing function of n, depending also on d and on the degree of the given surfaces. (Notice that the maximum complexity of the underlying arrangement is $\Theta(n^d)$.) Recently, Koltun improved this bound for fixed $d \geq 4$ to $O(n^{2d-4+\epsilon})$ by showing that the maximum complexity of the vertical decomposition of an arrangement of n fixed-degree algebraic surfaces or surface patches in four dimensions is $O(n^{4+\epsilon})$ [36].

In $\mathbb{R}^3$ it is easy to show that the maximum complexity of the vertical decomposition of an arrangement of well-behaved surfaces is $O(n^2\lambda_q(n))$ where $\lambda_q(n)$ is a near-linear function related to Davenport-Schinzel sequences [49]. Namely, in $\mathbb{R}^3$, the maximum complexity of the vertical decomposition is very close to that of the underlying arrangement. For arrangements of triangles, which are the focus of our thesis, de Berg *et al.* showed that the maximum complexity of the vertical decomposition is the same as that of the arrangement, that is $\Theta(n^3)$ [21]. They also showed that the complexity of the vertical decomposition in the case of triangles is $O(n^{2+\epsilon} + K)$ where K is the complexity of the arrangement; the near-quadratic overhead term is close to optimal as there are arrangements with linear complexity whose vertical decomposition has quadratic complexity. For more combinatorial bounds on the complexity of the vertical decomposition of arrangements, see [4, 32, 49].

In this thesis we present a new refinement, and compare it to the standard vertical decomposition studied in [21]. We call the decomposition which was studied in [21] *the full decomposition.* The full decomposition produces convex cells of constant complexity each. The decomposition we present here, which we call *the partial decomposition*, produces convex cells that might have $\Theta(n)$ complexity. The partial decomposition is similar to the two-dimensional partial vertical decomposition discussed in [34]. Here we extend vertical facets (*walls*) from some of the features of the arrangement to get convex cells that might have $\Theta(n)$ complexity.

The algorithm described in [21] to compute the decomposition of triangles in $\mathbb{R}^3$ runs in $O(n^2 \log n + V \log n)$ time, where V is the complexity of the decomposition. The algorithm was extended to compute the vertical decomposition of arrangements of n algebraic surface patches of constant maximum degree in $\mathbb{R}^3$ in time $O(n\lambda_q(n) \log n + V \log n)$, where V is the combinatorial complexity of the vertical decomposition, $\lambda_q(n)$ is a near-linear function related to Davenport-Schinzel sequences, and q is a constant that depends on the degree of the surface patches and their boundaries.

The overhead of the algorithm is the time we always need, irrespective of the complexity of the decomposition. The overhead of the algorithm in [21] is $O(n^2 \log n)$. An algorithm that achieves a sub-quadratic overhead time was shown in [21]. This algorithm uses multi-level data structures (for ray shooting and similar problems) [2, 3, 20]. It also uses a standard trick [22, 42] of "guessing" the output size, which is necessary to determine the amount of preprocessing to be spent. This algorithm is, however, substantially more complicated than the algorithm mentioned above, and the savings are small. This algorithm runs in time $O(\min(n^{4/5+\epsilon}V^{4/5}, n^2 \log n) + V \log n)$, for any $\epsilon > 0$, where V is the combinatorial complexity of the vertical decomposition. The latter algorithm is faster for $V = O(n^{3/2})$.

In this thesis we present algorithms for constructing the full decomposition and the partial decomposition which run in $O(n \log^2 n + V \log n)$ where V is the complexity of the output decomposition. These algorithms significantly improve the overhead for computing such decompositions while still being fairly simple. One non-trivial component that our new algorithm requires is a data structure for dynamic point location. In our implementation we substitute this component by a naive method which is applied exactly (and only) n times.

The algorithms we implemented have only one step (a space sweep), in contrast with the algorithm presented in [21] which has three steps (building lower and upper envelopes, a space sweep and building two dimensional arrangements). An algorithm which has only one step is more easy to implement, more easy to debug.

The algorithms we present make use of dynamic planar point location. Planar point location is a fundamental geometric searching problem and has been extensively studied. Given a subdivision S of the plane into polygonal regions, we want to perform on-line queries that ask for the region of S containing a given query point. In the *static* case, where S is fixed, there are optimal techniques that achieve $O(\log n)$ query time using $O(n \log n)$ preprocessing time and $O(n)$ space [24, 35, 48], where n is the size of S. Research on dynamic algorithms for geometric problems has received increasing attention in the last years. See [6, 7, 14, 15, 17, 28, 44] for important results in the subject, and [16] for a survey on the subject.

Goodrich and Tamassia [27, 29] show how to dynamically maintain a monotone subdivision so as to achieve $O(\log^2 n)$ query time, $O(\log n)$ time for vertex insertion and deletion, and $O(\log n + k)$ time for the insertion and deletion of a monotone chain of k edges. Their methods are based on the maintenance of two interlaced spanning trees, one for the subdivision and one for its graph-theoretic dual, to answer queries. Queries are performed by using a centroid decomposition of the dual tree to drive searches in the primal tree. Goodrich and Tamassia dynamized this approach using the edge-ordered dynamic tree data structure of Eppstein *et al.* [25], which is an extension of the link-cut trees data structure of Sleator and Tarjan [25, 51].

We use this result for implementing point location during the space sweep. The point location is done on the arrangement induced by triangles and walls on the sweep plane. This arrangement changes during the sweep and the point location data structure is updated respectively in $O(\log n)$ time per update. When we handle an event during the sweep we do a point location query to locate the face in which the event takes place. This query takes $O(\log^2 n)$ time, whereas we can locate the face with our own data structure in $O(\log n)$ time for most of the events. However, our data structure needs prior information on the triangles that participate in the event, therefore it is useless when we encounter a new triangle during the sweep. In such cases we do a query on the point location data structure. Overall these queries require $O(n \log^2 n)$ time. The sweep itself takes $O(V \log n)$ where V is the complexity of the output. And the algorithm runs in $O(n \log^2 n + V \log n)$ time.

We have also implemented the two algorithms for the two types of vertical decompositions. Implementors of geometric algorithms and data structures encounter several difficulties. Geometric algorithms are usually described assuming an infinite-precision real arithmetic model of computation. In this model, arithmetic operations, assignments and comparisons on real numbers take constant time. A program implemented using a naive substitution of floating-point arithmetic for real arithmetic can fail, since geometric primitives depend on sign-evaluation and may not be reliable if evaluated approximately. In [39], an example is given where using a floating point implementation of the orientation predicate causes a convex hull algorithm to give a wrong result. In many cases the program will go into an infinite loop or crash. For geometric algorithms that are restricted to rational computation (e.g., orientation and lexicographical comparison of input points), the infinite-precision real model can be realized using multi-precision integer or rational software packages (e.g. Gmpz [30] and LEDA's rational type [37]). Given a constant bound on the input bit length the predicates take $O(1)$ time to compute. However, the underlying constant may be very large.

A strategy to reduce the cost of exact computation is the use of *floating point filters*, which is a general name for a variety of adaptive techniques in which expressions are evaluated with floating point arithmetic, together with a bound on the error of the computation. Exact computation is used only when the result of the floating point computation is insufficient. There are many implementations of such filters, some are specifically tailored for a given predicate (e.g. [50] and the predicates in LEDA's rational kernel [37, 39]) and some are more general and can be used for evaluating general expressions (e.g. [26] and LEDA's real number type [9, 37, 39]). Implementations that evaluate the error bound at compile-time are called *static filters* [26] and those that evaluate the error at run-time are called *dynamic filters*.

The two decompositions mentioned above, the full and the partial decomposition were implemented as part of the thesis work. We have compared the time it took to construct each of the two decomposition. As expected, constructing the partial decomposition was much faster than constructing the full decomposition. We have also run two programs that employ the decompositions. One that finds a free path between two points in the space, and one that calculates the volume of the cells in the decomposition (a means to check the correctness of the program). We compared the time it took each program to compute on the partial decomposition and on the full decomposition. Most of the times the programs ran faster when the input was the partial decomposition. This result means that having a small number of cells compensates for their complexity. However, there were scenes where the programs ran faster on the full decomposition.

The rest of the thesis is organized as follows. In Chapter 2 we introduce the basic assumptions and terminology that are used throughout the thesis. The vertical decomposition schemes are introduced in Chapter 3. In Chapter 4 we describe the algorithms to compute the decompositions described in Chapter 3. A discussion of the implementation of these algorithm is given in Chapter 5. In Chapter 6 we give results of several tests we ran on the two decompositions and discuss these results. We also extend these two algorithms to compute the vertical decomposition of polyhedral surfaces. This extension is given in Chapter 7. Concluding remarks and suggestions for future directions are given in Chapter 8.

Chapter 2

Preliminaries

2.1 Two Dimensions

A point $p \in \mathbb{R}^2$ is above (below) a point $q \in \mathbb{R}^2$ if the segment $\overline{pq}$ is vertical and p is above (below) q, namely has a higher (smaller) y coordinate.

Let $C = \{c_1, c_2, ..., c_n\}$ be a collection of Jordan arcs in the xy-plane, such that each arc is x-monotone (i.e., every line parallel to the y-axis intersects an arc in at most one point). For defining the lower (upper) envelope we regard each curve c_i in C as the graph of a continuous univariate function $c_i(x)$ defined on an interval. The lower envelope Ψ of the collection C is the point-wise minimum of these functions: $\Psi(x) = \min c_i(x)$, where the minimum is taken over all functions defined in x. Similarly, the upper envelope of the collection C is defined as the point-wise maximum of these functions. See Figure 2.1 for an illustration of the lower envelope of a collection of line segments. Let S be a set of arcs, and let $A(S)$ denote the arrangement induced by S, namely, the subdivision of $\mathbb{R}^2$ into cells of dimensions 0,1 and 2, induced by the arcs in S. We call cells of dimension 0 vertices. A vertex is either an arc endpoint (an *outer vertex*), or an intersection of two arcs (an *inner vertex*). We assume that the arcs are in general position, that is, no arc intersects another arc at its endpoint. See Figure 2.1 for an illustration. In this example v_1 is an inner vertex and v_2 is an outer vertex.

We refer to an arc by *edge*. An *segment* is a maximal connected portion of an arc not intersecting any other arc in S. A *face* is a maximal connected region of the plane not intersecting an edge or a vertex.

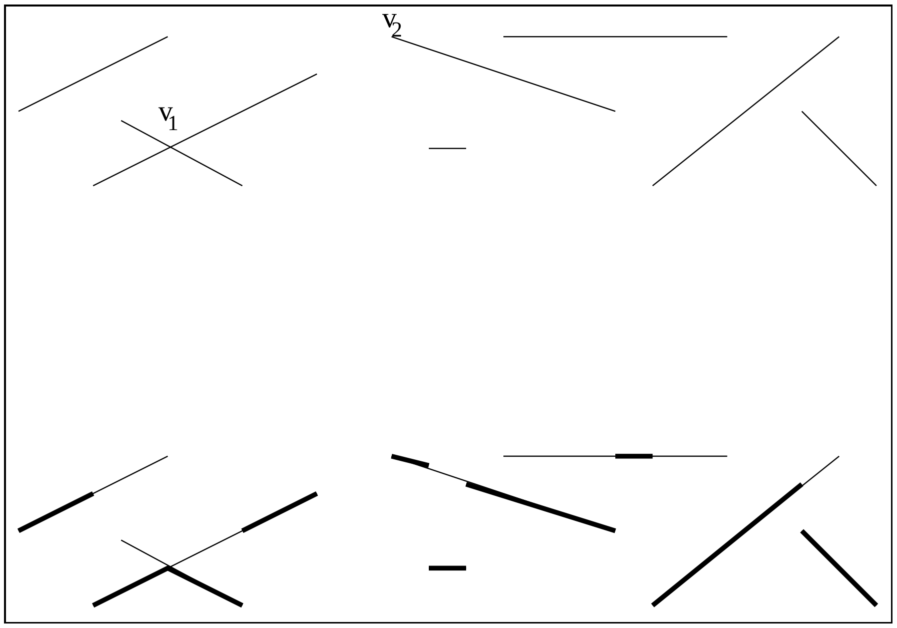

Figure 2.1: A set of line segments, and (in bold line) its lower envelope; v_1 is an inner vertex, v_2 is an outer vertex

2.2 Three Dimensions

We refer to the positive z direction as *up*, and to the positive y direction as *right*.

Let $T = \{t_1, t_2, ..., t_n\}$ be a collection of n (possibly intersecting) triangles in $\mathbb{R}^3$.

We use the following terminology: a triangle has three *boundary edges* which meet at the triangle's *corners*. We say that a corner is *before* another corner if it has a lower x coordinate, or if it has the same x coordinate and a lower y value. The corner of a triangle that is before the other corners is called the *first corner*. The corner having the other corners before it is called the *last corner*. The third corner is called the *middle corner*.

We say that the middle corner bounds the triangle from the left (right) if every point in the intersection of the triangle with a line parallel to the y-axis that passes through the middle point, has a bigger (smaller) y value than the middle point.

The intersection of two triangles is called *intersection edge*.

Two points $p, q \in \mathbb{R}^3$ are *vertically visible* with respect to T if the segment

$\overline{pq}$ connecting them is vertical and the relative interior of $\overline{pq}$ does not intersect any triangle in T. Usually the set T is clear from the context and we just say that p and q are vertically visible.

This definition is extended to objects other than points as follows: two sets $P, Q \subset \mathbb{R}^3$ are vertically visible if there are points $p \in P, q \in Q$ that are vertically visible.

If two sets P, Q are vertically visible we define their vertical distance, as the length of the shortest segment $\overline{pq}$ $p \in P, q \in Q$, such that $\overline{pq}$ is vertical.

2.2.1 General Position

Let $A(T)$ denote the arrangement induced by T, namely, the subdivision of $\mathbb{R}^3$ into cells of dimensions 0,1,2 and 3 induced by the triangles in T.

We make the same general position assumption as in [5]:

1. No triangle is parallel to the z-axis.
2. At most three triangles intersect at a point.
3. No two boundary edges intersect.
4. No corner of one triangle lies on another triangle.
5. No boundary edge of one triangle intersects the intersection of two additional triangles.
6. No line parallel to the z-axis intersects three boundary edges.
7. No line parallel to the z-axis intersects a vertex of one triangle and a boundary edge of another triangle.
8. No line parallel to the z-axis intersects a boundary edge of one triangle and an intersection of a boundary edge of another triangle with a third triangle.

2.2.2 Vertical Walls

Let the vertical wall extended from a three-dimensional segment s, denoted $W(s, T)$, be defined as follows: $W(s, T) = \{p \in \mathbb{R}^3 : p$ and s are vertically visible$\}$. In other words, $W(s, T)$ is the union of all the vertical segments of maximal length that have a point of s as an endpoint and whose interior does not intersect any triangle in T. Note that some of these segments can be rays.

Let T' be a collection of n triangles and m vertical walls as defined in the previous section. Let the vertical wall extended from a three-dimensional point p, denoted $W(p, T')$, be defined as follows: $W(p, T') = \{q \in \mathbb{R}^3 : q$ and p can be connected by an arc that is contained in a plane parallel to the yz-plane which does not intersect any feature in $T'\}$. In other words, $W(p, T')$ is a vertical flooding of faces containing p in their interior or on their boundary on the arrangement obtained from intersecting T' with the vertical plane orthogonal to the x-axis and passing through p.

Also define the left vertical wall extended from a three-dimensional point p, as $W_l(p, T') = \{q \in \mathbb{R}^3 : q \in W(p, T')$ and $q_y < p_y\}$. That is, the part of the wall that is to the left of p. Similarly the right vertical wall is defined as $W_r(p, T') = \{q \in \mathbb{R}^3 : q \in W(p, T')$ and $q_y > p_y\}$.

Chapter 3

Vertical Decompositions

In this chapter we present two variants of vertical decompositions of an arrangement of triangles in three-dimensional space. The first is the well-known standard vertical decomposition. This decomposition is so refined that every cell in it has constant descriptive complexity, and it is in fact a trapezoidal prism. An algorithm for computing it was presented in [21]. The second decomposition is in a sense more economical, with fewer cells, some of which may, however, be rather complex. A cell in the second novel decomposition is guaranteed to be convex, but may have $\Theta(n)$ complexity, where n is the number of triangles. Nevertheless, since each cell in the decomposition of the second type is convex, it is still fairly easy to manipulate it. For example planning a free motion for our point robot between source and target inside such a cell is trivial.

We call the first decomposition (the more refined one) *the full decomposition*, and the second decomposition *the partial decomposition*.

To the best of our knowledge, the partial decomposition for arrangement of triangles has not been studied before. Our motivation to study it comes from a similar two-dimensional decomposition studied in [34, 1] which proved to be efficient in practice.

3.1 Preliminaries

Let $T = \{t_1, t_2, ...t_n\}$ be a collection of n triangles in $\mathbb{R}^3$. To simplify the description of the decompositions we assume that the triangles are inside a bounding simplex, and in general position as defined in Section 2.2.1.

3.2 The Full Decomposition

Consider the decomposition $FD'(T)$ obtained from the arrangement $A(T)$ by erecting a wall $W(b,T)$ from every boundary edge b, and a wall $W(i,T)$ from every intersection edge i. Cells in this decomposition need not be convex; in fact, they need not even be simply connected. Consider for example the scene shown in Figure 3.1. In this scene a horizontal triangle intersects another triangle. Figure 3.2 shows a cell created by this refinement from the arrangement obtained from the scene in Figure 3.1. Here vertical walls have been erected from triangle boundary edges and intersection edges. The cell is a cylinder containing a cylindrical hole. If there is a bounding box the cell is bounded by the ceiling of that box, otherwise it is infinite. The grey area in the upper triangle is the part of the bounding box that contributes to the cell's ceiling. The grey area in the lower triangle is the part of the horizontal triangle that contributes to the cell's floor.

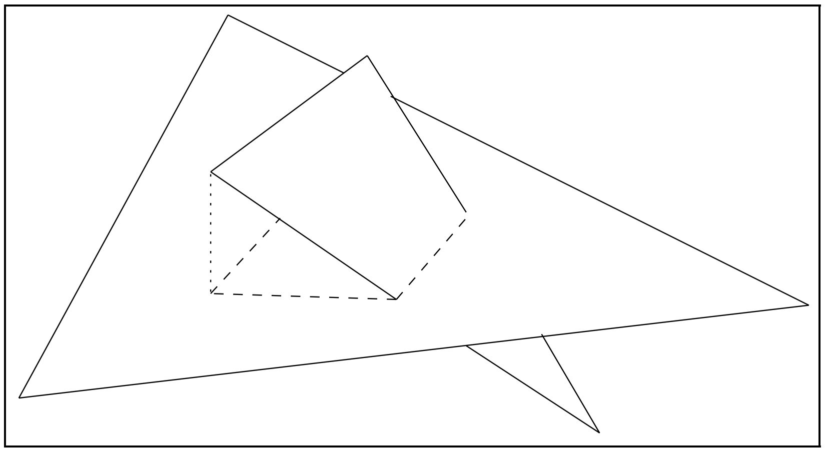

Figure 3.1: $A(T)$

However, the decomposition can easily be refined into a convex subdivision $FD(T)$ where each cell has constant complexity, without increasing the asymptotic complexity of the subdivision.

We consider each triangle to be two-sided, and let t_i^- denote the side of t_i facing downward and t_i^+ be the side of t_i facing upward. We define a two-dimensional arrangement of segments for the upper side of each triangle t_i^+. The arrangement is defined by a set $\Sigma(t_i^+)$ consisting of two types of segments: intersections of t_i with other triangles of T and boundary pieces of

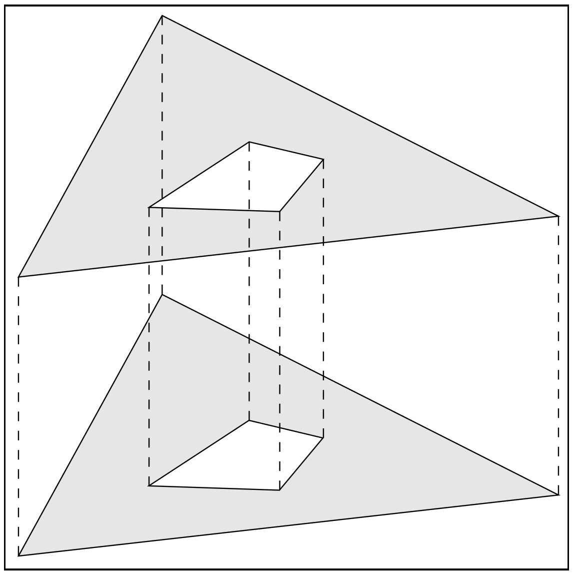

Figure 3.2: A cell in $A(T)$ which is not simply connected

walls $W(e, T)$ (for boundary edges e of triangles in T, or intersection edges i between two triangles in T) that lie on t_i^+. Denote this arrangement as $A(t_i^+)$. Segments in $A(t_i^+)$ of the second type, in other words, are each the contributions of t_i to the lower envelope that bounds some wall $W(e, T)$ from below. Every face of every $A(t_i^+)$ represents a vertical cylindrical cell that has the same xy-projection as that face, and that has a unique triangle bounding it on the top and a unique triangle bounding it on the bottom (which is t_i). A face of $A(t_i^+)$ may still be rather complex: it need not be simply connected, and it may have a large number of edges on its boundary. Consider the projection of $A(t_i^+)$ onto the xy-plane. We extend a y-vertical segment from each vertex of the projected arrangement upward and downward until it reaches another segment, or the boundary of the projection of t_i. In other words, we compute a trapezoidal decomposition of the arrangement. The added segments are projected back to t_i^+ to obtain a refinement of $A(t_i^+)$ into trapezoids. Finally, we extend each of these newly added segments in the z-direction into vertical walls inside the respective three-dimensional cells. We call these walls *yz walls*. Adding these yz walls obtains the full decomposition.

3.3 The Partial Decomposition

Consider the decomposition $PD'(T)$ obtained from the arrangement $A(T)$ by erecting a wall $W(b,T)$ from every boundary edge b. Cells in this decomposition need not be convex nor be simply connected.

Consider for example the scene shown in Figure 3.1. In this scene a horizontal triangle intersects another triangle. Figure 3.3 shows a cell created by this refinement of the arrangement depicted in Figure 3.1. Here vertical walls have been erected from triangle edges. The cell is a cylinder containing internal walls. If there is a bounding box the cell is bounded by the ceiling of that box, otherwise it is infinite. The grey area in the upper triangle is the part of the bounding box that contributes to the cell's ceiling. The grey area in the lower triangle is the part of the horizontal triangle that contributes to the cell's floor.

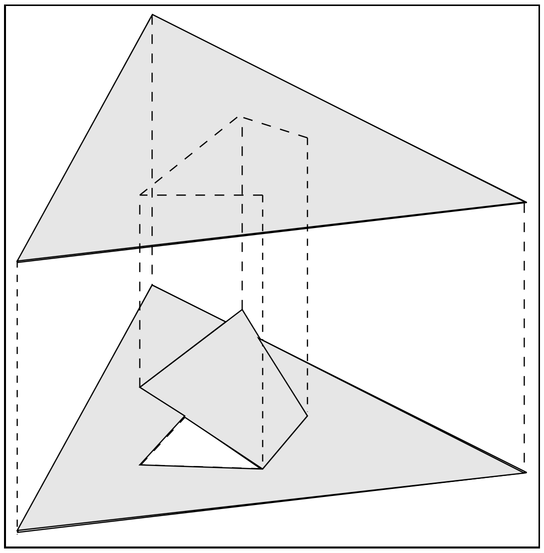

Figure 3.3: A face in $A(T)$ which is not simply connected

However, the decomposition can easily be refined into a subdivision $PD(T)$ where each cell is convex.

We erect vertical walls $W(c, PD'(T))$ from first corners of triangles and from last corners of triangles. These walls split the cell into two parts, separated by a wall parallel to the xz-plane that contains the corner. Such a wall is shaped as the face that results from the intersection of a plane parallel to the xz-plane with the cell that contains the corner. In the partial decomposition such an intersection is always convex. We also erect left vertical walls $W_l(c, PD'(T))$ from middle corners that bound triangles from the left, and right vertical walls $W_r(c, PD'(T))$ from middle corners that bound triangles from the right. We also erect a vertical wall $W(p, PD'(T))$ from points of intersection between a boundary edge of some triangle and another triangle. We refer to this wall erection as *flooding.* A flood wall is shaped as the face that results from the intersection of a plane parallel to the xz-plane with the cell that contains the point that defines the flooding.

The complexity of the partial decomposition can be $\Theta(n^3)$ as the following example, suggested to us by Boris Aronov, shows. Consider the scene described in Figure 3.4 where $n/3$ triangles $\alpha = \{\alpha_1, ..., \alpha_{n/3}\}$ intersect each other to form a convex ceiling, $n/3$ triangles $\beta = \{\beta_1, ..., \beta_{n/3}\}$ lie beneath the ceiling such that the projections of these triangles on the xy-plane do not intersect, and $n/3$, long and skinny triangles $\gamma = \{\gamma_1, ..., \gamma_{n/3}\}$ parallel to the x-axis that stab the triangles in β.

There are $\theta(n^2)$ intersections, each between a boundary edge of a triangle in β and a triangle in γ. We arrange the triangles in β such that all these intersection points have distinct x-coordinates. Each of these intersection points induces a flood wall with complexity $\theta(n)$, since it meets all the triangles in α. Thus the total complexity of all the flood wall is $\theta(n^3)$.

Figure 3.5 shows two dimensional arrangements obtained by intersecting the scene described above and planes parallel to the yz plane that pass through different x values.

Lemma 3.1 *After flooding, the resulting cells of the partial vertical decomposition are convex.*

Proof. It suffices to show that after the flooding there is no point of local non-convexity on a the boundary of any cell in the resulting decomposition. The first step of the partial decomposition eliminated the non-convexity at triangle boundary edges, as we erected vertical walls from them. However, it introduced new "exposed" edges that are the vertical boundary edges of the vertical walls. The flooding takes care of exactly these exposed edges by adding new faces through them. Finally we note that there are no new exposed edges on the boundary of flood faces since by the definition of the flooding process all the edges on the boundary of the flood faces are determined by existing faces of the decomposition. ■

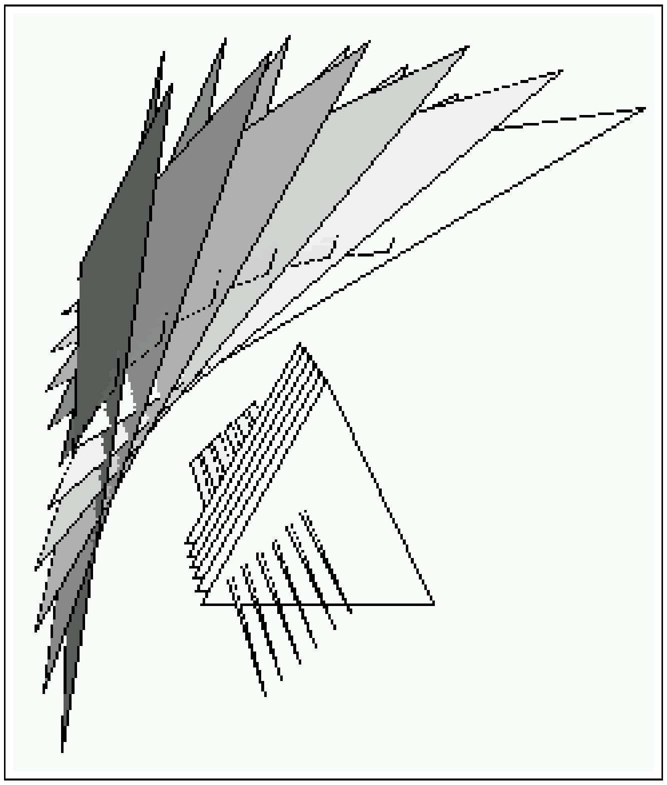

Figure 3.4: Partial vertical decomposition where the floods have total complexity $\Theta(n^3)$

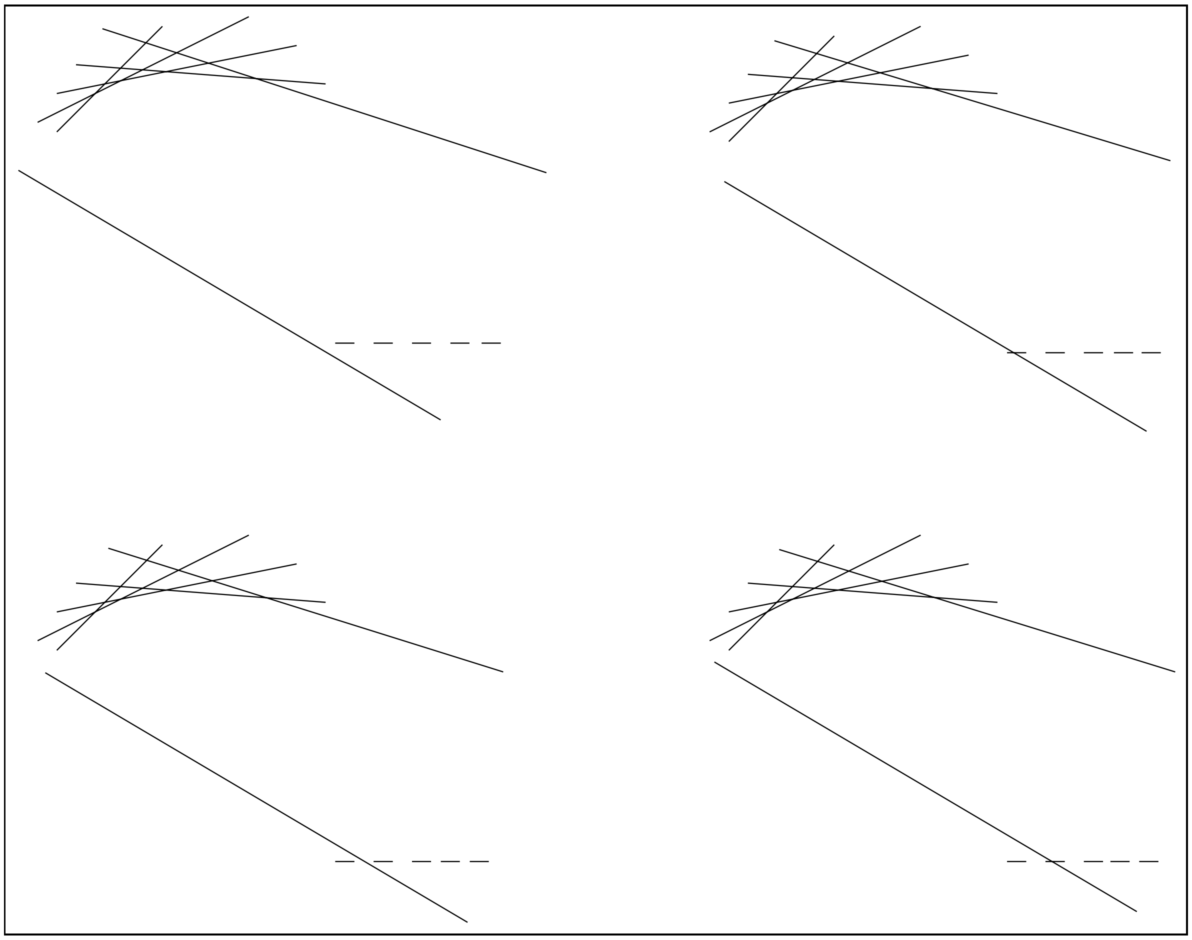

Figure 3.5: The arrangement induced on vertical planes at different x values.

Chapter 4

Vertical Decomposition Algorithms

In this chapter we present algorithms to compute the two variants of vertical decompositions introduced in the previous chapter.

4.1 Output Representation

We represent a vertical decomposition by a graph G. The nodes in G correspond to the cells (vertical prisms) of the decomposition. With each node we store an explicit description of the prism it represents. There is an edge between two prisms if they share a vertical wall. Thus we get a complete "network" that allows us to navigate from one point in a cell of the arrangement $A(T)$ to any other point in that cell. We cannot, however, go from one cell in $A(T)$ to an adjacent cell, because G only stores connections through vertical walls, not through triangles.

We attach the following information to each node v, thus providing a full description of the cell C_v:

1. A list of triangles that contribute to the cell's ceiling.

2. A list of triangles that contribute to the cell's floor.

3. A list of walls that bound the cell from the right.

4. A list of walls that bound the cell from the left.

5. Two walls parallel to the yz-plane that bound the cell between them.

6. A list of pointers to the neighbor cells of C_v.

4.2 The Partial Decomposition

We construct the partial decomposition by performing a space sweep. We sweep a plane P_x parallel to the yz-plane from $x = -\infty$ to $x = \infty$. The goal of the sweep is to erect the walls that refine $A(T)$, as described in Section 3.3, and create the data structure that represents the decomposition, as described in Section 4.1.

Denote the arrangement induced by the intersection of the triangle set T with the sweep plane P_x, by A'_x. Also denote the decomposition induced by the intersection of the triangle set T and the set of walls W extended from each boundary edge, with the sweep plane, by A_x.

Note that A_x is a two-dimensional arrangement on a plane parallel to the yz-plane. Recall that 'up' means the positive z direction, and 'right' means the positive y direction.

We say that A_{x_1} is before (after) A_{x_2} if $x_1 < x_2$ ($x_1 > x_2$).

The decomposition A_x changes continuously as the plane A_x sweeps along space, however its combinatorial structure changes only at a finite number of events:

1. A vertex may appear (disappear).

2. An edge may appear (disappear).

3. A face may appear (disappear).

   Since we are interested in the vertical decomposition of the arrangement we will also keep track of the vertical visibilities between vertices of A_x. So we shall have another type of event:

4. An outer vertex is vertically visible with another vertex (either inner or outer).

The case where two inner vertices vertically see each other is not mentioned here since it is the concern of the full decomposition alone.

During the sweep we maintain a dynamic point location structure described in [27] on the arrangement A_x. We also keep our own data structure to represent A_x. Recall that A_x is already a decomposition of A'_x. All the faces in A_x are convex. We represent a face of A_x by (see Figure 4.1):

1. A chain of *ceiling* triangles.
2. A chain of *floor* triangles.
3. A witness for the right boundary of the face, which is either the intersection of the upper and lower chains, or an endpoint of some edge.
4. Similarly, a witness for the left boundary of the face.

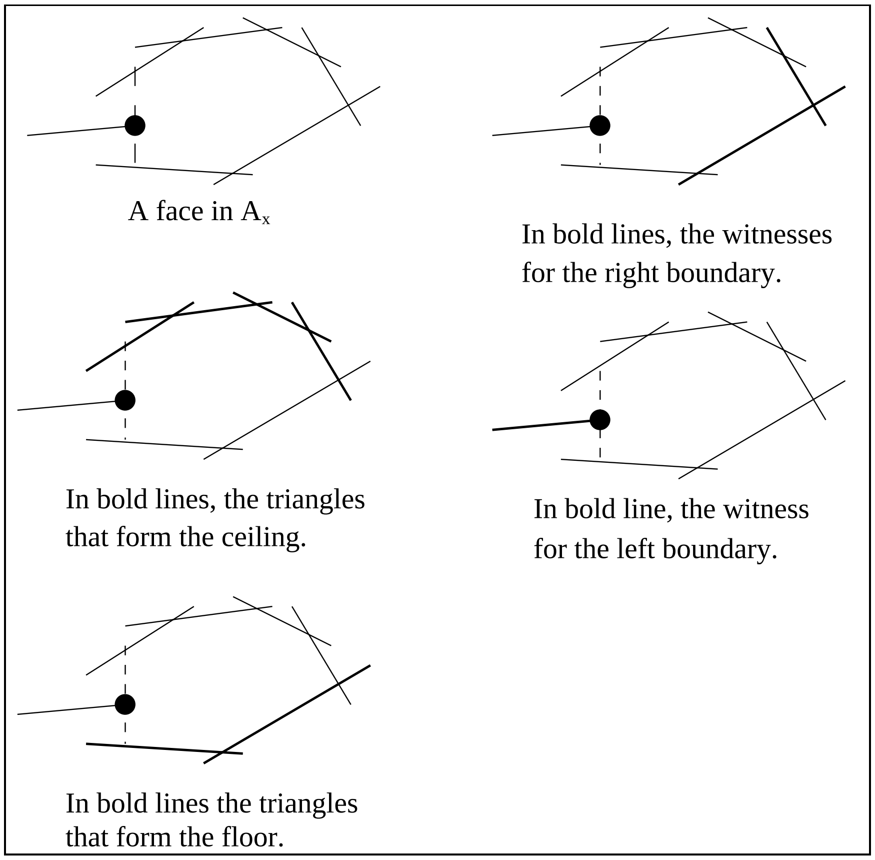
A face in A_x
In bold lines, the witnesses for the right boundary.
In bold lines, the triangles that form the ceiling.
In bold line, the witness for the left boundary.
In bold lines the triangles that form the floor.

Figure 4.1: A face in A_x

Maintaining the dynamic point location data structure during the sweep is done in $O(\log n)$ per update. In order to maintain our data structure during the sweep we need to create, delete, split and merge faces. Creating (deleting) a face is done simply by initializing (deleting) the data structure used to represent its two chains. Splitting a face f into f_1 and f_2 is done by splitting its chains and copying the appropriate witnesses to f_1 and f_2. Splitting the chains can be done in $O(\log n)$ time using, for example, red-black trees [52] or skiplists [45] or any other balanced structure. Notice that skiplist is a randomized structure, hence these operations can be done in expected time $O(\log n)$. Merging two faces f_1 and f_2 into one face f is done similarly by joining their chains and copying the appropriate witnesses. Joining the chains can be done in $O(\log n)$ using the structures mentioned above. See [21] for more information.

We maintain all the events in a priority queue Q, ordered by increasing x-coordinate. The operations we perform on Q are insert an event, delete an event, and fetch the next event, that is fetch the event with minimum x-value (we delete each event after it has been fetched and handled). We also need to perform a membership check of an event in the queue, to avoid inserting the same event several times. Such a queue can be implemented so that the time for each operation is $O(\log m)$, where m is the maximum number of events held simultaneously in the queue [43]. To each event we insert into the queue, we attach the local geometric and combinatorial information relevant to that event.

The sweep we describe here is different than the one described in [21]. The algorithm described here gives a better overhead and runs in $O(n \log^2 n + V \log n)$ time, where V is the complexity of the decomposition. We first make some observations about the events during the sweep and then we prove that our sweep is correct.

We start the sweep by inserting events where new edges appear and events where edges disappear (these events occur on first and last corners of triangles). We also insert events where the sweep plane reaches middle corners of triangles. The latter events do not change the combinatorial description of the arrangement A_x on the sweep plane P_x. It only changes a boundary edge of some triangle, which affects the way an outer vertex "moves" on the sweep plane.

As we sweep we discover more events and insert them into the queue. See Appendix A for a full list of the events and how they are detected.

When we come to handle an event we first have to find the faces that are changed in the event. Finding these faces can be done in $O(\log n)$ if we have some combinatorial information, such as triangles that appear on the boundary of these faces. When such information is not available (which is only in the case for events where the sweep reaches a first corner of some triangle) we use a (dynamic) point-location query which takes $O(\log^2 n)$ time.

Notice that a boundary edge can appear only as a right witness or as a left witness of a face. This is obvious from the way we build A_x and from the way we hold the description of a face f in A_x. In order to detect events where two boundary edges are vertically visible we check if the right boundary wall is vertically visible with the left boundary wall.

Notice that the left witness of a face can intersect only the leftmost triangle in the ceiling triangle chain or the leftmost triangle in the floor triangle chain. This is true because before the left witness intersects another triangle in the upper (or lower) chain an event where the left witness is vertically visible with some intersection edge must be encountered. Also, the right witness of a face can intersect only the rightmost triangle in the ceiling triangle chain or the rightmost triangle in the floor triangle chain. In order to detect events where a boundary edge intersects a triangle we check if the right witness intersects the rightmost triangle of the lower chain, or the rightmost triangle of the upper chain. We also check if the left witness intersects the leftmost triangle in the upper chain, or the leftmost triangle in the lower chain. Recall that one boundary edge appears once as a left witness of one cell and once as a right witness of another. Therefore these checks are applied to it twice, once as a right witness and once as a left witness.

Notice that the left witness of a face can be vertically visible only with the leftmost intersection edge in the ceiling triangle chain or with the leftmost intersection edge in the floor triangle chain. Also the right witness of a face can be vertically visible only with the rightmost intersection edge in the ceiling triangle chain or with the rightmost intersection edge in the floor triangle chain. In order to detect events where a vertical visibility occurs between a boundary edge and an intersection edge we check if the right witness is vertically visibly with one of the rightmost intersection edges in one of the chains, or if the left witness is vertically visibly with one of the leftmost intersection edges in one of the chains.

Notice that because of the general position assumption before a face disappears it is reduced to a face with a total of three triangles in both of its chains, and no boundary edges as witnesses, or to a face with a total of two triangles in both of its chains. In order to detect events where three triangles intersect we check for an intersection of three triangles that appear in faces that have a total of three triangles in both of its chains and no boundary edges as witnesses.

All these checks can be performed in $O(\log n)$ time. These checks detect potential events. We add events that were detected to the queue, with the x-coordinate of the event. We say potential because when we come to handle an event of vertical visibility, we may find that this vertical visibility is obscured by another triangle in a manner that we were unable to predict when the event was inserted into the queue. Therefore, we distinguish between two types of events: actual events, which are either events of intersections, events that occur at corners of triangles or events that indeed correspond to a vertical visibility, and false events, which are potential vertical visibilities that are discovered to be obscured when handled.

Lemma 4.1 *The set of events that is computed during the sweep contains the set of visibilities between boundary edges, the set of visibilities between boundary edges and intersection edges, the set of intersections between boundary edges and triangles and the set of intersections between three triangles.*

Proof. First we claim that no vertical visibility between a two boundary edges is missed by our algorithm. The reason is that before a pair of boundary edges become vertically visible, they must become witnesses for some face f of A_x. This is similar to a basic argument in the Bentley-Ottmann algorithm for finding intersections of line segments [8, 43]. The same argument is applied to events of vertical visibility between a boundary edge and an intersection edge, where the boundary edge and the intersection edge must become adjacent in the same face f of A_x before becoming vertically visible and to events of intersection of a boundary edge with a triangle, where the boundary edge and the triangle must become adjacent in the same face before intersecting; and to events of intersection of three triangles where the three triangles must contribute to the ceiling chain and to the floor chain of some face f in A_x, before they intersect. ■

Lemma 4.2 *The false events can be efficiently distinguished from the actual events. Each false event can be charged to an actual event, and no actual event gets charged more than a constant number of times this way.*

Proof. Notice that events of intersections cannot be obscured and once they are detected they are actual events. Once we handle a vertical visibility event it is easy to determine whether it is actual or false by checking whether the features involved in this event (boundary edges, intersection edges or triangles) belong to the same face and are adjacent in that face. The event is actual if and only if the features are in relative position as they were when the event was detected. That is, the topology of the arrangement is such that the event occurs as forecasted when it was detected. We charge each false event q' to the actual event q that has "spawned" it. Every vertical visibility event is added to the queue only at actual events; a false event does not create new events. Since no event creates more than a constant number of additional events, no actual event will be charged more than a constant number of times for false events. ■

Lemma 4.3 *The space sweep algorithm described above runs in time $O(n \log^2 n + E \log n)$, where E is the number of events and n is the number of triangles.*

Proof. We have shown that every event can be handled with a constant number of operations on some triangle chains (the full list of the event and how to handle them is given in Appendix B), and on the priority queue. We have shown that finding the faces that change in the event takes $O(\log n)$, except for the n events of first corners, which take $O(\log^2 n)$ each. Thus every event takes $O(\log n)$ time, except for n events which take $O(\log^2 n)$ each, and the whole sweep runs in $O(n \log^2 n + E \log n)$ time. ■

Notice that a face of A_x is an intersection of some three-dimensional cell with the sweep plane. The intersection of the sweep plane P_x with a cell consists of at most one face. However, since we allow the cells to be complicated, a single three-dimensional cell might intersect different sweep planes with faces which are different in their combinatorial structure.

During the sweep we gather the cell's information, i.e. the triangles that contribute to its floor and ceiling, and the vertical walls that bound it. For example, when a triangle is added to the ceiling chain or the floor chain we also add its contribution to the description of the cell.

When we gathered all the information regarding a cell, we write it to the output. This happens at events where we gather the information about the feature of the cell that has the largest x-value. A feature with the largest x-value may be of two types. Either the face that represented the cell on A_x has disappeared, or we erected a wall during a flooding process. In the first case, the description of the cell is complete and the structure representing the face is deleted. In the latter case, where we flood a face, the wall erected during the flood is added to the description of the cell and then the description is complete. The face structure is not deleted, but it is set to represent a different new cell. The flood is added to the description of the new cell as well.

Notice that the wall erected during the flood is the intersection of the cell with the sweep plane P_x. This intersection is equivalent to the face corresponding to the the cell. Therefore creating the description of the wall is trivial. The description added to the cell is simply the x value at which the flood occurred. This is sufficient to describe the flood since the cell already contains a description of its upper and lower boundaries and its boundaries on the positive and negative y directions (on the left and right).

Generally, we flood a face before we split it, before we join it with another face, or when the sweep plane reaches the middle corner of its left (right) witness. More specifically, we flood a face in these cases:

1. A new edge appears inside the face.
2. An edge disappears.
3. The sweep has reached a corner of the triangle that is the left boundary of the face, and the triangle lies to the left of the corner.
4. The sweep has reached a corner of the triangle that is the right boundary of the face, and the triangle lies to the right of the corner.
5. A boundary edge e of some triangle t_i intersects a triangle t_j which contributes to the upper (lower) chain of the face.

When two faces become neighbors, that is, two faces are separated by a vertical wall (erected from a boundary edge, or erected during a flood), we keep the information of this neighborhood, either by adding the description of the wall and a pointer to the face that lies behind it, or by adding an edge in the graph that represents the output between the two nodes that represent the neighboring cells.

Lemma 4.4 *Given a set of triangles T, the algorithm described above constructs the partial decomposition $PD(T)$.*

Proof. Cells that are the output of the algorithm described above have boundaries that consists of triangles, walls erected from boundary edges and walls erected during a flood.

Recall that every face f in A_x is an intersection of some cell with the sweep plane P_x.

Edges that result from the intersection of triangles with the sweep plane P_x can appear only on the boundary of a face f in A_x. Therefore triangles can appear only on the boundary of cells.

The intersection of a boundary edge e with the sweep plane P_x results in an outer vertex in A_x. An outer vertex can appear only as a witness for the left boundary or the right boundary of a face. This is equivalent to erecting a wall from the boundary edge e, which means that walls erected from boundary edges can appear only on the boundary of cells.

The flooding process described in Section 3.3 takes place in a decomposition where vertical walls have been erected from boundary edges. Such a flood corresponds to flooding a face in A_x. The algorithm floods faces in events mentioned in Section 3.3. Each of these floods is a vertical wall that splits the cylinder represented by the face into two cells. Therefore walls erected in the flood process appear on the boundary of cells. ■

Theorem 4.5 *The partial decomposition can be computed in $O(n \log^2 n + k \log n)$ time, where k is the complexity of the output.*

Proof. In the algorithm we have shown, it takes $O(\log n)$ time to handle an event which is not a first corner of a triangle (without flooding and without adding new features to cells' description). First corners of triangles can be handled in $O(\log^2 n)$ time each. We denote the overall complexity of flood walls by ϕ which may be $\Theta(n^3)$ in the worst case: There are at most $O(n^2)$ flood events and handling each takes at most $O(n)$ time. As mentioned earlier the overall complexity of all the flood faces can be $\Theta(n^3)$ in the worst case. We also denote the number of events by E. The sweep takes $(n \log^2 n + E \log n + \phi + k)$ time to complete.

Since every event during the sweep induces at least one feature in the output representation, and since every feature of the flood contributes a constant number of features in the output representation, we conclude that the algorithm runs in $O(n \log^2 n + k \log n)$, where k is the complexity of the output. ■

4.3 The Full Decomposition

We construct the full decomposition by performing a similar space sweep. We sweep a plane P_x parallel to the yz-plane from $x = -\infty$ to $x = \infty$. The goal of the sweep is to erect the walls that refine the $A(T)$, as described in Section 3.2, and create the data structure that represents the decomposition, as described in Section 4.1.

Denote the arrangement induced by the intersection of the triangle set T with the sweep plane P_x, by A'_x. Also denote the decomposition induced by the intersection of the triangle set T and the set of walls W extended from each boundary edge, with the sweep plane, by A_x.

As in the description of the partial decomposition, here as well, the decomposition A_x changes continuously along the sweep, however its combinatorial structure changes only at a finite number of events.

1. A vertex may appear (disappear).
2. An edge may appear (disappear).
3. A face may appear (disappear).
4. An outer vertex is vertically visible from a vertex (either inner or outer).

   Since the full decomposition erects walls from intersection edges, we shall have another type of event, that was not needed in the partial decomposition algorithm:

5. Two inner vertices are vertically visible.

As in the algorithm for the partial decomposition described in Section 4.2 we maintain a data structure to represent A_x, and a dynamic data structure that answers point-location queries. The data structures are the same as was described for the partial decomposition.

Maintaining the dynamic point location data structure and our data structure during the sweep is done as it is done during the sweep for the partial decomposition. As in the partial decomposition we start the sweep by inserting into a queue events that occur on triangle corners. And we discover more events and insert them into the queue as described for the partial decomposition algorithm in Section 4.2.

Since the full decomposition erects walls from intersection edge we need to gather information on these walls during the weep. In order to do so we need to keep track of vertical visibilities between intersection edges. This is done as in [21]. Whenever a new vertex v is created on the floor chain of f (the operations for a new vertex on the ceiling chain are symmetric), we look for its "neighbors" in the ceiling chain of f (mark them as u_1 and u_2), that is, we look for the two vertices of the ceiling of f whose projection onto the y-axis lie nearest to the projection of v onto the y-axis, where $(u_1)_y < (v)_y < (u_2)_y$. Each vertex w in the ceiling chain and in the floor chain represents an intersection edge $e(w)$. Since we have attached to each vertex some additional information, we can compare $e(v)$ with each of $e(u_1)$ and $e(u_2)$ to see if their projections onto the xy-plane intersect. If they intersect, we have detected a potential visibility between two intersection edges; we add this event to the queue, with the x-coordinate of the intersection point. We say potential because when we come to handle this event, we may find that this "vertical visibility" is obscured by another triangle in a manner that we were unable to predict when the event was inserted into the queue. Therefore, we distinguish between two types of events: actual events, which are either events of intersections, events that occur at corners of triangles or events that indeed correspond to a vertical visibility, and false events, which are potential vertical visibilities that are discovered to be obscured when handled.

When we handle an actual event that correspond to vertical visibility of a pair of vertices u and v on the ceiling and floor chains, respectively, each of u and v now have new neighbors on the opposite chain, so we check these new neighbors for additional potential events as above.

The proofs of Lemmas 4.6 and 4.7 are straightforward extensions of the proofs of Lemmas 4.1 and 4.2, we therefore omit them here.

Lemma 4.6 *The set of events that is computed during the sweep contains the set of visibilities between boundary edges, the set of visibilities between boundary edges and intersection edges, the set of visibilities between two intersection edges, the set of intersections between boundary edges and triangles and the set of intersections between three triangles.*

Lemma 4.7 *The false events can be efficiently distinguished from the actual events. Each false event can be charged to an actual event, and no actual event gets charged more than a constant number of times this way.*

Lemma 4.8 *The space sweep algorithm described above runs in time $O(n \log^2 n + E \log n)$, where E is the number of events and n is the number of triangles.*

Proof. We have shown that every event can be handled with a constant number of operations on some triangle chains (the full list of the event and how to handle them appears in Appendix B), and on the priority queue. We have shown that finding the faces that change in the event takes $O(\log n)$, except for the n events of first corners, which take $O(\log^2 n)$. Thus every event takes $O(\log n)$ time, except for n events which take $O(\log^2 n)$, and the whole sweep runs in $O(n \log^2 n + E \log n)$ time. ■

Recall that a face of A_x is an intersection of some three-dimensional cell with the sweep plane. The intersection of the sweep plane A_x with a cell consists of at most one face.

When we handle events during the sweep we add cells (and their description) to the output data structure. We gather the description of the cell from the description we maintain for the face. In order to do so we need to refine the faces.

Consider a face f. Erect a vertical segment from each vertex in the floor chain and from each vertex in the ceiling chain. We obtain a subdivision of the face into trapezoids. We call each of these trapezoids a *sub-face*. Consider Figure 4.2 for an illustration of a decomposition of a face into sub-faces.

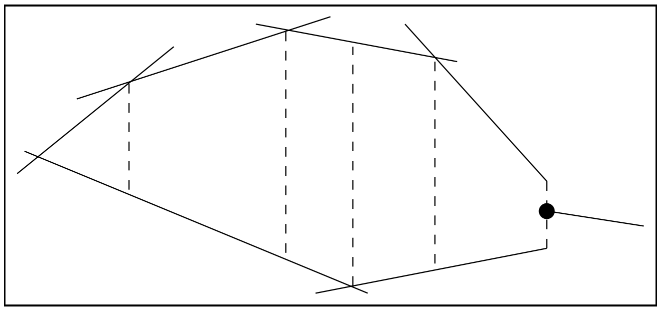

Figure 4.2: A decomposition of a face into sub-faces

When we handle an event during the sweep, we create a cell for each sub-face that has ended, or changed. We also set the cell description to be the information gathered in the sub-face. Figure 4.3 illustrates a case where three sub-faces are replaced with three different sub-faces in an event of vertical visibility of two intersection edges. In this example sub-faces A, B and C, are replaced by A', B' and C'. Notice that the sub-face D remains unchanged.

When two sub-faces become neighbors, that is, two sub-faces are separated by a vertical wall parallel to the xz-plane (erected from a boundary

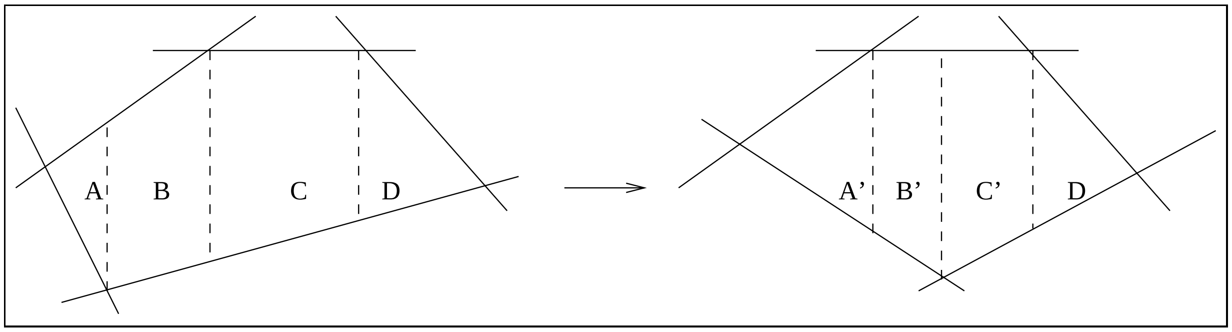

Figure 4.3: Two sub-faces are replaced by two other sub-faces in an event of vertical visibility

edge or from an intersection edge) or two sub-faces are separated by a wall parallel to the yz-plane (as happens when one sub-face replaces another sub-face), we keep this information in the face, we also add the information about the two neighboring cells represented by the faces to the output.

Lemma 4.9 *For a given set T of triangles the algorithm described above produces the full decomposition $FD(T)$.*

Proof. Cells that are the output of the algorithm described above have boundaries that consists of triangles, walls erected from boundary edges, walls parallel to the yz-pane erected from intersection edges and walls parallel to the xz-plane erected from points in which events took place.

Recall that every face f in A_x is an intersection of some cell with the sweep plane P_x.

Edges that result from the intersection of triangles with the sweep plane P_x can appear only on the boundary of a face f in A_x. Therefore triangles can appear only on the boundary of cells.

The intersection of a boundary edge e with the sweep plane P_x results in an outer vertex in A_x. An outer vertex can appear only as a witness for a left boundary of a face or a right boundary of a face. This is equivalent to erecting a wall from the boundary edge e, which means that walls erected from boundary edges can appear only on the boundary of cells.

Notice that the walls parallel to the yz-plane are erected only inside sub-faces that contain the point in which the event took place. This separation of sub-faces is equivalent to erecting these walls after walls parallel to the xz-plane have been erected from boundary edges and intersection edges. These walls are equivalent to the yz walls described in Section 3.2. ■

Theorem 4.10 *The full decomposition can be computed in $O(n \log^2 n + k \log n)$ time, where k is the complexity of the output.*

Proof. In the algorithm we have shown, it takes $O(\log n)$ time to handle an event which is not a first corner of a triangle. First corners of triangles can be handled in $O(\log^2 n)$ time each. The whole sweep takes $(n \log^2 n + E \log n)$ time to complete, where E is the number of events.

Clearly the algorithm we have shown runs in $O(n \log^2 n + E \log n + k)$.

Since every event during the sweep induces at least one feature in the output representation, we conclude that the algorithm runs in $O(n \log^2 n + k \log n)$, where k is the complexity of the output. ■

Chapter 5

Implementation Details

We implemented the algorithms for constructing partial decomposition and the full decomposition. The two implementations were written in C++. Since there is much similarity between the two algorithms some of the code in their implementations is the same, and most of the rest has some small modifications. Most of the code describes how events are handled during the sweep, and since except for the event of intersection edges which are vertically visible, all other events appear on both algorithms they have almost the same number of source lines - approximately 16000. 5000 of which describe how events are handled during the sweep. 2000 lines describe how additional events for the case of polyhedral surfaces are handled. 2000 lines describe how to detect the events.

We use the kernel layer of the CGAL library [10], which provides us with basic classes such as point, line, segment and triangle. The type of numbers we use was left as a parameter that can be decided in compilation time. We use the LEDA library [38] which provides rational numbers, a class that represents an exact value of a rational number and allows operations on it.

5.1 Exact Computation

LEDA's rational numbers class represents a rational by maintaining a numerator and a denominator, which are multi-precision integers. Operations on LEDA's rational might double the bit-length of the numerator and the denominator. Having a big numerator or denominator induces more processing time for the computation of operations. LEDA, therefore, introduced a "normalize" function which divides the numerator and denominator by the largest common denominator and by that reduces the bit-length of the numbers. In practice, however, division by the largest common denominator does not reduce the bit-length significantly. In theory, the worst case might be that every operation doubles the bit-length of the number. We can assume an adversary that perturbs the input to choose numbers so results of operations we perform cannot be "normalize".

The time it takes to calculate an operation on two LEDA rationals is proportional to the bit-length of the numerator and the denominator. This introduces a new consideration in analyzing the running time of an algorithm. More "complex" numbers take more time to calculate.

We say that a number is of depth 1 if it is a number that comes from the input of the algorithm. We say that a number is of depth-$(n+m)$ if it is a result of an operation on two numbers of depths n and m. Notice that the bit length of the representation of a number of depth i is $\Theta(2^i)$. The depth of an algorithm is the maximum depth of the numbers it uses. An algorithm with greater depth will generate numbers with higher bit-lengths, and thus will take more time compute.

5.1.1 The Depth of the Algorithms

The implementation of the partial decomposition uses 11 number types:

1. The coordinates of corners of triangles.

2. The coordinates of endpoints of intersection edges.

3. The coordinates of the event point where two boundary edges are vertically visible.

4. The coordinates of the event point where an intersection edge and a boundary edge are vertically visible.

5. The coordinates of the event point where three triangles intersect.

6. The coordinates of a feature of a vertical wall $W(p, T')$, where the feature comes from a boundary edge, and the point p is a corner of a triangle.

7. The coordinates of a feature of a vertical wall $W(p, T')$, where the feature comes from a boundary edge, and the point p is the event point where two boundary edges are vertically visible.

8. The coordinates of a feature of a vertical wall $W(p, T')$, where the feature comes from a boundary edge, and the point p is the event point where a boundary edge and an intersection edge are vertically visible.

9. The coordinates of a feature of a vertical wall $W(p, T')$, where the feature comes from a boundary edge, and the point p is the event point where three triangles intersect.

10. The coordinates of a feature of a vertical wall $W(p, T')$, where the feature comes from an intersection edge, and the point p is a corner of a triangle.

11. The coordinates of a feature of a vertical wall $W(p, T')$, where the feature comes from an intersection edge, and the point p is the event point where two boundary edges are vertically visible.

12. The coordinates of a feature of a vertical wall $W(p, T')$, where the feature comes from an intersection edge, and the point p is the event point where a boundary edge and an intersection edge are vertically visible.

13. The coordinates of a feature of a vertical wall $W(p, T')$, where the feature comes from an intersection edge, and the point p is the event point where three triangles intersect.

    The implementation of the full decomposition uses 3 additional number types:

14. The coordinates of the event point where two intersection edges are vertically visible.

15. The coordinates of a feature of a vertical wall $W(p, T')$, where the feature comes from a boundary edge, and the point p is the event point where two intersection edges are vertically visible.

16. The coordinates of a feature of a vertical wall $W(p, T')$, where the feature comes from an intersection edge, and the point p is the event point where two intersection edges are vertically visible.

The depth of calculation of points that are a result of events where two intersection edges are vertically visible is very large. Hence the full decomposition is more prone to robustness problems with imprecise arithmetic or to more time-consuming calculations when using exact arithmetic. Furthermore, programs that use the full decomposition suffer from the same problem. We leave the exact determination of the depth of the algorithm for future work.

5.2 General Position and Perturbation

We assume the triangles are in general position as detailed in Section 2.2.1.

A scheme was proposed in [47, 46] for the perturbation of polyhedral surfaces. At first, vertices of each polyhedral surface are perturbed so that degeneracies are removed inside any single surface. Then polyhedral surfaces are perturbed so that degeneracies are removed globally. Eventually, a coordinate system is chosen so degeneracies of vertical visibility are removed.

This scheme runs in $O(n \log^3 n + nDK^2)$ expected time and requires $O(n \log n + nK^2)$ working storage, where n is the number of triangles, D and K are parameters that might be as large as n in the worst case, but are often small constants in practice. See [47, 46] for more details.

We used this scheme to ensure that the scenes in our tests were in general position.

5.3 Output Representation

As mentioned earlier, we represent the decomposition as a graph. Each node v represents a three-dimensional cell C_v in the arrangement. Two nodes v, u are connected if C_v and C_u share a vertical wall.

Define 'up' as the positive z direction, and 'right' as the positive y direction.

A node v contains a full description of the cell C_v which includes:

1. A list of triangles that contribute to the cell's ceiling.

2. A list of triangles that contribute to the cell's floor.

3. A list of walls that bound the cell from the right.

4. A list of walls that bound the cell from the left.
5. Two walls parallel to the yz-plane that bound the cell between them.
6. A list of pointers to the neighbor cells of C_v.

5.4 Program Checking

In order to verify that our programs run correctly we used two checking schemes. The first scheme checks that the cells' volumes sum up correctly. The second scheme checks that the neighborhood connections are correct.

5.4.1 Cell Check

The convexity of each cell is guaranteed because we represent the cell by the intersection of half-spaces. In order to check that the cells have the correct volumes we calculate each cell's volume, sum the volumes for all the cells inside a bounding box and compare it to the volume of the bounding box. We use exact arithmetic to make this procedure meaningful.

For convenience of calculation we define an artificial bounding floor below the cell. Then we calculate the volume between the ceiling of the cell and the artificial floor. We also calculate the volume between the floor of the cell and the artificial floor. The difference between these two volumes is the cell's volume. For each cell we calculate the two volumes by partitioning them into triangular prisms.

5.4.2 Neighbor Check

In order to check that the neighborhood connections we calculated are correct we check for each pair of cells if they are neighbors, and compare the result we calculated with the output of the decomposition.

Define the *crossing boundary* of a cell as all the points that lie on the boundary of the cell and do not lie on any triangle in T. Also, define the intersection between the crossing boundary of two cells as the *border* between the two cells.

In order to calculate if two cells are neighbors we look for a point that lies on the border between the two cells.

Chapter 6

Results

6.1 Complexity of Decompositions

We tested both the partial and the full decomposition on eight input scenes, which are depicted in the following figures. All out tests were done on a computer with a Celeron 400Mhz CPU and 512 MByte RAM. Our experiments show that the partial decomposition proves itself more efficient in solving two problems we tested.

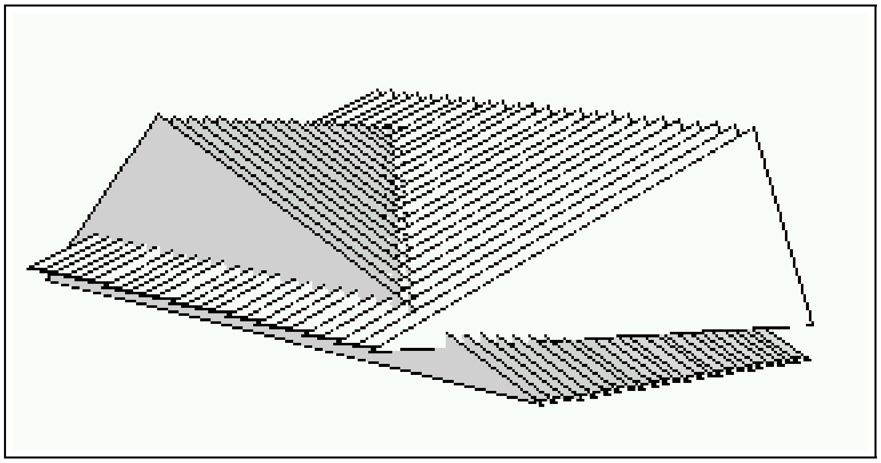

Figure 6.1: Test case 1

Test case 1 consists of two sets of $n/2$ triangles each (Figure 6.1). The triangles are laid in a grid so each triangle in one set intersects the $n/2$ triangles of the other set, Thus the scene has $\Theta(n^2)$ intersections. The partial

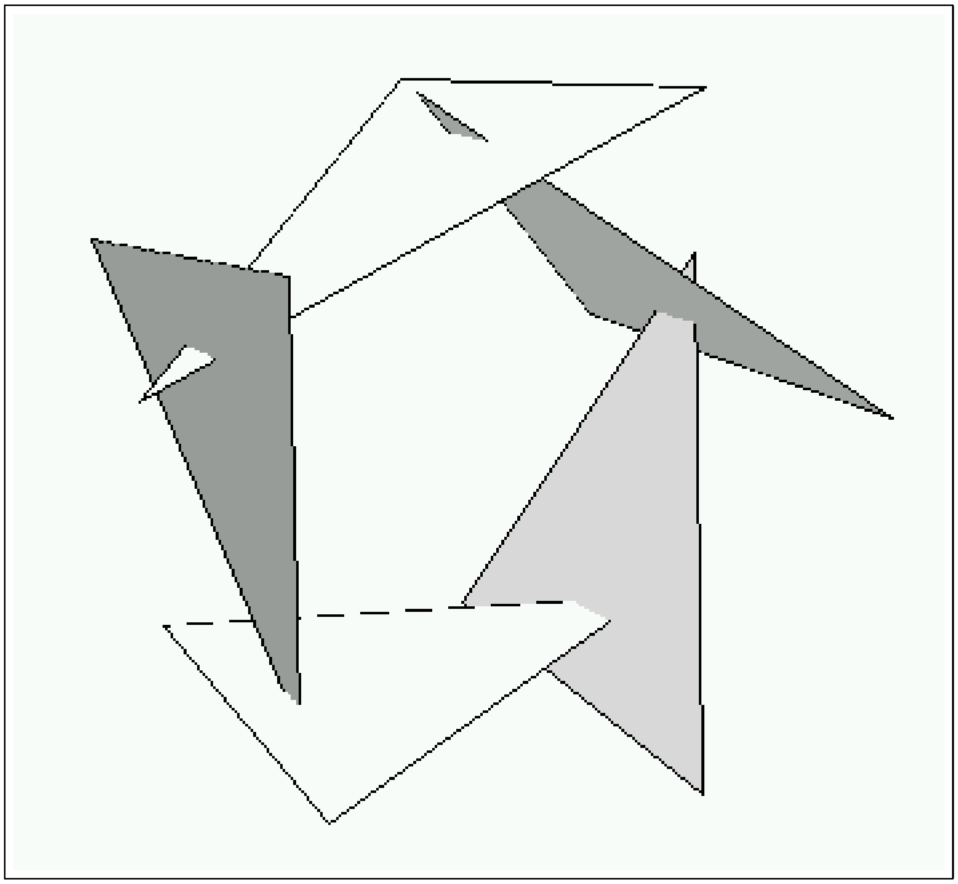

Figure 6.2: Test case 2

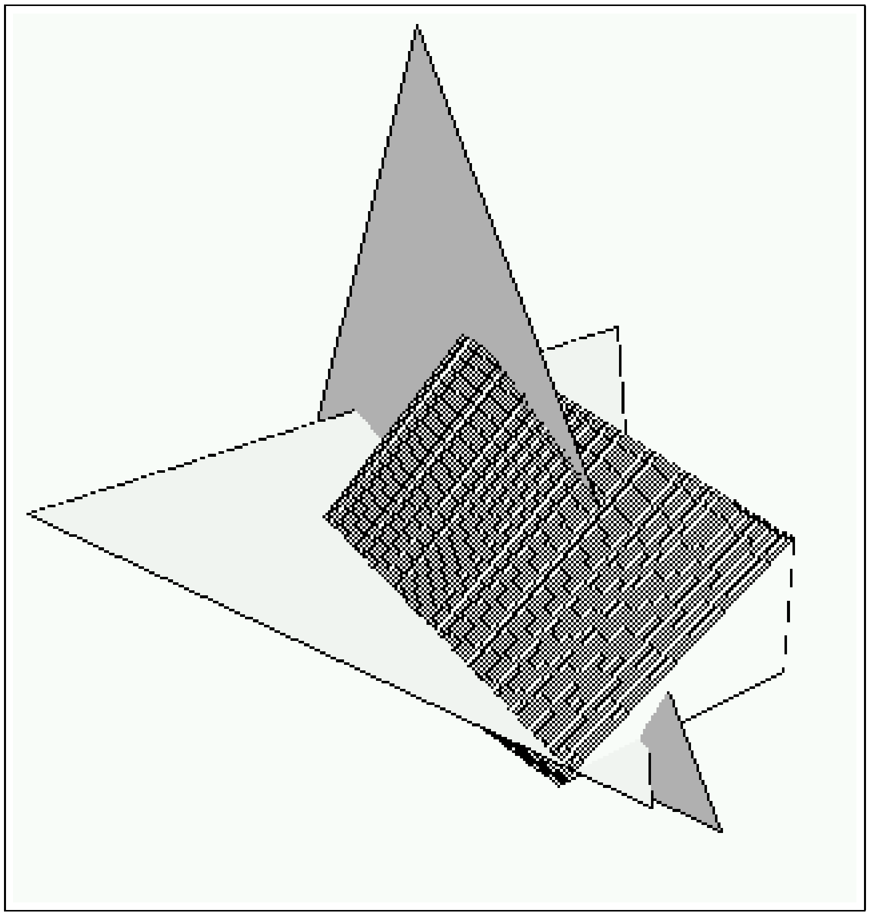

Figure 6.3: Test case 3

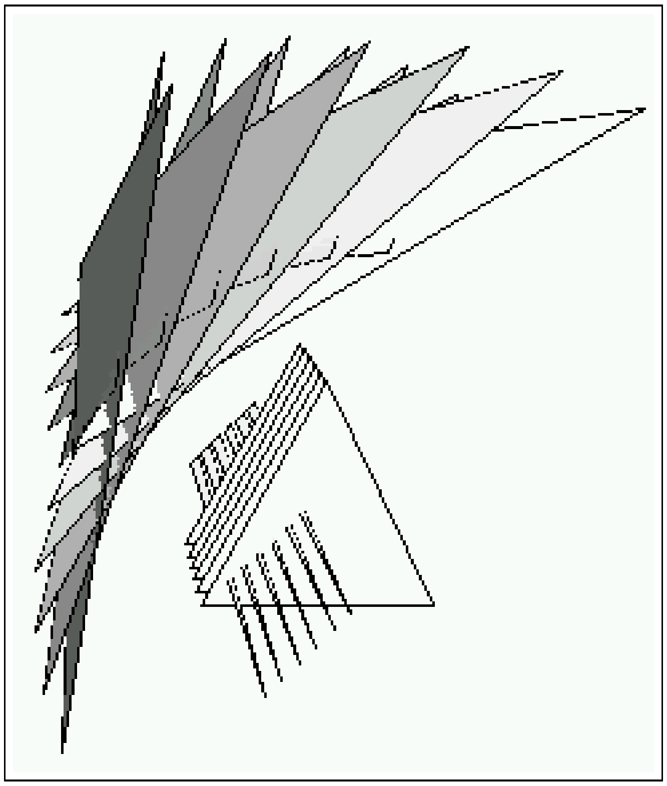

Figure 6.4: Test case 4

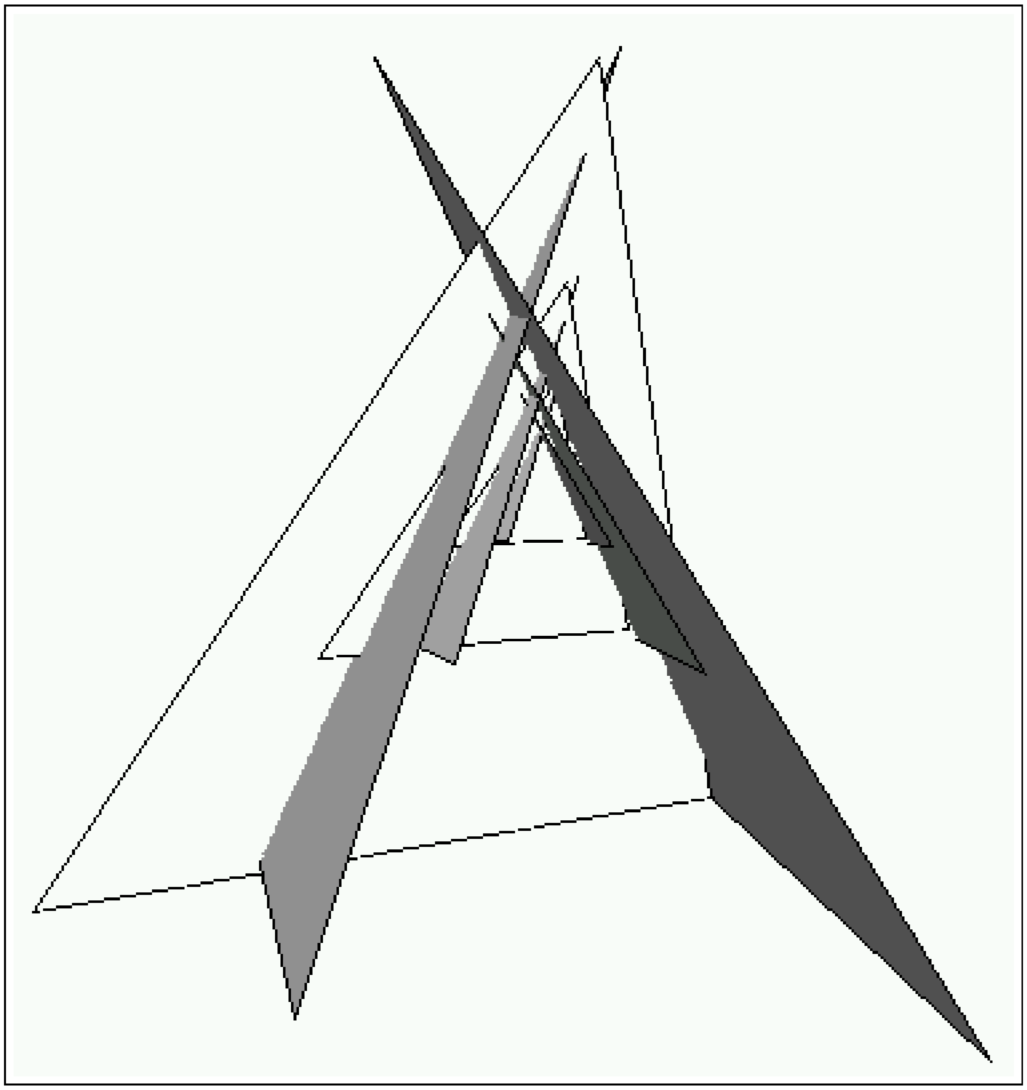
Figure 6.5: Test case 5

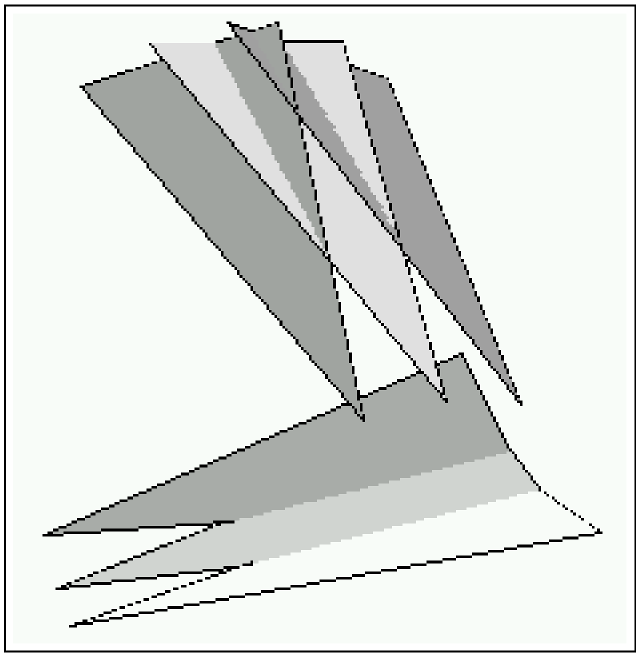
Figure 6.6: Test case 6

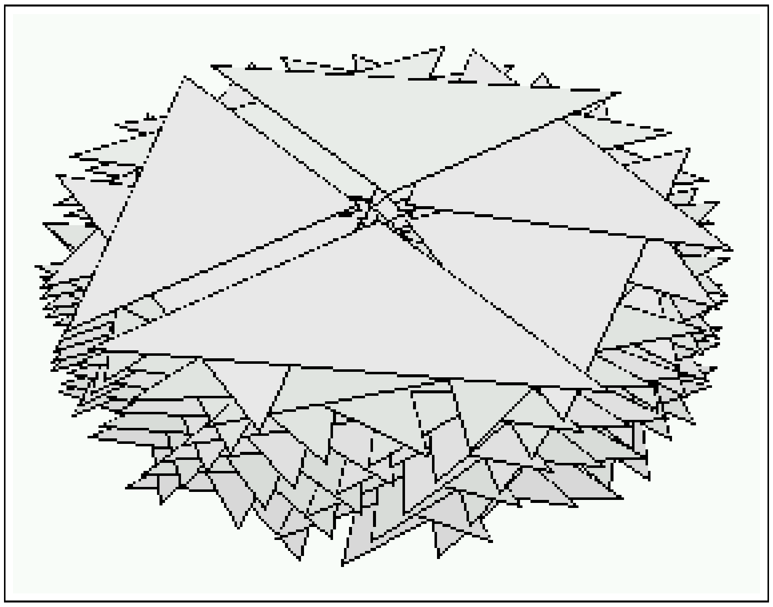
Figure 6.7: Test case 7

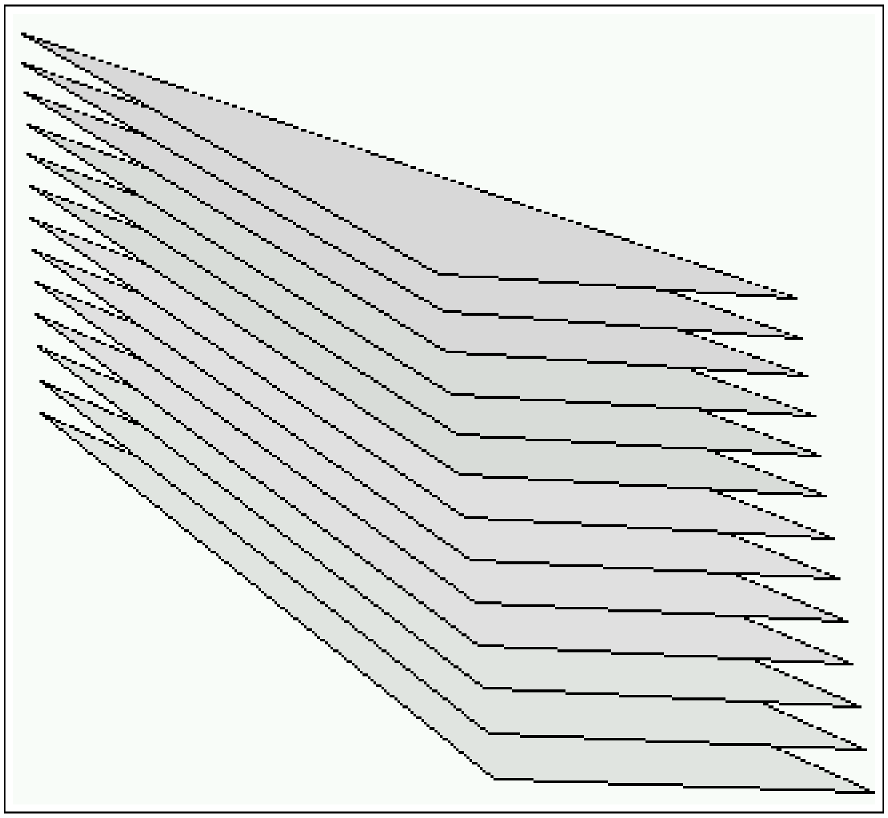
Figure 6.8: Test case 8

decomposition should perform better in a scene which has a lot of intersections. However, since the triangles are laid in a grid the cells of the partial decomposition are much bigger than the cells of the full decomposition.

Test case 2 consists of a “ring” of triangles (Figure 6.2). The partial decomposition creates a few large cells in the middle of the ring. The partial decomposition is expected to perform better than the full decomposition because of these large cells.

Test case 3 (Figure 6.3) consists of two big triangles, whose intersection is then intersected by $n-2$ triangles. The intersections of boundary edges with the two big triangles creates floods in the partial decomposition which do not exist in the full decomposition. These floods cause one cell of the full decomposition to be divided among many cells in the partial decomposition, although the number of cells will be smaller in the partial decomposition.

Test case 4 demonstrates the worst case complexity of the partial decomposition (Figure 6.4). This case was described before in Section 3.3. The full decomposition has a better complexity, although it has more cells. Applications using a vertical decomposition performed better with the partial decomposition.

Test case 5 (Figure 6.5) consists of pyramids built recursively inside pyramids.

Test case 6 (Figure 6.6) consists of a scene where an intersection of three triangles is vertically visible with another intersection of other three triangles.

Test case 7 (Figure 6.7) consists of a scene where quadruples of triangles are laid one above the other. None of the triangles are intersecting, but boundary edge are vertically visible. This scene shows that the partial decomposition is more economical than the full decomposition even in a scene with no intersections. The savings in this case would come from the way vertical visibility of boundary edges are handled. Recall that the full decomposition erects a wall $W(p, T')$ from every event point p, in addition to walls erected from every boundary edge, every intersection edge and every triangle corner. The partial decomposition erects walls only from boundary edges, triangle corners and intersection points. This means that in an event where two boundary edges are vertically visible the full decomposition erects a wall which the partial decomposition does not, thus splitting a cell. Consider, for example, event 31 shown in Figure B.2. Here faces 2 and 12 are contained in different cells in the full decomposition, but in the same cell in the partial decomposition.

Test case 8 (Figure 6.8) consists of a “stack” of triangles laid one above the other. None of the triangles are intersecting, and each boundary edge is vertically visible with only two other boundary edges, one from below and one from above. This scene shows that the partial decomposition is more economical than the full decomposition even in a scene with no intersections and little vertical visibility between edges. The savings in this case would come from the way middle corners of triangles are handled. In the full decomposition we erect a wall $W(p, T')$ from every corner of each triangle, whereas in the partial decomposition we erect a wall $W(p, T')$ from the first corner and the last corner of every triangle and half a wall $W_l(p, T')$ or $W_r(p, T')$ from the middle point of every triangle. The full decomposition adds half a wall at such event that the partial decomposition does not.

For each scene we computed the complexity of each decomposition. In the full decomposition we simply counted the number of cells. Since each cell in the full vertical decomposition has constant complexity it is sufficient to count the number of cells to know the complexity of the arrangement up to a small constant factor. In the partial vertical decomposition we counted the complexity of all the floors and ceilings and all the vertical walls. We counted the total complexity of all floors and ceilings by summing up the number of triangles in the floor and ceiling of all the cells. We write this as $\sum_i ic_i$ where c_i is the number of cells that have a total of i triangles in their floor and ceiling.

Similarly we counted the total complexity of all vertical walls by counting the number of vertical walls in each cell, and then summing up these numbers. We write this as $\sum_i iw_i$ where w_i is the number of cells that have a total of i vertical walls. Note that this notation is simply a compact way to write how many cells have a certain number of walls or ceilings.

The results in Tables 6.1 and 6.2 indicate that the partial decomposition is more economical than the full decomposition. One may suspect that because we invest less time in preprocessing, using the decomposition to solve problems will be more costly. We next show by experiments on two different problems that this is not the case. The opposite is true: the partial decomposition proves itself more efficient in solving these problems.

6.2 Calculating Volumes

As mentioned in Section 5.4, we have computed the volume of the arrangement in order to see that the decomposition is correct. The time it took to

Scene No.	Time to complete full decomp. (In sec.)	Time to complete partial decomp. (In sec.)	# of cells in full decomp.	# of cells in partial decomp.	Complx. of walls	Complx. of floors and ceilings
1	452	346	2630	1210	1*174 + 2*615 + 3*348 + 4*71 + 5*2	2*898 + 3*254 + 4*58
2	42809	42426	230	131	0*4 + 1*19 + 2*63 + 3*34 + 4*9 + 5*2	2*94 + 3*30 + 4*5 + 5*2
3	7595	4985	6050	2803	0*196 + 1*472 + 2*1550 + 3*530 + 4*55	2*1471 + 3*1064 + 4*268
4	149069	127070	2356	1170	0*27 + 1*247 + 2*498 + 3*332 + 4*63 + 5*2 + 7*1	2*753 + 3*347 + 4*50 + 5*4 + 6*4 + 7*12

Table 6.1: Complexity of vertical decompositions, scenes 1 – 4

Scene No.	Time to complete full decomp. (In sec.)	Time to complete partial decomp. (In sec.)	# of cells in full decomp.	# of cells in partial decomp.	Complx. of walls	Complx. of floors and ceilings
5	208259	144687	768	294	0*27 + 1*90 + 2*145 + 3*28 + 4*4	2*126 + 3*106 + 4*47 + 5*14 + 6*1
6	4350	3743	110	59	1*2 + 2*30 + 3*17 + 4*9 + 5*1	2*53 + 3*5 + 4*1
7	2068	942	7922	2971	2*322 + 3*881 + 4*1383 + 5*237 + 6*148	2*2971
8	1967	1963	2096	1199	2*601 + 3*300 + 4*298	2*1199

Table 6.2: Complexity of vertical decompositions, scenes 5 – 8

Scene No.	Full Decomposition	Partial Decomposition
1	7263	2720
2	149242	144784
3	346307	150133
4	740395	754799
5	308520	421512
6	130904	182949
7	347463	360315
8	603	568

Table 6.3: Time in seconds to compute the volume of all the cells in the decompositions

calculate these volumes is given in Table 6.3.

In order to calculate the volume of a cell we defined some artificial floor. We calculated the volume between the floor of the cell and the artificial floor, and the volume of the ceiling of the cell and the artificial floor. The volume of the cell is the difference between the two volumes. This way gives advantage to the partial decomposition since we do not overlay the map of the ceiling with the map of the floor. This overlay produces more complex numbers (as explained in Section 5.1.1). On the other hand the partial decomposition, as opposed to the full decomposition, needs to partition the ceiling (floor) to cylinders that have constant description. This partitioning is very expensive since we need to find the corners and edges that appear in the ceiling (floor). It is possible that adding more information to the output representation, such as the event points at which the cell's boundary was created, would improve the partial decomposition.

Notice that although all of the scenes had less cells in the partial decomposition, most of them had a considerable amount of complex cells (cells with a large number of triangles in the ceiling, floor or in the walls) and indeed in most of the scenes computing the volumes took roughly the same time using both decompositions (with the full decomposition being even slightly superior). The only exceptions are scenes 1 and 3 where the partial decomposition was significantly faster. Indeed the cells in the partial decomposition of scenes 1 and 3 are not very complex, in the sense that they do not have many triangles in their ceiling, floor and walls.

6.3 Finding a Free Path

We also performed a test in which we chose two points in $\mathbb{R}^3$ and calculated a path between those two points that does not intersect any triangle.

The points were chosen randomly from some bounding box that contain all the triangles in the scene. We searched for the starting node and the target node by checking for each cell if the points are contained in it.

Most of the computation time is spent on finding the nodes in the graph that contain the two points. We find these cells by checking every cell if it contains one of the points. Recall that the description of a cell actually consists of half spaces, when the cell is the intersection of them. Checking if a cell contains a a point is done by checking if the point is contained in each of these half spaces. Since the partial decomposition produces less cells, it is often the case that it takes less time to locate the relevant cells, although they are more complex.

In some cases, the full decomposition was faster than the partial decomposition. This is because the search terminates when the two cells are found. If the cells we search for are in the beginning of the full decomposition cell list, it would be found sooner.

The case of the third scene is very interesting. With the full decomposition it took, on the average, 5.68 milliseconds to compute a path between two randomly chosen points; we chose ten random pairs, and the average number of cells crossed by a path was 14.2. With the partial decomposition it took, on the average, 0.79 milliseconds to compute a path between two randomly chosen points; we chose ten random pairs, and the average number of cells crossed by a path was 17.9. The reason there are more cells in the path the partial decomposition found is the flooding done during the partial decomposition. In the full decomposition we flood only the sub-faces that have changed during the event, whereas in the partial decomposition we flood the whole face. Although there are more cells in the path of the partial decomposition on average, it takes less time to compute since it takes less time to locate the source cell and the destination cell.

In scenes 2, 6 and 8, computing a path between two random points in the partial decomposition was slower (on average) than computing a path between the same points in the full decomposition. In these scenes the number of cells the full decomposition produced was less than twice the number of cells the partial decomposition produced. The time "gained" by going over fewer cells was less than the time "lost" on going over more complex cells.

	Full Vertical Decomposition		Partial Vertical Decomposition	
Test No.	Average # of cells in path	Average time to calculate the path	Average # of cells in path	Average time to calculate the path
1	12.3	0.86	9	0.29
2	4.4	0.012	3.7	0.012
3	14.2	5.68	17.9	0.79
4	6.3	1.00	4.8	0.392
5	4.2	0.286	3	0.0065
6	4.2	0.0125	3.4	0.0313
7	7	12.39	4.9	3.471
8	33.4	0.083	33.4	0.1021

Table 6.4: Average time in seconds to compute ten paths between ten pairs of point

Chapter 7

Arrangement of Polyhedral Surfaces

We also extended the algorithms to produce the partial vertical decomposition and the full vertical decomposition of arrangements of polyhedral surfaces. The extension works for (possibly self intersecting) polyhedral surfaces that do not have holes. See Figure 7.1 for an illustration of a polyhedral surface of the type that we consider.

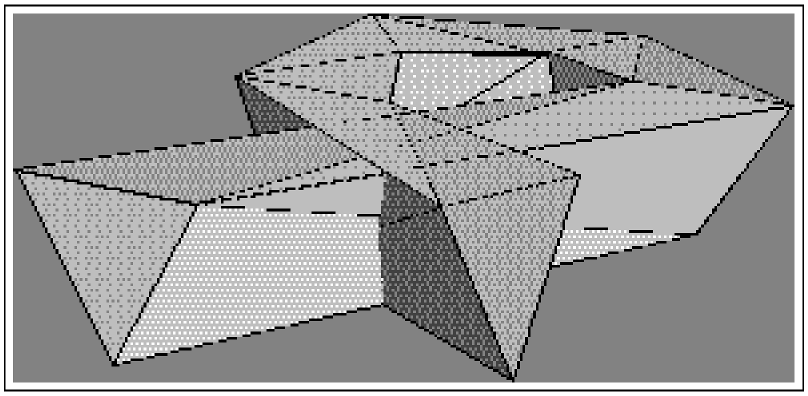

Figure 7.1: An arrangement of polyhedral surfaces

7.1 Additional Degeneracies

In addition to the general position assumption for triangles we also rule out the following degeneracies when dealing with polyhedral surfaces:

1. No more than two triangles can share a boundary edge.
2. Two triangles can share a corner, unless their interiors intersect.

7.2 Events During the Sweep

The space sweep, which gives the partial decomposition and the first refinement of the full decomposition, is not changed. The only modification is that there are more event types.

Five kinds of events are added to those already mentioned in Section 4.2.

1. A polyhedral line may appear (disappear).
2. A polyhedral line may change its combinatorial structure.
3. The sweep plane reaches an intersection between the common boundary edge of two triangles and another triangle.
4. The sweep plane reaches the vertical visibility between two boundary edges where one of the edges is common to two triangles.
5. The sweep plane reaches the vertical visibility between the common boundary edge of two triangles and an intersection of two other triangles.

The first two kinds of events occur when the sweep plane reaches a corner that is shared by two or more triangles. In the case where the corner is the first corner of all triangles a polygonal line will appear on the sweep plane. In the case where the corner is the last corner of all triangles a polygonal line will disappear from the sweep plane. Otherwise (the corner is a first corner for some triangles, last corner for some triangles and middle corner for some triangles), a polygonal line on the sweep plane will change its combinatorial structure.

The code we wrote for arrangements of triangles that handles events where two boundary edges are vertically visible, and the code for events where a boundary edge and an intersection between two triangles are vertically visibly is general enough to support the case where one or two of the boundary edges are common to two triangles.

In other events the support for boundary edges common to two triangles is not as trivial and causes a great deal of complication to the code. To simplify our code we distinguished between 16 events.

We also distinguish between two types of boundary edges common to two triangles. We call a boundary edge common to two triangles which are vertically visible a *non xy-monotone boundary edge*. We call a boundary edge common to two triangles which are not vertically visible an *xy-monotone boundary edge*.

Appendix C lists the events related to polyhedral surfaces and how to detect them. Appendix D describes how we handle events of decomposition of polyhedral surfaces.

Handling each of these new events takes $O(\log n)$ time, except for events where the sweep plane reaches a corner that is common to more than one triangle.

When the sweep plane reaches a corner that is common to more than one triangle polylines can appear, disappear and change combinatorial structure. Handling these events is done by computing the vertical decomposition of the arrangement of segments that intersect the sweep plane after the event occurs. In this decomposition we extend a line from each outer vertex.

Each face in this arrangement then becomes a face on A_x. Determining which faces are neighbors is easy given the decomposition we computed.

Handling such an event can take $\Omega(n \log n)$ time in the worst case. Since there are at most $O(n)$ such events, the sweep can take $O(E \log n + n * n \log n)$ in the worst case, where E is the number of events on the sweep. However, each feature in the two dimensional decomposition we compute in these events, becomes a feature of the three-dimensional decomposition. We deduce that handling the events during the sweep takes $O(k \log n)$ time, where k is the complexity of the output.

Theorem 7.1 *The partial decomposition of polyhedral surfaces takes $O(n \log^2 n + k \log n)$ time to compute, where k is the complexity of the output.*

Proof. We have shown that it takes $O(n \log^2 n + k \log n)$ to handle the events of the sweep. We have also shown in Section 4.2 that the total time spent on flooding is $O(k)$. ■

Theorem 7.2 *The full decomposition of polyhedral surfaces takes $O(n \log^2 n + k \log n)$ time to compute, where k is the complexity of the output.*

Proof. We have shown that it takes $O(k \log n)$ to handle the events of the sweep. It follows that the full decomposition can be computed in $O(n \log^2 n + k \log n)$ time. ■

Chapter 8

Conclusions

We have proposed a new vertical decomposition scheme for arrangement of triangles. The decomposition is more economical in the sense that it results in fewer cells than the standard vertical decomposition. We have also shown better ways to compute the vertical decompositions with only near-linear overhead complexity improving significantly over the best previously known algorithms. This way is also faster to run and easier to program since it does not need the first and the third steps suggested in [21]. Dropping these steps simplifies the implementation and hence increases the robustness of the program.

We have also shown that although the partial vertical decomposition results in more complex cells, two applications that use vertical decompositions do not suffer and usually benefit from replacing the standard vertical decomposition with the partial vertical decomposition.

Then, we extended the decompositions to polyhedral surfaces. This result is important because it allows to compute decompositions of polygons and polyhedra other than triangles, by triangulating the objects and computing the decomposition of the surfaces. Furthermore, it allows to compute an approximation of decompositions of general objects by approximating the objects with polyhedra and computing the decomposition of these polyhedra.

We propose several directions for future research:

1. Extend the algorithms we proposed to deal with algebraic, possibly not xy-monotone, surfaces and bodies. At least one new event type should be introduced in this case, the event where the interiors of two surfaces intersect. Event types that are used for arrangements of triangles are

relatively easy to convert since when we handle them we make assumptions only on their topological positions, and when we detect them we use a small set of functions.

2. Extend the algorithms we proposed to be more robust, so that they will not rely on a perturbation of the input [47, 46].

   We can, for example, regard a degeneracy as more than one event happening at the same point. The algorithm can then handle these events in some arbitrary order. The output will contain cells with zero volume.

   It would also be interesting to try to perturb the input as we sweep. When we reach x with the sweep plane, corners with lower x values are fixed, whereas corners with higher x values are loose and can be perturbed. In this case we should keep for each loose corner a sphere in which the corner can be perturbed without changing the topology of the decomposition. Each event in the sweep implies a restriction on spheres of corners that belong to triangles that participate in the event.

3. Search for a better output representation so the programs using the partial decomposition will be more efficient. In this work our orientation was improving motion planning for a robot. Other programs (such as the volume calculations) might not gain the most from using the partial decomposition this way. It is very likely that if we add more information to the output representation it will be more efficient for some programs. For example, adding a map to each cell that describes its ceiling.

4. Reduce the overhead of the algorithms. We use a dynamic point location data structure to locate the face in which the event of a first corner occur. However a weaker data structure might be just as good. For example a data structure that dynamicly locates the lowest segment above a given point.

Appendix A

Detecting Events

Event 0 is the event where a middle corner of a triangle is reached. In this event the combinatorial structure of the arrangement is not change. In event 10 a new triangle appears and in even 20 a triangle disappears. See Figure A.1. These events occur at corners of triangles and are trivial to detect. See Figure A.1.

Events 31, 32, 41, 42, 51, 52, 61 and 62 are all events where two boundary edges are vertically visible. This type of events occur on features of the vertical walls erected from the two boundary edges. All events occur on an outer vertex of the lower (upper) envelope. See Figures A.2 and A.3 for an illustration. The distinction between these events is based on the relative position of the two boundary edges and the triangles they bound. Consider for example event 31 (Figure A.2). Denote the intersection of the upper triangle with the sweep plane as t_i. Denote the intersection of the lower triangle with the sweep plane as t_j. We name the segments in the this figure after the triangles that induced them. Denote the intersection of the boundary edge l_i of the upper triangle with the sweep plane as e_i. Denote the intersection of the boundary edge l_j of the upper triangle with the sweep plane as e_j. t_i is to the right of e_i. t_j is to the right of e_j. e_j is to the right of e_i before the event, and e_j is to the left of e_i after the event. Hence the event is event 31.

Events 71, 72, 81, 82, 91, 92, 101 and 102 are all events where a boundary edge intersects a triangle. See Figures A.4 and A.5. This type of events also occur on features of the vertical wall in the three-dimensional decomposition erected from the boundary edge. These events occur in points on the boundary edge, where the segment erected from that point has a zero length, that

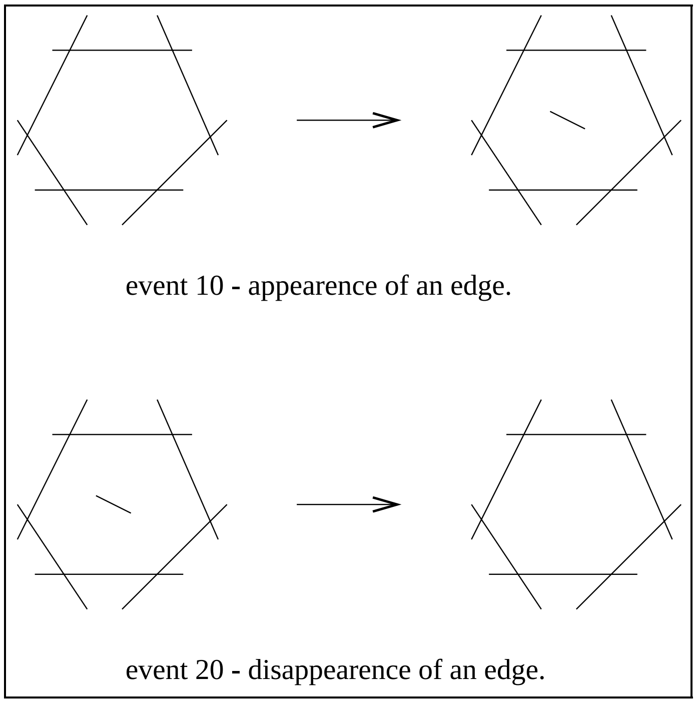
event 10 - appearence of an edge.
event 20 - disappearence of an edge.

Figure A.1: Appearance/disappearance of an edge

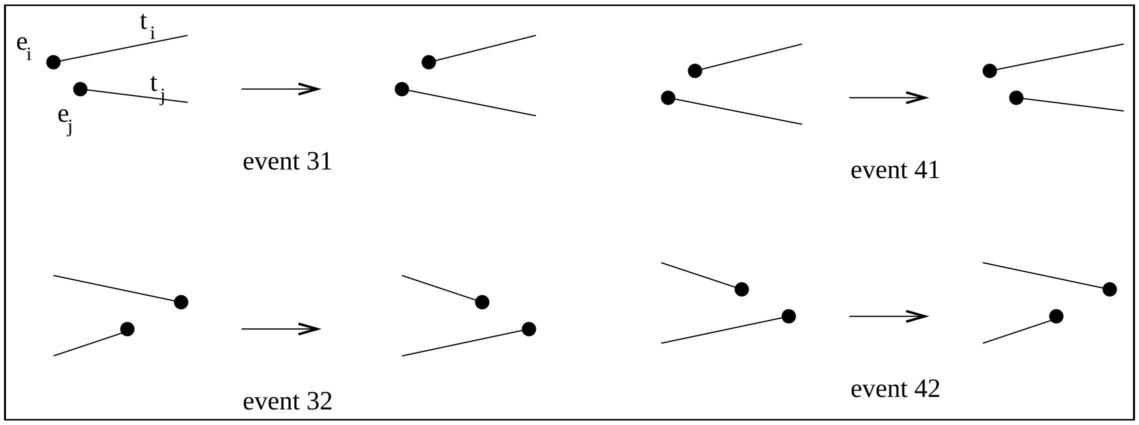

Figure A.2: Vertical visibility of two vertices

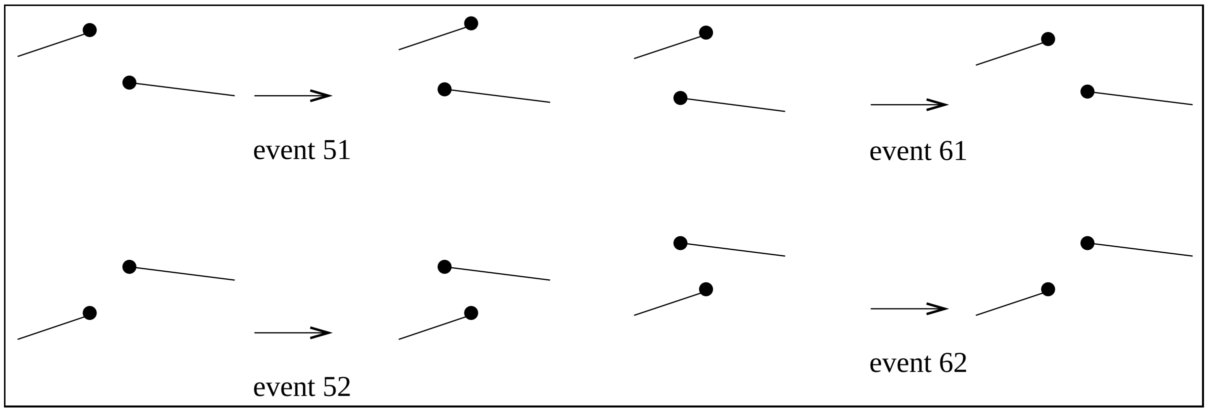

Figure A.3: Vertical visibility of two vertices

is, the wall at that point has zero height.

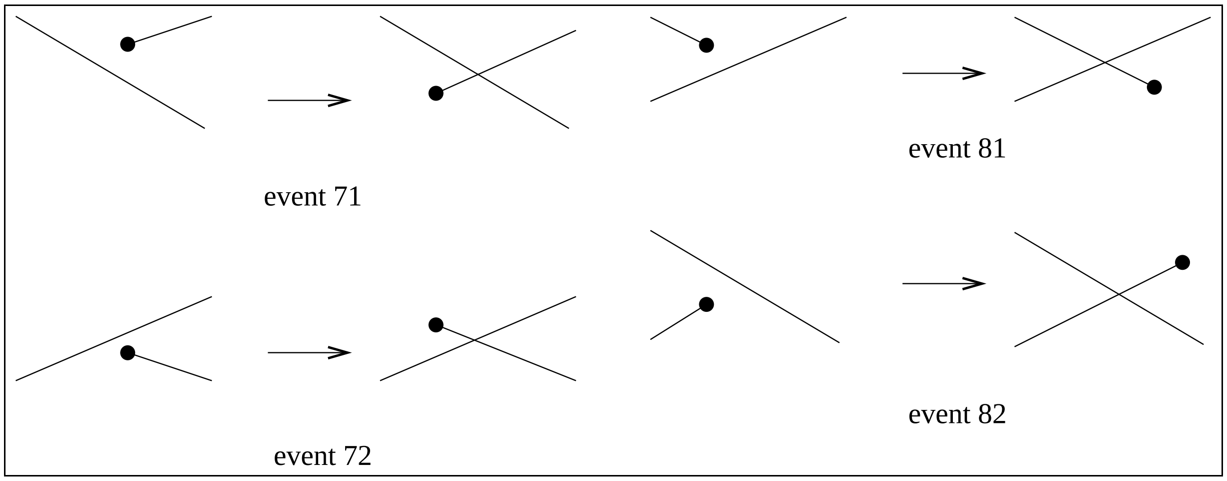

Figure A.4: Appearance of an edge

Events 201, 202, 211, 212, 221, 222, 231 and 232 are all events where a boundary edge and an intersection of two triangle are vertically visible. See Figures A.6 and A.7. This type of events occur on features of the vertical wall erected from the boundary edge. All these events occur on an inner vertex of the lower (upper) envelope.

Events 241 and 242 are events where three triangles intersect. See Figure A.8. This type of event is detected during the sweep. Whenever we obtain, during the sweep, a face which is bounded by three edges we check

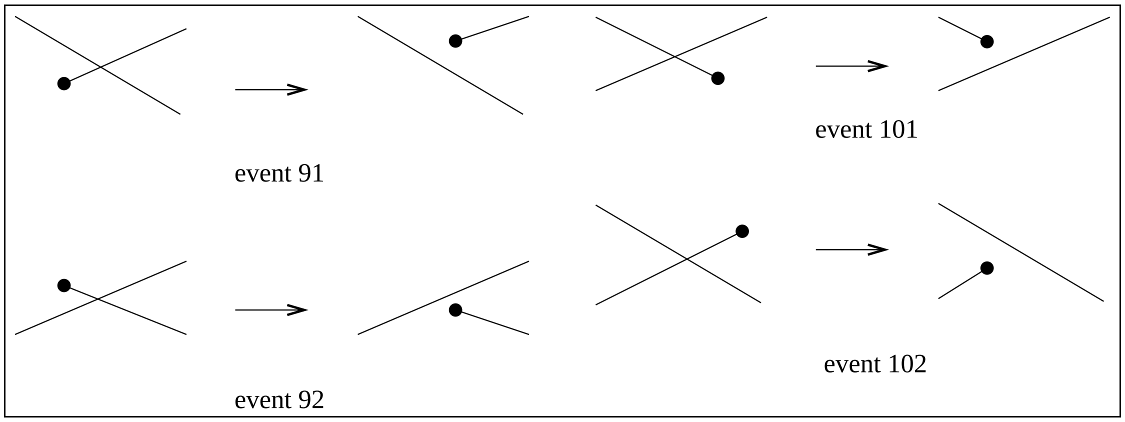

Figure A.5: Disappearance of an edge

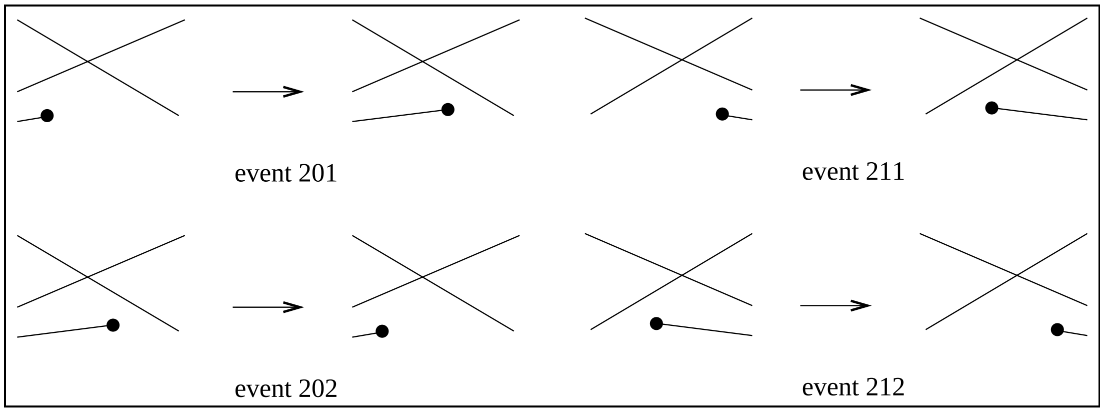

Figure A.6: A boundary edge and an intersection are vertically visible

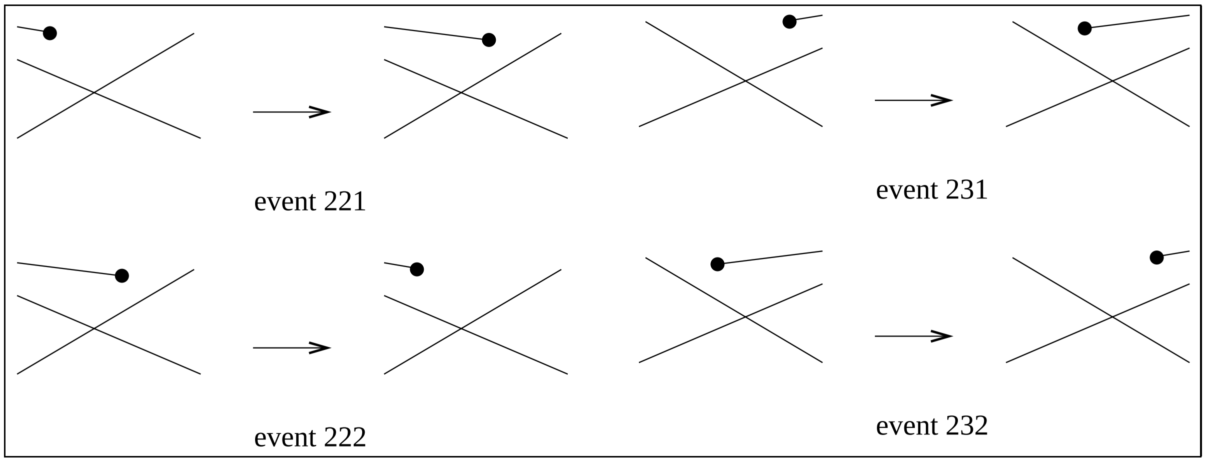

Figure A.7: A boundary edge and an intersection are vertically visible

if the triangles corresponding to those edges intersect in one point.

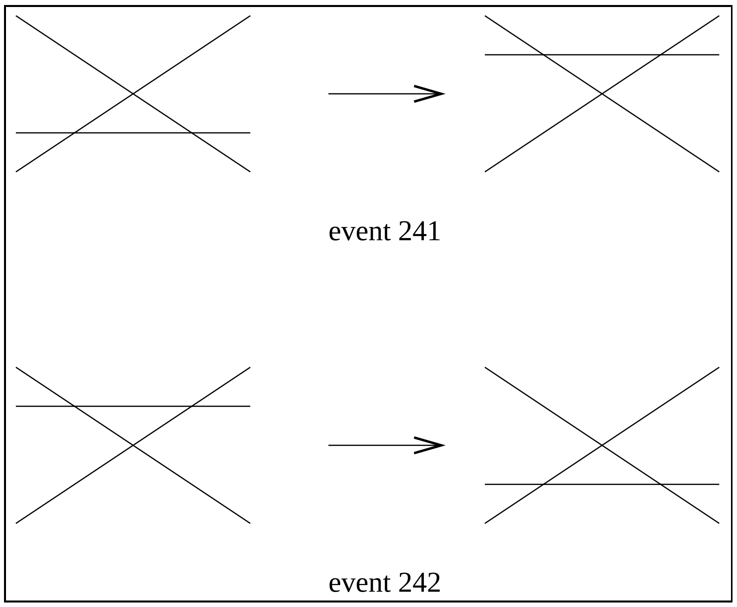

Figure A.8: Three triangles intersect

Appendix B

Handling Events

Since we divided the event types into so many subtypes handling each event is trivial.

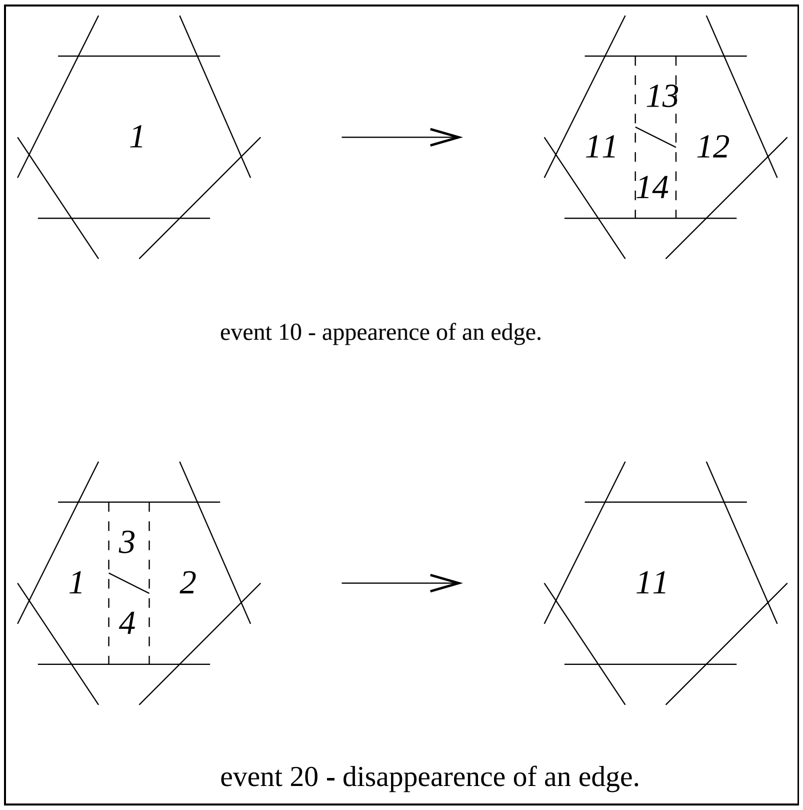
1
11
12
13
14
event 10 - appearence of an edge.
1
2
3
4
11
event 20 - disappearence of an edge.

Figure B.1: Faces in events 10 and 20

Events 10 and 20 are handled by creating a new face structure, or destroying a face structure respectively.

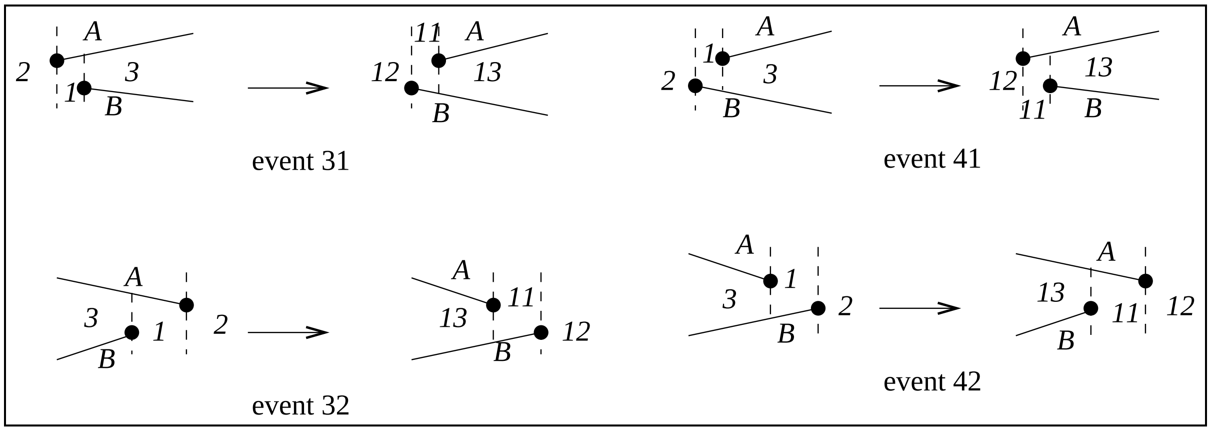

Figure B.2: Faces in events 31,32,41 and 42

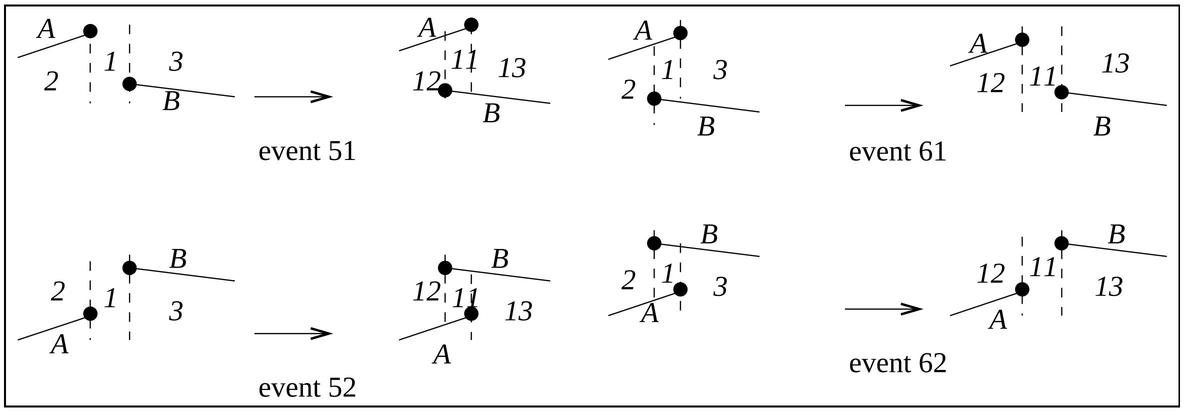

Figure B.3: Faces in events 51,52,61 and 62

Figure B.2 illustrates how we handle events 31, 32, 41 and 42, and Figure B.3 how we handle events 51, 52, 61 and 62. Event 31 is handled by deleting face 1, and creating face 11. The right wall of face 11 is set to be the left wall of face 1. The left wall of face 11 is set to be the right wall of face 1. The ceiling of face 11 is set to be the right most triangle in the ceiling chain of face 2. The floor of face 11 is set to be the triangle that contains the lower edge that generated this event. The witness for the right wall of face 2 is changed to be the witness of the right wall of face 1 to create face

12. The witness for the left wall of face 3 is changed to be the witness of the left wall of face 1 to create face 13.

Events 32, 41, 42, 51, 52, 61 and 62 are handled similarly to event 31, by swapping the witnesses for the right/left wall and the left/right wall of face 1 to create face 11. The witness for the left/right wall of face 2 is changed to create face 12. The witness for the right/left wall of face 3 is changed to create face 13.

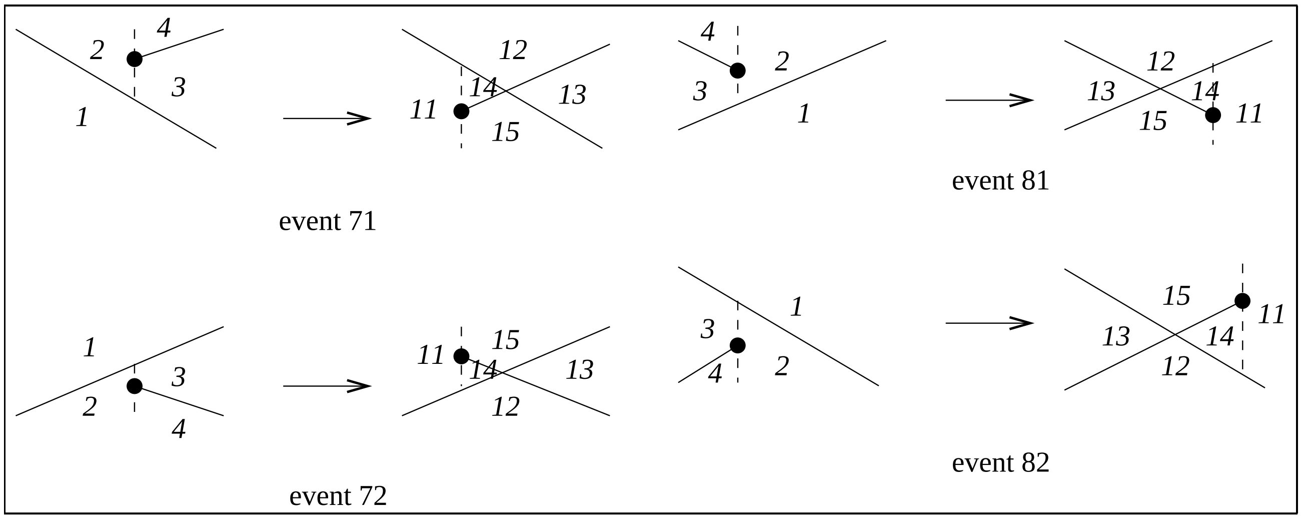

Figure B.4: Faces in events 71,72,81 and 82

Figure B.4 illustrates how we handle events 71, 72, 81 and 82, and Figure B.5 how we handle events 91, 92, 101 and 102. Event 71 is handled by splitting face 1 into faces 11 and 15. face 11 is the left part of face 1 and face 15 is the right part of face 1. We create face 14. Its ceiling is the "penetrated" edge and its floor is the "penetrating" edge. We also set its left witness to be the left endpoint of the "penetrating" edge, and the right witness to null. We set the witness of the left wall of face 3 to null in order to create face 13. We merge face 2 and face 4 to create face 12.

Events 72, 81 and 82 are handled similarly by splitting face 1 into face 11 and face 15. Creating face 14. Changing the witness of the left wall of face 3 to create face 13. Joining face 2 and face 4 to create face 12.

Event 91 is handled by merging face 5 and face 1 to create face 11. We split face 2 to create face 12 and face 14. face 12 is the left part of face 2, and face 14 is the right part of face 2. We change the witness of the left wall

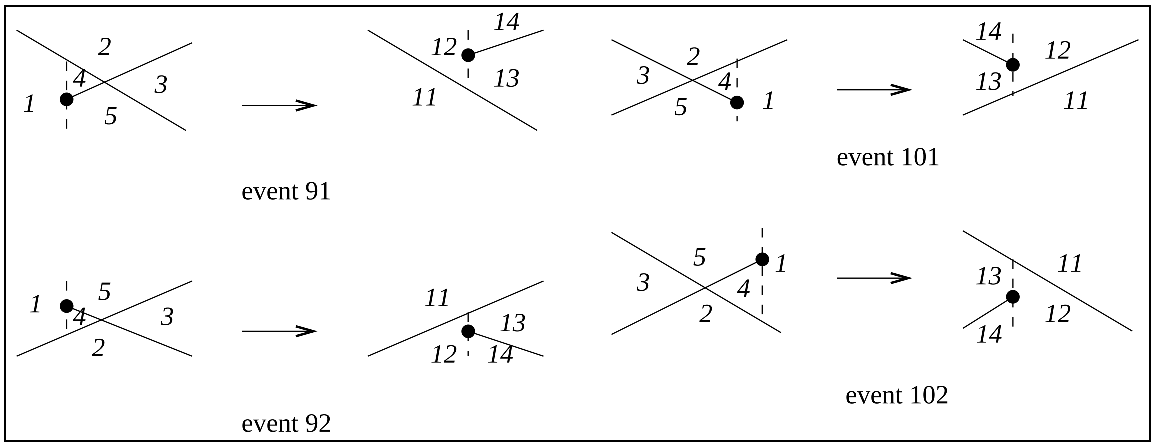

Figure B.5: Faces in events 91,92,101 and 102

of face 3 to be the left endpoint of the "penetrating" edge in order to create face 13.

Events 92, 101 and 102 are handled similarly by merging face 5 and face 1 to create face 11. Splitting face 2 to create face 12 and face 14. Changing the witness of the left wall of face 3 to create face 13.

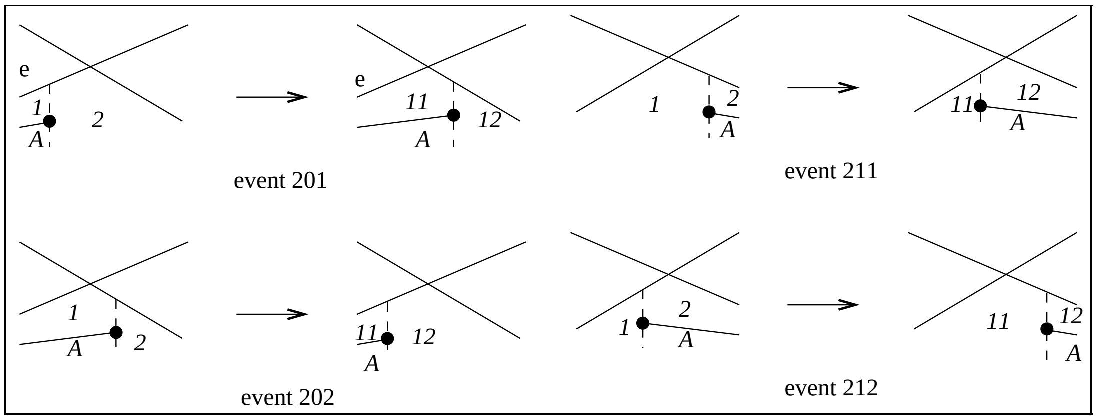

Figure B.6: Faces in events 201,202,211 and 212

Figure B.6 illustrates how we handle events 201, 202, 211 and 212, and Figure B.7 illustrates how we handle events 221, 222, 231 and 232. Event 201 is handled by adding the left edge of the intersection, e, to the ceiling of face 1 to create face 11, and removing this edge from the ceiling of face 2 to create face 12. Events 202, 211, 212, 221, 222, 231 and 232 are handled

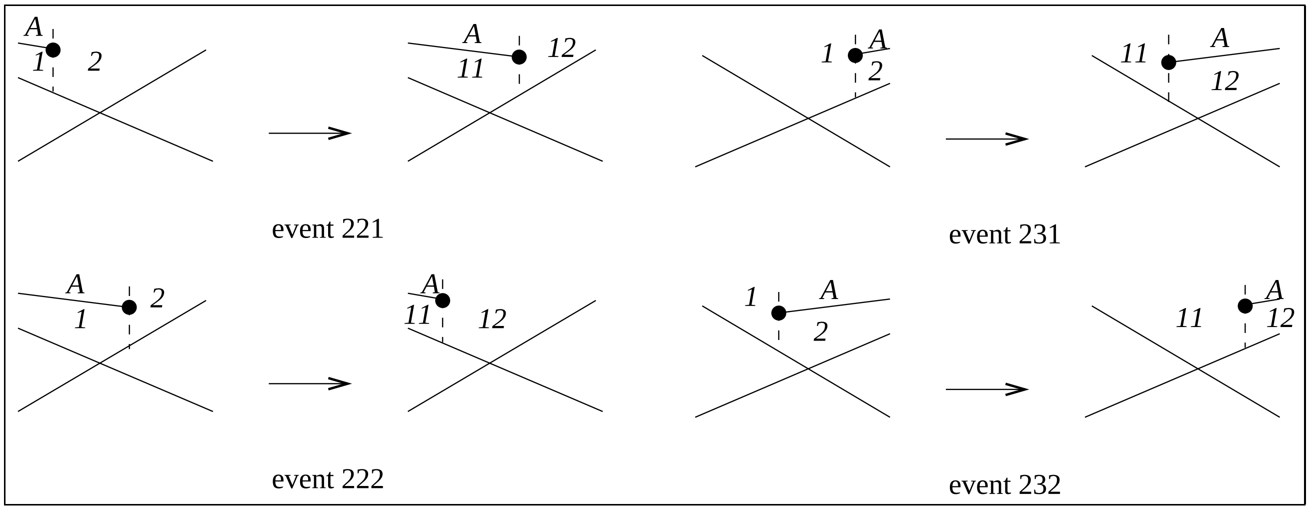

Figure B.7: Faces in events 221,222,231 and 232

similarly by adding and removing an intersection to faces.

Event 241 is handled by adding e_1 to the ceiling chain of face 1 from the left, to create face 11. We remove e_2 from the ceiling chain of face 2 to create face 12. We add e_2 to the ceiling chain of face 3 from the left to create face 13. We remove e_1 from the floor chain of face 4 to create face 14. We add e_2 to the floor chain of face 5 between e_1 and e_3 to create face 15. We remove e_2 from the floor chain of face 6 to create face 16. We destroy face 7, and create face 17 whose ceiling chain consists of e_2 and whose floor chain consists of e_1 and e_3.

Event 241 is handled similarly. See Figure B.8 for illustrations.

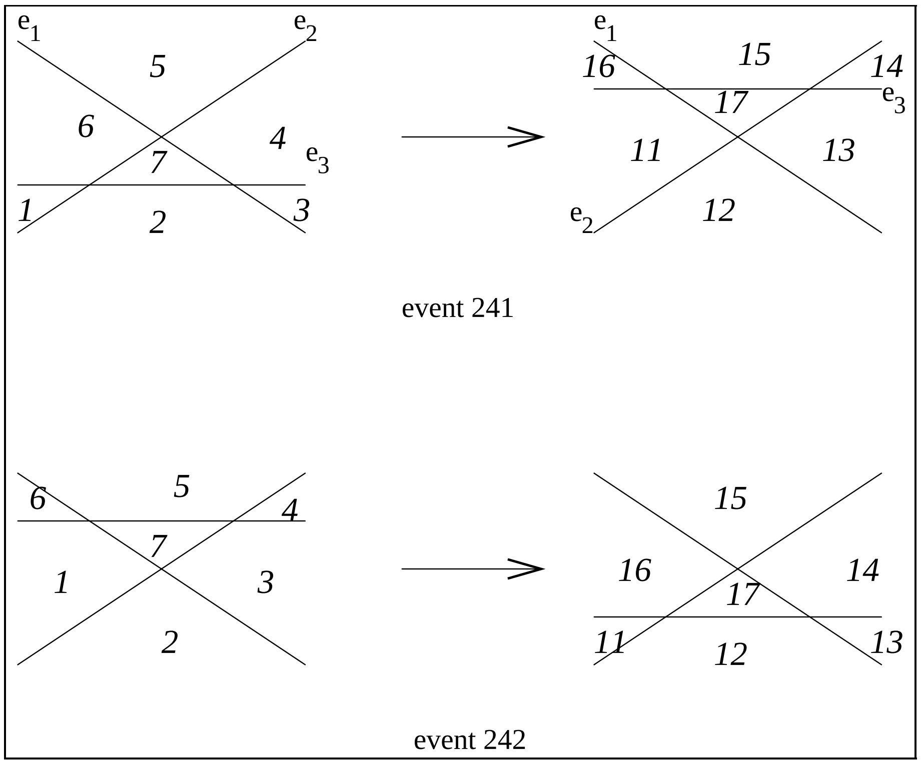

Figure B.8: Faces in events 241 and 242

Appendix C

Detecting Polyhedral Surfaces Events

Events 73, 74, 83, 84, 93, 94, 103 and 104 are all events where a non xy-monotone boundary edge intersects a triangle. This type of events occur on features of the vertical wall in the three-dimensional decomposition erected from the edge. See Figures C.1 and C.2

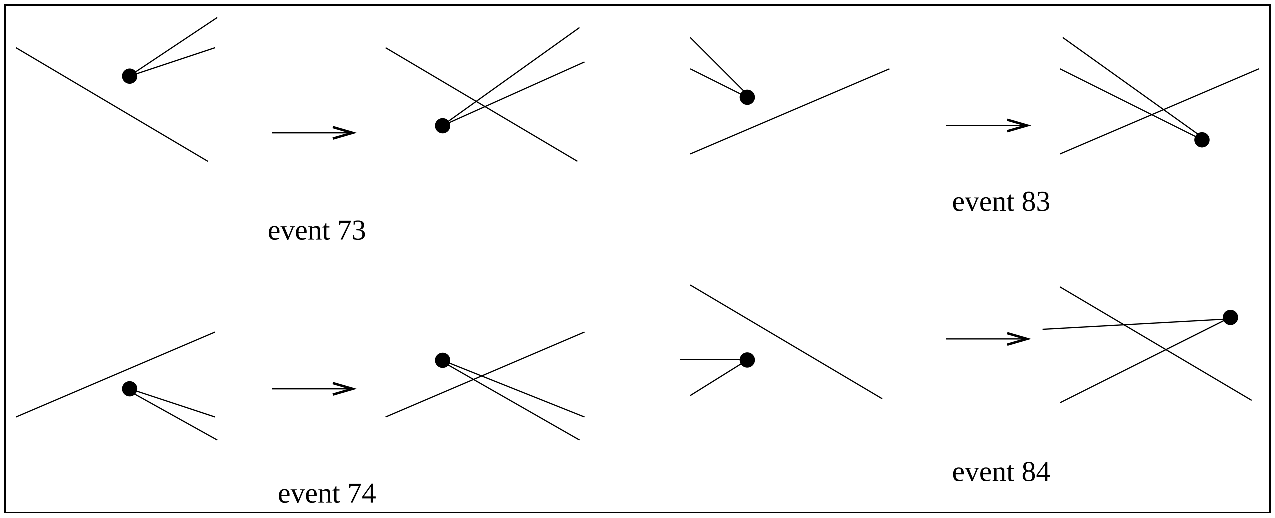

Figure C.1: Non xy-monotone boundary edge intersects a triangle

Events 111, 112, 113 and 114 are all events where an xy-monotone boundary edge which is common to two triangles, t_1 and t_2, intersect another triangle t. Before each of these events t_1 intersect t, and t_2 does not intersect

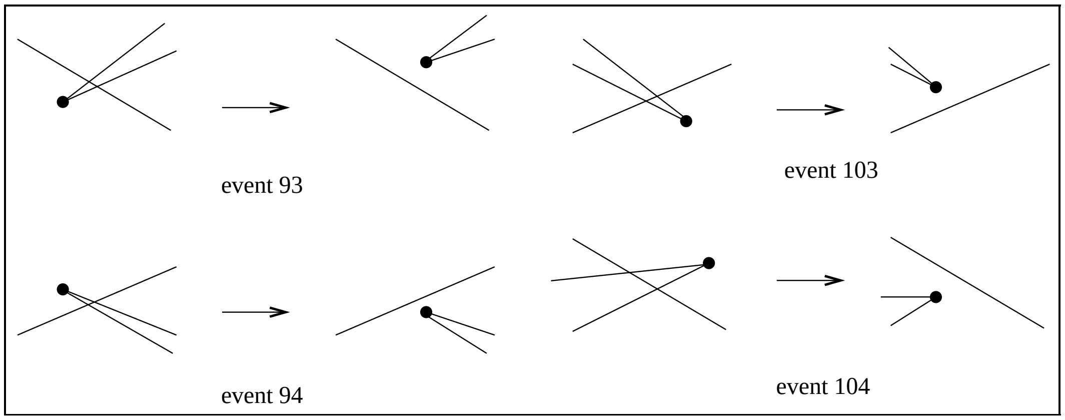

Figure C.2: Non xy-monotone edge boundary intersects a triangle

t. After each of these events t_1 does not intersect t, and t_2 intersects t. This type of events occur on features of the vertical wall in the three-dimensional decomposition erected from the edge. See Figure C.3.

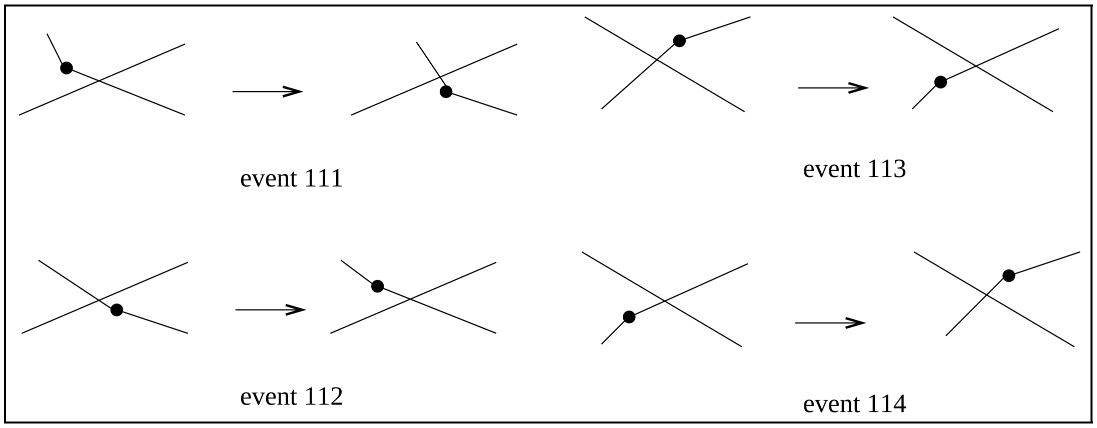

Figure C.3: xy-monotone boundary edge intersects a triangle

Events 121 and 124 are events where an xy-monotone boundary edge common to two triangles, t_1 and t_2, intersect a triangle t, when both of the triangle don't intersect t before the event, and both of them intersect t after the event.

Events 122 and 123 are events where an xy-monotone boundary edge common to two triangles, t_1 and t_2, intersect a triangle t, when both of the triangle intersect t before the event, and both of them don't intersect t after the event.

These types of events occur on features of the vertical wall in the three-dimensional decomposition erected from the edge. See Figure C.4.

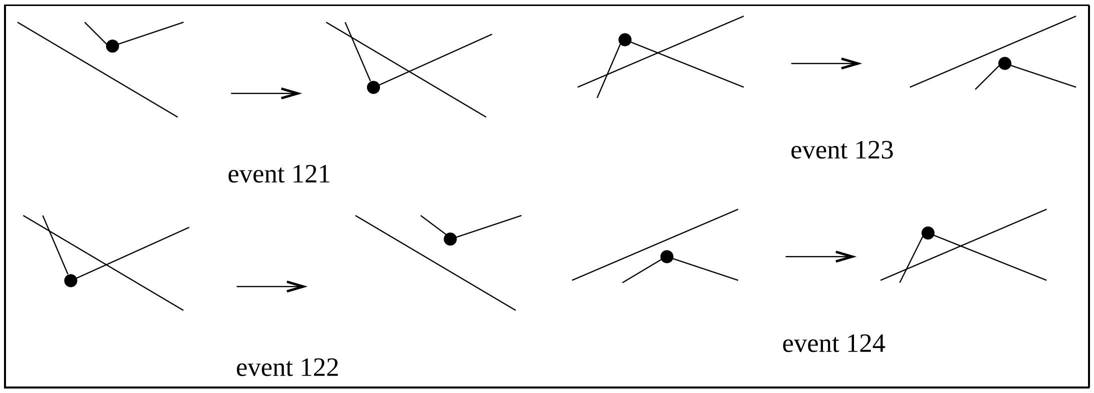

Figure C.4: xy-monotone boundary edge intersects a triangle

Events 400, 410 and 420 occur at corners common to two or more triangles and are trivial to detect. Event 410 is an event where a polygonal line appears on the sweep plane. Event 420 is an event where a polygonal line disappears on the sweep plane. Event 400 is an event where neither event 410 nor 420 occur. See Figure C.5.

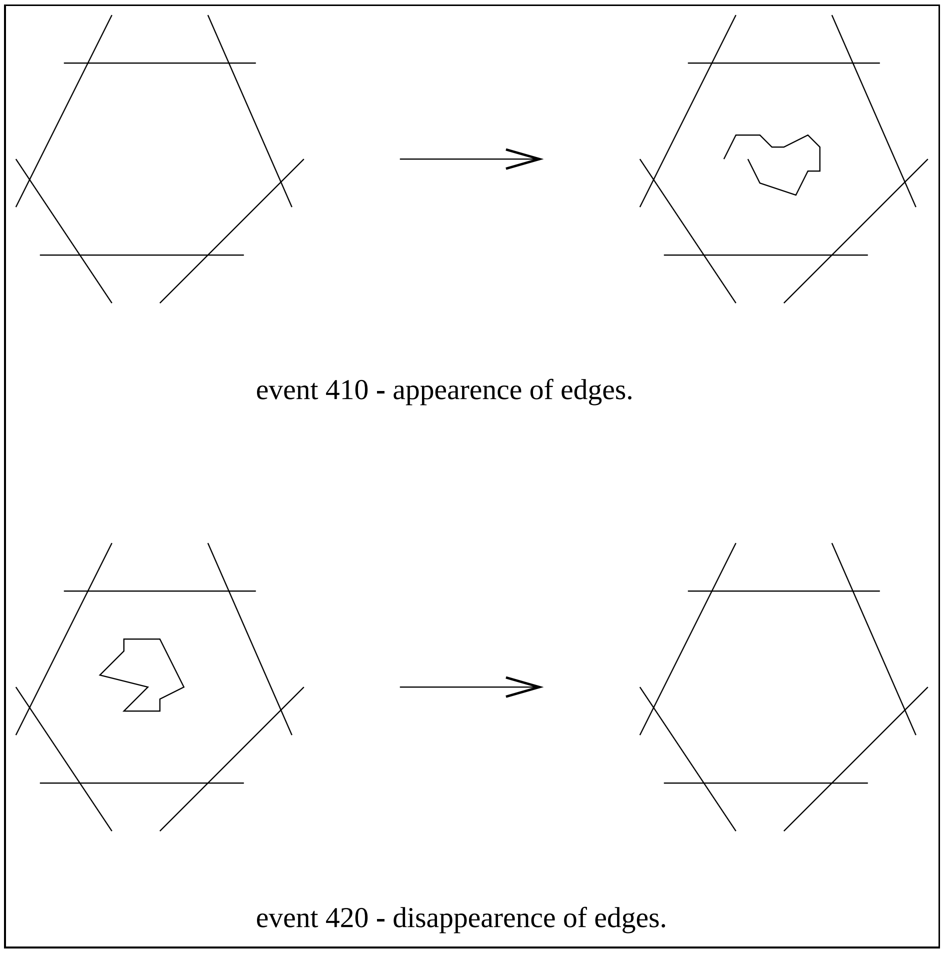
event 410 - appearence of edges.
event 420 - disappearence of edges.

Figure C.5: Appearance/disappearance of a polygonal line

Appendix D

Handling Polyhedral Surfaces Events

Again, since we divided the event types into so many sub-types handling each event is trivial.

Event 73 is handled by splitting face 1 into face 11 and face 16. Face 11 is the left part of face 1 and face 16 is the right part of face 1.

We create face 12. Its ceiling is the “penetrated” edge and its floor is the upper “penetrating” edge. We also set its left witness to be the left endpoint of the “penetrating” edge, and the right witness to null. We create face 17. Its ceiling consists of the upper “penetrating” edge and the “penetrated” edge. Its floor is the lower “penetrating” edge. We also set its left witness to be the left endpoint of the “penetrating” edge, and the right witness to null. We change the left witness of face 4 to null, and add the “penetrated” edge to its floor from the left to create face 14. We set the witness of the left wall of face 5 to null in order to create face 15. We merge face 2 and face 3 to create face 13. Events 74, 83 and 84 are handled similarly. See Figure D.1 for a description of these events.

Event 93 is handled by merging face 1 and face 6 to create face 11. We split face 3 to create face 12 and face 13. Face 12 is the left part of face 2, and face 13 is the right part of face 2. We change the witness of the left wall of face 5 to the left endpoint of the “penetrating” edge in order to create face 15. We change the left witness of face 4 to be the left endpoint of the “penetrating” edge and remove the “penetrated” edge from its floor to create face 14. We destroy face 7. Events 94, 103 and 104 are handled similarly. See Figure D.2 for a description of these events.

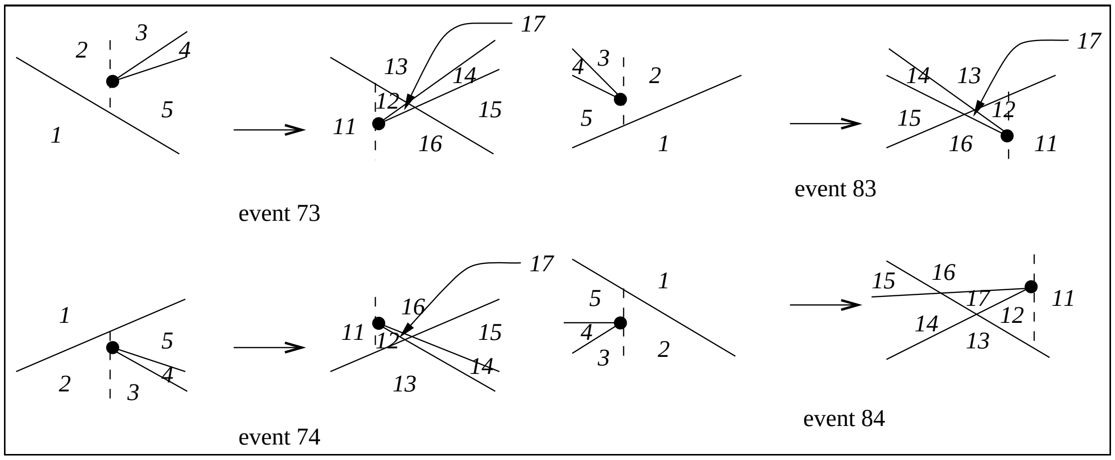

Figure D.1: Faces in events 73, 74, 83, and 84

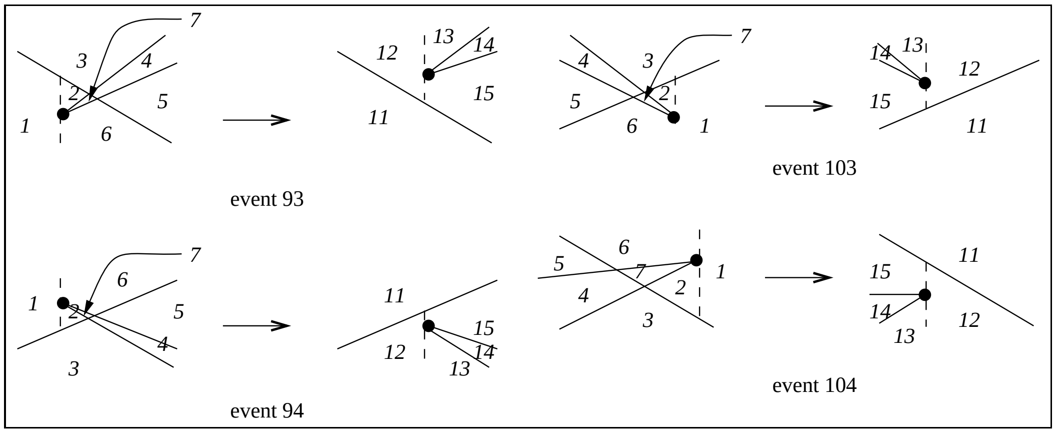

Figure D.2: Faces in events 93, 94, 103 and 104

Event 111 is handled by merging face 1 and face 2 to create face 11. We create face 12. Its ceiling is the "penetrated" edge. Its floor is the left "penetrating" edge. Its left witness is set to null, and its right witness is set to the right endpoint of the left "penetrating" edge. We change the witness of the left wall of face 3 to the left endpoint of the right "penetrating" edge in order to create face 13. We split face 4 to create face 14 and face 15. Face 14 is the right part of face 4 and face 15 is the left part of face 4. We change the right witness of face 6 to null in order to create face 16. We destroy face 5. Events 112, 113 and 114 are handled similarly. See Figure D.3 for a description of these events.

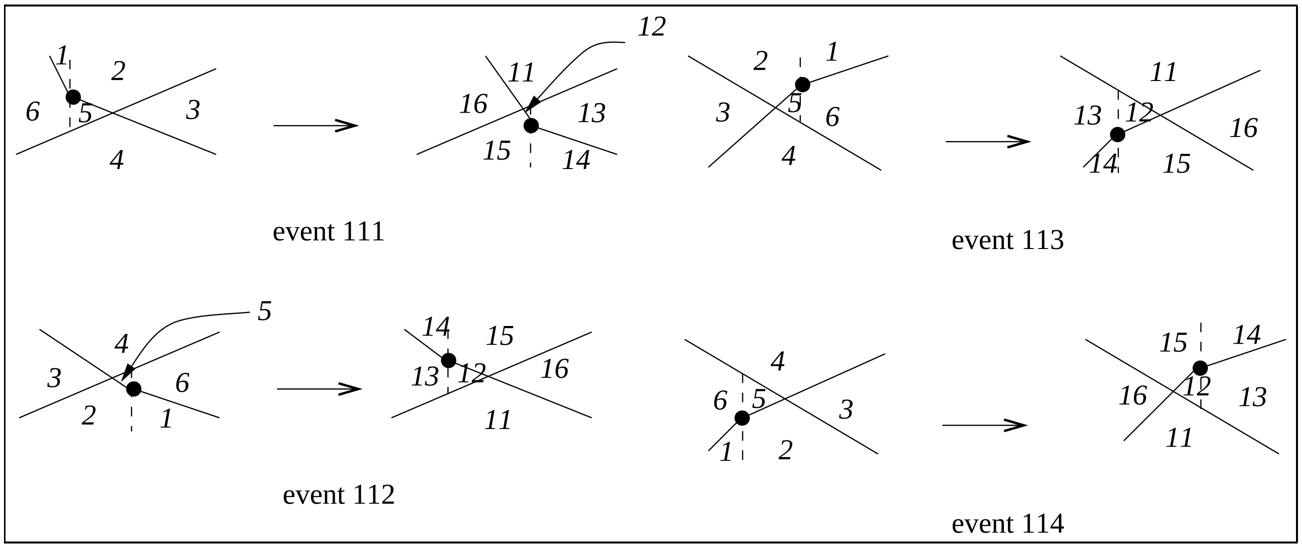

Figure D.3: Faces in events 111, 112, 113 and 114

Event 121 is handled by merging face 1 and face 2 to create face 11. We change the witness of the right wall of face 3 to null in order to create face 12. We split face 4 into faces 14 and 13. Face 14 is the left part of face 4 and face 13 is the right part of face 4. We also add the left "penetrating" edge to the ceiling of face 14, from the right, and we add the right "penetrating" edge to the ceiling of face 13, from the left. We change the witness of the left wall of face 5 to null in order to create face 15. We create face 16. Its floor is the left "penetrating" edge. Its ceiling is the "penetrated" edge. Its right witness is set to null, and its left witness is the right endpoint of the left "penetrating" edge. We create face 17. Its floor is the right "penetrating" edge. Its ceiling is the "penetrated" edge. Its left witness is set to null, and its right witness is the left endpoint of the right “penetrating” edge. Events 122, 123 and 124 are handled similarly. See Figure D.4 for a description of these events.

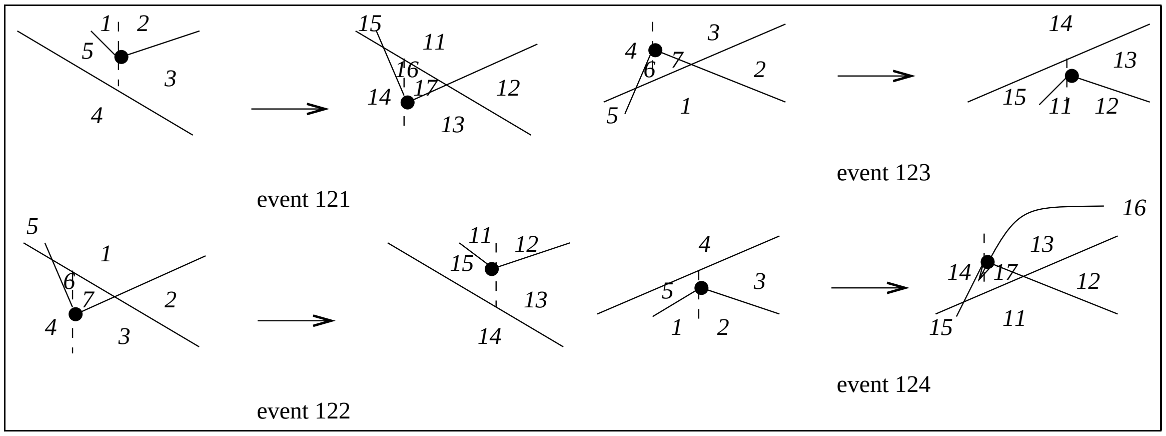

Figure D.4: Faces in events 121, 122, 123 and 124

Recall that event 410 is the event where one or more polygonal lines appear on the sweep plane because a first corner of some triangles have been reached. To handle event 410 we find the segments that result from intersection the sweep plane with the new appearing triangles. We compute the arrangement of these segments, and refine it by erecting a vertical segment from each outer vertex. We get a subdivision of face 1 into possibly many faces. We call the leftmost face face 11, and the rightmost face, face 12. We copy the left part of face 1 to face 11, and the right part of face 1 to face12. We also figure out which faces are neighbors, from the arrangement on the sweep plane.

Recall that event 420 is the event where a polygonal line disappears from the sweep plane because a last corner of some triangles have been reached. We handle event 420 by destroying all the faces that have a disappearing triangle as their floor or as their ceiling. We also destroy all the faces whose left witness, or right witness are disappearing triangles. We mark the leftmost face whose right witness is a disappearing triangle as face 1, and the rightmost face whose left witness is a disappearing triangle as face 2. We merge face 1 and face 2 to get face 11.

See Figure D.5 for an illustration of these events.

Recall that event 400 is the event where triangle corners have been reached but not all of the corners are first corners, and not all of the corners are last

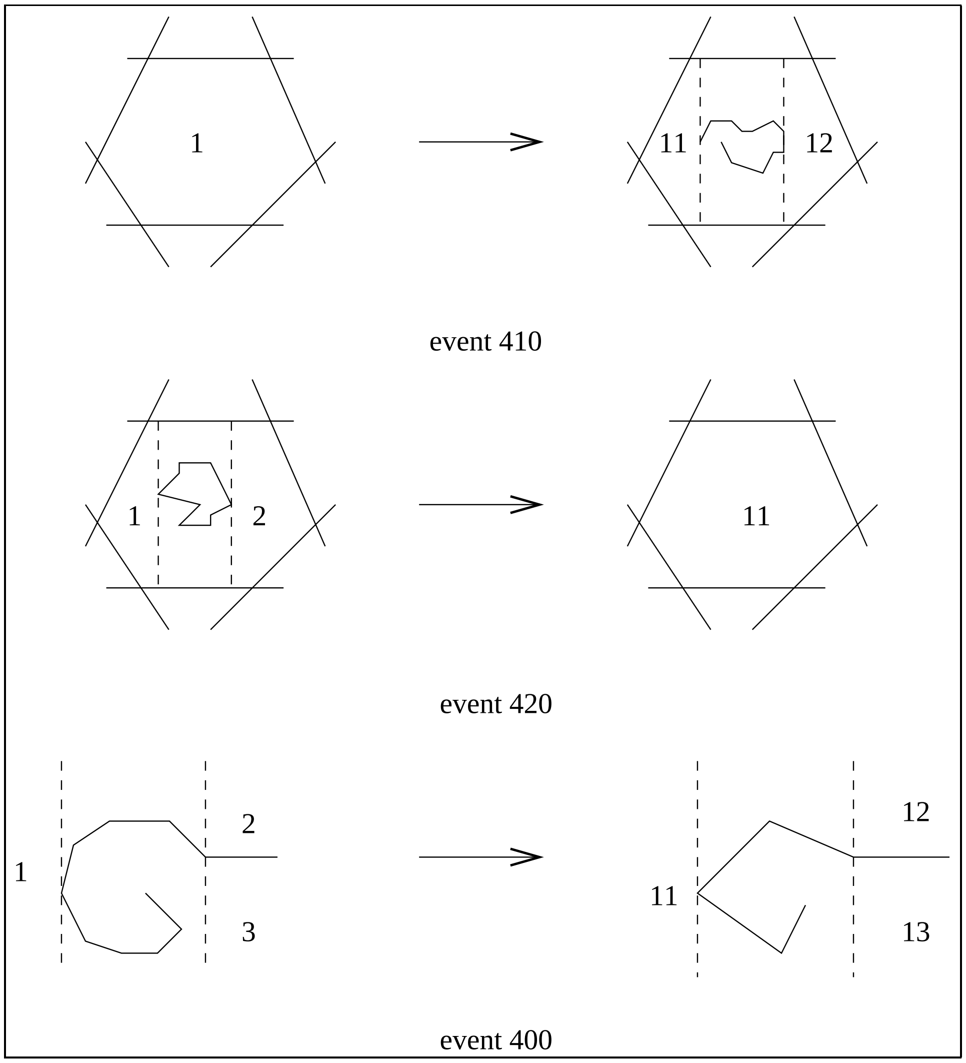

Figure D.5: Faces in events 410 and 420

corners. We destroy faces whose ceiling, floor or one of their witness triangles have reached a corner in this event. We find the segments on the sweep plane. These segments results from triangles that the sweep plane has reached their first corner, and from triangles that the sweep plane has reached their second corner. We conclude by computing the arrangement of these segments.

Bibliography

[1] P. K. Agarwal, E. Flato, and D. Halperin. Polygon decomposition for efficient construction of Minkowski sums. In *Proc. 8th European Symposium on Algorithms*, volume 1879 of *Lecture Notes in Computer Science*, pages 20–31. Springer-verlag, Saarbrücken, 2000.

[2] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22(4):794–806, 1993.

[3] P. K. Agarwal and J. Matoušek. On range searching with semialgebraic sets. *Discrete Comput. Geom.*, 11:393–418, 1994.

[4] P. K. Agarwal and M. Sharir. Arrangements. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, pages 49–119. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1999.

[5] B. Aronov and M. Sharir. Triangles in space or building (and analyzing) castles in the air. *Combinatorica 10(2)*, pages 137–173, 1990.

[6] M. J. Atallah, M. T. Goodrich, and K. Ramaiyer. Biased finger trees and three-dimensional layers of maxima. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 150–159, 1994.

[7] H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. *J. Algorithms*, 17:342–380, 1994.

[8] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, September 1979.

[9] C. Burkinel, R. Fleischer, K. Melhorn, and S. Schirra. Efficient exact geometric computation made easy. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 341–350, 1999.

[10] *The CGAL User Manual, Version 2.1*, 2000. http://www.cgal.org/.

[11] B. Chazelle. Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. *SIAM J. Comput.*, 13:488–507, 1984.

[12] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. A singly-exponential stratification scheme for real semi-algebraic varieties and its applications. In *Proc. 16th Internat. Colloq. Automata Lang. Program.*, volume 372 of *Lecture Notes Comput. Sci.*, pages 179–193. Springer-Verlag, 1989.

[13] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3):229–249, 1990.

[14] S. W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM J. Comput.*, 21:972–999, 1992.

[15] Y.-J. Chiang, F. P. Preparata, and R. Tamassia. A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 44–53, 1993.

[16] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9):1412–1434, September 1992.

[17] Y.-J. Chiang and R. Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *Internat. J. Comput. Geom. Appl.*, 2(3):311–333, 1992.

[18] K. Clarkson, H. Edelsbrunner, Leonidas J. Guibas, Micha Sharir, and Emo Welzl. Combinatorial complexity bounds for arrangements of curves and spheres. *Discrete Comput. Geom.*, 5:99–160, 1990.

[19] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Comput.*, 17:830–847, 1988.

[20] M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*, volume 703 of *Lecture Notes Comput. Sci.* Springer-Verlag, Berlin, Germany, 1993.

[21] M. de Berg, L.J. Guibas, and D. Halperin. Vertical decomposition for triangles in 3-space. *Discrete Comput. Geom. 14*, pages 35–62, 1995.

[22] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. *Algorithmica*, 12:30–53, 1994.

[23] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[24] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.

[25] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algorithms*, 13(1):33–54, 1992.

[26] S. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, July 1996.

[27] M. T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. *In Proc. 23rd Annu. ACM Sympos. Theory Comput.*, 1991.

[28] M. T. Goodrich and R. Tamassia. Dynamic ray shooting and shortest paths via balanced geodesic triangulations. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 318–327, 1993.

[29] M.T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd Annu. ACM Sympos. Theory Comput*, pages 523–533, 1991.

[30] T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library, version 2.0.2*, June 1996.

[31] L. Guibas and M. Sharir. Combinatorics and algorithms of arrangements, 1993.

[32] D. Halperin. Arrangements. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 21, pages 389–412. CRC Press LLC, Boca Raton, FL, 1997.

[33] D. Halperin and M. Sharir. Arrangements and their applications in robotics: Recent developments. In K. Goldberg, D. Halperin, J.-C. Latombe, and R. Wilson, editors, *Proc. Workshop Algorithmic Found. Robot.*, pages 495–511. A. K. Peters, Wellesley, MA, 1995.

[34] D. Halperin and C. R. Shelton. A perturbation scheme for spherical arrangement with application to molecular modeling. Report AI MEMO 1618, MIT, December 1997.

[35] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[36] V. Koltun. Almost tight upper bounds for vertical decompositions in four dimensions. manuscript.tel aviv university.

[37] K. Mehlhorn, S. Näher, C. Uhrig, and M. Seel. *The LEDA User Manual, Version 4.1.* Max-Planck-Insitut für Informatik, 66123 Saarbrücken, Germany, 2000.

[38] K. Mehlhorn, S. Näher, C. Uhrig, and M. Seel. *The LEDA User Manual, Version 4.1.* Max-Planck-Insitut für Informatik, 66123 Saarbrücken, Germany, 2000.

[39] K. Melhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[40] K. Mulmuley. Hidden surface removal with respect to a moving point. *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 512–522, 1991.

[41] K. Mulmuley. Randomized multidimensional search trees: Further results in dynamic sampling. *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 216–227, 1991.

[42] M. Overmars and M. Sharir. Output-sensitive hidden surface removal. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 598–603, 1989.

[43] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[44] F. P. Preparata and R. Tamassia. Efficient point location in a convex spatial cell-complex. *SIAM J. Comput.*, 21:267–280, 1992.

[45] W. Pugh. Skiplists: A probalistic alternative to balanced trees. *Algorithms and Data Structures*, pages 668 – 676, 1990.

[46] S. Raab. Controlled pertubation for arrangements of polyhedral surfaces with application to swept volumes. M.Sc. thesis, Bar-Ilan University, 1999.

[47] S. Raab. Controlled perturbation for arrangements of polyhedral surfaces with application to swept volumes. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, 1999.

[48] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986.

[49] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, 1995.

[50] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18(3):305–363, 1997.

[51] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–381, 1983.

[52] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.