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

Robust, Generic and Efficient Construction of Envelopes of Surfaces 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

Michal Meyerovitch

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

May 2006

Acknowledgments

I deeply thank my supervisor, Prof. Dan Halperin, for his guidance and his help during the work on this thesis. I wish to thank him for all his patience, support and encouragement.

I thank all the fellow students in the Geometry lab for their assistance.

I thank Lutz Kettner and Eric Berberich for their invitation and hospitality in the MPII, and for fruitful collaboration.

Finally, I would like to thank my husband Tom for all the support he has given me.

Work reported in this thesis has been supported in part by the IST Programme of the EU as Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes).

Abstract

Lower envelopes are fundamental structures in computational geometry that have many applications, such as computing general Voronoi diagrams and performing hidden surface removal in computer graphics. We present a generic, robust and efficient implementation of the divide-and-conquer algorithm for computing the envelopes of surfaces in $\mathbb{R}^3$. To the best of our knowledge, this is the first exact implementation that computes envelopes in three-dimensional space. Our implementation is based on CGAL (the Computational Geometry Algorithms Library) and is designated as a CGAL package. The separation of topology and geometry in our solution allows for the reuse of the algorithm with different families of surfaces, provided that a small set of geometric objects and operations on them is supplied. We used our algorithm to compute the lower and upper envelope for several types of surfaces. Exact arithmetic is typically slower than floating-point arithmetic, especially when higher order surfaces are involved. Since our implementation follows the exact geometric computation paradigm, we minimize the number of geometric operations, and by that significantly improve the performance of the algorithm in practice. Our experiments show interesting phenomena in the behavior of the divide-and-conquer algorithm and the combinatorics of lower envelopes of random surfaces.

Contents

Chapter 1

Introduction

Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be a collection of n (hyper)surface patches in $\mathbb{R}^d$. Let $x_1, \ldots, x_d$ denote the axes of $\mathbb{R}^d$, and assume that each s_i is monotone in $x_1, \ldots, x_{d-1}$, namely every line parallel to the x_d-axis intersects s_i in at most one point. Regard each surface patch s_i in $\mathcal{S}$ as the graph of a partially defined $(d-1)$-variate function $s_i(\bar{x})$. The *lower envelope* $\mathcal{E}_\mathcal{S}$ of $\mathcal{S}$ is the pointwise minimum of these functions: $\mathcal{E}_\mathcal{S}(\bar{x}) = \min s_i(\bar{x}),\ \bar{x} \in \mathbb{R}^{d-1}$, where the minimum is taken over all functions defined at $\bar{x}$. Similarly, the *upper envelope* of $\mathcal{S}$ is the pointwise maximum of these functions.

The *minimization diagram* $\mathcal{M}_\mathcal{S}$ of $\mathcal{S}$ is the subdivision of $\mathbb{R}^{d-1}$ into maximal connected cells such that $\mathcal{E}_\mathcal{S}$ is attained by a fixed subset of functions over the interior of each cell. The minimization diagram is the subdivision of $\mathbb{R}^{d-1}$ obtained by the projection of the lower envelope of $\mathcal{S}$ in the x_d direction. In the same manner, the *maximization diagram* of $\mathcal{S}$ is the subdivision of $\mathbb{R}^{d-1}$ induced by the upper envelope of $\mathcal{S}$.

Many efforts have been invested in recent years on solving two-dimensional geometric problems in practice [1, 67]. Some work has been done on three-dimensional problems also, but it was mainly concentrated on selective problem, namely problems in which the numerical data of the output is a subset of the data in the input, as opposed to constructive problems, where new numerical data (such as intersection points of geometric objects) is to be computed for the output. Computing envelopes is of course a constructive problem.

In this thesis we present an exact and generic implementation of the divide-and-conquer algorithm for constructing the envelope of surface patches in $\mathbb{R}^3$. Our solution is complete in the sense that it handles all degenerate cases, and at the same time it is efficient. To the best of our knowledge, this is the first implementation of this kind. Our implementation is based on the CGAL library and is designated as a CGAL package. The problem of computing the envelope is somewhat two-and-a-half dimensional, since the input is three-dimensional, but the output is naturally represented as a two-dimensional object, the minimization diagram. We use the CGAL two-dimensional arrangement package for the representation of the minimization diagram. We believe that computing envelopes in three-space is an important step toward solving practical problems in $\mathbb{R}^3$.

The separation of topology and geometry in our solution allows for the reuse of the algorithm with different families of surfaces, provided that a small set of geometric objects

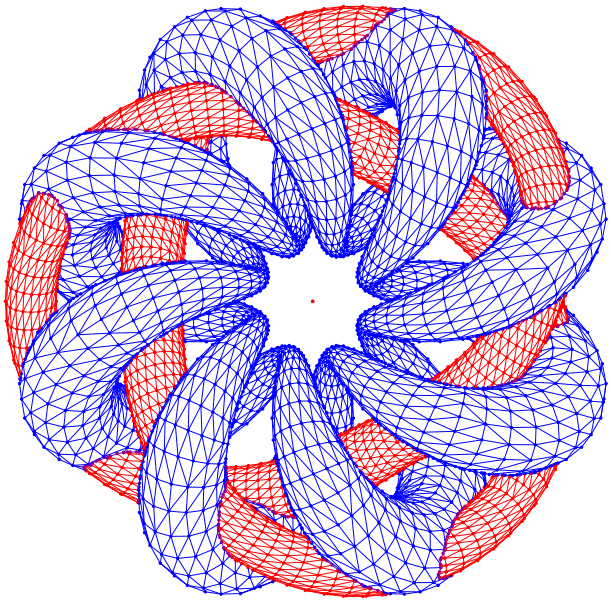


Figure 1.1: The minimization diagram of two triangulated surfaces with approximately 16,000 triangles. The triangulated surfaces input files were taken from `http://www.cs.duke.edu/~edels/Tubes/`.

and operations on them is supplied. We used our algorithm to compute the lower or upper envelope of sets of triangles, of sets of spheres and of sets of quadratic surfaces.

Our implementation follows the exact geometric computation paradigm. Exact arithmetic is typically slower than floating-point arithmetic, especially when higher order surfaces are involved. One of the main contributions of our work is minimizing the number of geometric operations, and by that significantly improving the performance of the algorithm in practice.

Our experiments show interesting phenomena in the behavior of the divide-and-conquer algorithm and the combinatorics of lower envelopes of random surfaces. In particular, they show that on some input sets the algorithm performs better than the worst-case bound, and the combinatorial size of the envelope is typically asymptotic much smaller than the worst-case bound. We hope that our work will motivate more theoretical research in this direction.

A paper describing the work of this thesis [53] will be presented at the 14$^{\text{th}}$ Annual European Symposium on Algorithms (ESA 2006).

Our code is available at `http://www.acm.org/jea/repository/esa06/esa2006.html`.

Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2 we review the background and related work on envelopes, as well as some preliminaries on the implementation of

geometric algorithms. In Chapter 3 we describe the divide-and-conquer algorithm for computing envelopes and give an overview of our solution. The *traits* class, which is a very significant component of our design is described in Chapter 4. Chapter 5 surveys the major algorithmic details, among them are the methods we use to reduce the amount of geometric computations and improve the performance of the algorithm. In Chapter 6 we briefly describe the work on envelopes of quadric surfaces. We present experimental results and discuss the practical performance of the algorithm in Chapter 7. In Chapter 8 we conclude the thesis and suggest directions for future work.

Chapter 2

Background

Before computing a geometric structure, one wishes to know its size. We begin this chapter with a brief review of the combinatorial complexity of envelopes in two, three and d-dimensions. We then present several applications of envelopes in three or more dimensions. Next, we describe some of the known algorithms for computing envelopes in three-space. Finally, we review related practical aspects. For simplicity, and since lower and upper envelopes are symmetric structures, we consider in this chapter only lower envelopes.

2.1 Combinatorial Complexity of Envelopes

The complexity of the lower envelope of a set of surfaces is defined as the complexity of its minimization diagram. Note that the minimization diagram might include features that do not appear in the arrangement of these surfaces. For example, in the lower envelope of a set of triangles in $\mathbb{R}^3$, the projection of edges of two triangles may intersect in the minimization diagram, although the triangles do not intersect in three-space. We review the combinatorial complexity of envelopes in two, three and d-dimensions.

Two-dimensional envelopes

The complexity of the lower envelope of n lines in the plane is $\Theta(n)$. The maximum combinatorial complexity of the lower envelope of n x-monotone Jordan arcs in the plane such that each pair intersects in at most s points, for some fixed constant s, is $\Theta(\lambda_{s+2}(n))$. The function $\lambda_s(n)$ is the maximum length of a *Davenport-Schinzel sequence* of order s on n symbols, and it is almost linear in n for any fixed s [62]. For example, for a set of line segments $k = 1$, and the combinatorial complexity of their lower envelope is $\Theta(\lambda_3(n)) = \Theta(n\alpha(n))$, where $\alpha(n)$ is the extremely slowly growing inverse of Ackermann's function. If the curves are unbounded, then the maximum complexity of their lower envelope is $\Theta(\lambda_s(n))$.

Three-dimensional envelopes

The maximum combinatorial complexity of the lower envelope of n triangles in $\mathbb{R}^3$ is $\Theta(n^2\alpha(n))$ [58]. If the triangles are pairwise disjoint the maximum complexity is $\Theta(n^2)$. Let $\mathcal{S}$ be a set of n surface patches in $\mathbb{R}^3$ which satisfy the following assumptions:

1. Each surface patch is contained in an algebraic surface of constant maximum degree.
2. The vertical projection of each surface patch onto the xy-plane is a planar region bounded by a constant number of algebraic arcs of constant maximum degree.
3. Every three surface patches meet in at most s points.
4. Every surface patch is monotone in x,y (from now on we call it xy-monotone), namely every line parallel to the z-axis intersects the surface patch in at most one point.

The combinatorial complexity of the lower envelope of $\mathcal{S}$ is $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$, where the constant of proportionality depends on ε and some surface-specific constants [39, 61].

d-dimensional envelopes

The maximum combinatorial complexity of the lower envelope of n hyperplanes in $\mathbb{R}^d$ is $\Theta(n^{\lfloor d/2 \rfloor})$ by the Upper Bound Theorem [50]. The maximum combinatorial complexity of the lower envelope of n $(d-1)$-simplices in $\mathbb{R}^d$ is $\Theta(n^{d-1}\alpha(n))$ [24]. The combinatorial complexity of the lower envelope of n surface patches in $\mathbb{R}^d$, which satisfy assumptions similar to the assumptions made on surface patches in $\mathbb{R}^3$, is $O(n^{d-1+\varepsilon})$, for any $\varepsilon > 0$, where the constant of proportionality depends on ε, d and some surface-specific constants [61].

2.2 Applications

We describe here several applications of lower envelopes. For more applications see [5, 62].

2.2.1 Hidden Surface Removal

In computer graphics one wants to compute the view of a collection of objects in three-dimensional space as seen from a certain viewpoint. The problem of determining which parts of each object are visible and which parts are hidden is called *hidden surface removal* [19, 33]. There are two types of algorithms for hidden surface removal: image space algorithms and object space algorithms. The former are algorithms which compute the image of the scene pixel by pixel, determining for each pixel the visible object there. Such algorithms are, for example, the z-buffer and the painter's algorithms. The latter type of algorithms compute a combinatorial representation of the viewing plane, called the visibility map. This is the subdivision of the viewing plane into maximal connected regions where a single object can be seen or no object is seen. The problem of computing the visibility map of objects in three-dimensional space is equivalent to the problem of computing

the minimization diagram of three-dimensional surfaces. Image space algorithms, such as the z-buffer algorithm, tend to be faster in practice, because they can be implemented in hardware. However, object space algorithms have some advantages over image space algorithms. For example, image space algorithms create a view for the scene for a specific resolution. When one wants to print the view of a scene on a paper instead of rendering it on the computer screen much higher resolution is needed, and processing the visibility map directly can result in higher quality pictures.

2.2.2 Voronoi Diagrams

Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be a set of n pairwise-disjoint convex objects in $\mathbb{R}^d$ and let ρ be a metric on $\mathbb{R}^d$. The *Voronoi diagram* of $\mathcal{S}$ with respect to the metric ρ is a partition of d-space into maximally connected cells, each of which consists of the points closer to one particular object than to any others. Every Voronoi diagram in $\mathbb{R}^d$ can be seen as the minimization diagram of surfaces in $\mathbb{R}^{d+1}$. Let $f_i : \mathbb{R}^d \longrightarrow \mathbb{R}$ be the function defined by $f_i(x) = \rho(x, s_i)$. The Voronoi cell of s_i is $\{p \in \mathbb{R}^d | f_i(p) \leq f_j(p), \forall j \neq i\}$. Thus, the Voronoi diagram of the objects $s_1, \ldots, s_n$ is exactly the minimization diagram of the graphs of the functions $f_1, \ldots, f_n$, namely the projection of their lower envelope, as was observed by Edelsbrunner and Seidel [25].

2.2.3 Hausdorff Distance

Let $\mathcal{A} = \{a_1, \ldots, a_u\}$ and $\mathcal{B} = \{b_1, \ldots, b_v\}$ be two point sets in $\mathbb{R}^d$, and let ρ be an L_p metric on $\mathbb{R}^d$, for some $1 \leq p \leq \infty$. The *Hausdorff distance* between $\mathcal{A}$ and $\mathcal{B}$ is defined as

$$\mathcal{H}(\mathcal{A}, \mathcal{B}) = \max\{h(\mathcal{A}, \mathcal{B}), h(\mathcal{B}, \mathcal{A})\},$$

where

$$h(\mathcal{A}, \mathcal{B}) = \max_{a \in \mathcal{A}} \min_{b \in \mathcal{B}} \rho(a, b).$$

The *minimum Hausdorff distance under translation* between $\mathcal{A}$ and $\mathcal{B}$ is defined to be

$$\mathcal{D}(\mathcal{A}, \mathcal{B}) = \min_{x} \mathcal{H}(\mathcal{A}, \mathcal{B} \oplus x),$$

where $\mathcal{B} \oplus x = \{b + x \mid b \in \mathcal{B}\}$. The minimum Hausdorff distance between two point sets under translation has been proposed as a measure of the degree to which the two sets resemble each other, and is thus a useful construct in pattern recognition. The value of x minimizing $\mathcal{D}$ gives the translation of $\mathcal{B}$ under which it most resembles $\mathcal{A}$.
The *Voronoi surface* of a set $\mathcal{S}$ of points in $\mathbb{R}^d$ is defined as $d(x) = \min_{q \in \mathcal{S}} \rho(q, x)$. We next explain how $\mathcal{D}(\mathcal{A}, \mathcal{B})$ can be viewed as the minimum point of the upper envelope of a set of Voronoi surfaces. For each point $a_i \in \mathcal{A}$, let $\mathcal{S}_i = \{a_i - b \mid b \in \mathcal{B}\}$, and for each point $b_i \in \mathcal{B}$, let $\mathcal{S}_{u+i} = \{a - b_i \mid a \in \mathcal{A}\}$. Let $d_i(x)$ denote the Voronoi surface of $\mathcal{S}_i$, for $i = 1, \ldots, u + v$, and $\mathcal{E}^*(x)$ denote the upper envelope of $d_1(x), \ldots, d_{u+v}(x)$. Then

$h(\mathcal{A}, \mathcal{B} \oplus x) = \max_{1 \le i \le u} d_i(x)$ and $h(\mathcal{B} \oplus x, \mathcal{A}) = \max_{u < i \le u+v} d_i(x)$. Hence

$$\mathcal{D}(\mathcal{A}, \mathcal{B}) = \min_x \mathcal{E}^*(x),$$

that is $\mathcal{D}(\mathcal{A}, \mathcal{B})$ is the lowest point on the upper envelope of $u + v$ Voronoi surfaces. For more details on Hausdorff distance and Voronoi surfaces see [62].

2.2.4 Plane Transversals in Three-Space

Let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be a collection of n compact convex sets in $\mathbb{R}^3$. A plane p is a transversal of $\mathcal{C}$ if it intersects every set in $\mathcal{C}$. In the dual space, where each non-vertical plane $z = ax + by + d$ is mapped to a point (a, b, d), and each point (u, v, w) is mapped to a plane $z = -ux - vy + w$, the space of all plane transversals of $\mathcal{C}$, denoted $T(\mathcal{C})$, is the region enclosed between a lower envelope and an upper envelope of two sets of functions. A plane $z = ax + by + d$ intersects a compact convex set C if and only if it is parallel to and lies between two parallel planes, one of which is tangent to C from below, and the other is tangent to C from above, that is $f_C(a, b) \le d \le g_C(a, b)$, where $f_C(a, b), g_C(a, b)$ are defined such that the plane $z = ax + by + f_C(a, b)$ ($z = ax + by + g_C(a, b)$) is tangent to C from below (above). Thus, in the dual space, $T(\mathcal{C})$ is

$$\{(a, b, d) \mid \max_{C \in \mathcal{C}} f_C(a, b) \le d \le \min_{C \in \mathcal{C}} g_C(a, b)\}.$$

That is, in the dual space, $T(\mathcal{C})$ is the region consisting of all points that lie below the lower envelope of the functions $g_{C_i}(a, b), i = 1, \ldots, n$ and above the upper envelope of the functions $f_{C_i}(a, b), i = 1, \ldots, n$, also known as the *sandwich region* of the two sets of functions.

2.3 Algorithms for Envelopes

We review a few algorithms for computing lower envelopes in three-dimensions. We tried to pick a variety of algorithms, some of which are more general and the others are more specifically tailored to special cases. Some of the algorithms were originally formulated in terms of a hidden surface removal problem, and compute the visibility map. We remind the reader that this is equivalent to the lower envelope problem (see Section 2.2.1).

We postpone the detailed description of the divide-and-conquer algorithm by Agarwal et al. [4] to Chapter 3. The running-time of this algorithm is $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$, which is based on their result that the combinatorial complexity of the overlay of two minimization diagrams of two collections of a total of n surface patches is also $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$.

2.3.1 A Randomized Incremental Algorithm

We review the randomized incremental algorithm presented by Boissonnat and Dobrindt [15]. The algorithm is an on-line algorithm, in which the expected cost of inserting the n-th surface patch is $O(\log n \sum_{r=1}^{n} \tau(r)/r^2)$, where $\tau(r)$ is the expected size of an intermediate result for r surface patches. Since $\tau(r)$ is bounded by $O(r^{2+\varepsilon})$, for any $\varepsilon > 0$, the insertion time of the n-th surface patch is $O(n^{1+\varepsilon})$, and the construction time of the lower envelope is $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$. In the case of triangles, $\tau(r)$ can be $\Theta(r^2\alpha(r))$ in the worst case, and the construction time of the lower envelope is bounded in the worst case by $O(n^2\alpha(n)\log n)$.

The algorithm applies to surface patches of fixed maximum algebraic degree, but for simplicity, we describe it here for a set $\mathcal{S}$ of n triangles in $\mathbb{R}^3$, which are not vertical. To simplify the description of the algorithm we assume that the plane $z = \infty$ is a triangle in $\mathcal{S}$ and is inserted first. The triangles are inserted one after the other, and the algorithm maintains the vertical decomposition of the current lower envelope as well as the adjacency relationships of the prisms (cylindrical cells) in the decomposition. The vertical decomposition of a three-dimensional arrangement of triangles is defined as follows. First vertical walls are created on every arrangement edge by extending vertical rays upwards and downwards from every point on the edge until they hit another arrangement feature, or extend to infinity. These walls are called primary walls. This process creates cylindrical cells which may have complicated shape. Then each floor of a cell is decomposed using a two-dimensional vertical decomposition, and the new edges are extended to three-dimensional walls inside the cell. These walls are called secondary walls. The vertical decomposition of the lower envelope is defined by the portion of the entire vertical decomposition that lies below the lower envelope. The algorithm uses an influence graph [16], which is a rooted, directed acyclic graph that allows to find efficiently the prisms (of the current lower envelope decomposition) that are intersected by a new triangle. Let t be a new triangle that is added to the current lower envelope $\mathcal{E}'$, and let $\mathcal{D}_{\mathcal{E}'}$ be the decomposition of $\mathcal{E}'$. Denote by $\mathcal{E}$ the lower envelope after inserting t, and by $\mathcal{D}_{\mathcal{E}}$ its decomposition. The insertion of t is performed by a location phase and an update phase:

Location Using the influence graph, all the prisms whose interior is intersected by t are located. If no such prism exists, t is not part of the lower envelope, and the algorithm can skip the update phase, and move on to the next triangle.

Update The algorithm first constructs all the prisms of $\mathcal{D}_{\mathcal{E}}$ whose ceiling is not supported by t. For each prism of $\mathcal{D}_{\mathcal{E}'}$ that was located in the location phase, the portion of it that appears on $\mathcal{E}$ is subdivided by t into a constant number of sub-prisms. Some of these sub-prisms may share a wall (which is a part of a secondary wall) that has to be removed, and the prisms have to be merged to obtain the decomposition of $\mathcal{E}$. The adjacency information and the influence graph can be easily updated.

It remains to compute the prisms of $\mathcal{D}_{\mathcal{E}}$ whose ceiling is supported by t. The union of these prisms is computed using the adjacency graph. This union is not connected in general, and each of its connected components corresponds to a face of the lower

envelope $\mathcal{E}$ that is supported by t. The decomposition of all these faces results in the prisms of $\mathcal{D}_{\mathcal{E}}$. Note that the removal of portions of primary walls may cause problems if these portions stopped some of the secondary walls, since these secondary walls will have to be extended. Thus, a prism of $\mathcal{D}_{\mathcal{E}'}$ can be divided into an unbounded number of parts, each corresponds to a different prism of $\mathcal{D}_{\mathcal{E}}$. This requires careful update of the influence graph, since the randomized analysis applies when the out degree of a node in the graph is bounded. To overcome this difficulty, the algorithm introduces a special node for each face of $\mathcal{E}$ that is supported by t. For a special node $\mathcal{F}$, all the relevant prisms of $\mathcal{D}_{\mathcal{E}'}$ are connected to it. This node is a root of a secondary influence graph, which is a result of applying a planar randomized incremental algorithm for trapezoidal decomposition on the face represented by $\mathcal{F}$.

2.3.2 A Quasi Output-Sensitive Algorithm

We review the randomized algorithm of Mulmuley [55] for computing the visibility map of non-intersecting polygons in $\mathbb{R}^3$. The expected running time of the algorithm is a sum of weights associated with all intersections of projected edges of the polygons, where the weight of an intersection decreases as the number of objects hiding it increases. This method is not output sensitive, but it is "quasi output sensitive", since the largest weight belongs to the intersection points which are not hidden by any object, namely the vertices of the visibility map. The algorithm can be extended to intersecting polygons [56].

Let $\mathcal{N}$ be a set of n non-intersecting polygons. Denote by $H(\mathcal{N})$ the trapezoidal decomposition of the visibility map of the polygons of $\mathcal{N}$. The algorithm constructs $H(\mathcal{N})$ incrementally by adding the polygons one at a time, in random order. For $1 \leq i \leq n$, let $\mathcal{N}^i$ denote the set of the first i polygons in this addition sequence. At the i-th step, the algorithm maintains $H(\mathcal{N}^i)$ and in addition:

- For each trapezoid $f \in H(\mathcal{N}^i)$, a list of polygons in $\mathcal{N} \setminus \mathcal{N}^i$ that are in conflict with f. A polygon $p \in \mathcal{N} \setminus \mathcal{N}^i$ is in conflict with f if the polygons in $\mathcal{N}^i$ do not hide p completely within f.
- For every polygon in $\mathcal{N} \setminus \mathcal{N}^i$, a list of trapezoids in conflict with it.

The addition of the $(i+1)$-st polygon, denoted by s, is carried out as follows:

1. All the trapezoids that are in conflict with s are considered. For such a trapezoid f, s can be seen over all f, or f can split by the projection of the boundary of s. In the latter case, the partition of f should be further refined into trapezoids. The relevant conflict lists are updated as necessary. Denote the resulting planar partition by $H_1(\mathcal{N}^{i+1})$.
2. Vertical edges of $H(\mathcal{N}^i)$ (which are part of the trapezoidal decomposition) may be intersected by the projected visible boundary of s, which was inserted to the structure in the previous step. Only one part of it is relevant to $H(\mathcal{N}^{i+1})$ and is retained; the other part is removed. Thus, several trapezoids may merge into one trapezoid (with

a unique visible polygon). The relevant conflict lists are merged accordingly. Denote the resulting planar partition by $H_2(\mathcal{N}^{i+1})$.

3. Denote by R the region formed by the union of all trapezoids of $H_2(\mathcal{N}^{i+1})$ labelled with s. In general, this region can have several components, each may have a complicated shape. R can contain edges that were visible in $H(\mathcal{N}^i)$ but are hidden by s, thus, not visible in $H(\mathcal{N}^{i+1})$. Such edges should be removed, and R should be redecomposed into trapezoids. This is done by applying the trapezoidal decomposition algorithm to the set of segments on the boundary of R. A point location structure for the trapezoidal decomposition of R is also obtained. This structure is used for building the conflict lists of the new trapezoids within R based on the conflict lists of the old trapezoids. The resulting partition is the desired partition $H(\mathcal{N}^{i+1})$.

2.3.3 Output-Sensitive Algorithms

Obtaining output sensitive algorithms that compute envelopes is a major challenge. Such algorithms exist for special cases only. We review two output-sensitive algorithms for computing envelopes in three-space. In the first algorithm a depth order on the input objects is assumed and the efficiency relies on certain property of the union of projected objects. The second algorithm is applicable to polyhedral objects only.

Efficient Hidden Surface Removal for Objects with Small Union Size

Let $\mathcal{S}$ be a set of n non-intersecting objects in $\mathbb{R}^3$, and assume that they are ordered by depth from the viewing point. We review the algorithm presented by Katz et al. [47] for computing the visibility map for $\mathcal{S}$. The algorithm runs in time $O((U(n)+k)\log^2 n)$, and uses $O(U(n)\log n)$ working storage, where k is the complexity of the output map and $U(n)$ is a super-additive bound on the maximal complexity of the union of the projections on the viewing plane of any n objects. The efficiency of the algorithm shows up when $U(n)$ is small (that is subquadratic), for example:

- For spheres (or disks) one has $U(n) = O(n)$, and the technique yields an algorithm to compute the visibility map in time $O((n+k)\log^2 n)$.
- For horizontal "fat" triangles, where each internal angle is at least some fixed θ, one has $U(n) = O(n \log\log n)$, and the algorithm runs in time $O((n\log\log n + k)\log^2 n)$.

As a first step, the method sorts the objects by depth order and stores them in the leafs of a balanced binary tree $\mathcal{T}$, the nearest object in the leftmost leaf. Note that the objects stored in the subtree rooted at node δ can be hidden only by objects that are stored in the tree to the left of δ. The algorithm performs two traversals on the tree, to compute for each node δ two planar maps:

- U_δ — the union of the projections of the objects in the subtree $\mathcal{T}_\delta$ of $\mathcal{T}$ rooted at δ.

- V_δ — the subset of U_δ consisting of the projections of the visible portions (with respect to the whole scene) of the objects in the subtree $\mathcal{T}_\delta$, i.e., the portions of the objects in the subtree $\mathcal{T}_\delta$ not hidden by any nearer object (stored in $\mathcal{T}$ to the left of δ).

In the first traversal of the tree U_δ is computed for all nodes δ of $\mathcal{T}$. This traversal is done in bottom-up manner, starting from the leaves, and going up level by level until reaching the root. For a leaf δ, U_δ is simply the projection of the object stored in that leaf. For a non-leaf node δ, $U_\delta = U_{\mathrm{lson}(\delta)} \cup U_{\mathrm{rson}(\delta)}$, where $\mathrm{lson}(\delta)$ and $\mathrm{rson}(\delta)$ denote the left and right children of δ respectively.

In the second traversal of $\mathcal{T}$ V_δ is computed for all nodes δ of $\mathcal{T}$. This traversal is done in a top-down manner, starting at the root and working the way down the tree. For the root of $\mathcal{T}$, $V_{\mathrm{root}} = U_{\mathrm{root}}$, because for every point inside the union of the objects there exists an object which is visible there. For a node δ other than the root, V_δ is computed using the following equations:

- $V_{\mathrm{lson}(\delta)} = V_\delta \cap U_{\mathrm{lson}(\delta)}$, which is the visible part of δ where the objects of $\mathrm{lson}(\delta)$ are defined.

- $V_{\mathrm{rson}(\delta)} = V_\delta \setminus U_{\mathrm{lson}(\delta)}$, which is the portion of V_δ not hidden by other objects stored to the left of $\mathrm{rson}(\delta)$.

For each leaf δ, V_δ consists precisely of those parts of the object stored in this leaf that are visible. So the final step of the algorithm is to properly glue all these regions in the leaves to a whole visibility map.

In order to achieve the cited working storage bound, one should be careful in the second traversal, performing it in preorder, and deleting unnecessary maps V_δ after the completion of the computation of such a map for δ and its two sons.

Efficient Ray Shooting and Hidden Surface Removal

We review the algorithm presented by de Berg et al. [20]. We describe the algorithm for a set of non-intersecting triangles in space, but it can be extended to any set of non-intersecting polyhedra. Let $\mathcal{S}$ be a set of non-intersecting triangles with n edges in total. The running time of the algorithm is $O(n^{2/3+\varepsilon}k^{2/3} + n^{1+\varepsilon})$ for any $\varepsilon > 0$, where k is the size of the result visibility map, when using a data structure for ray shooting among curtains presented in [3]. In [19] the idea is extended to a set of possibly intersecting polyhedra with n vertices in total, and the algorithm runs in time $O(n^{4/3+\varepsilon} + n^{4/5+\varepsilon}k^{4/5})$ and uses $O(n^{4/3+\varepsilon} + n^{4/5+\varepsilon}k^{4/5})$ storage.

Denote by $\mathcal{M}$ the visibility map of the input. The algorithm moves a horizontal sweep-line from top to bottom over the viewing plane, discovering $\mathcal{M}$ "on the fly", as it advances. The event queue $\mathcal{Q}$ stores event points in order of decreasing y-coordinate. It is initialized with all the projections of the triangles vertices. When a new vertex of $\mathcal{M}$ is discovered, it is inserted into $\mathcal{Q}$. While sweeping, the algorithm keeps track of the edges of $\mathcal{M}$ that

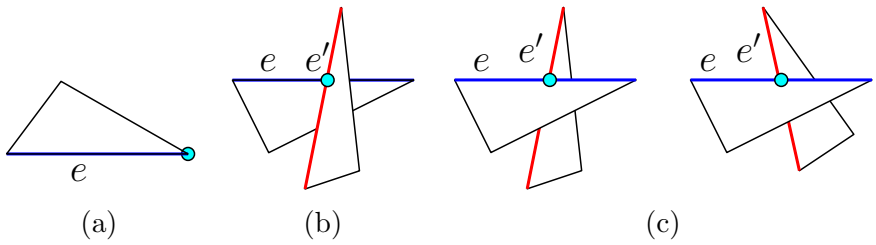


Figure 2.1: The possible cases for the location of the other endpoint of an edge in the visibility map: (a) the projection of a vertex of e, (b) the intersection of the projection of e with the projection of e', where e' is below e, and (c) the intersection of the projection of e with the projection of e', where e' is above e, and e' becomes visible (left) or invisible (right) at the intersection point.

are intersected by the sweep line. These edges are stored in a binary search tree $\mathcal{T}$ in the order of their intersection with the sweep line. For each edge, the triangle that is visible to its left is also stored.

The handling of an event point v is carried out in the following way. If v is the projection of a vertex of a triangle, the algorithm checks if it is visible. This is done by searching $\mathcal{T}$ to find the edge of $\mathcal{M}$ to the right of v, together with the triangle that is visible to the left of this edge. It is now possible to check whether v is visible by comparing it to that triangle. If v is not the projection of a vertex of a triangle, it is a vertex of $\mathcal{M}$, which means that it is visible. If v is not visible, the handling of the current event is finished.

If v is visible, it is a vertex of the result visibility map, and $\mathcal{Q}$ and $\mathcal{T}$ should be updated. This means that the algorithm should compute the other endpoints of those edges of the visibility map that are incident to v and will be intersected by the sweep-line when it is advanced. It is possible to ensure that the edges of the triangles that correspond to the edges of the visibility map are always known. Let a be an edge of the visibility map that is incident to v and let e be the corresponding triangle edge. The other endpoint w of a is either the projection of a vertex of e or the intersection of the projection of e with the projection of other edge e'. In the latter case, either e' is below e, in which case e becomes invisible at the intersection point (because it is hidden by the triangles incident to e'), or e' is above e, in which case e' becomes invisible/visible at the intersection point. Figure 2.1 shows an illustration of all possible cases where the other endpoint can be. In order to find e' in both cases, a ray ρ is defined in the following way. Let p be the point on e whose projection is v. Let f be the triangle that is immediately above p, and let ρ be the projection onto f of the ray starting at p along e (if there is no such triangle f, ρ is defined to be the projection of this ray onto a plane which is above all the triangles in the scene). Then w, the other node of a, is either the projection of a vertex of e or it is the intersection of the projection of e with the projection of the first edge passing below ρ. To find ρ, a data structure for ray shooting among the triangles in a fixed direction is used. To find the first edge passing below ρ a data structure for ray shooting among curtains is used: each triangle edge defines an (upside-down) curtain, which is the set of

points hidden by this edge; the desired edge corresponds to the first curtain hit by ρ.

A tradeoff between preprocessing time and query time of the ray shooting data structures is needed to accomplish a good running time. The structure for ray shooting in a fixed direction has a better performance than the structure for ray shooting among curtains, which thus dominates the performance of the algorithm. In [3], a data structure is presented for the ray shooting problem among curtains in $\mathbb{R}^3$ with $O(m^{1+\varepsilon})$ space and preprocessing time, and $O(\frac{n}{\sqrt{m}}\log^{O(1)} n)$ query time, for a parameter m, $n \leq m \leq n^2$. Choosing $m = n^{2/3}k^{2/3}$, and plugging the data structure into the algorithm presented above, yields a running time of $O(n^{2/3+\varepsilon}k^{2/3} + n^{1+\varepsilon})$, where k is the output size. Note that k is not known in advance, therefore an initial value of m should be guessed and updated as the algorithm proceeds. This is a standard trick, which in the algorithm above does not affect the asymptotic running-time.

2.4 Implementation

2.4.1 The Gap Between Theory and Practice

Transforming a geometric algorithm into a computer program is not a simple task. An algorithm implemented from a textbook is susceptible to robustness issues, and thus may yield incorrect results, enter infinite loops or crash (see, for example, [48, 60]). This is mainly due to two assumptions that are often made in the theoretical study of geometric algorithms, which are not realistic in practice. First, the general position assumption excludes all degenerate input. This assumption facilitates the description and analysis of algorithms significantly. However, one cannot be sure that real inputs will always be in general position (typically the contrary is true). Second, the real RAM model [59] is assumed, which allows for infinite precision arithmetic operations on real numbers. Moreover, every operation on a constant number of simple geometric objects is assumed to take constant time. This is of course not true in computer programs that use finite precision numbers. When using finite precision arithmetic, round-off errors may cause incorrect results for geometric operations and lead to unstability of the algorithm. Round-off errors are mostly problematic when the input is degenerate or near-degenerate.

2.4.2 Exact Geometric Computation

One approach to dealing with robustness problems is exact geometric computation [71]. The goal of exact geometric computation is determining geometric relations exactly. The geometric relations determine the combinatorial (or topological) part of a geometric structure, which is often defined by the numerical part. Some geometric problems need only deal with integer or rational numbers. This is the case, for example, when dealing only with linear objects that are represented with integer or rational numbers. Special number types were developed; unbounded integer number-types with exact $\{+, -, \times\}$ operators and exact comparisons on them, and exact rational types with exact $\{+, -, \times, \div\}$ op-

erators. Such number types are supported by several libraries, for example, GMP (Gnu multi-precision),[1] CORE[2] [46] and LEDA[3] [51, Chapter 4]. However, rational arithmetic does not suffice when dealing with non-linear geometric objects, where irrational numbers are involved. Among the irrational numbers, the class of algebraic numbers, which can be used when dealing with algebraic curves and surfaces, was mainly studied. The LEDA and CORE libraries supply exact algebraic number-types that support the exact operations $\{+, -, \times, \div, \sqrt{\cdot}\}$ and even $\sqrt[k]{\cdot}$ and *root-of* (to find roots of polynomials). The main disadvantage of using exact number-types is the significant running time overhead that they may incur.

2.4.3 The CGAL Library

CGAL — the Computational Geometry Algorithms Library[4] — is a product of a collaborative effort of several sites in Europe and Israel aiming to provide a robust, generic and efficient implementation of computational geometry data structures and algorithms. It is a software library written in C++ following the generic programming paradigm (see Section 2.4.6). The library consists of three major parts: (i) the *kernel* [27, 44], which consists of constant-size non-modifiable geometric primitive objects, such as points, segments, and lines, and operations on these objects, (ii) the *basic library*, which contains a large collection of basic geometric data structures and algorithms, for example convex hull, triangulations and two-dimensional arrangements, and (iii) the *support library*, which consists of non-geometric support facilities, such as support for number types, I/O and visualization.

2.4.4 Two-Dimensional Arrangements in CGAL

Given a set $\mathcal{C}$ of planar curves, the arrangement $\mathcal{A}(\mathcal{C})$ is the subdivision of the plane into zero-dimensional, one-dimensional and two-dimensional cells, called vertices, edges and faces, respectively induced by the curves in $\mathcal{C}$. Arrangements are ubiquitous in computational geometry and have many applications; see, for example [5, 37]. CGAL provides a robust implementation for constructing planar subdivisions of arbitrary bounded curves and supporting operations and queries on them [30, 41, 42, 66, 67]. This implementation follows the exact geometric computation approach and has already been used in several applications [2, 8, 18, 22, 29, 31, 45, 69].

The CGAL `Arrangement_2` package,[5] which is part of CGAL's basic library, represents planar arrangements of bounded curves and provides the interface needed to construct them, traverse them, maintain them and perform point-location queries on them. Robustness in this package is achieved both by handling all degenerate cases, and by using

[1] Gnu's multi-precision library `http://www.swox.com/gmp/`.
[2] `http://www.cs.nyu.edu/exact/core_pages/intro.html`.
[3] `http://www.algorithmic-solutions.com/enleda.htm`.
[4] See the CGAL project homepage: `http://www.cgal.org/`.
[5] We describe the `Arrangement_2` package of CGAL version 3.2.

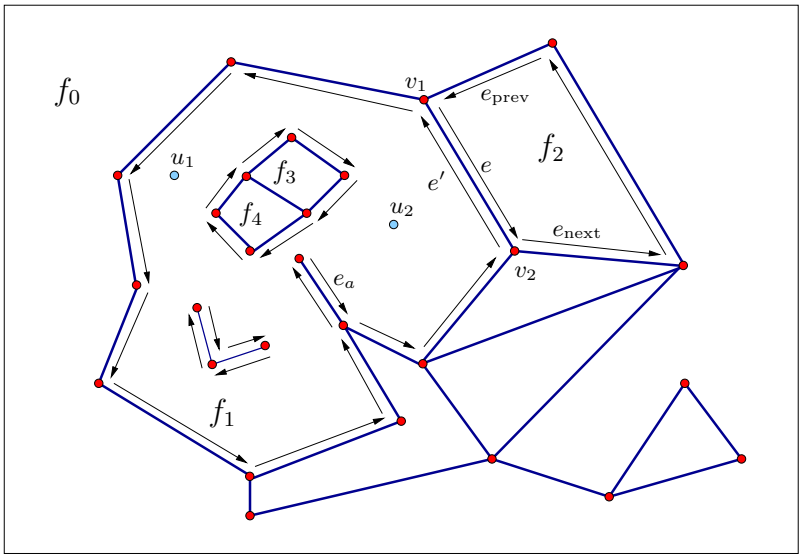


Figure 2.2: An arrangement of line segments with some of the DCEL records that represent it. The unbounded face f_0 has a single connected component that forms a hole inside it, and this hole is comprised of several faces. The halfedge e is directed from its source vertex v_1 to its target vertex v_2. This halfedge, together with its twin e', correspond to a line segment that connects the points associated with v_1 and v_2 and separates the face f_1 from f_2. The predecessor e_{prev} and successor e_{next} of e are part of the chain that forms the outer boundary of the face f_2. The face f_1 has a more complicated structure as it contains two holes in its interior: one hole consists of two adjacent faces f_3 and f_4, while the other hole is comprised of two edges. f_1 also contains two isolated vertices u_1 and u_2 in its interior. The outer boundary of f_1 includes a special edge e_a with both twin halfedges pointing to the same face f_1, one is successor of the other in the boundary chain. We call such an edge an "antenna". The figure has been adapted from the CGAL manual [1].

exact number types. The `Arrangement_2 <Traits, Dcel>` class is the main class in this package. The design of this class is guided by the need to separate the topological and the geometric aspects of the planar subdivision. This separation is realized by the two template parameters of the class:

- DCEL - a doubly-connected edge list (DCEL for short) data structure which represents the topological structure of the planar subdivision. The DCEL maintains the incidence relationships between the vertices, edges and faces of the arrangement.
- `Traits` - the geometric traits which determine the family of planar curves that form the arrangements. The traits class defines the types of two-dimensional points and x-monotone curves, and supports basic geometric predicates on them.

In a DCEL data structure every edge is represented by a pair of directed *halfedges*, called *twin* halfedges, one directed from the left endpoint of the curve to the right endpoint of

the curve, and the other is directed in the opposite direction. Since a halfedge is directed, it is possible to define its *source* vertex and its *target* vertex. Halfedges are used to connect vertices and separate faces. Each halfedge contains a pointer to its target vertex as well as a pointer to the face that lies to its left, which is called its *incident* face. All the halfedges that are incident to the same face are connected in circular lists, one that forms the outer boundary of the face (except for the unbounded face which does not have an outer boundary), and possibly other circular lists that form inner boundaries, called *holes*, of the face. In order to traverse these circular lists, every halfedge stores pointers to the next and to the previous halfedges in the list. Each face of the arrangement stores one halfedge of its outer boundary (if the face is bounded). In addition, each face stores a list, which may be empty, of holes, each represented by one halfedge on the hole, and a list of isolated vertices that lie inside the face, which may also be empty. A hole inside a face does not necessarily correspond to a single face — it may have no area, or even be composed of several connected faces. Figure 2.2 shows an example of a DCEL. For further details and examples regarding a DCEL data structure see [21, Chapter 2].

The package includes several traits classes that handle linear curves as well as non-linear curves. The supplied traits classes are: traits classes for line segments, a traits class that operates on continuous piecewise linear curves, namely polylines, a traits class for circular arcs, a traits class for general conic arcs, which are bounded segments of algebraic curves of degree two, and a traits class for arcs of graphs of rational functions. Other traits classes for the `Arrangement_2` package, not included in CGAL are also available. For example, traits classes for conic curves [11], cubic curves [26] and a special kind of quartic curves [12] were developed as part of the EXACUS project.

The `Arrangement_2` package provides implementation for two fundamental algorithmic procedures, which are common to many applications: the *sweep-line* algorithm, and the *zone*[6] computation. Specific algorithms based on these two algorithmic frameworks can be implemented using visitor classes. *Visitor* is a design-pattern which represents an operation to be performed on an object or on the elements of an object structure. Visitors allow the definition of new operations without changing the classes of the elements on which they operate [35]. The visitor classes of the sweep-line and zone algorithms receive notifications of the events handled by the basic procedure and can construct their output structures accordingly. The package supplies several sweep-line visitors, for example, a visitor for computing all intersection points induced by a set of curves, a visitor for constructing the arrangements of these curves, and a visitor for inserting the curves into an existing arrangement. The overlay of two arrangements is also implemented as a sweep-line visitor class. The package supplies a visitor to the zone-computation algorithm as well. This visitor is used in the incremental insertion of an x-monotone curve into an arrangement. For further information on the sweep-line and zone frameworks and their visitors we refer the reader to [67].

The `Arrangement_2` package is extensively used in our work. In the following chapters

[6] Given an arrangement $\mathcal{A}$ and a curve γ, the *zone* of γ in $\mathcal{A}$ is the set of all arrangement cells of $\mathcal{A}$ that γ crosses.

we give further details on it, as needed.

2.4.5 Two-Dimensional Envelopes

A divide-and-conquer algorithm which constructs the lower (or upper) envelope of planar curves was implemented [68]. As mentioned in Section 2.1, the complexity of the lower envelope of n "well behaved" curves that intersect in at most s points, for some fixed constant s, is $\Theta(\lambda_{s+2}(n))$. The divide-and-conquer algorithm runs in time $O(\lambda_{s+2}(n)\log(n))$.

The implementation is an extension to the arrangement package of CGAL (version 3.1), and has the CGAL look-and-feel. In the implementation, the topological structure of the envelope is separated from its geometry, thus allowing users to works with any type of planar curves, provided that they supply some geometric objects and predicates on them. This implementation is also robust, handling all possible degeneracies.

2.4.6 Generic Programming

The generic programming paradigm [7] aims at generalizing software components in order to make them easily reusable in a wide variety of situations. The generic software is designed with *concepts*, which are abstract sets of requirements on data types. A type that satisfies all the requirements of a concept is called a *model* of that concept. A concept that extends the set of requirements of another concept is a *refinement* of the latter. When designing generic software components, one of the main activities is the concept development — identifying the sets of requirements that are general enough to be met in many situations, and still restrictive enough to write programs that work efficiently with all models of the concept.

In C++, class and function templates are particularly effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency. Concepts correspond to template parameters, and models correspond to classes used to instantiate them. The C++ Standard Template Library (STL) and the CGAL library make extensive use of the generic programming paradigm.

Chapter 3

Overview of Our Solution

We devised an exact and generic implementation of the divide-and-conquer algorithm for constructing the envelope of surface patches in $\mathbb{R}^3$. Our implementation is complete in the sense that it handles all degenerate cases, and at the same time it is efficient. We opted for the divide-and-conquer algorithm as the first exact solution for three-dimensional envelopes for several reasons, the major ones being: (i) it is a very intuitive approach, (ii) there are good software tools for dealing with two-dimensional arrangements and operations on them, including the overlay operation — which is a central ingredient in this approach, available for us, and (iii) it applies to many types of surfaces. Our implementation is based on the CGAL library, has the CGAL look-and-feel and is designated as a CGAL package.

In the rest of this thesis, to simplify the exposition, we refer to lower envelopes. However, our code is generic and capable of computing envelopes in any direction, including upper envelopes.

3.1 The Divide-and-Conquer Algorithm

We are given as input a set $\mathcal{F}$ of n bounded surface patches[1] in $\mathbb{R}^3$. The first step is to extract all the xy-monotone[2] portions of these surfaces that are relevant to the envelope. We denote this set by $\mathcal{G}$. Henceforth, we only work on these xy-monotone surfaces in $\mathcal{G}$. The output of our program is a minimization diagram, represented as a planar arrangement where each arrangement feature (vertex, edge or face) is labelled with the set of xy-monotone surfaces that attain the minimum over that feature. The label can contain a single surface, several surfaces, or no surface at all, in which case we call it the no surface label.

When $\mathcal{G}$ consists of a single xy-monotone surface, we construct its minimization diagram using the projection of its boundary: we insert the projection of its boundary into an

[1] We restrict ourselves to working with bounded surface patches only since the current implementation of the CGAL arrangement package that we use to represent the minimization diagram supports only bounded curves. An extension of the CGAL arrangement package to handle infinite curves is underway.

[2] A surface is xy-monotone if every line parallel to the z-axis intersects it in at most one point.

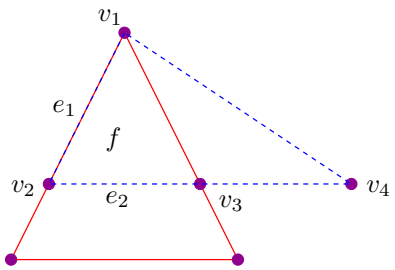


Figure 3.1: The different combinations for sub-features of a feature in the overlay of two arrangements $\mathcal{A}_1$ and $\mathcal{A}_2$ (up to a symmetry between the overlaid arrangements $\mathcal{A}_1$ and $\mathcal{A}_2$). Face f is a sub-face in each of the overlaid arrangements. Edge e_1 is a result of two overlapping edges in $\mathcal{A}_1$ and $\mathcal{A}_2$. Edge e_2 is a sub-edge in one arrangement, but a part of a face in the other arrangement. Vertex v_1 is the result of two coincident vertices, one from $\mathcal{A}_1$ and one from $\mathcal{A}_2$. Vertex v_2 results from a vertex in one arrangement splitting an edge of the other arrangement. Vertex v_3 is the intersection of two edges from $\mathcal{A}_1$ and $\mathcal{A}_2$. Vertex v_4 is a vertex of one arrangement lying inside a face of the other arrangement.

arrangement, then we label all the holes in the unbounded face and all the vertices and edges with this surface. Finally, we label the other faces, mainly the unbounded face, with the special label no surface.

When $\mathcal{G}$ contains more than one xy-monotone surface, we split $\mathcal{G}$ into two sets $\mathcal{G}_1$ and $\mathcal{G}_2$ of (roughly) equal size, recursively construct the minimization diagrams $\mathcal{M}_1$ and $\mathcal{M}_2$ of these sets respectively, and finally merge these two diagrams into the final minimization diagram $\mathcal{M}$. The merge step is carried out as follows:

- We overlay the two planar arrangements underlying the minimization diagrams $\mathcal{M}_1$ and $\mathcal{M}_2$ to obtain the arrangement $\mathcal{O}$, where each feature is a maximal connected portion of the intersection of one feature of $\mathcal{M}_1$ and one feature of $\mathcal{M}_2$. For example, a face in $\mathcal{O}$ can be a part of one face in $\mathcal{M}_1$ and one face in $\mathcal{M}_2$, an edge can be a part of an edge in $\mathcal{M}_1$ and a part of a face in $\mathcal{M}_2$, and so on. For each feature in $\mathcal{O}$ we keep two pointers to these features in $\mathcal{M}_1$ and $\mathcal{M}_2$. Figure 3.1 shows all the possible feature-feature combinations.

- We determine the structure of the minimization diagram over each feature in $\mathcal{O}$, to obtain the arrangement $\mathcal{O}'$, which is a refinement of the arrangement underlying $\mathcal{M}$. We then label each feature of $\mathcal{O}'$ with the correct envelope surfaces. Note that this is not a trivial step. We should consider here the two relevant features in $\mathcal{M}_1$ and $\mathcal{M}_2$ and their labels l_1 and l_2, respectively. If both labels are the no surface label, there is no surface defined over the current feature, and it should also be labelled by no surface. If only one of these labels is no surface, and the other represents a non-empty set of xy-monotone surfaces, we label the current feature with the latter, and we are done. When both labels represent non-empty sets of xy-monotone surfaces, these surfaces are defined over the entire current feature f, and their envelope over f is the

envelope of $\mathcal{G}_1 \cup \mathcal{G}_2$ there. Since all the xy-monotone surfaces of one label l_i overlap over the current feature f, it is possible to take only one representative surface s_i from each label and find the shape of their minimization diagram over f. Here we should consider the intersection between the representative surfaces s_1 and s_2, or more precisely, the projection onto the xy-plane of this intersection, denoted by $\mathcal{C}$, which may split the current feature, if it is an edge or a face; see Chapters 4 and 5 for more details. We then label all the features of the arrangement of $\mathcal{C}$ restricted to f (where f is considered relatively open) with the correct label, which might be either one of the labels l_1 and l_2 or $l_1 \cup l_2$ in case of an overlap.[3] A face of the overlay handled in this step, and hence the arrangement restricted to the face, can be very complicated, with arbitrary topology (including holes, isolated points and "antennas") and unbounded complexity. By preforming a *vertical decomposition* on $\mathcal{O}$, it is possible to get simpler faces. A vertical decomposition is a subdivision of an arrangement where a vertical ray is extended upwards and downwards from every vertex of the arrangement, until it hits another feature (vertex or edge) of the arrangement, or extends to infinity. A vertical decomposition creates simple faces, with constant number of edges on their boundary, and without holes, and at the same time preserves the asymptotic combinatorial complexity of the arrangement. Currently, we find dealing with the complicated faces directly preferable. The issue of vertical decomposition is addressed in Sections 5.4.2 and 7.7.

- We finally apply a cleanup step in order to remove redundant features of $\mathcal{O}'$ and obtain the minimization diagram $\mathcal{M}$ of $\mathcal{G}$. Edges with the same label as their two incident faces are redundant, and hence should not be part of the final minimization diagram. We remove such edges and merge the two faces into one face. Also vertices with degree two, which are labelled with the same surfaces as their two incident edges might be redundant. We remove those vertices and merge their edges into one edge if it is geometrically possible to do so, namely the curves represented by the edges originate from a single projected curve (either the projection of the boundary of an input surface or the projection of the intersection of two input surfaces).

3.2 Separation of Geometry and Topology

Our algorithm is parameterized with a *traits* class. The term traits was coined by Myers [57] for a concept of a class that should support certain predefined methods, passed as a parameter to another class template. In our case, the traits class encapsulates the geometric objects the algorithm operates on, and the predicates and constructions on these objects used by the algorithm. It serves as the geometric interface to the algorithm. The algorithm treats the geometric objects in an abstract manner, using only the operations defined by the traits class. In this manner the algorithm is made generic and independent

[3] With a slight abuse of notation we use the label l_i to denote the corresponding set of surfaces.

of the specific geometry needed to handle a special type of surfaces. By plugging in the application-specific geometry, the code for the algorithm can be reused.

The set of requirements from a traits class forms a concept.[4] The geometric-traits concept for the envelope algorithm, `EnvelopeTraits_3`, refines the `ArrangementTraits_2` concept for building a two-dimensional arrangement of general bounded curves. The latter concept, which is described in detail in [66, 67], defines three object types: planar points, x-monotone curves and general curves, and operations on them. The `EnvelopeTraits_3` concept adds to these requirements two more object types: three-dimensional xy-monotone surfaces and general surfaces, and also operations on them. For a detailed explanation of the requirements of these concepts see Chapter 4.

A key motivation behind this separation is to allow users to construct envelopes of their own types of surfaces without requiring any knowledge of the underlying computational-geometry algorithm and its intricacies. The traits that users have to supply rely on primitive algebraic/numeric operations applied to a small number of geometric objects.

3.3 The use of Two-Dimensional Arrangements

The problem of computing the envelope is somewhat two-and-a-half dimensional, since the input is three-dimensional, but the output is naturally represented as a two-dimensional object, the minimization diagram. In our solution we work as much as we can in the plane, as it is usually easier, and we can use available software. For the representation of the minimization diagram we use the CGAL `Arrangement_2` class with additional information on every feature — a list of xy-monotone surfaces which attain the envelope over this feature. The CGAL `Arrangement_2` package gives us all we need to represent the minimization diagram of bounded surfaces, including holes and isolated vertices, to access this representation and to modify it. For example, we can insert new points and curves into an arrangement after it was built, remove, split or merge edges, and so on. The `Arrangement_2` package also provides us with a method to overlay two arrangements and get the result as another arrangement. We use all these methods (and more), but are still left with a lot of work. Recall that after overlaying two minimization diagrams in the merge step, we have to determine the structure of the minimization diagram over each feature. If we work on a face, for example, and have to find the envelope of two xy-monotone surfaces over it, we find their projected intersection, and want to partition the face with it. But this projected intersection can cut many other irrelevant features of the overlaid arrangement. This may increase both the asymptotic complexity and the amount of geometric computation of the algorithm, so we want to refrain from naïve insertion of the projected intersection curves (see Section 5.1 for more details). Moreover, we have to correctly label all the sub-features, which are created after the insertion of the projected intersection curves, with the envelope surfaces. We can, of course, naïvely compare the surfaces over each feature. But we can be more careful, exploiting information from the traits and from the continuity (or discontinuity) of the envelopes that are merged, thus,

[4] Recall from Section 2.4.6 that a concept is an important component in generic programming.

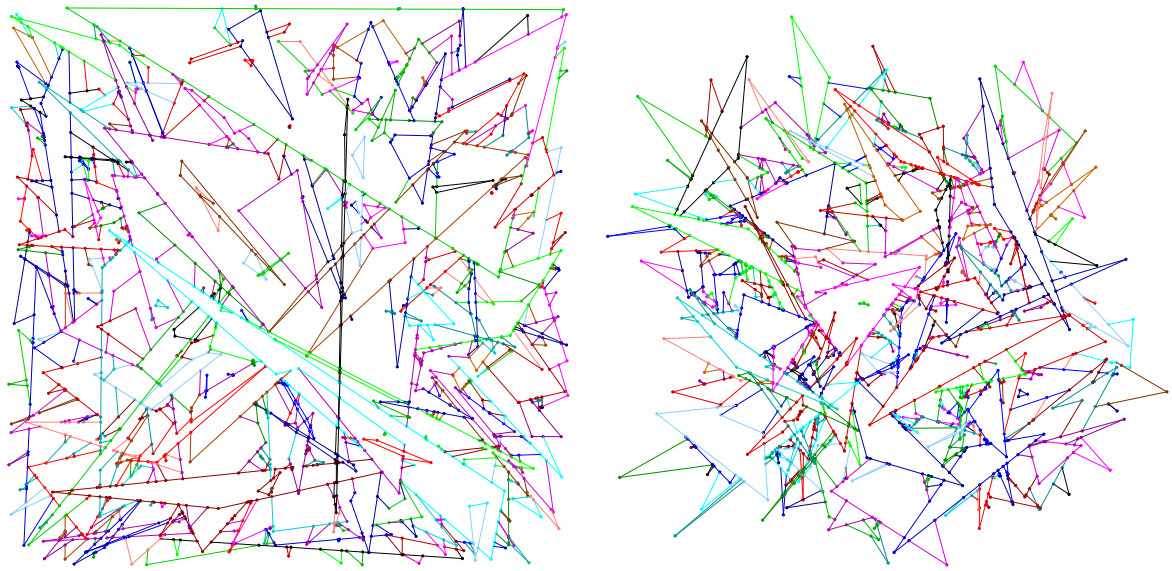


Figure 3.2: Lower envelopes of 300 triangles computed by our software.

reducing the number of needed comparisons significantly and greatly saving in algebraic computation. Section 5.2 describes the labelling step in detail.

The `Arrangement_2` package provides many public interface methods intended for the users of the package. Versions of these methods that are intended for internal use inside the package are available as well for most tasks. The internal version usually has more restrictions on its input than the public version of the same operation, and is more susceptible to mistakes regarding the input it receives, which may cause invalid representation of the arrangement structure. However, the absence of these restrictions on the input usually implies that geometric operations will be called to find geometric relations. Thus, the internal version of an operation is more efficient than its public version. In our algorithm we use the internal version of the arrangement operations whenever the needed geometric information is available, and by this avoid unnecessary geometric operations and gain efficiency.

3.4 Available Traits Classes

Recall from Section 2.4.6 that in generic programming, a type that satisfies all the requirements of a concept is a model of that concept. We implemented three models of the `EnvelopeTraits_3` concept, which can be plugged into our generic algorithm implementation. Our traits classes deal with triangles, spheres or quadrics. See Figures 3.2, 3.3 and 3.4 for examples of envelopes of these types, produced by our implementation.

- Triangles traits class. This traits class extends the two-dimensional arrangement traits for segments, since the minimization diagram is built from planar segments.

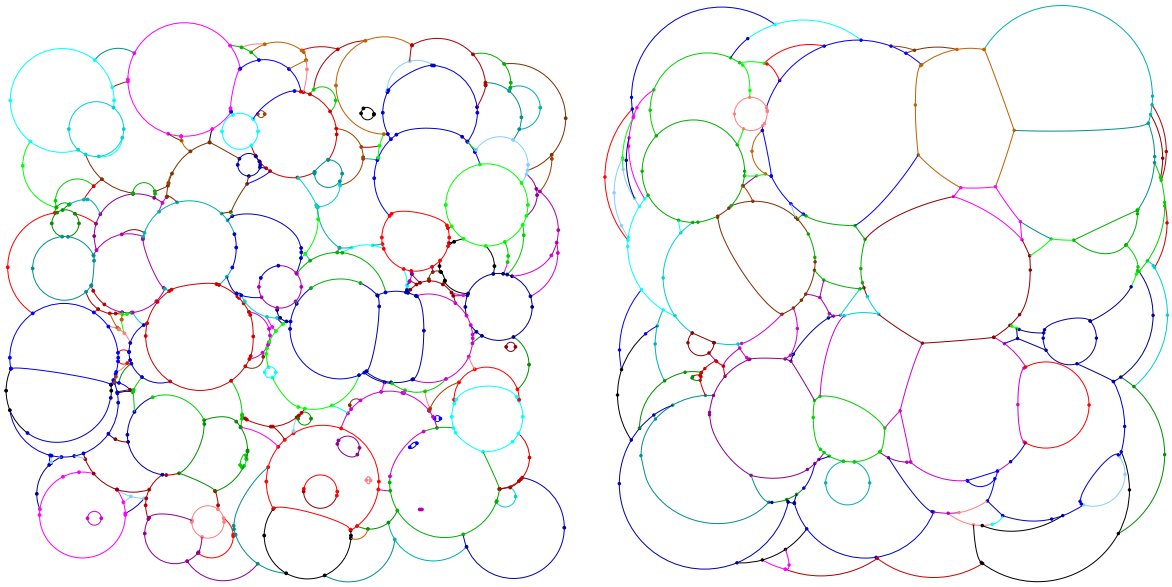
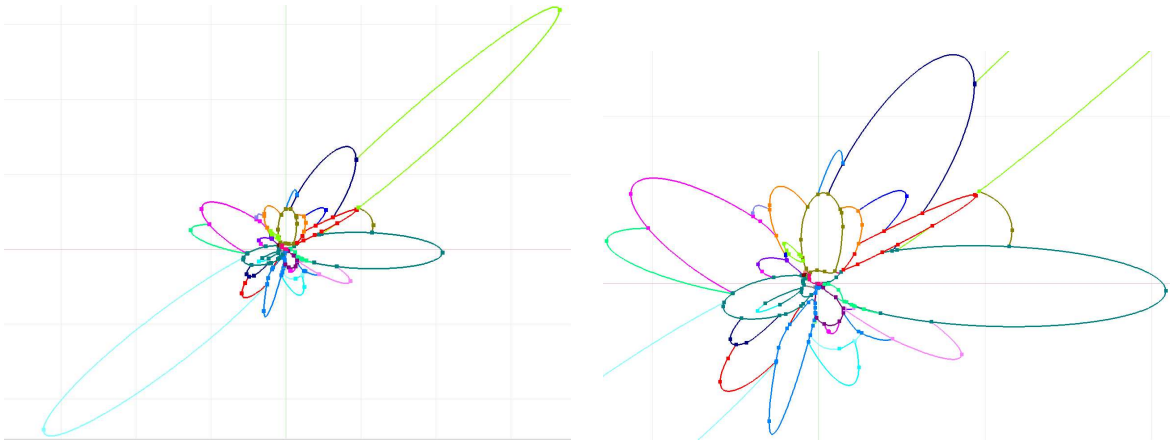


Figure 3.3: Lower envelopes of 300 spheres computed by our software.



(a) (b)

Figure 3.4: Lower envelope of 96 ellipsoids computed by our software (a), zoomed-in (b).

- Spheres traits class. The minimization diagram of spheres can contain segments of lines, circles and ellipses, since the intersection of two spheres can be empty, a point or a circle in three space. This traits class is an extension of the two-dimensional arrangement conics traits, which deals with segments of algebraic curves of degree two [64, 65].

- Quadrics traits class. Quadrics are algebraic surfaces of degree at most two. The minimization diagram of quadrics contains algebraic curves of degree at most four. EXACUS[5] [10] is an ongoing project aiming to provide efficient and exact algorithms for curves and surfaces. This traits class is based on the QuadriX package of EXACUS [9, 12], and is a joint effort with the authors of this package. Chapter 6 is dedicated to the quadrics traits.

[5]The EXACUS project homepage `http://www.mpi-sb.mpg.de/projects/EXACUS/`.

Chapter 4

Traits in Detail

Recall from Section 3.2, that our algorithm is parameterized with a traits class, which serves as the geometric interface to the algorithm. The traits class encapsulates the geometric objects the algorithm operates on, and the predicates and constructions on these objects used by the algorithm. Traits classes are used a lot in CGAL. For example, the convex-hull algorithm for planar points is parameterized with a traits class, which defines the type of points used and the needed predicates on them. Another example is the `Arrangement_2` class. This class is parameterized with a geometric-traits class, which defines the family of curves that are handled in the planar subdivision. In this chapter we explain the requirements of the divide-and-conquer algorithm for computing the envelope of surfaces from the traits class, and the reasons that these requirements are needed. We also describe some considerations in the design and implementation.

4.1 The EnvelopeTraits_3 Concept

As mentioned in Section 3.2, the geometric traits concept for the envelope algorithm, `EnvelopeTraits_3`, refines the `ArrangementTraits_2` concept for building planar arrangements of general bounded curves. We briefly review the `ArrangementTraits_2` concept requirements here. This concept defines three object types: planar points, x-monotone curves and general curves. It also defines the following operations on them:

1. Compare the x-coordinate of two points.
2. Compare two points lexicographically, first by their x-coordinates, then by their y-coordinates.
3. Return the left (similarly right) endpoint of an x-monotone curve.
4. Determine whether a weakly x-monotone curve is a vertical segment.
5. Given an x-monotone curve c and a point p that lies in its x-range, determine whether p lies below, above or on c.

6. Check two curves for equality.

7. Split an x-monotone curve c into two x-monotone sub-curves at a point p lying in the interior of c.

8. Given two x-monotone curves that share a common endpoint, determine whether they are mergeable, that is, can be merged into one continuous x-monotone curve of the type supported by the traits class.

9. Merge two mergeable x-monotone curves.

10. Compute all intersection points and overlapping sections of two x-monotone curves. If possible compute also the multiplicity of each intersection point.[1] Information about the multiplicity of an intersection point can be used to speed up the construction of a two-dimensional arrangement [11].

11. Given two x-monotone curves c_1 and c_2 that share a common left endpoint (similarly, right endpoint) p, determine whether c_1 lies above or below c_2 immediately to the right (to the left) of p, or whether the two curves coincide there.

12. Subdivide a general curve into maximal continuous x-monotone curves and isolated points.

The `EnvelopeTraits_3` concept adds to these requirements two more object types: three-dimensional xy-monotone surfaces and general surfaces. In practice, these two types might map to the same object type, or to two different object types. The concept defines the following operations on these types:

1. Given a general surface, extract maximal continuous xy-monotone patches of the surface which contribute to its envelope.

2. Construct all the planar curves that form the boundary of the vertical projection of a given xy-monotone surface onto the xy-plane.

   This operation is used at the bottom of the recursion to build the minimization diagram of a single xy-monotone surface.

3. Construct all the planar curves and points, which compose the projection (onto the xy-plane) of the intersection between two xy-monotone surfaces s_1 and s_2. If possible, indicate, for each projected intersection curve, whether the *envelope order*[2] of s_1 and s_2 changes when crossing that curve, or not. The envelope order of s_1 and s_2 indicates whether s_1 is below/coincides with/is above s_2. This information (referred to as the intersection type information) is optional — when provided, it is used by

[1] The multiplicity of intersection point p of two polynomials $c_1(x)$ and $c_2(x)$ is the least positive k such that $f^{(k)}(p) \neq 0$, where $f(x) = c_1(x) - c_2(x)$.

[2] We use the term *envelope order* as it applies equally to lower and upper envelopes.

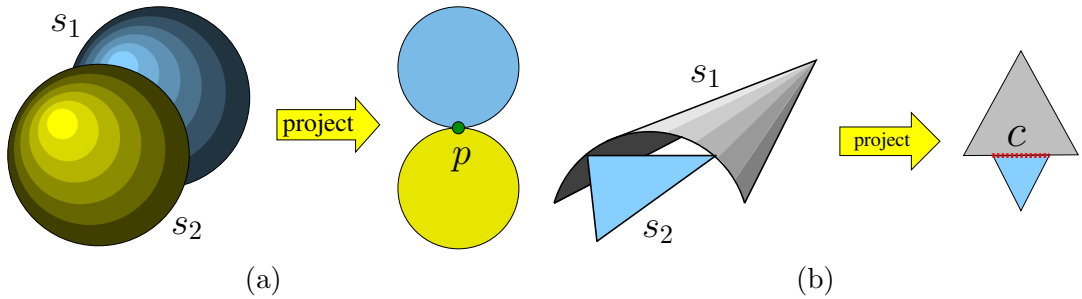


Figure 4.1: (a) The spheres s_1 and s_2 have only one two-dimensional point in their common xy-definition range. Over this point the algorithm will compare them. (b) The surfaces s_1 and s_2 will be compared over the edge c. The example shows why the comparison should be done over the interior of an x-monotone curve, excluding its endpoints.

the algorithm to determine the envelope order of two xy-monotone surfaces on one side of their projected intersection curve when their order on the other side of that curve is known, thus improving the performance of the algorithm, as is explained in Section 5.2.

Note that this method is used also in situations where the surfaces do not intersect, and it is the responsibility of this method to indicate this fact, and return the empty set. This design gives the traits the maximum flexibility in the implementation of this operation, as sometimes there is an easy and efficient intersection test and the implementation can immediately return when it identifies that the surfaces do not intersect.

4. Given two xy-monotone surfaces s_1 and s_2, and a planar point p, which lies in their common xy definition range, determine the envelope order of s_1 and s_2 at the xy-coordinates of p.

   This operation is used by the algorithm to determine the label of a vertex, whose associated point is p. It is needed only for degenerate cases. See Figure 4.1(a) for an illustration of a situation where this operation is used.

5. Given two xy-monotone surfaces s_1 and s_2, and a planar x-monotone curve c, which is a part of their projected intersection, determine the envelope order of s_1 and s_2 immediately above (similarly below) the curve c (in the plane). Note that c is a curve in the plane, and we refer to the region above/below c in the *plane*, here and in the description of Operation 6. If c is a vertical curve, we regard the region to its left (similarly right) as the region above (similarly below) c.

   This operation is used by the algorithm to determine the label of a face incident on c, immediately above (below) it.

6. Given two xy-monotone surfaces s_1 and s_2, and a planar x-monotone curve c, which

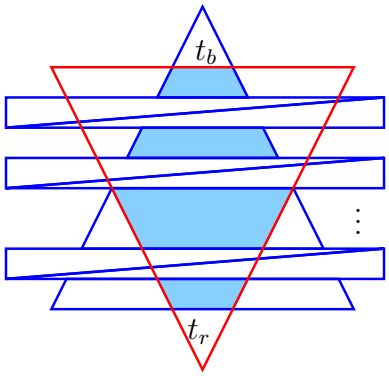


Figure 4.2: The two big triangles t_r and t_b should be compared over all the filled faces in the overlay.

lies fully in their common xy definition range, such that s_1 and s_2 do not intersect over the interior of c, determine the envelope order of s_1 and s_2 over the interior of c.

This operation is used by the algorithm to determine the label of an edge, whose x-monotone curve is c, or of a face incident on c, for which Operation 5 cannot be used. See Figure 4.1(b) for an illustration of a situation where this operation is used.

All the traits operations should be defined as function objects (*functors*) [7], as is massively used in the CGAL geometry kernel. The main advantage of functors over regular methods is the ability to extend the traits geometric objects without the need to redefine the operations on them (see [44] for an explanation about the CGAL extensible kernel).

4.2 The Caching Traits Classes

The caching traits classes are components that extend a traits class with a caching ability. There are two possible caches:

1. **Projected intersection cache.** In the overlaid arrangement, there may be several features with the same pair of xy-monotone surfaces (one surface from each of the two envelopes currently being merged), for which the minimization diagram shape is to be determined (see Figure 4.2). The first step in the process of resolving a feature is to find the projected intersection of the surface, and this is usually a very costly operation. Thus, caching these projected intersections can improve performance.

2. **Cache for comparison result of two xy-monotone surfaces in some special cases.** If two xy-monotone surfaces do not intersect and the projection of the two of them onto the xy-plane is convex, then if we know their envelope order over one point in their xy definition range, this order is valid for all other points in their common definition range. Suppose that p is a planar point in the common definition range

of the surfaces, where we know their envelope order. Let p' be another point, where we wish to determine the envelope order of the surfaces. If the projection of the two surfaces onto the xy-plane is convex, then all points of the segment pp' lie in the definition range of both surfaces, and the surfaces can change their order over that segment only if they intersect over it, since we assume they are continuous. So when they do not intersect, their envelope order over p' is the same as over p. Note that this characteristic is not true for general xy-monotone surfaces. The non-intersection test is carried out using the projected intersection operation, so it makes sense to use this cache together with intersections cache. Considering the algorithm flow, this is a natural requirement, since the projected intersection operation for two xy-monotone surfaces is always computed before any call to a comparison operation between these two surfaces.

Each cache is implemented in a separate class. From the algorithm point of view, each caching traits class plays the role of an ordinary traits class. The main advantages of a separate component for the caching are reusability and modularity. Reusability, since one can use the caching support on top of any available traits class, and modularity, since one can choose not to use caching at all in some situations, for example if memory usage is more costly than recomputing time, or when the caching is built inside a specific traits class.

When using the projected intersections caching traits class on top of a given traits class, the latter should be a model of the `EnvelopeCachingTraits_3` concept. This concept is a refinement of the `EnvelopeTraits_3` concept with the additional requirement that a pair of xy-monotone surfaces can be uniquely identified inside the cache.

When using the second caching traits class on top of a given traits class, the latter should be a model of the `EnvelopeCompareCachingTraits_3` concept. This concept is a refinement of the `EnvelopeCachingTraits_3` concept with the following additional operation requirement: given an xy-monotone surface, check if its projection onto the xy-plane is convex. This operation is used to check if it is possible to cache comparison results, where this surface is involved.

4.3 More Design Issues

Working with xy-monotone surfaces instead of with general surfaces that are not necessarily xy-monotone is more convenient. Some of our methods described in Chapter 5 are valid only under the assumption that the surfaces are xy-monotone. Yet, this assumption does not contradict the generality of the algorithm, since at an initial step we extract the relevant (xy-monotone) portions of the (not necessarily xy-monotone) surfaces.

4.3.1 Minimal Set of Requirements

Making the traits-class concept as tight as possible, by identifying the minimal number of required methods, is crucial. It can make the whole difference between being able to

implement a traits class for a specific type of surfaces or not, and may have a major effect on the efficiency of the algorithm, especially for non-linear objects. This observation has guided us in our design. Our algorithm does not require the traits class to define any three-dimensional types except the surface types. The interface with the traits class contains only the types that are necessary for the input and the output of the algorithm, and a small set of operations defined on these types. This minimization of the requirements gives the maximum flexibility to the traits implementation. In situations where it is easy and helpful, three-dimensional objects can still be defined inside the traits class, and used as intermediate objects. Three-dimensional data may also be attached to the two-dimensional objects. However, sometimes the intermediate spatial objects are not needed or not known in the traits implementation (see, for example, the quadrics traits in Chapter 6) and our algorithm can still be used to successfully compute the envelope.

4.3.2 Envelope Viewpoint

Our implementation of the divide-and-conquer algorithm is completely independent of the direction in which the envelope is to be computed. The traits class is responsible for controlling this direction. The main operations where this independence is manifested are:

1. The different comparisons operations. These operations indicate to the algorithm which of the two input xy-monotone surfaces appears on the envelope of these two surfaces over the given query region. This is exactly what the algorithm should know in order to correctly label the relevant feature.

2. The extraction of xy-monotone surfaces from a general surface. This operation is defined to return the xy-monotone portions of the surface that are relevant to the envelope. Thus, computation regarding the other portions of the surface can be avoided, which is highly desirable, since these portions will not appear on the final envelope anyway. For example, if we compute the lower envelope of spheres, we can ignore all the upper hemispheres, since they do not appear on the lower envelope.

All our traits classes support the computation of a lower and an upper envelope. It is also possible to use our algorithm to compute the envelope seen from a direction that is not parallel to the z-axis, provided that the traits class correctly implements all operations with respect to that direction.

Chapter 5

Algorithmic Details

In this chapter we provide more details on our algorithm. First, we explain the process of handling a face in the overlay of two minimization diagrams in order to determine its shape in the result minimization diagram. Second, we describe how the features are labelled with the correct surfaces. We describe the methods (sometimes tricks) we use in order to minimize the number of geometric/algebraic operations, substituting them with combinatorial labelling. We continue with a description of some of the degeneracies, and the way we handle them. We conclude the chapter with a complexity analysis of both time and working storage, and some ideas for improvement of our work.

In the following sections, we assume that we perform the merge step of two minimization diagrams $\mathcal{M}_1$ and $\mathcal{M}_2$ (representing the lower envelopes $\mathcal{E}_1$ and $\mathcal{E}_2$ respectively) of two sets of xy-monotone surfaces $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively. The result of the merge is a minimization diagram $\mathcal{M}$ (representing the lower envelope $\mathcal{E}$) of the set $\mathcal{G}_1 \cup \mathcal{G}_2$.

5.1 Handling a Face in the Overlay of Two Minimization Diagrams

Recall that in the merge step of the algorithm, after we overlay the two minimization diagrams from the previous recursive step, we have to determine how the envelope looks over each feature. We call this step *resolving the feature*. Since a vertex cannot split, and the resolving of an edge is quite simple, we will describe here only the process of resolving a face. We have a face f with two xy-monotone surfaces s_1 and s_2 defined over the entire face. We wish to compute the envelope of s_1 and s_2 over f. We do not assume any restrictions on the size or shape of f. The envelope of s_1 and s_2 over f is affected only by the intersections of these surfaces. If we look at the minimization diagram of s_1 and s_2 restricted to f, it is affected exactly by the projection of the intersection of s_1 and s_2. Thus, we reduce to the problem of partitioning f with these projected intersection. Let us denote by $\mathcal{W}$ the arrangement that we work on; in the beginning of this step, it contains the overlay of $\mathcal{M}_1$ and $\mathcal{M}_2$, and at the end of this step, it would contain a refinement of the overlay with projected intersection curves. A simple but dangerous solution is to use the arrangement

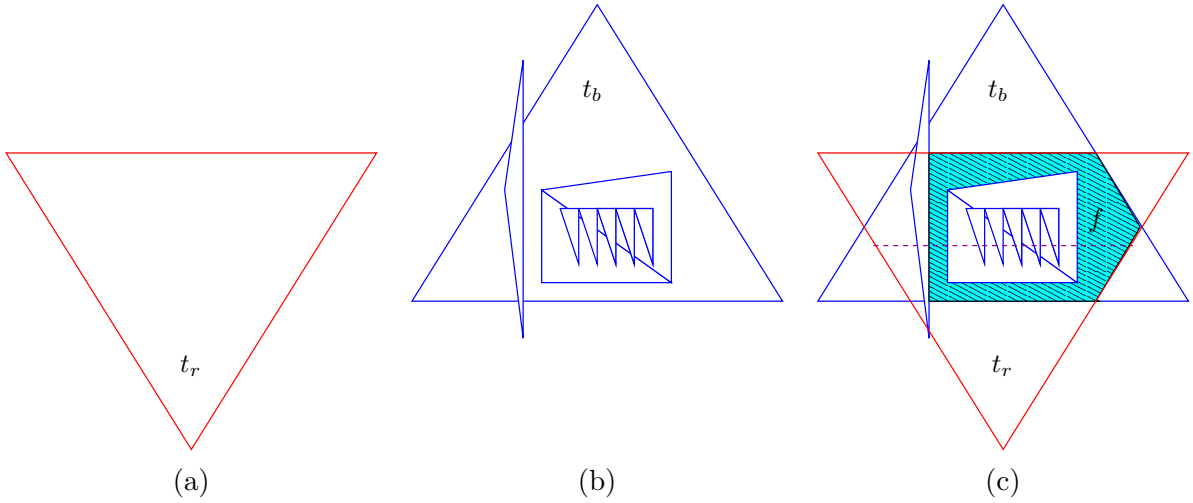


Figure 5.1: Given the minimization diagrams shown in (a) and (b), for handling the shaded face in their overlay (c), one has to split f with the projected intersection segment of t_b and t_r, shown in dashed line, which intersects many irrelevant edges of the overlaid arrangement.

interface insertion method as is on $\mathcal{W}$. As already mentioned in Chapter 3, this may result in a lot of extra computation. The curves of the projected intersections might intersect many edges outside f, even inside the holes of f; see Figure 5.1 for an illustration. These planar intersections outside f are irrelevant to the envelope of s_1 and s_2 over f, and we wish to avoid them. We wish to compute intersections only within the boundary of f. We achieve this goal in the following way. First, we create an empty arrangement arr_f, and copy only the boundary of f there, including the outer boundary, and all the inner boundaries (of the holes). This step incurs minimal overhead, while making the problem much easier. Then, we insert the projected intersection curves into arr_f in a controlled way: only the parts of the curves that fall inside f are inserted, the other parts are discarded. The insertion is based on finding the zone of the inserted curves in the arrangement; see more details in Section 5.4.2. Eventually, we will need those parts inside $\mathcal{W}$, and not just in arr_f, so we need to know all the changes made to arr_f, to repeat them on $\mathcal{W}$. For this task we attach an observer to arr_f. *Observer* is a design pattern which defines a one-to-many dependency between objects, such that when one object changes state, all the dependents are notified and updated automatically [35]. The observer design pattern is used in the `Arrangement_2` package for the implementation of the *notification mechanism* — the arrangement class has a list of observers, that registered themselves with the arrangement object to be notified about any change to its structure [67]. Whenever something happens in arr_f — a new vertex is created, an edge of arr_f is split, a new edge is created, a face is split, a new hole is created, and so on — the observer gets the details of the event, and it is possible to mimic the same actions exactly in the original arrangement, exploiting the geometric information already computed in arr_f without computing it again. In order for this mechanism to work properly, we maintain cross pointers between the features in arr_f and their counterparts

in the original arrangement.

5.2 The Labelling Step

We invested considerable effort in trying to minimize the amount of algebraic computation whenever possible, as such computation is usually very costly. This actually means substituting calls to the different traits-class methods by the propagation of information (in the form of labels) between incident features. In Section 4.2 we already presented two types of information caching that can be used to avoid recomputing the projected intersection for the same pair of xy-monotone surfaces and the envelope order of a pair of xy-monotone surfaces in a special case. In this section, we concentrate on a careful usage of the different comparison operations for two xy-monotone surfaces in the general case, which reduces the number of algebraic operations significantly.

Let $\mathcal{O}'$ be the arrangement, which is created by overlaying the arrangements underlying $\mathcal{M}_1$ and $\mathcal{M}_2$, and determining the shape of the minimization diagram over each feature of the overlay. $\mathcal{O}'$ is a refinement of the arrangement underlying $\mathcal{M}$, the result minimization diagram. For each feature f of $\mathcal{O}'$ the envelope $\mathcal{E}$ is attained by the same set of surfaces over all points of f. $\mathcal{O}'$ is the input to the labelling step, which determines the correct label of all the features of $\mathcal{O}'$. For each feature f of $\mathcal{O}'$ we have to decide between two labels from the two minimization diagrams currently being merged; we say that f is associated with a *decision*. A *decision* can be one of three values: first, when the feature should be labelled with all the surfaces of the first label, second, when the feature should be labelled with all the surfaces of the second label and both, when the surfaces of both labels overlap over the feature. We work with decisions in the labelling step, and after the cleanup step, we translate the decisions into the relevant labels.

Obviously, in order to make a decision for a feature, we can use one of the three types of comparison operations described in Section 4.1. However, we can use the following observations to significantly reduce the number of such operations. These savings are demonstrated in the experiments reported in Section 7.6.

- No need to compare xy-monotone surfaces over their projected intersection, since they are equal there. Thus, edges and vertices that coincide with the projected intersection of the surfaces that are compared over them are associated with the decision value both.

- Information on intersection type can be used when available, to avoid the comparison of two xy-monotone surfaces on one side of a curve (which is a part of their projected intersection) if their envelope order on the other side of that curve is known. Recall that such information is (optionally) given by the traits operation which constructs the projected intersection of two xy-monotone surfaces. Similar information is extremely helpful in constructing two-dimensional arrangements of curves [32].

- Information on the continuity or discontinuity of the two envelopes currently being merged can be used in order to conclude the decision for a feature from a decision

on an incident feature. We regard the following as incident features: (i) a face and an edge on its boundary, (ii) a face and a vertex on its boundary, and (iii) an edge and its endpoint vertices. In the following section we explain how this information is used.

Table 7.3 in Chapter 7 demonstrates the significant savings in geometric operations by the techniques presented in this section. For example, for a set of 1000 random triangles the total number of comparison operations reduces from 265,211 to 13,620, and for a set of 1000 random spheres the total number of such operations reduces from 48,842 to 2,457, which is roughly a saving of 95% in either case.

5.2.1 Using Continuity or Discontinuity Information

We consider the xy-monotone surfaces as graphs of partially defined bivariate continuous functions, and their envelope as a function defined over the entire xy-plane. In addition, we consider the boundary of an xy-monotone surface to be part of this surface.

Definition 5.1 *Let $\mathcal{E}$ be an envelope and $\mathcal{M}$ its minimization diagram. Let f and e be two incident features of $\mathcal{M}$, such that e lies on the boundary of f. We say that $\mathcal{E}$ meets f and e continuously if $\mathcal{E}$ restricted to $f \cup e$ is continuous over e.*

Observation 5.2 *Let $\mathcal{E}$ be an envelope of a set $\mathcal{S}$ of surfaces and $\mathcal{M}$ its minimization diagram. Let f and e be two incident features of $\mathcal{M}$, such that e lies on the boundary of f. $\mathcal{E}$ meets f and e continuously if and only if there exists an xy-monotone surface $s \in \mathcal{S}$ which appears over f and e on the envelope $\mathcal{E}$.*

If s as defined above exists, since it is continuous, $\mathcal{E}$ restricted to f and e is also continuous. If $\mathcal{E}$ meets f and e continuously, an xy-monotone surface s, which appears over f on $\mathcal{E}$ is defined on f's boundary, thus, over e, and must appear on the envelope over e because of the envelope's continuity there.

We use this observation in our implementation to decide where an envelope meets two incident features continuously. We now explain how we use this information to reduce the number of geometric comparisons.

Lemma 5.3 *Let f be a face of $\mathcal{O}'$ and e be an edge on its boundary. Suppose that:*

1. *A decision over a face f is known.*
2. *Both envelopes $\mathcal{E}_1$ and $\mathcal{E}_2$ meet f and e continuously. Let s_1 and s_2 be the xy-monotone surfaces that appear over f and e on these envelopes respectively.*
3. *e is not part of the projected intersection of the surfaces s_1 and s_2.*

Then the decision made on f is valid also on e.

Proof: Since the decision over f is known, the envelope order of s_1 and s_2 over f is known, even if s_1 and s_2 were not compared over f directly. Recall that a label on a feature may contain several surfaces, and as a result, it is possible that another pair of surfaces were compared over f. But all the surfaces of a label overlap over the feature, thus, the envelope order of s_1 and s_2 over f is known. From the continuity of the xy-monotone surfaces, and the fact that they do not coincide over e, it is clear that the same envelope order is valid over e. □

Remark 5.4

1. In order to use Lemma 5.3, the algorithm must know whether e is part of the projected intersection of s_1 and s_2. This information is not trivial to extract, but it is available in the step of resolving the features. We save this information until the labelling is completed.

2. In the conditions of Lemma 5.3, when e is part of the projected intersection of s_1 and s_2, e is associated with the decision value both. We mentioned above that the algorithm does not compare surfaces over their projected intersection, since they are equal there. If the projected intersection falls inside the resolved face, then the new features are immediately associated with the decision value both. When the projected intersection overlaps features on the boundary of this face, they can be associated with the decision value both only if the conditions of Lemma 5.3 are met.

Observation 5.5 *Similar arguments as in the proof of Lemma 5.3 can be used to carry a decision over from an edge e to an incident face f, and between other types of incident features.*

Sometimes, we can use also the *discontinuity* information to deduce a decision for a feature without comparing the relevant surfaces.

Observation 5.6 *Let $\mathcal{E}$ be a lower envelope and $\mathcal{M}$ its minimization diagram. Let f and e be two incident features of $\mathcal{M}$, such that e lies on the boundary of f. Let s_f and s_e be representative xy-monotone surfaces of $\mathcal{E}$ over f and e respectively. $\mathcal{E}$ does not meet f and e continuously if and only if s_e lies below s_f over e (note that s_f is defined over e).*

Applying Observation 5.6 to the labelling step we get:

Lemma 5.7 *Let f be a face of $\mathcal{O}'$ and e be an edge on its boundary. Assume that the lower envelope $\mathcal{E}_1$ meets f and e continuously, but the lower envelope $\mathcal{E}_2$ does not.*

1. *If $\mathcal{E}_2$ is below $\mathcal{E}_1$ over f, then $\mathcal{E}_2$ is below $\mathcal{E}_1$ also over e.*
2. *If $\mathcal{E}_1$ is below $\mathcal{E}_2$ over e, then $\mathcal{E}_1$ is below $\mathcal{E}_2$ also over f.*

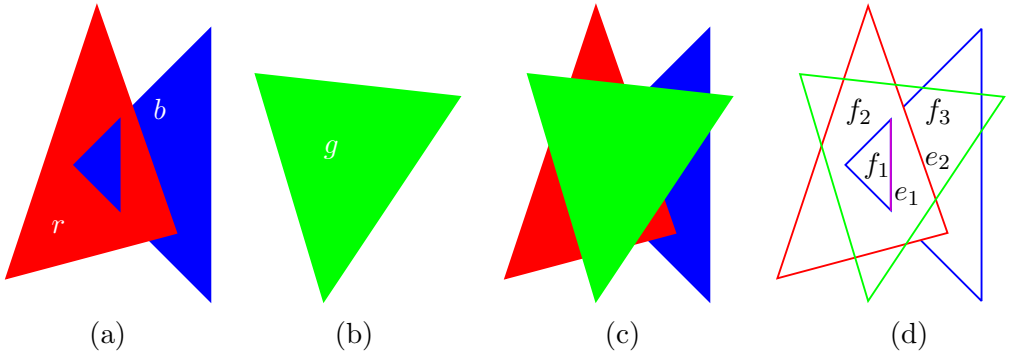


Figure 5.2: Applying continuity or discontinuity arguments to carry on a decision between incident features: (a) the envelope of two triangles r and b, (b) the envelope of one triangle g, (c) the envelope of r, b and g, (d) the overlaid arrangement before the labelling step. To label face f_1 we compare triangles b and g and find out that g appears on the envelope there. Using Lemma 5.3, we label all the features on the boundary of f_1 with g. To label face f_2, where triangles r and g should be compared, we use Observation 5.5 and edge e_1 to conclude that g appears on the envelope, without actually comparing r and g. Using Lemma 5.3, we label all the features on the boundary of f_2 with g. To label face f_3, we use Lemma 5.7 and edge e_2 to set the label to g. To summarize, we need only compare triangles b and g once, and all the other decisions follow.

Proof: Denote by t the xy-monotone surface that appears over f and e on $\mathcal{E}_1$, and by s_f, s_e the xy-monotone surfaces that appear on $\mathcal{E}_2$ over f and e respectively. By Observation 5.6, s_e is below s_f over e.

1. $\mathcal{E}_2$ is below $\mathcal{E}_1$ over f, thus s_f is below t over f, and also over e (s_f and t may coincide over e). Since s_e is below s_f over e, it is below t there too, i.e., $\mathcal{E}_2$ is below $\mathcal{E}_1$ over e.

2. $\mathcal{E}_1$ is below $\mathcal{E}_2$ over e, thus t is below s_e over e. Since s_e is below s_f over e, t is below s_f there too. By the continuity of the xy-monotone surfaces, t is below s_f over f, i.e., $\mathcal{E}_1$ is below $\mathcal{E}_2$ over f.

□

Observation 5.8 *The arguments of Lemma 5.7 apply similarly to all other incidence relationships.*

Figure 5.2 illustrates how the observations above are used in the labelling step.

5.2.2 Implementation Details

We saw that it is possible to carry a decision over from a face to a boundary edge and vice versa, so the order in which we traverse the faces of the overlaid arrangement and resolve

them seems important. We wish to traverse the faces in a breadth-first order, moving from a face to its neighboring faces. The BOOST[1] graph library [63] is a generic library of graph algorithms and data structures. The CGAL `Arrangement_2` package provides an adaptor of the arrangement class into a graph, which associates a graph vertex with each arrangement face, such that two vertices are connected if and only if there is an arrangement edge that separates the two corresponding faces. We use this adaptor to perform a breadth-first traversal of the faces.

We already mentioned that in order to decide whether an envelope meets two incident features continuously, it is possible to check if there exists a common xy-monotone surface in their labels. This test is automatically interpreted as finding whether the intersection of two lists of xy-monotone surfaces is non-empty. Operations on such lists are not desirable, though, neither from a theoretical point of view nor practically. Since a label may contain up to n xy-monotone surfaces, we may end up with $\Theta(n)$ time test just to save one geometric comparison. Such a test also poses an additional requirement from the traits class, to be able to compare two xy-monotone surfaces. Fortunately, there is a better solution. Using a constant number of Boolean flags for every halfedge and vertex of the minimization diagram, it is possible to perform the continuity test in constant time, with minimal overhead, as follows. For every halfedge we keep three types of Boolean flags: (i) to indicate whether the halfedge and its incident face have a common xy-monotone surface in their labels, (ii) to indicate the same thing for the halfedge and its target vertex, and (iii) to indicate the same relation for the halfedge's target vertex and the halfedge's face — note that a halfedge is a natural place to put additional information regarding a relation between a face and a vertex on its boundary. For an isolated vertex, we keep a flag to indicate whether the vertex and its containing face share a common xy-monotone surface in their label. We use the flags of the minimization diagrams $\mathcal{M}_1$ and $\mathcal{M}_2$ to set the decisions over the features in the labelling step. After the cleanup step, we use these flags along with the decisions to update the correct flag values for the features of the result minimization diagram.

A similar issue arises in the cleanup step, where a test for deciding whether to remove an edge or a vertex involves a test for equality of labels of incident features. As in the labelling step, we can avoid comparing labels by maintaining a constant number of Boolean flags for halfedges and for isolated vertices. These flags indicate the relations between incident features as does the set of flags described above for performing the continuity test. However, they have different meanings: the flags described above indicate whether two incident features have a common xy-monotone surface in their labels, whereas the flags described here indicate whether the labels of two incident features are equal. For every halfedge we keep two types of Boolean flags: (i) to indicate whether the halfedge and its incident face have the same set of xy-monotone surfaces in their labels, and (ii) to indicate the same thing for the halfedge and its target vertex. The information of equality of labels between a face and a vertex on its boundary, though needed in some degenerate situations, can be deduced from the other two flags, as opposed to the continuity test between a face

[1] BOOST C++ Libraries `http://www.boost.org`

and a vertex on its boundary. In addition, a flag for an isolated vertex is stored, which indicates whether the vertex and its containing face have the same label. We use the flags of the minimization diagrams $\mathcal{M}_1$ and $\mathcal{M}_2$ together with the decisions made over the features in the labelling step in order to decide whether a feature is redundant and should be removed. We also use them to set the correct values of these flags for features of the result minimization diagram.

5.3 Handling Degeneracies

Our implementation does not assume general position, rather, it handles all types of degenerate input. Many degeneracies require straightforward handling. In this section we describe two degeneracies where the handling is less obvious: vertical surfaces and overlapping surfaces. We briefly list some other degeneracies and the way of handling them.

5.3.1 Vertical Surfaces

Vertical surfaces are not handled directly by the algorithm, but should be handled inside the traits class, whenever the family of surfaces supported by the traits includes vertical surfaces. The main reason is that vertical surfaces are not xy-monotone, and our algorithm works on xy-monotone surfaces. From the algorithm point of view, vertical surfaces are treated just like any other general surfaces, where xy-monotone portions need to be supplied by the traits. So actually, the algorithm treats vertical surfaces as the relevant curves, that are their envelope. It is the traits's responsibility to implement the three-dimensional methods in a way that supports this view.

5.3.2 Overlapping Surfaces

Recall that, as we do not assume general position, the label of each feature in the minimization diagram may consist of several xy-monotone surfaces, all of which overlap over the feature. Our representation of a minimization diagram supports labels containing several overlapping surfaces. When dealing with overlapping xy-monotone surfaces in the merge step, the algorithm expects the comparison operations of the traits class to return the value equal over the relevant feature, in which case the decision value both is set on that feature, and it is labelled with the union of the two labels whose surfaces were compared. In order to get the correct shape of the envelope in the step of resolving the features, the projected-intersection operation of the traits class is expected to return the projection of the boundary of the overlap region(s).

Our triangles traits class uses a more efficient way of handling overlaps. It does not compute the intersection between overlapping triangles at all. The important observation is that the boundary of an overlap region of two triangles consists only of segments that are part of the triangles boundary. Thus, these segments do not give any new information on the shape of the minimization diagram. Moreover, handling the projection of the segments

that bound the overlap region in the resolving process may result in unnecessary geometric computation. Instead, if the traits discovers that two triangles have the same supporting plane, the result of the projected-intersection operation is the empty set (note that identical supporting plane is an indication of an overlap because the projected-intersection operation is always called for surfaces that have a common xy definition region). In such a case, the various comparison operations return the value equal without computing and comparing the z-coordinates of points on the triangles, as is done in the general cases.

Another solution (which is not implemented and we propose for future work) for handling overlapping surfaces avoids comparing xy-monotone surfaces over their overlap region at the algorithm level. Quite often when using *exact* geometric computation, comparing equal numbers is more time consuming than comparing distinct numbers. Thus, this solution may be more efficient. The idea is to give the traits the possibility to mark an x-monotone curve, which is a part of a projected intersection, as a projected boundary of an overlap region, and to indicate on which side of that x-monotone curve the overlap region is located. By checking this indicator, the algorithm can set the decision value both on the relevant region, without actually comparing the surfaces.

5.3.3 Additional Degeneracies

Many different degenerate situations may arise when the general position assumption is removed. We point out some of them: more than three or two xy-monotone surfaces may intersect in a three-dimensional point or curve, the projection of boundary/intersection curves of several xy-monotone surfaces may intersect at one point or overlap, even when there is no intersection in three-space, projected curves may be vertical, and many more. Our algorithm works mainly in the plane. Since the `Arrangement_2` package deals with all possible degeneracies, we are given some degeneracy handling "for free". For example, in the overlay of two minimization diagrams, for all the features in the overlay we get the correct pair of features in the two input diagrams that created it: two coinciding vertices, two overlapping edges, a vertex lying on an edge, and so on. Another example is the zone algorithm we use for resolving the faces, which identifies exactly which features are intersected by a given curve, including intersection at vertices and overlapping segments of the curves. In addition to the degeneracies dealt by the `Arrangement_2` package operations, our algorithm has some work left to do by itself. For example, when resolving a face in the overlay, the algorithm inserts projected intersection curves into an arrangement that contains that face's boundary. The projected intersection curves may not only cross edges of the face's boundary, but may also overlap with boundary edges, and touch boundary vertices. These are special cases for the labelling step, which follows the step of resolving the face, since features that coincide with projected intersections are labelled differently than other boundary features.

5.4 Complexity Analysis

First, we analyze the complexity of the algorithm in theory (both running time and working storage), assuming general position. Then, we list issues that come up in our implementation, usually caused by particularly degenerate situations, which may increase the worst case running time. We also give ideas for possible improvements of the implementation in these cases.

5.4.1 Theoretical Analysis

In this section we analyze the combinatorial complexity of the divide-and-conquer algorithm for computing the envelope of surfaces, under the general position assumption. Denote by $T(n)$ the running time for n input surfaces. $T(n) = 2T(\frac{n}{2}) + M(n)$, where $M(n)$ is the cost of the merge step in which a total of n surfaces are involved. We concentrate on the merge step. As already mentioned in Chapter 2, the combinatorial complexity of the lower envelope of n "well behaved" surfaces in $\mathbb{R}^3$ is $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$. This means that each of the two minimization diagrams $\mathcal{M}_1$ and $\mathcal{M}_2$ that we merge has complexity $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$.

- Overlaying $\mathcal{M}_1$ and $\mathcal{M}_2$ is conducted using a sweep-line algorithm in time $O((|\mathcal{M}_1| + |\mathcal{M}_2| + |\mathcal{O}|) \log n)$, where $\mathcal{O}$ is the overlay result. The overlay of two minimization diagrams of a total of n surfaces is of size $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$ [4] (note that this result is non-trivial since the overlay of two arrangements of sizes n and m is in general of size $O(mn)$). Thus, the overlay step takes $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$.

- Performing a vertical decomposition on $\mathcal{O}$, yields a refinement of $\mathcal{O}$ with complexity that is linear in the complexity of $\mathcal{O}$. This can be done using a sweep-line algorithm in time $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$.

- Resolving each feature and labelling the sub-features is carried out in constant time, since each feature is of constant size, and is cut with a constant number of projected intersection curves, into a constant number of sub-features. Thus, the whole step takes time $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$.

- In the cleanup step, checking whether an edge (or a vertex) is relevant to the result, or should be removed from the result, involves only the feature and a constant number of incident features. Thus, each such decision is carried out in constant time, and the overall cleanup step takes $O(n^{2+\varepsilon})$ time, for any $\varepsilon > 0$. One possible way to achieve this running time is to build the result from the relevant curves only using a sweep-line algorithm.

The merge step takes $O(n^{2+\varepsilon})$ time, for any $\varepsilon > 0$, and thus, the whole algorithm has this running time bound.

Regarding the working storage used by the algorithm, we should also look at the merge step. The two minimization diagrams $\mathcal{M}_1$ and $\mathcal{M}_2$ have complexity $O(n^{2+\varepsilon})$, for any

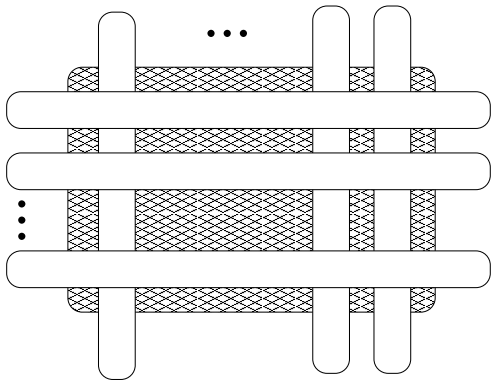


Figure 5.3: Example where the naïve representation of the minimization diagram can be of size $\Theta(n^3)$. $\frac{n}{2}$ surfaces overlap over the shaded region. The remaining $\frac{n}{2}$ surfaces form a grid below the overlap region in three-dimensions, dividing the two-dimensional overlap region into $\Theta(n^2)$ sub-regions, each with a surface list of size $\frac{n}{2}$.

$\varepsilon > 0$, and so has their overlay. Performing a vertical decomposition does not increase the asymptotic complexity of the arrangement. After inserting all the relevant parts of the projected intersections, the arrangement size does not increase asymptotically. The cleanup step can only reduce the working storage. Thus, the total amount of memory used in the merge step, and in the whole algorithm is $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$.

5.4.2 Implementation Issues and Possible Improvements

In this section we describe some implementation details that may affect the worst case running time of the algorithm, together with ideas for improvements.

1. **Complexity of the minimization diagram allowing many overlaps**. In the current naïve representation of a minimization diagram, each feature contains a list of surfaces that appear on the envelope over its region. Each such list can be of size $\Theta(n)$, in case of overlaps. This means that the overall complexity of the minimization diagram, including all these lists, may be of size $O(n^{3+\varepsilon})$. See Figure 5.3 for an example, where the representation is of size $\Theta(n^3)$. This can, of course, lead to worsening of the algorithm running time and working storage.

   A possible solution is to gather all the surfaces of a list in one label object, and point from each feature onto one such label object. Every label object will point to one surface, or to two other label objects (that unite into one label in a recursion step). Every label should also contain a pointer to one representative surface, to be quickly accessible when needed, and to be used in the process of determining the shape of a face in the minimization diagram. Now, every feature of the arrangement contains a constant size additional information. The total number of labels created throughout the algorithm is $O(n^{2+\varepsilon})$, each of which adds constant storage to the overall storage.

Thus, we gain a representation for envelope of size $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$, which is worst-case near optimal.

2. **Resolving a complex face vs. vertical decomposition**. In theory, we only have to deal with constant size faces, because a preliminary vertical decomposition is performed on the arrangement. In practice, it seems better to deal with the complicated faces instead (see experimental results and discussion in Section 7.7). However, working with large size faces, which may contain many holes, affects the worst case running time in the current implementation.

   Let f be a face of the overlaid arrangement, and h_f the number of its holes. Recall that the first step in the resolving process is to copy f into an empty arrangement. This is done in time $O(|f|)$. Then, a constant number of x-monotone curves and points (which form the projected intersection of two surfaces) are inserted into this arrangement. Each of these curves may intersect every edge of f a constant number of times, and f may split into $O(|f|)$ sub-faces. The current implementation uses the available zone algorithm, provided by CGAL's `Arrangement_2` package. The zone algorithm identifies all the arrangement features that are crossed by the inserted curve. The zone is computed by locating the left endpoint of the query x-monotone curve and then "walking" along the curve towards the right endpoint, keeping track of the vertices, edges and faces crossed on the way [67]. Whenever the inserted curve enters the face, the whole face boundary is scanned to find the next intersection, with no history saved for that traversal. Thus, this implementation runs in time $O(|f|^2)$, in the specific case of our use, even when the face contains no holes at all. We implemented another version of the zone algorithm which runs in time $O(|f|(h_f + \log|f|))$, h_f is the number of holes in f. The main improvement in this implementation is the maintenance of a sorted set of two-dimensional intersections computed already. The first time the x-monotone curve enters a specific face, all its boundary edges are traversed and the intersections with them are found and inserted into the set, sorted from left to right. When the x-monotone curve re-enters a previously traversed face, the next intersection point with the boundary of the face can be determined in constant time using the sorted set. Thus the boundary of every face is traversed only once. The multiplicative h_f comes from relocating the holes in the appropriate face, when a face splits into two sub-faces. However, this improvement is still not sufficient for getting the theoretical bound.

   A possible solution for this problem is to use a variation of the sweep-line algorithm for this task, which runs in time $O(|f|\log|f|)$. With this solution, the step of determining the shape of all features and labelling them takes $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$, preserving the theoretical asymptotic bound.

   We remark that resolving a complex face as conducted in the current implementation does not affect the working storage of the algorithm, only its running time. The copied arrangement of a face f has size $O(|f|)$, and is destroyed after the handling of that face. Furthermore, every feature is associated with a constant amount of

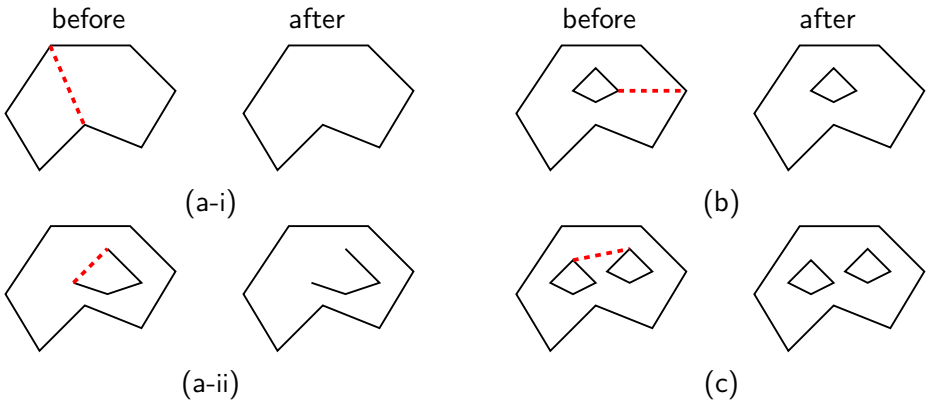


Figure 5.4: Problematic situations when an edge is removed from the arrangement. The removed edge is shown in a dashed line. (a-i),(a-ii) two faces merge, (b) the outer boundary of a face splits, creating a new hole inside the face, and (c) an inner boundary of a face splits into two separate holes.

additional data that help in the handling of the face and in the labelling of the relevant features (disregarding the labels' sizes as explained before). Hence, the working storage used is $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$, as well.

3. **Cleanup**. As described in Section 5.4.1, the cleanup step should take $O(n^{2+\varepsilon})$ time. In our current implementation, we maintain a valid arrangement after every edge removal. A single edge removal can take more than constant time, and this time can even be linear in the arrangement size. This may result in worsening the asymptotic running time of the cleanup step, and in turn of the whole algorithm. To understand why, recall from Section 2.4.4 that the topology of the arrangement is maintained in a Dcel data structure. Every halfedge in a Dcel contains a pointer to its incident face, and every halfedge that is part of an inner boundary contains a pointer to the hole it belongs to. When an edge is removed there are few problematic situations:

   (a) Two faces may merge into one face. In the current implementation, one face becomes the merged face and the other is removed. As a result the face pointer of all the halfedges of the removed face should be updated. In addition, all the holes and isolated vertices of the removed face should be inserted into the new face. Figures 5.4(a-i) and 5.4(a-ii) show illustrations of two situations where two faces merge when removing an edge.

   (b) The outer boundary of a face may split into two parts, creating a new hole inside the face; see Figure 5.4(b) for an illustration of this situation. The hole pointer of all the halfedges of the new hole should be updated.

   (c) An inner boundary of a face may split into two parts, creating a new hole inside the face; see Figure 5.4(c) for an illustration of this situation. The hole pointer of all the halfedges of the new hole should be updated.

A possible solution is to do all the expensive updates of the Dcel data in bulks, instead of after every single removal. First, deal with edges that cause two adjacent faces to merge — but create a full face of the result, merging all the possible faces, and only then update the halfedge's face and hole data, once for each face in the result. Second, deal with the connected components of the boundary — but split every connected component to all the holes it should split into, before updating the relevant data of the halfedges. In this way, every halfedge is updated at most twice, and the total cost of the removal step is as desired.

In practice, all the cleanup steps during the recursion take a small percentage of the total algorithm time, for all the input sets that we tested (see Section 7.5), so it seems reasonable to use a naïve solution here.

Chapter 6

Envelopes of Quadrics

The work reported in this chapter was conducted jointly with Eric Berberich from Max-Planck-Institut für Informatik (MPII).

Quadratic surfaces (*quadrics*) in three-space are defined as the set of roots of trivariate quadratic polynomials. Quadrics are at the front of research on robust three-dimensional geometric modelling. For recent results in the computational study of quadrics see [12, 23, 36, 49, 54, 70]. For a given set of quadrics, Berberich et al. [12] presented an exact, complete and efficient implementation of computing the two-dimensional arrangement induced by their intersection curves on the surface of each quadric. The main difficulty in computing arrangements of quadrics exactly is that irrational algebraic numbers are involved, even when the quadrics are defined by rational coefficients. In Section 6.1 we provide more details on the work of Berberich et al. One of the results of this work was a quadrics model for the `ArrangementTraits_2` concept, to be used with the CGAL `Arrangement_2` package. In Section 6.2 we describe the extension of this model to the `EnvelopeTraits_3` model. The latter work was conducted together with the author of [12]. Figure 6.1 shows an example of the minimization diagram of ellipsoids computed by our program, where curves with degree four can be seen.

6.1 Background and Terminology

In this section we briefly describe the work of Berberich et al. [12]. Let $\mathcal{Q} = \{q_1, \ ... \ q_n\}$ be a set of n quadrics. The intersection curves $q_1 \cap q_i, 2 \leq i \leq n$ induce a two-dimensional arrangement on the surface of q_1. The algorithm for constructing this arrangement is based on the cylindrical algebraic decomposition method. The intersection curves as well as the silhouette of q_1 are projected onto the xy-plane. The projection onto the xy-plane of intersection curves of pairs of quadrics are called *cut-curves*; the projections onto the xy-plane of the silhouettes of the quadrics are called *silhouette-curves*. Cut-curves are bivariate polynomial with algebraic degree at most four, whereas silhouette-curves are bivariate polynomials with algebraic degree at most two. The topology of all curves and all pairs of curves are analyzed in a y-per-x view. This means looking at the vertical line

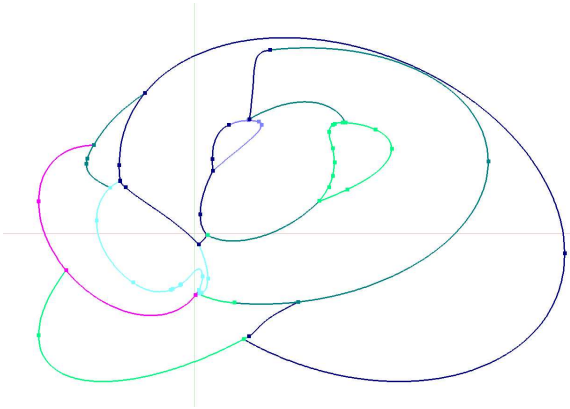


Figure 6.1: Lower envelope of 7 ellipsoids computed by our software, where curves with degree four can be seen.

$x = x_0$ and considering how the arcs of the curves evolve when moving x_0 along the x-axis. When considering one curve, there is only a finite number of events where the number and relative position of the arcs of the curve changes. These events are x-extreme points, singularities and vertical asymptotes. For a pair of curves, the number and relative position of the arcs of both curves changes at a finite number of events, which are the intersection points of the two curves as well as the critical events of each curve separately. A curve and a curve pair are analyzed over each of their relevant x-critical points, and over the intervals between two successive such points. Each curve is split at its event points (and points co-vertical to them) into segments which are x-monotone and smooth. The arrangement is built from these segments. In this method the representation of a planar point is as follows. The x-coordinate is represented explicitly and the y-coordinate is represented implicitly by a curve and the index of the curve arc on which the point lies.

In the projection of the three-dimensional curves onto the xy-plane, the spatial information is lost. Branches of curves on the upper and on the lower part of q_1 may intersect in the projection, even when a three-dimensional intersection does not exist. A method is given to regain this information, by first cutting a cut-curve of q_1 and q_i into segments that belong completely to the upper or to the lower part of q_1, and then determining to which part each such segment belongs. For more details see [12].

6.2 Implementation of the Traits Operations

The work of [12] has led to a quadrics model for the `ArrangementTraits_2` concept. We describe here only the extension of this model to a model of the `EnvelopeTraits_3` concept.

Recall that we have to define two additional types, and few operations involving these types. The two types, which represent general surfaces and xy-monotone surfaces, are both mapped to the same quadric type. This may be surprising at first, since a quadric in general is not xy-monotone, but it is only an implementation detail to simplify matters.

All the operations that work on xy-monotone surfaces consider the quadric object as the appropriate lower or upper part of that quadric.

We now describe the implementation of the `EnvelopeTraits_3` concept operations for quadrics.

1. *Extract xy-monotone surfaces from a general surface.*
   This operation is trivial according to our type choice.

2. *Construct the planar curves that form the boundary of the vertical projection of an xy-monotone surface onto the xy-plane.*
   This operation returns all the segments of the silhouette-curve of the input quadric.

3. *Construct the planar curves that form the projection onto the xy-plane of the intersection between two xy-monotone surfaces.*
   This operation is carried out as follows: First, find the cut-curve of the two full quadrics. Then, find only the segments of the cut-curve, which lie on both quadrics lower (upper) part, when computing the lower (upper) envelope. For the second task, the cut-curve is cut into segments in its intersection points with the silhouette-curve of the first quadric as well as with the silhouette-curve of the second quadric. Each of these segments belongs completely to the lower or to the upper part of each of the two quadrics. It remains to check for every segment to which part of the first quadric and to which part of the second quadric it belongs, and return only the segments that belong to the desired part of both quadrics.

4. *Compare two xy-monotone surfaces s_1 and s_2 immediately above (below) a given two-dimensional x-monotone curve c, which is part of their projected intersection.*
   This operation is implemented as follows: A point $p = (x_0, y_0)$, with rational coordinates x_0 and y_0 is chosen in the x-range of the x-monotone curve c above (below) c, such that the vertical segment starting at p and ending on c does not intersect the silhouette-curve of s_1, the silhouette-curve of s_2 or any other part of the cut-curve of s_1 and s_2. These conditions imply that the surfaces s_1 and s_2 are defined over p, and comparing them over p gives the desired result. The comparison is conducted by substituting the coordinates of p in both quadrics' equations, to get quadratic equations for the z-coordinate, solving these equations to find the appropriate z-coordinate of the point that belongs to the lower (or upper) part of the quadrics, and comparing the z-coordinates of the two quadrics to find their envelope order.

5. *Compare two xy-monotone surfaces s_1 and s_2 over the interior of a given two-dimensional x-monotone curve c.*
   This operation is implemented in the following manner: A point p, with rational x-coordinate in the x-range of c is chosen, such that p satisfies one of the following:

   (a) p lies on c, and its y-coordinate is a rational number, or an algebraic number with degree not greater than two.

(b) p's y-coordinate is a rational number, and p lies above (below) c, such that the vertical segment starting at p and ending on c does not intersect the silhouette-curve of s_1, the silhouette-curve of s_2 or the cut-curve of s_1 and s_2. The latter condition ensures that both s_1 and s_2 are defined over p, and their envelope order over p equals their envelope order over c.

When c is a segment of the silhouette-curve of s_1 or s_2, it is easy to find a point that satisfies Condition 5a. Thus, if it is hard to find a point p that satisfies Condition 5a, it is always possible to choose a point p that satisfies Condition 5b. After computing p, the comparison of the surfaces over it is done in the same way described in Operation 4.

6. *Compare two xy-monotone surfaces s_1 and s_2 over a given two-dimensional point p.* The implementation of this operation is the most complicated, since when two surfaces are to be compared over a point, there is no guarantee that a nearby point p' exists, which has "nicer" coordinates, and comparing the surfaces over p' gives the same answer as comparing them over p. On the contrary, since this method is used by the algorithm only in degenerate situations, most of the times, a "nice" point p' will not exist. This observation forces the implementation of this operation to exactly compare the surfaces over p, the coordinates of which may have higher algebraic complexity than the points considered in the previous operations. Fortunately, in the case of the divide-and-conquer algorithm for computing the envelope of surfaces, there are mitigating circumstances — some assumptions can be made on the given point p, so it is not necessary to implement the method for all possible input points. Considering all the possible cases in the merge step where the comparison method over a point is invoked leads to the important observation that due to the exploitation of the continuity of the envelopes to carry a decision over between incident features (see Chapter 5), the method is invoked only when the point p is an isolated point (of a curve) or it lies on a silhouette-curve. This observation actually means that the algebraic complexity of p is not the highest that is possible in the process of the algorithm. The x-coordinate of p may be of algebraic degree up to eight, whereas in the general case, it may be of size sixteen. The y-coordinate of p may be found by solving a quadratic equation, whereas in the general case, there may be a need to solve a polynomial of degree four. The implementation of the current operation finds the x and y coordinates of p and uses them to compare the quadrics over p as is done in the previous operations. When the x-coordinate of p is an algebraic number of degree between three and eight, finding the x-coordinate of p involves using the root-of operator of the algebraic number type that is used (from the library of CORE or LEDA). The comparison of the quadrics' z-coordinates in this case involves high-degree algebraic numbers and is the most expensive part of the operation, especially, when the numbers are equal.

The quadrics traits class was successfully implemented and used with input sets consisting of up to 1000 ellipsoids. See Chapter 7 for experimental results. We remark that

we used only ellipsoids in the input sets and not other types of quadrics, since the current implementation of the minimization diagram data structure, and of the envelope algorithm support only bounded curves and surfaces. Work is underway to extend the implementation to unbounded objects as well. More details on the quadrics traits class can be found in [13].

Chapter 7

Experimental Results

In this chapter we present experimental results that demonstrate the performance of our algorithm and show its behavior on various input sets. In Section 7.1 we describe the input sets that we used. In Section 7.2 we present the running-time of the algorithm on various input sets and of varying size. Section 7.3 examines the size of the lower envelope of some of our inputs. We give statistics on the running time, output size and process details of different large input sets in Section 7.4, and in Section 7.5 the breakdown of the running time is shown. In Section 7.6 we show the influence of saving algebraic computation on the performance of our algorithm. In Section 7.7 we discuss the issue of performing vertical decomposition in practice. We summarize the results in Section 7.8.

The running times reported in this chapter were obtained on a 3 GHz Pentium IV machine with 2 Gb of RAM, running under Linux. The software was compiled using the Gnu C++ compiler (`g++` version 3.3.2). We use CGAL version 3.2. For the exact rational number-type we use GMP version 4.1.4. We use CORE version 1.7 for exact algebraic numbers, and the EXACUS internal version for the quadrics traits implementation. We also use the libraries of BOOST GRAPH (version 1.31) for the breadth-first search graph algorithm, and QT (version 3.3.4) for visualization.

Whenever a result is presented for triangles, it is the average over thirty input sets. Whenever a result is presented for spheres/ellipsoids it is the average over ten input sets.

7.1 Input Sets

We describe here the input sets that we used in the experiments; some of them are illustrated in Figure 7.1.

- `rnd_triangles_`n n triangles, each of which was generated by choosing the coordinates of its three vertices uniformly at random as integers in the cube $[0, 10000]^3$.

- `rnd_small_`p`_triangles_`n n triangles, each of which was generated by first choosing the coordinates of one corner of the triangle uniformly at random in the cube

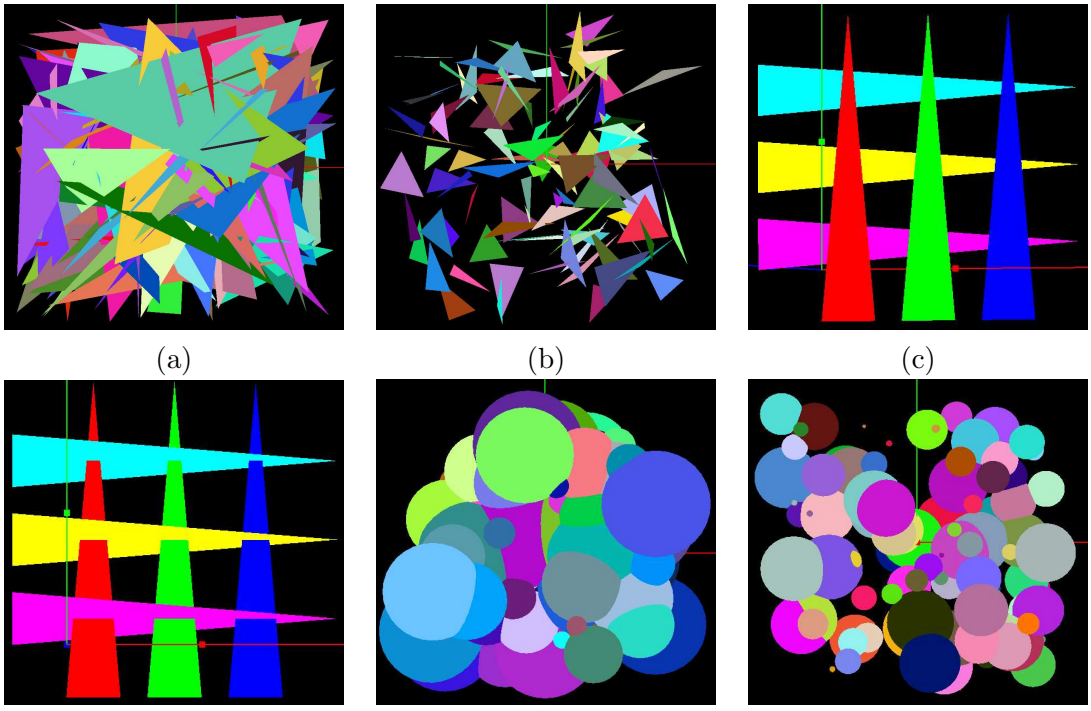


Figure 7.1: Some input files: (a) `rnd_triangles_100`, (b) `rnd_small_0.5_triangles_100`, (c) `grid_triangles_disjoint_6`, (d) `grid_triangles_intersect_6`, (e) `rnd_spheres_100`, and (f) `rnd_small_spheres_100`.

$[0, 10000]^3$, then choosing two random points in the sphere with radius $p * 10000$ around this point. All the vertex coordinates are integers.

- `grid_triangles_disjoint_`n n narrow, long, pairwise disjoint triangles, half of which lie on the plane $z = 0$, and the other half lie on the plane $z = 1$. The projections of these triangles on the xy-plane form an arrangement of size $\Theta(n^2)$, and the lower envelope is of size $\Theta(n^2)$.

- `grid_triangles_intersect_`n n narrow, long triangles, each of which intersects with exactly half of the triangles. The projections of these triangles is as in the above example, and the lower envelope is of size $\Theta(n^2)$.

- `rnd_small_spheres_`n n spheres, where the centers were chosen with integer coordinates uniformly at random in the range $[-1000, 1000]^3$, and the integer radii were chosen uniformly at random in the range $[1, 250]$.

- `rnd_spheres_`n n spheres, where the centers were chosen with integer coordinates uniformly at random in the range $[-1000, 1000]^3$, and the integer radii were chosen uniformly at random in the range $[1, 500]$.

- `rnd_ellipsoids_`n n ellipsoids, each of which was generated by choosing ten random coefficients of the quadratic equation

$$Ax^2 + Bxy + Cxz + Dy^2 + Eyz + Fz^2 + Gx + Hy + Kz + L = 0$$

and checking whether an ellipsoid was created. All coefficient are ten-bit integers. Let $X = (x, y, z, 1)$ and Q be the symmetric 4×4 matrix:

$$Q = \begin{pmatrix} 2A & B & C & G \\ B & 2D & E & H \\ C & E & 2F & K \\ G & H & K & 2L \end{pmatrix}.$$

Then the quadratic equation above can be rewritten as $X^T Q X = 0$. Let Q_u denote the upper left 3×3 matrix of Q. The quadric type that is represented by the equation is determined by I_Q, the inertia[1] of Q, and I_{Q_u}, the inertia of Q_u; see [43] for more information. The quadratic equation represents an ellipsoid if $I_Q = (3, 1, 0)$ and $I_{Q_u} = (3, 0, 0)$.

7.1.1 Degenerate Input Sets

In addition to the input sets described above, we used a few degenerate input sets in some of the experiments:

- `degenerate_triangles_`n a set of n triangles in the cube $[0, 10000]^3$ with degeneracies. The set is composed of 20% pairs of triangles with one common vertex, 20% pairs of triangles with one common edge, 20% pairs of triangles where one vertex of the first triangle touches the interior of one edge of the second triangles, 20% pairs of triangles where one vertex of the first triangle touches the interior of the second triangle and 20% pairs of overlapping triangles.

- `degenerate_spheres_`n a set of n spheres with degeneracies. The set is composed of pairs of spheres; 50% are pairs of tangent spheres that touch from the outside, where their projection is two tangent circles, and the other 50% are pairs of spheres that are tangent in their lowest point (one spheres inside the other). For each pair of spheres, the center of the first sphere was chosen with integer coordinates uniformly at random in the range $[-1000, 1000]^3$, and the integer radius was chosen uniformly at random in the range $[1, 500]$. The second sphere in each pair was calculated from the first one.

[1] The inertia I_A of a matrix A is defined as the triple containing the numbers of positive, negative and zero eigenvalues of A.

7.2 Running Time Behavior

We measured the running time of our algorithm on various types of examples, to investigate the behavior of the algorithm in practice. The results are shown in Figures 7.2 and 7.3. Our results show that the algorithm performs better than the worst-case estimate on some inputs sets. On the grid-like inputs, it obviously cannot run in sub-quadratic time, which is the size of the output.

In Figure 7.4 we show the results of comparing the running time for the triangles input, with two geometric kernels.[2] The first one is a kernel which uses an exact rational number type. The second kernel uses arithmetic filters based on interval arithmetic [17] on the same exact rational number type: only when the filter fails, in degenerate or near degenerate situations, the exact computation is invoked. When the input file contains no degeneracies, it is expected that using an arithmetic filter will improve the performance, since the computation of the filter is less expensive than the exact computation. However, when there are too many filter failures, the running time may be slower than directly conducting exact computation, because of the redundant tests. On our input sets, both the random and the grid-like, there are not many degenerate situations during the running of the algorithm. It can be seen that using the filtered kernel reduces the running time for these input sets by approximately a factor of two.

7.3 Size of the Output

We give statistics of the output size for various types of examples, to investigate the behavior of the size of the lower envelope in practice. Figures 7.5 and 7.6 relate to the `rnd_triangles_`n input sets. It can be seen that the size of the minimization diagram and the number of triangles that appear on it is roughly[3] $\Theta(n^{2/3})$. Figures 7.7 and 7.8 show results for other input sets, which are sub-linear in the input size as well.

7.4 Comparing Different Input Sets

In Figure 7.9 we report on the running time of the algorithm and the size of the minimization diagram for the `rnd_small_`p`_trianlges_1000` input sets, for different values of p. The effect of the input triangles' size can be seen. As could be expected, the running time increases as the size of the triangles increases. This is because smaller triangles have less interactions with each other, both in the two-dimensional projection and in three-dimensions, than bigger triangles do. Thus, the algorithm has much work to do when

[2] Recall from Section 2.4.3 that a geometric kernel defines constant size geometric objects and operations on them. A geometric kernel also encapsulates the number type that is used for numerical operations on the data.

[3] This bound of roughly $\Theta(n^{2/3})$ is inspired by recent results of Alon et al. [6] for the size of the outer face in the arrangement of random segments in the plane. It seems that some of their results extend to 3D implying this bound.

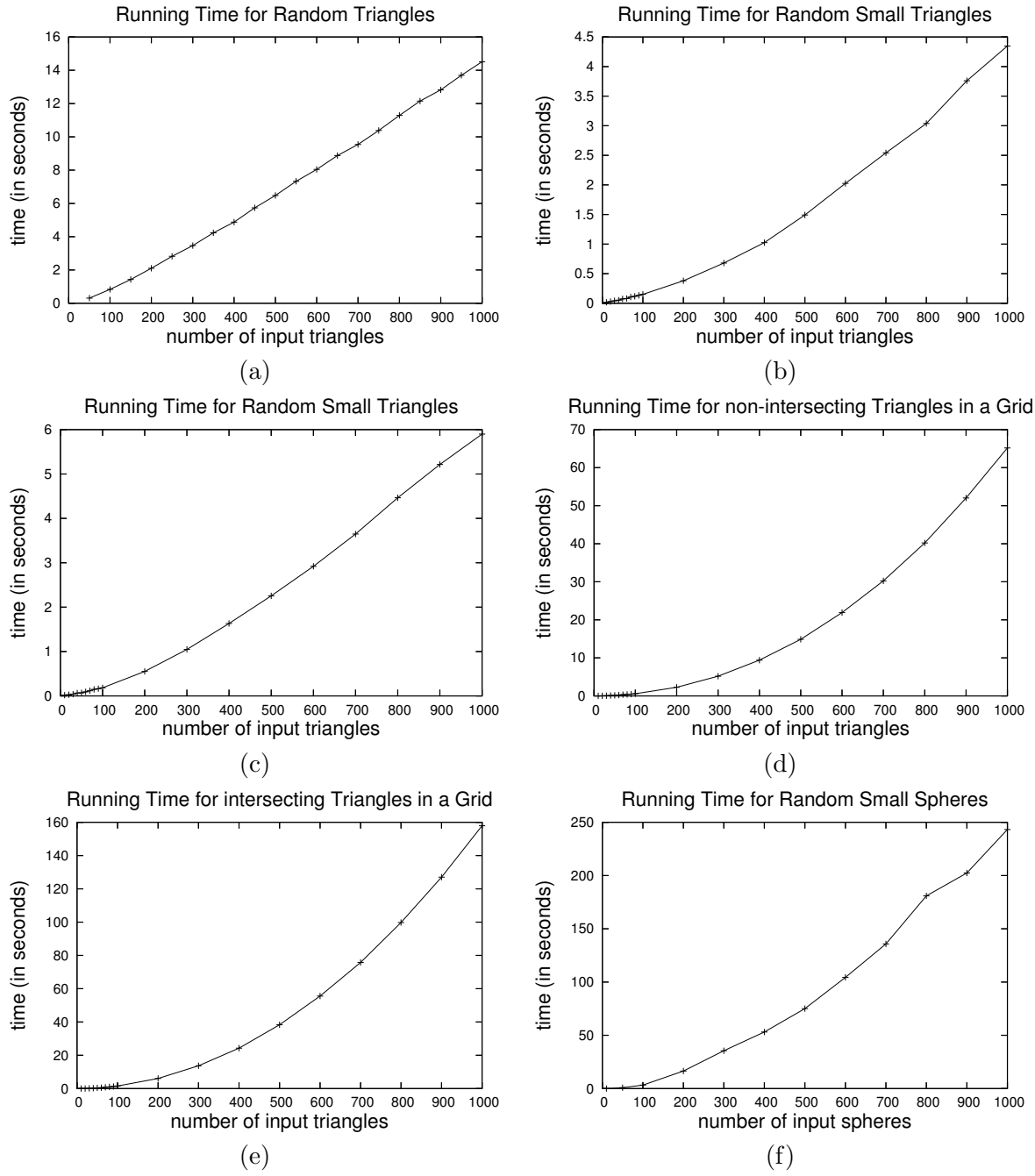
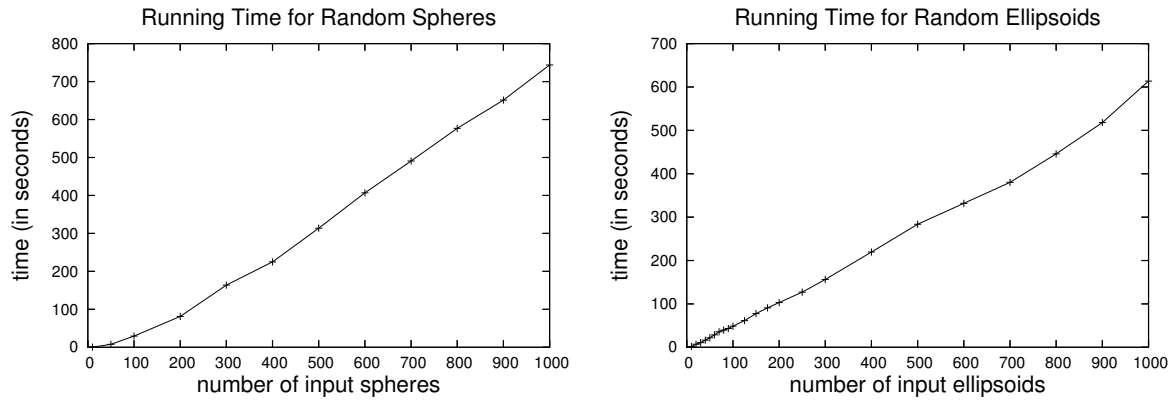


Figure 7.2: The running time of computing the envelope of different input sets for different input sizes: (a) `rnd_triangles_`n, (b) `rnd_small_0.3_triangles_`n, (c) `rnd_small_0.5_triangles_`n, (d) `grid_triangles_disjoint_`n, (e) `grid_triangles_intersect_`n, and (f) `rnd_small_spheres_`n.



(a) (b)

Figure 7.3: The running time of computing the envelope of different input sets for different input sizes: (a) `rnd_spheres_`n, (b) `rnd_ellipsoids_`n.

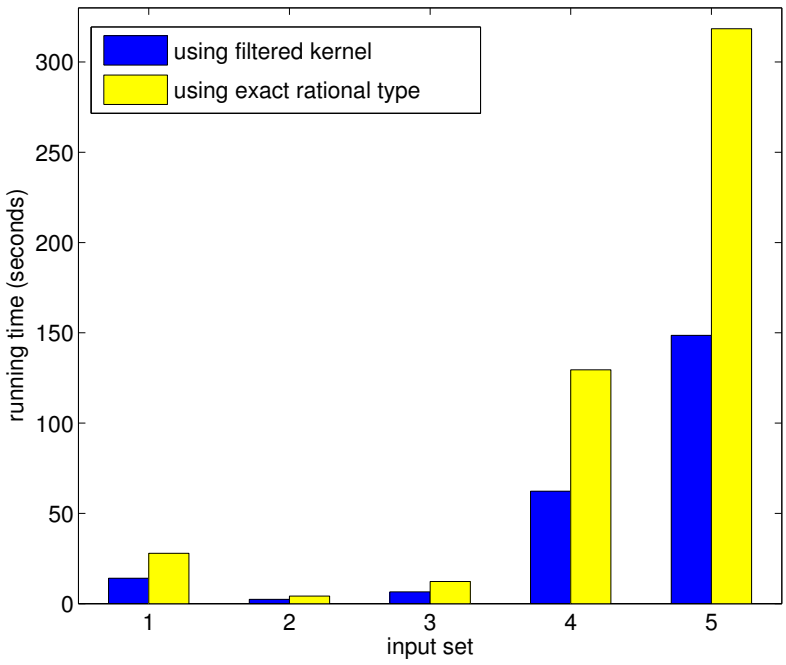


Figure 7.4: The running time of the algorithm using two geometric kernels, for triangles input sets with 1000 triangles. The input sets are: (1) rnd_triangles, (2) rnd_small_0.1_triangles, (3) rnd_small_0.5_triangles, (4) grid_triangles_disjoint, and (5) grid_triangles_intersect.

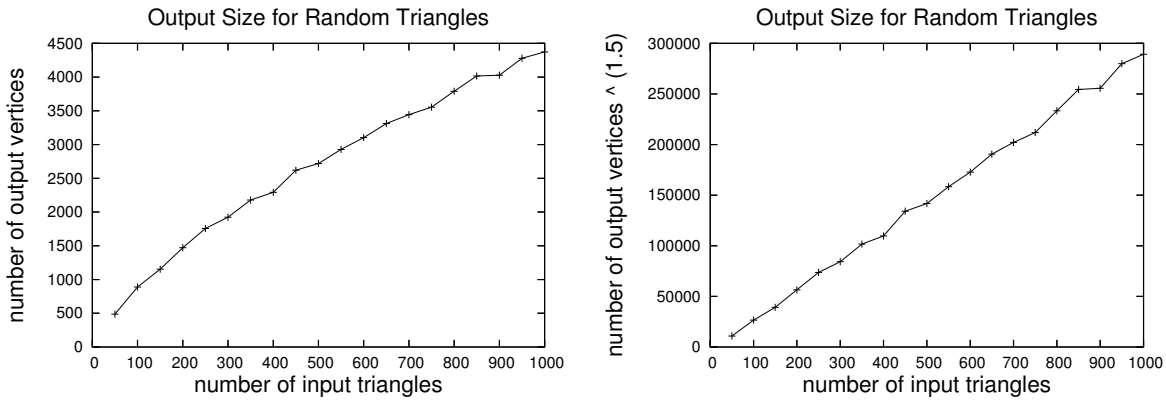


Figure 7.5: (a) the number of vertices in the minimization diagram of random triangles for different input sizes, (b) the number of vertices powered to 1.5.

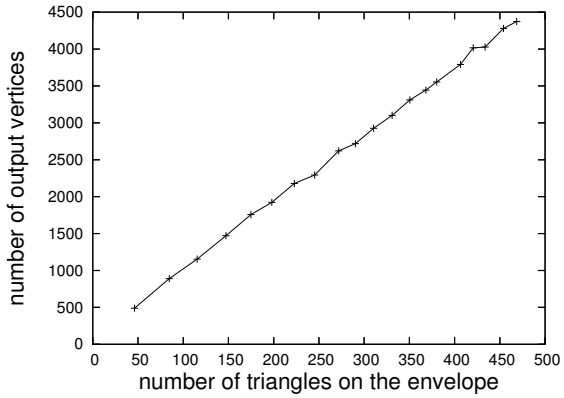


Figure 7.6: The number of vertices in the minimization diagram of random triangles as a function of the number of triangles that appear on the lower envelope.

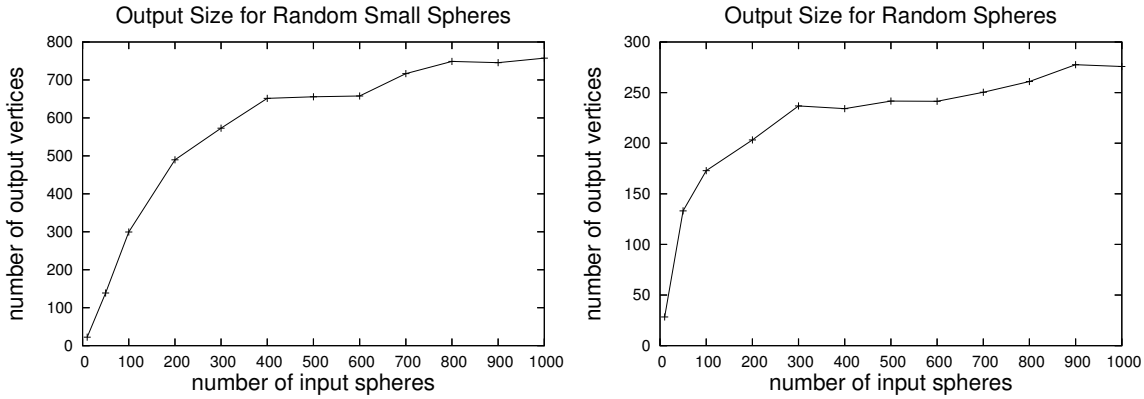


Figure 7.7: The number of vertices in the minimization diagram of the `rnd_small_spheres` input sets (on the left) and `rnd_spheres` input sets (on the right) for different input sizes.

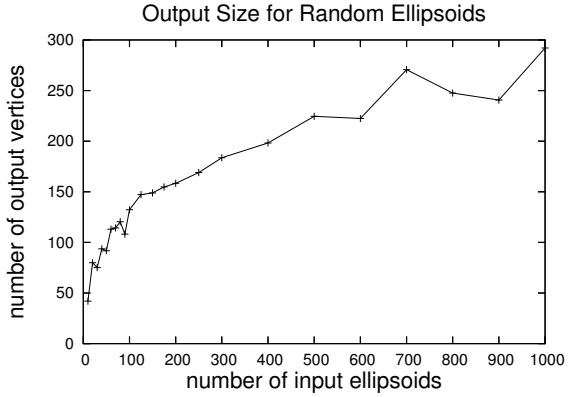


Figure 7.8: The number of vertices in the minimization diagram of the `rnd_ellipsoids` input sets for different input sizes.

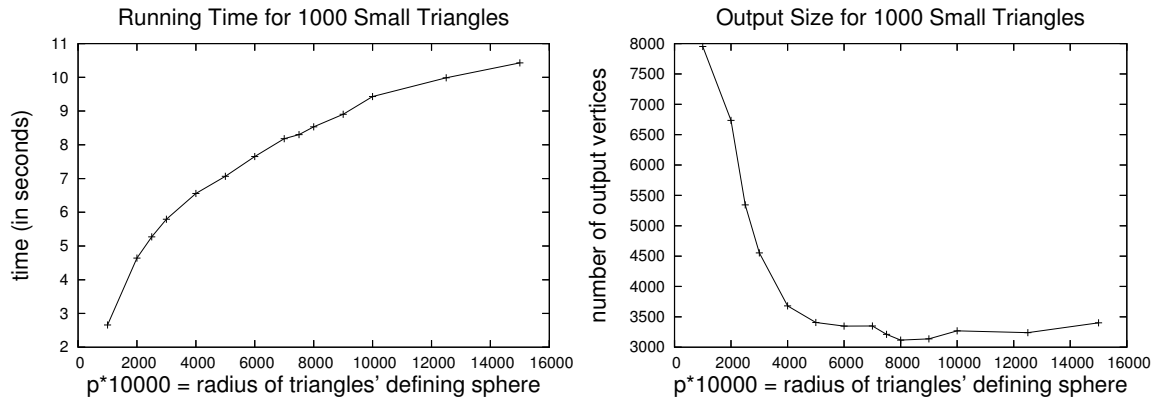


Figure 7.9: (a) The running time of our algorithm on `rnd_small_`p`_triangles_1000` input sets for different values of p. (b) The minimization diagram size for `rnd_small_`p`_triangles_1000` input sets for different values of p.

the triangles are bigger. The graph of the output size is explained as follows: when the triangles are tiny, most of them appear on the envelope, and so the minimization diagram is big; as the triangles grow bigger, they start to hide each other, such that fewer of them appear on the envelope, and the envelope size becomes smaller.

Table 7.1 shows the actual running time for different input sets, each consisting of 1000 input surfaces. In the last three columns we give statistics of the whole process — the total sum of the sizes of all the minimization diagrams computed during the recursion, the number of intersections between pairs of surfaces that were found by the algorithm and the number of two-dimensional intersections between projected curves that were found during the algorithmic process. We show these quantities since they can give an idea about the amount of work that is carried out during the whole execution. Table 7.2 shows statistics on the size of the minimization diagram: the number of surfaces that appear on the envelope and the number of vertices, edges and faces in the minimization diagram. As the algorithm is not output-sensitive, this table does not, in general, reflect the amount of work carried out by the algorithm. However, it is an interesting example of the huge variance in output size for the same (combinatorial) input size. It can be seen that the algorithm is much slower when run on non-linear input than on linear input; this is expected when using exact arithmetic, since with linear input, rational arithmetic suffices, whereas with non-linear input, algebraic numbers should be used.

7.5 Breakdown of the Running Time

In Figures 7.10 and 7.11 we present the breakdown of the running time of the algorithm by steps. It can be seen that the two most costly steps are the overlay and the resolving of the features. This is not surprising since these steps use the two intersection operations, for xy-monotone surfaces in three-dimensions and for x-monotone curves in the plane, which are

Table 7.1: Results for different input sets; times are measured in seconds. Each input set contains 1000 surfaces. *Interm. features* is the total sum of the combinatorial size of all the minimization diagrams computed during the recursion. *Intersections* is the number of intersections between pairs of surfaces that were found by the algorithm. *2d-Intersections* is the number of two-dimensional intersections between x-monotone curves that were found during the entire run of the algorithm.

Input File	Time in Seconds	Process details		
		Interm. features	Intersections	2d-Intersections
rnd_triangles	14.0739	190,942	12,007	52,990
rnd_small_0.1_triangles	2.36964	94,906	117	11,134
rnd_small_0.5_triangles	6.53201	144,383	2,676	29,093
grid_triangles_intersect	148.621	71,830	250,000	1,502,500
grid_triangles_disjoint	62.3065	71,830	0	1,002,500
degenerate_triangles	13.5728	182,815	11,373	51,106
rnd_small_spheres	249.111	60,465	842	7,472
rnd_spheres	654.044	53,188	1,565	8,547
degenerate_spheres	610.27	53,001	1,383	7,891
rnd_ellipsoids	581.138	60,357	4,945	15,915

Table 7.2: Lower envelope statistics for different input sets. Each input set contains 1000 surfaces. *Surfaces on envelope* is the number of surfaces that appear on the envelope. *V, E, F* are the number of vertices, edges and faces in the minimization diagram.

Input File	Output size			
	Surfaces on envelope	V	E	F
rnd_triangles	482	4,899	6,632	1,764
rnd_small_0.1_triangles	992	8,072	11,061	3,012
rnd_small_0.5_triangles	413	3,339	4,561	1,237
grid_triangles_intersect	1000	1,503,000	2,003,000	500,002
grid_triangles_disjoint	1000	1,003,000	1,503,000	500,002
degenerate_triangles	441	4,336	5,919	1,606
rnd_small_spheres	207	712	925	220
rnd_spheres	100	305	405	102
degenerate_spheres	100	281	359	92
rnd_ellipsoids	75	269	344	77

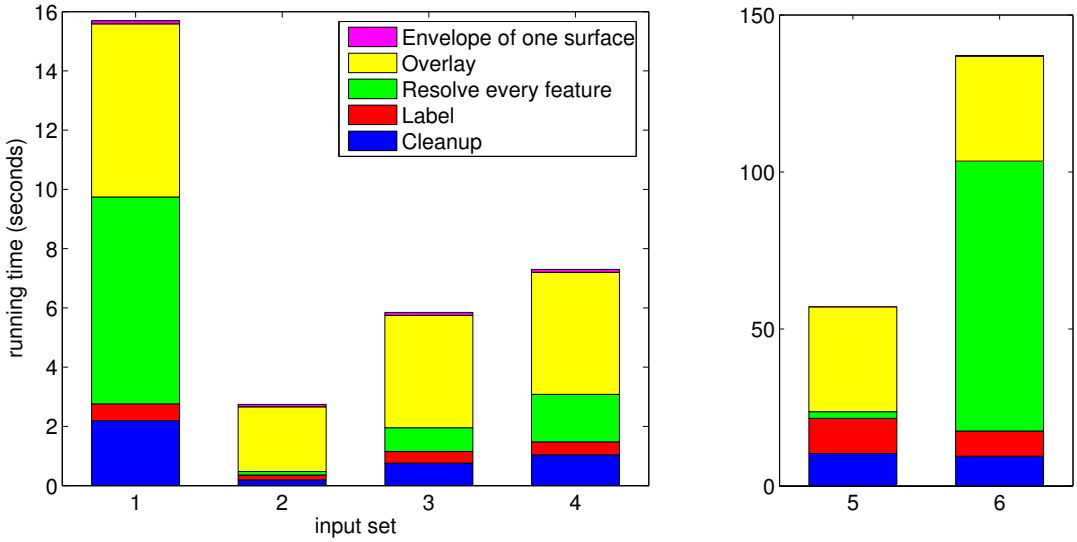


Figure 7.10: The running time of the algorithm by its steps, for triangles input sets, each consisting of 1000 triangles. The input sets are: (1) rnd_triangles, (2) rnd_small_0.1_triangles, (3) rnd_small_0.3_triangles, (4) rnd_small_0.5_triangles, (5) grid_triangles_disjoint, and (6) grid_triangles_intersect.

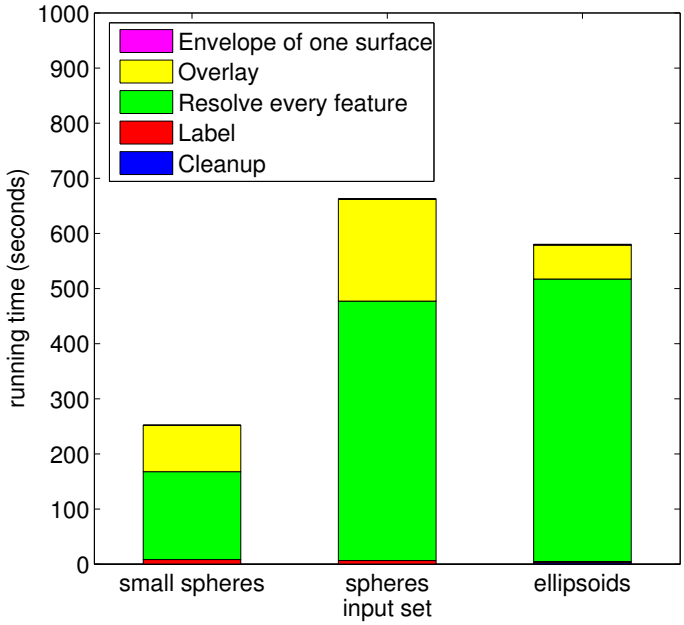


Figure 7.11: The running time of the algorithm by its steps, for different input sets, each consisting of 1000 spheres/ellipsoids.

expected to be the most costly among the geometric operations, since they construct new geometric objects. Note how much this issue is significant when working with non-linear objects (and irrational computation).

7.6 The Effect of Saving Algebraic Computation

Table 7.3 shows the number of calls to the three types of comparison methods made by the algorithm in the labelling process, comparing between the naïve approach and our approach. Table 7.4 shows the algorithm running time, comparing between the same two approaches. The naïve approach means comparing surfaces over all features, except over projected intersections. Our approach is described in Chapter 5, and uses the intersection type and continuity/discontinuity information together with a breadth-first traversal of the faces. Both approaches use the caching information described in Chapter 4. Note that a comparison could not be made on quadrics input data since the comparison over a point operation is not implemented for the general case, so the computation of envelopes of quadrics is only available using our approach. It can be seen that the reduction in the number of operations is highly significant for all the input sets. The number of comparison operations over a two-dimensional point reduces to zero in our approach in all the examples that do not contain degeneracies in which this operation is invoked. We remark that the improvement in the running time is not gained only by the reduction in the three-dimensional comparison operations. Recall that we can only label features after they have been resolved. When it is possible to carry a decision made over a face to its boundary edge, then resolving this edge will not split it. Thus, we can avoid the resolve step for this edge, which means that we avoid unnecessary three-dimensional and two-dimensional costly intersection operations.

7.7 The Issue of Vertical Decomposition

Recall that a vertical decomposition is a subdivision of an arrangement where a vertical ray is extended upwards and downwards from every vertex of the arrangement, until it hits another feature (vertex or edge) of the arrangement, or extends to infinity. A vertical decomposition creates simple faces, with constant number of edges on their boundary, and without holes, and at the same time preserves the asymptotic combinatorial complexity of the arrangement.

In a *partial vertical decomposition* only a subset of the vertical segments defined by the full vertical decomposition are added. A vertical segment is added only from a curve endpoint, not from an intersection point. The partial vertical decomposition creates less new features than the full vertical decomposition. The faces that are created are no longer of constant size, but they do not contain any hole, and their upper and lower curve chain is x-monotone.

In Table 7.5, we summarize experiments computing the minimization diagram of trian-

Table 7.3: Comparing the number of comparison operations used by the algorithm with and without our means for reducing the number of algebraic operations. Each input set contains 1000 surfaces. The Pt., Cv. and Cv.-side columns represent the number of calls made to the appropriate version of the comparison method: comparison over a two-dimensional point, comparison over a two-dimensional x-monotone curve and comparison above/below a two-dimensional x-monotone curve respectively.

	Naïve solution			Using our improvements		
Input File	Pt.	Cv.	Cv.-side	Pt.	Cv.	Cv.-side
rnd_triangles	85,091	166,405	13,715	0	10,333	3,287
rnd_small_0.3_triangles	42,090	76,763	1,934	0	8,244	791
rnd_small_0.5_triangles	51,598	96,662	3,703	0	8,851	1,325
grid_triangles_intersect	1,000,000	1,500,000	500,000	0	0	250,000
grid_triangles_disjoint	1,000,000	1,250,000	0	0	250,000	0
degenerate_triangles	79,660	155,636	13,406	10	9,394	3,187
rnd_small_spheres	14,901	25,853	1297	0	2,466	450
rnd_spheres	14,965	25,630	2247	0	1,840	617
degenerate_spheres	14,141	24,114	2,006	8	1,851	571

Table 7.4: Comparing the running time of the algorithm with and without our means for reducing algebraic operations. Times are measured in seconds.

Input File	Naïve solution	Using our improvements
rnd_triangles	25.0282	14.0739
rnd_small_0.3_triangles	6.86196	5.2632
rnd_small_0.5_triangles	9.59354	6.53201
degenerate_triangles	22.9374	13.5728
grid_triangles_intersect	N/A[a]	148.621
grid_triangles_disjoint	68.9785	62.3065
rnd_small_spheres	399.327	249.111
rnd_spheres	1116.43	654.044
degenerate_spheres	990.578	610.27

[a]The naïve approach does not run on this input. We believe that it consumes more memory than is available on our computer.

Table 7.5: Comparing the running time with and without using the partial vertical decomposition. Times are measured in seconds.

	Using Partial VD			
Input File	Total time	PVD time	Total - PVD	Without VD
rnd_triangles	28.2307	12.1931	16.0376	14.0739
rnd_small_0.1_triangles	8.64469	5.20921	3.43548	2.36964
rnd_small_0.3_triangles	12.8021	6.82096	5.98114	5.2632
rnd_small_0.5_triangles	15.1287	7.6998	7.4289	6.53201
grid_triangles_intersect	207.597	59.234	148.363	148.621
grid_triangles_disjoint	120.757	59.7989	60.9581	62.3065

gles using the partial vertical decomposition and without using a decomposition, namely directly handling complicated faces. The implementation of the partial vertical decomposition is done by inserting the vertical segments into the arrangement as regular curves.

We carried out our experiments only for triangles, because the other traits classes that we have, those that handle spheres or quadrics, are not able to represent the vertical segments that are created by the decomposition. Both traits represent only two-dimensional curves which are algebraic functions with rational coefficients. Since a vertex in the arrangement of such curves might represent a point with irrational algebraic x-coordinate x_0, the line $x = x_0$ may not have rational coefficients, and a vertical segment with this vertex as an endpoint cannot be created.

In theory, vertical decomposition is performed on the overlaid arrangement in order to get constant size faces, leading to constant amount of work invested at each face, and a running time proportional to the size of the arrangement of the merge step. In practice, though, the full vertical decomposition seems to be an overkill. Even the partial vertical decomposition has a little overhead, as demonstrated by the experiments. But in this case, since a face does not contain holes, the asymptotic combinatorial complexity of the algorithm can be assured. The overhead in the running time of the decomposition stems mainly from the additional geometric constructions that are computed to form the vertical segments, and by intersecting the surfaces' projected intersections with them. Such constructions are known to be costly when using exact computation.

We remark that the presented results may not be the final word about the effectiveness of vertical decomposition in practice. We suggest the following improvements of the implementation. First, in our implementation, the decomposition is done on the whole arrangement, including the unbounded face, and other faces where the label can be trivially determined. This may be wasteful, and it may be desirable to perform the decomposition only in faces of the overlay that will be resolved, and contain holes. Secondly, our implementation inserts the vertical segments into the arrangement as regular curves/edges. It may be better to treat them as special curves/edges, only topologically, not geometrically. By this we mean that the topology of the arrangement with the decomposition will be

known, but the coordinates of the split points (which are supposed to disappear in the cleanup step) will not be computed exactly.

7.8 Summary of the Experimental Results

In this chapter we presented the results of computing the lower envelope of three families of surfaces: triangles, spheres and quadrics using our algorithm. We showed that the algorithm performs well in terms of running-time, and for some input sets, it even performs better than the worst-case bound predicts. We measured the size of the lower envelope in our experiments, and found out that for some input sets, this size is sub-linear, which is significantly smaller than the theoretical bound. We used our algorithm with various input sets consisting of 1000 surfaces each, and saw the huge variance in the output size for the same combinatorial input size. In addition, we found out that as expected, the algorithm performs better when run on linear input (using rational arithmetic) than when run on non-linear input (using algebraic numbers). We demonstrated the significant reduction in the number of geometric operation carried out by our algorithm, yielding a significant improvement in the algorithm's performance. Finally, we discussed the issue of performing a vertical decomposition vs. handling a complex face, and concluded that currently the latter is preferable.

Chapter 8

Conclusions and Future Work

We presented a generic, robust and efficient implementation of the divide-and-conquer algorithm for computing the envelope of three-dimensional surfaces. To our knowledge, it is the first such implementation. Our implementation uses a geometric traits class to separate the topology and geometry. This separation allows for the reuse of the algorithm for different families of surfaces. We provide three traits classes that handle a set of triangles, a set of spheres and a set of quadrics in $\mathbb{R}^3$. We reported the results of our experiments showing that our algorithm performs well on various input sets.

We propose several directions for further research:

1. **Voronoi diagrams**. As explained in Section 2.2.2, every two-dimensional Voronoi diagram can be seen as a minimization diagram of three-dimensional surfaces. It is possible to use our algorithm to compute the Voronoi diagram of a set of planar objects, provided that an appropriate traits class is supplied. The two-dimensional geometric objects and operations are those that build the planar subdivision of the Voronoi diagram. The three-dimensional objects and operations involve surfaces, which are added to the problem artificially, and may increase the algebraic complexity of the problem. An important observation here is that the artificial surfaces need not be represented explicitly. Rather, they can be represented by the two-dimensional sites they stand for. The projected intersection operation is interpreted as the bisector between the Voronoi cells of the two corresponding sites. The different comparison operations between two surfaces are interpreted as a comparison of the distances to the two respective sites.

2. **A "lazy" arithmetic scheme**. In a lazy scheme, exact computation is postponed until it is actually needed. Interval arithmetic is used to estimate the values of expressions on the input geometric objects as well as on new constructions. These estimations are used to filter out situation where exact computation is not needed. Only when a filter fails, that is the estimation cannot guarantee a correct answer, the exact arithmetic is used. A work on such a lazy kernel for CGAL is in progress [28]. At the moment, it supports two-dimensional objects only. The experiments

on the CGAL arrangement package [67], and our experiments that show a better performance of the currently available filtered kernel over a non-filtered kernel make us believe that using this lazy kernel (when it includes a support for objects in three-dimensions) can improve the performance of the envelope computation, at least for some input sets, which do not contain many degeneracies.

3. **A special traits class for polyhedral surfaces**. Polyhedral surfaces are a very useful type of surfaces in many applications, since they can represent approximations of other, more complicated surfaces. If the polyhedral surfaces are triangulated, it is possible to use our algorithm with our triangles traits class, to compute their envelope. However, some assumptions can be made on the input in the case of polyhedral surfaces, which we believe, can be exploit to yield much better performance. Finding the intersection between two triangles is usually a costly operation. Adjacent triangles on the surface share a common edge, which is their intersection. Thus, in this case, the primary knowledge of the adjacency relation of the triangles can make the intersection operation trivial. This information can be useful for handling the triangles in three-space and their projections as well, since the overlay procedure may work hard to glue up projections of adjacent triangles from a single surface. Moreover, the minimization diagram of an entire xy-monotone polyhedral surface is known trivially, thus, it can be computed in the bottom of the recursion, which in turn will lead to less recursive steps. This can save a lot of unnecessary computation, especially in the overlay procedure.

4. **Support for unbounded surfaces**. The current implementation of the only class that can represent a minimization diagram, namely the CGAL Arrangement_2 class, supports only bounded curves, and one unbounded face in every arrangement. As a result, the algorithm for computing the minimization diagram supports only bounded surfaces. The need for unbounded curves shows up already when dealing with quadrics, and exists also in computing Voronoi diagrams. We believe that a fundamental solution to the topological structure of the arrangement will enable an easy support for unbounded surfaces in the divide-and-conquer algorithm for computing envelopes in three-dimensional space.

5. **The complexity of the minimization diagram and of the divide-and-conquer algorithm of random objects**. Our experiments show that the combinatorial complexity of the minimization diagram and the running-time of the divide-and-conquer algorithm are much less than the worst-case bound, for some types of inputs sets of random objects. This raises the question whether better theoretical bounds can be proved in such cases.

6. **Controlled perturbation for envelopes**. In controlled perturbation, the input set is perturbed slightly to get a new set, which is free of degeneracies, and on which the algorithm using a floating-point number type performs in the same way as if it was using an exact number type, see, for example [34, 38, 40]. The question is how

to perform such a perturbation on a set of triangles and other surfaces in order to compute their lower envelope.

7. **Sandwich Regions**. A sandwich region consists of all points that lie above the upper envelope of one set of surfaces and below the lower envelope of another set of surfaces. An example application of sandwich regions is given in Section 2.2.4. Given a lower envelope $\mathcal{L}$ of one set of surfaces and an upper envelope $\mathcal{U}$ of another set of surfaces, computing the sandwich region of $\mathcal{L}$ and $\mathcal{U}$ can be carried out in a way similar to the merging of two envelopes in the divide-and-conquer algorithm. There are a few differences though; for example, the representation of the result differs from the representation of a minimization diagram in the additional data attached to the arrangement features, the meaning of the comparison methods used in the labelling step is different (though they return a three-valued result in both cases), and the criteria of the cleanup is not the same as well. Making the code slightly more generic, it is possible to reuse it for computing sandwich regions. This can be achieved by introducing a new parameter to the merge algorithm, whose responsibility would be to perform all the result-specific operations.

Appendix A

Program Checking

Programming is a very error-prone task. There are two common ways to find coding errors in programs: program testing and program checking. In program testing, the program is run on inputs for which the output is known by other means, for example, by manually computing it or by using an alternative program for the same task. A program checker verifies that the output of a particular program is correct [14, 52]. When computing envelopes of surfaces in three-dimensional space, the output itself is very complicated, to be computed manually. Moreover, we are not aware of any other program that computes a similar output. In this appendix we describe two methods that we developed in order to check the results of our divide-and-conquer algorithm. Though these methods are not perfect (each with its own drawbacks), they are very helpful in automatically checking the algorithm on many different input sets, with different number of input surfaces, including large examples. These methods are described below.

1. Random shuffle the input, and use the same divide-and-conquer algorithm to create the output minimization diagram. The division of the surfaces' input set will be totally different with each shuffle, and this will result in a completely different algorithm flow. Next, compare two minimization diagrams that were computed for distinct shuffles (and should be equal) as follows. First, compare the number of faces, edges and vertices. Second, overlay the two underlying arrangements, and compare the labels of each pair of overlapping regions of the two arrangements.
   This test does not guarantee the correctness of the minimization diagram, however, we found it very effective in practice. The test is relatively inexpensive, as a result, it is usable on large input sets, and many shuffles can be performed on each input.

2. Construct a refinement of the actual minimization diagram using a simple algorithm, and check the labelling using the same idea described above. We take all the projected boundaries of the input xy-monotone surfaces together with the projected intersections of all possible pairs of xy-monotone surfaces, and build their arrangement $\mathcal{A}_t$. $\mathcal{A}_t$ is a refinement of the underlying arrangement of the minimization diagram. Now we find the correct label for each feature of $\mathcal{A}_t$ in a straightforward way. For vertices, we compare all the relevant surfaces over the related point, using the appropriate

comparison operation supplied by the traits. By relevant surfaces, we mean only surfaces that are actually defined over that point. For edges and faces, we find a point in their interior, and operate in the same way as for points, since for every edge or face in $\mathcal{A}_t$, an xy-monotone surface is fully defined or fully undefined over that feature, and the envelope order of two xy-monotone surface that are defined there is preserved for all the points of the feature. To find a point in the interior of an edge, we simply find a point in the interior of the underlying x-monotone curve it represents. Finding a point inside a face is a little more complicated since the face can have a complicated shape. We take a non-vertical edge on the face boundary, and shoot a vertical ray up or down from a point on (the interior of) that edge, call it the starting point, towards the face interior, until it hits another feature. The vertical segment that is formed between the starting point and the hit point lies totally inside the face, and we can take any point on (the interior of) it, to represent the face.

This simple algorithm is parameterized with a geometric traits, as the divide-and-conquer algorithm does. The traits concept of the test algorithm is a refinement of the traits concept needed for the divide-and-conquer algorithm. We describe here the additional traits requirement for the test algorithm:

(a) Given xy-monotone surface s and a two-dimensional point p, determine whether p is part of the planar definition domain of s. This operation is used for checking whether an xy-monotone surface is to be compared over a feature that contains p.

(b) Given planar x-monotone curve, construct a point in its interior. This operation is used for finding a point in the interior of an edge and inside a face.

(c) Given a planar x-monotone curve c and a point p, defined in the x-range of c, construct the point on c with the same x-coordinate as p. This operation is used for finding a point inside a face.

The test algorithm described here is very costly in terms of both time and space, since we build an arrangement of $O(n^2)$ curves, where n is the number of input surfaces. This arrangement can thus be of size $\Theta(n^4)$, and for each of its features we check for all the n surfaces whether they are defined over the feature, so we get a time and space bound of $O(n^5)$ (this is also a bound for the working space since we save for each feature a list, which may be of size $\Theta(n)$, of all the surfaces seen on the envelope there; an example can be built similarly to the one in Figure 5.3). Consequently, we can use it for small to medium size input only. Another disadvantage of this test is that it does not check the cleanup step of the algorithm — every refinement of the actual result (which includes only projected boundary and intersection curves), will pass this test.

We remark that the methods described above check only the divide-and-conquer algorithm, assuming correctness of all other code blocks it uses, among them are the geometric

traits-class operations, the number types, the two-dimensional arrangement data-structure and its operations. The geometric traits-class operations are easier to test in the sense that each operation takes constant number of geometric objects and its output is relatively simple, with limited constant size. Thus, it is possible to write predefined tests for each operation, and test all the different cases in their implementation. The code of the CGAL arrangement package was already tested by its authors as part of the CGAL test suite.

In addition to the above methods, we developed a visualization tool, which greatly assisted us in testing the algorithm results, especially for small input sets. We also carefully checked small predefined examples manually.

Bibliography

[1] *CGAL User and Reference Manual, version 3.2*, 2006.

[2] P. K. Agarwal, E. Flato, and D. Halperin. Polygon decomposition for efficient construction of Minkowski sums. *Computational Geometry — Theory and Applications*, 21:39–61, 2002.

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

[4] P. K. Agarwal, O. Schwarzkopf, and M. Sharir. The overlay of lower envelopes and its applications. *Discrete and Computational Geometry*, 15:1–13, 1996.

[5] P. K. Agarwal and M. Sharir. Arrangements and their applications. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 49–119. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[6] N. Alon, D. Halperin, O. Nechushtan, and M. Sharir. The complexity of the outer face in arrangements of random segments. Manuscript, 2006.

[7] M. H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1999.

[8] G. Barequet and V. Rogol. Maximizing the area of an axis-symmetric polygon inscribed by a convex polygon. In *Proc. 16th Canadian Conference on Computational Geometry*, pages 128–131, 2004.

[9] E. Berberich. Exact arrangements of quadric intersection curves. M.Sc. thesis, Universität des Saarlandes, March 2004.

[10] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, and N. Wolpert. Exacus: Efficient and exact algorithms for curves and surfaces. In *Proc. 13th Annual European Symposium on Algorithms*, volume 3669 of *Lecture Notes in Computer Science*, pages 155–166. Springer, 2005.

[11] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schömer. A computational basis for conic arcs and Boolean operations on conic polygons. In *Proc. 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 174–186, 2002.

[12] E. Berberich, M. Hemmer, L. Kettner, E. Schömer, and N. Wolpert. An exact, complete and efficient implementation for computing planar maps of quadric intersection curves. In *Proc. 21st Annual ACM Symposium on Computational Geometry*, pages 99–106, 2005.

[13] E. Berberich and M. Meyerovitch. Computing envelopes of quadrics. In preparation.

[14] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995.

[15] J.-D. Boissonnat and K. T. G. Dobrindt. On-line construction of the upper envelope of triangles and surface patches in three dimensions. *Computational Geometry — Theory and Applications*, 5(6):303–320, 1996.

[16] J.-D. Boissonnat and M. Yvinec. *Algorithmic geometry*. Cambridge University Press, New York, NY, USA, 1998.

[17] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109(1–2):25–47, 2001.

[18] D. Cohen-Or, S. Lev-Yehudi, A. Karol, and A. Tal. Inner-cover of non-convex shapes. *International Journal on Shape Modeling*, **9**(2):223–238, 2003.

[19] M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*, volume 703 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1993.

[20] 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.

[21] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[22] D. A. Duc, N. D. Ha, and L. T. Hang. Proposing a model to store and a method to edit spatial data in topological maps. Technical report, Ho Chi Minh University of Natural Sciences, Ho Chi Minh City, Vietnam, 2001.

[23] L. Dupont, D. Lazard, S. Lazard, and S. Petitjean. Towards the robust intersection of implicit quadrics. In J. Winkler and M. Niranjan, editors, *Workshop on Uncertainty in Geometric Computations 2001*, volume 704 of *International Series in Engineering and Computer Science*, chapter 5, pages 59–68. Kluwer Academic Publishers, Aug 2002.

[24] H. Edelsbrunner. The upper envelope of piecewise linear functions: Tight complexity bounds in higher dimensions. *Discrete and Computational Geometry*, 4:337–343, 1989.

[25] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Discrete and Computational Geometry*, 1:25–44, 1986.

[26] A. Eigenwillig, L. Kettner, E. Schömer, and N. Wolpert. Complete, exact and efficient computations with cubic curves. In *Proc. 20th Annual ACM Symposium on Computational Geometry*, pages 409–418, 2004.

[27] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the Computational Geometry Algorithms Library. *Software — Practice and Experience*, 30(11):1167–1202, 2000.

[28] A. Fabri and S. Pion. Lazy exact computation of geometric constructions. Technical Report ACS-TR-121201-01, INRIA Sophia-Antipolis, 2006.

[29] E. Flato. Robust and efficient construction of planar Minkowski sums. Master's thesis, Department of Computer Science, Tel-Aviv University, 2000.

[30] E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The design and implementation of planar maps in CGAL. *ACM Journal of Experimental Algorithmics*, 5:1–23, 2000.

[31] E. Fogel and D. Halperin. Exact and efficient construction of Minkowski sums of convex polyhedra with applications. In *Proc. 8th Wrkshp. Alg. Eng. Exper. (ALENEX)*, 2006. To appear.

[32] E. Fogel et al. An empirical comparison of software for constructing arrangements of curved arcs. Technical Report ECG-TR-361200-01, Tel-Aviv Univ., 2004.

[33] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice. 2nd Edition in C.* Addison-Wesley, 1996.

[34] S. Funke, C. Klein, K. Mehlhorn, and S. Schmitt. Controlled perturbation for delaunay triangulations. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1047–1056, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[36] N. Geismann, M. Hemmer, and E. Schomer. Computing a 3-dimensional cell in an arrangement of quadrics: exactly and actually! In *Proc. 17th Annual ACM Symposium on Computational Geometry*, pages 264–273, 2001.

[37] D. Halperin. Arrangements. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. Chapman & Hall/CRC, 2nd edition, 2004.

[38] D. Halperin and E. Leiserowitz. Controlled perturbation for arrangements of circles. *International Journal of Computational Geometry and Applications*, 14(4–5):277–310, 2004. Special issue, papers from SoCG 2003.

[39] D. Halperin and M. Sharir. New bounds for lower envelopes in three dimensions, with applications to visibility in terrains. *Discrete and Computational Geometry*, 12:313–326, 1994.

[40] D. Halperin and C. R. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *Computational Geometry — Theory and Applications*, 10:273–287, 1998.

[41] I. Hanniel. The design and implementation of planar arrangements of curves in CGAL. M.Sc. thesis, School of Computer Science, Tel Aviv University, 2000.

[42] I. Hanniel and D. Halperin. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *Proc. 4th International Workshop on Algorithm Engineering*, volume 1982 of *Lecture Notes in Computer Science*, pages 171–182, 2000.

[43] M. Hemmer. Reliable computation of planar and spatial arrangements of quadrics. Master's thesis, Universität des Saarlandes, April 2002.

[44] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proc. 5th International Workshop on Algorithm Engineering*, volume 2141 of *Lecture Notes in Computer Science*, pages 79–90, 2001.

[45] S. Hirsch and D. Halperin. Hybrid motion planning: Coordinating two discs moving among polygonal obstacles in the plane. In J.-D. Boissonnat, J. Burdick, K. Goldberg, and S. Hutchinson, editors, *Algorithmic Foundations of Robotics V*, pages 239–255. Springer, 2003.

[46] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proc. 15th Annual ACM Symposium on Computational Geometry*, pages 351–359. ACM Press, 1999.

[47] M. J. Katz, M. H. Overmars, and M. Sharir. Efficient hidden surface removal for objects with small union size. *Computational Geometry — Theory and Applications*, 2:223–234, 1992.

[48] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th Annual European Symposium on Algorithms*, volume 3221 of *Lecture Notes in Computer Science*, pages 702–713, 2004.

[49] S. Lazard, L. M. Peñaranda, and S. Petitjean. Intersecting quadrics: An efficient and exact implementation. In *Proc. 20th Annual ACM Symposium on Computational Geometry*, pages 419–428, 2004.

[50] P. McMullen. The maximal number of faces of a convex polytope. *Mathematika*, 17:179–184, 1970.

[51] K. Mehlhorn and S. Näher. LEDA*: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[52] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *Computational Geometry — Theory and Applications*, 12(1–2):85–103, 1999.

[53] M. Meyerovitch. Robust, generic and efficient construction of envelopes of surfaces in three-dimensional space. In *Proc. 14th Annual European Symposium on Algorithms*, 2006. To appear.

[54] B. Mourrain, J.-P. Técourt, and M. Teillaud. Sweeping an arrangement of quadrics in 3d. In *Proc. 19th European Workshop on Computational Geometry*, pages 31–34, 2003.

[55] K. Mulmuley. An efficient algorithm for hidden surface removal. In *Proc. 16th Annual Conference on Computer graphics and Interactive Techniques (SIGGRAPH '89)*, pages 379–388, New York, NY, USA, 1989. ACM Press.

[56] K. Mulmuley. An efficient algorithm for hidden surface removal, ii. *Journal of Computer and System Sciences*, 49(3):427–453, 1994.

[57] N. Myers. "Traits": A new and useful template technique. In S. B. Lippman, editor, *C++ Gems*, volume 5 of *SIGS Reference Library*, pages 451–458. 1997.

[58] J. Pach and M. Sharir. The upper envelope of piecewise linear functions and the boundary of a region enclosed by convex plates: Combinatorial analysis. *Discrete and Computational Geometry*, 4:291–309, 1989.

[59] F. P. Preparata and M. I. Shamos. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[60] S. Schirra. Robustness and precision issues in geometric computation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1999.

[61] M. Sharir. Almost tight upper bounds for lower envelopes in higher dimensions. *Discrete and Computational Geometry*, 12:327–345, 1994.

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

[63] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library, User guide and reference manual*. Addison-Wesley, 2002.

[64] R. Wein. High level filtering for arrangements of conic arcs. In *Proc. 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 884–895. 2002.

[65] R. Wein. High-level filtering for arrangements of conic arcs. M.Sc. thesis, School of Computer Science, Tel Aviv University, Tel Aviv, Israel, 2002.

[66] R. Wein and E. Fogel. The new design of CGAL's arrangement package. Technical report, Tel-Aviv University, 2005. `http://www.cs.tau.ac.il/~wein/publications/pdfs/Arr_new_design.pdf`.

[67] R. Wein, E. Fogel, B. Zukerman, and D. Halperin. Advanced programming techniques applied to CGAL's arrangement package. In *Library-Centric Software Design Workshop (LCSD'05), part of OOPSLA*, 2005. Available online at `http://lcsd05.cs.tamu.edu/#program`.

[68] R. Wein and D. Halperin. Generic implementation of the construction of lower envelopes of planar curves. Technical Report ECG-TR-361100-01, Tel-Aviv University, 2004.

[69] R. Wein, J. P. van den Berg, and D. Halperin. The visibility-Voronoi complex and its applications. In *Proc. 21st Annual ACM Symposium on Computational Geometry*, pages 63–72, 2005.

[70] N. Wolpert. *An Exact and Efficient Approach for Computing a Cell in an Arrangement of Quadrics*. PhD thesis, Universität des Saarlandes, October 2002.

[71] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, 2nd edition, 2004.