TEL AVIV UNIVERSITY אוניברסיטת תל-אביב

Raymond and Beverly Sackler
Faculty of Exact Sciences
School of Computer Science

The Integration of Exact Arrangements with Effective Motion Planning

Thesis submitted for the degree of “Doctor of Philosophy”

by

Ron Wein

This work has been carried out under the supervision of
Prof. Dan Halperin

Submitted to the Senate of Tel-Aviv University
March 2007

Acknowledgements

I wish to thank Prof. Dan Halperin for his guidance and his help during the work on this thesis. I consider myself very lucky to have a supervisor like Danny.

I would like to thank Efi Fogel and Baruch Zukerman from Tel-Aviv University; Gershon Elber, Iddo Hanniel and Oleg Ilushin for the Technion; Jur van den Berg from Utrecht University; and Eric Berberich from Max-Planck-Insitut für Informatik, for ongoing and fruitful collaboration.

In the design of the software packages described in this thesis I got many useful remarks and advises from members of the Cgal Editorial Board. I would especially like to mention Andreas Fabri, Lutz Kettner and Sylvain Pion. I would also like to thank all the members of the Cgal developers' community, and the members of the Tel-Aviv group in particular. I also thank Chee Yap's group for providing support for the Core library.

During the years I participated in the EU-funded projects Ecg (Effective Computational Geometry for Curves and Surfaces; contract No. IST-2000-26473), Movie (Motion Planning in Virtual Environments, contract No. IST-2001-39250) and Acs (Algorithms for Complex Shapes; contract No. IST-006413) projects. I wish to thank all the other participants in these projects, with whom I enjoyed working.

Finally, I would like to express my gratitude to my wife Hélène for all the love and support she has given me, to my parents Nili and Hezi for their encouragement, and to my daughters Nogah and Shir for the joy they brought into my life.

Abstract

The field of computational geometry has greatly evolved over the last three decades, yielding a multitude of algorithms to solve real-life problems emerging from diverse application fields, such as robotics, solid modeling and geographical information systems. Yet implementing a geometric algorithm from a textbook is far from being a trivial task. Common assumptions in the design of computational-geometry algorithms are that all calculations can be carried out using exact arithmetic, and that the input objects are in general position. Unfortunately, these assumptions usually do not hold in practice.

This thesis describes a robust infrastructure for geometric computing with two-dimensional arrangements, and the integration of software solutions based on robust geometric algorithms into various applications, especially in the areas of motion planning and computer-aided design. Our software combines state-of-the-art techniques and achieves unprecedented running times when constructing and manipulating arrangements in an exact manner, thus it serves as a solid infrastructure for developing efficient software solutions to many real-life problems. Most of the work described in the thesis has been integrated into CGAL, the Computational Geometry Algorithms Library.

The main results presented in this thesis are as follows.

The CGAL **Arrangement Package.** Given a set of planar curves, their *arrangement* is the subdivision they induce on the plane into maximally connected cells. Arrangements are ubiquitous in computational geometry and have many applications. The 2D arrangement package of CGAL allows the construction and maintenance of planar arrangements of arbitrary families of curves, such as line segments, conic arcs, Bézier curves, etc. It assumes that the curves can be handled in a *certified* manner, and employs enhanced methods for certified geometric and algebraic computation to achieve this task. The package is robust, namely it can seamlessly handle all kinds of degenerate inputs.

The arrangement package has undergone a complete re-design. By employing advanced software-design techniques, the package is made more flexible and extendible. At the same time, a lot of effort was spent on exploiting all combinatorial information available in the various arrangement-related algorithms in order to minimize the number of geometric operations they perform. In conjunction with the development in certified computation techniques, the new arrangement package now offers highly efficient construction times of exact arrangements. The modular and extendible design of the package, as well as its efficient operation, made it a convenient basis for the development of several peripheral CGAL packages, most notably computing Boolean operations on polygons, and constructing lower envelopes of 3D surfaces. Chapter 2 reviews the guidelines that led the design of the package and highlights its main features and capabilities.

Chapter 2 also describes another aspect of the package evolvement, namely the development of *traits classes* that enable the package to work with various families of curves. Computing with curved objects is a real challenge: we wish to carry out the computations in an exact manner, while at the same time the computation should

be efficient, so the running times are acceptable for practical applications. Special attention is given to the central issue of using exact number types.

Computing Minkowski sums and offset polygons. Computing the Minkowski sum (namely the pointwise sum) of two sets is a fundamental operation for solving problems in motion planning and computer-aided design. Chapter 3 overviews a new CGAL package that supports the Minkowski-sum computation of two straight-edge polygons, or of a polygon and a disc (an operation known as *offsetting* the polygon). This package, also based on the arrangement infrastructure, is the first to provide a robust implementation of the *convolution* method, achieving faster running times in comparison to software packages that use convex polygon decomposition for the same task.

The package also addresses the problem of offsetting a polygon. The main difficulty in performing this operation in an exact manner is algebraic, as it requires the exact manipulation of algebraic numbers of degree up to four. Our package therefore includes an offset approximation algorithm with guaranteed precision bounds, which employs only exact rational arithmetic. This algorithm may expedite the offset computation by a factor of 6–15.

Using the Minkowski-sum package in conjunction with the operations on general polygons provided by the Boolean set-operations package, it is now possible to provide complete and exact solutions to many practical large-scale problems in real time, a task that was regarded impossible a few years back.

Planning collision-free paths for NC-machining. A typical industrial NC-machine comprises a *table* that can move along three axes and is also able to rotate, and a *milling cutter* with another degree of rotational-motion freedom. The cutter rotates about its axis of symmetry and sculpts a *workpiece* placed on the table. The machine is controlled by a sequence of commands that allow the cutter (also known as a *tool*) five degrees of motion freedom. The goal is to devise motion paths for the cutter that allow it to carve the workpiece. Namely, the tool tip is allowed to make contact with the workpiece, but the rest of the cutter should not touch it, or any other part of the machine.

Chapter 4 describes the design and implementation of algorithms that verify that a given motion path of the tool is collision-free, and are based on our robust software infrastructure. The verification can be done by sampling the path at several discrete positions and verifying that each position is collision free, or by considering the continuous motion of the tool along short path segments. In both cases it is possible to exploit the axial symmetry of the tool and reduce the problem to the computation of the lower envelope of a set of line segments and hyperbolic arcs, which our software can robustly handle, and to compare this envelope to the tools profile.

The verification algorithms we suggest offer superior accuracy in comparison to other algorithms that have appeared in the CAD literature. In particular, the dynamic version for the algorithm allows for collision detection at a very fine precision level.

Planning high-quality paths and corridors. Since many motion-planning variants are hard to solve, most algorithms focus on just planning some collision-free motion path for the moving entity, regardless of its quality. Applications that require high-quality paths often employ a postprocessing step that enhance the quality of the path by smoothing it, eliminating unnecessary loops or detours, etc. Such path-enhancement techniques are often heuristic and give no guarantee on the quality of their output.

As complete solutions to some motion-planning variants are now available, we turn to the problem of planning high-quality paths. In Chapter 5 we devise a data structure called the *visibility–Voronoi diagram*. Given a set of obstacles and a preferred clearance value c, our diagram interpolates between the visibility graph of the obstacles and their Voronoi diagram, including paths that offer a natural balance between two optimization criteria: minimizing the path length and maximizing its clearance, up to the preferred value c. We also propose an algorithm that is capable of preprocessing a scene of polygonal obstacles and constructs a data structure called the *visibility–Voronoi Complex*. The visibility–Voronoi Complex can be used to efficiently plan motion paths for any start and goal configuration and *any clearance value* c, without having to explicitly construct the *visibility–Voronoi diagram* for that c-value. The preprocessing time is $O(n^2 \log n)$, the same as the time needed for constructing a single visibility–Voronoi diagram, where n is the total number of obstacle vertices. The complex can be queried directly for any c-value by merely performing a Dijkstra search.

In Chapter 6 we address the problem of planning high-quality corridors. Planning corridors among obstacles has arisen as a central problem in application fields like robotics and game design: instead of devising a one-dimensional motion path for a moving entity, it is possible to let it move in a corridor, where the exact motion path is determined by a local planner. We introduce a measure for the quality of such corridors. We analyze the structure of optimal corridors amidst point obstacles and polygonal obstacles in the plane, and propose an algorithm to compute tight approximations for optimal corridors.

We conclude the thesis with some future prospects, presented in Chapter 7 and in the two appendices to the thesis. First, we describe prospective traits-classes which will support additional families on curves. In particular, we show how a traits class that handles algebraic curves with rational coefficients can be used for solving the motion-planning problem of a polygonal robot translating and rotating amidst polygonal obstacles. We also describe an extension of the arrangement package to support two-dimensional arrangements of curves embedded on general surfaces, such as spheres, cylinders, etc. We devise a framework for handling various surfaces in a unified manner, thus maximizing code reuse and reducing development efforts. The extended arrangement package will pave the way to develop new applications in fields like molecular modeling and solid modeling.

Contents

Chapter 1

Introduction

Given a set $\mathcal{C}$ of planar curves, we refer to the planar subdivision they induce as their *arrangement*. Arrangements are ubiquitous in the computational-geometry literature and have numerous applications in many fields (see, e.g., [AS00a, Hal04]). In this thesis we mainly consider applications in motion planning, computer-aided design (Cad) and computer-aided manufacturing (Cam).

While some computational geometry problems, such as computing the convex hull of a set of points or constructing a Delaunay triangulation for a set of points (see, e.g., [dBvKOS00]), can be solved using *predicates* that involve the geometric entities given as an input (the points in this case), computing the arrangements of a set of curves requires the *construction* of new geometric objects based on the input, as one obviously needs to compute all intersections between pairs of curves in the set. The assumption often made in the theoretical study of geometric algorithms (which constitutes the vast majority of the computational-geometry literature), that one can carry out infinite-precision computations on real numbers, often breaks down in practice, especially when we encounter degenerate situations (e.g., three curves intersecting at a common point), or nearly-degenerate situations that are indistinguishable from degenerate ones due to the usage of inexact machine-precision arithmetic.

The study of motion-planning problems was at first motivated by the design of autonomous robots that should — among their other capabilities — be able to navigate their way without having a prescribed path. We therefore refer to the entity in motion as a "robot", keeping in mind that motion planning has many other applications in other fields, such as assembly planning, computer animation and structural bio-informatics [Lat99], where the moving entities can be parts of machines, virtual characters in a computer game, or even molecules. The motion-planning problem has many variants, defined by the shape of the moving entities, their possible motions and by the shape of the obstacles [HKL04, Lat91].

Motion-planning problems have drawn much attention over the last three decades and many algorithms were suggested to tackle the more common motion-planning variants or to supply unified frameworks for handling families of motion-planning variants. While some methods offer a complete theoretical solution (see [Sha04] for a comprehensive survey), they tend to be rather complicated and almost impossible to implement — and indeed, there hardly exist exact implementations of such complete algorithms. On the other hand, several heuristic approaches were designed to tackle complicated motion-planning problems, the most popular being the probabilistic roadmap (PRM) approach [KŠLO96]. While this method is very intuitive and easy to implement, it is known to have a major drawback — it usually fails to find a motion path when the obstacles are cluttered and the robot has to go through narrow passages amidst these obstacles. Moreover, the running time of a PRM-based algorithm is often unpredictable, and even when a program runs for hours on a given motion-planning query and has not found a solution yet, one cannot be certain that there is no solution (namely, a collision-free path) to that query.

In this thesis we provide some tools that can help bridging the gap between the theoretically complete methods and the heuristic approaches, yielding robust motion-planning algorithms and providing efficient software solutions to related problems. Almost all exact motion-planning algorithms consider a set of critical contact curves (or surfaces), which corresponds to the locus of robot positions that bring some robot feature to be in contact with one of the obstacles without penetrating it, and study the subdivision these critical curves (or surfaces) induce. Even in low dimensions, the critical curves tend to be quite complicated and computing the subdivision they induce cannot be implemented with machine-precision arithmetic without causing runtime failures. It is therefore clear that a reliable software tool for constructing and maintaining arrangements is a prerequisite for providing a robust implementation of a motion-planning algorithm. We describe a robust software package for planar arrangements, along with several peripheral packages implemented on top of it that offer fundamental operations such as Boolean set-operations and Minkowski-sum computation. Most packages are included in the latest public release of CGAL, the Computational Geometry Algorithms' Library (Version 3.2), and others will soon become publicly available under the forthcoming release (Version 3.3). We show how the exact solution of some variants of the motion-planning problems can be obtained in a robust manner while not incurring a prohibitive running-time penalty.

Since many motion-planning variants are hard to solve, most algorithms focus on just planning some motion path for the moving entity, regardless of its quality. Applications that require a high-quality path often employ a post-processing step that enhance the quality of the path by smoothing it, eliminating unnecessary loops or detours, etc. Such path-enhancement techniques are often heuristic and give no guarantee on the quality of their output.

As complete solutions to some motion-planning variants are now available, we turn to the problem of planning high-quality paths. We devise data structures that can be efficiently queried and output a high-quality path, without the need for any post-processing phase. We also develop a quality measure for motion paths and show that our data structures yield nearly-optimal paths with respect to this measure.

The rest of this chapter is organized as follows. In Section 1.1 we survey the principles of exact motion-planning algorithms, discuss their complexity and describe some motion-planning variants that can be solved in an exact manner. Section 1.2 focuses on techniques for computing high-quality motion paths. In Section 1.3 we outline the rest of the thesis and give references to the related publications.

1.1 Exact Arrangements and Their Applications for Motion Planning

Early works on motion-planning problems were conducted by Lozano-Pérez [LP83, LPW79]. In his work, Lozano-Pérez defined the terms that are used in almost all following works: The robot moves in an environment that is usually 2-dimensional or 3-dimensional, called the *workspace*, amidst a set of obstacles it should avoid colliding with. We say that the *configuration space* of the robot is d-dimensional if it is possible to uniquely characterize the position of the moving entity by a d-tuple, which we refer to as a *configuration*. In this case we say that the motion-planning problem has d *degrees of freedom*.

For instance, for a disc robot moving in the plane it is sufficient to specify the location of the center of the robot $\langle x, y\rangle$. In case the robot is polygonal and can also rotate, we can fix one of its vertices as a reference point and specify the location of this point (x, y), as well as the angle θ that the edge between this vertex and one of its neighboring vertices forms with the x-axis. The position of the robot is thus characterized by $\langle x, y, \theta\rangle$. A "free flyer" entity in space has six degrees of motion freedom: the location of some reference point on the entity and three angles that determine its orientation. In case there exist m moving entities, each with d degrees of freedom, the motion-planning problem gets more complicated, as the robots have to coordinate their motions such that they do not collide with one another. Indeed, the dimension of the configuration space in this case is md.

Using the notion of configuration space, it is possible to convert each obstacle O in the workspace into a C-obstacle $C(O)$ in the configuration space, such that the robot does not collide with the original workspace obstacle O when its location is specified by the configuration q if and only if $q \notin C(O)$ in the configuration space (see Figure 1.1 for an illustration). The union of all C-obstacles defines the *forbidden configuration space*, a subset of the configuration space that contains all the forbidden robot configurations, denoted C_{forb}. The complement of this set is the *free configuration space*, denoted C_{free} . It is also possible to define *semi-free configurations*, where the moving entity is in contact with one (or more) of the obstacles, but does not penetrate it. A maximal connected component within C_{free} or C_{forb} is called a *cell*. The decomposition of the configuration space to free and forbidden cells is therefore a convenient basic framework for solving motion-planning queries: Given a start and a goal configurations of the robot, we check whether they are in the same free cell, and if so — we find a path that connects the two configurations within this cell.

In his work, Lozano-Pérez used an approximate representation of the configuration-space regions. Schwartz and Sharir [SS83b] used more careful algebraic topological analysis, based on Collins' decomposition method [Col75], to obtain a description of the free configuration space. Their algorithm takes $n^{2^{O(d)}}$ time, where n is the number of C-obstacle features and d is the degree of the configuration space. Canny [Can87] suggested the "roadmap" algorithm, which runs in $O(n^d m^{O(d^4)} \log n)$ time, where m is the maximal degree of the polynomial constraints that define the motion-planning problem. This algorithm is singly-exponential in d and polynomial in n. Basu *et al.* [BPR96] used careful algebraic analysis to improve the decomposition scheme and achieve an algorithm that takes $O(n^{d+1})b^{O(d)}$ time, where b is the complexity of the robot. Chazelle *et al.* [CEGS91] suggested an improved general

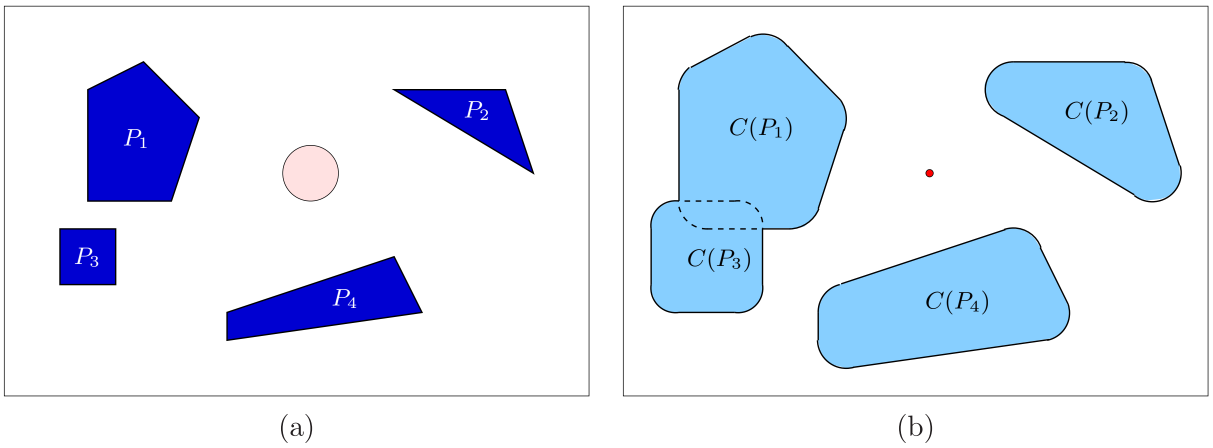


Figure 1.1: (a) The workspace of a disc robot moving among three polygonal obstacles. (b) The corresponding C-obstacles. Both the workspace and the configuration space are 2-dimensional in this case, but notice that the robot becomes a single point in the configuration space.

decomposition method called *vertical decomposition*, whose complexity is singly-exponential in d. Currently the best bound on the size of the vertical decomposition is $O(n^{2d-4+\varepsilon})$ and is due to Koltun [Kol04].

The general motion-planning problem is therefore extremely difficult to solve in an exact manner. In fact, if the dimension of the configuration space (namely, the number of degrees of freedom of the moving entities) is part of the input, it is possible to show that even restricted variants of the motion-planning problem are PSPACE-hard; see, e.g., [Rei87, RS94, SM88]. Canny [Can88] showed that the motion-planning problem is also PSPACE-complete. However, for the simpler variants of the problem, where the dimension of the configuration space is low, there exist theoretically efficient algorithms.

1.1.1 Translational Motion in the Plane

An efficient algorithm for the case of a convex polygonal robot Q that translates in the plane among obstacles $P_1, \ldots, P_k$ was given by Kedem *et al.* [KLPS86]. The C-obstacles in this case are given by the Minkowski sum of each polygon with $-Q$, a copy of the robot reflected about the origin (that is, Q rotated by π), namely $C(P_k) = P_k \oplus (-Q)$, and are polygonal as well. It is possible to compute $C_{\text{forb}} = \bigcup_{i=1}^{k} C(P_i)$ by considering the arrangement of the C-obstacle boundaries. The free cells of this arrangement are then decomposed into pseudo-trapezoids by introducing artificial "vertical walls" (see Figure 1.2), and it is possible to construct a graph that captures the connectivity of the free configuration space. In this *connectivity graph* we assign a vertex for each free pseudo-trapezoid and connect each pair of free pseudo-trapezoids that share a common vertical wall by an edge. This graph is used for answering motion-planning queries: given a start and a goal configurations of the robot $s, g \in \mathbb{R}^2$ (as mentioned before, the configuration in this case is the position of some fixed reference point on Q), we can efficiently locate the two pseudo-trapezoids that contain the two points (see, e.g., [Mul90]) and then check if there exists a path between them in the connectivity graph. This variant of motion-planning can thus be solved in $O(n \log^2 n)$

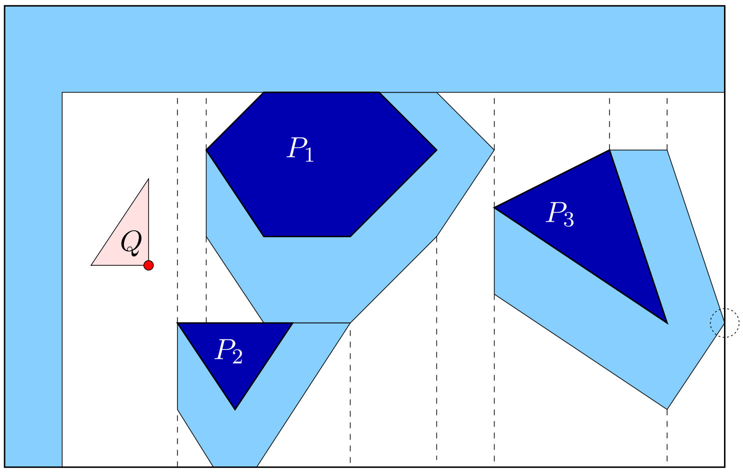


Figure 1.2: The vertical decomposition of the free configuration space for a triangular robot (lightly shaded; the reference point is the bottom-right vertex of the robot). The original workspace obstacles are darkly shaded. Note that in this example there exist several tight passages, and thus some pseudo-trapezoids have a degenerate form of a line segment or even of a single point (marked by the dotted circle on the right).

time, by using a divide-and-conquer approach to compute the union of the C-obstacles. It is also possible to solve this variant in $O(n \log n)$ time (see, e.g, [BZ88]). See also Leven and Sharir [LS87], who suggest an $O(n \log n)$ solution to this motion-planning variant by considering generalized Voronoi diagrams.

Our software packages include all the ingredients for implementing this motion-planning algorithm: namely it can efficiently construct Minkowski sums of polygons and compute the arrangement of their boundary edges. The vertical decomposition scheme is also provided by our arrangement package. For the construction of the connectivity graph and the reachability algorithms we can use the Boost graph library [SLL02], which is becoming a generic-programming standard. The data structures in our arrangement package can be treated as Boost graphs, allowing users to conveniently interface with the graph algorithms this library provides.

The problem of a disc moving amidst polygonal obstacles can be solved in a similar fashion. The type of C-obstacles in this case is interesting in its own right and is often called the *offset polygon*. These offset polygons are computed by "inflating" each polygon by the radius of the robot, obtaining forbidden regions bounded by line segments and circular arcs. As our arrangement package is generic and is able to handle various families of curves, this variant of the motion-planning problem is also conveniently solved.

We mention that due to a large number of improvements in the latest releases of the arrangement package and its peripheral algorithms, we obtain software that is two orders of magnitude faster than previous implementations of motion planners, implemented using previous Cgal versions (Flato [Fla00] for a translating polygonal robot, and Hirsch and Leiserowitz [HL02] for a disc robot). In addition, we present an approximation scheme for computing the Minkowski sum of a polygon and a disc, which helps achieving ever faster running times, especially when it is necessary to perform union operations on the

C-obstacles. Our approximation scheme is *conservative*, namely it computes a super-set of the true forbidden configuration space. The approximation error can be made arbitrarily small without incurring prohibitive running-time penalties.

1.1.2 Translation and Rotation in the Plane

Schwartz and Sharir [SS83a] were the first to suggest an exact algorithm for planning the motion of a robot that can translate and rotate in the plane. This algorithm is commonly known as the *piano-movers' algorithm.* They first solved the problem for the case of a line segment (also called a "*ladder*") translating and rotating amidst polygonal obstacles. Instead of constructing a 3-dimensional representation of the free configuration space, it is possible to project the configuration space onto the plane. We thus obtain a set of critical curves that define the boundaries of planar cells, where all positions within each cell share the same characteristics of free orientations. Namely, if we place the reference point of the robot in any point in one of these cells and rotate it, it will hit the same set of obstacles in the same order (clockwise or counterclockwise order). By assigning a suitable combinatorial label to each cell, it is possible to construct a connectivity graph and to reduce every motion-planning query to a simple graph-reachability query. A similar approach can be applied when the robot is a polygon, where in both cases the critical curves are algebraic arcs of degree 4 at most. The complexity of the process is $O(n^5)$ in case of a ladder, where n is the total number of vertices in all obstacles, and $O(n^5m^5)$ in case the robot is a polygon with m vertices.

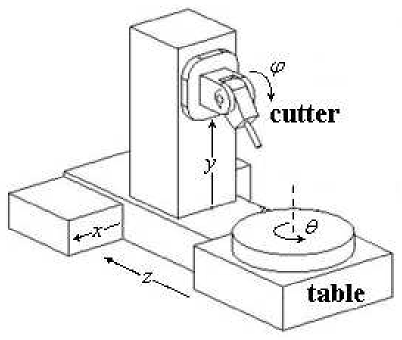
While not being the most efficient motion-planning algorithm for a translating and rotating polygon (see Halperin and Sharir [HS96] for an algorithm with near-quadratic running time — i.e., a worst-case near-optimal algorithm), the nice property of piano-movers' algorithm is that it solves the problem in the plane, without having to consider the three-dimensional configuration space directly. Moreover, it can be shown that if the obstacles are *"fat"*, namely they contain no skinny features, then the worst-case complexity of the algorithm drastically improves [vdS94].

We do not present a software implementation to the piano-movers' algorithm, but show that the critical algebraic curves we have to consider all have rational coefficients. As computation with rational algebraic curves is becoming a reality, and future version of CGAL will include components that can handle algebraic curves of arbitrary degree, implementing the exact solution of the piano-movers' algorithms will soon be possible.

1.1.3 Hybrid Motion-Planning Algorithms

A natural extension of the basic motion-planning problem of a single robot is having several robots moving around in the scene. In this case, the motion of the robots should be coordinated so they do not collide with one another while trying to avoid the static obstacles. Complete solutions to coordination problems are very complicated because of the relatively large number of degrees of freedom, causing the running time to be exponential in the number of robots (for example, when we have to coordinate k disc robots in space, the dimension of the configuration space is $2k$). Sharir and Sifrony [SS91] described a complete $O(n^2)$ solution for two independent robots moving among polygonal obstacles in the plane, thus constituting a system with four degrees of freedom, which is a special case of the complete algorithm to the general motion planning problem described in [SS83c].

The evolvement of the Cgal arrangement package opened the door for hybrid solutions to more complicated motion-planning variants, such as the coordination problem. Hirsch and Halperin [HH03] give a solution for the problem of two disc robots moving in the plane among polygonal obstacles, introducing a method they named *hybrid motion planning*. They first construct the free configuration space for each robot separately using exact methods, discovering a relatively large portion of C_{forb}. Now it is necessary to coordinate the motions of the two robots so that they do not collide with one another. This is done in part by examining the cells in $\mathbb{R}^4$ obtained by the Cartesian product of each pair of two free planar cells, taken respectively from the two free configuration spaces of the individual robots. If the two planar cells are sufficiently distant from one another, the 4-dimensional cell is completely free. Otherwise, a local Prm is computed within this cell. These local roadmaps are then stitched together to capture the connectivity of the 4-dimensional configuration space.



In this thesis we apply a hybrid approach to plan collision-free motion paths in 5-axis NC-machining. A typical industrial *numerically-controlled machine*, or *NC-machine* for short, is sketched on the right. It comprises a *table* that can move along three axes and is also able to rotate, and a *milling cutter* with another degree of rotational-motion freedom. The cutter rotates about its axis of symmetry and sculpts a workpiece that is placed on the table (not shown in the illustration). The machine is controlled by a sequence of commands that allow to slide the table or to rotate it, to change the angle of the cutter, etc. It is possible to view the table and the workpiece as stationary, such that the cutter (also known as a *tool*) has five degrees of motion freedom. The goal is to devise motion-paths for the tool that allow it to carve the workpiece. Namely, the tool tip is allowed to make contact with the workpiece, but the rest of the cutter should not touch the workpiece (or any other part of the machine). This problem, having five degrees of motion freedom, is very difficult to handle using exact methods, but we can devise methods that can exactly solve some related fundamental primitives that can serve as building blocks for a probabilistic solution.

We first consider the static collision-detection problem, namely determining whether a given configuration of the tool is collision-free. We show that it is possible to answer such queries by considering the lower envelope of a set of planar curves, obtained by the radial projection of the workpiece about the symmetry axis of the tool, and comparing it to the profile of the cutter. We then extend this approach to verify that simple motions of the tool, the most important one being translation of the tool while keeping a fixed orientation. In this case we consider the lower envelope of the so-called "silhouettes" of the triangles constituting the workpiece. We show that these silhouette curves are rather simple planar curves, so the problem is once again reduced to the lower-envelope computation. Lower envelopes are special sub-structures of planar arrangements, and we can efficiently compute envelopes of arbitrary planar relevant curves using our software.

We use these collision-detection primitives to verify more complicated motion paths. When we encounter collision, we use these primitives as oracles for a Prm used to plan a local correction for the path that resolves the collision.

1.2 Computing High-Quality Motion Paths

Since the motion-planning problem in general is very difficult, most algorithms that were suggested for solving variants of the motion-planning problem focus on computing *some* collision-free motion-path for the moving entity (or entities), regardless of its quality. In the second part of this thesis we devise data structures that can efficiently answer motion-planning queries and output *high-quality* motion-paths. We utilize our robust software tools to implement one of these data structures, that can compute nearly-optimal paths amidst polygonal obstacles in the plane, according to natural quality measures we propose.

The exact definition of a high-quality path is somewhat elusive, as it depends on the type of application that utilizes such a path. For many applications, such as planning the motions of virtual characters in game design, it is important that the motion paths look natural. By “natural-looking” we mean that the moving entity should select a path that mimics as closely as possible the path a human would take in the same scene to reach the goal configuration from a given start configuration. This essentially means the following:

1. The path should be *short* — that is, it should not contain long detours when significantly shorter routes are possible.
2. It should have a guaranteed amount of *clearance*. Namely, the distance of any point of the moving entity along the path to the closest obstacle should not be below some prescribed value.
3. The path should be *smooth*, not containing any sharp turns.

The same principles are adequate for applications in other fields. For instance, it is important that the motion-path for an automated vehicle is as short as possible in order to minimize its fuel consumption — but at the same time, we want it to keep away from obstacles and to avoid sharp turns, which are impossible due to the stirring mechanism. However, requirements 2 and 3 may conflict with requirement 1 in case it is possible to considerably shorten the path by taking a shortcut through a narrow passage. In such cases we may prefer a path with less clearance, allowing the moving entity to arrive at the goal configuration more quickly.

Let us consider the motion-planning problem for a robot with two degrees of freedom presented in Section 1.1.1. The connectivity graph indeed captures the connectivity of the free configuration space and in this sense the algorithm is complete. However, paths extracted from this graph are piecewise linear (hence not smooth) and are often not the shortest paths. Indeed, it is possible to perform path smoothing as a post-processing stage and produce a more natural-looking path (see [GO03] for a summary of applicable post-processing techniques), but as there is no guarantee that the initial path is in the same homotopy class as the best path possible, the smoothed path may still be different from the most natural-looking path, as smoothing techniques typically do not change the homotopy class of the path they process. Another drawback of the post-processing approach is that it usually employs expensive procedures, which may incur running-time penalties that are prohibitive in a real-time application (e.g., planning the motion of virtual characters in a computer game). Having a data structure that can be queried at real-time and provide a high-quality motion path, without any need for expensive post-processing, is therefore essential for such applications. We mention that in many cases the construction of such a data structure can be done as a preprocessing stage.

Similar artifacts occur if we apply a Prm-based approach for planning a collision-free path: since the edges of the Prm graph usually correspond to straight line segments in the configuration space, the resulting path is piecewise linear and may be much longer than the shortest possible path. Nieuwenhuisen and Overmars [NO04b] suggest using a Prm with cycles rather than a minimally-connected Prm (which is actually a forest and contains no cycles) in order to obtain shorter paths from the Prm. Another idea is to incorporate circular arcs into the Prm, thus eliminating the sharp turns and replacing them by smooth ones [NKMO04]. While these methods considerably improve the quality of the paths generated by Prm techniques, they still suffer from the general drawbacks of the probabilistic nature of the Prm method. Namely, due to poor sampling we may plan a path that is considerably longer than the best path possible, on even worse — fail to detect a collision-free motion path when one actually exists. Moreover, even when a path is generated it can be in a different homotopy class than that of the optimal path, so the two paths are substantially different.

The *visibility graph* is a well-known data structure for computing the shortest collision-free path between a start and a goal configuration (see, e.g., [dBvKOS00, Chapter 15] and [Mit04]). However, shortest paths are in general tangent to obstacles, so a path computed from a visibility graph usually contains semi-free configurations and therefore does not have any clearance. This not only looks unnatural, it is also unacceptable for many motion-planning applications. Moreover, having no clearance makes it impossible to apply smoothing techniques on the paths extracted from a visibility graph.

On the other hand, planning motion paths using the *Voronoi diagram* of the obstacles [ÓY85] yields a path with maximal clearance. The drawback of this method (known as the *retraction* method), is that the path it outputs may be significantly longer than the shortest path possible and may also contain sharp turns. It is worth mentioning that post-processing techniques that try to “push” the path away from the obstacles and to approximately retract it to the Voronoi diagram of the obstacles (see, e.g., [GO04]), suffer from the same problem.

1.2.1 The Visibility–Voronoi Diagram

We introduce a new type of diagram called the VV$^{(c)}$-diagram (the *Visibility–Voronoi diagram* for clearance c), which is a hybrid between the visibility graph and the Voronoi diagram of polygons in the plane. It evolves from the visibility graph to the Voronoi diagram as the parameter c grows from 0 to ∞. This diagram can be used for planning natural-looking paths for a robot translating amidst polygonal obstacles in the plane.

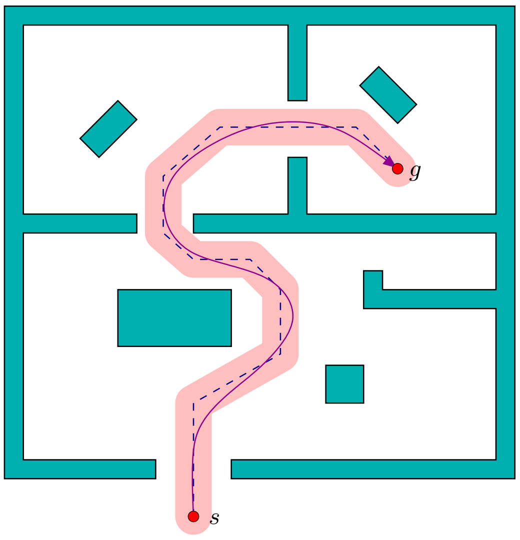


Figure 1.3: A small disc robot moving within a corridor (lightly shaded) amidst polygonal obstacles (darkly shaded). The backbone of the corridor is drawn with a dashed line, while the path actually taken by the robot is drawn as a solid curve.

We also propose an algorithm that is capable of preprocessing a scene of configuration-space polygonal obstacles and constructs a data structure called the *visibility–Voronoi complex*, or VV-complex for short. The VV-complex can be used to efficiently plan motion paths for any start and goal configuration and *any clearance value* c, without having to explicitly construct the $\mathrm{VV}^{(c)}$-diagram for that c-value. The preprocessing time of the VV-complex is $O(n^2 \log n)$, where n is the total number of obstacle vertices, and the data structure can be queried directly for any c-value by merely performing a Dijkstra search.

It is important to mention that intensive geometric computation is needed only when constructing the $\mathrm{VV}^{(c)}$-diagram (or the VV-complex), and can thus be performed at a preprocessing stage. The resulting structures can be queried efficiently using machine-precision arithmetic, without causing any computational instabilities.

1.2.2 Planning Corridors

A common drawback of most traditional motion-planning methods (the complete ones and the heuristic ones) is that they output a fixed path in response to a query. This path is often not the ideal solution for motion planning, as it lacks flexibility to avoid local hazards (such as small obstacles, other moving entities, etc.) that are encountered during the motion. It also leads to predictable, and possibly unrealistic motions, which are not suitable for some applications, such as computer games. One approach for tackling these problems is a potential-field planner, in which the moving entity is attracted to its goal configuration, and repelled by obstacles, or other moving entities (see, e.g., [Kha86]). However, this approach is prone to get stuck in local minima of the potential field; while there are methods that help in resolving such situations (see, e.g., [Lat91]), they may still not yield valid motions at all.

We would therefore like to indicate the global direction of movement for the moving entity, while leaving enough flexibility for some *local planner* to avoid local hazards. An ideal solution for this is to use *corridors*, which have recently been introduced in the game-design field [Ove05]. In contrast to one-dimensional motion-paths, corridors are defined as a union of balls whose center points lie along a backbone path, and therefore have a volume (see Figure 1.3). Once we compute a collision-free corridor, the more restricted task of locally planning the motion around the backbone path can be successfully performed by applying potential-field methods. In order to guarantee that the local planner operates on a restricted environment, the width of the corridor should be upper bounded by some predetermined value. Corridors thus give a strict global direction of movement, yet allow for the local flexibility of motion that is essential in many cases.

Let us consider the following interesting application of planning the motion of a group of small entities in a two-dimensional workspace cluttered with polygonal obstacles, where it is desirable that the entities move together as a coherent group. Kamphuis and Overmars [KO04a] solve this problem by planning a collision-free path for a single entity, then "inflating" this backbone path up to a diameter of a preferred group width w, wherever possible, and governing the motions of the individual entities inside this inflated path using a social potential field [RW95], where the robots are attracted to the backbone path and are repelled from the corridor boundaries and from one another. As we mentioned before, the local nature of the potential field within the corridor avoids the problem of local minima.

Planning within corridors has many other interesting applications. For example, it is possible to use a corridor to plan the motion of a camera that follows a moving character (a *guide*) [NO04a]. The motion-path of the guide constitutes the backbone of the corridor, which is wide enough to allow the camera the flexibility to swerve if necessary. Another advantage of corridors is that they allow for non-holonomic and kinodynamic planning, if the motion of a single entity (or multiple entities) is planned using a potential-field method within the corridor [KPOL05]. This is very difficult to achieve and incorporate into a fixed path. As a rule of thumb, corridors are very successful in solving motion-planning variants where small entities move amidst some relatively large static obstacles. Corridors are thus suitable for applications in many fields, such as open field robotic navigation and game design, that consider motions of robots, cars or virtual characters amidst buildings, rocks, etc.

In previous works that consider motions in corridors (see, e.g., [But06, KO04b]), heuristic methods are used to construct the corridors, without any guarantee of their quality (for example, [KO04b] use a Prm with cycles [NO04b] to compute the initial path, then post-process it to obtain a smooth path). We introduce a quality measure for corridors, which captures the desirable properties given in the beginning of this section. We show that the backbone path of an optimal corridor is essentially smooth and offers a balance between its length and its clearance. We analyze the structure of optimal corridors amidst point obstacles and amidst polygonal obstacles in the plane. We show that optimal corridors are hard to compute, yet we devise an approximation algorithm that can compute near-optimal corridors.

We also show that the path computed by the $\mathrm{VV}^{(c)}$-diagram can be conveniently used as approximation to an optimal backbone path amidst polygonal obstacles, where c is the preferred corridor width. We have implemented a CGAL-based software package for computing the VV$^{(c)}$-diagram in an *exact* manner for a given clearance value and used it to plan high-quality corridors in various applications.

1.3 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2 we give an overview of the CGAL arrangement package, which serves as a common infrastructure for almost all software solutions described in this thesis. This package has undergone a major redesign and re-implementation, which enhanced its efficiency, genericity, extendibility and ease-of-use. This chapter is an extract of a paper (joint work with Efi Fogel and with Baruch Zukerman) that gives the details of the new design and compares the performance of the new arrangement package (in CGAL Version 3.2) to the previous version (CGAL Version 3.1), which has recently appeared in *Computational Geometry — Theory and Applications* (special issue on CGAL) [WFZH07]. A preliminary version of the paper appeared in the proceedings of the *1st Workshop on Library-Centric Software Design* [WFZH05].

In Chapter 3 we consider the computation of Minkowski sums and describe a robust, yet efficient, implementation of algorithms for computing planar Minkowski sums of two polygons and of a polygon and a disc. The first part of this chapter has appeared in the Proceedings of the *14th European Symposium on Algorithms* [Wei06c]. The second part, which describes an efficient offset-approximation algorithm has recently been published in *Computer-Aided Design* [Wei07].

In Chapter 4 we consider the collision-detection problem for a rotating milling-cutter, give an exact solution for some basic primitives, and use them to verify more involved motion paths. This work has appeared in the *International Journal on Computational Geometry and Applications* [WIEH05]. A preliminary version appeared in the proceedings of the *20th Annual Symposium on Computational Geometry* [WIEH04]. The work presented in Chapter 4 is joint work with Oleg Ilushin and Gershon Elber from the Technion.

The next two chapters consider the problem of planning high-quality motion paths amidst polygonal obstacles in the plane. In Chapter 5 we present the VV-complex that is capable of answering motion-planning queries and provide natural-looking paths with some preferred amount of clearance. This paper has recently been published in *Computational Geometry: Theory and Applications* [WvdBH07]; a preliminary appeared in the proceedings of the *21st Annual Symposium on Computational Geometry* [WvdBH05].

In Chapter 6 we develop a quality measure for corridors and show that the Visibility–Voronoi diagram for some preferred corridor width produces very close approximations of optimal backbone paths under this quality measure. This work has appeared in the proceedings of the *7th International Workshop on the Algorithmic Foundations of Robotics* [WvdBH06]. This, as well as the paper on the VV-complex, are joint work with Jur van den Berg from Utrecht University.

We conclude the thesis by mentioning some ongoing research and describing future prospects in Chapter 7.

Chapter 2

Exact Manipulation of Curves using Cgal Arrangements

Given a set $\mathcal{C}$ of planar curves, the *arrangement* $\mathcal{A}(\mathcal{C})$ is the subdivision of the plane induced by the curves in $\mathcal{C}$ into maximally connected cells of dimensions 0 (*vertices*), 1 (*edges*), or 2 (*faces*). The *planar map* of $\mathcal{A}(\mathcal{C})$ is the embedding of the arrangement as a planar graph, such that each arrangement vertex corresponds to a planar point, and each edge corresponds to a planar subcurve of one of the curves in $\mathcal{C}$, whose interior is disjoint from all other subcurves.

In this chapter we describe a software package that constructs and maintains arrangements of planar curves in an exact and robust manner. While the package had already existed before the research described in this thesis was conducted, the problems tackled in the thesis motivated significant further development of the software, which finally led to its complete re-design. We give an overview of this new design of the arrangement package and highlight its flexibility and extendibility. By *flexibility* we mean the ability to work with various families of curves, while *extendibility* refers to the various ways in which users can extend the package and use it for diverse applications. We also mention two related packages, for computing set-operations among polygons and for computing envelopes of curves, that will be used in the next chapters.

This is the only chapter of the thesis that focuses on implementation issues (rather than algorithms). It assumes the reader has some familiarity with modern programming techniques and paradigms. The rest of the chapters in this thesis barely require such background.

2.1 Introduction: The Cgal Library

Cgal, the Computational Geometry Algorithms Library,[1] is the product of a collaborative effort of several sites in Europe and Israel, aiming to provide a generic and robust, yet efficient, implementation of widely used geometric data structures and algorithms. The basic layer of the library comprises geometric kernels [FGK$^+$00, HHK$^+$01]. Each *kernel* defines types of constant-size primitive objects (such as points, line segments, triangles, etc.) and implements predicates and operations on objects of these types. The library also consists

[1]See the Cgal project homepage: ⟨`http://www.cgal.org/`⟩.

of a collection of modules built on top of the kernel layer, which provide implementation of many fundamental geometric data structures and algorithms. The arrangement package is a part of this layer.

In the classic computational geometry literature two assumptions are usually made to simplify the design and analysis of geometric algorithms: First, inputs are in "general position". That is, degenerate input (e.g., three curves intersecting at a common point) is precluded. Secondly, operations on real numbers yield accurate results — that is, the "real RAM" model [PS85] is used. The model also assumes that each basic geometric operation (e.g., comparing the distances of two points from a third point) takes constant time. Unfortunately, these assumptions do not hold in practice. Thus, an algorithm implemented from a textbook may yield incorrect results, get into an infinite loop, or just crash, while running on a degenerate, or nearly degenerate, input (see [KMP$^+$04, Sch00, Yap04] for examples). This is one of the problems addressed successfully by CGAL in general and by the CGAL arrangement package in particular, which adapts the *exact computation paradigm* [YD95]. The various algorithms used in the arrangement package therefore make no general-position assumptions on their input and are designed to robustly handle all possible degenerate situations (e.g., several curves intersecting at a common point, curves that tangentially intersect but do not cross one another, etc.). At the same time, these algorithms assume that the geometric primitives they use are implemented in an *exact* manner. To guarantee correct results, we have to use computations whose exactness is certified.

2.1.1 Generic Programming

The software described in this chapter rigorously adapts, as does CGAL in general, the *generic programming* paradigm [Aus98], making extensive use of C++ class-templates and function-templates. The generic-programming paradigm uses a formal hierarchy of abstract requirements on data types referred to as *concepts*, and a set of components that conform precisely to the specified requirements, referred to as *models*. Concepts correspond to template parameters, and models correspond to classes used to instantiate them.

In software engineering, *design patterns* are frequently used to supply standard solutions to common problems recurring in software design. Design patterns supply a systematic high-level approach that focuses on the relations between classes and objects, rather than designing individual components tailored for a specific programming task. See the book by Gamma *et al.* [GHJV95] for a catalog of the most common design patterns.

While relations between objects in a design pattern are usually realized in terms of abstract data types and polymorphism, design patterns can successfully be applied in generic programming as well, as done in the context of the arrangement package. The implementations of the point-location algorithms bundled with the package provide a good example. One of the most frequently used operations on arrangements is answering *point-location* queries: Given a query point q, find the arrangement cell that contains q. The arrangement package provides several point-location algorithms, and enables users to select the algorithm best suited for their application. To this end, we use the *strategy* design-pattern, which defines a family of algorithms, each implemented by a separate class, and we make them interchangeable, letting the algorithm in use vary according to the client choice.

In traditional object-oriented programming, the point-location process could be realized with an abstract base class that provides a pure virtual function, `locate(q)`, which accepts a point q, and results with the arrangement cell containing it. All concrete point-location classes would inherit from the base class, and all arrangement algorithms that issue point-location queries would use a pointer to an abstract base object, which would actually point to one of the concrete point-location classes. When using generic programming, we rely less on inheritance or virtual functions. Instead, we define a concept named *ArrangementPoint-Location*, such that all models of this concept must supply a `locate()` function. All the various point-location classes model this concept. Note that the concept definition has no trace in the actual C++ code, so from a syntactical point of view, these classes are completely unrelated. Any generic algorithm that issues point-location queries is implemented as a template parameterized by a point-location type, which must be instantiated with a model of the *ArrangementPointLocation* concept.

2.1.2 Chapter Overview

We next describe the main components of the arrangement package with an emphasis on some of the design patterns we use to allow better flexibility and extendibility for the package users. In Section 2.2 we present the main arrangement class and explain it functionality. Section 2.3 describes how the arrangement package can handle various families of curves in an exact and robust manner, reviewing methods for exact geometric computation and putting special emphasis on techniques for minimizing the trade-off between the exactness of the computation and the efficiency of the package.

Section 2.4 presents two important algorithmic frameworks — the sweep-line framework and the zone framework — used as a basis for many of the arrangement functionalities. In Section 2.5 we explain how to adapt CGAL arrangements to graphs and apply generic graph algorithms on them.

Section 2.6, which concludes this chapter, describes two additional CGAL packages that are closely related to the arrangement package: the Boolean set-operation package and the envelope package.

2.2 The Main Component

The `Arrangement_2` class-template[2] represents the planar embedding of a set of weakly x-monotone[3] planar curves that are pairwise disjoint in their interiors. It provides the necessary capabilities for maintaining the planar graph induced by the curves, while associating geometric data with the vertices, edges, and faces of the graph. The arrangement is represented using a *doubly-connected edge list* (DCEL), a data structure that enables efficient maintenance of two-dimensional subdivisions.

[2]CGAL prescribes the suffix _2 for all data structures of planar objects as a convention.

[3]A continuous planar curve C is *x-monotone*, if every vertical line intersects it at most once. Vertical segments are defined to be *weakly* x-monotone and can also be handled by the arrangement class. In what follows, the term *x-monotone* refers to weakly x-monotone as well.

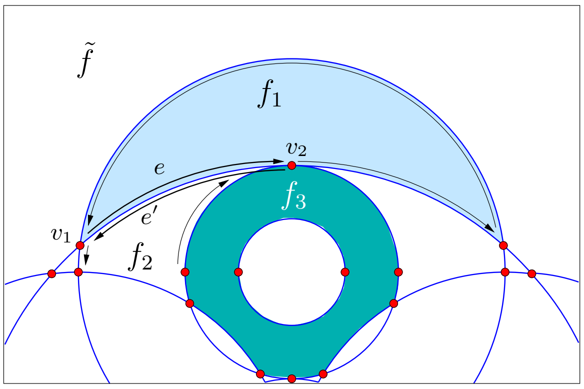


Figure 2.1: A portion of an arrangement of circles with some of the DCEL records that represent it. $\tilde{f}$ is the unbounded face. The halfedge e (and its twin e') correspond to a circular arc that connects the vertices v_1 and v_2 and separates the face f_1 from f_2. The predecessors and successors of e and e' are also shown. Note that e, together with its predecessor and successor halfedges, form a closed chain representing the outer boundary of f_1 (f_1 is lightly shaded). Also note that the face f_3 (darkly shaded) has a more complicated structure, as it contains a hole.

In a common DCEL data-structure each curve is represented using a pair of directed *halfedges*, one directed from the xy-lexicographically smaller endpoint of the curve to its larger endpoint, and the other — its *twin* halfedge — going in the opposite direction. The DCEL structure consists of containers of *vertices* (associated with planar points), *halfedges*, and *faces*, where halfedges are used to separate faces and to connect vertices. We store a pointer from each halfedge to the face lying to its left. Moreover, halfedges are connected in circular lists and form chains, such that all edges of a chain are incident to the same face and wind in a counterclockwise direction along its outer boundary (see Figure 2.1 for an illustration). A face may be simply connected, or it may store a non-empty container of *holes*, where each hole is represented by an arbitrary halfedge on the clockwise-oriented chain that forms its boundary. The full details concerning the DCEL structure are omitted here; see [dBvKOS00, Section 2.2] for further details and examples. We also extend the DCEL data-structure, allowing *isolated vertices* to be located in the interior of a face, such that each face stores a container of the isolated vertices contained in its interior.

The `Arrangement_2<Traits,Dcel>` class-template must be instantiated with two classes as follows:

- A traits class, which provides the geometric functionality, and is tailored to handle a specific family of curves. It encapsulates implementation details, such as the number-types used, the coordinate representation, and the geometric or algebraic computation methods; see Section 2.3 for more details.

- A DCEL class, which represents the underlying topological data structure. It defaults to `Arr_default_dcel<Traits>`, which simply associates a point with each DCEL vertex and an x-monotone curve with each halfedge pair, where the geometric types of the point and the x-monotone curve are defined by the traits class given as the `Traits` parameter. Users may extend the default DCEL implementation, in order to attach

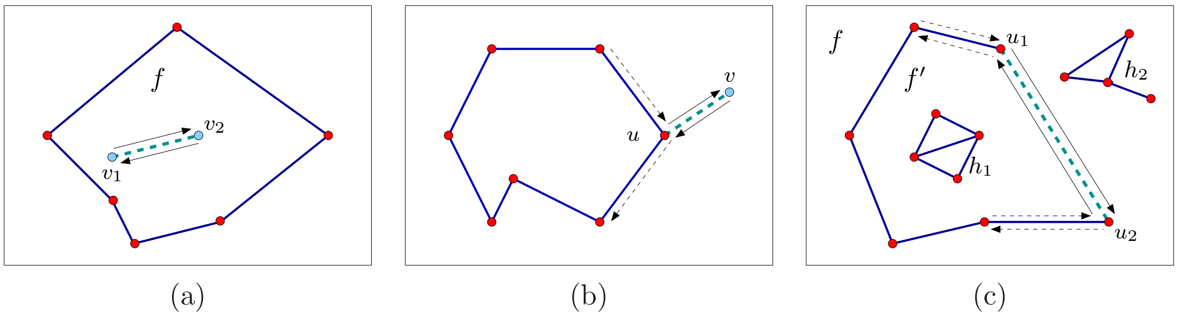


Figure 2.2: The basic insertion procedures. The inserted x-monotone curve is drawn with a light dashed line, surrounded by two solid arrows that represent the twin halfedges added to the DCEL. Existing vertices are shown as dark dots while new vertices are shown as light dots. Existing halfedges that are affected by the insertion operations are drawn as dashed arrows. (a) Inserting a subcurve into the interior of face f, which becomes a hole of this face. (b) Inserting a subcurve, one endpoint of which corresponds to the existing vertex u. (c) Inserting a subcurve, both endpoints of which correspond to the existing vertices u_1 and u_2. In this case, the new pair of halfedges close a new face f', where the hole h_1, which used to belong to f, now becomes an enclave in this new face.

additional data to the DCEL records, or even supply their own DCEL class written from scratch.

The two template parameters enable the separation between the topological and geometric aspects of the planar subdivision. This separation is advantageous, as it allows users with limited expertise in computational geometry to employ the package with their own representation of any special family of curves. They must however supply the relevant traits-class types and methods, which mainly involve algebraic computation. The separation is enabled by the modular design and conveniently implemented within the generic-programming paradigm.

The interface of `Arrangement_2` consists of various methods that enable the traversal of arrangement features. Namely, the class supplies iterators over its vertices, halfedges, or faces. In addition, the classes `Vertex`, `Halfedge`, and `Face`, nested in the `Arrangement_2` class, supply in turn methods for local traversal. For example, it is possible to visit all halfedges incident to a specific vertex, or to go over all the halfedges along the outer boundary of a given face.

In addition to the traversal methods, the arrangement class also supports several methods that locally modify the arrangement, such as inserting an isolated point, inserting an x-monotone curve, splitting an edge, or removing an edge. In case of insertion, the interior of the inserted curve must be disjoint from all other arrangement vertices and edges, but its endpoints may coincide with existing vertices (see Figure 2.2, which illustrates the various insertion cases). This interface seems rather limited; in Section 2.4 we explain how we use these basic insertion procedures to insert new curves that intersect the arrangement.

2.3 Arrangement-Traits Classes

As mentioned in the previous section, the `Arrangement_2` class-template is parameterized by a geometric *traits* class that defines the abstract interface between the arrangement data-structure and the geometric primitives it uses. The name "traits" was given by Myers [Mye97] for a concept, a model of which supports certain predefined methods that have a common denominator. In our case, a geometric traits class defines the family of curves that induce the arrangement. Moreover, details such as the number type used to represent coordinates, the type of coordinate system used (i.e., Cartesian or homogeneous), the algebraic methods used, and auxiliary data stored with the geometric objects, if present, are all determined by the traits class and are encapsulated within it.

2.3.1 The Arrangement-Traits Concepts

The traits concept is factored into a hierarchy of refined concepts. The refinement hierarchy is defined according to the identified minimal requirements imposed by different algorithms that operate on arrangements. This fact makes the production of new traits classes an easier task, as implementers can choose to write a model to the tightest concept that fits their needs. The concept hierarchy also increases the usability of the arrangement class and its related algorithms.

Every model of the geometry-traits concept must define two types of objects, namely `Point_2` and `X_monotone_curve_2`. The latter represents a planar x-monotone curve, and the former is the type of the endpoints of the curves, representing a point in the plane. The basic concept *ArrangementBasicTraits_2* lists the minimal set of predicates on objects of these two types sufficient to enable the operations provided by the `Arrangement_2` class-template itself, namely the insertion of x-monotone curves that are interior disjoint from any vertex and edge in the arrangement.

Compare x: Compare the x-coordinates of two given points.

Compare xy: Compare two points lexicographically, by their x-coordinates, and in case of equality by their y-coordinates.

Min/max endpoint: Return the lexicographically smaller (left), or the lexicographically larger (right), endpoint of a given x-monotone curve.

Is vertical: Determine whether a weakly x-monotone curve is a vertical segment.

Compare y at x: Given an x-monotone curve C and a point $p = (x_0, y_0)$ such that x_0 is in the x-range of C (namely x_0 lies between the x-coordinates of C's endpoints), determine whether p is above, below, or lies on C.

Compare y to right: Given two x-monotone curves C_1 and C_2 that share a common left endpoint p, determine the relative position of the two curves immediately to the right of p.

The set of predicates listed above is also sufficient for answering point-location queries by the various point-location strategies.

If users wish to construct arrangements of x-monotone curves that may intersect in their interior, they must instantiate the arrangement class-template with a traits class that models the concept *ArrangementXMonotoneTraits_2*. This concept refines the basic arrangement-traits concept described above, as it adds methods for computing *intersections* between x-monotone curves. An intersection point between two curves is also represented by the `Point_2` type. The refined traits concept also lists a method for splitting curves at these intersection points to obtain a set of interior disjoint subcurves. A model of the refined concept must therefore provide the following additional operations:

Intersect: Compute the intersections between two given x-monotone curves C_1 and C_2, returning them in increasing lexicographical order. Each intersection is represented by an intersection point and its *geometric multiplicity*,[4] if the multiplicity is defined and known, or by an x-monotone curve representing an overlapping portion of C_1 and C_2.

The introduction of multiplicity of intersection points enables the arrangement construction algorithms to exploit the geometric knowledge they may have, in order to avoid costly calls to other traits-class functions, since in many cases the order of incident curves to the right of a common intersection point can be deduced from their order to the left of the point and the intersection multiplicity.[5]

Split: Split a given x-monotone curve C at a given point p, which lies in C's interior, into two subcurves.

Merge: The reverse of the split operation, namely merging two contiguous subcurves to form a single x-monotone curve, is optional. If provided, it can be used for eliminating redundant vertices from the arrangement.

The construction of an arrangement of *general* curves requires a model of the further refined concept *ArrangementTraits_2*. In addition to the point and x-monotone curve types, a model of the refined concept must define a third type that represents a general (not necessarily x-monotone) curve in the plane, named `Curve_2`. It also has to supply a method that subdivides a given curve into simple x-monotone subcurves, and possibly isolated points.[6]

2.3.2 Traits Classes for Linear Curves

Line segments probably form the simplest family of curves that induce planar arrangements, and have some properties that makes the implementation of a traits class for segments a relatively easy task: for example, a pair of line segments have at most one transversal

[4]See, e.g., ⟨`http://en.wikipedia.org/wiki/Intersection_number`⟩ for an exact definition.

[5]This is quite clear when we have two curves intersecting at a point, as they swap their relative order if and only if the multiplicity of intersection is odd. See [FHK+06] for a generalization to the case of multiple curves intersecting at a common point.

[6]For example, the curve $(x^2+y^2)(x^2+y^2-1)=0$ comprises two x-monotone circular arcs, which together form the unit circle, and a singular isolated point at the origin.

intersection point (of multiplicity 1). It is also possible to consider continuous chains of line segment known as *polylines*. Polylines are slightly more difficult to handle, as they are not necessarily x-monotone and may form complex intersection patterns. However, both these families of (piecewise) linear curves share an important property: if the points that define the curves all have rational coordinates, then all intersection points they induce also have rational coordinates. As a consequence, to ensure the reliability of the various geometric operations required by the arrangement-traits concept, it is sufficient that the traits class uses exact rational arithmetic. We note that even if the input is given as floating-point numbers, we can easily convert them to a rational representation as they have a bounded mantissa.

Implementing a C++ class that represents an exact rational number-type is not too difficult for an experienced programmer. There are also several implementations that are publicly available, the most efficient one given by the number-types defined in GMP, the Gnu Multi-Precision library.[7] However, exact computation with rational numbers is far more expensive compared with the machine-precision floating-point arithmetic. To overcome this problem, it is possible to use floating-point filters. In geometric computing, when we compute some expression we are usually not interested in its exact value, and instead just have to consider the sign of the expression. It is thus possible to first perform the computation using fast floating-point arithmetic, and only in the relatively rare cases when inexactness results in an ambiguous result (namely, when the absolute value of the expression is too small), resort to exact computation and obtain the correct answer. Such a mechanism is provided in CGAL as a number-type that uses interval arithmetic to filter exact computations with rational numbers [BBP01]. The idea is to isolate the true value of the expression in an interval defined by two floating-point numbers, and in addition to store an *expression tree* that records the basic arithmetic operations that form the expression at hand. The expression tree allows for the re-computation of the expression using exact arithmetic when necessary — namely, when the interval contains 0 and its sign cannot be determined.

To minimize the computation efforts even further, it is possible to apply filtering at a geometric level. For example, instead of storing two expression trees for the x and y-coordinates of an intersection point p, it is sufficient to record that p is the intersection point of two line objects ℓ_1 and ℓ_2. This principle of geometric filtering is implemented by CGAL's "lazy" kernel [FP06].

Segment-Traits Classes

The arrangement package provides two traits classes that handle line segments. The `Arr_segment_traits_2<Kernel>` class-template is parameterized by a geometric *kernel*, that conforms to the CGAL-kernel concept [FGK$^+$00]. We note that the `Segment_2` type defined by most CGAL kernels is represented only by its two endpoints. When a segment is split several times, the bit-length needed to represent the coordinates of its endpoints may grow exponentially (see [FWH04] for a discussion), which may significantly slow down the computation. Therefore, instead of using the `Kernel::Segment_2` type, our traits class represents

[7]See GMP's homepage at ⟨`http://www.swox.com/gmp/`⟩.

a segment by its supporting line and two endpoints. When it computes an intersection point of two line segments, it uses the coefficients of their supporting lines. When a segment is split at an intersection point, the supporting line of the two resulting sub-segments remains the same, and only their endpoints are updated. The `Arr_segment_traits_2<Kernel>` thus overcomes the undesired effect of the cascading of intersection-point representation.

The `Arr_non_caching_segment_basic_traits_2<Kernel>` class-template is a model of the *ArrangementBasicTraits_2* concept. It declares `Kernel::Segment_2` as its x-monotone-curve type, and it uses the kernel functions to operate on such segments. As the segments it handles are non-intersecting, the undesired effect of cascaded representation of intersection points does not occur. The traits class `Arr_non_caching_segment_traits_2<Kernel>` models the concept *ArrangementTraits_2*. It extends the basic-traits class with the capability to handle intersections of segments. Naturally, it uses less space than the traits class `Arr_segment_traits_2` uses. However, it achieves (slightly) faster running times only in case of very sparse arrangements of line segments. In most cases the `Arr_segment_traits_2` class is more efficient than the "non-caching" traits class.

As mentioned in the introduction to this section, optimal performance is achieved when instantiating the segment-traits class with the "lazy" kernel. While guaranteeing the correctness of the result, our experiments show that using the lazy kernel is typically only 10–20% slower than employing a kernel that uses inexact machine-precision arithmetic.

Polyline-Traits Classes

Continuous piecewise linear curves, referred to as polylines, are of particular interest, as they can be used to approximate more complex curves. At the same time they are easier to deal with in comparison to higher-degree algebraic curves, as rational arithmetic is sufficient to carry out exact computations on polylines.

Previous releases of CGAL included a stand-alone polyline-traits class, which represented a *polyline* as a list of points, and performed all geometric operations on this list [Han00]. The current version (CGAL 3.2) introduces the `Arr_polyline_traits_2<SegmentTraits>` class-template, which must be instantiated with a geometric traits class that is able to handle line segments. A polyline curve is represented as a vector of `SegmentTraits::X_monotone_curve_2` objects (namely segments). Unlike the earlier version, the new polyline-traits class does not perform any geometric operations directly. Instead, it solely relies on the functionality of the instantiated segment-traits class. For example, when we need to determine the position of a point with respect to an x-monotone polyline, we use binary search to locate the relevant segment that contains the point in its x-range, then we compute the position of the point with respect to this segment. Thus, operations on x-monotone polylines of size m typically take $O(\log m)$ time.

Users are free to choose the underlying segment-traits class based on the number of expected intersection points (see the discussion in the previous subsection). Moreover, it is possible to instantiate the polyline-traits class-template with a traits class that handles segments with some additional data attached to them (see Section 2.3.4). This makes it possible to associate different data objects with the different segments that form a single polyline. This can come in handy if we have polyline curves that represent roads, and we would like to mark the speed limit (which can of course change along the road) for every segment of the road.

2.3.3 Traits Classes for Non-Linear Curves

Handling non-linear curves and surfaces in an exact manner has attracted a lot of attention from several research groups during the recent few years; see, e.g., [DLLP03, EK06], in addition to the other references given in this section.

The most important type of curves we consider are algebraic curves, or segments of such curves. An *algebraic curve* of degree d is defined by the locus of all points (x, y) in the plane satisfying the equation:

$$\sum_{i=0}^{d} \sum_{j=0}^{d-i} c_{ij} x^i y^j = 0 \ .$$

That is, an algebraic curve of degree d is defined by a bivariate polynomial of degree d. If all curve coefficients c_{ij} are rational, we call it a *rational algebraic curve.*

The main difficulty is that in the general case, the coordinates of intersection point between two rational curves of degrees d_1 and d_2, respectively, are irrational — they are algebraic number of degree $d_1 d_2$.[8]

At first glance, exact computation with algebraic expressions seems impossible — after all, in contrast with rational numbers, we need infinitely many bits to represent an algebraic number. Indeed, it is impossible to obtain an exact binary representation of the value of an algebraic expression E, but it is possible to determine its sign in a certified way. Certified computation with algebraic numbers relies on the theory of separation bounds [Mig82]. For each algebraic expression E, there exists a *separation bound* $\text{sep}(E) > 0$, which can be easily evaluated from the structure of E, such that either $|\text{val}(E)| > \text{sep}(E)$ or $\text{val}(E) = 0$. In other words, the value of a non-zero algebraic expression cannot be arbitrarily small. See, for example, [BFM+01, LY01] for constructive proofs of the existence of separation bounds. It is therefore possible to exploit this fact and compute an approximation $\text{app}(E)$ of the value of E with a finite number of bits, such that $|\text{val}(E) - \text{app}(E)| < \delta$. If $|\text{app}(E)| > \delta$, then $\text{sign}(\text{val}(E)) = \text{sign}(\text{app}(E))$ and we are done. Otherwise, we have to check whether $\delta < \frac{1}{2}\text{sep}(E)$. If so, $|\text{val}(E)| < \text{sep}(E)$ and we conclude the expression equals zero. Otherwise, we repeat the process, this time with more bits of precision, such that the value of the error bound δ is halved. Note that the first iteration of the evaluation of E, which is usually done using machine-precision floating-point arithmetic, suffices in most cases to correctly compute $\text{sign}(E)$. It thus serves as a floating-point filter for the expensive computation process.

We mention that it is not known how to compute the maximal separation bound for a given expression E. The value of $\text{sep}(E)$ we use is usually some easily computable under-estimation of this bound. A main challenge in this area is to find a good estimation of the

[8] A real number $\alpha \in \mathbb{R}$ is called *algebraic* if there exists a non-trivial univariate polynomial $P(x)$ with *integer coefficients* such that $P(\alpha) = 0$. We say that α is of degree d if $\deg(P) = d$, and if P divides any other polynomial with integer coefficients $\hat{P}(x)$ such that $\hat{P}(\alpha) = 0$.

separation bound, namely make sep(E) as large as possible; see, e.g., [BFM+01, LY01] for some recent results.

The CORE library[9] [KLPY99] and the numerical facilities of LEDA[10] [MN00, Chapter 4]) include number-type classes that can perform certified calculations with algebraic numbers. Yet computing with algebraic numbers is far more difficult and computationally demanding than operating on rational numbers. Moreover, CORE is currently capable of performing simple arithmetic operations on algebraic numbers but cannot perform cascaded root operations (namely one cannot compute a root of a polynomial whose coefficients are themselves irrational algebraic numbers). This operation is possible in LEDA, but may incur a prohibitive running-time penalty. In the rest of this subsection we overview some of the traits classes for curved objects and describe the algebraic foundations of their implementation. We also explain how we avoid the more computationally demanding operations to keep reasonable running times.

In this context we should also mention the SYNAPS library,[11] which provides efficient solvers for polynomial equations and handles exact comparisons of algebraic numbers using the Sturm sequences of the originating polynomials. SYNAPS contains optimized comparison procedures for algebraic numbers of small degree, based on the work of Emiris and Tsigaridas [ET04], that are very convenient for computing with low-degree algebraic curves. However, the algebraic computations of SYNAPS are not numerically filtered, thus the running-time penalty they incur is relatively high.

The Circle/Segment Traits Class

The development of this traits class, which can handle circular arcs and line segments, was primarily motivated by the task of computing the union of large sets of generalized polygons (having also circular edges) that comprise a VLSI model. The success in operating on very large data sets also led to the development of the algorithm for approximating offset polygons (see Section 3.4.2) which is based on this circle/segment traits-class.

Given rational numbers $\alpha, \beta, \gamma \in \mathbb{Q}$ with $\gamma > 0$, the real number $\alpha + \beta\sqrt{\gamma}$ is called a *one-root number*. The term "one-root number" was given by Berberich *et al.* [BEH+02]. In their work, they used the fact that one-root numbers can be handled by LEDA rather efficiently; this gave them the ability to compare two such numbers in an exact manner. In our context, one-root numbers play an important role, since the solution of any quadratic equation with rational coefficients (namely $ax^2 + bx + c = 0$, where $a, b, c \in \mathbb{Q}$) is a one-root number, as it equals $\frac{-b}{2a} \pm \frac{1}{2a}\sqrt{b^2 - 4ac}$.

Observe that if $x = \alpha + \beta\sqrt{\gamma}$ is a one-root number and $q \in \mathbb{Q}$, then $x \pm q$, $x \cdot q$ and $\frac{x}{q}$ are obviously one-root numbers. In addition,

$$\frac{q}{x} = \frac{q}{\alpha + \beta\sqrt{\gamma}} = \frac{q \cdot (\alpha - \beta\sqrt{\gamma})}{\alpha^2 - \beta^2\gamma} \ ,$$

[9]See CORE's homepage at ⟨`http://www.cs.nyu.edu/exact/core/`⟩.
[10]See ⟨`http://www.algorithmic-solutions.com/enleda.htm`⟩.
[11]See SYNAPS' homepage at ⟨`http://www-sop.inria.fr/galaad/logiciels/synaps/`⟩.

and

$$x^2 = (\alpha + \beta\sqrt{\gamma})^2 = (\alpha^2 + \beta^2\gamma) + 2\alpha\beta\sqrt{\gamma}\ ,$$

are also one-root numbers.

Lemma 2.1 *It is possible to determine the sign of a one-root number $x = \alpha + \beta\sqrt{\gamma}$ using only rational arithmetic.*

Proof: In case that sign(α) = sign(β), then this is also the sign of the entire expression x and we are done. Otherwise, we have to compare $|\alpha|$ and $|\beta|\sqrt{\gamma}$, such that the sign of x is the sign of the term whose absolute value is larger. But this is easily done be comparing α^2 and $\beta^2\gamma$, both are rational numbers. □

Lemma 2.2 *It is possible to compare two one-root numbers $x_1 = \alpha_1 + \beta_1\sqrt{\gamma_1}$ and $x_2 = \alpha_2 + \beta_2\sqrt{\gamma_2}$ using only rational arithmetic.*

Proof: We first note that if $\beta_2 = 0$ then $x_2 \in \mathbb{Q}$, and we can perform the comparison by evaluating the sign of the one-root number $x_1 - \alpha_2$. (Similarly, if $\beta_1 = 0$ we check the sign of $x_2 - \alpha_1$.)

If both $x_1 = \alpha_1 + \beta_1\sqrt{\gamma_1}$ and $x_2 = \alpha_2 + \beta_2\sqrt{\gamma_2}$ are non-trivial one-root numbers, then comparing them is equivalent to comparing $\alpha_1 - \alpha_2$ and $\beta_2\sqrt{\gamma_2} - \beta_1\sqrt{\gamma_1}$. We therefore compute the sign of $\beta_2\sqrt{\gamma_2} - \beta_1\sqrt{\gamma_1}$ (this is easily done be comparing $\beta_1^2\gamma_1$ and $\beta_2^2\gamma_2$, if sign(β_1) $\neq$ sign(β_2)) and check whether it is equal to the sign of $\alpha_1 - \alpha_2$. If the two terms have different signs we can deduce the comparison result at this stage. Otherwise, we continue by squaring both terms, such that the comparison result is equivalent to evaluating the sign of the one-root number $((\alpha_1 - \alpha_2)^2 - (\beta_1^2\gamma_1 + \beta_2^2\gamma_2)) + 2\beta_1\beta_2\sqrt{\gamma_1\gamma_2}$ (if both terms are negative, we have to negate the sign). As we showed in Lemma 2.1, this can be done using only rational arithmetic. □

We finally note that in the general case, given two non-trivial one-root numbers $x_1 = \alpha_1 + \beta_1\sqrt{\gamma_1}$ and $x_2 = \alpha_2 + \beta_2\sqrt{\gamma_2}$ (with $\beta_1, \beta_2 \neq 0$), the numbers $x_1 \pm x_2$, $x_1 \cdot x_2$ and $\frac{x_1}{x_2}$ are *not* one-root numbers, unless of course $\gamma_1 = q^2\gamma_2$, where $q \in \mathbb{Q}$. We next explain how we design a traits class for handling circular arcs and line segments, while taking special care so that the traits-class methods do not invoke that kind of operations.

The class-template `Arr_circle_segment_traits_2<Kernel>` (known as the *circle/segment traits class*) can handle curves that are either:

- Arcs of *rational circles*, namely circles of the form $(x - x_0)^2 + (y - y_0)^2 = R$, where the circle center (x_0, y_0) has rational coordinates and the *squared* radius R is also rational. Note that the radius itself may not be rational.

  A general circular arc is given by its supporting circle, two endpoints s and t that satisfy the equation of the circle (these endpoints may be rational, or have one-root coordinates), and the orientation of the arc between the endpoints (clockwise or counterclockwise). An arc may also represent a whole circle.

- Segments of *rational lines*, namely lines whose equation is $ax + by + c = 0$, where a, b and c are rational. The segment is given by its supporting line and its two endpoints s and t, whose coordinates can either be rational or one-root numbers.

Note that the coordinates of the intersection points of two rational circles, or of a rational circle and a rational line, are one-root numbers, as they are the roots of quadratic equations with rational coefficients. Therefore, when we split a circular arc at its intersection point with another arc or with a line segment, the two resulting arcs are representable by the curve type defined by the traits class. Also note that a non x-monotone circular arc can be subdivided into three x-monotone arcs at most, depending on whether it contains the two points with one-root coordinates $(x_0 \pm \sqrt{R}, y_0)$. A whole circle is subdivided into two x-monotone arcs. Moreover, we can label each x-monotone as a *"lower"* arc, if it lies below the horizontal line $y = y_0$, or as an *"upper"* arc, if it lies above this line.

We next describe how the traits class implements the set of predicates and constructions involving circular arcs and points with one-root coordinates, as listed by the *ArrangementTraits_2* concept (see Section 2.3.1). The treatment of line segments is simple and straightforward, so we omit its details here.

Compare xy: As all coordinates are one-root numbers, these operations can be easily performed using rational arithmetic, as shown in Lemma 2.2.

Point position: Given an x-monotone arc C of the rational circle $(x - x_0)^2 + (y - y_0)^2 = R$ and a point $p = (\hat{x}, \hat{y})$, we need to determine whether p is above, below, or lies on C. If C is a lower arc, it lies below the horizontal line $y = y_0$, so if $\hat{y} > y_0$, p obviously lies above C. Otherwise, we have to substitute p into the equation of the supporting circle, but as $\hat{x}$ and $\hat{y}$ are one-root numbers (and not just rational numbers), we have to be a bit careful: we compare the one-root numbers $z_1 = (\hat{y} - y_0)^2$ and $z_2 = r^2 - (\hat{x} - x_0)^2$. The point p lies above C if $z_1 < z_2$, below C if $z_1 > z_2$, and on C in case of equality. Evaluating the predicate for an upper arc is symmetric.

Compare to right: Given two x-monotone arcs C_1 and C_2 that share a common left endpoint $p = (\hat{x}, \hat{y})$, we can determine their relative vertical position immediately to the right of p by comparing the slopes of the two circles there. As the slope of C_j at p is given by $\frac{\hat{x} - x_j}{y_j - \hat{y}}$, where (x_j, y_j) is C_j's center, it is easy to show that comparing these two slopes is equivalent to comparing the two one-root numbers $y_2(\hat{x} - x_1) - y_1(\hat{x} - x_2)$ and $\hat{y}(x_2 - x_1)$. In case of equality, the two supporting circles are tangent at p, and we can simply compare their radii in order to determine their relative order near the tangency point.

Intersect: Given two x-monotone arcs C_1 and C_2, We first compute the intersection points between their supporting circles by solving two quadratic equations whose solutions are the x and y-coordinates of the intersection point. For each intersection point $p = (\hat{x}, \hat{y})$, we have to make sure it really lies on both C_1 and C_2: for this, we simply check whether $\hat{y}$ is above or below the y-coordinate of C_j's supporting circle (depending on whether C_j is an upper or a lower arc), and if so we just need to check whether it is in the x-range of the arc.

The circle/segment traits class is therefore capable of constructing and maintaining arrangements of arcs of rational circles and segments of rational lines. It does so using only *exact rational arithmetic* and avoids computations with algebraic numbers of degree 2 that involve square roots. This fact makes it very efficient. It usually performs about an order of magnitude faster than a straightforward implementation that relies on the exact squared-root operation provides by LEDA or by CORE (see, for example, the experimental results reported in [WZ06] and in [WFZH07]).

The Conic-Traits Class

Conic curves are algebraic curves of degree 2, namely the locus of all points (x, y) in the plane satisfying the equation:

$$c_{2,0}x^2 + c_{0,2}y^2 + c_{1,1}xy + c_{1,0}x + c_{0,1}y + c_{0,0} = 0 \ .$$

The type of the conic curve is in general determined by considering the sign of the so-called *conic discriminant* $\Delta_c = 4c_{2,0} \cdot c_{0,2} - c_{1,1}^2$:

- If $\Delta_c > 0$, then the curve is an *ellipse*, which is a bounded and connected curve.
- If $\Delta_c < 0$, then the conic curve is a *hyperbola*, which is an unbounded curve with two connected branches.
- If $\Delta_c = 0$, then the curve is a *parabola*, which is an unbounded and connected curve.

In addition, degenerate forms of conic curves, such as a line (where $c_{2,0} = c_{0,2} = c_{1,1} = 0$) or a line-pair (e.g. $(x + y - 3)(4x - 5y + 2) = 0$), also exist. Conic arcs are defined as finite segments of conic curves, and are characterized by their supporting conic and two endpoints.

Conic arcs are important as they occur in many useful geometric constructs, such as offset polygons (see Section 3.4 for the exact details) and Voronoi diagrams of circles and of line segments [EK06, Kar04].

Intersection points between two general conic curves with rational coefficients have algebraic coordinates of degree 4. The `Arr_conic_traits_2` class included in the arrangement package utilizes the algebraic number-type provided by the CORE library, known as `CORE::Expr`, and relies on its ability to carry out certified algebraic computations. Some precautions have to be taken, however. For example, given two conic curves, we can compute the x-coordinates of the intersection points, then obtain the y-coordinates by substituting these x-coordinates into the equation of one of the conics and solving a quadratic equation. This would however result in a very complicated representation of the y-coordinates. The approach the conic-traits class uses is to compute the x and the y-coordinates of the intersection points separately, using resultant calculus, then pair them together and form the intersections points; see more details in [Wei02a]. The traits class also utilizes caching mechanisms and some of the high-level filtering techniques presented in [Wei02b] to avoid redundant (and computationally expensive) geometric and algebraic operations, replacing them by simpler combinatorial tests.

Other Traits Classes

The arrangement package also includes a traits class for arcs of graphs of rational functions. A *rational function* is given by $y = \frac{P(x)}{Q(x)}$, where $P(x)$ and $Q(x)$ are polynomials with rational coefficients. Rational functions are widely used to interpolate or simplify more complicated curves (recall that the polynomial function $y = P(x)$ is a special case of a rational function). At the same time, compared to general algebraic curves that are given in implicit representation, rational functions are relatively easy to handle: it is easy to intersect two rational functions, to compute the derivatives of a rational function, or to locate a point on a rational curve given its x-coordinate. The class `Arr_rational_arc_traits_2` can handle arcs of rational functions of arbitrary degree. It is also based on the algebraic number-type provided by the Core library.

Additional traits classes that are also compatible with the arrangement-traits concepts have been developed by other groups of researchers. Among these we can list traits classes for circular arcs and for conic arcs developed by Emiris *et al.* [EKP+04], extending the predicates described by Devillers *et al.* [DFMT02]. The work of [EKP+04] is available in Cgal Version 3.2, under the circular kernel package.

Traits classes for conic curves [BEH+02], cubic curves [EKSW04], and special types of quartic curves [BHK+05] were developed as part of the Exacus project [BEH+05]. These Exacus-based traits classes are planned to be integrated into future versions of Cgal, as the basis for a curved kernel that handles algebraic curves of arbitrary degree.

2.3.4 The Traits-Class Decorators

The *decorator* design-pattern is used to dynamically attach additional responsibilities to an object [GHJV95]. In the context of the arrangement package, we use traits-class decorators to extend the geometric entities defined by the traits class with additional, possibly non-geometric, data. For example, we may wish to construct an arrangement of line segments that are either *red* or *blue* and examine the vertices induced by red–blue intersections.

The `Arr_curve_data_traits_2<BaseTraits,XData,Merge,CData,Convert>` template enables the extension of the curve types defined by a geometric base-traits class, which must be a model of the *ArrangementTraits_2* concept. It inherits its `Curve_2` from the curve type defined in the base-traits class and extends it with an additional data field of type `CData`. The `X_monotone_curve_2` type is also inherited from the corresponding type in the base-traits class and is extended with a data field of type `XData`. The class also supplies constructors and methods to access the data fields of its extended curve types.

The curve-data traits-decorator derives itself from the base-traits class and relies on the geometric operations defined by this class. It extends these operations by maintaining the data fields associated with the curves as follows:

- When a curve is subdivided into x-monotone subcurves, its `CData` field is converted using the `Convert` functor and propagated to all subcurves. By default, the `CData` and `XData` types are the same, and the data field is simply copied to the x-monotone subcurves.

- When we split an x-monotone curve into two, its data field is duplicated and stored with both resulting subcurves.
- When two x-monotone curves overlap, their data fields are merged using the `Merge` functor and the result is stored with the resulting subcurve that represents the overlap.
- We allow the merger of two x-monotone curves, only if they are geometrically mergeable (as determined by the base-traits class) *and* their data fields are equivalent.

The `Arr_consolidated_curve_data_traits_2<BaseTraits,Data>` decorator specializes the `Arr_curve_data_traits_2` template by associating each curve with a single `Data` object and by attaching a set of `Data` objects to each x-monotone curve. This set usually contains a single data object, unless the x-monotone curve corresponds to an overlapping section of two curves or more. When a curve with a data field d is split into x-monotone subcurves, each subcurve is associated with a singleton set $\{d\}$. When two x-monotone curves overlap, the decorator takes the union of their data sets, and associates it with the resulting overlapping subcurve.

The traits-class decorators make it easy to attach external data fields to curves. Typically, all the user has to do is to assign the extra fields to the input curves upon construction and to pass the extended curves to the arrangement class. The decorator takes care of maintaining the extra data fields when the input curves are split into subcurves that are eventually associated with the arrangement edges. In the next chapters of the thesis we will come across several situations where curves need to be extended — and this is trivially done, as we have just explained.

2.4 Major Algorithmic Frameworks

While sweep-line is a well-known algorithmic framework in computational geometry and serves as the basis of many algorithms, we have also identified a second algorithmic framework that serves as the foundation of a family of concrete arrangement-related algorithms: the zone-computation framework. For instance, the implementation of operations like aggregated insertion of a set of curves into an arrangement and the overlay computation of two arrangements is based on the *sweep-line* framework, while the incremental insertion of a single curve into an arrangement involves the computation of the *zone* of this curve.

We provide two class-templates, namely `Sweep_line_2` and `Arrangement_zone_2`, that implement these two fundamental algorithms common to the two families of concrete algorithms, respectively. Among their other template parameters, both these classes are parameterized by a *visitor*, which should be a model of the appropriate visitor concept. The concrete algorithms are thus realized through *sweep-line visitors* or through *zone-computation visitors*. The visitor classes receive notifications on the events handled by the basic procedure and can construct their output structures accordingly. We gain a centralized, reusable, and easy to maintain code.[12] For example, we use sweep-line visitors to obtain a variety of dif-

[12]It is worth mentioning that the BOOST Graph Library, for example, uses visitors to support user-defined extensions to its fundamental graph algorithms; see [SLL02, Section 12.3] for details.

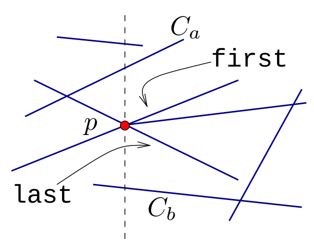
ferent results: computing all intersection points induced by a set of curves, constructing the arrangements of these curves, inserting the curves into an existing arrangement, etc. Moreover, users may introduce their own sweep-based or zone-based algorithms, as implementing such an algorithm reduces to implementing an appropriate visitor class.

2.4.1 The Generic Sweep-Line Algorithm

Sweeping the plane with a line is one of the most fundamental algorithmic frameworks in computational geometry. The famous *sweep-line* algorithm of Bentley and Ottmann [BO79] was originally formulated for sets of non-vertical line segments, with the "general position" assumptions that no three segments intersect at a common point and no two segments overlap. An imaginary vertical line is swept over the plane from left to right, transforming the static two-dimensional problem into a dynamic one-dimensional one. At each time during the sweep a subset of the input segments intersect this vertical line in a certain order. The subset of segments and their order along the sweep line may change, as the line moves along the x-axis, only at a finite number of *event points*, namely intersection points of two segments and left endpoints or right endpoints of segments. The known event points, namely segment endpoints and all intersection points that have already been discovered, are stored in an xy-lexicographic order in a dynamic *event queue*. The ordered sequence of segments intersecting the imaginary vertical line is stored in a dynamic structure called the *status line*. Both structures are maintained as balanced binary trees, such as red-black trees, that enable their efficient maintenance. In particular, we use an advanced implementation of red-black trees [Wei05] that offers extended functionality over other alternatives such as STL maps.

The `Sweep_line_2<Traits,Event,Subcurve,Visitor>` class-template implements a generic sweep-line algorithm that can handle any set of arbitrary x-monotone curves [SH89], containing all possible kinds of degeneracies (see [dBvKOS00, Section 2.1] and [MN00, Section 10.7] for the treatment of degeneracies induced by line segments), using a small set of geometric predicates and constructions involving the curves. The `Traits` parameter must be instantiated with a model of the *ArrangementXMonotoneTraits_2* concept (see Section 2.3.1). The `Visitor` parameter must be a model of the *SweepLineVisitor_2* concept, whose functionality is explained in details next.

The `Sweep_line_2` class-template uses two auxiliary classes: `Event_base`, which stores a `Point_2` object that represents the coordinates of an event point, and `Subcurve_base`, associated with a portion of an x-monotone curve (represented as an `X_monotone_curve_2` object), whose interior is disjoint from all other subcurves at the current location of the sweep line (it may intersect yet undiscovered subcurves as the sweep line advances). These two auxiliary classes also store additional data members needed internally by the sweep-line algorithm, which are not exposed to external users. The `Sweep_line_2` parameters `Event` and `Subcurve` are instantiated with these two types by default. Users may however extend these types with data required by their visitor class by inheriting an event class and a subcurve class from the respective base classes, and using these extended classes to initialize the sweep-line template.



During the sweep-line process the event objects in the event queue are sorted lexicographically, and the subcurve objects are stored in the status line in the same order as the lexicographic order of their intersection with the imaginary sweep-line. The `Sweep_line_2` class performs only the operations required to maintain the event queue and the status line, while the visitor class is responsible for producing the actual output of the algorithm. Whenever the sweep-line class handles an event point p, it sends a notification to its visitor, with the relevant `Event` object and the `Subcurve` objects incident to it. The latter is specified by a pair of iterators that define the relevant subcurve range in the status line. Using this information, the visitor can access not only the subcurves incident to p, but also the neighboring subcurves from above and below. In the example depicted on the right, the event point p is sent to the visitor with the iterator range [`first`, `last`], which defines the three subcurves that share p as a common left endpoint; the subcurves C_a and C_b lying above and below p can be accessed by dereferencing the expressions `--first` and `++last`, respectively. The sweep-line visitor is capable of attaching auxiliary data members and adding functionality to the event and subcurve objects. It can also construct its output accordingly.

It should be mentioned that Bartuschka *et al.* [BMS97, BNS00] designed and implemented a generic sweep-line algorithm in the LEDA library. They offer a class that couples a sweep-traits class with a visitor. However, in their implementation the traits class is responsible for performing almost the entire sweep-line algorithm, whereas our class performs the bare sweep-line procedure, and only requires a traits class that supplies a small set of geometric primitives. Hence, our approach provides a more modular framework that is easier to extend.

A simple sweep-line visitor class is used for reporting all intersection points induced by a set of input curves.[13] This visitor does not require storing any auxiliary data structures with events or with subcurves. The default `Event_base` and `Subcurve_base` types are sufficient and used to instantiate the sweep-line class-template. The visitor simply reports an event point p, if it has more than a single incident subcurve.

As mentioned above, a key operation implemented with the aid of a sweep-line visitor is the construction of a DCEL that corresponds to the arrangement induced by a set of input curves. The visitor class in this case is more complicated, as it needs to store extra data with the subcurves and the events as follows. The event class is extended by a handle for a DCEL vertex that corresponds to the event point. As long as the vertex has not been created yet, the handle is invalid. The subcurve class is extended with a pointer to an event point that corresponds to the left endpoint of the subcurve. When processing an event point p, it is possible to go over all subcurves such that p is their right endpoint (they lie to the left of p) and use this auxiliary data to insert the subcurves into the arrangement using one of the basic insertion methods for a non-intersecting x-monotone curve (see Section 2.2). In fact, additional information stored with each subcurve helps performing the insertion in the most efficient manner, utilizing all available geometric and topological information. We omit the related technical details here.

[13]Indeed, this operation is not directly related to arrangements. However, it is implemented using the sweep-line framework.

Another operation closely related to the construction of a Dcel structure from scratch is the aggregated insertion of new curves into an *existing* arrangement and efficiently updating an existing Dcel structure. In this case we have to sweep over the plane and account for the set $\mathcal{C}$ of new curves as well as the consolidated set of all subcurves associated with the existing Dcel halfedges. Our goal is to discover the intersections involving the new curves, and to update the existing Dcel accordingly. We first extend the x-monotone curve type defined by the traits class with a handle for one of its corresponding halfedge twins (this handle is invalid for the newly inserted curves). We do this using a traits-class decorator, as explained in Section 2.3.4. It is also possible to extend the `Subcurve` type of the visitor, but attaching the auxiliary data at the traits-class level enables a more efficient implementation of the traits-class methods. For example, it is possible for the decorator to override the function that computes intersections between curves, such that it avoids the computation of intersections between two curves that already correspond to valid halfedges. This way we can filter the unnecessary geometric operations and perform only the ones in which newly inserted curves are involved.

2.4.2 Overlaying Arrangements

A fundamental operation on arrangements that is straightforwardly implemented using a sweep-line visitor is the *overlay* of two given arrangements. We refer to the two input arrangements as the *blue* and the *red* arrangements. We compute their overlay by sweeping a vertical line over the plane, processing a consolidated set of the blue and red curves. As explained in the previous subsection, it is convenient to use an extended traits class that extends the x-monotone curves with a color attribute (whose value is either `BLUE` or `RED` in our case) and a halfedge handle. The extended traits class helps us to filter out unnecessary computations. For example, we can ignore "monochromatic" intersections, and compute only red–blue intersection points (or overlaps). This way the arrangement of a consolidated set of blue and red curves is computed efficiently.

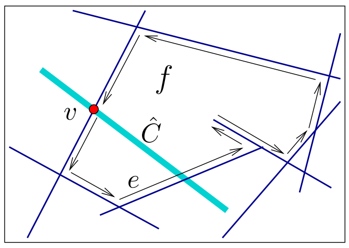
The major added difficulty over the previously mentioned visitors is the need to construct a Dcel that properly represents the overlay of two potentially extended input arrangements. That is, the features of each one of the two Dcel data-structures that represent the two respective input arrangements could have been extended with additional data. If we put our arrangements one on top of the other, we get an arrangement whose faces correspond to overlapping regions of the blue and red faces. An edge in the overlaid arrangement may be a blue edge, a red edge, or an overlap of two differently colored edges. An overlay vertex may be a blue vertex, a red vertex, a coincidence of two differently colored vertices, or it may correspond to a blue–red intersection. In each case, the data associated with the overlaid Dcel feature should be computed from the data associated with the red and blue Dcel features that induce it. To this end, the overlay visitor is parameterized by an overlay-traits type, which defines the merger operations between various Dcel features, achieving maximum genericity and flexibility for the users; see [FWZH06] for the technical details.

2.4.3 Zone-Computation Visitors

Many applications can make use of the following operation: Given an arrangement $\mathcal{A}$ and an x-monotone curve C, compute the *zone* of C in $\mathcal{A}$. That is, identify all arrangement cells that the curve crosses. The zone can be computed by locating the left endpoint of C in the arrangement, and then "walking" along the curve towards the right endpoint, keeping track of the vertices, edges, and faces crossed on the way (see, for example, [dBvKOS00, Section 8.3] for the computation of the zone of a line in an arrangement of lines).

The primary usage of the zone-computation algorithm is the incremental insertion of an x-monotone curve into the arrangement. However, it is sometimes necessary to compute the zone of a curve in an arrangement without actually inserting the curve. In other situations, the entire zone is not required, as in the case of a process that only checks whether a query curve passes through an existing arrangement vertex. If the answer is positive, the process can terminate as soon as the vertex is located.

While the sweep-line algorithm operates on a set of input x-monotone curves and its visitors can just use the notifications they receive to construct their output structures, the zone-computation algorithm operates on an arrangement object and its visitors may *modify* the same arrangement object as the computation progresses. This makes the interaction of the main class with its visitors slightly more intricate.



The template `Arrangement_zone_2<Arrangement,Visitor>` implements a generic zone-computation algorithm. It is parameterized by an arrangement type and by a visitor type. Given a curve C, the zone visitor is notified whenever a maximal subcurve $\hat{C}$ of C is found, and $\hat{C}$ is reported. The interior of every reported subcurve does not coincide with any arrangement vertex or halfedge and lies within a face f. The arrangement features that define the subcurve endpoints are also reported, as well as the face f. In the example depicted to the right, the interior of $\hat{C}$, a maximal subcurve of the line segment whose zone we compute (drawn in a thick light line) is contained in the face f whose outer boundary is also shown; the vertex v corresponds to $\hat{C}$'s left endpoint, while the right endpoint lies on the halfedge e. Thus, if the visitor inserts this subcurve into the arrangement, it first has to split e at this point. A similar notification is issued whenever a subcurve $\hat{C}$ that overlaps with an arrangement edge is detected. In both cases, the visitor returns a pair that consists of a halfedge handle and a Boolean flag as a result. In case the visitor inserts the subcurve $\hat{C}$ into the arrangement, it returns a handle to one of the newly created halfedge twins. Otherwise, it returns an invalid handle. The Boolean value indicates whether the zone-computation process could terminate. This is conveniently used by the zone procedure to gain efficiency in those applications that do not require the computation to proceed.

The most important zone-visitor is a class that performs the incremental insertion of an x-monotone curve. It uses the notifications its receives from the zone algorithm to obtain maximal subcurves of the inserted curve. The zone visitor then inserts these subcurves into the arrangement using the basic insertion functions (Section 2.2). Other zone visitors, such as a visitor that determines whether a query curve intersects with the curves of an arrangement, are even easier to implement.

2.5 Adapting Arrangements to Boost Graphs

The Boost graph library (Bgl) [SLL02] is a generic library of graph algorithms and data structures designed in the same spirit as Stl, the Standard Template Library [Aus98]. It supports graph algorithms, and as our arrangements are embedded as planar graphs, it is only natural to augment the Dcel with the interface that the Bgl expects, and gain the ability to perform the operations that the Bgl supports, such as shortest-path computations. We adapt `Arrangement_2` instances to Boost graphs by specializing the `boost::graph_traits` template for the `Arrangement_2` class and providing a set of free functions for traversing the arrangement features that conform with the interface prescribed by the Bgl.

In addition to the straightforward adaptation, which associates a vertex with each Dcel vertex and an edge with each Dcel halfedge, we also offer a *dual* adapter, which associates a graph vertex with each Dcel face, such that two vertices are connected if and only if there is a Dcel halfedge that separates the two corresponding faces. Using this dual adapter it is possible, for example, to perform breadth-first or depth-first traversals on the arrangement faces. The dual representation is very useful for many applications, such as answering motion-planning queries (see, e.g., [HH03]). Assume that we have an arrangement of line segments and circular arcs that represents the configuration space of a disc robot moving amidst polygonal obstacles in the plane. We extend the Dcel by attaching a Boolean flag to each arrangement face, indicating whether it is *free* or *forbidden*. As the Bgl enables the applications of filters on graph vertices, we can perform a breadth-first traversal starting at a given free face and filter out faces that are marked as forbidden, considering only the free arrangement faces. This way we can efficiently answer motion-planning queries.

2.6 Related Cgal Packages

2.6.1 Operations on Polygonal Sets

Computing Boolean set operations on polygonal regions is a fundamental operation in fields like motion planning and computer-aided design. We may consider *straight-edge polygons* or *general polygons*, namely regions bounded by a closed simple chain of arbitrary x-monotone curves. Given two polygons P and Q, we may wish to compute their union, their intersection, their difference, or their symmetric difference. It is also possible that P and Q are not simple, but contain polygonal holes in their interior.

Operations on polygonal sets is one of the most important applications of the overlay procedure (see Section 2.4.2). In this section we describe a Cgal package that performs such operations and is based on the functionality of the arrangement package and the traits classes it contains.

Given two simple polygons P and Q, we construct the arrangements $\mathcal{A}_P$ and $\mathcal{A}_Q$ of the boundary segments of P and of Q, respectively. Both these arrangements comprise one bounded face, which represents the interior of the respective polygon and is marked by a *true* flag, and an unbounded face marked by a *false* flag. We can now perform any of the basic Boolean set-operations on P and Q by overlaying $\mathcal{A}_P$ and $\mathcal{A}_Q$ and using a properly defined overlay-traits class. For example, for computing $P \cap Q$ we use an overlay-traits class that performs a Boolean *and* operation on the face marks, and for computing $P \cup Q$ we use a different traits class that performs a Boolean *or* operation. The result of the set-operation consists of the overlay faces marked as *true*. It is also not difficult to generalize this scheme to perform Boolean operations on sets with polygonal outer boundaries that contain a set of disjoint polygonal holes in their interior, or even on sets that comprise a collection of pairwise disjoint polygonal regions of this nature.

If we are given a set $P_1, \dots P_n$ of polygonal regions (possibly with holes) and wish to compute their *multi-way* union $\bigcup_{k=1}^{n} P_k$, it is convenient to employ a divide-and-conquer approach and recursively compute $\bigcup_{k=1}^{h} P_k$ and $\bigcup_{k=h+1}^{n} P_k$, where $h = \lfloor \frac{n}{2} \rfloor$. We can then compute the overall union by a simple overlay of the two resulting sets. This principle can also be used of the other associative set-operations, namely intersection and symmetric difference.

CGAL's Boolean set-operations package [FWZH06] uses the principles described above to perform regularized operations on sets of general polygons. A *regularized* operation op* can be obtained by first taking the interior of the resultant point set of an *ordinary* set-operation $(P \text{ op } Q)$ and then by taking the closure of the result; see, e.g., [Hof04]. That is:

$$P \text{ op}^* \, Q = \text{closure}(\text{interior}(P \text{ op } Q)) \ .$$

Regularized Boolean set-operations appear in areas like Constructive Solid Geometry, as regular sets are closed under regularized Boolean set-operations. This is desirable since regularization eliminates lower dimensional features, namely isolated vertices and antennas, thus simplifying and restricting the representation to physically meaningful solids.

The package contains some specializations for straight-edge polygons (implemented using the segment-traits class) and for polygons with circular edges (whose edges are realized by the circle/segment traits-class), but in principle a general polygon can comprise edges of any type of an x-monotone curve defined by one of the arrangement-traits classes.

2.6.2 Envelopes of Planar Curves

Given a set $\mathcal{C}$ of bounded planar curves it is sometimes not necessary to consider the entire planar subdivision they induce, and it is more convenient to study just a sub-structure of their arrangement. Let us assume, without loss of generality, that $\mathcal{C}$ only contains x-monotone curves $\{C_1, C_2, \dots, C_n\}$ — if this is not the case, we can always subdivide the curves in $\mathcal{C}$ to x-monotone subcurves. Any x-monotone curve $C_k \in \mathcal{C}$ can therefore be represented as a univariate function $y = C_k(x)$, defined over some continuous range $R_k \subseteq \mathbb{R}$. The *lower envelope* of $\mathcal{C}$ is defined as the point-wise minimum of all curves — namely, it is given by the following function:

$$\mathcal{L}_{\mathcal{C}}(x) = \min_{1 \le k \le n} \overline{C}_k(x) \ , \tag{2.1}$$

where we define $\overline{C}_k(x) = C_k(x)$ for $x \in R_k$, and $\overline{C}_k(x) = \infty$ otherwise.

Similarly, the *upper envelope* of $\mathcal{C}$ is the point-wise maximum of the x-monotone curves in the set:

$$\mathcal{U}_{\mathcal{C}}(x) = \max_{1 \le k \le n} \underline{C}_k(x) \ , \tag{2.2}$$

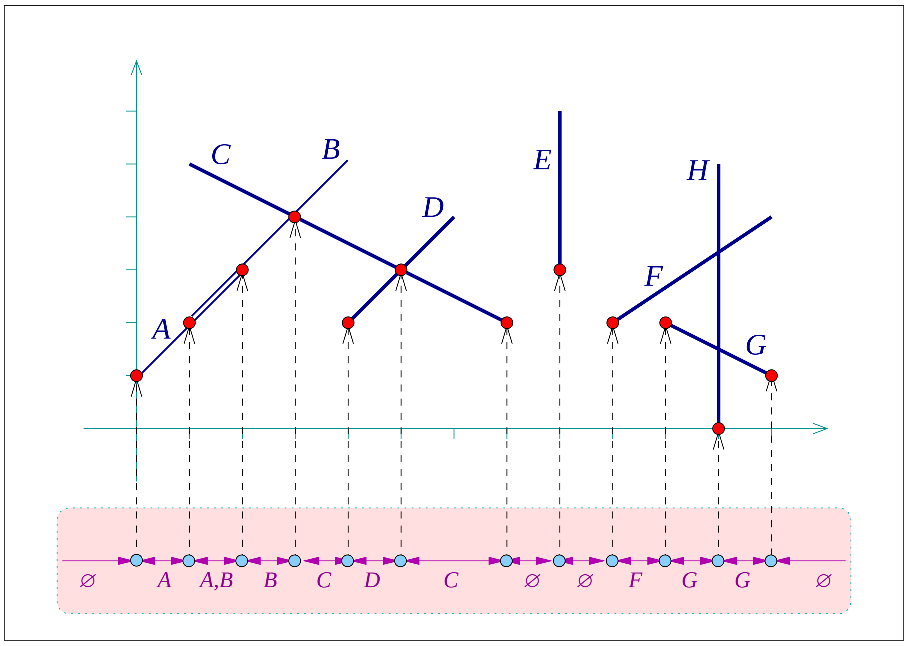


Figure 2.3: The lower envelope of eight line segments, labeled $A, \ldots, H$. The minimization diagram is shown at the bottom, where each diagram vertex points to the point associated with it, and the labels of the segment that induce a diagram edge are displayed below this edge. Note that there exists one edge that represents an overlap (i.e., more than a single curve induces it), and there are also a few edges that represent empty intervals.

where in this case $\underline{C}_k(x) = -\infty$ for $x \notin R_k$.

Given a set of x-monotone curves $\mathcal{C}$, the *minimization diagram* of $\mathcal{C}$ is a subdivision of the x-axis into maximal cells, such that the lower envelope over a specific cell of the subdivision (an edge or a vertex) is attained by the same subset of the curves in $\mathcal{C}$. In non-degenerate situation, an edge — which represents a continuous interval on the x-axis — is induced by a single curve (or by no curves at all, if there are no x-monotone curves defined over the interval), and a vertex is either induced by a single curve and corresponds to one of its endpoints, or by two curves and corresponds to their intersection point. The *maximization diagram* is symmetrically defined for upper envelopes. In the rest of this section, we refer to both these diagrams as *envelope diagrams*.

Let us assume that each pair of curves in $\mathcal{C}$ may intersect at s points at most, where s is a constant. The complexity of their envelope diagram is therefore $O(\lambda_{s+2}(n))$, where $\lambda_\sigma(n)$ denotes the maximal length of a Davenport–Schinzel sequence of n elements with order σ. For small values of σ, this function is almost linear in n. See [SA95] for an analysis of Davenport–Schinzel sequences and the complexity of lower envelopes.

While it is possible to construct the entire arrangement of $\mathcal{C}$ and extract the lower and upper envelope from the resulting DCEL, this process takes $O(n^2 \log n)$, as the arrangement may contain $O(n^2)$ vertices. It is more efficient to compute the envelopes directly in $O(\lambda_{s+2}(n) \log n)$ time, using a divide-and-conquer approach; see [Ata85, HS86, WH04].[14] First, note that the envelope diagram for a single x-monotone curve C_k is trivial to compute: we project the boundary of its range of definition R_k onto the x-axis and label the features it induces accordingly. Given a set that contains more than two curves, we split the set into two disjoint subsets $\mathcal{C}_1$ and $\mathcal{C}_2$, and compute their envelope diagrams recursively. Finally, we merge the diagrams, and we do this in linear time in the complexity of the envelopes by traversing both diagrams in parallel.

The 2D envelope package of CGAL [Wei06a] contains functions for computing the lower and the upper envelopes of a given range of planar curves. The package contains a robust implementation of the divide-and-conquer algorithm described above, but relies on the traits classes included in the arrangement package (see Section 2.3) that provide the primitive geometric operations on the curves it handles. The output of these functions is given in the form of an envelope diagram, which comprises a sequence of interleaved vertices and edges, each associated with a set of curves that induce the envelope over it; see Figure 2.3 for an illustration. Note that each vertex is associated with at least one curve. An edge may, on the other hand, have an empty set of curves, or be associated with multiple curves in case of an overlap.

More details on the construction and representation of envelope diagrams of planar curves can be found in [WH04]. In this context, we also mention the work of Meyerovitch [Mey06], who implemented a generic CGAL package for computing envelopes of surfaces in 3D, based on the infrastructure provided by the arrangement package.

◇◇
◇

In this chapter we reviewed the arrangement package of CGAL in depth and explained how the arrangement primitives can be used in applications developed on top of the package. We also described the algebraic principles used in state-of-the-art certified computation, and explained how it is possible to use exact computation while avoiding the prohibitive running-time penalty such computations may incur.

We concluded with an overview of two related packages that will also be used in applications that we present in the rest of this thesis.

[14]A slightly faster but more involved algorithm, and more difficult to implement, is proposed by Hershberger [Her89]. Its running time is $O(\lambda_{s+1}(n) \log n)$.

Chapter 3

2D Minkowski Sums and Offsets

Given two sets $A, B \in \mathbb{R}^d$, their *Minkowski sum*, denoted by $A \oplus B$, is defined to be their point-wise sum:

$$A \oplus B = \{a + b \mid a \in A, b \in B\} \ .$$

Planar Minkowski sums are used in many applications, such as motion planning and computer-aided design and manufacturing. In this chapter we focus on two important sub-classes of the *planar* Minkowski-sum computation problem: computing the sum of two simple polygons with straight edges, and computing the Minkowski sum of a simple polygon with a disc, an operation widely known as *offsetting* a polygon. We describe a CGAL package that can compute Minkowski sums and offsets in an exact and efficient manner [Wei06b]. This package will be included in the forthcoming release of CGAL (Version 3.3).

3.1 Introduction and Related Work

If P and Q are simple planar polygons having m and n vertices respectively, then $P \oplus Q$ is a subset of the arrangement of $O(mn)$ line segments, where each segment is the Minkowski sum of an edge of P with a vertex of Q, or vice-versa. The size of the sum is therefore bounded by $O(m^2 n^2)$, and this bound is tight [KOS91]. However, if both P and Q are convex, then $P \oplus Q$ is a convex polygon with $m + n$ vertices at most, and can be computed in $O(m + n)$ time (see, e.g., [dBvKOS00, Chapter 13]). If only P is convex, the Minkowski sum of P and Q is bounded by $O(mn)$ [KLPS86], and this bound is tight as well.

3.1.1 Decomposition vs. Convolution

As mentioned above, computing the Minkowski sum of two convex polygons can be performed in linear time in the total number of their edges using a simple procedure that is easily implemented in software. The prevailing method for computing the sum of two non-convex polygons P and Q, is therefore based on *convex decomposition*: we decompose P into convex sub-polygons $P_1, \ldots, P_k$, and Q into convex sub-polygons $Q_1, \ldots, Q_\ell$, obtain the Minkowski sum of each pair of sub-polygons, and compute the union of the $k\ell$ pairwise sub-sums. Namely, we compute $P \oplus Q = \bigcup_{i,j} (P_i \oplus Q_j)$.

Flato [Fla00] (see also [AFH02]) developed an exact and robust implementation of the decomposition method for computing the Minkowski sum of two simple polygons. He implemented about a dozen different polygon-decomposition strategies and programmed several methods for the union computation based on the arrangement package of CGAL Version 2.0 (at the time, CGAL did not include a generic implementation of the sweep-line algorithm, and a package for set-operations, as the one described in Section 2.4.2, did not exist). He also conducted thorough experiments to determine the optimal decomposition and union strategies. Flato's code employs exact rational arithmetic to guarantee robustness and produces exact results even on degenerate inputs. This was the first implementation capable of handling degenerate inputs, and the only one that correctly identifies low-dimensional elements of the Minkowski sum, such as antennas or isolated vertices (see more details in Section 3.2.2).

The LEDA library [MN00] also contains functions for robust Minkowski-sum computation based on convex polygon decomposition that use exact rational arithmetic.[1] However, these functions are limited to performing *regularized* Minkowski-sum computations, which eliminate low-dimensional features of the output (see also Section 2.6.1).

Another approach to computing the Minkowski sum of two polygons is to calculate the *convolution* of the boundaries of P and Q [GRS83, GS87], which comprises a set of closed polygonal curves called the *convolution cycles*. By examining the planar arrangement of the convolution cycles one can deduce the structure of the Minkowski sum of P and Q. The exact definition of polygon convolution and an algorithm for computing it are given in Section 3.2. To the best of our knowledge, our code is the first implementation of software for robust and exact computation of Minkowski sums that is based on the convolution method.

Ramkumar [Ram96] used the convolution approach to devise an efficient algorithm for computing the outer boundary of the Minkowski sum of two polygons. The complexity of this boundary is $O(mn \cdot \alpha(n))$, where $\alpha(\cdot)$ is the functional inverse of Ackermann's function [HPCA+95], hence it is possible to compute it in $o(m^2n^2)$ time. Ramkumar's algorithm traverses each cycle of the convolution, detecting self-intersections, and snipping off the loops created by these self-intersections. The algorithm constructs the outer face of the Minkowski sum in $O((K+(m+n)\sqrt{N_c})\log^2(m+n))$ time, where K is the size of the convolution (which may be $O(mn)$ in the worst case) and N_c is the number of convolution cycles. However, Ramkumar's algorithm uses very complicated data structures, and it is therefore not practical to implement.

In his work, Flato demonstrated that the efficiency of the Minkowski-sum computation is closely related to the number and length of the segments that constitute the boundaries of the pairwise sums of the convex sub-polygons produced in the decomposition step, and given as input to the union-computation procedure. Our experiments show that using the convolution method we construct intermediate geometric entities that are more compact than the ones constructed using the decomposition method. Subsequently, we obtain faster running

[1]For more details and a detailed online documentation, see:
⟨`http://www.algorithmic-solutions.info/leda_guide/geo_algs/minkowski.html`⟩.

times. The experimental results presented in Section 3.3 demonstrate the effectiveness of the convolution method.

3.1.2 Exact vs. Approximate Offsetting

While the Minkowski sum of two polygons with m and n vertices, respectively, can be as combinatorially complex as $\Omega(m^2n^2)$, the complexity of the Minkowski sum of a polygon with n vertices with a disc is always $O(n)$. We refer to the result of such an offset operation as a *dilated polygon*. However, the difficulty in offsetting a polygon in a robust manner is *numerical* and not *combinatorial*. Let us assume that all polygons we handle are *rational* — namely, all the polygon vertices have rational coordinates. The Minkowski sum of two such polygons is also a rational polygon, and we need only exact rational arithmetic to compute it in a precise manner. On the other hand, rational arithmetic is insufficient for offsetting a rational polygon by a rational radius. As we show in Section 3.4, the resulting dilated polygon comprises line segments and circular arcs, and the coordinates of its vertices are typically algebraic numbers of degree up to four. Handling algebraic numbers in a precise manner is a difficult and a highly time-consuming task (see the discussion in Section 2.3.3), so using the algebraic number-types provided by Core or by Leda often yields running times unacceptable for industrial applications.

We present a simple yet powerful approximation algorithm for offsetting a simple polygon, which overcomes the algebraic difficulties by using only exact rational arithmetic. Our algorithm is conservative, namely if $P_r = P \oplus B_r$ is the exact Minkowski sum of a polygon P with a disc B_r of radius r, we compute a generalized polygon $\tilde{P}_r$, bounded by line segments and circular arcs with rational coefficients, such that $P_r \subseteq \tilde{P}_r$. We can also control the approximation error and make it arbitrarily small. The experimental results we bring in Section 3.5 show the considerable speedup our approximation algorithm achieves over the exact algorithm. However, if users must obtain an exact representation of the dilated polygons they compute, our software package also provides the functionality of performing the offset computation in an exact manner.

3.2 The Convolution Method

The Minkowski sum of two sets A and B in $\mathbb{R}^d$ is sometimes referred to as their *convolution*. The reason is that the characteristic function[2] of $A \oplus B$ can be expressed as the convolution of the characteristic functions of A and B, namely:

$$\chi_{A \oplus B}(x) = \int_{\mathbb{R}^d} \chi_A(y)\chi_B(x-y)\,dy \ ,$$

where the integration should be interpreted as a cumulative logical *or* operation. In this chapter we stick to the term *Minkowski sum*, and use the term *convolution* for the operation we apply on the boundary curves of two planar sets, as we describe next.

[2]The *characteristic function* of a set $S \in \mathbb{R}^d$ is defined as $\chi_S(x) = 0$ for $x \notin S$, and $\chi_S(x) = 1$ for $x \in S$.

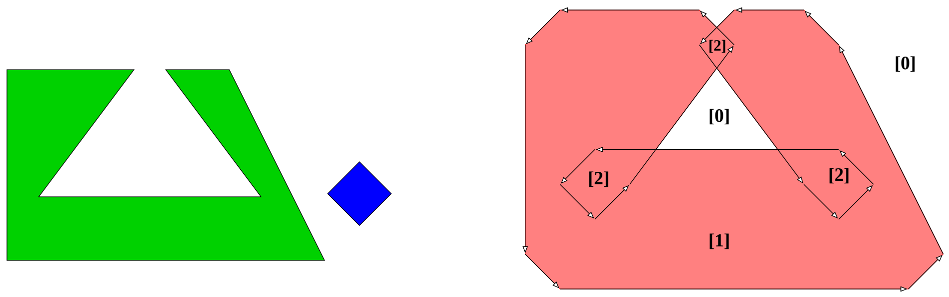


Figure 3.1: Computing the convolution of a convex polygon and a non-convex polygon (left). The convolution consists of a single self-intersecting cycle, drawn as a sequence of arrows (right). The winding number associated with each face of the arrangement induced by the segments forming the cycle appears in brackets. The Minkowski sum of the two polygons is shaded.

Guibas *et al.* [GRS83] prove the connection between the convolution of *planar tracings*, which correspond to the boundary curves of two planar sets, and the Minkowski sum of these sets. Let α and β be the boundary curves of two planar sets A and B, respectively. For now, let us assume that both curves are smooth, so the tangent to every point $x \in \alpha$ (similarly for β) is well-defined; we will denote it by $\vec{x}$. Moreover, let us assume without loss of generality that both boundary curves are counterclockwise oriented, so $\vec{x}$ has the same orientation as the curve at x. The convolution $\alpha * \beta$ is a curve that contains all points of the form $x + y$, where $x \in \alpha, y \in \beta$ and $\vec{x} = \vec{y}$; the orientation of the convolution curve at $x+y$ equals $\vec{x} = \vec{y}$. The Minkowski sum of A and B can be expressed as:

$$A \oplus B = \{x \mid W_{\alpha * \beta}(x) > 0\} \ ,$$

where $W_\gamma(x)$ is the winding number of the curve γ with respect to x. If we view γ as a curve in the complex plane, the *winding number* of $z_0 \in \mathbb{C}$ with respect to this curve is defined by (see, e.g., [Kra99, Section 4.4.4]):

$$W_\gamma(z_0) = \frac{1}{2\pi i} \cdot \oint_\gamma \frac{dz}{z - z_0} \ .$$

Note that the value of $W_\gamma(z_0)$ is integral for any $z_0 \in \mathbb{C}$. Informally, $W_\gamma(x)$ is the number of times γ winds in counterclockwise orientation around the point x, minus the number of times it winds in clockwise orientation around this point (see also [Nee97, Chapter 7] for some examples).

In their work, Guibas *et al.* give special attention to *polygonal tracings*, which are composed of a series of interleaved *moves* (translations in a fixed direction) and *turns* (rotations at a fixed location), and form boundaries of polygons. In the following we use the notation ∂P to designate the boundary curve of the polygon P.

Given two polygons, P with vertices $(p_0, \ldots, p_{m-1})$ and Q with vertices $(q_0, \ldots, q_{n-1})$, we make a move by traversing a polygon edge $\overrightarrow{p_i p_{i+1}}$, and make a turn be rotating on a polygon vertex p_i from the direction of $\overrightarrow{p_{i-1} p_i}$ to the direction of $\overrightarrow{p_i p_{i+1}}$.[3] Without loss of generality,

[3]Throughout this chapter, when we increment or decrement an index of a vertex, we do so modulo the size of the polygon. Indeed, $\overrightarrow{p_{m-1} p_0}$ is also a valid polygon edge.

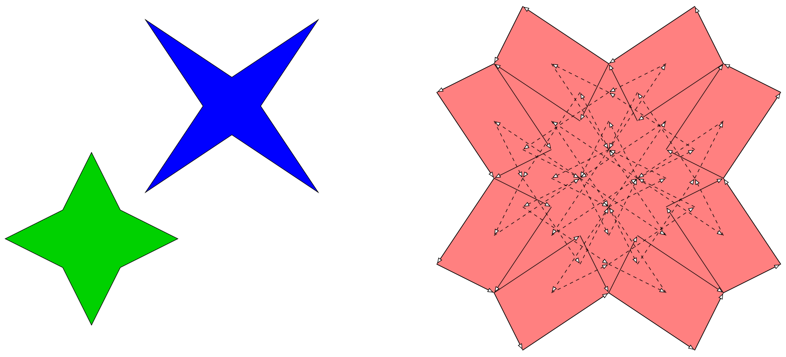


Figure 3.2: Computing the convolution of two non-convex octagons (left). The convolution consists of two cycles (right). The Minkowski sum of the polygons is shaded. One cycle (solid arrows) comprises 32 line segments, while the other consists of 48 line segments, non of which lies on the boundary of the Minkowski sum. The latter cycle is drawn using dashed arrows.

we can assume that both polygons are counterclockwise oriented — that is, if we traverse the polygon boundary in the given order, the interior of the polygon is always to our left. The direction of any point x in the interior of a polygonal edge $p_i p_{i+1}$ is therefore $\overrightarrow{p_i p_{i+1}}$, while the direction of a polygon vertex p_i is defined as the range $[\overrightarrow{p_{i-1} p_i}, \overrightarrow{p_i p_{i+1}}]$. The convolution of the boundaries of these two polygons, namely $\partial P * \partial Q$, is in this case a collection of line segments of the form $\overrightarrow{(p_i + q_j)(p_{i+1} + q_j)}$, where the vector $\overrightarrow{p_i p_{i+1}}$ lies between $\overrightarrow{q_{j-1} q_j}$ and $\overrightarrow{q_j q_{j+1}}$,[4] and — symmetrically — of segments of the form $\overrightarrow{(p_i + q_j)(p_i + q_{j+1})}$, where the vector $\overrightarrow{q_j q_{j+1}}$ lies between $\overrightarrow{p_{i-1} p_i}$ and $\overrightarrow{p_i p_{i+1}}$. We can label the convolution segment as $\langle (i, i+1), j \rangle$ in the former case or $\langle i, (j, j+1) \rangle$ in the latter case. From the definition, it is clear that $\partial P * \partial Q$ contains at most $O(mn)$ line segments.

The segments of the convolution form a number of closed (not necessarily simple) polygonal curves called *convolution cycles*. The Minkowski sum $P \oplus Q$ is the set of points having a positive winding number with respect to these cycles. Figure 3.1 illustrates the winding number of the various regions in the plane induced by a single convolution cycle.

In case both input polygons P and Q are convex, their convolution is a convex polygonal tracing. If only one polygon (say P) is convex, then $\partial P * \partial Q$ still contains a single cycle; see [Ram96] for a proof. This cycle may not be simple, as illustrated in Figure 3.1. If both P and Q are non-convex, their convolution may comprise several cycles, and in order to compute the Minkowski sum of the polygons one has to consider the set of points having a non-zero winding number with respect to any of these cycles; see Figure 3.2 for an illustration.

Given two simple polygons P and Q having m and n vertices respectively, we compute their Minkowski sum in three steps: first we compute the cycles of the convolution $\partial P * \partial Q$, then we construct the planar arrangement induced by the segments that form the convolution cycles, and finally we extract the Minkowski sum from this arrangement. We next explain each of these steps in detail.

[4]We say that a vector $\vec{v}$ lies between two vectors $\vec{u}$ and $\vec{w}$, if we reach $\vec{v}$ strictly before reaching $\vec{w}$ when we move all three vectors to the origin and rotate $\vec{u}$ counterclockwise. Note that this also covers the degenerate case where $\vec{u}$ has the same direction as $\vec{v}$.

Table 3.1: A pseudo-code listing of the procedure ComputeConvolutionCycle.

```
ComputeConvolutionCycle ($P, i_0;\ Q, j_0$)
let $\mathcal{C} \longleftarrow \emptyset$.
let $i \longleftarrow i_0$. let $j \longleftarrow j_0$.
let $s \longleftarrow (p_i + q_j)$.
do:
    let $inc\_P \longleftarrow$ IsBetweenCounterclockwise ($\overrightarrow{p_i, p_{i+1}};\ \overrightarrow{q_{j-1}q_j}, \overrightarrow{q_j q_{j+1}}$).
    let $inc\_Q \longleftarrow$ IsBetweenCounterclockwise ($\overrightarrow{q_j, q_{j+1}};\ \overrightarrow{p_{i-1}p_i}, \overrightarrow{p_i p_{i+1}}$).
    if $inc\_P$ = True, then:
        let $t \longleftarrow (p_{i+1} + q_j)$.
        Push the segment $\vec{st}$ into $\mathcal{C}$, and mark the label $\langle (i, i+1), j\rangle$ as used.
        let $s \longleftarrow t$.
        let $i \longleftarrow i + 1$ (modulo the size of $P$).
    if $inc\_Q$ = True, then:
        let $t \longleftarrow (p_i + q_{j+1})$.
        Push the segment $\vec{st}$ into $\mathcal{C}$, and mark the label $\langle i, (j, j+1)\rangle$ as used.
        let $s \longleftarrow t$.
        let $j \longleftarrow j + 1$ (modulo the size of $Q$).
while $i \neq i_0$ or $j \neq j_0$.
return $\mathcal{C}$.
```

3.2.1 Computing the Convolution Cycles

Guibas and Seidel [GS87] show how to compute the convolution cycles of two polygons in optimal $O(m + n + K)$ time, where $K = |\partial P * \partial Q|$. However, their method cannot handle degenerate inputs, as it makes some general-position assumptions on the polygons (e.g., an edge of P cannot have the same direction as an edge of Q). In this section we present a simple and robust algorithm, whose asymptotic running time is $O(m+n+\min\{m_r n, n_r m\}+K)$, where m_r and n_r are the number of reflex vertices in P and Q, respectively. As our experiments show, the running time of the construction of the convolution cycles is in practice negligible with respect to the overall Minkowski-sum computation.

The procedure ComputeConvolutionCycle, listed in pseudo-code in Table 3.1, describes a simple algorithm that constructs a single convolution cycle $\mathcal{C}$ of two polygons $P = (p_0, \ldots, p_{m-1})$ and $Q = (q_0, \ldots, q_{n-1})$. It starts from two given vertices p_{i_0} and q_{j_0}, such that $p_{i_0} + q_{j_0}$ is a vertex on the cycle $\mathcal{C}$, and proceeds iteratively, adding at least one convolution segment in each iteration (in a degenerate situation, when $\overrightarrow{p_i, p_{i+1}}$ and $\overrightarrow{q_j, q_{j+1}}$ have the same direction, two segments are added to the cycle in a single iteration; these two segment have the same supporting line and share a common endpoint), until the cycle closes. Note that we also maintain a set of *used labels* representing all convolution segments that have already been computed. This set is represented using a hash table, such that each access to the set takes $O(1)$ time on average (see, e.g. [CLRS01, Chapter 12]).

We next describe how to locate the pairs of indices i_0 and j_0 needed for the procedure above. We start by locating the bottommost vertex of P (the minimal vertex with respect to a yx-lexicographical order) and the bottommost vertex of Q. Assume, without loss of generality, that these are the vertices p_0 and q_0. These vertices are not reflex, and it is clear that either $\overrightarrow{p_0p_1}$ lies between the edges incident to q_0, or vice-versa. We can therefore compute a convolution cycle starting from this pair of vertices.

If either of the polygons is convex (that is, $m_r = 0$ or $n_r = 0$), then the convolution consists of a single cycle, and we are done with the convolution step. This is due the fact that multiple convolution cycles can only be induced by pair of reflex vertices in P and in Q, as stated by Ramkumar [Ram96]. Otherwise, we traverse the reflex vertices of Q (we assume, without loss of generality that $n_r m < m_r n$), and for each such vertex q_{j_0} we go over all vertices of P and try to locate a vertex p_{i_0}, such that $\overrightarrow{p_{i_0}p_{i_0+1}}$ lies between $\overrightarrow{q_{j_0-1}q_{j_0}}$ and $\overrightarrow{q_{j_0}q_{j_0+1}}$. When we locate such a vertex pair and such that the label $\langle(i_0, i_0+1), j_0\rangle$ has not been used (we query the *used labels* set to check that), we call ComputeConvolutionCycle to compute an additional convolution cycle.

3.2.2 Computing the Winding Numbers

Flato [Fla00, Appendix A.1] describes a simple algorithm for computing the union of a set of simple polygons with a counterclockwise orientation. In the context of his work, these polygons correspond to the pairwise Minkowski sums of convex sub-polygons. The first step is to construct the arrangement of the directed segments that correspond to the polygon edges, referred to as the *boundary segments*. In a non-degenerate scenario, each arrangement edge is associated with a portion of a single polygon edge. As the arrangement edge is represented by a pair of twin halfedges (see Section 2.2), one halfedge has the same direction as the associated segment and the other has an opposite direction. In an arrangement of simple polygons, no antennas can exist, so each of the those two halfedges is incident to a different arrangement face.

The arrangement of the boundary segments has the property that all points in a single arrangement face are covered by the same subset of polygons. As all points inside a polygon have a winding number 1 with respect to this polygon boundary and all points outside it have a winding number 0 with respect to the polygon, we say that all points in an arrangement face have the same winding number with respect to the input polygons. It is thus possible to associate a winding number $W(f)$ with each arrangement face f.[5] The unbounded face of the arrangement obviously has a winding number 0, and the winding numbers of all other (bounded) faces are computed using a breadth-first traversal of the arrangement faces. Suppose that the faces f_1 and f_2 are separated by a pair of twin halfedges e_1 and e_2, where f_i is the incident face of e_i. In case the segment associated with the edge has the same direction as e_1, we set $W(f_2) \longleftarrow W(f_1) - 1$, otherwise we set $W(f_2) \longleftarrow W(f_1) + 1$. We continue in this manner until all arrangement faces have been visited.

This simple scheme does not work in case of overlaps between the boundary segments. We mention that such overlaps not only occur in degenerate scenarios, but they are inherent to pairwise Minkowski sums obtained by the decomposition method. We therefore need to

[5]As we mentioned in Section 2.2, the Dcel structure can be easily extended so auxiliary data fields may be stored with the various Dcel records.

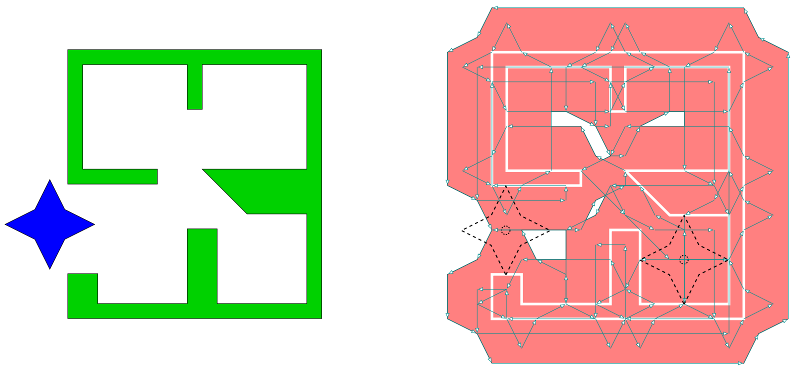


Figure 3.3: A house plan and a star-shaped furniture (left). The Minkowski sum of the two polygons (right) consists of low-dimensional features. For clarity, two copies of the star are drawn using a dashed line with their center positioned on these features. The left copy is located on an *antenna* on the Minkowski-sum boundary, such that the star can move along this antenna while touching the walls of the house without penetrating into the walls. The right copy is located such that the star center is on an *isolated vertex*, which designates a location where the star does not penetrate into the walls but it is immobilized.

slightly augment the winding-number algorithm. The idea is to associate a number $B(e)$ with each halfedge e, counting the number of overlapping boundary segments incident and equal in direction to this half-edge. We can automatically compute this number if we use the traits-class decorator (see Section 2.3.4) to extend each segment with two counters C_{right} and C_{left}. As each sub-segment in the arrangement may be induced by several overlapping boundary segments, C_{right} counts the number of boundary segments directed from left to right, while C_{left} is the number of overlapping boundary segments directed to the left. The traits-class decorator automatically maintains the counter values, so given a halfedge e we consider the counters stored with its associated sub-segment, and simply set $B(e) \longleftarrow C_{\text{right}}$ if the halfedge is directed from left to right, and $B(e) \longleftarrow C_{\text{left}}$ otherwise.

Using this generalized scheme, we can still traverse the arrangement faces in a breadth-first order, yet this time we set $W(f_2) \longleftarrow W(f_1) - B(e_1) + B(e_2)$. Having visited all faces, we can output the boundary of the single hole in the unbounded face as the outer boundary of the union, and add the outer boundary of each bounded arrangement face $\hat{f}$ with $W(\hat{f}) = 0$ as a hole in the union.

When employing the convolution method we need to compute the winding numbers with respect to counterclockwise-oriented convolution cycles. Luckily, the same arguments used for proving the correctness of the polygon-union algorithm also hold for the case of convolution cycles. We therefore have to compute the arrangement of all directed segments constituting the convolution cycles and compute $W(f)$ of each face in a breadth-first manner. Consider for example Figure 3.1, where we correctly identify the hole in the Minkowski sum, as it is represented by a face whose winding number is 0. We mention that in this case overlapping segments only occur in case of degenerate inputs (namely, two polygon edges that have the same direction), which is one of the advantages the convolution method has over the decomposition method.

So far we have described an algorithm that computes the regularized Minkowski sum $P \hat{\oplus} Q$, namely the closure of the interior of $P \oplus Q$, as all 1-dimensional or 0-dimensional features of the sum (*antennas* and *isolated vertices*, respectively) are discarded. In this case, the output is given as a polygon that represents the outer boundary of the sum and an additional, possibly empty, set of polygons that represents the holes inside this polygon. This representation is sufficient for many applications, but for other applications, such as motion planning and especially assembly planning, low-dimensional features play an important role as they represent *tight passages*.

Consider the example depicted in Figure 3.3, where we wish to move a star-shaped furniture in a house, allowing translations only. The interior of the Minkowski sum in this case consists of all forbidden placements for the star center, and each point on the boundary of the sum corresponds to a *semi-free* placement of the star, where it touches the walls but does not penetrate them. It is possible to tightly move the furniture from the outside to the bottom-left room through the doorway, while its center is moved on an antenna on the sum boundary. It is also possible to exactly fit the furniture, in its current orientation, inside the bottom-right room, but without any possible way to move it in any direction. In this case, its center is positioned on an isolated vertex of the Minkowski-sum boundary.

It is easy to extend the algorithm for computing winding numbers to locate the antennas. Having computed the winding numbers of all faces, we go over all arrangement edges. Let the current edge be realized by the twin halfedges e_1 and e_2, whose incident faces are f_1 and f_2, respectively. Then the edge forms an antenna if and only if $W(f_1), W(f_2) \neq 0$ and $W(f_1) - B(e_1) = W(f_2) - B(e_2) = 0$. Locating the isolated vertices is slightly more complicated, and involves computing an additional attribute for each halfedge; the reader is referred to [Fla00] for the full details including a proof of correctness. We just mention that locating the low-dimensional features incurs no asymptotic run-time penalty.

Beside its ability to compute a polygon with holes that represents a regularized Minkowski sum, our software is also capable of outputting a planar arrangement that captures all features of the Minkowski sum of two input polygons. Such an arrangement consists of properly marked faces that compose the interior of the sum, and marked edges and vertices that represent the sum boundary. Recall that the arrangement package supports isolated vertices, so it is possible to report 0-dimensional features in the output as well.

3.3 Experimental Results for Polygonal Minkowski Sums

We have adapted parts of Flato's software [Fla00] to comply with the interface of CGAL Version 3.2. As the original software consists of several thousands of lines of code, we did not convert all polygon-decomposition strategies; instead, we use the three convex-decomposition algorithms bundled with the Polygon Partitioning package of CGAL [Her06]. In addition, we added the small-side angle-bisector decomposition strategy (see below). In his experiments, Flato reports this strategy as the one that yields the fastest running times for the overall Minkowski-sum computation process (see also [AFH02]). We ran our experiments with all four strategies listed below.

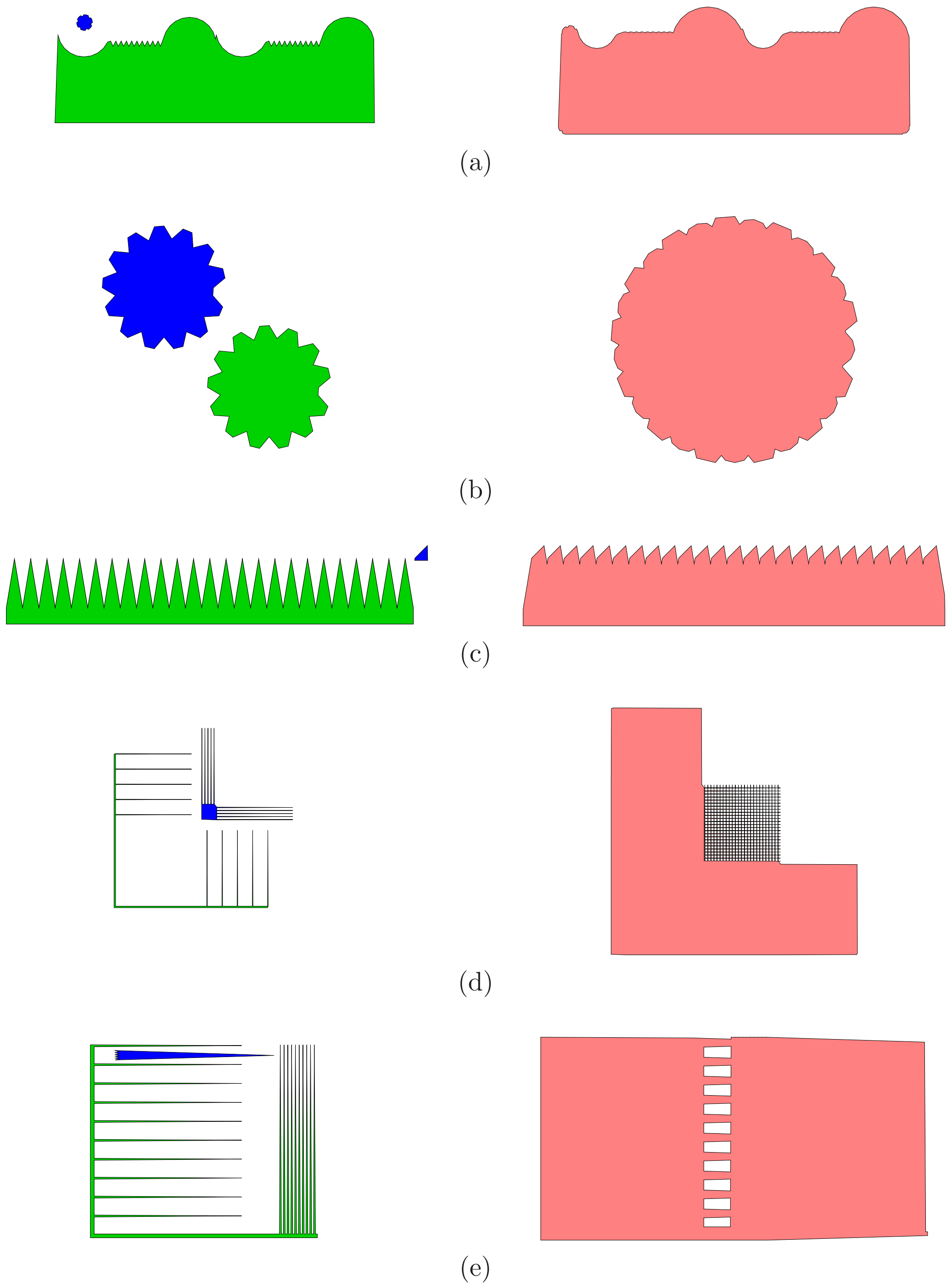


Figure 3.4: Samples of input polygons (left) and their Minkowski sums (right): (a) *chain*; (b) *wheels*; (c) *comb*; (d) *fork*; (e) *spiked*.

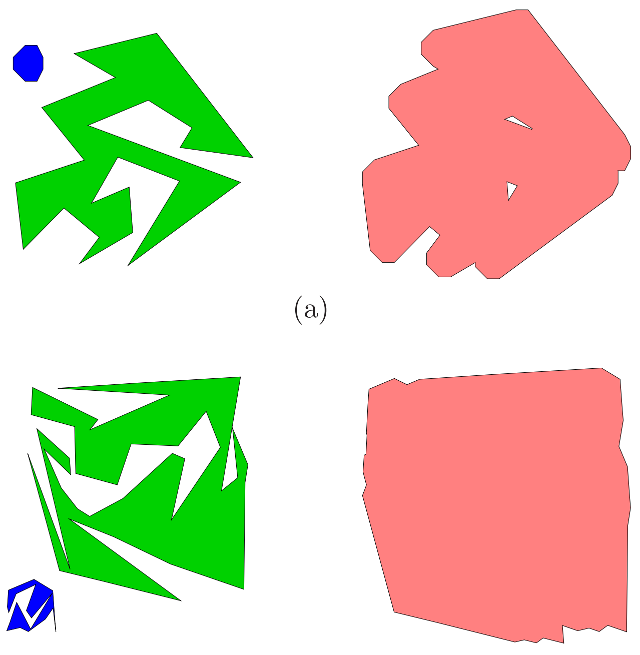


Figure 3.5: Samples of input polygons (left) and their Minkowski sums (right): (a) *cavity*; (b) *random*.

Optimal (Opt.): The dynamic-programming algorithm of Greene [Gre83] for computing an optimal decomposition of a polygon into a minimal number of convex sub-polygons. The main drawback of this strategy is that it runs in $O(n^4)$ time and requires $O(n^3)$ space in the worst case, where n is the number of vertices in the input polygon.[6]

Hertel–Mehlhorn (HM): The approximation algorithm suggested by Hertel and Mehlhorn [HM83], which triangulates the input polygon and proceeds by throwing away unnecessary triangulation edges. This algorithm requires $O(n)$ time and space and guarantees that the number of sub-polygons it generates is not more than four times the optimum. The CGAL implementation runs in $O(n \log n)$ time.

Greene (Gre.): Greene's approximation algorithm [Gre83] which computes a convex decomposition of the polygon based on its partitioning into y-monotone polygons. This algorithm runs in $O(n \log n)$ time and $O(n)$ space, and has the same approximation guarantee as Hertel and Mehlhorn's algorithm.

Small-side angle-bisector (SSAB): A heuristic improvement to the angle-bisector decomposition method suggested by Chazelle and Dobkin [CD85]. It starts by examining each pair of reflex vertices in the input polygon such that the entire interior of the diagonal connecting these vertices is contained in the polygon. Out of all available pairs, it selects p_i and p_j, such that the number of reflex vertices along the polygon boundary from p_i to p_j (or from p_j to p_i) is minimal. The polygon is split by the diagonal $p_i p_j$, and the process continues recursively on both resulting sub-polygons.

[6]A more efficient algorithm for the optimal convex decomposition was given by Keil and Snoeyink [KS02], but is not implemented in CGAL.

In case it is not possible to eliminate two reflex vertices at once any more, each reflex vertex is eliminated by an angle bisector emanating from it. The entire process takes $O(n^2)$ time.

In the original implementation, the intersections between the angle bisectors and the polygon edges induce *Steiner points* in the decomposed sub-polygons, namely intersection points between original polygon edges and angle bisectors. We have slightly modified the algorithm, such that instead of eliminating a reflex vertex p_i using an angle bisector, we look for another vertex p_{j^*}, such that $p_i p_{j^*}$ is contained in the polygon, and such that the ratio between the two angles $\measuredangle(p_{i-1}, p_i, p_{j^*})$ and $\measuredangle(p_{j^*}, p_i, p_{i+1})$ that $p_i p_{j^*}$ induces is as close to 1 as possible among all candidates. Our experiments show that this modified approach yields very good decompositions, while avoiding the introduction of Steiner points, which may lead to more complex computations.

The original Minkowski-sum software was based on the arrangement package of Cgal Version 2.0, which supported only the *incremental* construction of arrangements, inserting curves one at a time. In the current Cgal version, it is possible to construct arrangements *aggregately*, so all boundary segments are inserted together using a sweep-line algorithm. Constructing an arrangement aggregately is asymptotically more efficient for κ line segments that sparsely intersect (namely the total number of intersection points is $o\left(\frac{\kappa^2}{\log \kappa}\right)$), and this is almost always the case in Minkowski-sum computations (see Tables 3.2 and 3.3, where the combinatorial complexity of the arrangement depends on the number of segment intersections). This argument is also backed up by experiments [WFZH07] that show that constructing a Cgal arrangement of line segments in an aggregated manner is usually 5–10 times faster than constructing it incrementally. We have therefore implemented an aggregated version of the union algorithm, as described in Section 3.2.2, and did not try the incremental union algorithm, as described in [Fla00, Chapter 3].

We have conducted a large number of tests with various input sets. Here we report on eight representative input sets containing polygon pairs, most of them taken from [AFH02] (see Figures 3.4 and 3.5 for illustrations):

Chain: A chain-shaped polygon with 82 vertices (37 of them are reflex), and a star-shaped polygon with 30 vertices (6 reflex).

Wheels: Two star-shaped polygons, each containing 40 vertices (14 reflex in each).

Comb: A comb-shaped polygon containing 53 vertices (24 reflex) and a convex polygon with 22 vertices. This input set introduces the worst-case complexity for the sum of a convex and a non-convex polygon.

Fork: The well-known example for a Minkowski sum of size $O(m^2 n^2)$ [KOS91]. The large "fork" consists of 34 vertices (19 reflex) and the smaller one has 31 vertices (18 reflex).

Cavity: A random-looking polygon with 22 vertices (10 reflex) and a small convex octagon, chosen to fit some of the cavities in the larger polygon.

Random: Two random-looking polygons, with 40 and 20 vertices (19 and 8 reflex vertices, respectively).

Spiked: A large polygon with 64 vertices (40 reflex) and a small polygon with 12 vertices (5 reflex), that can fit into the cavities on the larger polygon.

Country: The map of Israel, represented as a polygon with 50 vertices (24 reflex), and a smaller polygon with 30 vertices (8 reflex). (These polygons are not shown in the figures.)

Table 3.2 summarizes the performance of the Minkowski-sum computations for the selected input sets, using the polygon-decomposition method. We give the running times, as obtained on a Pentium IV 3 GHz machine with 2 Gb of Ram, and averaged over 100 executions for each decomposition strategy. We also indicate — for the strategy that achieved the fastest running time on each input set — the numbers of sub-polygons k and ℓ in the decompositions of the two input polygons, the total number S of segments in all $k\ell$ Minkowski sums, the time required to perform the decomposition and compute these S segments, and the complexity of the arrangement (number of vertices, edges and faces) induced by the boundary segments.

It should be mentioned that the running times stated in Table 3.2 are sometimes about a hundred times faster than the ones reported in [AFH02]. This can be partly attributed to the fact that we used a faster machine in our experiments,[7] and mostly due to the improvements in the new version of Cgal's arrangement package. The new package handles intersections of line segments more efficiently [FWH04] due to numerous improvements, such as the reduction of the number of geometric operations and predicates the arrangement-construction algorithms invoke [WFZH07]. Moreover, we use the predefined Cgal kernel [FP06], which uses interval arithmetic to filter exact computations with rational numbers, as provided by Version 4.1 of Gnu's Multi-Precision library (Gmp), and helps reducing the running times even further (see also Section 2.3.2).

Table 3.3 summarizes the performance of the Minkowski-sum computations for the selected input sets using the convolution method. We indicate the numbers of convolution cycles N_c, the total number K of convolution segments, the time it took to compute the convolution, and the complexity of the induced arrangement. We also applied the Minkowski-sum function provided by Leda (Version 4.4) on our input sets, using the exact rational kernel of Leda, which — similarly to Cgal's predefined kernel — also employs arithmetic filtering to speed up the computations with an exact rational number-type. The running times of Leda are also given in the table; unfortunately, here we do not have access to the code and we are therefore unable to provide more detailed construction statistics.

In almost all cases, the convolution method yields faster running times than the best decomposition scheme (recall that Leda also employs the polygon-decomposition method). This is due to the fact that the convolution method usually induces a smaller arrangement as its intermediate structure. As the total running-time is clearly dominated by the arrangement-construction time, the convolution method may achieve a speed-up of up to a

[7]Flato reports using a 500 MHz Pentium III machine.

Table 3.2: Running times (measured in milliseconds) for the Minkowski-sum computation using various decomposition strategies. The running time of the fastest strategy in each case is shown in bold, and its construction statistics are also given.

Input set	Total running time				The fastest method						
								Decomp.	Arrangement size		
	Opt.	HM	Gre.	SSAB	k	ℓ	S	time	$\|V\|$	$\|E\|$	$\|F\|$
chain	**390**	424	578	444	35	7	2520	123	7239	13627	6390
wheels	488	744	867	**477**	17	17	2446	13	12955	25624	12671
comb	32	58	**17**	37	26	1	650	4	671	769	100
fork	287	524	1220	**260**	12	11	1048	6	6827	13377	6552
cavity	8	7	**6**	8	14	1	160	1	167	244	79
random	**234**	349	376	320	20	10	1540	15	8209	16310	8103
spiked	249	370	1630	**232**	22	6	1108	9	7721	15150	7431
country	338	798	410	**188**	16	8	1344	8	5126	9772	4648

Table 3.3: Running times (measured in milliseconds) and construction statistics for the Minkowski-sum computation using the convolution method.

Input set	N_c	K	Conv. time	Arrangement size			Total running time		
				$\|V\|$	$\|E\|$	$\|F\|$	Conv.	Best decomp.	Using LEDA
chain	1	1452	7	2077	2868	793	**69**	390	463
wheels	2	1200	7	3225	5482	2259	**92**	477	332
comb	1	603	7	627	651	26	**17**	**17**	97
fork	1	1266	5	11203	22063	10862	521	**260**	266
cavity	1	110	1	135	161	28	**4**	6	21
random	1	580	3	2589	4698	2111	**62**	234	229
spiked	1	876	5	5075	9749	4676	184	232	**175**
country	2	1050	5	1940	3064	1126	**61**	188	275

factor of 5 in some cases. Moreover, the memory requirements for storing the intermediate arrangement are also considerably smaller.

The convolution method has another important advantage. As we learn from Table 3.2, the choice of a decomposition strategy may have drastic effects on the running time of the Minkowski-sum computation. However, the best strategy that yields the fastest running time for a specific input can be found only by experimenting. When using the convolution method, no such tuning experiments are needed.

The *fork* input set is the only example that exhibits slower running times when using the convolution method. This is due to the fact that the input polygons are decomposed very nicely by the SSAB decomposition-strategy. Each of the fork prongs is separated into a single thin rectangle, such that the Minkowski sums of the sub-polygons are nearly pairwise disjoint. On the other hand, as we have many pairs of parallel edges in the input polygons (note that all edges of the two "forks" are axis-parallel), the convolution cycle in this case contains many redundant loops that incur the greater run-time consumption.

We note that it is possible to use a hybrid approach: given two polygons, use the small-side angle-bisector decomposition scheme and compute the set of S line segments in all pairwise Minkowski sums; in addition, compute the K line segments that form the convolution cycles of the input polygons. If $K < S$ (which is almost always the case), compute the arrangement of the convolution cycles and extract the Minkowski sum from this arrangement. Otherwise, go on computing the union of all pairwise sums of convex sub-polygons. Since the time needed for computing the SSAB decomposition and obtaining the pairwise sums is just a few milliseconds (see Table 3.2), this hybrid approach can successfully handle degenerate input polygons such as the *fork* input set, while incurring only a negligible computational overhead in comparison to the straightforward convolution method.

3.4 Exact and Approximate Offset Polygons

Having described our efficient implementations of Minkowski-sum algorithms for two polygons, we now turn to the closely related problem of offsetting a polygon, namely computing the boundary of the Minkowski sum of the polygon with a disc of a given radius. Offsetting is a fundamental task in CAD/CAM. The main body of CAD literature on this subject concentrates on computing offsets of curves and surfaces (see, e.g., [KF95, LKE98, Mae99] and the references therein). In the general case, the offset curves of rational planar curves are not rational, so it is possible to compute them only using approximate techniques. As we focus here on the special case of offsetting a polygon, we are able to provide exact construction of the resulting dilated polygon, or alternatively an approximate construction with guaranteed quality.

At first glance offsetting may seem an easier task compared to computing the Minkowski sum of two polygons. However, when we aim for a robust implementation, offsetting efficiently is much more demanding. Let us assume that we are given a polygon P with n vertices $(p_0, \ldots, p_{n-1})$ that are ordered counterclockwise around P's interior. All vertices have rational coordinates. We wish to compute the dilated polygon P_r, namely the Minkowski sum of P with a disc of radius r, where r is rational. We show that even this relatively simple task involves exact computation with algebraic numbers, if we wish our computations to be exact.

If P is a convex polygon, the offset is easily computed by shifting each polygon edge by r away from the polygon, namely to the right-hand side of the edge. As a result we obtain a collection of n disconnected *dilated edges*. Each pair of adjacent dilated edges, induced by $p_{i-1}p_i$ and p_ip_{i+1} (recall that incrementing or decrementing an index is always done modulo n), are connected by a circular arc of radius r, whose supporting circle is centered at p_i. The angle that defines such a circular arc equals $\pi - \measuredangle(p_{i-1}, p_i, p_{i+1})$; see Figure 3.6(a) for an illustration. Naturally, the running time of this simple process is linear in the size of the polygon.

If P is not convex, its offset can be obtained by decomposing it into convex sub-polygons $P_1, \ldots, P_m$ such that $\bigcup_{i=1}^{m} P_i = P$, computing the offset of each sub-polygon and finally computing the union of these dilated sub-polygons (see Figure 3.6(b)). However, we already

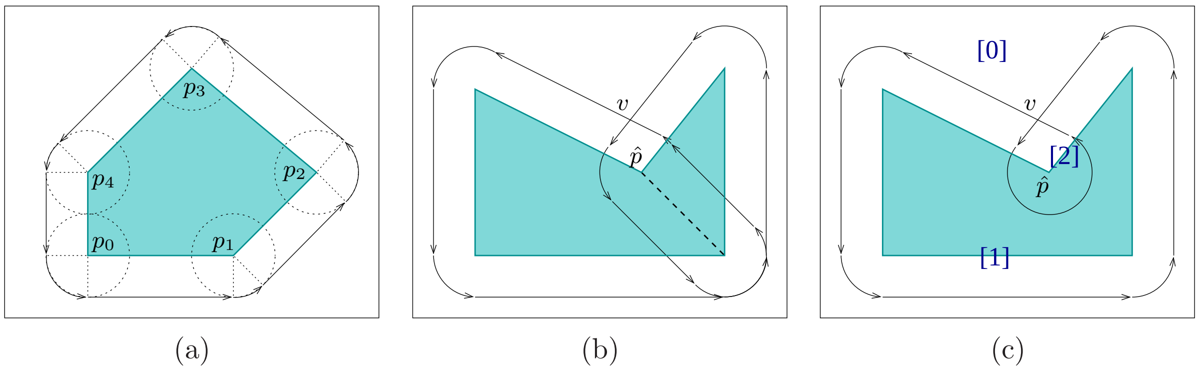


Figure 3.6: (a) Offsetting a convex polygon. (b) Computing the offset of a non-convex polygon by decomposing it into convex sub-polygons; $\hat{p}$ is a reflex vertex. (c) Offsetting a non-convex polygon by computing its convolution with a disc. The convolution cycle induces an arrangement with three faces, whose winding numbers are shown in brackets.

know that it is more efficient to compute the convolution cycle of the polygon with a disc, which in our case is a smooth curve comprising line segments and circular arcs. The sub-segments of the convolution cycle can be constructed by applying the process described in the previous paragraph. The only difference is that a circular arc induced by a reflex vertex p_i is defined by an angle $3\pi - \measuredangle(p_{i-1}, p_i, p_{i+1})$; see Figure 3.6(c) for an illustration. Once we obtain the convolution cycle (in this case there is only one convolution cycle since the disc is convex), we construct the arrangement of the line segments and circular arcs that constitute this cycle, compute the winding numbers of the arrangement faces, and output the union of the faces having a positive winding number.

3.4.1 The Offset Convolution Cycle

Let us now take a closer look at the algebraic characterization of the convolution cycle constructed in the course of the offset computation. If the offset radius r is rational, the circular arcs of the convolution cycle, representing dilated polygon vertices, are clearly supported by rational circles, as they are centered at the vertices of the input polygon, which have rational coordinates.

Let us examine how the dilated edges look like. We consider the polygon edge p_1p_2 to be directed from $p_1 = (x_1, y_1)$ to $p_2 = (x_2, y_2)$. We denote by θ the angle it forms with the x axis. Let $\ell = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ be the edge length, so we have $\cos\theta = \frac{1}{\ell}(x_2 - x_1)$ and $\sin\theta = \frac{1}{\ell}(y_2 - y_1)$. As we traverse the polygon edges in a counterclockwise orientation, we construct the dilated edge v_1v_2 that corresponds to p_1p_2 by shifting either polygon vertex by a vector whose length is r and which forms an angle of $\phi = \theta - \frac{\pi}{2}$ with the x-axis. It is easy to see that:

$$\sin\phi = \sin\theta \cdot \cos\frac{\pi}{2} - \cos\theta \cdot \sin\frac{\pi}{2} = -\cos\theta = \frac{1}{\ell}(x_1 - x_2)\ , \tag{3.1}$$

$$\cos\phi = \cos\theta \cdot \cos\frac{\pi}{2} + \sin\theta \cdot \sin\frac{\pi}{2} = \sin\theta = \frac{1}{\ell}(y_2 - y_1)\ . \tag{3.2}$$

Thus, the endpoints of the dilated edge are given by ($j = 1, 2$):

$$v_j = \left(x_j + \frac{r}{\ell}(y_2 - y_1),\; y_j + \frac{r}{\ell}(x_1 - x_2) \right) \; . \tag{3.3}$$

Indeed, the coordinates of these points are solutions of quadratic equations with rational coefficients (*one-root numbers*, or algebraic numbers of degree two), but the segment v_1v_2 is supported by a line with *irrational* coefficients: it is easy to show that if the supporting line of p_1p_2 is $ax+by+c = 0$ (where $a, b, c \in \mathbb{Q}$), then the line supporting v_1v_2 is $ax+by+(c+\ell r) = 0$, where ℓ is usually an irrational number. The intersection points between two segments representing dilated edges (see for example the point v in Figure 3.6(b) and (c)), or between a dilated edge and a circular arc that represents a dilated vertex, are algebraic numbers of degree four, namely roots of polynomials with integer coefficients of degree 4.

A simpler representation of the dilated edges is based on the fact that the locus of all points lying at distance r from the line $ax + by + c = 0$ is given by:

$$\frac{(ax + by + c)^2}{a^2 + b^2} = r^2 \; ,$$

which is a degenerate conic curve (a pair of parallel lines) with rational coefficients. The line segments and the circular arcs that constitute the convolution cycle can be therefore represented as arcs of conic arcs with rational coefficients, and it is possible to compute their arrangement and obtain the offset polygon using the conic-traits class of the arrangement package (see Section 2.3.3). However, the computational overhead incurred by the exact computation with algebraic numbers makes this process relatively slow (see more in Section 3.5). If we could work with segments of rational lines, it would be possible to construct the intermediate arrangement using the circle/segment traits-class (see Section 2.3.3), which employs only exact rational arithmetic, and achieve faster running times. We next describe a conservative and tight approximation scheme that enables us to accelerate the computation in this fashion.

3.4.2 The Approximation Scheme

We next describe our approximation algorithm that avoids using expensive computations with algebraic numbers. First, we note that in case of a horizontal edge (where $y_1 = y_2$) or a vertical edge (where $x_1 = x_2$), the edge length ℓ is a rational number. In these cases we can trivially construct the dilated edge v_1v_2 in an exact manner using rational arithmetic, so in the following we assume that $x_1 \neq x_2$ and $y_1 \neq y_2$.

We approximate the dilated edge by two line segments with rational coefficients, as shown in Figure 3.7: in a manner we describe next, we find two points v'_1 and v'_2 with *rational* coefficients, such that v'_j lies on the circle $(x - x_j)^2 + (y - y_j)^2 = r^2$ (for $j = 1, 2$). Moreover, v'_1 and v'_2 are selected such that the angle ϕ'_1 that $\overrightarrow{p_1v'_1}$ forms with the x-axis is slightly smaller than ϕ, and the angle ϕ'_2 that $\overrightarrow{p_2v'_2}$ forms with the x-axis is slightly larger than ϕ. We let $\Delta\phi_1 = \phi - \phi'_1$ and $\Delta\phi_2 = \phi'_2 - \phi$. Observe that the lines tangent to the two circles at v'_1 and v'_2 have rational coefficients and their intersection point w' has rational

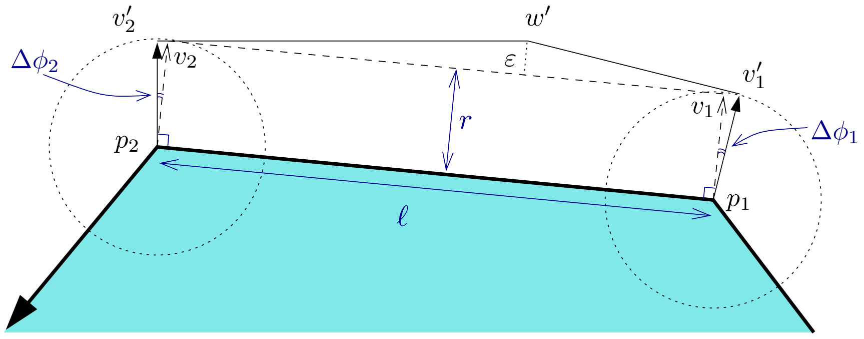


Figure 3.7: Approximating the dilated edge v_1v_2, which is induced by the polygon edge p_1p_2, by the polyline $v'_1w'v'_2$. The approximation error is defined as the maximal distance of this polyline from the original dilated edge.

coordinates. We use the two line segments $v_1'w'$ and $w'v_2'$ to approximate the dilated edge v_1v_2.

We now explain how to compute the rational points v_1' and v_2' with the properties mentioned above. Note that if $\tau = \tan\frac{\phi}{2}$ were a rational number, then $\sin\phi = \frac{2\tau}{1+\tau^2}$ and $\cos\phi = \frac{1-\tau^2}{1+\tau^2}$ would be rational as well, and v_1 and v_2 would both have rational coordinates. We therefore aim for a rational approximation of τ. Using the half-angle formulae[8] we can write:

$$\tau \;=\; \tan\frac{\phi}{2} = \frac{1-\cos\phi}{\sin\phi} = \frac{1-\frac{1}{\ell}(y_2-y_1)}{\frac{1}{\ell}(x_1-x_2)} = \frac{\ell+(y_1-y_2)}{x_1-x_2}\;, \tag{3.4}$$

$$\tau \;=\; \tan\frac{\phi}{2} = \frac{\sin\phi}{1+\cos\phi} = \frac{\frac{1}{\ell}(x_1-x_2)}{1+\frac{1}{\ell}(y_2-y_1)} = \frac{x_1-x_2}{\ell+(y_2-y_1)}\;. \tag{3.5}$$

As $\ell > |y_2 - y_1|$ (recall that the edge is not vertical), the sign of τ is determined by $\mathrm{sign}(x_1 - x_2)$. If $x_1 > x_2$ (as is the case in the example depicted in Figure 3.7), we have $\frac{\pi}{2} < \theta < \frac{3\pi}{2}$, hence $0 < \phi < \pi$ and $\tau > 0$. Let $\underline{\ell} \in \mathbb{Q}$ be a rational approximation of ℓ from below (that is, $0 < \ell - \underline{\ell} < \eta$ for some small $\eta > 0$). We now define the angles ϕ_1' and ϕ_2', based on Equations (3.4) and (3.5), respectively:

$$\tau_1' \;=\; \tan\frac{\phi_1'}{2} = \frac{\underline{\ell}+(y_1-y_2)}{x_1-x_2} < \frac{\ell+(y_1-y_2)}{x_1-x_2} = \tau\;, \tag{3.6}$$

$$\tau_2' \;=\; \tan\frac{\phi_2'}{2} = \frac{x_1-x_2}{\underline{\ell}+(y_2-y_1)} > \frac{x_1-x_2}{\ell+(y_2-y_1)} = \tau\;. \tag{3.7}$$

It is clear that $\phi_1' < \phi < \phi_2'$. The two tangency points of the approximating segments are therefore given by (j = 1,2):

$$v_j' = \left(x_j + \frac{1-\tau_j'^2}{1+\tau_j'^2}\cdot r\;,\;\; y_j + \frac{2\tau_j'}{1+\tau_j'^2}\cdot r \right)\;.$$

[8]See, e.g., ⟨`http://mathworld.wolfram.com/Half-AngleFormulas.html`⟩.

In case $x_1 < x_2$ we have $\tau < 0$. In this case we compute a rational approximation of ℓ from *above*, denoted $\overline{\ell}$ (thus $0 < \overline{\ell} - \ell < \eta$ for some small $\eta > 0$), and define $\tau_1' < \tau$ and $\tau_2' > \tau$ in an analogous manner to the definitions in Equations (3.6) and (3.7).

Obtaining a rational approximation for ℓ is easy. Recall that $\ell^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$ is a rational number; for any rational $l_0 > 0$, the recursively defined series $l_{i+1} = \frac{1}{2}\left(l_i + \frac{\ell^2}{l_i}\right)$ converges to ℓ.[9] If we need to approximate ℓ from below, we simply look for the minimal index k such that $0 < \ell^2 - l_k^2 < \delta$, or such that $0 < \ell^2 - \left(\frac{\ell^2}{l_k}\right)^2 < \delta$; we take $\underline{\ell} \longleftarrow l_k$ in the former case and $\underline{\ell} \longleftarrow \frac{\ell^2}{l_k}$ in the latter case. Computing an approximation from above is symmetric. Observe that if we fix a rational $\delta > 0$ value, all calculations are carried out using rational arithmetic.

We next show how tight should the approximation of the edge length ℓ be, in order to guarantee that the polyline $v_1'w'v_2'$ we use for approximating the dilated edge does not lie too far from the exact offset v_1v_2.

3.4.3 The Approximation Quality

Theorem 3.1 *For any polygon edge connecting $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, where $x_1 \neq x_2$ and $y_1 \neq y_2$, and any given $\varepsilon > 0$, let $\hat{\ell}$ be a rational approximation of the edge length $\ell = \|p_2 - p_1\|$ such that $|\ell^2 - \hat{\ell}^2| < \ell \left|\frac{\ell + (y_1 - y_2)}{2(x_1 - x_2)}\right| \cdot \varepsilon$. If we compute a polyline approximation $v_1'w'v_2'$ of the dilated edge as described above, then the distance of the point w' from the supporting line of the true dilated edge v_1v_2 is upper bounded by ε.*

Proof: Let us assume that $x_1 > x_2$ and that $\hat{\ell}$ is an approximation of the edge length from below, so that we have $\ell^2 - \hat{\ell}^2 < \delta$ for some $\delta > 0$ (the proof for $x_1 < x_2$ is symmetric). Note that:

$$\ell^2 - \hat{\ell}^2 = (\ell + \hat{\ell})(\ell - \hat{\ell}) > 2\hat{\ell}(\ell - \hat{\ell}) ,$$

so $\ell - \hat{\ell} < \frac{\delta}{2\hat{\ell}}$. We now use the fact that $\tan(\alpha - \beta) = \frac{\tan\alpha - \tan\beta}{1 + \tan\alpha\tan\beta}$ and using Equations (3.4) and (3.6) we obtain:

$$\begin{aligned}
\tan\left(\frac{\phi - \phi_1'}{2}\right) &= \frac{\tau - \tau_1'}{1 + \tau\tau_1'} = \\
&= \frac{\frac{\ell + (y_1 - y_2)}{x_1 - x_2} - \frac{\hat{\ell} + (y_1 - y_2)}{x_1 - x_2}}{1 + \frac{\ell + (y_1 - y_2)}{x_1 - x_2} \cdot \frac{\hat{\ell} + (y_1 - y_2)}{x_1 - x_2}} = \frac{(x_1 - x_2)(\ell - \hat{\ell})}{(x_1 - x_2)^2 + \ell\hat{\ell} + (\ell + \hat{\ell})(y_1 - y_2) + (y_1 - y_2)^2} \\
&< \frac{(x_1 - x_2)\frac{\delta}{2\hat{\ell}}}{\ell^2 + \ell\hat{\ell} + (\ell + \hat{\ell})(y_1 - y_2)} < \frac{x_1 - x_2}{4\hat{\ell}^2(\hat{\ell} + (y_1 - y_2))} \cdot \delta \ .
\end{aligned}$$

The approximated angle ϕ_1' is always very close to ϕ, namely $\Delta\phi_1 = \phi - \phi_1'$ is small, and we can safely bound $\tan(\Delta\phi_1)$ by $4 \cdot \tan\left(\frac{\phi - \phi_1'}{2}\right)$. Note that $\Delta\phi_1 = \measuredangle(v_1, p_1, v_1')$ is equal to

[9] This method is named the *Babylonian method*, and is known to converge quadratically to the square-root; see, e.g., ⟨`http://en.wikipedia/wiki/Methods_of_computing_square_roots`⟩.

the angle between the supporting lines of v_1v_2 and $v_1'w'$ (see Figure 3.7 for an illustration), so as $1 < \frac{\ell}{\hat{\ell}} \ll 2$, the distance of w' from v_1v_2 is upper bounded by:

$$\ell \tan(\Delta\phi_1) < \frac{\ell \cdot (x_1 - x_2)}{\hat{\ell}^2(\hat{\ell} + (y_1 - y_2))} \cdot \delta < \frac{2(x_1 - x_2)}{\hat{\ell}(\hat{\ell} + (y_1 - y_2))} \cdot \delta \quad .$$

We conclude that if $\delta < \ell \left| \frac{\ell + (y_1 - y_2)}{2(x_1 - x_2)} \right| \cdot \varepsilon$, then this distance is smaller than ε. $\square$

An important property of our approximation algorithm is that it is *conservative*. That is, given a polygon P and an offset radius r it always computes a super-set $\tilde{P}_r$ of the exact dilated polygon P_r. This property is crucial for many applications. For example, if we use our algorithm to approximate the forbidden configuration space of a round tool-tip moving amidst polygonal obstacles, we will never have “false positives” — namely, we will never declare a location of the tool center as collision-free when in fact it collides with an obstacle. We can have “false negatives”, namely regarding a collision-free location as forbidden, but these errors are usually not crucial to the successful performance of the algorithm. Moreover, the probability of having “false negatives” can be made arbitrarily small by selecting a small enough approximation error ε, as proved above in Theorem 3.1.

In other applications, where one wishes the approximate dilated polygon $\tilde{P}_r$ to be *contained* in the exact dilated polygon P_r, we proceed as follows. Given a rational $\varepsilon > 0$, we let $\underline{r} = r - \varepsilon$ and apply the approximation algorithm with the offset radius $\underline{r}$. As the polylines that approximate the dilated edges can lie at most $\underline{r} + \varepsilon = r$ away from the original polygon edges, the result is guaranteed to be a subset of the exact dilated polygon.

3.5 Experimental Results for Offsetting Polygons

As explained in the previous section, using the various traits classes available in the arrangement package of Cgal we can compute the offset of a polygon in an exact manner by representing its boundary using conic arcs, or provide a conservative approximation using rational line segments and circular arcs. Here we compare the performance of the exact construction using the conic-traits class with our approximate construction scheme based on the circle/segment traits-class. Table 3.4 summarizes the running times of the two approaches; the various input polygons and their offset boundaries are shown in Figure 3.8 (the *country* polygon is a map of Israel).

The offset radius chosen for each polygon is typically two orders of magnitude smaller than the size of the bounding box of the polygon. For the approximation scheme we selected the error bound accordingly. Table 3.4 includes the running time for error bounds of 10^{-7} and 10^{-10} times the offset radius. Our approximation scheme yields a significant speedup in the offset computations over the exact computation, especially for polygons that contain many small-size features (e.g., *chain*), spikes (e.g., *spiked*), or cavities (e.g., *random*). In such cases, the intermediate arrangement induced by the convolution cycle contains relatively many intersections; as computing and manipulating the intersection points of two arcs in

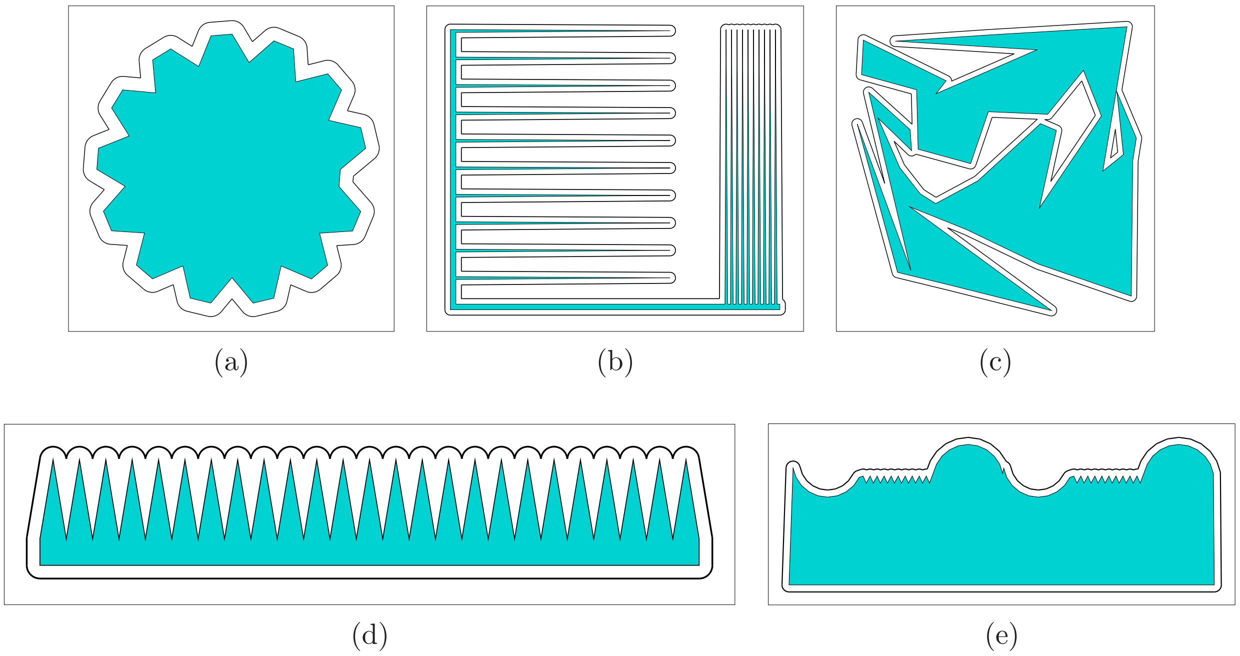
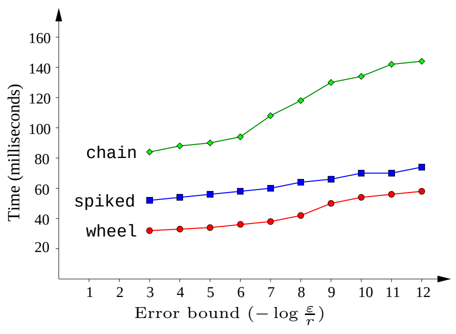


Figure 3.8: Selected polygons used in the polygon-offset benchmarks: (a) *wheel*, (b) *spiked*, (c) *random*, (d) *comb*, (e) *chain*. The boundary of each dilated polygon is drawn in a thick black line.

the circle/segment traits-class is more efficient than in the conic-traits class, we can gain considerable speed-ups in these cases.



Note that as we decrease the error bound ε, we have to use rational numbers with longer bit-lengths (namely the sizes of the numerators and denominators increase), which incurs some running-time penalty. The graph to the right shows the running time of the approximate offset-computation process for three selected polygons as a function of the error bound (for $\varepsilon = 10^{-k}r$, where $k = 3, 4, \ldots, 12$). It shows that the running time is linear, or moderately super-linear, in $\log \frac{\varepsilon}{r}$. This is due to the fact that the number of bits used to represent the coordinates of the approximated dilated edges are proportional to the logarithm of the relative error $\frac{\varepsilon}{r}$. The times needed to add or to subtract two rational numbers is proportional to their bit-lengths, while the time needed to multiply them is super-linear in their bit-lengths.

As we have already mentioned, the Minkowski-sum package integrates well with other CGAL packages. In particular, it is possible to perform Boolean operations on dilated polygons using the Boolean set-operations package [FWZH06] (see also Section 2.6.1). The last set of experiments demonstrates the application of the union operation on a set of dilated polygons, which has many important applications in many fields, such as computer-aided design and robotic motion planning.

Table 3.4: The running times (measured in milliseconds) of exact and approximate offset computations. The *Size* column lists the number of polygon vertices and the number of reflex vertices (in parentheses).

Input Polygon	Size	Bounding Box	Offset Radius (r)	Running times: Exact Offset	Running times: Approx. Offset ($\varepsilon = 10^{-7}r$)	Running times: Approx. Offset ($\varepsilon = 10^{-10}r$)
wheel	40 (14)	$10^8 \times 10^8$	$5 \cdot 10^6$	88	35	54
spiked	64 (40)	600×510	5	1378	60	71
random	40 (19)	775×788	15	95	56	68
comb	53 (24)	1250×200	25	138	45	50
chain	82 (37)	$1.2 \cdot 10^8 \times 4 \cdot 10^7$	$2 \cdot 10^6$	1210	109	134
country	50 (24)	$1.7 \cdot 10^6 \times 4 \cdot 10^6$	10^5	451	66	82

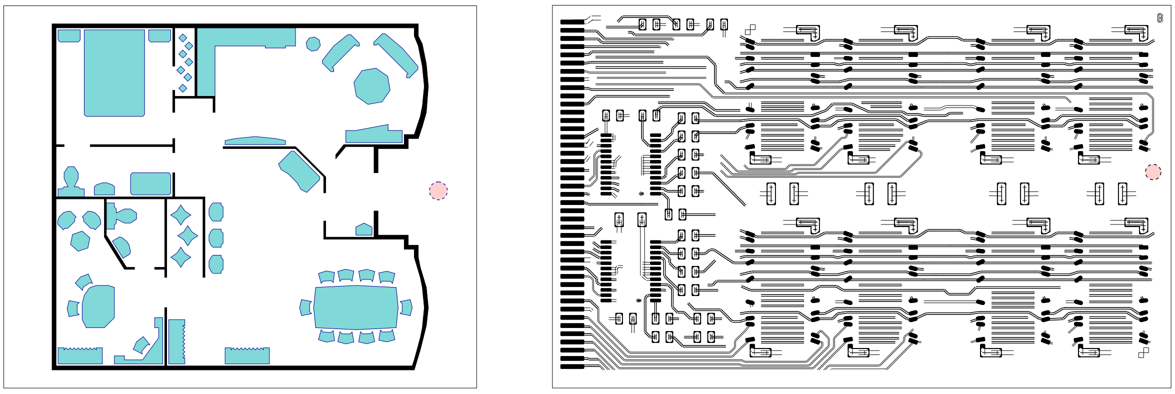

(a) (b)

Figure 3.9: Inputs of polygon sets: (a) *house*, (b) *VLSI*. The dashed disc that appear in each figure illustrate the offset radius we use in each case.

Given a set of straight-edge polygons we compute the Minkowski sum of each polygon with a disc of radius r, either in an exact manner *or* using the approximation algorithm, and finally compute the union of all dilated polygons using the multi-way union procedure provided by the set-operations package. In the former case we have to use the conic-traits class to carry out the union computation in an exact manner, while in the latter case it is possible to use the circle/segment traits-class, as we operate on segments of rational lines and arcs of rational circles. Note that the results of the exact and the approximate computations are *not* equal in the geometric sense. Yet, we choose an approximation error-bound such that the two results are topologically equivalent (namely they contain the same number of polygons and the same number of holes in each polygon).

The *house* data set (Figure 3.9(a)) consists of 55 polygons bounded by the box $[0, 5000] \times [0, 4000]$. We use an offset radius $r = 200$. It takes 0.376 seconds to compute the union of the approximate dilated polygons (with an error bound of $\varepsilon = 10^{-6}r$), while the exact construction takes 3.176 seconds. The *VLSI* data set (Figure 3.9(b)) is much larger and contains 22,400 polygons and straight line segments bounded in $[0, 80] \times [0, 50]$; here we use an offset radius of $r = 1$. The approximate computation (with $\varepsilon = 10^{-6}r$) takes 51.81 seconds, while constructing the union of the exact dilated polygons takes 30.87 minutes.

◇◇◇

In this chapter we introduced the new CGAL package for computing planar Minkowski sums, with robust implementations that can handle all inputs, including highly degenerate ones, yielding topologically correct results. The two main contributions of our package are:

- An efficient implementation for computing Minkowski sums of two simple polygons using the convolution method. To the best of our knowledge, this is the first software implementation of a robust algorithm based on the polygon-convolution method. As our experiments show, the convolution method is superior to the polygon-decomposition method on almost all input sets, and improves the running times by a factor of 2–5.
- An algorithm that yields a conservative (and tight) approximation of the Minkowski sum of a polygon with rational vertices with a disc with a rational radius. This approximation scheme allows the robust handling of dilated polygons in an efficient manner, using only exact rational arithmetic. It significantly reduces the processing time compare to handling exact offset polygons, which our software can also compute.

Chapter 4

Continuous Path Verification in Multi-Axis NC-Machining

In a multi-axis NC-machining collision-avoidance problem, we are given a rotating milling-cutter, also called a *tool*, whose profile — with respect to its axis of symmetry — is typically piecewise linear or circular, moving in space among polyhedral solids bounded by triangular facets. These triangles model the workpiece sculptured by the tool as well as other static parts of the NC-machine. Our goal is to verify that the motion path of the tool between two given configurations[1] is collision-free, so it can move near the workpiece without damaging it (or any of the other static parts of the machine). See Held's book [Hel91] on the geometric foundations of NC-machining for practical algorithms for tool-path generation.

4.1 Introduction: 5-Axis Machining

In recent years, evermore complex surfaces have been evolving in various engineering processes and designs, thus creating the demand for more efficient and accurate machining of such surfaces. 5-axis machining, where the tool can be translated and rotated, offers many advantages over traditional 3-axis machining (where only translations are allowed), such as faster machining times, better tool accessibility and improved surface finish. Yet, there are still difficult geometric problems to solve in order to fully benefit from 5-axis machining.

Most of the research in the context of collision detection in multi-axis NC-machining has focused on the discrete tool-workpiece interference checks. That is, the verification is performed only at several discrete points along the tool-path, without any guarantee that the motion of the tool between these points is indeed collision-free.

In this chapter, we consider the problem of continuous collision detection between two given configurations of the tool. The cutter, the chucks and other rotating parts are all considered in the verification process. Although the tool configuration has five degrees of freedom (because of the axial symmetry of the tool, it is sufficient to specify its position with its tilt and yaw angles), we reduce the dimensionality of the problem in two steps: First,

[1]In the NC-machining literature, a configuration is often referred to as a *contact location (CL) point*.

we radially project the model around the symmetry axis of the tool to a three-dimensional space, obtaining a set of surface patches. We then examine the projected surfaces and reduce the dimensionality even further, computing the planar silhouette curves of these surface patches and constructing their lower envelope. Finally, we compare this lower envelope against the profile of the tool and check whether the two entities intersect. The general guideline followed in this thesis, namely using exact computational techniques in order to obtain improved results, is also applied here. Our approach yields accurate results for purely translational motion, and provides guaranteed (and good) approximation bounds when the motion includes rotation.

4.1.1 Related work

Research in the area of 5-axis machining has focused mostly on generating proper cutter tool-paths, optimizing tool orientation (for maximal material removal rates, for example), and solving local interference problems (gouging); see, e.g., [Elb95, Ver94]. While some algorithms were developed to avoid global collisions between the tool and the machined part in 5-axis machining (see, e.g., [LDK03, LC95]), these methods do not allow for a general form of a tool and assume a cylindrical approximation for it. The earlier method [LC95] utilizes convex hulls in order to quickly find the feasible set of tool orientations at a given location, and in the case of a collision, a correction vector is calculated in the direction of the surface normal-vector at the interference point. Lauwers *et al.* [LDK03] integrate the collision detection into the tool-path generation stage. Once collision is detected, the collision vector is computed and is later used to calculate the correction vector.

Ho *et al.* [HSA01] and Balasubramaniam *et al.* [BSM03] allow a more general representation of tool geometry for the purpose of collision detection in 5-axis machining. They use a point-cloud representation for the workpiece, along with an efficient bounding volumes hierarchy, thereby reducing the interference problem to a series of simple point-inclusion queries. However, this representation tends to lose efficiency as the number of sampled points is increased in order to obtain a good approximation for the machined part. Furthermore, interference between the tool and other static parts of the NC-machine is not handled by these methods. Bohez *et al.* [BMK$^+$03] use planar slices of the workpiece and of the tool, sampled at constant intervals, and perform intersection tests between these slices. While this method allows for general tool geometry and enables detecting collisions with the fixed parts of the NC-machine, its precision is limited, as it depends on the distance between slices and on the behavior of the surfaces involved.

All previous methods mentioned above are able to perform only *static* collision checks — that is, to detect tool-model interference when the tool has a fixed location. The continuous tool-path is therefore verified at a finite number of *discrete* configurations, assuming a dense enough sampling of intermediate configurations. The obvious drawback of these discrete approaches is that they do not guarantee that the motion between two consecutive configurations is indeed collision-free.

Continuous tool-path verification method was proposed by Jerard *et al.*, [JHDS89] where the surface is approximated by a set of points with direction vector associated with each point. Tool-path envelope is constructed for a translational movement of a flat-end or a ball-end cutter, and interference with the surface is verified by intersecting the associated direction vectors with the envelope. This method, however, does not support arbitrary tool geometry and cannot detect interference with the tool holder. Moreover, as the surface is represented by a discrete number of direction vectors, interference in between the direction vectors could be missed.

The problem of interference detection was intensively studied in the fields of robotics and computer animation, where we usually have to check whether a robot (i.e., a moving body), which is typically modeled as a polyhedron in $\mathbb{R}^3$, collides with an obstacle, which can be static or dynamic. A common approach is to construct a hierarchy of simple bounding volumes of the robot and of the obstacles, which helps filtering away unnecessary collision checks and focusing on the regions where potential interference may occur; see [LM04] for a recent survey of collision-detection techniques. Continuous collision-detection queries are usually answered by reducing the problem to a set of static collision-detection queries along the motion path (see, e.g., [Can86]). For example, probabilistic motion planners (Prms — see, e.g., [KŠLO96]) typically sample a random set of configurations (sometimes referred to as *milestones*) and try to connect pairs of configurations that lie close to one another, according to some distance metric. Most planners connect two milestones by sampling a discrete sequence of intermediate configurations along the straight line connecting the milestones and checking whether the robot collides with the obstacles when placed at each intermediate configuration. The robot is "inflated" in such a way that guarantees that if there is a collision along the line we do not miss it, but it is possible to have "false alarms" due to this inflation. As a result, a path computed using a Prm is usually guaranteed to be collision-free. However, the Prm is not guaranteed to find a collision-free path, even if one exists.

Significant progress has been achieved during the last few years in developing efficient software for detecting collisions between complex models. Kim *et al.* [KVLM03] solve the continuous collision-detection problem between a general polyhedral body, having six degrees of motion freedom, and a set of polyhedral obstacles by approximating the swept volume of the moving polyhedron along the given trajectory. Redon *et al.* [RKLM04] use a similar approach for continuous collision detection between articulated models. The work by Kim and Rossignac [KR03] approximates the relative motion between two objects by a sequence of screw motion segments, thereby reducing the collision check to the numeric extraction of the roots of simple univariate analytic functions. We note that the general collision-detection problem in $\mathbb{R}^3$ is very difficult to solve in an exact manner. In the special case of a rotating cutter of an NC-machine it is possible, however, to use axial symmetry and obtain exact results, as we show in this chapter.

4.1.2 Chapter Overview

The work presented in this chapter extends the framework described in [IEH+04], which introduces an efficient and precise algorithm for discrete collision detection in the context of 5-axis machining. The algorithm combines the usage of efficient data structures, which help one focus on the relevant parts of the model for any given tool position and orientation, with the exact computation of the lower envelope of a set of planar curves that describe the

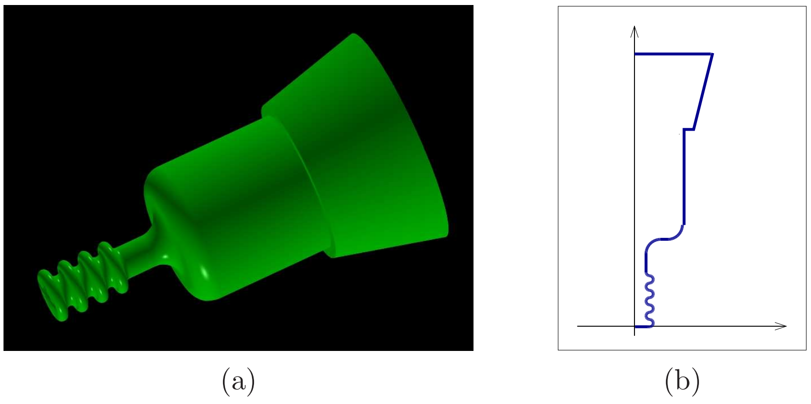


Figure 4.1: A complex milling-cutter (a) and its profile (b).

distance between the relevant model triangles and the tool's symmetry axis. The discrete case of detecting a tool-model collision at a given position is thus solved efficiently and robustly, even for complex tool geometry. In Section 4.2, we give an overview of the exact solution used for the discrete case and develop notation that will be used throughout the chapter.

Section 4.3 presents the extension of the approach used for static collision checks to the detection of collisions along a continuous motion path of the tool. The geometry of the curves we have to handle here is more complicated than in the discrete case, but we show how to reduce the problem such that the curves we obtain can be practically handled in an efficient manner. We elaborate on the software implementation in Section 4.4 and report on experimental results in Section 4.5.

4.2 The Discrete Case

Due to the radial symmetry of the milling-cutter, we can represent it using a planar curve, which we call the tool's *profile*, such that the boundary of the tool is the surface of revolution created when rotating the profile around the tool's axis of symmetry. In many cases the tool looks like a narrow cylinder with a ball-end, but it can also have a more complex shape, as the milling-cutter shown in Figure 4.1. In our work, we assume that the tool's profile is given as an arbitrary polyline (namely, a polygonal curve), which is weakly monotone with respect to the symmetry axis.

In the discrete case we sample several configurations along the tool-path and verify that each intermediate configuration is indeed collision-free. Namely, given a description of the profile of a rotating milling cutter along with its position and its orientation, we wish to determine whether the tool collides with the machined workpiece, or with other static parts of the NC-machine, all modeled using triangular surfaces.

Given a tool position and orientation, we use a line-distance query (Ldq) data structure [IEH$^+$04] that efficiently identifies all triangles in the model the tool can potentially intersect with (the *relevant* triangles) and filters out triangles in distant parts of the model.

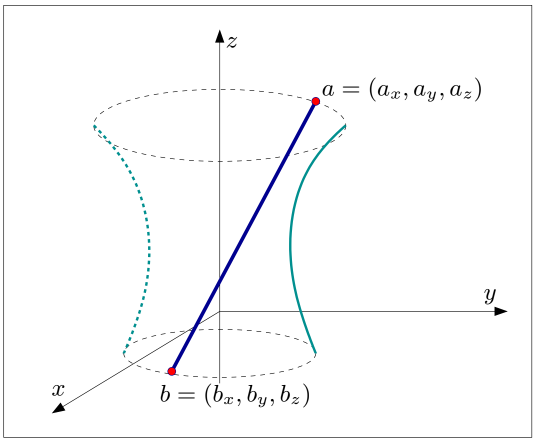


Figure 4.2: The radial projection of the line segment ab around the z-axis onto the yz-plane. We consider only the half-plane $y > 0$, thus the left hyperbolic arc (dotted) is ignored.

Toward this end, the tool is bounded by a cylinder and its central axis is used as the query line in the LDQ data structure, which returns all the triangles that are in the vicinity of the cylinder. If the radius of the bounding cylinder is r, this can be done by offsetting each triangle by r (that is, computing the Minkowski sum of the triangle with a ball of radius r — the resulting shape is referred to in [AS00b] as *krepl*) and then use ray-tracing techniques in order to identify the dilated triangles hit by a ray emanating from the given position and coinciding with the symmetry axis. This operation is clearly equivalent to detecting all triangles that intersects with the cylinder. The LDQ data structure stores a hierarchy of uniform grids with decreasing cell size (thus with increasing granularity) that enables efficient ray tracing. This hierarchy can also be dynamically updated in an efficient manner as the workpiece is sculptured and its shape is modified. More details on the implementation of the LDQ structure can be found in [IEH$^+$04].

Having considerably reduced the number of triangular facets we have to consider, it is now possible to perform a collision check in $\mathbb{R}^3$ between the tool and the set of triangles we obtained from the LDQ structure. However, we can take advantage of the symmetry of the rotating tool and project the problem onto the plane. We first apply a rigid transformation on the entire scene that brings the tool tip to be positioned at the origin, with the z-axis being its axis of symmetry (see Figure 4.1(b)). We proceed by radially projecting the relevant model triangles around the z-axis onto the yz-plane. This radial projection is the *trace* that the triangle etches on the yz-plane (more precisely, on the half-plane $y > 0$) when rotated around the z-axis.

Consider the line segment ab, where $a = (a_x, a_y, a_z)$ and $b = (b_x, b_y, b_z)$, rotated around the z-axis. The trace of ab in the yz-plane is given by the explicit quadratic equation, derived by looking at the distance between a point on the segment and the z-axis (see Figure 4.2 for

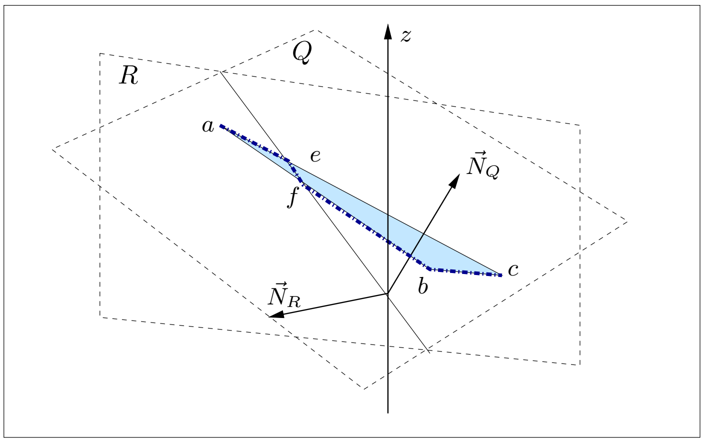


Figure 4.3: The plane R, whose normal is defined as $\vec{N}_R = \vec{N}_Q \times \vec{z}$, where $\vec{N}_Q$ is normal to the triangle $\triangle abc$, intersects the interior of the triangle. The lower envelope of the trace of $\triangle abc$ in this case is formed by the radial projection of the line segments ae, ef, fb and bc around the z-axis.

an illustration):

$$\begin{aligned} d^2(z) &= \left(\frac{1}{b_z - a_z}\big((b_z - z)a_x + (z - a_z)b_x\big)\right)^2 \\ &+ \left(\frac{1}{b_z - a_z}\big((b_z - z)a_y + (z - a_z)b_y\big)\right)^2 \quad , \end{aligned} \tag{4.1}$$

where z changes continuously between a_z and b_z.

For our convenience, we shall transform the original coordinate system (x, y, z) to the $\hat{x}\hat{y}$-plane, where $\hat{x} \longleftarrow z$ and $\hat{y} \longleftarrow \sqrt{x^2 + y^2}$, thus the above expression becomes a canonical hyperbola, with $\hat{x} = 0$ being its major axis. The representation of the curve as a canonical hyperbola is advantageous for exact computation; see details in Section 4.4. If we let:

$$\begin{aligned} D_x &= a_x - b_x \ , & D_y &= a_y - b_y \ , & D_z &= a_z - b_z \ , \\ E_x &= a_x b_z - a_z b_x \ , & E_y &= a_y b_z - a_z b_y \ , \end{aligned} \tag{4.2}$$

the equation of this hyperbola becomes:

$$\hat{y}^2 = \frac{D_x^2 + D_y^2}{D_z^2}\hat{x}^2 - 2\frac{D_x E_x + D_y E_y}{D_z^2}\hat{x} + \frac{E_x^2 + E_y^2}{D_z^2} \ . \tag{4.3}$$

Note that the entire hyperbola is the radial projection of the line containing ab, when rotated around the z-axis. We are only interested in the hyperbolic arc that lies in the half-plane $\hat{y} > 0$ and whose two endpoints are given by $(a_z, \hat{y}(a_z))$ and $(b_z, \hat{y}(b_z))$.

Given a triangle $\triangle abc$, we should radially project it around the z-axis and examine its trace on the $\hat{x}\hat{y}$-plane. We observe that it is sufficient to consider the lower envelope of this trace, induced by the points of $\triangle abc$ that are closest to the z-axis at each value of z: If the milling-cutter intersects $\triangle abc$ at some point (x_0, y_0, z_0) , then the triangle point closest to

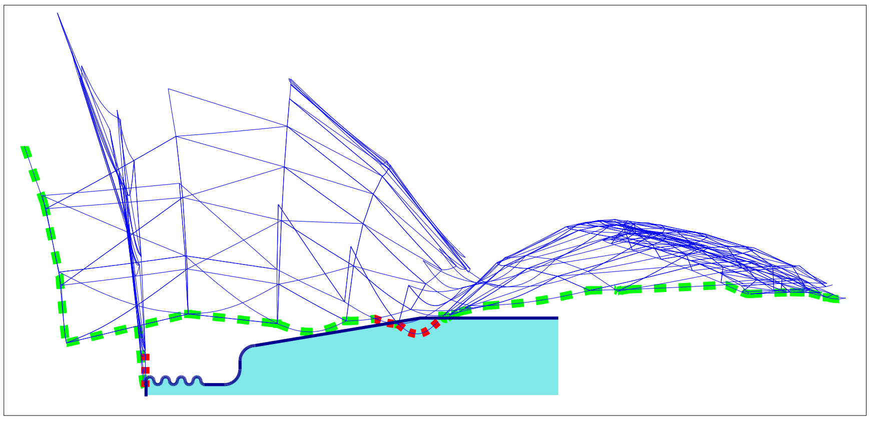


Figure 4.4: The lower envelope of a set of about 800 hyperbolic arcs and line segments is marked with a light dashed line. A complex tool (shaded), with a profile containing 5000 line segments, interferes with the lower envelope in two places, and the lower-envelope arcs it intersects are drawn with darker dotted lines.

the z-axis for $z = z_0$ must be contained in the interior of the cutter, as the z-axis is the symmetry axis of the tool. Without loss of generality we assume that the triangle vertices a, b and c, are given in descending z-order. Let Q be the plane containing $\triangle abc$ and $\vec{N}_Q$ be its normal. Let R be the plane whose normal $\vec{N}_R$ is given by the cross-product of $\vec{N}_Q$ and the z-axis, and which contains the z-axis. It is not difficult to show that the closest points of Q to the z-axis are the points that lie closest to R:

- In case R does not intersect the triangle, the closest points lie on one or on two of the triangle edges.
- If the plane R intersects the interior of $\triangle abc$ such that their intersection forms a segment ef, the closest points to the z-axis in $\triangle abc$ lie on the segments ae, ef, fb and bc (see Figure 4.3 for an illustration).

(This technical issue becomes more difficult to handle in the continuous case — see the next section.)

We can therefore go over all relevant triangles, and for each triangle identify the segments of interest and radially project them onto the $\hat{x}\hat{y}$-plane. As a result, we obtain a set of canonical hyperbolic arcs and of line segments: Notice that the segment ef in Figure 4.3 is coplanar with the z-axis, hence it remains a straight line segment after the projection (and in some degenerate cases, a triangle edge may remain a line segment after the projection — that is, when it is coplanar with the original z-axis). We compute the lower envelope of this set in the $\hat{x}\hat{y}$-plane and identify the closest points to the symmetry axis. Recall that each $\hat{x}$-value represents a z-value in the original coordinate system while the $\hat{y}$-value represents squared distances to this axis, thus the lower envelope identifies the closest point to the z-axis (the tool symmetry-axis) at any z-value.

Recall that all hyperbolic arcs we consider are supported by curves of the form:

$$\hat{y}^2 = \alpha\hat{x}^2 + \beta\hat{x} + \gamma \ .$$

Two such curves may have four distinct intersection points, given by $(\hat{x}_1, \pm\hat{y}_1)$ and $(\hat{x}_2, \pm\hat{y}_2)$. As we consider only curve portions lying in the halfplane $\hat{y} > 0$, it is obvious that two such arcs may intersect at most twice. In addition, every line segment can intersect at most twice with any parabolic arc. We conclude that the complexity of the lower envelope is $O(\lambda_4(n))$, and we can compute it in $O(\lambda_4(n) \log n)$ time, using a divide-and-conquer approach; see Section 2.6.2 for the details. Sharir and Agarwal [SA95] show that $\lambda_4(n) = O(n2^{\alpha(n)})$, where $\alpha(\cdot)$ is the functional inverse of Ackermann's function.

Having computed the lower envelope of the projected triangles, we now perform a simultaneous traversal over the lower envelope and the tool's profile along the z-axis, and compare the two entities. If at some point the profile lies above the lower envelope, we conclude that there is a collision between the tool and the model (see Figure 4.4 for an illustration). The last step is clearly linear in the complexity of the envelope and of the tool, thus the entire process takes $O\big(\lambda_4(n) \log n + m\big)$, where m is the number of segments in the tool's profile.

4.3 The Continuous Case

In the discrete case we computed the lower envelope of the traces of the relevant triangles and compared it to the tool's profile in order to detect collisions. We now show how to extend this approach to detect collisions with the model while the tool is in continuous motion.

We are given a continuous 5-axis tool-path, defined by $(c(t), \vec{w}(t))$, where $c : [0, t_{\max}] \longrightarrow \mathbb{R}^3$ defines the position of the bottom-end of the tool's symmetry axis, while the orientation of the tool at any $0 \le t \le t_{\max}$ is given by the direction of $\frac{\vec{w}(t)}{\|\vec{w}(t)\|}$, where $\vec{w} : [0, t_{\max}] \longrightarrow \mathbb{R}^3$. Both $c(t)$ and $\vec{w}(t)$ are piecewise rational functions and can be represented as B-splines. Our goal is to determine whether this path is collision-free. We shall first show how we approximate the given path by a sequence of purely translational and purely rotational motions, and then we describe how to detect collision for each sub-path in this sequence.

4.3.1 Decomposition of the Tool-Path

Definition 4.1 *Given a tool-path $\mathcal{T} = (c(t), \vec{w}(t))$, with $t \in [0, t_{\max}]$, we call the finite configuration sequence $\mathcal{D} = \big\{(p_0, \vec{o}_0),\ (p_1, \vec{o}_1),\ \ldots, (p_{2k}, \vec{o}_{2k})\big\}$ a* rotational–translational decomposition *(or* Rt-decomposition *for short) of $\mathcal{T}$, if there exist $0 = t_0 < t_1 < \ldots < t_k = t_{\max}$, such that $p_0 = c(0)$ and $p_{2i-1} = p_{2i} = c(t_i)$ for each $1 \le i \le k$, and such that $\vec{o}_{2k} = \vec{w}(t_{\max})$ and $\vec{o}_{2i} = \vec{o}_{2i+1} = \vec{w}(t_i)$ for each $0 \le i \le k-1$.*

It is clear from the definition that if we examine two neighboring configurations $(p_i, \vec{o}_i)$ and $(p_{i+1}, \vec{o}_{i+1})$ in a given Rt-decomposition $\mathcal{D}$, then either $p_i = p_{i+1}$ or $\vec{o}_i = \vec{o}_{i+1}$. Thus, a linear interpolation between each pair of consecutive configurations (we shall refer to it as the *RT-path* induced by $\mathcal{D}$) gives rise to a purely translational or a purely rotational motion.

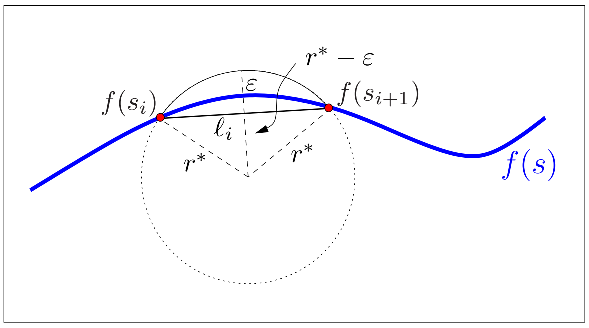


Figure 4.5: Bounding the maximal deviation ε when approximating a rational function $f(s)$ using a polyline.

Definition 4.2 *Let Γ be the bounding cylinder of our tool. Given a tool-path $\mathcal{T}$ and an* Rt-*decomposition $\mathcal{D}$, let $\mathbf{V}$ be the volume defined by sweeping Γ along the original path $\mathcal{T}$ and let $\mathbf{V}'$ be the volume defined by sweeping it along the RT-path induced by $\mathcal{D}$. We define the* approximation error *of the* Rt-*decomposition as the Hausdorff distance, $\mathcal{H}(\mathbf{V}, \mathbf{V}')$,*[2] *between these two volumes.*

Lemma 4.3 *Given a rational arc-length parameterized function $f : [0, L] \longrightarrow \mathbb{R}^2$ (thus L is the arc length) and $\varepsilon > 0$, it is possible to select $0 = s_0 < s_1 < \ldots < s_k = L$ such that the polyline $(f(s_0), f(s_1), \ldots, f(s_k))$ is an ε-approximation of $f(s)$ and $k = O(\frac{1}{\sqrt{\varepsilon}})$.*

Proof: Since $f(s)$ is a rational function, its curvature on the interval $[0, L]$ is bounded. Let us denote the maximal curvature of f by κ^* and let $r^* = \frac{1}{\kappa^*}$ be the minimal radius of curvature. It is possible to select the values $0 = s_0 < s_1 < \ldots < s_k = L$, equally spaced in the interval $[0, L]$, such that $s_{i+1} - s_i \leq 2r^*$. Since $f(s)$ is arc-length parameterized, the length ℓ_i of the line segment $(f(s_i), f(s_{i+1}))$ satisfies: $\ell_i = \|f(s_{i+1}) - f(s_i)\| \leq 2r^*$. For each interval i, we can fit a unique circle of radius r^* supported by the chord $(f(s_i), f(s_{i+1}))$, such that the original function $f(s)$, $s_i < s < s_{i+1}$, lies between the circular arc and the line segment $(f(s_i), f(s_{i+1}))$ (as in Figure 4.5). In order to bound the deviation of the polyline $(f(s_0), f(s_1), \ldots, f(s_k))$ from $f(s)$, we bound the length ℓ_i of the chord in each circular domain by ℓ^*, where:

$$\ell^* = 2\sqrt{r^{*2} - (r^* - \varepsilon)^2} = 2\sqrt{2r^*\varepsilon - \varepsilon^2} = O(\sqrt{\varepsilon}) \tag{4.4}$$

It is now clear that if we select $k = \frac{L}{\ell^*} = O(\frac{1}{\sqrt{\varepsilon}})$, the resulting piecewise linear approximation is at most ε away from the original curve. $\square$

[2]Given two sets X and Y in a metric space, their *Hausdorff distance*, denoted $\mathcal{H}(X, Y)$, is defined as:

$$\mathcal{H}(X, Y) = \max\left\{\sup_{x \in X} \inf_{y \in Y} d(x, y),\ \sup_{y \in Y} \inf_{x \in X} d(x, y)\right\},$$

where $d(\cdot)$ is the distance function of the metric space.

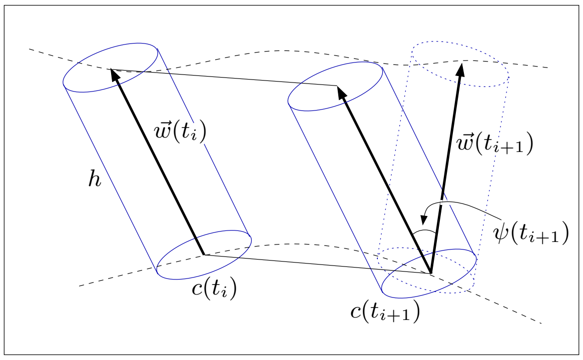


Figure 4.6: Bounding the deviation of the top-end of the symmetry axis of the tool's bounding cylinder. The lower and the upper dashed curves describe the positions of the bottom-end and the top-end of the symmetry axis during the original motion path, respectively.

Corollary 4.4 *Since arc-length parametrization exists for all bounded regular curves,*[3] *Lemma 4.3 holds for all bounded regular rational curves.*

Theorem 4.5 *Given a tool-path $\mathcal{T} = (c(t), \vec{w}(t))$, where both $c(t)$ and $\vec{w}(t)$ are rational functions and can be represented as B-splines in $\mathbb{R}^3$, and $\varepsilon > 0$, it is possible to construct an* Rt*-decomposition $\mathcal{D}$ of size $O(\varepsilon^{-\frac{3}{2}})$ whose distance from the original path is less than ε.*

Proof: For two congruent cylinders, if both the distance between their axial top-ends and the distance between their axial bottom-ends is less than ε, then the Hausdorff distance between the two cylinders is also less than ε. As Γ is the bounding cylinder of the tool, it is therefore sufficient to bound the deviation of the two ends of Γ's axis of symmetry while moving along the RT-path $\mathcal{D}$ that we construct.

We begin by selecting $0 = t_0 < t_1 < \ldots < t_{k_0} = t_{\max}$ such that the polyline $(c(t_0), c(t_1), \ldots, c(t_{k_0}))$ is an $\frac{\varepsilon}{2}$-approximation of the curve $c(t)$. By Lemma 4.3, k_0 is $O(\frac{1}{\sqrt{\varepsilon}})$. We define our initial decomposition $\mathcal{D}^{(0)}$ by using the selected t values. It is clear that the maximal deviation of the bottom-end of the cylinder's symmetry axis while moving along $\mathcal{D}^{(0)}$ is $\frac{\varepsilon}{2}$ since $c(t)$ specifies the position of the bottom-end of the symmetry axis during the motion, so we just have to bound the deviation of its top-end.

Let us assume we have constructed $\mathcal{D}^{(j)}$. We proceed by examining, for each $0 \le i < k_j$, the two motions between $(c(t_i), \vec{w}(t_i))$ and $(c(t_{i+1}), \vec{w}(t_{i+1}))$. We begin by analyzing the purely translational sub-path. It is clear that the distance between the location of the top-end while it moves on $\mathcal{T}$ and while it translates on the interpolated Rt-decomposition between t_i and t_{i+1} is bounded by (we denote the length of the tool, which is also the height of the bounding cylinder, by h):

$$\frac{\varepsilon}{2} + h \cdot \sup_{t_i < t < t_{i+1}} \sin \psi(t) \ ,$$

[3]A parametric curve $f(\tau)$ is called *regular* if its derivative, $f'(\tau)$, never vanishes.

where $\psi(t)$ is the angle between $\vec{w}(t_i)$, which is the constant orientation of the tool along the purely translational motion, and the vector $\vec{w}(t)$ (for $t_i < t < t_{i+1}$). See Figure 4.6 for an illustration.

The function $\vec{w}(t)$ is a B-spline in $\mathbb{R}^3$, with its control polyline defined by the points $q_1, \ldots, q_m \in \mathbb{R}^3$. Since the B-spline is contained in the convex hull of its control points [CER01], the maximal angle between $\vec{w}(t_i)$ and $\vec{w}(t)$ is bounded by one of the vectors $\vec{g}_l = q_l - \vec{w}(t_i)$ (for each $1 \leq l \leq m$), denoted $\vec{g}^*$. We can therefore bound the deviation of the top-end of the symmetry axis from $\mathcal{T}$ by:

$$\frac{\varepsilon}{2} + h \cdot \left\| \frac{\vec{w}(t_i)}{\|\vec{w}(t_i)\|} - \frac{\vec{g}^*}{\|\vec{g}^*\|} \right\| \quad .$$

If the overall error is greater than ε, we construct $\mathcal{D}^{(j+1)}$ with $k_{j+1} = k_j + 1$ by adding the midpoint $\frac{1}{2}(t_i + t_{i+1})$ to our decomposition, thus introducing two additional sub-paths. It is important to notice that the translational error also accounts for the error introduced by the rotation of the cylinder from $\vec{w}(t_i)$ to $\vec{w}(t_{i+1})$, as $\vec{w}(t_{i+1})$ is also contained in the convex hull of the control polygon.

We proceed in this manner until obtaining the desired approximation between each two consecutive configurations. Since $\vec{w}(t)$ is a B-spline, it is a Lipschitz function (see, e.g., [CER01] for more details) — thus, by bisecting the interval $[t_i, t_{i+1}]$ the approximation error is also halved. If we view this splitting process as growing an imaginary binary tree from each of the initial k_0 sub-paths, such that the tree leaves represent the final Rt-decomposition, it is clear that the depth of each tree is $O(\log \frac{1}{\varepsilon})$. Since $k_0 = O(\frac{1}{\sqrt{\varepsilon}})$ the total number of leaves in the forest, and therefore the size of the final Rt-decomposition $\mathcal{D}$, is $O(\varepsilon^{-\frac{3}{2}})$, while the distance of $\mathcal{D}$ from the original tool-path $\mathcal{T}$ is bounded by ε. $\square$

4.3.2 Trace surface patches

Let us assume that we are given a continuous tool-path $(c(t), \vec{w}(t))$. This path is collision-free if for each $t \in [0, t_{\max}]$ the tool profile does not lie above the trace of any model triangle. In other words, we can imagine that the tool remains fixed at the origin with the z-axis being its symmetry axis, while the model triangles continuously move. Thus, for each $t \in [0, t_{\max}]$ and for each triangle $\triangle abc$ we can apply a rigid transformation on each vertex, so the triangle becomes $\triangle a(t)b(t)c(t)$. We can project each triangle edge onto the $\hat{x}\hat{y}$ plane, as we did in the discrete case, and obtain a hyperbolic arc. Note however that this hyperbolic arc changes continuously as t changes, and as a result we obtain a surface patch, defined on the $\hat{x}t$-plane.

Purely translational surface patches

We start with the case where the tool's orientation remains fixed along the sub-path. Given the initial tool position p_0 and the goal position p_1, we first apply a rigid transformation on the entire scene such that p_0 is the origin, with the z-axis being the symmetry axis, and $p_1 = (x_1, 0, z_1)$. Thus, our motion is along the line $z = sx$ in the xz-plane, where $s = \frac{z_1}{x_1}$.

We do not consider translation along the z-axis, as this case can be reduced to the discrete case of collision detection by "stretching" the tool's profile.[4]

We wish to view the scene as if the tool is static and the model is moving. Thus, if the tool moves by ξ along the x-axis, we can change the coordinates of every point $p = (p_x, p_y, p_z)$ in the model to $p(\xi) = (p_x - \xi, p_y, p_z - s\xi)$. If we consider the line segment $a(\xi)b(\xi)$, we can view the coefficients defined in Equation (4.2) as functions of ξ:

$$\begin{aligned} D_x(\xi) &= a_x(\xi) - b_x(\xi) = a_x - b_x = D_x \ , \\ D_y(\xi) &= a_y(\xi) - b_y(\xi) = D_y \ , \\ D_z(\xi) &= a_z(\xi) - b_z(\xi) = D_z \ , \\ E_x(\xi) &= a_x(\xi)b_z(\xi) - a_z(\xi)b_x(\xi) = E_x + (D_z - sD_x)\xi \ , \\ E_y(\xi) &= a_y(\xi)b_z(\xi) - a_z(\xi)b_y(\xi) = E_y - sD_y\xi \ . \end{aligned} \tag{4.5}$$

We can therefore express the hyperbola we obtain in this case using the coefficients of the hyperbola (4.3) we obtained for $\xi = 0$ (by substituting D_x by $D_x(\xi)$, D_y by $D_y(\xi)$, etc.):

$$\begin{aligned} \hat{y}^2 &= \frac{D_x^2 + D_y^2}{D_z^2}\hat{x}^2 \\ &- 2\frac{D_x(E_x + (D_z - sD_x)\xi) + D_y(E_y - sD_y\xi)}{D_z^2}\hat{x} \\ &+ \frac{(E_x + (D_z - sD_x)\xi)^2 + (E_y - sD_y\xi)^2}{D_z^2} \ . \end{aligned} \tag{4.6}$$

Thus, while the tool moves from the origin by ξ, the radial projection of the segment ab sweeps a quadric surface patch in the $\hat{x}\hat{y}\xi$-space, defined over the parallelogram $(a_z, 0)$, $(b_z, 0)$, $(a_z - s\xi, \xi)$, $(b_z - s\xi, \xi)$ in the $\hat{x}\xi$-plane.

Purely rotational surface patches

Here we shall assume that the position of the tool remains fixed and only its orientation changes. We further assume, without loss of generality, that at first the tool tip is positioned at the origin, with the z-axis being its symmetry axis, and then it is continuously rotated counterclockwise on the xz-plane.

Once again, instead of rotating the tool by θ, we can imagine that the tool is fixed and that the model is rotated by $-\theta$. Thus, each model point $p = (p_x, p_y, p_z)$ is transformed into $p(\theta) = (p_x \cos\theta + p_z \sin\theta, p_y, p_z \cos\theta - p_x \sin\theta)$. Looking at the coefficients in Equation (4.2), it is clear that $D_y(\theta) = D_y$, and it is easy to show that $E_x(\theta) = E_x$ using elementary trigonometric equalities. The other coefficients can be expressed as follows (we let $F = a_xb_y - a_yb_x$):

[4]If the tool is translated from the origin to $p_1 = (0, 0, z_1)$, we can examine the area swept by the tool during this translation and compute its envelope, so we only have to verify that this envelope (which is a polyline, like the original profile) does not interfere with any of the projected triangles.

$$\begin{aligned} D_x(\theta) &= a_x(\theta) - b_x(\theta) = D_x \cos\theta + D_z \sin\theta \ , \\ D_z(\theta) &= a_z(\theta) - b_z(\theta) = D_z \cos\theta - D_x \sin\theta \ , \\ E_y(\theta) &= a_y(\theta)b_z(\theta) - a_z(\theta)b_y(\theta) = E_y \cos\theta + F \sin\theta \ , \end{aligned} \tag{4.7}$$

so we can substitute these expressions into Equation (4.3) and express $\hat{y}^2$ as a function of $\hat{x}$ and θ. Moreover, if we parameterize the rotation by $\tau = \tan\frac{\theta}{2}$ (thus $\sin\theta = \frac{2\tau}{1+\tau^2}$ and $\cos\theta = \frac{1-\tau^2}{1+\tau^2}$), we obtain that during the rotation the radial projection of a segment sweeps a patch of an algebraic surface of degree 6.

4.3.3 Silhouettes of surfaces

We saw that in case of a purely translational or a purely rotational motion, the hyperbolic segment obtained by radially projecting a triangle edge ab with respect to the z-axis onto the $\hat{x}\hat{y}$-plane forms an algebraic surface patch in the $\hat{x}\hat{y}t$-space, where $t \in [0, t_{\max}]$ parameterizes the motion of the tool. The surfaces we obtain are graphs of bivariate functions (which we refer to as *terrains*) of the form $\hat{y} = \sqrt{S_{ab}(\hat{x}, t)}$, where S is a polynomial in case of translation and a rational function in case of rotation.

A straightforward way to proceed is to compute the lower envelope of the set of surface patches we obtain from all relevant segments in the scene, and compare it with the surface obtained by sweeping the tool's profile along the t-axis. This operation, however, involves the maintenance of the two-dimensional lower envelopes of rather complicated surfaces. Meyerovitch [Mey06] has recently implemented a Cgal package that is capable of constructing lower envelopes of arbitrary surfaces, yet this construction may incur prohibitive running times for the kind of surfaces we have to consider. Instead, we reduce the collision-detection problem to a problem on lower envelopes of planar curves.

Definition 4.6 *Given a terrain $S_{ab}(\hat{x}, t)$, its* silhouette[5] *curve is given by the following function:*

$$\mathrm{sil}_{ab}(\hat{x}) = \inf_t \sqrt{S_{ab}(\hat{x}, t)} \ . \tag{4.8}$$

The silhouette function can be computed for each value of $\hat{x}$ using the following algorithm: First, we fix $\hat{x} \longleftarrow x_0$, so $S_{ab,x_0}(t) = S_{ab}(x_0, t)$ is a function of one variable t, defined over some interval $[t_{\mathrm{low}}(x_0), t_{\mathrm{high}}(x_0)]$. We derive this function and find all the solutions for $S'_{ab,x_0}(t) = 0$ in order to locate the minima of this function. Let $t_1, \ldots, t_m$ be the zeroes of S'_{ab,x_0}. We can then locate $t^*(x_0) = \arg\min_i S_{ab,x_0}(t_i)$. As we also have to account for the endpoints of our interval, the desired value of the silhouette function at x_0 is therefore:

$$\mathrm{sil}_{ab}(x_0) = \sqrt{\min\left\{S_{ab}(x_0, t^*(x_0)),\ S_{ab}(x_0, t_{\mathrm{low}}(x_0)),\ S_{ab}(x_0, t_{\mathrm{high}}(x_0))\right\}} \ .$$

[5]The silhouette is often referred to as the *envelope*. To avoid confusion with lower envelopes of finite sets of curves, we stick with the term silhouette.

Corollary 4.7 *Given a motion path of the tool which is parameterized by t, the tool does not collide with the segment ab if and only if the silhouette of this segment lies above the tool in the $\hat{x}\hat{y}$-plane.*

The continuous collision-detection problem is thus reduced to computing the lower envelope of a collection of planar silhouette curves. A collision during the motion occurs if and only if this envelope does not lie strictly above the profile of the tool. We continue by stating two lemmas regarding the nature of the silhouette curves. We give the proofs of these lemmas in the following two subsections.

Lemma 4.8 *The silhouette of the surface patch swept by the radial projection of a line segment as it continuously translates in the xz-plane is a continuous curve comprising at most four hyperbolic arcs.*

Lemma 4.9 *The silhouette of the surface patch swept by the radial projection of a line segment as it continuously rotates in the xz-plane can be approximated, for any given $\eta > 0$, using $O(\frac{1}{\sqrt{\eta}})$ algebraic arcs of a low degree, such that the relative approximation error is bounded by η.*

As we can robustly compute the lower envelope of a set of line segments and hyperbolic arcs, and of a set of low-degree algebraic arcs, we can conclude that:

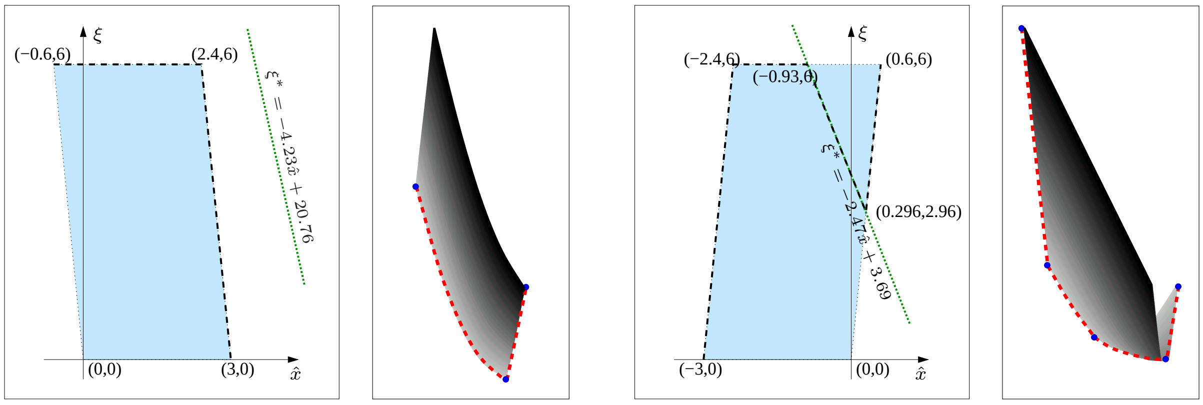
Corollary 4.10 *Let $P(\hat{x})$ be a piecewise linear function that describes the profile of the tool over $[\hat{x}_{\min}, \hat{x}_{\max}]$ and let $\mathcal{L}_\pi(\hat{x})$ be the lower envelope of the set of silhouette curves defined by the radial projection of the relevant triangles onto the $\hat{x}\hat{y}$-plane, as the tool moves along a sub-path π of the* Rt*-decomposition:*

- *If π is a translational sub-path and $P(\hat{x}) < \mathcal{L}_\pi(\hat{x})$ for all $\hat{x} \in [\hat{x}_{\min}, \hat{x}_{\max}]$ (if the lower envelope contains "gaps", we set the value of $\mathcal{L}_\pi(\hat{x})$ to ∞ over these gaps), then the translational motion of the tool along π is collision-free.*
- *If π is a rotational sub-path and $P(\hat{x}) + \eta < \mathcal{L}_\pi(\hat{x})$ for all $\hat{x} \in [\hat{x}_{\min}, \hat{x}_{\max}]$ then the rotational motion of the tool along π is collision-free.*

Translational silhouettes

If we fix $\hat{x} \longleftarrow x_0$ in Equation (4.6), we can re-arrange it so it becomes:

$$\begin{aligned}
\hat{y}^2 &= S_{ab,x_0}(\xi) = \frac{(D_z - sD_x)^2 + s^2 D_y^2}{D_z^2}\xi^2 \\
&+ 2\frac{E_x(D_z - sD_x) - sD_yE_y + \big(s(D_x^2 + D_y^2) - D_xD_z\big)x_0}{D_z^2}\xi \\
&+ \frac{(D_x^2 + D_y^2)x_0^2 - 2(D_xE_x + D_yE_y)x_0 + E_x^2 + E_y^2}{D_z^2}\ . \qquad (4.9)
\end{aligned}$$



(a) - i (a) - ii (b) - i (b) - ii

Figure 4.7: The computation of translational silhouettes: The figures marked by 'i' show the parallelograms over which the surface patch is defined in the $\hat{x}\xi$-plane, along with a light dotted line showing ξ^* as a function of $\hat{x}$. The silhouette is formed by selecting, for each $\hat{x}$ value, a ξ value in the parallelogram which is closest to this line (the thick dashed polyline). The figures marked by 'ii' show the hyperbolic arcs on the $\hat{x}\hat{y}$-plane as they continuously change from $\xi = 0$ (the black end) to $\xi = \xi_{\max}$ (the light gray end). The silhouette curves are also drawn. We examine here two cases: (a) The segment endpoints are $(40, -20, 0)$ and $(20, -30, 3)$ with $\xi_{\max} = 6$ and $s = 0.1$. The line $\xi^* = -4.23\hat{x} + 20.76$ does not intersect the parallelogram, and the silhouette therefore contains two hyperbolic arcs. (b) The segment endpoints are $(9, 2, -3)$ and $(3, 1, 0)$ with $\xi_{\max} = 6$ and $s = -0.1$. The line $\xi^* = -2.47\hat{x} + 3.69$ crosses the parallelogram, so the silhouette comprises four hyperbolic arcs.

The expression on the right-hand side is a parabola of the general form $f(\xi) = \alpha\xi^2 + 2\beta\xi + \gamma$, with $\alpha > 0$. This parabola has a single minimum, obtained at $\xi^* = -\frac{\beta}{\alpha}$ where $f(\xi^*) = \gamma - \frac{\beta^2}{\alpha}$.

If we use the parabola coefficients from Equation (4.9) we obtain the minimum at:

$$\xi^*(x_0) = \frac{\big(s(D_x^2 + D_y^2) - D_x D_z\big)x_0 + (D_z - sD_x)E_x - sD_y E_y}{(D_z - sD_x)^2 + s^2 D_y^2} \,. \tag{4.10}$$

Since ξ^* is a linear function of $\hat{x}$, it is clear that when we substitute $\xi^*(x_0)$ into Equation (4.9), we obtain that $\hat{y}^2$ is a quadratic expression of x_0. The silhouette curve is therefore a hyperbolic arc.

One should notice, however, that $S_{ab}(\hat{x}, \xi)$ is defined over the parallelogram $(a_z, 0)$, $(b_z, 0)$, $(a_z - s\xi_{\max}, \xi_{\max})$, $(b_z - s\xi_{\max}, \xi_{\max})$ in the $\hat{x}\xi$-plane, where $\xi_{\max}$ equals the x-coordinate of the target position of the translation. If we assume that $s > 0$, it is clear that the range $[\xi_{\text{low}}(x_0), \xi_{\text{high}}(x_0)]$ of valid ξ values for a fixed x_0 is defined as follows:

$$\begin{aligned}
\xi_{\text{low}}(x_0) &= \max\left\{0, \frac{a_z - x_0}{s}\right\} \,, \\
\xi_{\text{high}}(x_0) &= \min\left\{\xi_{\max}, \frac{b_z - x_0}{s}\right\} \,.
\end{aligned}$$

The analysis for $s < 0$ is similar, and if $s = 0$ we simply take $\xi_{\text{low}} = 0$ and $\xi_{\text{high}} = \xi_{\max}$. Since $\xi^*(x_0)$ as given in Equation (4.10) is a linear expression in x_0, it can intersect at most two edges of the parallelogram. Thus, the relevant $\hat{x}$-range of the silhouette curve should be subdivided into at most four intervals. In at most one of the intervals we use $S_{ab,x_0}(\xi^*(x_0))$, while for the other intervals the silhouette is $S_{ab}(\hat{x}, 0)$, or $S_{ab}(\hat{x}, \frac{a_z-\hat{x}}{s})$, or $S_{ab}(\hat{x}, \xi_{\max})$, or $S_{ab}(\hat{x}, \frac{b_z-\hat{x}}{s})$, depending on the identity of the relevant parallelogram edge. In any case, the silhouette is a hyperbolic arc over each interval. See Figure 4.7 for an illustration of two typical cases.

Computing the silhouette of the line segment that contains the closest triangle points to the z-axis (we shall call it the *z-segment* for short), when such a segment exists (see Section 4.2), is a lot easier. Since the normal to the triangle $\triangle abc$ does not change as the triangle translates, the equation of the supporting line of the z-segment does not depend on ξ. To find the silhouette of the z-segment we only need to determine the endpoints of this silhouette segment.

Rotational silhouettes

As mentioned in Section 4.3.2, working with an exact parametrization of the rotation yields an algebraic surface patch of degree 6, and it is therefore impossible to obtain a closed-form representation of the silhouette curve, as we did in the case of purely translational motions. To make our analysis simpler, we use instead the first-order Taylor approximations $\sin\theta \simeq \theta$ and $\cos\theta \simeq 1$ in our calculations, assuming the rotation angle is small.[6] In case of larger rotation angles, we can break the rotational motion into several contiguous rotations with sufficiently small rotation angles: If the rotation angle is $\theta_{\max}$ and we wish our relative error to be smaller that η, we can divide the motion into $\lceil h\frac{\theta_{\max}}{\sqrt{\eta}}\rceil = O(\frac{1}{\sqrt{\eta}})$ continuous rotations, where h is the height of the tool profile. As the error introduced by the first-order Taylor approximation of the sine and cosine functions for an angle θ is bounded by $O(\theta^2)$.

Using the first-order Taylor approximation, we obtain from Equation (4.7):

$$\begin{aligned} D_x(\theta) &\simeq D_x + D_z\theta \ , \\ D_z(\theta) &\simeq D_z - D_x\theta \ , \\ E_y(\theta) &\simeq E_y + F\theta \ . \end{aligned}$$

Fixing $\hat{x} \longleftarrow x_0$, we can express $\hat{y}^2 = S_{ab,x_0}(\theta)$ as a rational function of the form $f(\theta) = \frac{\alpha\theta^2+2\beta\theta+\gamma}{(D_z-D_x\theta)^2}$, where:

$$\begin{aligned} \alpha &= D_z^2x_0^2 + F^2 > 0 \ , \\ \beta &= D_xD_zx_0^2 - (D_zE_x + D_yF)x_0 + E_yF \ , \\ \gamma &= (D_x^2 + D_y^2)x_0^2 - 2(D_xE_x + D_yE_y)x_0 + E_x^2 + E_y^2 \ . \end{aligned}$$

[6]For example, the relative error for a unit-length tool introduced by the first-order Taylor approximation is less than $\frac{1}{1000}$ for angles smaller than 3° and less than $\frac{4}{1000}$ for angles smaller than 5°.

This function is defined everywhere except for $\theta = \frac{D_z}{D_x}$. When we derive $f(\theta)$ we get:

$$\begin{aligned} f'(\theta) &= \frac{(2\alpha\theta + 2\beta)(D_z - D_x\theta)^2 + 2D_x(D_z - D_x\theta)(\alpha\theta^2 + 2\beta\theta + \gamma)}{(D_z - D_x\theta)^4} = \\ &= \frac{(2\alpha\theta + 2\beta)(D_z - D_x\theta) + 2D_x(\alpha\theta^2 + 2\beta\theta + \gamma)}{(D_z - D_x\theta)^3} = \\ &= \frac{2}{(D_z - D_x\theta)^3}\Big((\alpha D_z + \beta D_x)\theta + \beta D_z + \gamma D_x\Big) . \end{aligned}$$

Solving $f'(\theta) = 0$ we get that our rational function has a single extremum, obtained at $\theta^*(x_0) = -\frac{\beta D_z + \gamma D_x}{\alpha D_z + \beta D_x}$. It is easy to show that this is a minimum, since both coefficients of θ^2, in the numerator and in the denominator of $f(\theta)$, (α and D_x^2, respectively) are positive and thus $f''(\theta^*) > 0$. By substituting $\theta^*(x_0)$ into $S_{ab,x_0}(\theta)$, we get a rational function in x_0, whose numerator is a polynomial of degree four and whose denominator is a quadratic polynomial:

$$S_{ab}(\hat{x}, \theta^*(\hat{x})) = \frac{\sum_{i=0}^{4} \nu_i \hat{x}^i}{\sum_{i=0}^{2} \delta_i \hat{x}^i} , \tag{4.11}$$

where:

$$\begin{aligned} \nu_4 &= D_x^2 D_y^2 , \\ \nu_3 &= 2D_y D_z (D_x F - D_z E_y) , \\ \nu_2 &= D_z^2 E_y^2 + D_x^2 F^2 + 2D_y D_z (E_x + E_y) F , \\ \nu_1 &= 2E_x F (D_z E_y - D_x F) , \\ \nu_0 &= E_x^2 F^2 , \\ \delta_2 &= (D_x^2 + D_z^2)^2 + D_x^2 D_y^2 , \\ \delta_1 &= -2D_x\big(D_z(D_z E_x + D_y F) + D_x(D_x E_x + D_y E_y)\big) , \\ \delta_0 &= D_x^2 E_x^2 + (D_x E_y + D_z F)^2 . \end{aligned}$$

Notice that the surface patch given by $S_{ab}(\hat{x}, \theta)$ is defined over the trapezoid $(a_z, 0)$, $(b_z, 0)$, $(a_z, a_z \cos\theta_{\max} - a_x \sin\theta_{\max})$, $(b_z, b_z \cos\theta_{\max} - b_x \sin\theta_{\max})$ in the $\hat{x}\theta$-plane, but as we assume that the rotation angle is small, we can approximate this trapezoid as a rectangle such that $\theta_{\text{low}}(x_0) = 0$, $\theta_{\text{high}}(x_0) = \theta_{\max}$ for each $x_0 \in [a_z, b_z]$. As we did in the translational case, for each $\hat{x}$ value we choose a $0 \le \theta \le \theta_{\max}$ value which is closest to $\theta^*(x_0)$. Our silhouette may therefore contain a constant number of segments, some are rational arcs (as described in the previous paragraph), while for the others we use the value of either $S_{ab}(\hat{x}, 0)$ or $S_{ab}(\hat{x}, \theta_{\max})$ and obtain hyperbolic arcs.

4.3.4 The overall algorithm

We are now ready to describe an algorithm for detecting collisions between a moving tool and a 3-D model formed by triangles. We first break the motion path into sub-paths, such that the motion in each sub-path is either purely translational or purely rotational, as described in Section 4.3.1. We now construct the LDQ data structure (see Section 4.2), where each triangle in the LDQ structure is offset by $\rho = r + \Delta$ where r is the radius of the bounding cylinder of the tool and Δ is the maximum distance traveled by any point in this cylinder during the execution of a sub-path. For each sub-path, we obtain from our LDQ structure the set of all relevant model triangles and detect potential collisions with them using the following process (where $t \in I \subseteq [0, t_{\max}]$ parameterizes the motion in this sub-path):

1. For each relevant model triangle $\triangle abc$:
   (a) Determine if for any continuous set $I_s \subseteq I$ of t values the plane R intersects the triangle and forms a line segment, as in the example depicted in Figure 4.3. If so, compute the silhouette of this segment in I_s. This silhouette describes the closest points on the triangle to the z-axis during the motion.
   (b) Compute the silhouette of each triangle edge over the entire interval I.
2. Compute the lower envelope of all silhouette curves obtained from the previous stage.
3. Compare the profile of the tool to the lower envelope. The motion in the sub-path we examine is collision-free if and only if the profile of the tool lies strictly below the lower envelope we have computed in Step (2).

We have showed that if we have n relevant triangles, the number of curves that form the silhouettes is $O(n)$, so that the entire process for a single sub-path takes $O(\lambda_{\sigma+2}(n) \log n + m)$ time, where σ is the maximal number of intersections between any two silhouette curves and m is the complexity of the tool.

4.4 Implementation Details

The collision-detection algorithm presented in the previous sections was implemented in the IRIT modeling environment.[7] The lower envelope calculations and the comparison of the envelope with the tool profile are carried out using the 2D envelope package of CGAL; see Section 2.6.2 for the details.

In the envelope computations we have to support operations on line segments and hyperbolic arcs, both types of curves are special cases of conic arcs, which can be handled in an exact manner by CGAL's arrangement package (see Section 2.3.3). However, as the hyperbolic arcs we handle in the discrete case are supported by canonical hyperbolas whose major axis is the x-axis, the intersections between such curves can actually be computed only by solving quadratic equations (see more details in [IEH+04, WH04]). This fact not only simplifies the code that handles the geometric constructions and predicates for our planar curves, but also helps in significantly reducing the running times of the program when performing an exact computation of the lower envelope.

[7]IRIT modeling environment, © G. Elber, Department of Computer Science, Technion: ⟨http://www.cs.technion.ac.il/~irit/⟩.

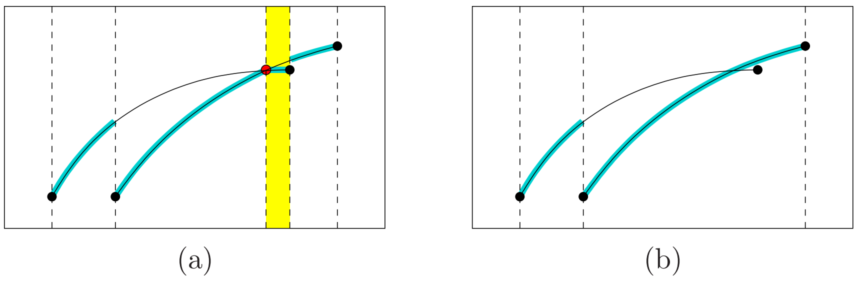


Figure 4.8: (a) The lower envelope of two hyperbolic arcs, with their intersection point lying very close to the right endpoint of one of the arcs. (b) The intersection point is missed, due to floating-point errors, and the lower envelope is erroneously computed as a result.

In order to benefit from the special properties of our arcs, we have implemented a special traits class that handles only line segments and hyperbolic arcs, and used it instead of the generic traits class for conic arcs distributed with CGAL. Our traits class is named `Arr_hyperbolic_arc_traits_2<Kernel>`, and can represent arcs of curves of the form $y^2 = Ax^2 + Bx + C$, where $A, B, C \in \mathbb{Q}$. Like the circle/segment traits class presented in Section 2.3.3, this traits class is parameterized by a kernel that supports exact rational arithmetic, while the coordinates of the `Point_2` type it defines are one-root numbers (note that the x-coordinates of the intersection points between two arcs are solutions of a quadratic equation with rational coefficients, and the squared y-coordinates are rational expressions involving these solutions). Indeed, rather than storing the y-coordinates of the points, we store their *squared* values, in order to avoid unnecessary square-root operations. The hyperbolic-traits class, which contains all the necessary geometric predicates and constructions needed for the computation of lower envelopes in our case, is more compact and more efficient than the generic conic-traits class, and has the advantage of using only exact rational arithmetic.

We mention that in order to expedite the computation even further, it is possible to instantiate the hyperbolic-traits class with a kernel that uses floating-point arithmetic. Indeed, the lower-envelope construction is prone to errors when relying floating-point arithmetic, but it is possible to show that these errors can be bounded. For example, assume we merge two lower envelopes and have to compute the intersection points of two hyperbolic arcs. We start by computing the intersection points of the supporting hyperbolas that have positive $\hat{y}$-coordinates, and for each point check if it lies in the $\hat{x}$-range of both arcs. Let us assume that the two arcs have one intersection point lying very close to the right endpoint of one of the arcs, and because of floating-point errors we mistakenly decided that this point is not in the $\hat{x}$-range of this arc and missed the fact that the two arcs switch their position. This will lead to a wrong representation of the output lower envelope, but as we are very close to an endpoint of one of the arcs, the error will actually be very small, as illustrated in Figure 4.8. Moreover, the error is local and does not affect other regions of the envelope.

The continuous case gives rise to more difficult computations. Given a tool-path, we use the algorithm presented in Section 4.3.1 to approximate it using an RT-decomposition path $\mathcal{D}$, while guaranteeing that the error introduced by this approximation is bounded by a given $\varepsilon > 0$. We proceed by computing the lower envelope of all relevant silhouette curves along

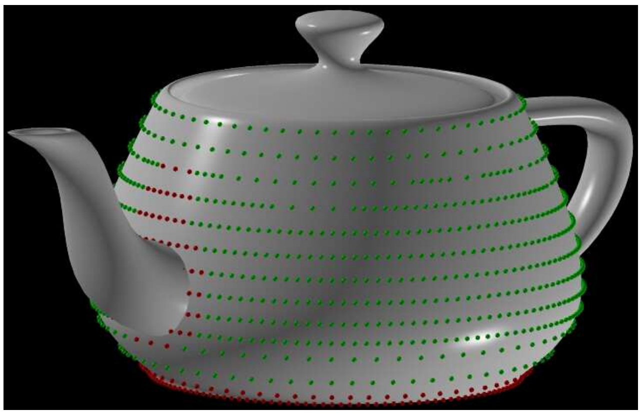


Figure 4.9: The endpoints of the Rt-decomposition of a tool-path over the body of the Utah teapot model. Green (lightly shaded) points designate collision-free tool motions, red (darker) points indicate gouging.

each sub-path of purely translational motion in $\mathcal{D}$ and verifying that the profile of the tool lies strictly below this envelope. We actually compare the lower envelope to the profile of the tool inflated by ε, to account for the approximation error and guarantee that we never accept a path that causes a collision between the tool and the model (but we allow "false alarms" – that is, we may reject paths that do not cause any collisions).

As we showed in the proof of Theorem 4.5, we examine each translational sub-path and guarantee that the deviation of the translation from $c(t_i)$ to $c(t_{i+1})$ with a fixed orientation of $\vec{w}(t_i)$ is bounded by ε. From the construction of the Rt-decomposition (see Lemma 4.3), it is clear that if we rotate the tool to the goal orientation $\vec{w}(t_{i+1})$ without changing its position we only decrease this deviation. As a consequence, the deviations caused by the rotational motions from the original tool-path we wish to verify are already accounted for by the purely translational sub-paths, hence we do not have to verify the sub-paths of purely rotational motions. This allows us to work just with lower envelopes of hyperbolic arcs and line segments, allowing exact computations and yielding a running time of $O(\lambda_4(n)\log n + m)$ for each sub-path we examine, where n is the number of relevant model triangles and m is the complexity of the tool, asymptotically the same as in the discrete case.

It should be noted that even computing translational silhouettes is far from being trivial, and it is necessary to examine several cases in order to obtain correct results (see Figure 4.7). Among the steps, one has to form a vertical decomposition of the parallelogram over which the surface patch is defined to (at most three) pseudo-trapezoids, compute the silhouette over each such pseudo-trapezoid and finally stitch the silhouettes together.

4.5 Experimental Results

We begin with a simple experiment that demonstrates the strength of our continuous collision checker. In this experiment, we use the Utah teapot model, which consists of 12600 triangles (see Figure 4.9), as our workpiece. We wish to verify a circular tool-path, where the milling

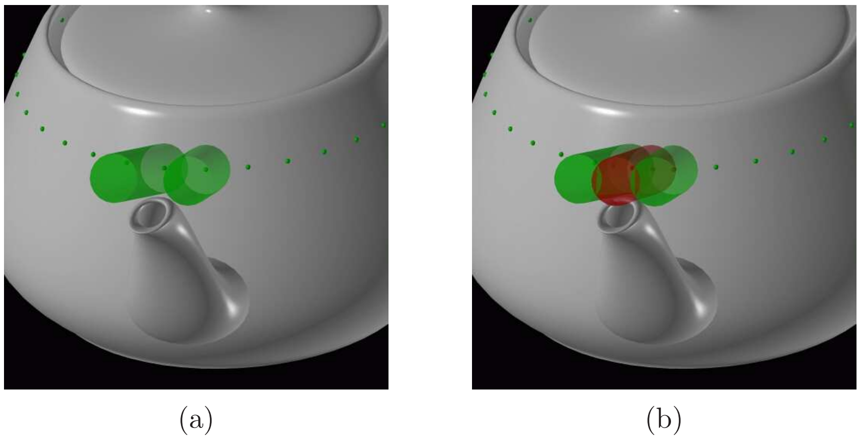


Figure 4.10: A single circular tool-path around the body of the *Utah teapot*: Discrete collision tests at the intermediate configurations miss the interference with the teapot's spout (a), where two relevant free positions of the cylindrical tool are shown in green (lighter color). The continuous collision detector, in contrast, discovers the gouging with the teapot's spout in-between these configurations (b), where the colliding configuration is shown in red (the darker cylinder in the middle).

cutter slides around the teapot without touching its surface and with its symmetry axis perpendicular to the teapot. We select a value for ε and generate an RT-decomposition that approximates the original tool-path. The endpoints of the sub-paths we obtain are shown in Figure 4.10. If we perform a discrete collision check at each of these configurations, we may reach the (wrong) conclusion that the path is collision-free. On the other hand, if we use our continuous collision detector, we detect a collision between the tool and the teapot's spout. It is possible of course to use a denser sample of intermediate configurations and use the discrete collision detector, but this will considerably increase the number of LDQ queries and lower-envelope computations.

Table 4.1 shows some statistics for the path verification using several models. In the Utah teapot model we use circular paths covering the teapot body, while in the *wineglass* model (Figure 4.11(a)) we use a ball-end cutter with the tool-path given by an offset surface, such that only its tip touches the interior of the glass. The tool-path we verify in the *turbine* model (Figure 4.11(b)) goes over each of the 30 turbine blades. We examine different values of ε, and show the size k of the RT-decomposition, which dominates the total time needed for the path verification.

For each translational sub-path generated by the RT-decomposition, we have calculated the number of relevant triangles $N_{\triangle}$, the number of silhouette arcs N_C we consider (recall that the silhouette of each of the three triangle edges may contains up to four arcs) and the number N_{env} of arcs in the lower envelope. Note that these values tend to decrease with ε, as the average length of each sub-path becomes shorter while k grows, hence the number of triangles it can potentially intersect with is smaller. We summarize the average values in Table 4.2.

On a Pentium IV 2.4 GHz machine with 512 MB of RAM, the time needed to compute the *exact* representation of the lower envelope of a set of 800 hyperbolic arcs and line segments, as in Figure 4.4, is 4.3 seconds when using the conic-traits class with the algebraic number

Table 4.1: Verifying tool-paths with varying ε values. Times are given in seconds and were obtained on a Pentium IV 2.4 GHz machine with 512 MB of RAM. The model size is the number of triangles that comprise the model.

Model Name	Model Size	ε	k	Total Time	Average Query Time
wineglass	2700	0.5	911	32	0.0351
		0.1	4902	73	0.0149
		0.05	9837	141	0.0143
teapot	12600	0.5	1288	122	0.0945
		0.1	6872	245	0.0357
		0.05	13760	456	0.0331
turbine	40046	0.05	2430	255	0.1048
		0.04	10900	733	0.0672

Table 4.2: Average query complexity for different models and varying ε values.

Model Name	ε	Average $N_{\triangle}$	Average N_C	Average N_{env}
wineglass	0.5	234	1751	61
	0.1	164	1127	86
	0.05	156	1051	91
teapot	0.5	378	3137	53
	0.1	211	1488	43
	0.05	192	1310	48
turbine	0.05	555	3877	152
	0.04	526	3639	154

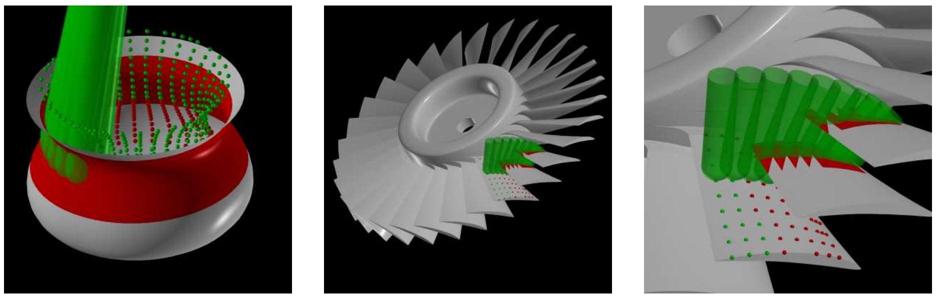

(a) (b) (c)

Figure 4.11: Path verification for other models: (a) The wineglass model. (b) The turbine model, comprised of more than 40000 triangles, (c) zooming in on several collision-free (lightly shaded points) and interfering (darker points) tool positions on one of the turbine blades.

types provided by the Leda library, and 3.7 seconds using the Core number-types (see Section 2.3.3). However, as we explained in Section 4.4, we can employ the specialized hyperbolic-traits class that uses only rational arithmetic to robustly compute the envelope. The running time in this case drops to 0.6 seconds. When using this traits class with the machine double-precision floating-point arithmetic, the envelope is computed in 0.04 seconds. Indeed, the lower envelope we compute in this latter case may slightly deviate from the real lower envelope, but as was mentioned in Section 4.4 we can bound the approximation error. Our experiments also show that these deviations are negligible and that the overall accuracy of the collision checker is not damaged. In fact, we performed all our large-scale experiments using floating-point arithmetic and obtained satisfactory results. The running times reported in Table 4.1 are based on floating-point arithmetic alone. We emphasize however that the computations can also be performed using exact arithmetic — this may be feasible in some cases, as the tool-path verification process can be performed offline.

We do not show the times needed for the construction Ldq data structure and for the Rt-decomposition of the input path, both being negligible in comparison to the sequence of lower-envelope computations. For example, it takes about 0.5 seconds to construct the Ldq data structure for the teapot and the wineglass models and 1.5 seconds for the turbine model.

It should be noted that the complexity of the tool has almost no effect on the running times, as the time needed for the comparison between the lower envelope and the tool's profile is negligible with respect to the lower-envelope construction time.

◇◇
◇

In this chapter we introduced a novel approach for detecting collisions between a moving milling-cutter and a solid workpiece. Unlike traditional approaches, which sample the tool-path at a finite number of configurations and verify each one separately, we are able to verify the entire continuous motion path of the tool. In order to allow exact computations we limit ourselves to verify only purely translational or purely rotational motions, but show that any tool-path can be approximated by such motion to any desired precision.

It is possible to use the discrete and the continuous collision checkers we developed as oracles for the construction of a probabilistic roadmap. Namely, the static collision check can be used to determine whether a random configuration is free or not, while the continuous collision check can be utilized for checking whether two nearby configurations can be connected using a simple path. We mention that while composing a valid tool-path from scratch remains a difficult problem, we can use such a roadmap for planning correction paths for the tool whenever we detect that the original path, given to us for verification, incurs collisions.

Chapter 5

The Visibility–Voronoi Complex

Having developed the software tools that enable us to solve simple variants of the motion-planning problem in a robust manner, we now study the problem of planning a natural-looking collision-free path for a robot with two degrees of motion freedom moving in the plane among polygonal obstacles. By "natural-looking" we mean the following: (a) the path should be *short* and avoid long detours when significantly shorter routes are possible; (b) it should have a guaranteed amount of *clearance* — that is, the distance of any point on the path to the closest obstacle should not be lower than some prescribed value; and (c) it should be *smooth*, not containing any sharp turns.

Requirements (b) and (c) may conflict with requirement (a): The *visibility graph* is a well-known data structure for computing the shortest collision-free path between a start and a goal configuration (see, e.g., [dBvKOS00, Chapter 15]). However, along some portions of such a shortest path the robot is in contact with the obstacles. This not only looks unnatural, it is also unacceptable for many applications. On the other hand, planning motion paths using the *Voronoi diagram* of the obstacles [ÓY85] yields a path with maximal clearance, but this path may be significantly longer than the shortest path possible and may also contain sharp turns.

We suggest a hybrid of these two latter approaches, called the $\mathrm{VV}^{(c)}$-*diagram* (the *Visibility–Voronoi diagram* for clearance c), yielding natural-looking motion paths, meeting all three criteria mentioned above (with some reservations in narrow passages, as we note in Section 5.2). It evolves from the visibility graph to the Voronoi diagram as c grows from 0 to ∞, where c is the preferred amount of clearance. The $\mathrm{VV}^{(c)}$-diagram contains the visibility graph of the obstacles dilated with a disc of radius c. The dilated obstacle vertices become circular arcs in this case, and the visibility edges are bitangent to these arcs. This guarantees that the paths in the diagram are not only *short* but also *smooth*. However, due to this obstacle inflation, narrow passages in the scene may disappear, which implies that it is not possible to pass through these narrow passages keeping a distance of at least c from the obstacles. As we still want to keep the option of traversing these narrow passages (for example when a path going through a narrow passage is significantly shorter than any alternative path), we integrate into the diagram paths with the maximal possible clearance in regions where the preferred clearance c cannot be obtained. It is easy to see that these additional paths are portions of the Voronoi diagram of the original obstacles.

Besides the straightforward algorithm for constructing the $\mathrm{VV}^{(c)}$-diagram for a given clearance value c, we also propose an algorithm for preprocessing a scene of configuration-space polygonal obstacles and constructing a data structure called the *visibility–Voronoi complex*, or VV-*complex* for short.[1] The VV-complex can be used to efficiently plan motion paths for any start and goal configuration and any given clearance value c, without having to explicitly construct the $\mathrm{VV}^{(c)}$-diagram for that c-value. We achieve this by performing a Dijkstra search on an implicitly constructed graph encoded by the VV-complex. The preprocessing time is $O(n^2 \log n)$, where n is the total number of obstacle vertices, and the query takes $O(n \log n + \ell)$ time, where ℓ is the number of edges encountered during the search and is bounded by the number of diagram edges. Furthermore, we reduce the number of costly geometric operations in the query stage and perform the most time-consuming computations in the preprocessing stage.

A direct application of our construct is planning natural translational motion paths for a polygonal robot among polygonal obstacles. We can compute the Minkowski sum of each obstacle with the robot rotated by π to obtain a set of configuration-space obstacles, which are also polygonal. After this initial step we may assume that the robot is a point. Constructing the $\mathrm{VV}^{(c)}$-diagram of these configuration-space obstacles and giving adequate weights to the diagram edges (see the discussion in Section 5.2) yield more natural motion paths, compared, for example, to the implementation of [AFH02].

The principles of our construction may also be applied to sensor-based coverage using a robot with a limited sensor radius. Acar *et al.* [ACA01] devised an algorithm for a disc robot of radius r, carrying a detector with a range $R > r$, to detect all points in an unknown environment. They decompose the free space into *vast* cells, where the robot traverses the boundary of the obstacles dilated by radius R, and *narrow* cells, where the obstacles are within the detector range and the robot has to follow the Voronoi diagram of the obstacles. It is possible to use the $\mathrm{VV}^{(c)}$-diagram in this case for traversing the narrow cells, as it naturally connects the relevant portions of the Voronoi diagram to the vast cells.

Finally, the visibility–Voronoi diagram can be used to compute high-quality corridors for entities moving amidst polygonal obstacles, as we show in the next chapter. See the discussion in Section 1.2.2 about the importance of corridors for various applications.

We have implemented our algorithm for constructing the $\mathrm{VV}^{(c)}$-diagram for a given clearance value and applied it to the problem of motion planning for coherent groups of entities [KO04a]. The paths contained in the $\mathrm{VV}^{(c)}$-diagram yield convincing group motions, and the approach we propose has several advantages over methods used so far to generate group paths. We note that the robust construction of the $\mathrm{VV}^{(c)}$-diagram involves many non-trivial geometric procedures and requires careful algebraic computations, which we also discuss in this chapter.

[1]Despite the similarity in names, our structure is different from the *visibility complex* introduced by Pocchiola and Vegter [PV96] for efficiently computing the visibility among disjoint convex objects in the plane.

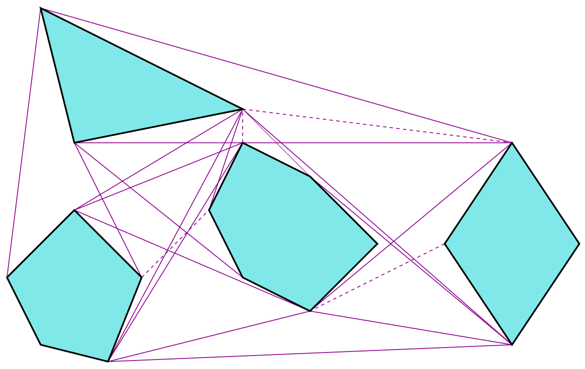


Figure 5.1: The (partial) visibility graph of a set of four convex polygons. The valid visibility edges are drawn with solid lines, while some invalid edges are also shown, drawn with dashed lines. Notice that all obstacle edges are also valid visibility edges.

Chapter Outline

The rest of this chapter is organized as follows: In Section 5.1 we give a short review of the geometric data structures we use for constructing the VV$^{(c)}$-diagram. In Section 5.2 we present the VV$^{(c)}$-diagram in more detail and explain how to construct it, given a scene with obstacles and a preferred clearance value c. In Section 5.3 we introduce the VV-complex, show how to efficiently construct it and explain how to query it. In Section 5.4 we review the software we have developed to robustly compute the VV$^{(c)}$-diagram of a set of obstacles and a given c-value. We conclude with some experimental results in Section 5.5.

5.1 Preliminaries

5.1.1 Visibility Graphs

Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a set of simple pairwise interior-disjoint polygons having n vertices in total. The *visibility graph* of $\mathcal{P}$ is an undirected graph defined on the set of polygon vertices, whose set of edges consists of those pairs of vertices that are mutually visible. Two vertices are *mutually visible* if the straight line segment connecting them does not intersect the *interior* of any of the polygons in $\mathcal{P}$ — in this case, we call this segment a *visibility edge*.

The visibility graph can be used to compute shortest paths amidst configuration-space polygonal obstacles, where the polygons are considered as open sets. Each edge is given a weight equal to the Euclidean distance between its two end-vertices. To find a shortest path between a start and a goal configuration, one simply needs to connect them to the visibility graph and execute Dijkstra's algorithm starting from the vertex representing the start configuration. In fact, it is sufficient to consider only the edges that are bitangent to the polygons they connect, namely edges that can be infinitesimally extended in both directions without penetrating any polygon. Such bitangent edges are called *valid* visibility edges (see Figure 5.1 for an illustration).

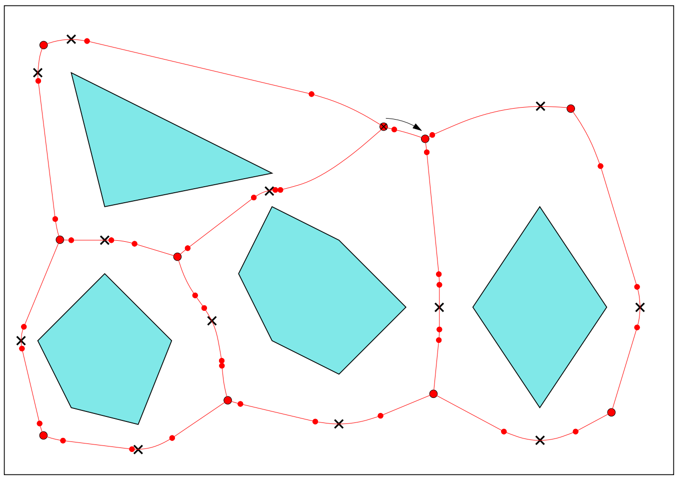


Figure 5.2: The Voronoi diagram of four convex polygons contained inside a rectangle. Small dots mark the endpoints of each Voronoi arc, while the Voronoi vertices are marked by larger dots. The point of minimum clearance along each chain is marked by $\times$. Notice that the chain marked by an arrow is a monotone chain and attains its minimal clearance on its left Voronoi vertex — so when we traverse it in the arrow's direction, the clearance only increases.

The visibility graph can be computed in $O(n^2 \log n)$ time, performing a straightforward radial sweep around each of the polygon vertices (see, e.g., [dBvKOS00, Chapter 15]). Ghosh and Mount [GM91] were the first to give an output-sensitive algorithm for computing the visibility graph in optimal $O(n \log n + k)$ time, where k is the number of visibility edges in the output visibility graph. For more information on shortest paths, see [Mit04].

5.1.2 Voronoi Diagrams of Polygons

Given a set S of geometric entities in $\mathbb{R}^d$ and a distance metric $\| \cdot \|$, the *Voronoi diagram* of S, denoted $\mathrm{Vor}(S)$, is the subdivision of $\mathbb{R}^d$ into maximal connected cells, such that the points in each *Voronoi cell* are closer to a specific entity of S than to all other entities of S.

There are many variants of Voronoi diagrams (see [AK00, For04] for extensive reviews). Here we focus on the Voronoi diagram of a set of pairwise interior-disjoint polygons in $\mathbb{R}^2$ under the Euclidean distance metric, which can be regarded as a special case of a Voronoi diagram of line segments [LD81]. For each point $p \in \mathbb{R}^2$ we consider the polygon feature (a polygon *feature* is either a vertex or an edge) closest to p. The *Voronoi vertices* in this case are points equidistant to closest features of three (or more) different polygons. The vertices are connected by continuous *chains* of *Voronoi arcs*. An arc may be equidistant to two closest vertices or to two closest polygon edges — in which case it is a straight line segment, or to a polygon vertex and a (non-incident) polygon edge — in which case it is a segment of a parabola (parabolic arc). Each arc has two *endpoints*, which either connect it to the next arc in the chain or to a Voronoi vertex.

For any point p in the plane, let the *clearance value* of p be the distance from the point to the closest polygon. If we examine the clearance value along a Voronoi chain, we notice that in most cases the minimum clearance value is attained in the interior of a vertex–vertex or a vertex–edge arc inside the chain (note that the interior of an edge–edge arc will never contain a clearance minimum). In such cases, the clearance value increases as we move from this minimum point toward either of the chain's end-vertices. However, for some chains the minimum clearance value is attained at one of their end-vertices and grows as we move along the chain toward its other end. We call such a chain a *monotone Voronoi chain* (see Figure 5.2 for an illustration).

The Voronoi diagram can be used to compute paths with maximal amounts of clearance from the obstacles. It can be shown that the total complexity of the Voronoi diagram is $O(n)$, where n is the total number of polygon vertices, and that it can be constructed in $O(n \log n)$ time (see, e.g, [AK00, LD81]). For more details on the connection between Voronoi diagrams and motion planning see [ÓSY83, ÓY85, Roh91].

5.2 The VV$^{(c)}$-Diagram

Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a set of simple pairwise interior-disjoint polygons in the plane, having n vertices in total, representing two-dimensional configuration-space obstacles. Let $c > 0$ be the preferred distance we wish to keep from these obstacles. Our goal is to preprocess $\mathcal{P}$, so that given a start configuration s and a goal configuration g, we can efficiently compute a shortest path between s and g, keeping a clearance of at least c from the obstacles where possible, but allowing to get closer to the obstacles in narrow passages when it is possible to make considerable shortcuts.

We begin by dilating each obstacle by c — that is, computing the Minkowski sum of each polygon with a disc of radius c (see Section 3.4 for more details). The visibility graph of the dilated obstacles contains all shortest paths with a clearance of at least c from the obstacles. Note that the dilated polygon edges are also valid visibility edges. Moreover, as each convex polygon vertex becomes a circular arc of radius c, the valid visibility edges are bitangents to two circular arcs. This guarantees that a shortest path extracted from such a visibility graph is $\mathcal{C}_1$-smooth and contains no sharp turns. The only disadvantage in this approach is that narrow, yet collision-free, passages can be blocked when we dilate the obstacles (for example, in Figure 5.3 there exists such a narrow passage between P_1 and P_3). It is clearly not possible to pass through such passages with a clearance of at least c, but we still wish to allow a path with the maximal clearance possible in this region. To do this, we compute the portions of the free configuration space that are contained in at least two dilated obstacles and add their intersection with the Voronoi diagram of the original polygons to our diagram. The resulting structure is called the VV$^{(c)}$-diagram.

Formally, given a collection of disjoint convex obstacles $P_1, \ldots, P_m$ (we will later discuss non-convex obstacles as well) and a preferred clearance value c, we perform the following steps:

1. We construct the Minkowski sum $M_i^{(c)} = P_i \oplus B_c$ for every obstacle P_i, where B_c is a disc with radius c. Note that the inflated obstacles $M_i^{(c)}$ may no longer be disjoint.

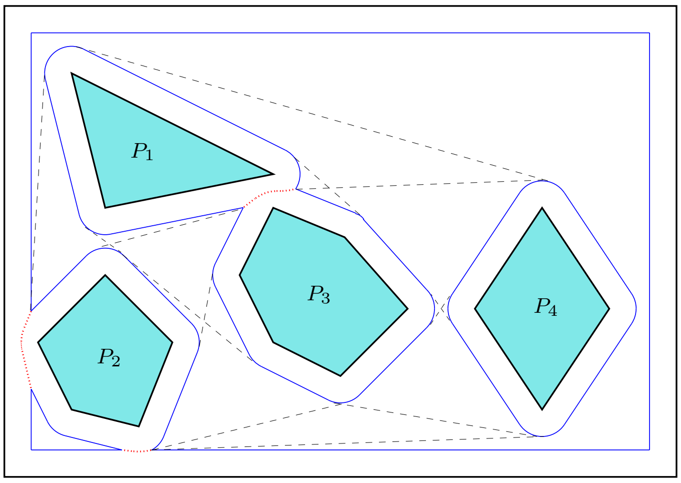


Figure 5.3: The VV$^{(c)}$-diagram for four convex obstacles located in a rectangular room. The boundary of the union of the dilated obstacles is drawn in a solid line, the relevant portion of the Voronoi diagram is dotted. The visibility edges are drawn using a dashed line. Notice that an endpoint of a visibility edge may either lie on a circular arc or on the intersection of two dilated obstacle boundaries (a chain point).

2. We compute the union $\mathcal{M}^{(c)}$ of all $M_i^{(c)}$. The boundary of $\mathcal{M}^{(c)}$ consists of circular arcs and straight line segments. Reflex vertices may appear on the boundary of $\mathcal{M}^{(c)}$, which are the intersection of the boundary arcs of two dilated obstacles. We refer to them as *chain points*, as they lie on Voronoi chains, since their distance from both relevant polygons is exactly c.

3. We compute the modified visibility graph $\mathcal{G}^{(c)}$ of $\mathcal{M}^{(c)}$. This graph consists of every free[2] bitangent of two circular arcs of the boundary of $\mathcal{M}^{(c)}$ (the edges that form the boundary of $\mathcal{M}^{(c)}$ are also regarded as bitangents to two neighboring dilated vertices), every free line segment between two chain points, and every free line segment from a chain point tangent to a circular arc.

4. We construct $\mathcal{V}$, the Voronoi diagram of the original set of polygons and compute the intersection $\mathcal{V} \cap \mathcal{M}^{(c)}$, namely the portion of the Voronoi diagram that is contained within the union of the dilated obstacles. We combine the corresponding Voronoi arcs (and sub-arcs) with $\mathcal{G}^{(c)}$ to connect the chain points via narrow passages and form the final VV$^{(c)}$-diagram.

As mentioned in Section 3.4, step 1 can be carried out in linear time. Step 2 takes $O(n \log^2 n)$ time, or $O(n \log n)$ expected time using a randomized algorithm [Mul90]. In step 4 we construct the Voronoi diagram in $O(n \log n)$ time, while computing the intersection $\mathcal{V} \cap \mathcal{M}^{(c)}$ can be carried out in linear time. Thus, step 3, which can easily be performed in $O(n^2 \log n)$ time, clearly dominates the running time of the VV$^{(c)}$-diagram construction process. We conclude that it takes $O(n^2 \log n)$ time to construct the VV$^{(c)}$-diagram of an input set $\mathcal{P}$ of pairwise interior-disjoint convex polygons for a given c-value, when using a

[2]A line segment is *free* if its interior is not contained in the interior of any dilated obstacle.

straightforward approach. We note that it might also be possible to improve the running time to $O(n \log n + k)$, where k is the number of visibility edges, by constructing the visibility complex of the dilated polygons [PV96].[3]

In case our polygons are not convex, we decompose them to obtain a set of convex polygons and compute $\mathcal{M}^{(c)}$ for this set. Note that in this case not every reflex vertex of $\mathcal{M}^{(c)}$ is now a chain point, since reflex vertices can also be induced by reflex vertices of the original polygons. However, these reflex vertices of $\mathcal{M}^{(c)}$ can be easily identified and are not taken into account in the VV$^{(c)}$-diagram (namely the diagram does not contain visibility edges emanating from these vertices).

Querying the VV$^{(c)}$-Diagram

Having constructed the VV$^{(c)}$-diagram, once we are given a start configuration s and a goal configuration g we just have to connect them to our diagram and compute the shortest path between s and g using Dijkstra's algorithm. It takes $O(n \log n)$ time to connect s and g to the diagram, by performing a radial sweep from each configuration, and connecting s and g to the circular arcs and chain points visible from these configurations. The execution of Dijkstra's algorithm takes $O(n \log n + \ell)$, where ℓ is the number of diagram edges we encounter during the search (which is at most k).

As mentioned before, we may compromise on the amount of clearance our motion path keeps from the obstacles if we can make a shortcut going through a narrow passage. It should be noted that if a path contains a portion of the Voronoi diagram it may not be smooth any more (this is however acceptable, as we consider making sharp turns inside narrow passages to be natural). In order to balance between the length and the clearance of the selected path we have to associate the appropriate weight with each diagram edge, so the Dijkstra algorithm outputs the path which is most suitable for our application. The weight of a visibility edge can simply be equal to its length (the lengths of the circular arcs we traverse must also be taken into consideration), while for Voronoi edges we may add some penalty to the edge length, taking into account their clearance values, which are below the preferred c-value. For example, if the minimal clearance of a Voronoi arc is $c' < c$, we can give it the weight of its length multiplied by $\left(\frac{c}{c'}\right)^{\kappa}$, where $\kappa > 0$ is a parameter controlling the amount of extra weight given to Voronoi arcs.

Another option of weighting the edges, especially suitable for the application of coherent group motion, where the path serves as a backbone to a wider corridor through which the entities flow [KO04b], is to estimate the time it takes the group to traverse each edge. For edges with a clearance of at least $c = \frac{w}{2}$, where w is the preferred group width, this time is clearly proportional to the edge length. On the other hand, for Voronoi edges the actual clearance of the edge would also be taken into account, as the moving entities will have to traverse this edge in a long row. The resulting path will therefore be the one enabling the group to reach its goal as quickly as possible. In Chapter 6 we address the topic of assigning weights to the diagram edges more thoroughly.

[3]The main difficulty here is that we handle *dilated* obstacles, which may *not* be disjoint. Moreover, the obstacles (and of course the dilated obstacle) are not of constant complexity.

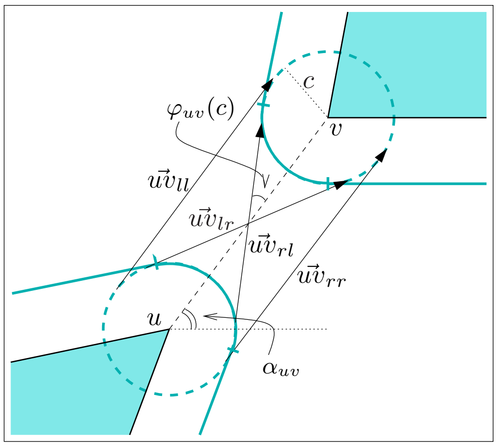


Figure 5.4: The four possible bitangents to the circles $B_c(u)$ and $B_c(v)$ of radius c centered at two obstacle vertices u and v. Notice that in this specific scenario only the bitangent $\vec{uv}_{rl}$ is a valid visibility edge.

5.3 The VV-Complex

The construction of the VV$^{(c)}$-diagram for a given c-value is straightforward, yet it requires some non-trivial geometric and algebraic operations that should be computed in a robust manner — see Section 5.4 for more details. Moreover, if we wish to plan motion paths for different c-values and select the best one (according to some criterion), we must construct the VV$^{(c)}$-diagram for each c-value from scratch. In this section we explain how to efficiently preprocess an input set of polygonal obstacles and construct a data structure called the VV-complex, which can be queried to produce a natural-looking path for every start and goal configuration and for *any* preferred clearance value c.

Let us examine what happens to the VV$^{(c)}$-diagram as c continuously changes from zero to infinity. For simplicity, we consider only convex obstacles in this section. As we mentioned before, VV$^{(0)}$ is the visibility graph of the original obstacles, while VV$^{(\infty)}$ is their Voronoi diagram, so as c grows visibility edges disappear from VV$^{(c)}$ and make way to Voronoi chains. We start with a set of visibility edges containing all pairs of the polygonal obstacle vertices that are mutually visible, regardless whether these edges are bitangents of the obstacles.[4] We also include the original obstacle edges in this set, as they can be viewed as visibility edges between two adjacent polygon vertices. Furthermore, we treat our visibility edges as directed, such that if the vertex u "sees" the vertex v, we will have two directed visibility edges in our structure, $\vec{uv}$ and $\vec{vu}$.

As c grows larger than zero, each of the *original* visibility edges potentially spawns as many as four bitangent visibility edges. These edges are the bitangents to the circles $B_c(u)$ and $B_c(v)$ (where $B_r(p)$ denotes a circle centered at p whose radius is r) that we name $\vec{uv}_{ll}$, $\vec{uv}_{lr}$, $\vec{uv}_{rl}$ and $\vec{uv}_{rr}$, according to the relative position (left or right) of the bitangent with

[4]Visibility edges are only *valid* when they are bitangents, otherwise they do not contribute to shortest paths in the visibility graph. However, as c grows larger the invalid edges may become bitangents, as shown in Figure 5.6(b), so we need them in our data structure.

respect to u and to v (see Figure 5.4).[5] Let α_{uv} be the angle between the vector $\vec{uv}$ and the x-axis, and $d(u,v)$ the Euclidean distance between u and v, then it is easy to see that the two bitangents $\vec{uv}_{ll}$ and $\vec{uv}_{rr}$ retain the same slope α_{uv} for increasing c-values. The slope of the other two bitangents changes as c grows: $\vec{uv}_{rl}$ rotates counterclockwise and $\vec{uv}_{lr}$ rotates clockwise by the same amount, both around the midpoint $\frac{1}{2}(u+v)$ of the original edge, so their slopes become $\alpha_{uv} + \varphi_{uv}(c)$ and $\alpha_{uv} - \varphi_{uv}(c)$, respectively, where $\varphi_{uv}(c) = \arcsin(\frac{2c}{d(u,v)})$. For $c > \frac{1}{2}d(u,v)$ the two edges $\vec{uv}_{rl}$ and $\vec{uv}_{lr}$ disappear.

Note that for a given c-value, it is impossible that all four edges are valid. As we consider only obstacles with non-empty interiors, line segments do not qualify, and therefore each circular arc on a boundary of a dilated obstacle is of angle less than π. We conclude that at most three visibility edges can be valid, and that the edges $\vec{uv}_{ll}$ and $\vec{uv}_{rr}$ can never be valid simultaneously. Our goal is to compute a *validity range* $R(e) = [c_{\min}(e), c_{\max}(e)]$ for each edge e, such that e is part of the VV$^{(c)}$-diagram for each $c \in R(e)$.[6] If an edge is valid, then it must be tangent to both circular arcs associated with its end-vertices. There are several reasons for an edge to change its validity status:

- The tangency point of e to either $B_c(u)$ or to $B_c(v)$ leaves one of the respective circular arcs.

  Note that the circular arc $a_r(u)$ representing the dilation of a convex polygon vertex u by radius r lies on $B_r(u)$ and is defined by the endpoints $(x_u + r\cos\theta_1(u), y_u + r\sin\theta_1(u))$ and $(x_u + r\cos\theta_2(u), y_u + r\sin\theta_2(u))$, where $\theta_1(u)$ and $\theta_2(u)$ are determined by the shape of the polygon. Let us denote the tangency point of the visibility edge e to $a_r(u)$ by $(x_u + r\cos\varphi_e(u,r), y_u + r\sin\varphi_e(u,r))$. We say that the tangency point of e *leaves* the circular arc $a_c(u)$ if $\varphi_e(u,r)$ equals $\theta_1(u)$ or $\theta_2(u)$, and for small enough $\varepsilon > 0$, we have $\varphi_e(u, r-\varepsilon) \in [\theta_1(u), \theta_2(u)]$ and $\varphi_e(u, r+\varepsilon) \notin [\theta_1(u), \theta_2(u)]$.

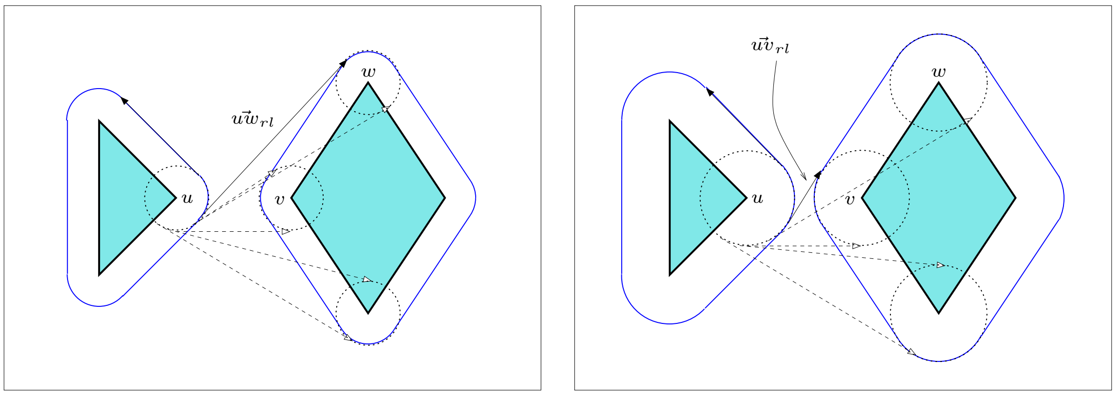
- The tangency point of e to either $B_c(u)$ or to $B_c(v)$ enters one of the respective circular arcs. Similar to the definition above, we say that the tangency point of e *enters* the circular arc $a_c(u)$ if $\varphi_e(u,r)$ equals $\theta_1(u)$ or $\theta_2(u)$, and for small enough $\varepsilon > 0$, we have $\varphi_e(u, r-\varepsilon) \notin [\theta_1(u), \theta_2(u)]$ and $\varphi_e(u, r+\varepsilon) \in [\theta_1(u), \theta_2(u)]$.

- The visibility edge becomes blocked by the interior of a dilated obstacle.

The important observation is that at the moment that a visibility edge $\vec{uv}$ gets blocked, it becomes tangent to another dilated obstacle vertex w, so essentially one of the edges associated with $\vec{uv}$ becomes equally sloped with one of the edges associated with $\vec{uw}$ (see Figure 5.6(a)). The first two cases mentioned above can also be realized as events of the same nature, as they occur when one of the $\vec{uv}$ edges becomes equally sloped with $\vec{uw}_{lr}$ (or $\vec{uw}_{rl}$), when v and w are adjacent vertices in a polygonal obstacle — see Figure 5.6(b).

[5]Recall that edges in the visibility graph are *undirected*, thus our *directed* visibility edges come in pairs. According to our notation, $\vec{uv}_{ll}$ and $\vec{uv}_{rr}$ are equivalent to the opposite edges $\vec{vu}_{rr}$ and $\vec{vu}_{ll}$, respectively, while $\vec{uv}_{lr}$ and $\vec{uv}_{rl}$ are equivalent to $\vec{vu}_{lr}$ and $\vec{vu}_{rl}$, respectively. A pair of opposite edges always become valid or invalid simultaneously.

[6]Liu and Arimoto [LA95] use a similar notion to construct a structure that answers shortest-path queries for disc robots, where the radius of the robot is given in the query. They do not, however, incorporate portions of the Voronoi diagram in their construct.



(a) (b)

Figure 5.5: The circular list $\mathcal{L}_r(u)$ of "right" visibility edges associated with an obstacle vertex u. Valid visibility edges are drawn as solid arrows while invalid edges are drawn as dashed arrows (for clarity, some edges are omitted). Note that in (a) $\vec{uw}_{rl}$ is a valid visibility edge, but as we increase c in (b), it is blocked by $\vec{uv}_{rl}$ (which becomes a valid edge) and removed from the list.

This observation stands at the basis of the algorithm we devise for constructing the VV-complex. We sweep through increasing c-values, namely we start from $c = 0$ and gradually increase c, keeping at each time the VV$^{(c)}$-diagram for that clearance value. The main idea is that we have to consider only a finite number of critical c-values, where the combinatorial structure of the VV$^{(c)}$-diagram changes. As we have just explained, we should stop at c-values associated with *visibility events*, which occur when two edges become equally sloped.[7] We note that the edge $\vec{uv}_{ll}$ (or $\vec{uv}_{lr}$) can only be involved in visibility events with arcs of the form $\vec{uw}_{ll}$ or $\vec{uw}_{lr}$, while the edge $\vec{uv}_{rl}$ (or $\vec{uv}_{rr}$) can only have events with arcs of the form $\vec{uw}_{rl}$ or $\vec{uw}_{rr}$. Hence, we can associate two circular lists $\mathcal{L}_l(u)$ and $\mathcal{L}_r(u)$ of the left and right edges of the vertex u, respectively, both sorted by the slopes of the edges. Two edges participate in an event at some c-value only if they are neighbors in one of these lists for infinitesimally smaller c (see Figure 5.5 for an illustration). At these event points, we should update the validity range of the edges involved and also update the adjacencies in their appropriate lists, resulting in new events.

As mentioned in Section 5.2, an endpoint of a visibility edge in the VV$^{(c)}$-diagram may also be a chain point, so we must consider chain points in our algorithm as well. As a Voronoi chain is either monotone or has a single point with minimal clearance, we need to associate at most two chain points with every Voronoi chain. These chain points move towards the end-vertices of the Voronoi edge as we increase c. Our algorithm will also have to compute the validity ranges of edges connecting a chain point with a dilated vertex or with another chain point. For that purpose, we will have a list $\mathcal{L}(p)$ of the outgoing edges of each chain point p, sorted by their slopes (notice that we do not have to separate the "left" edges from the "right" edges in this case).

[7]Our visibility events are reminiscent of the *merge events* and *split events* that occur in the algorithm for drawing "fat" planar edges, as suggested by Duncan *et al.* [DEKW01].

In the next subsection we review the algorithmic details of the preprocessing stage for constructing the VV-complex, and describe how to query this data structure in Section 5.3.2. We continue the presentation of the algorithm by a proof of correctness in Section 5.3.3 and a complexity analysis in Section 5.3.4. We finally explain how the algorithm can be generalized for non-convex polygons in Section 5.3.5.

5.3.1 The Preprocessing Stage

Initialization

Given an input set $P_1, \ldots, P_m$ of convex pairwise interior-disjoint polygonal obstacles, we start by computing their visibility graph and classifying the visibility edges as valid (bitangent) or invalid. We examine each bitangent visibility edge uv: For an infinitesimally small c only one of the four $\vec{uv}$ edges it spawns is valid — we assign 0 to be the minimal value of the validity range of this edge (and of the opposite $\vec{vu}$ edge).

As our algorithm is event-driven, we initialize an empty event queue $\mathcal{Q}$, storing events by increasing c-order.

We proceed by constructing the circular lists $\mathcal{L}_l(u)$ and $\mathcal{L}_r(u)$ for each obstacle vertex u, based on the visibility edges we have just computed. We examine each pair of adjacent edges e_1, e_2 in $\mathcal{L}_l(u)$ (and in $\mathcal{L}_r(u)$), compute the c-value at which e_1 and e_2 become equally sloped — if one exists — and insert the *visibility event* $\langle c, e_1, e_2 \rangle$ into the event queue. In a visibility event some edges become blocked and reach the end of their validity range, while some new edges may become valid.

As our VV-complex also contains Voronoi chains, we have to compute the Voronoi diagram of the polygonal obstacles. For each *non-monotone* Voronoi chain we locate the arc a that contains the minimal clearance value $c_{\min}$ of the chain in its interior and insert the *chain event* $\langle c_{\min}, a \rangle$ into $\mathcal{Q}$. A chain event occurs when a Voronoi chain starts contributing to the VV$^{(c)}$-diagram, namely when we sweep through its minimal clearance value. For now, we do not need to worry about monotone chains — in the next section we explain how we can correctly handle them without associating chain events with them.

Event Handling

While the event queue is not empty, we proceed by extracting the event in the front of $\mathcal{Q}$, associated with a minimal c-value, and handle it according to its type.

Visibility event: Visibility events always come in pairs — that is, if $\vec{uv}$ becomes equally sloped with $\vec{uw}$,[8] we will either have an event for the opposite edges $\vec{vu}$ and $\vec{vw}$, or for the opposite edges $\vec{wu}$ and $\vec{wv}$. We therefore handle a pair of visibility events as a single event. Let us assume that the edges $\vec{uv}$ and $\vec{uw}$ become equally sloped for a

[8] In the rest of this section, we use the notation $\vec{uv}$ to represent any of the four edges $\vec{uv}_{ll}$, $\vec{uv}_{lr}$, $\vec{uv}_{rl}$ or $\vec{uv}_{rr}$. We also use $\mathcal{L}(u)$ to denote either $\mathcal{L}_l(u)$ or $\mathcal{L}_r(u)$ (whether we choose the "left" or the "right" list depends on the type of edge involved).

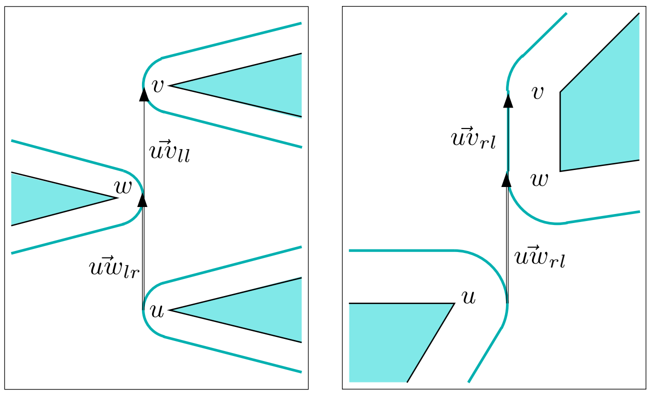


Figure 5.6: Visibility events involving u, v and w: (a) The dilated vertex w blocks the visibility of u and v. (b) As $\vec{uw}_{rl}$ becomes equally sloped with $\vec{uv}_{rl}$ (where vw is an obstacle edge), it becomes a valid visibility edge.

clearance value c', and at the same time the edges $\vec{vu}$ and $\vec{vw}$ become equally sloped (see Figure 5.6).

As the edges $\vec{uv}$ and $\vec{vu}$ now become blocked, we assign c' to be the maximal c-value of the validity range of these edges. We also remove the other event, if any, involving $\vec{uv}$ (based on its other adjacency in $\mathcal{L}(u)$) from $\mathcal{Q}$, and delete this edge from $\mathcal{L}(u)$. We examine the new adjacency created in $\mathcal{L}(u)$ and insert the corresponding visibility event into the event queue $\mathcal{Q}$. We repeat this procedure for the opposite edge $\vec{vu}$.

If the edge $\vec{uv}$ was valid before it was deleted and the edge $\vec{uw}$ (or $\vec{vw}$) does not have a minimal validity value yet, we assign c' to it, because this edge has become bitangent for this c-value (see Figure 5.6(b) for an illustration).

Chain event: The value c equals the minimal clearance of a Voronoi chain χ_a, attained on the arc a, which is equidistant from an obstacle vertex u and another obstacle feature (see Figure 5.7(b)).[9] Let z_1 and z_2 be a's endpoints. We initiate two chain points $p_1(\chi_a)$ and $p_2(\chi_a)$ associated with the Voronoi chain χ_a. As c grows, $p_1(\chi_a)$ moves toward z_1 and $p_2(\chi_a)$ moves toward z_2 (see Figure 5.7(c) for an illustration).

As we increase c, larger portions of χ_a will enter the VV$^{(c)}$-diagram and visibility edges will become incident to its chain points, rather than to dilated vertices. We therefore have to examine all edges e incident to u, compute the minimum c-value c' for which e becomes incident to one of the chain points $p_i(\chi_a)$, and insert the *tangency event* $\langle c', e, p_i(\chi_a)\rangle$ into the event queue. If a is equidistant from u and from another obstacle vertex v (i.e., a is a vertex–vertex Voronoi arc), we do the same for the edges incident to v.

Finally, we create two *endpoint events*, $\langle c_1, p_1(\chi_a), z_1\rangle$ and $\langle c_2, p_2(\chi_a), z_2\rangle$, associated with the clearance values c_1 and c_2 attained at z_1 and z_2, respectively.

[9]Recall that a Voronoi arc equidistant to two polygon edges is always monotone with respect to the clearance and can never contain a chain minimum in its interior.

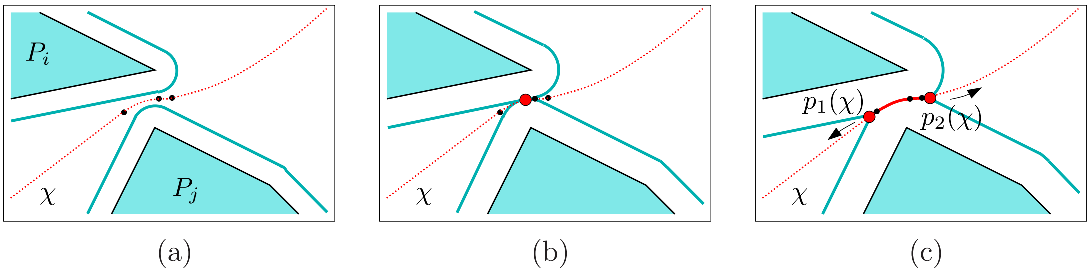


Figure 5.7: A chain event associated with the Voronoi chain χ (dotted) induced by the two obstacles P_i and P_j. The endpoints of the arcs forming χ are drawn as small black dots. (a) The clearance value c is less than the minimal clearance of the chain χ, so this chain does not contribute to the VV$^{(c)}$-diagram. (b) c equals the minimal clearance of the chain χ and a chain event occurs. Note that the two dilated obstacles now begin to intersect. (c) When c grows the two chain points $p_1(\chi)$ and $p_2(\chi)$, that define the portion of χ lying inside the VV$^{(c)}$-diagram (drawn in a solid line) move along the arcs of the chain χ toward its end-vertices (not shown in this figure).

When dealing with a chain event, we introduced two additional types of events, used to handle chain points: tangency events and endpoint events. For a small enough positive c value (smaller than the clearance value of any point on the Voronoi diagram) the endpoints of all visibility edges lie on dilated obstacle vertices, but as c grows these endpoints gradually become chain points. A *tangency event* occurs when a visibility edge becomes incident to a chain point. The *endpoint events* are used to transfer the chain points along Voronoi chains. We next explain how we deal with these events.

Tangency event: A visibility edge $e = \vec{ux}$ (the endpoint x may either represent a dilated vertex or a chain point) becomes tangent to $B_{c'}(u)$ at a chain point $p(\chi_a)$ associated with the Voronoi arc a (see Figure 5.8 for an illustration for the case when $x = v$ is a dilated obstacle vertex). In this case we have to replace e by the visibility edge $\overrightarrow{p(\chi_a)x}$ associated with the chain point $p(\chi_a)$. We assign c' to be the maximal validity value of the edge e, and remove it from $\mathcal{L}(u)$. We now insert a *reincarnate* of e to $\mathcal{L}(p(\chi_a))$, and assign c' as its minimal validity value. We examine the new adjacency in $\mathcal{L}(p(\chi_a))$ and insert, if necessary, a new visibility event into $\mathcal{Q}$.[10] Finally, we replace the edge $\vec{xu}$ in $\mathcal{L}(x)$ by $\overrightarrow{xp(\chi_a)}$, recompute the critical c-values of the visibility events of this edge with its neighbors (notice that the slope of $\overrightarrow{xp(a)}$ becomes a different function of c from now on) and modify the corresponding visibility events in $\mathcal{Q}$.

In case x is a dilated obstacle vertex, we may have another tangency event in the queue, associated with $\vec{xu}$, which was computed under the (false) assumption that the tangency point of the edge on x coincides with a chain point before the one on u does. In this case, we have to locate the tangency event in $\mathcal{Q}$ that is associated with $\vec{xu}$ and recompute the c-value associated with it.

[10]For a given c-value, let $\vec{\omega}$ be the direction of the tangent to the Voronoi chain χ_a, such that when we infinitesimally increase c, the chain point $p(\chi_a)$ moves in this direction. Note that even though $\mathcal{L}(p(\chi_a))$ is represented as a circular list, the vector -$\vec{\omega}$ naturally splits it into a linear list. We note that a tangency event always results in the insertion of a new edge at one of the list ends, so only one true adjacency is created.

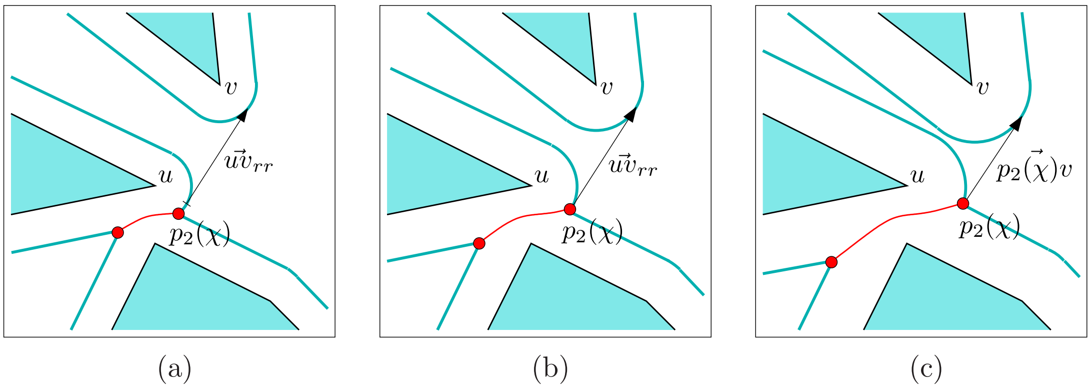


Figure 5.8: A tangency event: (a) The chain point $p_2(\chi)$, whose creation is depicted in Figure 5.7, lies on the supporting circle of the dilated vertex u. (b) The visibility edge $\vec{uv}_{rr}$ becomes tangent to $B_c(u)$ exactly at $p_2(\chi)$, so a tangency event occurs. (c) The reincarnated visibility edge $\overrightarrow{p_2(\chi)v}$ replaces $\vec{uv}_{rr}$ as c grows. Note that this edge is not tangent to $B_c(u)$ any more.

Endpoint event: A chain point $p(\chi_a)$ reaches the endpoint z of the Voronoi arc a. We should consider the following cases here:

- The endpoint z is incident only to two Voronoi arcs a and a' belonging to the same chain (i.e., $\chi_a = \chi_{a'}$). In this case the chain point $p(\chi_a)$ is transferred from a to a', and we only have to examine the adjacencies in $\mathcal{L}(p(\chi_{a'}))$ and modify the corresponding visibility events in the queue (as the slopes of these arcs become a different function of c from now on). We also have to handle the opposite edges, as we did in the tangency-event procedure. Moreover, if there are tangency events associated with the opposite edges we should modify them as well.

  As the chain point $p(\chi_a)$ now moves on the Voronoi arc a', we have to take care of tangency events that occur in the range of this new arc. Thus, if one of the polygon features associated with a' is a vertex u, we iterate over all edges incident to u and check whether each edge has a tangency event in the range of the new Voronoi arc a' — if so, we insert the appropriate tangency event into the event queue.[11] In case a' is a vertex–vertex arc, associated with two vertices u and v, we repeat this procedure for v as well.

- If z is a Voronoi vertex *and* a local maximum of the clearance function, there are multiple endpoint events associated with it. In non-degenerate cases, the edge lists of all chain points coinciding with z are already empty. Only in degenerate cases may chain points involved in an endpoint-event at z still have incident edges, and in this case we just assign a maximal validity value to these edges and empty the edge lists associated with these chain points.

- Otherwise, z is the endpoint of the chain χ_a (i.e., a Voronoi vertex) and it is *not* a local maximum of the clearance function. In this case we may have several chains $\chi_1, \chi_2, \ldots$ ending at z, having a simultaneous endpoint event, and a single

[11]Note that edges that had a tangency event in the range of the previous Voronoi arc a have already been deleted from the incident-edge list of the vertex at the moment this endpoint event occurs.

monotone chain $\hat{\chi}$ beginning at z (see for example the left Voronoi vertex of the marked chain in Figure 5.2). We therefore create a new chain point $p(\hat{\chi})$ associated with the monotone chain, assign a maximal validity value c' to each edge in $\mathcal{L}(p(\chi_1)), \mathcal{L}(p(\chi_2)), \ldots$, where c' is the clearance value at z. We remove all visibility events associated with these edges from $\mathcal{Q}$ and insert their reincarnates into $\mathcal{L}(p(\hat{\chi}))$. We examine all adjacencies in $\mathcal{L}(p(\hat{\chi}))$ and add the appropriate visibility events into $\mathcal{Q}$. We also have to deal with the opposite edges and modify any tangency events they are involved in.

We note that in order to avoid duplicate work, when we have several events occurring at the same c-value, we deal with endpoint events first, to make sure that edges are associated with the correct chain. We can then handle the visibility events, chain events and finally the tangency events.

5.3.2 Querying the VV-Complex

The result of the preprocessing stage is the VV-complex $\langle \mathcal{V}, \mathcal{T} \rangle$, where:

- $\mathcal{V}$ is the Voronoi diagram of the polygonal obstacles. We also store the clearance value $c(z)$ of each vertex z in the Voronoi diagram, and for each non-monotone chain χ we store its minimal clearance value $c_{\min}(\chi)$.
- $\mathcal{T}$ is a set of interval trees: For each obstacle vertex u, $T_u \in \mathcal{T}$ contains, for each edge incident to u, its validity range (namely the intervals are the valid c-ranges of the edges incident to u). For each Voronoi chain χ, $T_{\chi,i} \in \mathcal{T}$ is an interval tree storing edges and Voronoi arcs incident to the ith chain point ($i \in \{1, 2\}$) of the chain χ, along with their validity ranges.

A query on the VV-complex is defined by a triple $\langle s, g, \hat{c} \rangle$, where s and g are the start and goal configurations, respectively, and $\hat{c}$ is the preferred clearance value. We assume that s and g themselves have a clearance larger than $\hat{c}$ (one could apply standard techniques for testing whether s and g have sufficient clearance). Given a query, we start by computing the relevant portion of the Voronoi diagram: For each Voronoi chain we can examine the clearance values of its end-vertices, as well as the chain minimum, and determine which portion of the chain (if at all) we should consider. This way we also obtain all the chain points for the given c-value $\hat{c}$.

Next we need to find the incident edges of s and g. This means that we should obtain two lists $\mathcal{L}(s)$ and $\mathcal{L}(g)$ containing the visibility edges emanating from s and g (respectively) to every visible circular arc and chain point (or to original obstacle vertices if $c = 0$). This can be done using a radial sweep-line algorithm. We can now start searching the implicitly constructed VV$^{(\hat{c})}$-diagram using a Dijkstra-like search to find the "shortest" path between s and g.

When we reach a vertex x (a dilated polygon vertex or a chain point) during the Dijkstra search we query T_x with the given c-value $\hat{c}$ to obtain the valid edges incident to x, as we do not have an explicit representation of the graph. In addition, we add g to the list of x's

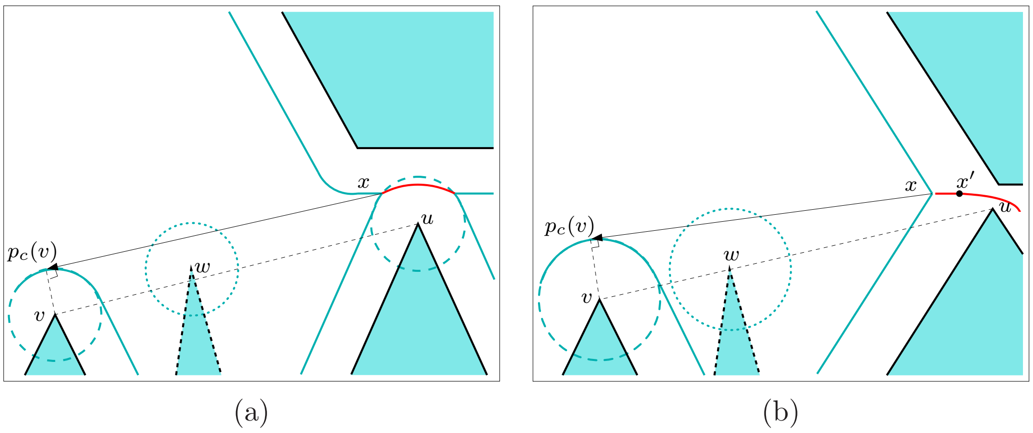


Figure 5.9: Visible dilated vertices from a chain point x. (a) If a vertex v is visible from a vertex–edge Voronoi arc, it must be also visible from the vertex u inducing this arc, and cannot be blocked by the dashed polygon. (b) In case of an edge–edge Voronoi arc, we consider the vertex u, which lies closest to the arc endpoint with the minimum clearance x', as the "inducing vertex".

neighbors if $x \in \mathcal{L}(g)$ (that is, if the goal is visible from x). If x is an obstacle vertex, we should keep in mind to add the length of the portion of the corresponding circular arc to the distance.[12] We proceed until the goal configuration g is reached.

The way we select the weights associated with the graph edges may depend on the path-planning strategy we employ. All visibility edges (and portions of the circular arcs which need to be traversed) have a clearance of at least $\hat{c}$, so their distance measure depends only on their length. For the portions of the Voronoi diagram, the limited amount of clearance may add extra weight (see the discussion in Section 5.2 about the weight we give the graph edges). Since the graph edges are implicitly represented, we have to dynamically compute their associated weights, but this can be done in $O(1)$ time per edge and does not incur a significant computational load.

5.3.3 Proof of Correctness

We begin by stating a lemma that asserts that the manner in which we move visibility events from dilated vertices to chain points when handling tangency events is indeed correct — i.e., that a chain point cannot start "seeing" an object (a dilated vertex or another chain point) all of a sudden, unless this object is visible from one of the vertices inducing the Voronoi arcs along the chain.

Lemma 5.1 *If a dilated obstacle vertex $B_c(v)$ is visible from a chain point on a Voronoi arc, then the original vertex v is visible from the vertices inducing this arc. In case of an*

[12] In some cases we will have fictitious visibility edges of length 0, for example when we have a chain point y that lies on a vertex–vertex or a vertex–edge Voronoi edge (see Figure 5.8(a) for an illustration). In this case, y is connected to the polygon vertices that induce this Voronoi edge with visibility edges of distance 0, and when we examine a path through the relevant Voronoi edge and involving a visibility edge incident to one of the vertices inducing y, we should only consider the length of the circular arcs between y and the endpoint of the visibility edge.

edge–edge Voronoi arc, we consider the arc endpoint with the minimum clearance value, and refer to the obstacle vertex that lies closest to this point as the "inducing vertex".

Proof: Consider the example depicted in Figure 5.9(a), where the dilated vertex $B_c(v)$ is visible from the chain point x, which lies on a vertex–edge Voronoi arc. Let u be the obstacle vertex inducing this arc. Assume u and v are not mutually visible, then there must exist some polygon blocking the straight line segment uv — let w be an extreme vertex of this polygon. Let $p_c(v)$ be the tangency point of the visibility edge emanating from x toward $B_c(v)$. It is clear that the distance of v from the line supporting $(x, p_c(v))$ is exactly c, but the distance of u from this line is *less* than c, as it cannot be tangent to $B_c(u)$ and it penetrates the interior of this circle. We conclude that the distance of w from this line must also be less than c, thus $B_c(w)$ blocks the visibility of $B_c(v)$ from the chain point x. We have reached a contradiction, so we conclude that the original vertices u and v are mutually visible.

The same arguments hold for a chain point located on a vertex–vertex Voronoi arc, and we conclude that v is visible from *both* vertices inducing the arc. The case of a chain point which lies on an edge–edge Voronoi arc is depicted in Figure 5.9(b). Note that in this case the two dilated polygon edges incident to x define the portion of the plane it can "see". Once again, assume w blocks the visibility edge of u and v (recall that u is the vertex inducing the Voronoi arc). Since the distance of u from the supporting line of $(x, p_c(v))$ must be less than c (notice that also in this case this line intersects the interior of $B_c(u)$), the distance of w from this line is also less than c. Again, we have reached a contradiction, as $B_c(w)$ blocks the segment $(x, p_c(v))$. □

Theorem 5.2 *Every visibility edge has only one continuous range $[c_{\min}, c_{\max}]$ of c-values for which it is valid. Thus, once it has been deleted it will not become valid again for a higher c-value.*

Proof: Consider Figure 5.10, which describes the schematic "life-cycle" of a visibility edge along the preprocessing step described in Section 5.3.1. When we construct the VV-complex by gradually increasing the c-value, edges can only be deleted when a visibility event occurs and they become blocked by some dilated vertex: It is clear that just before a dilated vertex w starts blocking the visibility of x and y (x and y may be dilated vertices or chain points), it must lie on the line segment connecting x and y, so a visibility event must occur and no visibility edge can "disappear" as c grows without being involved in a visibility event. Note that an edge can also reincarnate as a different edge (see Figure 5.8), but in this case we can treat the validity range of its reincarnate as a direct continuation of the range of the original edge.[13] Here we show that once an edge becomes blocked, it does not become unblocked again for a higher c-value.

[13]When presenting the algorithm we created a new validity range for reincarnated visibility edges instead of treating the validity ranges as a single continuum, as we do in this theorem. This representation simplifies the algorithm without incurring any asymptotic run-time penalty. Our theorem is therefore slightly stronger than what we need for proving the correctness of our algorithm.

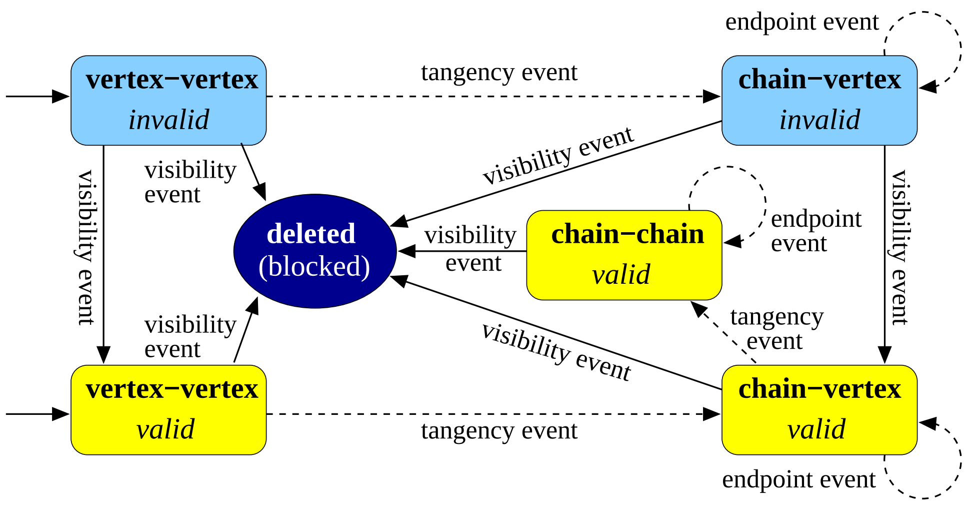


Figure 5.10: The schematic "life-cycle" of a visibility edge during the execution of the preprocessing stage. The rounded-corner rectangles denote possible visibility edges by the type of their endpoints. The solid arrows denote a change in the validity of the edge while the dashed arrows denote a reincarnation of the edge. For $c = 0$, all visibility edges, valid or invalid, are incident to two vertices (represented by the two rectangles on the left). As c grows and parts of the Voronoi diagram are included in the VV$^{(c)}$-diagram, an endpoint of such an edge may become incident to a Voronoi chain — namely a tangency event occurs and the edge is reincarnated as a vertex–chain edge (and later on as a chain–chain edge). Such edges are affected by endpoint events that occur along the Voronoi chain, but their validity status remains unchanged. Visibility events can turn invalid edge to valid ones, or block visibility edges. In the latter case, the blocked edge is deleted.

Consider a visibility edge $\vec{uv}$ (it may either be invalid or valid) tangent to the supporting circles of the dilated vertices u and v for some clearance value $c_1 > 0$. Let $\zeta_1(u)$ and $\zeta_1(v)$ be the two endpoints of this edge, lying on $B_{c_1}(u)$ and $B_{c_1}(v)$, respectively. As illustrated in Figure 5.11, for a clearance value $c_2 > c_1$, the edge $(\zeta_2(u), \zeta_2(v))$ between u and v for clearance c_2 is contained in the Minkowski sum $(\zeta_1(u), \zeta_1(v)) \oplus B_{c_2-c_1}$, as the distance of both $\zeta_2(u)$ and $\zeta_2(v)$ from the line segment $(\zeta_1(u), \zeta_1(v))$ is clearly less than $c_2 - c_1$.

Let us assume that for the clearance value c_1 the visibility edge $\vec{uv}$ becomes blocked by a dilated obstacle vertex w, which touches $(\zeta_1(u), \zeta_1(v))$ at some point q — then for each $c_2 > c_1$ the disc $B_{c_2-c_1}(q)$ of radius $c_2 - c_1$ centered at q is fully contained in a dilated obstacle, and no visibility edges can cross it: note that this disc subdivides the region $(\zeta_1(u), \zeta_1(v)) \oplus B_{c_2-c_1}$ into two, making it impossible for the edge $(\zeta_2(u), \zeta_2(v))$ to be valid.

It is therefore clear that once a visibility edge between two dilated vertices becomes blocked, it can never become unblocked again.[14] Moreover, similar arguments apply if one of the endpoints of the visibility edge (or both its endpoints) is a chain point lying on a Voronoi arc. We begin by showing that the chain point for the clearance value c_2 lies inside

[14]In this case, there is also a simple algebraic proof for this fact: The bitangent to $B_{c_1}(u)$ and $B_{c_1}(v)$ is also tangent to $B_{c_1}(w)$ only when c_1 equals half the distance between u and the line connecting v and w. For the edge to become unblocked at some $c_2 > c_1$, the three circles $B_{c_2}(u)$, $B_{c_2}(v)$ and $B_{c_2}(w)$ must have another common tangent, but this is of course impossible.

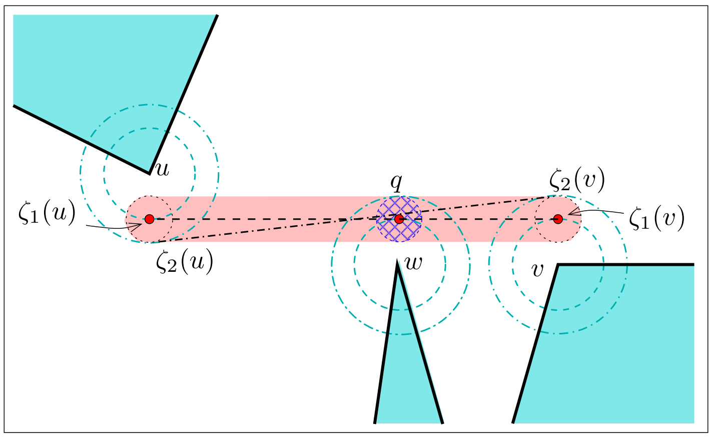


Figure 5.11: The visibility edges $\vec{uv}_{rl}$ and $\vec{vu}_{rl}$, realized as the segment $(\zeta_1(u), \zeta_1(v))$ (the dashed black line), are blocked at q by the dilated vertex $B_{c_1}(w)$. For $c_2 > c_1$, $(\zeta_2(u), \zeta_2(v))$ (the dash-dotted line segment) is contained in the region $(\zeta_1(u), \zeta_1(v)) \oplus B_{c_2-c_1}$ (lightly shaded), which is divided into two by the disc $B_{c_2-c_1}(q)$.

the cigar-shaped region obtained by taking the Minkowski sum of the original visibility edge with $B_{c_2-c_1}$:

- The endpoint ζ_1 of a visibility edge for a clearance value c_1 lies on a vertex–vertex Voronoi arc (see Figure 5.12(a) for an illustration). Without loss of generality, let us assume that the two vertices u and v inducing this Voronoi arc are located at $(0, -\delta)$ and $(0, \delta)$, where $2\delta < c_1$ is the distance between the vertices. In this case the Voronoi arc is supported by the line $y = 0$ and the two chain points for c_i $(i = 1, 2)$ are given by $\zeta_i = (\sqrt{c_i^2 - \delta^2}, 0)$.

  Let us consider the extremal case where the visibility edge is tangent to $B_{c_1}(0, \delta)$ — that is, it is tangent to one of the dilated obstacles and if its slope is increased by $\varepsilon > 0$ it will penetrate this dilated obstacle and become blocked. In this case, the lower part of the "cigar" intersects $y = 0$ at $\tilde{\zeta}$, where:

$$x_{\tilde{\zeta}} = x_{\zeta_1} + \frac{c_2 - c_1}{\sin\theta} = \sqrt{c_1^2 - \delta^2} + \frac{c_1(c_2 - c_1)}{\sqrt{c_1^2 - \delta^2}} = \frac{c_1 c_2 - \delta^2}{\sqrt{c_1^2 - \delta^2}}$$

  It is straightforward to show that $x_{\tilde{\zeta}} > x_{\zeta_2}$, hence ζ_2 is contained in the "cigar".

- The endpoint ζ_1 lies on a vertex–edge Voronoi arc. Without loss of generality, we assume that the obstacle edge inducing the arc is supported by the line $y = \delta$ and the obstacle vertex is given by $(0, -\delta)$ (again, we have $2\delta < c_1$). It is clear that the slope of a visibility edge emanating from ζ_1 is non-positive. In the extremal case, depicted in Figure 5.12(b), it is a horizontal segment, and since $|y_{\zeta_2} - y_{\zeta_1}| = c_2 - c_1$ then ζ_2 is located on the boundary of the cigar-shaped region around the horizontal visibility edge. It is also clear that in other cases, when the slope of the original visibility is negative, then ζ_2 is located in the interior of the "cigar" around this edge.

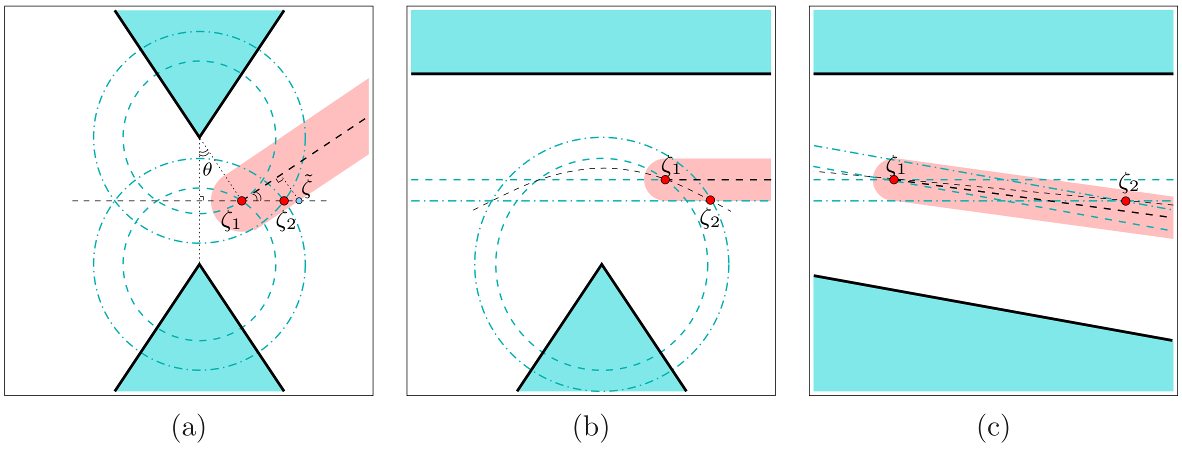


Figure 5.12: The chain points ζ_1 and ζ_2, at clearance values c_1 and c_2, respectively ($c_2 > c_1$). The relevant Voronoi arcs are drawn as thin dashed lines, where the light dashed (dash-dotted) segments and circles correspond to clearance c_1 (c_2, respectively) from the obstacle features inducing these arcs. The visibility edges emanating from ζ_1 are drawn in a thick dashed line, with the Minkowski sum of the edge with $B_{c_2-c_1}$ is lightly shaded. (a) An extreme case where the visibility edge from a chain point lying on a vertex–vertex arc is tangent to one of the dilated obstacles. (b) Another extreme case where the visibility edge from a chain point lying on a vertex–edge arc is parallel to the edge. (c) The case of chain points lying on an edge–edge arc.

- The same arguments also apply if ζ_1 and ζ_2 lie on an edge–edge Voronoi arc. Note that in this case we should consider the slopes of both obstacle edge involved: indeed, $\|\zeta_1 - \zeta_2\|$ may be significantly larger that $c_2 - c_1$, as shown in Figure 5.12(c), but since the slope of the visibility edge is bounded by the slope of the obstacle edges, it follows that ζ_2 must be contained in the "cigar".

We have showed that a visibility edge $\bar{e}_2$ for c_2 is always contained in the cigar-shaped region, which is the Minkowski sum of the visibility edge $\bar{e}_1$ for $c_1 < c_2$ with $B_{c_2-c_1}$. According to our assumption, $\bar{e}_1$ is blocked by some point q, so $\bar{e}_1 \oplus B_{c_2-c_1}$ is divided into two by the disc $B_{c_2-c_1}(q)$. We argue that each part contains exactly one endpoint of $\bar{e}_2$, which can be easily verified by examining the various cases in Figure 5.12. If this is not the case, then q must lie between ζ_1 and the projection of ζ_2 onto $\bar{e}_1$ — this is of course impossible, as it implies that there exists another obstacle on the way, other than the ones defining the Voronoi arc. As a consequence, the visibility edge $\bar{e}_2$ must also be blocked.

We conclude that once a visibility edge has been blocked, it will never become valid again. Note that what we have shown so far is that we can associate a single validity range with a visibility edge one of whose endpoints lie on a Voronoi *arc*, while our edges are actually associated with chain points that move along Voronoi *chains*. However, when a chain point is created, there are no visibility edges associated with it. By Lemma 5.1, visibility edges can be associated with a chain point only when it is involved in tangency events, as it traverses a vertex–vertex or a vertex–edge Voronoi arc, and it cannot "see" any object not visible from the relevant vertex. As the chain point moves along the chain, these visibility edges are eventually blocked (the chain point can never move from an edge–edge arc to another edge–edge arc, as there should always be a vertex on the way). We conclude that the association of a single validity range with each visibility edge (and with its reincarnates) is indeed correct. □

5.3.4 Complexity Analysis

Theorem 5.3 *Constructing the* VV*-complex takes* $O(n^2 \log n)$ *in total, where* n *is the total number of obstacle vertices.*

Proof: In the initialization of the preprocessing stage we first have to compute the visibility graph, which can be performed in $O(n^2 \log n)$ time — this also accounts for the time needed to construct the initial edge lists $\mathcal{L}(u)$ for each obstacle vertex u (we need $O(n \log n)$ time to construct each of the $2n$ edge lists) and label the valid visibility edges. The construction of the Voronoi diagram can be performed in $O(n \log n)$, and the complexity of the diagram (the number of arcs) is linear in n.

After the initialization, the priority queue $\mathcal{Q}$ contains $O(1)$ events per visibility edge, of which there are $O(n^2)$ in total, and in addition $O(n)$ chain events. Any operation on the event queue thus takes $O(\log n)$. The initialization takes $O(n^2 \log n)$ time in total.

As the preprocessing algorithm proceeds, it starts handling events: In total, Theorem 5.2 implies that we have $O(n^2)$ visibility events:[15] Every vertex can be involved at most once in a visibility event with another vertex, where a visibility edge between the two vertices (or their dilated version) is created. Each of the visibility events can be handled in $O(\log n)$ time as it involves a constant number of operations on the queue and on the edge lists. There are $O(n)$ chain events, each of them can be handled in $O(n \log n)$ time. Each chain event spawns $O(n)$ tangency events, so in total there are $O(n^2)$ tangency events, each of them can be handled in $O(\log n)$ time. Finally, there are $O(n)$ endpoint events, and we need $O(n \log n)$ time to handle each of these events.[16] □

The query phase starts with a stage that takes $O(n \log n)$ time, which is spent on calculating the valid visibility edges emanating from s and g. Calculating the relevant portions of the Voronoi diagram takes $O(n)$ time (note that the Voronoi diagram itself has already been constructed in the preprocessing phase).

The rest of the query phase consists of executing Dijkstra's algorithm, or an equally suited A*-algorithm. The worst-case running-time of these algorithms is $O(n \log n + \ell)$ where $\ell = O(k)$ is the number of edges encountered during the search (recall that k is the number of visibility edges). In practice, Dijkstra's algorithm turns out to be very fast, because hardly any geometric operations have to be performed anymore. In particular the

[15] We consider all potential events in our analysis. In practice, some of these events were computed under false assumptions (see Section 5.3.1) and will be eventually discarded.

[16] It is in fact possible to construct the visibility graph of the input polygons in $O(n \log n + k)$ time, where k is the number of visibility edges in this graph (valid and invalid ones), construct the initial edge lists in $O(k \log n)$ time and then charge each of the $O(k)$ directed visibility edges with $O(\log n)$ operations, to account for all visibility events, chain events and tangency events. Unfortunately, the entire preprocessing stage cannot be completed in $O(k \log n)$ time even if $k = o(n^2)$, as there are cases where $\Theta(n^2 \log n)$ operations are needed to handle the endpoint events.

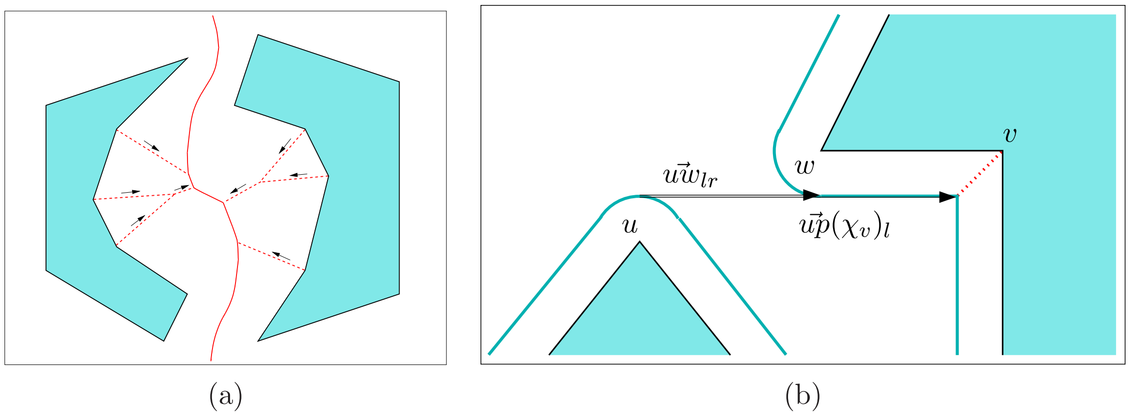


Figure 5.13: (a) A portion of the Voronoi diagram of two non-convex polygons. The Voronoi chain separating the two obstacles is drawn with a solid line, while the Voronoi chains induced by features of the same polygon are drawn with a dashed line. (b) The edge $\vec{uw}_{lr}$ becomes valid after being involved in a visibility event with a visibility edge to the chain point $p(\chi_v)$ that is associated with the reflex obstacle vertex v.

A*-variant of Dijkstra may be the method of choice here, as it biases the search toward the goal configuration, which keeps the number ℓ low.

As we noted in Section 5.2, the $\mathrm{VV}^{(c)}$-diagram for a fixed c-value may be constructed in $O(n \log n + k)$ time, so it may seem we do not need any preprocessing stage, and it is better to construct the $\mathrm{VV}^{(c)}$-diagram from scratch whenever we are given a preferred clearance value. However, this algorithm involves the construction of the planar arrangement of line segments, circular arcs and parabolic arcs, which is very complicated when carried out in a *robust* manner (see the next section). Such an approach will require longer running times than the query stage of the second algorithm. We note that Dijkstra's algorithm, whose running time theoretically dominates the query phase, is in practice very fast if after preprocessing our set of input obstacles in an exact manner, we switch to machine-precision floating-point arithmetic in the query stage.[17]

5.3.5 Handling Non-Convex Obstacles

So far we described the algorithm for constructing a VV-complex for a set of convex polygonal obstacles. Our algorithm can however be easily adapted to work with non-convex obstacles as well. The only thing that is changed is the way in which the Voronoi diagram is constructed.

Due to the non-convexity of the obstacles, some obstacles may contain reflex vertices. These reflex vertices are treated as normal vertices in the initial construction (for $c = 0$) of the visibility graph. Note that the visibility edges emanating from reflex vertices will never be part of a shortest path, but we still need to keep track of these edges, as they may induce visibility events that give other valid edges the correct c-values of their validity ranges (see Figure 5.13(b) for an illustration).

[17]Indeed, we lose some accuracy here, but as our constructed diagram is topologically correct, the worst thing that can happen is that we may compute a path that is only slightly longer than the shortest possible path.

As c grows, the reflex vertices will be treated as chain points. These chain points move over monotone Voronoi chains originating in the reflex vertices themselves (see Figure 5.13(a)). To this end, the definition of the Voronoi diagram should be adapted such that Voronoi arcs can be equidistant to two edges of the same polygon as well. Still, this new Voronoi diagram is an instance of the Voronoi diagram of line segments, so this change is easily carried through.

The rest of the algorithm remains unchanged. Also, the complexity analysis is still valid, since the construction time and the complexity of both the visibility graph and the Voronoi diagram are not affected by the non-convexity of the input obstacles. We should mention that when we query the VV-complex we do not compute the chain points along Voronoi chains induced by reflex vertices, and therefore do not account for these "reflex" chains, as these chains lead to a dead-end (a reflex vertex) and can never be used for making shortcuts in the motion path.

5.4 Implementation Details

Cgal offers the infrastructure we need for developing a robust software for computing the VV$^{(c)}$-diagram. The Minkowski-sum package described in Chapter 3 is able to compute the offset of the polygonal obstacles by radius c and obtain the dilated obstacles. We then use the package described in Section 2.6.1 to compute the union of the dilated obstacles.

The Voronoi diagram of the polygons is computed using the Cgal package for computing Voronoi diagrams of line segments, developed by Karavelas [Kar04]. We have to preprocess the input of the package by adding a label to each segment (polygon edge) and each segment endpoint (polygon vertex) that identifies the source polygon and the feature index within this polygon. Let us assume that all obstacles are convex; even in this simple case, the complete Voronoi diagram of the polygon edges contains also the *medial axis* of each polygon, which is redundant in our case. However, having computed the diagram, we can then conveniently disregard Voronoi chains induced by features of the same polygon by examining the labels of the sites that induce arcs along these chains.

In case the obstacles are not convex, we have to keep some of the Voronoi edges induced by distant features of the same polygon. To this end, we subdivide the obstacles into convex sub-polygons [Her06]. In this case we label each polygon feature with both the index of the convex sub-polygon and the input (non-convex) polygonal obstacle from which it originated. This labeling helps us to determine which Voronoi arcs should be discarded.

The intersection among the dilated obstacles and between the boundary of the union of the dilated obstacles and the Voronoi arcs is robustly computed using the conic-arc traits of the arrangement package (see Section 2.3.3). We exploit the fact that our polygonal obstacles are given as sequences of points with *rational* coordinates, so that the supporting curves of each dilated obstacle boundary and each Voronoi arc can be represented as algebraic curves of degree 2 with rational coefficients if the squared clearance value is also rational (see below), to robustly maintain the arrangement of such curves. The endpoints of the line segments, the circular arcs and the parabolic arcs that form our arrangement are in general algebraic numbers of degree 4.

In the rest of this section we give a constructive proof of a lemma that enables us to robustly construct the skeleton of the VV$^{(c)}$-diagram for rational inputs, based on robust computations with the conic-arc arrangement traits:

Lemma 5.4 *Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a set of pairwise interior-disjoint simple polygons, such that all polygon vertices have* rational *coordinates. Then all Voronoi arcs have supporting algebraic curves of degree 2 at most with rational coefficients and all chain minima are also points with rational coordinates. Moreover, for a clearance value c such that c^2 is rational, the dilated obstacle boundaries are also supported by algebraic curves of degree 2 with rational coefficients.*

Proof: The case of offsetting a rational polygon by a radius r, such that r^2 is rational, was discussed in Section 3.4. The dilated obstacles vertices correspond to arcs of rational circles, and the dilated edges can be expressed as segments of line-pairs with rational coefficients.

Let us now examine the representation of the Voronoi arcs. An arc a of the Voronoi diagram corresponds to the locus of all points equidistant from two polygon features, and the following cases are possible:

Vertex–vertex arc: The arc is equidistant from two polygon vertices u and v. The equation of its supporting curve, a line in this case, is simply given by (throughout this section we use the squared distance, in order to avoid the square-root operation):

$$\begin{aligned}(x - x_u)^2 + (y - y_u)^2 &= (x - x_v)^2 + (y - y_v)^2 \\ 2(x_v - x_u)x + 2(y_v - y_u)y &= x_v^2 + y_v^2 - (x_u^2 + y_u^2) \ . \end{aligned} \tag{5.1}$$

This line is perpendicular to the line segment connecting u an v and bisects it. The point with minimal clearance on the arc is therefore the midpoint between u and v, $z_{\min} = \frac{1}{2}(x_u + x_v, y_u + y_v)$, and its clearance is of course $c_{\min} = \frac{1}{2}d(u, v)$.

Vertex–edge arc: The arc is equidistant from a polygon vertex u and a polygon edge vw, whose supporting line will be denoted $\ell : \ Ax + By + C = 0$, where A, B and C are rational (since the vertices have rational coordinates). The equation of its supporting curve, a parabola in this case, is thus given by:

$$\frac{(Ax + By + C)^2}{A^2 + B^2} = (x - x_u)^2 + (y - y_u)^2 \ . \tag{5.2}$$

In this case, to find the point with minimal clearance on the arc we compute a line perpendicular to ℓ that passes through u. The equation of this line is $\ell^{\perp} : \ By - Ax + (Ay_u - Bx_u) = 0$, and the point with minimal clearance is the midpoint between u and the intersection point of ℓ and $\ell^{\perp}$:

$$z_{\min} = \frac{1}{2}\left(x_u + \frac{B^2 x_u - A(By_u + C)}{A^2 + B^2}, y_u + \frac{A^2 y_u - B(Ax_u + C)}{A^2 + B^2}\right) \ . \tag{5.3}$$

The minimal clearance value, attained at $z_{\min}$ is half the distance between u and the line ℓ.

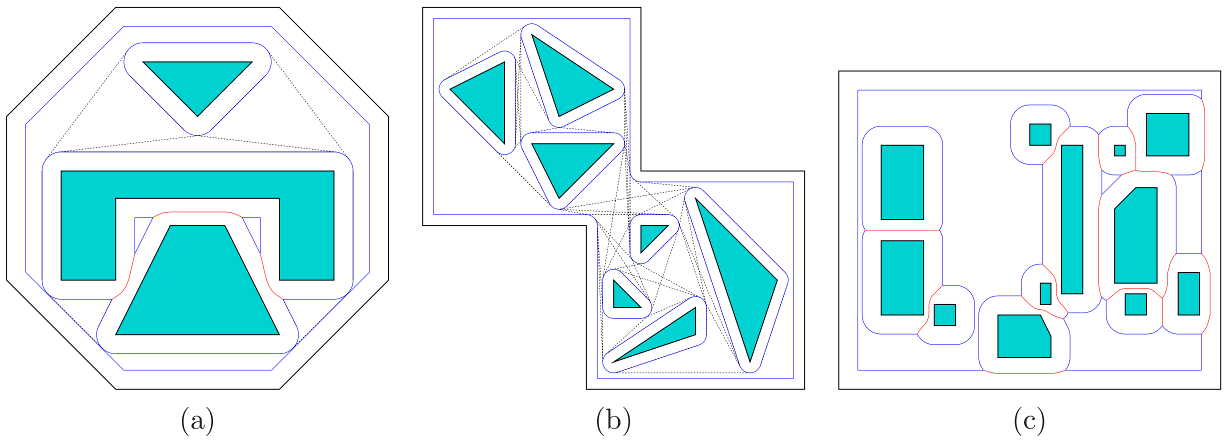


Figure 5.14: The $\mathrm{VV}^{(c)}$-diagrams constructed for several input files and c-values: (a) *octagon* with $c = \frac{7}{10}$, (b) *two_rooms* with $c = \frac{2}{5}$, and (c) *rectangles* with $c = \frac{9}{10}$ (visibility edges are not shown in this case).

Edge–edge arc: The arc is equidistant from two polygon edges, whose supporting lines are denoted $\ell_1 : \ A_1x + B_1y + C_1 = 0$ and $\ell_2 : \ A_2x + B_2y + C_2 = 0$, respectively. The supporting curve of this edge is a line bisecting the angle formed between ℓ_1 and ℓ_2, but in general this line cannot be represented as a linear curve with rational coefficients.[18] Instead, we represent the edge as a segment of a pair of perpendicular lines (naturally, only one line in this pair supports the relevant segment), which form the two angle bisectors of ℓ_1 and ℓ_2:

$$\frac{(A_1x + B_1y + C_1)^2}{A_1^2 + B_1^2} = \frac{(A_2x + B_2y + C_2)^2}{A_2^2 + B_2^2} \ . \tag{5.4}$$

Using this representation, it is possible to represent the Voronoi arc as a segment of a curve of degree 2 with rational coefficients. As we mentioned before, such an arc is always monotone — that is, as we traverse it from the endpoint with smaller clearance value to the other endpoint, we get further away from the obstacles.

$\square$

5.5 Experimental Results

Our software is implemented using CGAL 3.2, relying on the exact number types supplied by CORE 1.7 that can handle algebraic numbers in a certified way (see also Section 2.3.3). As we wish to obtain an exact representation of the $\mathrm{VV}^{(c)}$-diagram, we may spend some time on the diagram construction, especially if it contains chain points, which are algebraically more difficult to handle. For example, the construction of the $\mathrm{VV}^{(c)}$-diagram depicted in

[18]For example, if $\ell_1 : y = 0$ and $\ell_2 : \ y = x$, the slope of the line bisecting the angle between ℓ_1 and ℓ_2 is $\tan\frac{\pi}{8} = \frac{1}{1+\sqrt{2}}$, and this line ($y = \frac{1}{1+\sqrt{2}}x$) cannot be represented using rational coefficients. Note however that the perpendicular line $y = \frac{1}{1-\sqrt{2}x}$ is also an angle bisector in this case, and the bisector curve therefore consists of a *pair* of perpendicular lines.

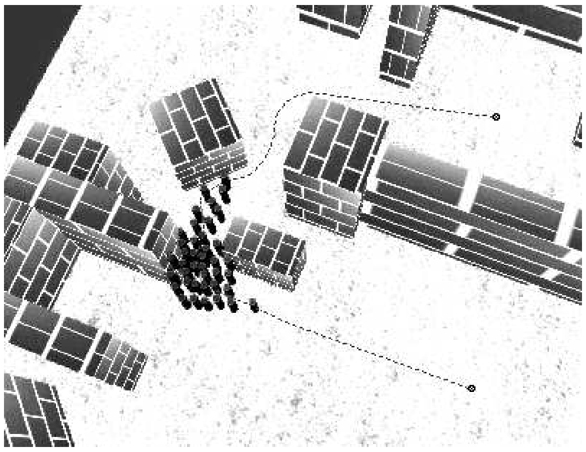


Figure 5.15: A group of 40 entities moving in a virtual scene along a backbone path, drawn with a dashed line. (Courtesy of Arno Kamphuis.)

Figure 5.3 (the *four_shapes* scene) takes about 10 seconds (running on a Pentium IV 2 GHz machine with 512 MB of RAM), but if we choose a smaller clearance value for the same scene, such that no chain points appear in the diagram, the construction time drops to 2.3 seconds (see Table 5.1). In more involved scenes, the construction of the diagram may take 15–20 seconds (see Figure 5.14 and Table 5.1).

However, once the VV$^{(c)}$-diagram is constructed, it is possible to use a floating-point approximation of the edge lengths to speed up the time needed for answering motion-planning queries, so that the average query time is only a few milliseconds.

Table 5.1: The construction time of the VV$^{(c)}$-diagram for several input scenes and different c-values. The query times were averaged over five manually entered queries for each scene.

Input	Bounding-box dimensions	c	Construction time (sec.)	Average query time (sec.)
four_shapes	10×7	$^1/_5$	2.3	0.01
four_shapes	10×7	$^2/_5$	9.7	0.01
octagon	14×14	$^3/_{10}$	4.9	0.01
octagon	14×14	$^7/_{10}$	15.2	0.01
two_rooms	14×14	$^2/_5$	2.8	0.02
rectangles	18×15	$^9/_{10}$	15.4	0.02

We also used the VV$^{(c)}$-diagram to generate convincing group motions in a more complex scene, as the one depicted in Figure 5.15. The construction of such diagrams takes about 40–60 seconds (for clearance values that induce chain points), but the average query time was only a few milliseconds. This is a considerable improvement over previous techniques, which require smoothing operations in the query stage, taking about one second on average.

◇◇
◇

We introduced a simple, yet powerful, data structure — the VV$^{(c)}$-diagram — which contains all shortest paths for a robot in a planar environment of configuration-space obstacles, given a preferred clearance value and that allows for a trade-off between path length and clearance in the presence of narrow passages. We have implemented a robust software package that maintains this data structure and used it to plan natural-looking paths for coherent groups of moving entities in the plane. Our method, which requires some preprocessing for constructing the diagram but can answer queries very efficiently without the need for smoothing or additional post-processing, is especially suitable to real-time applications, such as computer games.

We have also introduced the VV-complex, a data structure that efficiently encodes all VV$^{(c)}$-diagrams for all possible clearance values. We showed how to efficiently construct the VV-complex for a given set of obstacles and how to query it given start and goal configurations and a preferred clearance value.

Chapter 6

Planning Near-Optimal Corridors amidst Planar Obstacles

Planning corridors among obstacles has arisen as a central problem in game design, but can also be used for other motion-planning applications. Instead of devising a one-dimensional motion path for a moving entity, it is possible to let it move in a corridor, where the exact motion path is determined by a local planner. A high-quality path should be short, but at the same time it should be sufficiently wide to allow for maneuvering inside it. As already discussed in the previous chapter, these two requirement often contradict.

In this chapter we introduce a measure for the quality of such corridors that aims to balance between these two desirable properties of the corridor. We analyze the structure of optimal corridors amidst point obstacles and polygonal obstacles in the plane, and propose methods to plan corridors that are (nearly) optimal, with respect to this measure, amidst point obstacles or polygonal obstacles in the plane.

The rest of this chapter is organized as follows. In Section 6.1, we formally define corridors and introduce the quality measure. Section 6.2 discusses properties of optimal corridors amidst point obstacles in the plane, and in Section 6.3 we generalize our results to the case of polygonal obstacles. As mentioned in the previous chapter, it is often preferred that the path of the moving entity is not too winding. In Section 6.4 we therefore also take the curvature of the corridor into account and augment the quality measure accordingly.

6.1 Measuring Corridors

A *corridor* $C = \langle \gamma(t), w(t), w_{\max} \rangle$ in a d-dimensional workspace (typically $d = 2$ or $d = 3$) is defined as the union of a set of d-dimensional balls whose center points lie along the *backbone path* of the corridor, which is given by the continuous function $\gamma : [0, L] \longrightarrow \mathbb{R}^d$. The radii of the balls along the backbone path are given by the function $w : [0, L] \longrightarrow (0, w_{\max}]$. Both γ and w are parameterized by the length of the backbone path. In the following, we will refer to $w(t)$ as the *width* of the corridor at point t. The width is positive at any point along the corridor, and does not exceed $w_{\max}$, a prescribed *desired width* of the corridor.

Given a corridor $C = \langle \gamma(t), w(t), w_{\max} \rangle$ of length L in $\mathbb{R}^d$, the interior of the corridor is thus defined by:

$$\bigcup_{t \in [0,L]} B_{w(t)}\left(\gamma(t)\right) \ ,$$

where $B_r(p)$ is an open d-dimensional ball with radius r that is centered at p. In typical motion-planning applications we are given a set of obstacles $\mathcal{O}$ that the moving entities should avoid. The interior of the corridor should be disjoint from the given obstacles, otherwise it is an *invalid* corridor. In the rest of this chapter we consider only *valid* corridors.

6.1.1 The Weighted Length Measure

As we have already indicated, a good corridor must be short — namely its backbone path should avoid unnecessarily long detours — and its width should be as wide as some predefined maximum in order to allow maximal flexibility for the motion within the corridor. The corridor should contain narrow passages only if they allow considerable shortcuts.

Note that if we examine the intersection of the corridor $C = \langle \gamma(t), w(t), w_{\max} \rangle$ with a $(d-1)$-dimensional hyperplane through $\gamma(t)$ whose normal is tangent to γ at $\gamma(t)$, the volume of the cut is proportional to $w^{d-1}(t)$. Thus, in order to combine the two desired properties of the corridor as discussed above, we define the *weighted length* $L^*(C)$ of a corridor $C = \langle \gamma(t), w(t), w_{\max} \rangle$ to be:

$$L^*(C) = \int_{\gamma} \left(\frac{w_{\max}}{w(t)} \right)^{d-1} dt \ . \tag{6.1}$$

We wish to minimize the weighted length by either shortening the backbone path or by extending the corridor's width (up to $w_{\max}$). Given a start position $s \in \mathbb{R}^d$ and a goal position $g \in \mathbb{R}^d$, a corridor $C = \langle \gamma(t), w(t), w_{\max} \rangle$ satisfying $\gamma(0) = s$ and $\gamma(L) = g$ is *optimal* if for any other valid corridor C' connecting the two endpoints we have $L^*(C) \leq L^*(C')$.

We note that unlike the algorithms and methods discussed in previous chapters, where obtaining an exact solution to the problem at hand played an important role, here it is not important to compute *exactly* the optimal corridor for a given scene. Obtaining a good approximation to the optimal corridor is enough for the motion-planning applications we consider. However, in order to devise a good approximation strategy, one has to have a good idea of the structure of the entity one wishes to approximate. We therefore proceed and study the properties of optimal corridors more rigorously.

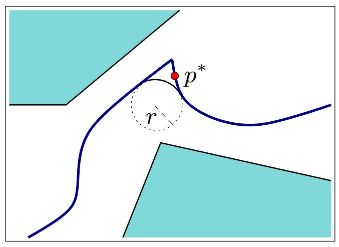
6.1.2 Properties of an Optimal Corridor

Observation 6.1 *If for some portion of the backbone path γ of a corridor C, we have $w(t) < \min\{c(\gamma(t)), w_{\max}\}$ for $t \in [t_0, t_0 + \tau]$ $(\tau > 0)$, where $c(p)$ is the* clearance *of the point p, namely its distance to the nearest obstacle, we can improve the quality of the corridor by letting $w(t) \longleftarrow \min\{c(\gamma(t)), w_{\max}\}$ for each $t \in [t_0, t_0 + \tau]$.*

Given a set of obstacles and a $w_{\max}$ value, we can associate the *bounded clearance* measure $\hat{c}(p)$ with each point $p \in \mathbb{R}^d$, where $\hat{c}(p) = \min\{c(p), w_{\max}\}$. Using the observation above, it is clear that the width function of an optimal path $C = \langle \gamma(t), w(t), w_{\max} \rangle$ is simply $w(t) = \hat{c}(\gamma(t))$. From now on we can therefore concentrate on computing an optimal backbone path, given its two endpoints. Note that $\hat{c}(p)$, hence the width function of an optimal corridor, is a continuous function. Moreover, for any $p_1, p_2 \in \mathbb{R}^d$ we have $|\hat{c}(p_2) - \hat{c}(p_1)| \leq \|p_2 - p_1\|$, hence the width function also satisfies the Lipschitz condition for $K = 1$.

Our weighting scheme can be directly applied to extracting backbone paths from PRMs that contain cycles, as suggested in [NKMO04, NO04b]. However, instead of weighting each edge in the PRM by its Euclidean length and extracting the shortest path from the graph, we can consider some preferred width value, and give each edge e the weight of $L^*(e)$, with respect to the bounded clearance measure of the edge. We can thus extract the optimal corridor the PRM contains with respect to $w_{\max}$. If the PRM is adequately sampled, this corridor can serve as an approximation for the optimal corridor. However, for some sets of obstacles we can actually devise a complete scheme for calculating an optimal corridor, as we show in the next section. We conclude this section with two lemmas, which will be used in the subsequent analysis and point out interesting properties of optimal corridors.

Lemma 6.2 *Given a set of obstacles and $w_{\max}$, the backbone path of the optimal corridor connecting any given start position s and to any goal position g is* smooth*.*



Proof: We have already observed that the width function of the optimal corridor connecting s and g is the bounded clearance function of the backbone path. Thus $w(t)$ is a continuous function and satisfies the Lipschitz condition. Assume that γ contains a sharp turn (a $\mathcal{C}_1$-discontinuity). Let us approximate the sharp turn using a circular arc of radius r that smoothly connects to the original path. As illustrated in the figure on the right, as r approaches 0 the approximation is tighter. Let ℓ_1 be the length of the original path segment we approximate and let ℓ_2 be the length of the circular arc. It is easy to show that there exists $\hat{r} > 0$ and some constants $A_1 > A_2 > 0$ such that for each $0 < r < \hat{r}$ we have $\ell_1 \geq A_1 r$ and $\ell_2 = A_2 r$. If the maximal width w^* along the original path segment is obtained at some point p^*, then as the distance of any point p along the circular arc from p^* is bounded by Kr, where K is some constant, and as the width function is 1-Lipschitz, we can write $w^* - w(p) < Kr$. Let L_1^* be the weighted length of the original path segment and let L_2^* be the weighted length of the circular arc. We therefore know that:

$$L_1^* \geq \frac{w_{\max}}{w^*} l_1 \quad , \qquad L_2^* \leq \frac{w_{\max}}{w^* - Kr} l_2 \ ,$$

so we can write:

$$\frac{L_1^*}{L_2^*} \geq \frac{\frac{w_{\max}}{w^*} l_1}{\frac{w_{\max}}{w^* - Kr} l_2} \geq \frac{w^* - Kr}{w^*} \cdot \frac{A_1}{A_2} \ .$$

As $A_1 > A_2$, we can choose $0 < r < \min\left\{\frac{w^*}{K}\left(1 - \frac{A_2}{A_1}\right), \hat{r}\right\}$ such that the entire expression above is greater than 1. We thus have $L_1^* > L_2^*$, and we managed to decrease the weighted

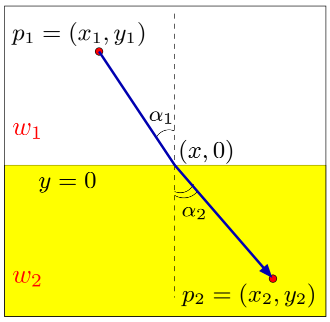


Figure 6.1: Refraction of the optimal backbone in case of a discontinuity in the width function.

length of the corridor, in contradiction to its optimality. We conclude that $\gamma(t)$ must be a smooth function. □

At several places in this chapter we apply infinitesimal analysis, where we assume that the bounded clearance measure (hence the width function) is not continuous. Assume that we have some hyperplane $\mathcal{H}$ in $\mathbb{R}^d$ that separates two regions, such that in one region the bounded clearance is w_1 and in the second it is w_2. Minimizing the weighted length between two endpoints that are separated by $\mathcal{H}$ is equivalent to applying Fermat's principle, stating that the actual path between two points taken by a beam of light is the one which is traversed in the least time. The optimal backbone thus crosses the separating hyperplane once, such that the angles α_1 and α_2 it forms with the normal to $\mathcal{H}$ obey Snell's Law of refraction,[1] with w_1 and w_2 playing the role of the "speed of light" in the respective regions. We next bring the proof for the two-dimensional case:

Lemma 6.3 (Snell's Law of Refraction) *Consider the example depicted in Figure 6.1, where we have two regions separated by the line $y = 0$, such that if our path is given by $\gamma(t) = (x(t), y(t))$, then $w(t) = w_1$ for $y(t) \geq 0$ and $w(t) = w_2$ for $y(t) < 0$. The angles α_1 and α_2 that the backbone of the optimal corridor between $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, where $y_1 > 0$ and $y_2 < 0$, forms with a vertical line perpendicular to $y = 0$ satisfy:*

$$w_2 \sin \alpha_1 = w_1 \sin \alpha_2 \ . \tag{6.2}$$

Proof: It is clear that while the width of the corridor remains constant when moving inside either one of the regions, the optimal backbone path in each region is a line segment. Thus, we have to find a cross point $(x, 0)$ minimizing the weighted length, as defined by Equation (6.1), which is in our case given by the following function of x:

$$L^*(x) = \frac{w_{\max}}{w_1}\sqrt{(x - x_1)^2 + y_1^2} + \frac{w_{\max}}{w_2}\sqrt{(x_2 - x)^2 + y_2^2} \ . \tag{6.3}$$

We therefore derive the weighted length function and obtain:

$$L^{*\prime}(x) = \frac{w_{\max}}{w_1} \cdot \frac{2(x - x_1)}{2\sqrt{(x - x_1)^2 + y_1^2}} - \frac{w_{\max}}{w_2} \cdot \frac{2(x_2 - x)}{2\sqrt{(x_2 - x)^2 + y_2^2}} \ . \tag{6.4}$$

[1]See also Mitchell and Papadimitriou [MP91], who used this observation in a similar setting of the problem.

To find the minimal weighted length we need an x value such that $L^{*'}(x) = 0$, thus we get:

$$\begin{aligned}\frac{1}{w_1} \cdot \frac{x - x_1}{\sqrt{(x-x_1)^2 + y_1^2}} &= \frac{1}{w_2} \cdot \frac{x_2 - x}{\sqrt{(x_2-x)^2 + y_2^2}} , \\ w_2 \cdot \frac{\frac{x-x_1}{y_1}}{\sqrt{\left(\frac{x-x_1}{y_1}\right)^2 + 1}} &= w_1 \cdot \frac{\frac{x_2-x}{y_2}}{\sqrt{\left(\frac{x_2-x}{y_2}\right)^2 + 1}} .\end{aligned}$$

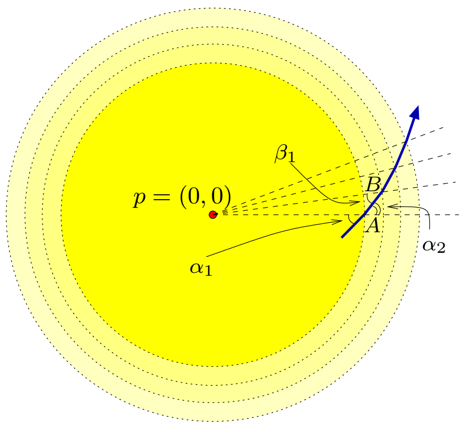
However, $\tan \alpha_1 = \frac{x-x_1}{y_1}$ and $\tan \alpha_2 = \frac{x_2-x}{y_2}$. As for any angle φ we can write $\frac{\tan\varphi}{\sqrt{\tan^2\varphi+1}} = \tan\varphi \cdot \cos\varphi = \sin\varphi$, we obtain that $w_2 \sin\alpha_1 = w_1 \sin\alpha_2$. □

6.2 Optimal Corridors amidst Point Obstacles

In this section we consider planar environments cluttered with point obstacles $p_1, \ldots, p_n \in \mathbb{R}^2$. Given two endpoints $s, g \in \mathbb{R}^2$ and a preferred corridor width $w_{\max}$, we show how to compute a (near-)optimal corridor that connects s and g.

6.2.1 A Single Point Obstacle

Let us assume we have a single point obstacle p. Without loss of generality we assume p is located at the origin. We start with computing an optimal corridor between two endpoints whose distance from p is smaller than or equal to $w_{\max}$. Note that the width of such a corridor at $\gamma(t)$ along its backbone is $c(\gamma(t)) = \|\gamma(t)\|$.



We first approximate the optimal backbone by a polyline: for some small $\Delta r > 0$, if we look at the circles of radii $\Delta r, 2\Delta r, 3\Delta r, \ldots$ that are centered at the origin. Each pair of neighboring circles define an annulus. Since Δr is small we assume that the distance from p of all points in the kth annulus is constant and equals $k\Delta r$. Consider the scenario depicted to the right, where γ enters one of the annuli at some point A, where $\|A\| = r_1$, and leaves this annulus at B, where $\|B\| = r_2 = r_1 + \Delta r$. The angles that the backbone path forms with pA and pB are α_1 and β_1, respectively. When entering the annulus we have $w_1 = r_1$ and $w_2 = r_2$, so applying Equation (6.2) we can express the refracted angle α_2, using:

$$\sin\alpha_2 = \frac{r_2}{r_1} \sin\alpha_1 .$$

By applying the Law of Sines on the triangle $\triangle pAB$, we get:

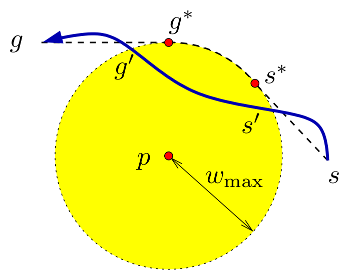
$$\begin{aligned}\frac{r_2}{\sin(\pi-\alpha_2)} &= \frac{r_1}{\sin\beta_1}\ ,\\ \sin\beta_1 &= \frac{r_1}{r_2}\sin(\pi-\alpha_2)=\frac{r_1}{r_2}\sin\alpha_2=\sin\alpha_1\ .\end{aligned}$$

As both angles are less than $\frac{\pi}{2}$, we conclude that $\beta_1=\alpha_1$. Taking $\Delta r \longrightarrow 0$, we obtain a smooth curve γ, such that the angle that $\nabla\gamma(t)$ forms with $\overrightarrow{p\gamma(t)}$ is a constant ψ. It is possible to show that a curve that has this property must be a segment of a *logarithmic spiral* (also named an *equiangular spiral*)[2] whose polar equation is given by $r(t)=ae^{b\theta(t)}$, where a is a constant and $b=\cot\psi$. See, e.g., [Gra97] for a proof of this latter fact.

Proposition 6.4 *Given a single point obstacle located at the origin, a start position $s = r_s e^{i\theta_s}$ and a goal position $g = r_g e^{i\theta_g}$ (in polar coordinates), where $r_s, r_g \le w_{\max}$, the backbone of the optimal corridor connecting s and g is a spiral arc supported by a logarithmic spiral $r = a^* e^{b^*\theta}$. Since both s and g lie on this spiral, we have (assuming $\theta_s \ne \theta_g$, otherwise the optimal backbone path is simply a line segment):*

$$a^* = {r_g}^{\frac{\theta_s}{\theta_s-\theta_g}} \cdot {r_s}^{-\frac{\theta_g}{\theta_s-\theta_g}}\ , \tag{6.5}$$

$$b^* = \frac{1}{\theta_g-\theta_s}\cdot \ln\frac{r_g}{r_s}\ . \tag{6.6}$$



We now consider the case where the clearance of the two endpoints exceeds $w_{\max}$, namely the two endpoints of our path lie outside the closure of the disc $B_{w_{\max}}(p)$. There are two possible scenarios: (i) The straight line segment sg does not intersect $B_{w_{\max}}(p)$; in this case, this segment is the backbone of the optimal corridor. (ii) sg intersects $B_{w_{\max}}(p)$. In this latter case the optimal backbone path is a bit more involved. Consider some backbone path γ connecting s and g. It is clear that the intersection of γ with $B_{w_{\max}}(p)$ comprises a single component (otherwise we have a segment of the backbone path lying outside the circle, which we can shortcut be traversing the circular arc that connects its endpoints), so we denote the point where the path enters the disc by s' and the point where it leaves the disc by g' (see the illustration to the right). As s' and g' lie on the disc boundary, their polar representation is $s' = w_{\max}e^{i\theta_{s'}}$ and $g' = w_{\max}e^{i\theta_{g'}}$, so we use Equations (6.5) and (6.6) and obtain that $a^* = w_{\max}$ and $b^* = 0$. The optimal path between s' and g' therefore lies on the degenerate spiral $r = w_{\max}$, namely the circle that forms the boundary of $B_{w_{\max}}(p)$. We conclude that the optimal backbone path between s and g must contain a circular arc on the boundary of $B_{w_{\max}}(p)$. As according to Lemma 6.2 this path must be smooth, it should comprise two line segments ss^* and g^*g that are tangent to the disc and a circular arc that connects the two tangency points s^* and g^* (see the dashed path in the figure above). Note that as there are two possible tangents emanating from each endpoint, we should consider the four possible paths and select the shortest one.

[2] See, e.g., ⟨http://www-groups.dcs.st-and.ac.uk/∼history/Curves/Equiangular.html⟩ for more details.

6.2.2 Multiple Well-Separated Point Obstacles

Let us now go back to our original setting, where we are given a set of n point obstacles $\mathcal{O} = \{p_1, \ldots, p_n\}$, along with a preferred width $w_{\max}$, and wish to compute the optimal corridor from s to g, where we assume that $c(s) = \min_i \|s - p_i\| \geq w_{\max}$ and $c(g) = \min_i \|g - p_i\| \geq w_{\max}$.

In case the points are well separated — that is, for each $i \neq j$ the discs $B_{w_{\max}}(p_i)$ and $B_{w_{\max}}(p_j)$ are disjoint in their interiors (implying that $\|p_i - p_j\| \geq 2w_{\max}$ for each $1 \leq i < j \leq n$), we can follow the same arguments we used above for a single obstacle and conclude that the optimal backbone is either the straight line segment sg (in case it is *free*, namely its interior does not intersect the interior of any of the discs), or it comprises circular arcs and line segments that connect them.

We can therefore construct the visibility graph of the dilated obstacles (namely, the obstacles offset by radius $w_{\max}$) and use it to construct optimal paths. The vertices of this graph are the endpoints of the free bitangents to two dilated obstacles, which in turn are represented as graph edges. In addition, each pair of neighboring tangency points on a disc $B_{w_{\max}}(p_i)$ are connected by a circular arc. This visibility graph can be constructed in $O(n \log n + E)$ time, where E is the number of visibility edges in the graph [PV96].

Given a path-planning query, namely two endpoints s and g, we first check if the straight line segment sg is free. If it is, it should serve as the backbone of the corridor connecting s and g. Otherwise, we treat s and g as graph vertices and add all free tangents from s and from g to the discs as graph edges. We then perform Dijkstra's algorithm from s to find a shortest path to g in the resulting graph. Dijkstra's algorithm minimizes the length of the path according to some weight measure $\omega(e)$ we define for each graph edge e, and in our case we define $\omega(e)$ to be the weighted length of e. Note that all edges in the graph represent line segments or circular arcs that have clearance of at least $w_{\max}$, so $\omega(e)$ is simply the length of the curve associated with e.

Proposition 6.5 *Given a set $\mathcal{O}$ of n point obstacles in the plane that are well-separated with respect to $w_{\max}$, and two endpoints s and g with clearance at least $w_{\max}$, it is possible to compute the optimal corridor connecting s and g in $O(E \log n)$ time using the visibility graph of the dilated obstacles, where E is the number of visibility edges in this graph.*

6.2.3 Corridors amidst Point Obstacles: The General Case

The Bounded Voronoi Diagram

In case the endpoints s and g have arbitrary clearance, and the dilated obstacles $B_{w_{\max}}(p_1), \ldots, B_{w_{\max}}(p_n)$ are not necessarily pairwise disjoint in their interiors, let us consider $\mathcal{M} = \bigcup_{i=1}^{n} B_{w_{\max}}(p_i)$. The boundary of $\mathcal{M}$ comprises whole circles and circular arcs, such that a common endpoint of two arcs is a reflex vertex. We now construct $\mathcal{V}$, the Voronoi diagram of the points, and compute the intersection $\mathcal{V} \cap \mathcal{M}$, namely the portions of the Voronoi edges contained within the union of the dilated obstacles. Note that reflex vertices are equidistant to two point obstacles, so they serve as the connection points between the

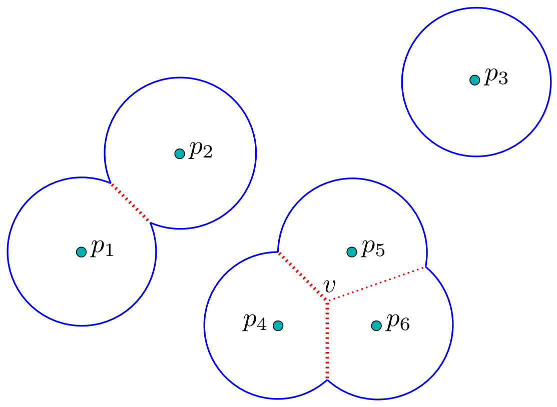


Figure 6.2: The bounded Voronoi diagram of six points. The boundary of $\mathcal{M}$, the union of the dilated point obstacles, is drawn is solid lines. The Voronoi edges are dotted.

Voronoi edges and the boundary arcs of $\mathcal{M}$; see Figure 6.2 for an illustration. The Voronoi edges, together with the arcs that form the boundary of $\mathcal{M}$, constitute the *bounded Voronoi diagram* of the point set $\mathcal{O} = \{p_1, \ldots, p_n\}$, which we denote by $\hat{\mathcal{V}}(\mathcal{O})$.

Note that $\hat{\mathcal{V}}(\mathcal{O})$ partitions the plane into two-dimensional cells of two types: bounded Voronoi regions of the point obstacles, and regions where the clearance is larger than $w_{\max}$. Given two points s' and g' that belong to the same cell ζ, we know that:

- If ζ is a cell whose clearance is greater than $w_{\max}$, the optimal backbone path between $s' = (x_1, y_1)$ and $g' = (x_2, y_2)$ is the straight line segment σ that connects them, provided that σ does not intersect any feature of $\hat{\mathcal{V}}(\mathcal{O})$. The weighted length of this segment simply equals the Euclidean distance:

$$L^*(\sigma) = \|g' - s'\| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}\ . \tag{6.7}$$

- If ζ is a bounded Voronoi cell of a point obstacle p_i, the optimal backbone path between s' and g' is a spiral arc σ centered at p_i, provided that a does not intersect any feature of $\hat{\mathcal{V}}(\mathcal{O})$. If $s' = r_1 e^{i\theta_1}$ and $g' = r_2 e^{i\theta_2}$ are the polar coordinates of the endpoints with respect to p_i, the weighted length of a is given by (recall that from Equation (6.6) we have $b = \frac{1}{\theta_2 - \theta_1} \cdot \ln \frac{r_2}{r_1}$):

$$\begin{aligned} L^*(\sigma) &= \int_{\theta_1}^{\theta_2} \frac{w_{\max}}{r(\theta)} \sqrt{r^2(\theta) + \left(\frac{dr}{d\theta}\right)^2(\theta)}\ d\theta = \int_{\theta_1}^{\theta_2} \frac{w_{\max}}{ae^{b\theta}} \sqrt{1+b^2}ae^{b\theta}\ d\theta = \\ &= \int_{\theta_1}^{\theta_2} w_{\max}\sqrt{1+b^2}\ d\theta = w_{\max}\sqrt{1+b^2}(\theta_2 - \theta_1) = \\ &= w_{\max}\sqrt{(\theta_2-\theta_1)^2 + (\ln r_2 - \ln r_1)^2}\ . \end{aligned} \tag{6.8}$$

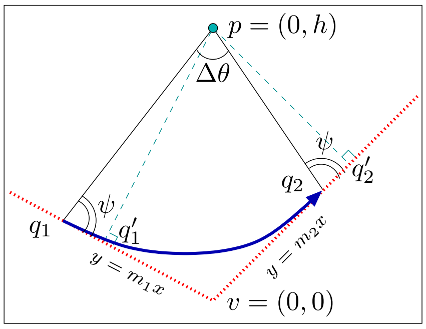
We characterized the optimal backbone paths in each two-dimensional cell of $\hat{\mathcal{V}}(\mathcal{O})$. Let us examine the other features of the diagram. It is not difficult to see that the edges of $\hat{\mathcal{V}}(\mathcal{O})$ are *locally optimal*, namely they can serve as backbone paths of optimal corridors (see

Figure 6.3(a)). We already know that portions of the circular arcs that form the boundary of $\mathcal{M}$ are locally optimal, and that the weighted length of such a circular arc simply equals its length. The Voronoi edges are also locally optimal: given s' and g' on the same Voronoi edge, the optimal backbone path that connects them is simply the straight line segment $s'g'$ which coincides with the Voronoi edge.

Following the construction of the visibility graph of the dilated point obstacles (Section 6.2.2), it is possible to add *visibility edges* to the bounded Voronoi diagram, namely to consider every free bitangent of two circular arcs, every free line segment from a reflex vertex tangent to a circular arc and every free line segment between two reflex vertices. The resulting construct is the visibility–Voronoi diagram of the point obstacles for a clearance value $w_{\max}$; compare with the construction for polygonal obstacles presented in Section 5.2. However, a path extracted from the $\mathrm{VV}^{(w_{\max})}$-diagram may pass through Voronoi vertices and reflex vertices, thus it may contain sharp turns. According to Lemma 6.2, such a path cannot serve as a backbone to an optimal corridor.

One may try to rectify this problem by adding edges to the $\mathrm{VV}^{(w_{\max})}$-diagram that allow smooth shortcuts and avoid the sharp turns. We introduce such a *shortcut edge* between each pair of Voronoi edges that are incident to a common Voronoi vertex. Similarly, we introduce a shortcut edge between each pair consisting of a Voronoi edge and a visibility edge that are both incident to a common reflex vertex, moving the visibility edge so it would be tangent to one of the endpoints of this arc.

Let us consider two Voronoi chains that are incident to a common Voronoi vertex v. One of the chains separates the Voronoi cell of some point obstacle p from the cell of another point p', while the other chain separates the cells of p and p''. We form the shortcut by penetrating the Voronoi cell of p. Note that all points in this cell are closer to p than to any other point obstacle, so it is possible to view p as a single obstacle. As we have shown in Section 6.2.1, in this scenario the shortcut curve must be an arc of a logarithmic spiral centered at p. We next explain how to compute the endpoints of this arc and deduce the parameterization of the spiral.



Without loss of generality, we assume that the Voronoi vertex v is located at the origin and that the obstacle inducing the Voronoi cell is $p = (0, h)$ (we can always apply a rigid transformation such that this assumption holds). The two Voronoi chains χ_1 and χ_2 incident to v are supported by the lines $y = m_1 x$ and $y = m_2 x$, respectively. Note that h, m_1 and m_2 are known constants; see the figure to the right for an illustration. The angle $\measuredangle q_1 v q_2$ is also known and equals $\left(\arctan \frac{m_1 - m_2}{1 + m_1 m_2}\right)$. We look for two points $q_1 = (x_1, m_1 x_1)$ and $q_2 = (x_2, m_2 x_2)$ that form the endpoints of the logarithmic spiral. In a polar coordinate system whose origin is p, we denote the coordinates of these points as $q_1 = r_1 e^{i\theta_1}$ and $q_2 = r_2 e^{i\theta_2}$.

First note that the angle that χ_1 forms with $\vec{q_1 p}$ and the angle that χ_2 forms with $\vec{q_2 p}$ are equal, and we denote them by ψ. Let q_1' and q_2' be two points on χ_1 and χ_2 respectively, such that $\vec{q_i' p}$ is perpendicular to χ_i. We thus have $\triangle p q_1 q_1' \sim \triangle p q_2 q_2'$, and as the length of

$\vec{q_i'p}$ equals the distance of the point p from the line $y = m_i x$, we can write:

$$\frac{r_2}{r_1} = \frac{\|q_2 - p\|}{\|q_1 - p\|} = \frac{\|q_2' - p\|}{\|q_1' - p\|} = \left|\frac{m_2}{m_1}\right| \cdot \sqrt{\frac{1 + m_1^2}{1 + m_2^2}} \ .$$

Note that as $\measuredangle pq_1v + \measuredangle pq_2v = \psi + (\pi - \psi) = \pi$, and since the sum of the angles in the quadrangle $\Box pq_1vq_2$ is 2π, then $\measuredangle q_1pq_2 = \pi - \measuredangle q_1vq_2$. Note that $\Delta\theta = \theta_2 - \theta_1$, and as both q_1 and q_2 lie on the logarithmic spiral $r = ae^{b\theta}$, we can write:

$$\begin{aligned} \frac{r_2}{r_1} &= \frac{ae^{b\theta_2}}{ae^{b\theta_1}} = e^{b\Delta\theta} \ , \\ b &= \frac{1}{\Delta\theta} \cdot \ln\frac{r_2}{r_1} \ . \end{aligned} \tag{6.9}$$

As the angle that $y = m_i x$ forms with the x-axis is $(\arctan m_i)$ and the angle between $\vec{q_ip}$ and the x-axis is $\left(\arctan \frac{m_i x_i - h}{x_i}\right)$, we can express ψ as follows (for $i \in 1, 2$):

$$\tan\psi = \frac{\frac{m_i x_i - h}{x_i} - m_i}{1 + \frac{m_i x_i - h}{x_i} m_i} = \frac{-h}{(1 + m_i^2)x_i - m_i h} \ .$$

However, b is known from Equation (6.9) and equals $\cot(\pi - \psi) = -\frac{1}{\tan\psi}$. We can thus compute x_1 and x_2 simply by solving the linear equations:

$$b = \frac{1 + m_i^2}{h} \cdot x_i - m_i \qquad (i \in 1, 2) \ .$$

Computing the shortcut arcs for reflex vertices is similar. In this case we look for a point q' on the boundary of $B_{w_{\max}}(p)$ where the spiral arc that shortcuts the reflex vertex smoothly connects to the visibility segment; the visibility segment is also modified as q' now becomes its endpoint, instead of the reflex vertex. However, introducing shortcuts for a single Voronoi vertex or for a single reflex vertex is not sufficient. We can show, for instance, that it is sometimes possible to shortcut two Voronoi vertices v_1 and v_2 at once by connecting two Voronoi edges that are separated by another edge using a single curve. This curve may be contained in a single Voronoi cell, as in the example depicted in Figure 6.3(b) (compare it to the case depicted in Figure 6.3(a)), or it may cross the Voronoi edge v_1v_2 at some point q' (see Figure 6.3(c)).

We should therefore continue and examine the possibility of shortcutting $k > 2$ Voronoi vertices by considering sequences of $(k+1)$ contiguous Voronoi edges and trying to locate an endpoint q_1 on the first edge and q_2 on the last edge that are connected by a smooth curve that comprises spiral arcs. This operation is not trivial, and requires solving a system of low-degree polynomial equations with $2(c+1)$ unknowns, where c is the number of crossings between the shortcut curve and the Voronoi diagram. In some scenarios it may be possible to construct shortcuts to $\Theta(n)$ Voronoi vertices by considering sequences of $\Theta(n)$ contiguous Voronoi edges, thus the size of the augmented diagram may blow up exponentially.

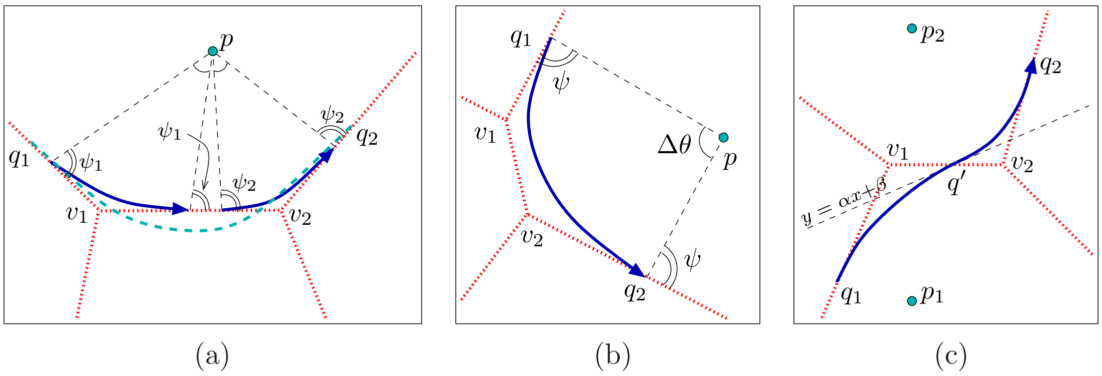


Figure 6.3: (a) The spiral arc connecting q_1 and q_2 (dashed) crosses the Voronoi edge v_1v_2; the optimal backbone path between q_1 and q_2 therefore comprises two spiral arcs that shortcut v_1 and v_2 (solid arrows) and portions of Voronoi edges. (b) Shortcutting two adjacent Voronoi vertices v_1 and v_2 by a single spiral arc. Computing the spiral shortcut in this case is very similar to shortcutting a single vertex; the only difference is that we consider the pentagon $pq_1v_1v_2q_2$ and therefore $\measuredangle q_1pq_2 = 2\pi - (\measuredangle q_1v_1v_2 + \measuredangle v_1v_2q_2)$. (c) Shortcutting two Voronoi vertices by a cross-cell curve obtained by a smooth concatenation of two spiral arcs. Both arcs have a common tangent $y = \alpha x + b$, which crosses the Voronoi edge v_1v_2 at q'.

An ε-Approximation Algorithm for Optimal Backbone Paths

We therefore devise an approximation algorithm based on the structure of the bounded Voronoi diagram $\hat{\mathcal{V}}(\mathcal{O})$ and the planar partition it induces. Given $\varepsilon > 0$, we subdivide the line segments and the circular arcs that form the features of $\hat{\mathcal{V}}(\mathcal{O})$ into small intervals of length $\frac{c(I)}{w_{\max}}\varepsilon$: as ε is small, we consider the clearance of an interval I to be constant and denote it by $c(I)$. Notice that (i) the intervals are shorter in regions where the clearance is smaller, and (ii) that each interval has weighted length ε. It follows that if Λ is the total weighted length of the features of $\hat{\mathcal{V}}(\mathcal{O})$, then there are $\frac{\Lambda}{\varepsilon}$ intervals in total. Let us now define a graph $\mathcal{D}$ whose set of nodes equals the set of intervals $\mathcal{I}$. Each interval is incident to two of the cells defined by the bounded Voronoi diagram, and we connect $I_1, I_2 \in \mathcal{I}$ by an edge only if they are incident to a common cell. This edge is:

- a line segment in a cell where the clearance is larger than $w_{\max}$,
- a spiral segment in a Voronoi region of one of the point obstacles,
- a circular arc on the boundary of a dilated obstacle, or
- a straight line segment on a Voronoi edge.

In addition, an edge should not cross any of the features of $\hat{\mathcal{V}}(\mathcal{O})$. Using a brute-force algorithm that checks each candidate edge versus the $O(n)$ diagram features, $\mathcal{D}$ can be constructed in $O\left(\frac{\Lambda^2}{\varepsilon^2}n\right)$ time.

Given two endpoints s and g, we can connect them to the graph and use Dijkstra's algorithm to compute a near-optimal backbone connecting s and g in $O\left(\frac{\Lambda^2}{\varepsilon^2}\right)$ time. Let γ^*

be the backbone path of the optimal corridor between s and g. We can subdivide the path into k *maximal segments* $\gamma_1, \ldots, \gamma_k$, such that the interior of each path segment is contained in a single cell of the bounded Voronoi diagram, or overlap an edge of $\hat{\mathcal{V}}(\mathcal{O})$ (a maximal path segment may thus be a straight line segment, a spiral arc, a portion of a circular arc or a portion of a Voronoi edge). We next show that $k = O(n)$ and that each such segment is approximated by an edge in the graph $\mathcal{D}$ we have constructed.

Lemma 6.6 *An optimal path γ^* consists of $O(n)$ maximal segments.*

Proof: We have already seen that the edges of the bounded Voronoi diagram are locally optimal paths. Hence, if there are two points on an edge of $\hat{\mathcal{V}}(\mathcal{O})$ belonging to γ^*, these points are connected by a portion of the diagram edge. For each edge e in $\hat{\mathcal{V}}(\mathcal{O})$, one of the following holds:

- The optimal path γ^* does not intersect e at all, or
- γ^* crosses e exactly once, or
- γ^* contains one continuous portion of e, and this portion is adjacent to two path segments that do not lie on $\hat{\mathcal{V}}(\mathcal{O})$.

At the same time, the endpoints of a maximal path segment whose interior does not lie on the bounded Voronoi diagram must coincide with $\hat{\mathcal{V}}(\mathcal{O})$, as they are cross points between two diagram cells.

As the complexity of $\hat{\mathcal{V}}(\mathcal{O})$ is linear in the number of point obstacles n, the complexity of γ^* is $O(n)$ as well. $\square$

Lemma 6.7 *For each maximal segment γ_i of the optimal backbone path γ^*, there exists an edge e in $\mathcal{D}$ such that $L^*(e) < L^*(\gamma_i) + 2\varepsilon$.*

Proof: Let us denote the endpoints of the path segment γ_i by q_1 and q_2, and let I_1 and I_2 be the intervals that contain these endpoints, respectively.

In case γ_i is a straight line segment in a cell ζ whose clearance is greater than $w_{\max}$, then its weighted length is $\|q_2 - q_1\|$, the Euclidean distance between its endpoints. In the graph $\mathcal{D}$ there exists an edge connecting I_1 and I_2, and we denote its endpoints by $\tilde{q}_1$ and $\tilde{q}_2$. By the construction of the intervals, we know that $\|q_j - \tilde{q}_j\| \leq \frac{c(I_j)}{w_{\max}}\varepsilon = \varepsilon$ (for $j = 1, 2$), hence:

$$\|\tilde{q}_2 - \tilde{q}_1\| < \|q_2 - q_1\| + 2\varepsilon \ .$$

Similar arguments hold when γ_i is a circular arc with clearance $w_{\max}$.

In case γ_i is a segment on a Voronoi edge, the graph $\mathcal{D}$ contains a segment $\tilde{q}_1\tilde{q}_2$ that in the worst case extends $\frac{c(q_1)}{w_{\max}}\varepsilon$ to one side of q_1 and $\frac{c(q_2)}{w_{\max}}\varepsilon$ to the other side of q_2. Since the contribution of each of these extensions is $\frac{w_{\max}}{c(q_j)}$ times its length (for $j = 1, 2$), the weighted length of $\tilde{q}_1\tilde{q}_2$ is at most 2ε more than $L^*(\gamma_i)$.

In case that γ_i is a spiral arc contained in a Voronoi cell of a point obstacle p_i, let us denote by $\tilde{\gamma}_i$ the spiral arc connecting the intervals I_1 and I_2 in $\mathcal{D}$, which is the optimal path connecting two endpoints $\tilde{q}_1 \in I_1$ and $\tilde{q}_2 \in I_2$. In particular, the weighted length of $\tilde{\gamma}_i$ is less than the weighted length of the path comprising the line segment $\tilde{q}_1 q_1$, the spiral arc γ_i, and the line segment $q_2 \tilde{q}_2$. We therefore obtain:

$$\begin{aligned} L^*(\tilde{\gamma}_i) &< L^*(\tilde{q}_1 q_1) + L^*(\gamma_i) + L^*(q_2 \tilde{q}_2) \\ &\leq \frac{w_{\max}}{c(I_1)} \cdot \|\tilde{q}_1 - q_1\| + L^*(\gamma_i) + \frac{w_{\max}}{c(I_2)} \cdot \|q_2 - \tilde{q}_2\| \leq L^*(\gamma_i) + 2\varepsilon \ . \end{aligned}$$

The length of the approximated spiral arc contained in $\mathcal{D}$ can therefore be at most $L^*(\gamma_i)+2\varepsilon$. $\square$

As two consecutive segments γ_i and γ_j of the optimal path both have an endpoint in a common interval, we know that the edges from $\mathcal{D}$ approximating γ_i and γ_j are connected in a common vertex of $\mathcal{D}$. Hence, the sequence of edges approximating each of the optimal path segments form a continuous path $\tilde{\gamma}$. This path is near-optimal:

Corollary 6.8 *Given two endpoints s and g, it is possible to use the graph $\mathcal{D}$ and compute a near-optimal backbone path $\tilde{\gamma}$ connecting s and g in $\tilde{O}\left(\frac{\Lambda^2}{\varepsilon^2}\right)$ time,*[3] *such that $L^*(\tilde{\gamma}) < L^*(\gamma^*) + O(n)\varepsilon$.*

Alternatively, we can choose the length of the intervals to be $\frac{c(I)}{w_{\max}} \cdot \frac{\varepsilon}{n}$. In this case, a near-optimal backbone path $\tilde{\gamma}$ can be found in $\tilde{O}\left(\frac{\Lambda^2}{\varepsilon^2} n^2\right)$ time, such that $L^*(\tilde{\gamma}) < L^*(\gamma^*)+2\varepsilon$.

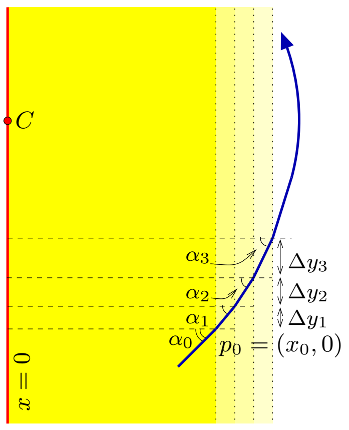
6.3 Optimal Corridors amidst Polygonal Obstacles

In this section we generalize the data structures introduced in Section 6.2 to compute optimal corridors amidst polygonal obstacles.

6.3.1 Moving Near a Single Polygon

As we did in the case of point obstacles, we first examine how an optimal backbone path looks like in the vicinity of a single obstacle. Note that a polygon P can be viewed as a collection of points (vertices) and line segments (edges), such that the distance of a point $q \in \mathbb{R}^2$ to P is either attained on a polygon vertex or in the interior of an edge. We can thus subdivide the plane into regions, such that the closest polygon feature is the same for all points in any of the regions. Using the analysis we performed in Section 6.2.1 we already know that the optimal backbone path in a region closest to a polygon vertex is an arc of a logarithmic spiral. We now study the case of two points that lie in a region closest to a polygon edge.

[3]We use the $\tilde{O}$-notation as we neglect poly-logarithmic factors.



Without loss of generality, we shall assume that the polygon edge we consider is an arbitrarily long segment of the vertical line $x = 0$, and analyze the optimal backbone path γ between two points s and g, whose distance from this line is less than $w_{\max}$ (see the figure to the right for an illustration). Note that the width of the corridor at $\gamma(t) = (x(t), y(t))$ simply equals $|x(t)|$. We begin by approximating the backbone path by a polyline. Assume that $\gamma(t)$ passes through a point $p_0 = (x_0, 0)$ and forms an angle α_0 with the line $y = 0$ perpendicular to the obstacle. For some small $\Delta x > 0$ we can define the lines $x = x_0, x = x_0 + \Delta x, x = x_0+2\Delta x, \ldots$, where each pair of neighboring lines define a vertical slab; since Δx is small we assume that the distance of all points in the slab from the obstacle is constant and equals $x_0 + k\Delta x$. We can now use Equation (6.2) and write:

$$\begin{aligned}
\sin\alpha_1 &= \frac{x_0 + \Delta x}{x_0} \sin\alpha_0 \ , \\
\sin\alpha_2 &= \frac{x_0 + 2\Delta x}{x_0 + \Delta x} \sin\alpha_1 = \frac{x_0 + 2\Delta x}{x_0} \sin\alpha_0 \ , \\
\vdots \quad & \qquad \vdots \\
\sin\alpha_k &= \frac{x_0 + k\Delta x}{x_0} \sin\alpha_0 \ .
\end{aligned}$$

If we examine the kth slab we can write $x = x_0 + k\Delta x$, so we have:

$$\Delta y_k = \Delta x \tan\alpha_k = \Delta x \cdot \frac{\sin\alpha_k}{\sqrt{1 - \sin^2\alpha_k}} = \Delta x \cdot \frac{x \sin\alpha_0}{\sqrt{x_0^2 - x^2 \sin^2\alpha_0}} \ . \tag{6.10}$$

Letting Δx tend to zero we obtain a smooth curve. We can use Equation (6.10) to express the derivative of the curve and we obtain:

$$y'(x) = \lim_{\Delta x \longrightarrow 0} \frac{\Delta y_k}{\Delta x} = \frac{x \sin\alpha_0}{\sqrt{x_0^2 - x^2 \sin^2\alpha_0}} \ , \tag{6.11}$$

$$y(x) = -\frac{1}{\sin\alpha_0} \sqrt{x_0^2 - x^2 \sin^2\alpha_0} + K \ . \tag{6.12}$$

As the point $(x_0, 0)$ lies on the curve we can express the constant K:

$$K = 0 + \frac{1}{\sin\alpha_0} \sqrt{x_0^2 - x_0^2 \sin^2\alpha_0} = \frac{\sqrt{1 - \sin^2\alpha_0}}{\sin\alpha_0} x_0 = x_0 \cot\alpha_0$$

Observe that $y(x)$ is defined only for $x < \frac{x_0}{\sin\alpha_0}$. When $x = \frac{x_0}{\sin\alpha_0}$ the path is reflected from the vertical wall and starts approaching the obstacle. Indeed, by squaring and re-arranging Equation (6.12) we obtain:

$$x^2 + (y - x_0 \cot\alpha_0)^2 = \left(\frac{x_0}{\sin\alpha_0}\right)^2 \ ,$$

thus we conclude that γ is a circular arc, whose supporting circle is centered at $C = (0, x_0 \cot\alpha_0)$ and its radius is $\frac{x_0}{\sin\alpha_0}$.

Proposition 6.9 *Given a start position $s = (x_s, y_s)$ and a goal position $g = (x_g, y_g)$ in the vicinity of a single segment obstacle supported by $x = 0$ and with $0 < x_s, x_g \leq w_{\max}$, the backbone of the optimal corridor between these two endpoints is a circular arc supported by the a circle of radius r^* that is centered at $(0, y^*)$, where (we assume that $y_s \neq y_g$, otherwise the optimal backbone path is simply the line segment sg):*

$$y^* = \frac{y_s + y_g}{2} + \frac{x_g^2 - x_s^2}{2(y_g - y_s)} \ , \tag{6.13}$$

$$r^* = \sqrt{\frac{1}{2}(x_s^2 + x_g^2) + \frac{1}{4}(y_g - y_s)^2 + \frac{(x_g^2 - x_s^2)^2}{4(y_g - y_s)^2}} \ . \tag{6.14}$$

6.3.2 Moving amidst Multiple Polygons

We are given a set $\mathcal{P} = \{P_1, \ldots, P_k\}$ of polygonal obstacles having n vertices in total, along with a preferred corridor width $w_{\max}$.

We first mention that if the polygons are well-separated, namely the distance between each P_i and P_j $(1 \leq i < j \leq k)$ is more than $2w_{\max}$, we can use the visibility graph of the dilated polygons to plan optimal backbone paths. The dilated obstacles in this case are Minkowski sums of the polygonal obstacles with a disc of radius $w_{\max}$ and their boundary comprises line segments, which correspond to dilated polygon edges, and circular arcs, which correspond to dilated vertices. Visibility edges in this case correspond to line segments tangent to two circular arcs. As the polygons may not be convex, computing the visibility graph of the dilated obstacles may not be possible in an output-sensitive manner and requires $O(n^2 \log n)$ time. Proving that the visibility graph indeed contains optimal backbone paths is done in exactly the same manner as we did in Section 6.2.2 for point obstacles.

In case there exist narrow passages between the obstacles, we generalize the construction detailed in Section 6.2.3 to polygons, and introduce the bounded Voronoi diagram of the set of polygons $\mathcal{P}$. We note the following facts:

- The portions of the Voronoi diagram we consider comprise *Voronoi chains* that are sequences of Voronoi edges (see also Section 5.1.2). A Voronoi edge may be induced by two polygon vertices or by two polygon edges, in which case it is a line segment, or by a polygon vertex and an edge of another polygon, in which case it is a parabolic arc.

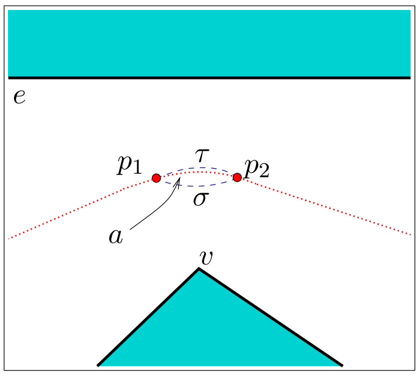
- As polygons in $\mathcal{P}$ need not be convex, the dilated obstacle boundaries may contain reflex vertices induced by a single polygon, which are of no interest. We only need to consider reflex vertices that are generated by the intersection of two (or more) dilated obstacle boundaries. As in Section 5.2, we refer to such vertices as *chain points*.

- The bounded Voronoi diagram $\hat{\mathcal{V}}(\mathcal{P})$ also contains edges that separate the Voronoi cells of adjacent polygon features, namely a polygon edge and a vertex incident to this edge. These edges are line segments perpendicular to the obstacles (see Figure 6.4 for an illustration).

Observe that if we are given two points on the same Voronoi chain, then the locally optimal backbone path between them is simply the segment of the chain they define. This is clear in case of point obstacles, as the edges are straight line segments. In case of chains that separate Voronoi cells of polygons and may contain parabolic arcs this fact is less obvious. We therefore prove the following lemma:

Lemma 6.10 *Parabolic arcs on a Voronoi chain are locally optimal.*

Proof: To prove local optimality, we show that it is not possible to shortcut such an arc segment defined by two points on the parabolic edges by choosing a shorter route that is closer to one of the polygons, as such a route always has a larger weighted length.



Consider a parabolic Voronoi arc a induced by a polygon vertex v and an edge e of another polygon, and let p_1 and p_2 be two points on a. Assume that it is possible to shortcut the portion of a defined by p_1 and p_2 by penetrating the Voronoi cell of e. In this case, the shortcut is a circular arc σ centered at some point on e; this arc clearly penetrates the Voronoi cell of the vertex v, as can be seen in the illustration to the right. On the other hand, if we try to create a shortcut contained in the Voronoi cell of v, we end up with a spiral arc τ centered at v. As p_1 and p_2 are equidistant from v, τ is a circular arc, whose curvature is larger than that of the parabolic edge, hence it penetrates the Voronoi cell of e. Either way, we reach a contradiction, and we conclude that the parabolic arc is locally optimal. □

Corollary 6.11 *As all features of the bounded Voronoi diagram $\hat{\mathcal{V}}(\mathcal{P})$ are locally optimal, Lemma 6.6 also applies for optimal paths amidst polygonal obstacles. Namely, the complexity of an optimal backbone path amidst polygonal obstacles is linear in the number of obstacle vertices.*

$\hat{\mathcal{V}}(\mathcal{P})$ subdivides the plane into cells of three types: regions where the clearance is greater than $w_{\max}$, Voronoi cells of polygon vertices, and Voronoi cells of polygon edges. We have already encountered cells of the first two types in the bounded Voronoi diagram of a set of points (Section 6.2.3). We also know from Proposition 6.9 that if we have two points in the Voronoi cell of a polygon edge, the optimal backbone path connecting them is a circular arc whose center lies on this edge. Assume, without loss of generality, that the obstacle edge lies on the line $y = 0$ and that the center of the circular arc a is the origin, and let $r^* e^{i\theta_1}$ and $r^* e^{i\theta_2}$ be the arc endpoints. The weighted length of the circular arc is therefore given by (note that $r(\theta) = r^*$):

$$\begin{aligned} L^*(a) &= \int_{\theta_1}^{\theta_2} \frac{w_{\max}}{r^* \sin\theta} \sqrt{r^2(\theta) + \left(\frac{dr}{d\theta}\right)^2(\theta)}\, d\theta = \int_{\theta_1}^{\theta_2} \frac{w_{\max}}{\sin\theta}\, d\theta = \\ &= w_{\max} \left(\ln \frac{1-\cos\theta}{\sin\theta} \right)\Bigg|_{\theta_1}^{\theta_2} = w_{\max} \left(\ln\tan\frac{\theta_2}{2} - \ln\tan\frac{\theta_1}{2} \right) . \end{aligned} \tag{6.15}$$

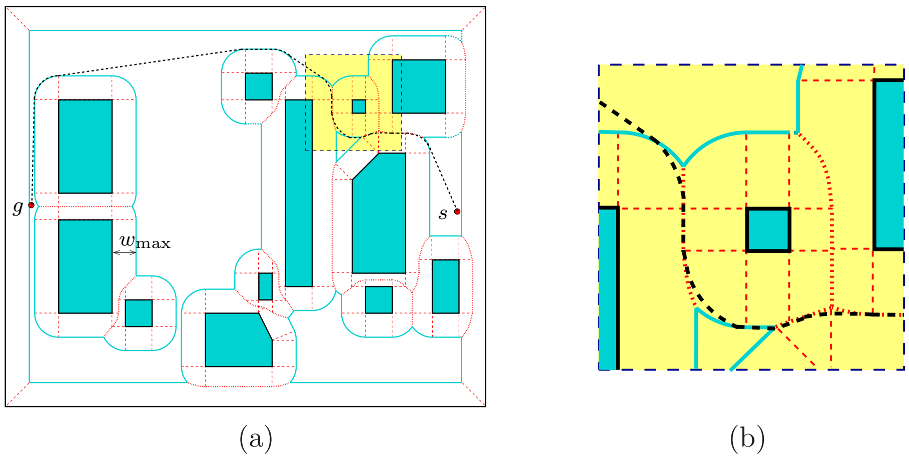


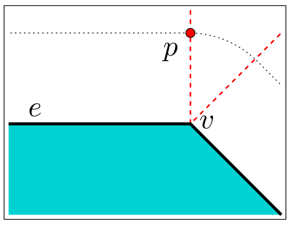
Figure 6.4: (a) A near-optimal backbone path (dashed) amidst polygonal obstacles, overlayed on top of the bounded Voronoi diagram of the obstacles. Boundary edges are drawn in light solid lines, Voronoi chains between polygons are dotted, and Voronoi edges that separate cells of adjacent polygon features are drawn in a light dashed line. The bounded Voronoi diagram was computed using the software described in Section 5.4. The backbone path was computed using an A* algorithm on a fine grid discretizing the environment. (b) Zooming in on a portion of the path; note the shortcuts the path takes.

(The last transition is due to the half-angle formula $\tan\frac{\varphi}{2} = \frac{1-\cos\varphi}{\sin\varphi}$.)

The approximation algorithm given in Section 6.2.3 can also be extended to handle polygonal obstacles. In this case we refine the Voronoi cells according to the closest polygon feature (vertex or edge), and accordingly we also consider intervals that lie on Voronoi edges that subdivide the Voronoi cell of each polygon into simple regions. Each region is induced by a polygon vertex, a polygon edge, or corresponds to regions where the clearance is above $w_{\max}$. However, such Voronoi edges have zero clearance in one of their endpoints, which would make the intervals (which should be of length $\frac{c(I)}{w_{\max}}\varepsilon$), arbitrarily small as $c(I)$ approaches zero. In addition, Λ, the total weighted length of $\hat{\mathcal{V}}(\mathcal{P})$, becomes infinity. This would render the bounds of Corollary 6.8 useless. Fortunately, we can prove that the minimal clearance attained on an optimal path γ^* is never smaller than the minimal clearance attained at the Voronoi *chains* of $\hat{\mathcal{V}}(\mathcal{P})$ (which only contain Voronoi arcs induced by features of *different* polygons). As a consequence, we only need to subdivide into intervals the portions of the Voronoi edges of $\hat{\mathcal{V}}(\mathcal{P})$ that have clearance larger than this minimum, and simply disregard the portions that lie too close to the obstacles.

Lemma 6.12 *The minimal clearance attained on an optimal backbone path γ^* is greater than (or equal to) the minimal clearance attained at the Voronoi chains of $\hat{\mathcal{V}}(\mathcal{P})$.*

Proof: First we observe that for each of the spiral segments and circular segments of the optimal backbone path γ^* (recall that such segments correspond to portions of the path that are contained within a bounded Voronoi cell of a polygon vertex or a polygon edge) the minimal clearance is attained at one of its endpoints. This means that a local minimum along γ^* is attained at a point where it crosses a feature of $\hat{\mathcal{V}}(\mathcal{P})$, or where γ^* consists of a portion of a Voronoi chain.



However, a local minimum of γ^* cannot be attained at a Voronoi edge separating two Voronoi cells of features of the same polygon. Consider the scenario depicted to the right, and suppose that the optimal path γ^* goes through point p on a Voronoi edge separating a Voronoi region of a vertex v and a Voronoi region of a polygon edge e incident to this vertex. We assume, without loss of generality, that e is supported by some horizontal line. If p is a local minimum of γ^* in terms of clearance, the path should not go closer to the polygon than the dotted curve in the vicinity of p. As we know that γ^* is smooth, it has a well-defined slope at p. This slope should be strictly negative, otherwise the circular segment of γ^* on the left-hand side of p goes below the dotted line. However, this means that the spiral segment of γ^* on the right-hand side of p will go below the dotted circular arc, whose slope at p is exactly zero. Thus, p cannot be a local minimum of γ^*.

We conclude that local minima can only be attained at Voronoi chains, which separate features of different polygons. The clearance attained along γ^* is therefore never smaller than the minimal clearance attained at the Voronoi chains of $\hat{\mathcal{V}}(\mathcal{P})$. □

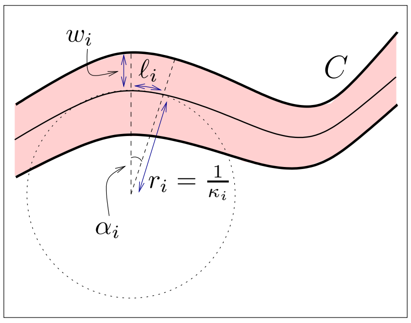
We can show that Lemma 6.7 also applies for the circular arcs inside a Voronoi cell of a polygon edge, using similar arguments to the ones we used in case of a spiral arc inside a Voronoi cell of a point. Let γ_i be the circular arc contained in a Voronoi cell of a polygon edge, and let $\tilde{\gamma}_i$ be the circular arc connecting the intervals I_1 and I_2 in $\mathcal{D}$. If we denote the endpoints of this latter arc by $\tilde{q}_1 \in I_1$ and $\tilde{q}_2 \in I_2$, we use the fact that $L^*(\tilde{\gamma}_i)$ is bounded by the weighted length of the path comprising the line segment $\tilde{q}_1 q_1$, the circular arc γ_i, and the line segment $q_2 \tilde{q}_2$, and conclude that $L^*(\tilde{\gamma}_i) < L^*(\gamma_i) + 2\varepsilon$.

Corollary 6.13 *Given a set of polygonal obstacles $\mathcal{P}$ having n vertices in total, where d is the minimal distance between a pair of polygons in $\mathcal{P}$ (namely $\min_{P,Q \in \mathcal{P}} \operatorname{dist}(P,Q)$). Let $\overline{\Lambda}$ be the total weighted length of the bounded Voronoi diagram $\hat{\mathcal{V}}(\mathcal{P})$ with respect to a given $w_{\max}$ value, ignoring portions of the diagram having clearance less than $\frac{d}{2}$. Given $\varepsilon > 0$, we can construct a graph $\mathcal{D}$ over the intervals of $\hat{\mathcal{V}}(\mathcal{P})$ in $O\left(\frac{\overline{\Lambda}^2}{\varepsilon^2} n\right)$ time, such that for each pair of endpoints s and g it is possible use $\mathcal{D}$ and compute a near-optimal backbone of a corridor C connecting s and g. $L^*(C)$ is at most $O(n)\varepsilon$ more than the weighted length of the optimal corridor connecting s and g.*

6.4 Accounting for the Corridor Curvature

In some applications having a winding backbone path decreases the quality of the corridor. In the group-motion application [KO04b], for example, when the entities move along a straight line, they can all move at the maximal possible speed. Assume that the backbone path is a circular arc and the corridor width is w, such that it is bounded by two concentric circular arcs. The entities moving along the outer arc in this case have to take a longer route, so even if we let them move at maximal speed, the other entities have to move at a lower speed and the time it takes the group to traverse such a path increases. Bounding the curvature is also very important if we wish to consider kinematic and dynamic constraints of the moving entity; see, e.g., [BK05, BL03] and the references therein.

6.4.1 Augmenting the Weighted-Length Measure



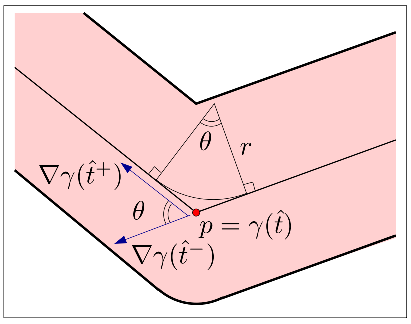
Assume that the backbone path γ is smooth and let $\kappa(t)$ be the curvature of γ at time t. We can subdivide the path into infinitesimally small segments, such that the length of the ith path segment is ℓ_i (with $\sum_i \ell_i = L$), the width of each segment, denoted w_i, is assumed constant and the curvature is also assumed constant and denoted by κ_i — see the figure on the right for an illustration. Hence, each path segment can be considered as a circular arc whose radius is $r_i = \frac{1}{\kappa_i}$ and defined by the angle $\alpha_i = \frac{\ell_i}{r_i}$. The length of the outer boundary of the corridor along the ith path segment is given by $\alpha_i(r_i + w_i)$, and we can thus bound the length of each of the corridor boundary-curves by:

$$\sum_i \alpha_i(r_i + w_i) = \sum_i \frac{\ell_i}{r_i}(r_i + w_i) = \sum_i \ell_i + \sum_i \frac{w_i}{r_i}\ell_i = L + \sum_i w_i \kappa_i \ell_i \ .$$

We therefore wish to augment the weighted length function by adding penalty for the extra length induced by the curvature of the backbone path, which is proportional to $\sum_i w_i \kappa_i \ell_i$. However, as we can make our path segments infinitesimally small, and as γ is parameterized by its length, we can simply redefine our weighted-length function for $C = \langle \gamma(t), w(t), w_{\max} \rangle$ to be:

$$L^*_\mu(C) = \int_\gamma \left(\frac{w_{\max}}{w(t)} \right)^{d-1} dt + \mu \int_\gamma w(t)\kappa(t) dt \ , \tag{6.16}$$

where $0 < \mu \leq 1$ is the weight we give to the curvature measure.



We can also account for backbone paths that contain sharp turns, and are only *piecewise* $\mathcal{C}_1$-continuous, thus the curvature of γ is not defined at a finite number of points. Let $p = \gamma(\hat{t})$ be such a point, and let θ be the angle between $\nabla\gamma(\hat{t}^-)$ and $\nabla\gamma(\hat{t}^+)$. Let us replace the sharp turn with a circular arc a of a small radius r. The arc is defined by the angle θ (see the illustration to the right), so its length is θr (θ is of course measured in radians). If r is small enough, we can assume that the corridor has a fixed width $w_p = w(\hat{t})$ over the circular arc, so we have:

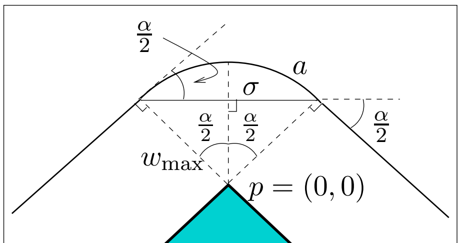
$$\lim_{r \to 0} \int_a w(t)\kappa(t) dt = \lim_{r \to 0} \int_a \frac{w_p}{r} dt = \lim_{r \to 0} \theta r \cdot \frac{w_p}{r} = \theta w_p \ .$$

We can thus abuse the curvature-integral notation, as appears in Equation (6.16), and account for sharp turns by adding the discrete weight as explained above. We note however that backbone paths of optimal corridors with respect to the augmented weighed-length measure, as defined in Equation (6.16), should be smooth and cannot contain sharp turns. To see why, we can follow the proof of Lemma 6.2, and assume that we have an optimal backbone path γ^* that contains a sharp turn, defined by the angle θ. In the original proof we show that it is always possible to shortcut the sharp turn by a circular arc that decreases the weighted length of the path. While the original path makes a sharp turn of θ radians, the shortcut also makes the same turn, but "spreads" it over the entire arc, which contains points with less clearance. The curvature penalty we give the circular shortcut is thus smaller than the penalty of the original path, so our circular shortcut decreases the augmented weighted length of the path. We conclude that a sharp turn is not possible in an optimal corridor also when we take the curvature into account.

6.4.2 Moving Amidst Well-Separated Obstacles

We are given a set $\mathcal{P}$ of obstacles (point obstacles or polygonal obstacles) in the plane, and preferred width $w_{\max}$, such that the obstacles of $\mathcal{P}$ are well-separated with respect to $w_{\max}$. Given two query points $s, g \in \mathbb{R}^2$ whose clearance value is at least $w_{\max}$, we would like to compute the backbone path connecting s and g that induces an optimal corridor with respect to the augmented measure L^*_μ.

In Section 6.2.2 we showed that such an optimal path is contained in the visibility graph of the dilated obstacles for the special case where $\mu = 0$. We next show that the same holds also for any $0 < \mu \leq 1$. We do so by examining the various types of arcs that can comprise a backbone path extracted from the visibility graph and show it is impossible to shortcut them, namely to replace any of them by a different backbone curve having a smaller weighted length. Recall that the path extracted from the visibility graph is smooth and contains line segments and circular arcs. It is clear that it is impossible to shortcut the straight line segments. We now show that it is also impossible to shortcut a circular arc in a manner that reduces the augmented weighted length of the corridor.



Consider a circular arc a defined by an angle α that lies on a dilated obstacle vertex p (we can consider point obstacles as degenerate polygons having a single vertex). Without loss of generality, we assume that p is the origin and that both arc endpoints have the same y-coordinate; see the figure to the right. We have already mentioned that the backbone path lies at a distance $w_{\max}$ from the closest obstacle, so we allow the corridor to have maximal width along the arc. Since the arc has constant curvature $\frac{1}{w_{\max}}$, the contribution of the arc a to the weighted length of the path is simply $(1 + \mu)\alpha w_{\max}$.

Let us examine what happens if we try to shortcut the arc by moving on the straight line segment σ connecting its two endpoints, which are $(-w_{\max} \sin \frac{\alpha}{2}, w_{\max} \cos \frac{\alpha}{2})$ and $(w_{\max} \sin \frac{\alpha}{2}, w_{\max} \cos \frac{\alpha}{2})$. Note that we make two sharp turns of size $\frac{\alpha}{2}$ each, so the curvature integral contributes $\mu\alpha w_{\max}$ to the weighted length, as was the case with the original arc. Let us consider the width integral — here the segment length is obviously shorter, but we have to account for a smaller corridor width. Note that because of symmetry we can write:

$$\begin{aligned}
\int_\sigma \frac{w_{\max}}{w(t)}dt &= 2\int_0^{w_{\max}\sin\frac{\alpha}{2}} \frac{w_{\max}}{\sqrt{x^2+w_{\max}^2\cos^2\frac{\alpha}{2}}} = \\
&= 2w_{\max}\cdot\ln\left(x+\sqrt{x^2+w_{\max}^2\cos^2\frac{\alpha}{2}}\right)\Bigg|_0^{w_{\max}\sin\frac{\alpha}{2}} = \\
&= 2w_{\max}\cdot\ln\frac{1+\sin\frac{\alpha}{2}}{\cos\frac{\alpha}{2}} = 2w_{\max}\cdot\ln\frac{1-\cos(\frac{\alpha}{2}+\frac{\pi}{2})}{\sin(\frac{\alpha}{2}+\frac{\pi}{2})} = \\
&= 2w_{\max}\cdot\ln\left(\tan\left(\frac{\alpha}{4}+\frac{\pi}{4}\right)\right) .
\end{aligned}$$

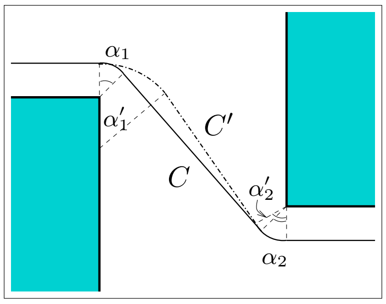
If we look at the function $f(\alpha) = w_{\max}\left(2\ln\tan(\frac{\alpha}{4}+\frac{\pi}{4})-\alpha\right)$, which describes the difference between the width integrals over σ and over a, we have $f(0)=0$ and $\lim_{\alpha\to\pi} f(\alpha)=\infty$ (note that α can never reach π). At the same time:

$$\begin{aligned}
\frac{1}{w_{\max}}\cdot f'(\alpha) &= 2\cdot\frac{1}{\tan(\frac{\alpha}{4}+\frac{\pi}{4})}\cdot\frac{1}{\cos^2(\frac{\alpha}{4}+\frac{\pi}{4})}\cdot\frac{1}{4}-1 = \\
&= \frac{1}{2\sin(\frac{\alpha}{4}+\frac{\pi}{4})\cos(\frac{\alpha}{4}+\frac{\pi}{4})}-1 = \frac{1}{\sin(\frac{\alpha}{2}+\frac{\pi}{2})}-1 .
\end{aligned}$$

Thus, $f'(\alpha)=0$ only if $\alpha=0$, and f is positive for all $0<\alpha<\pi$. We conclude that the weighted length of σ is greater than the weighted length of a.

We can show that for every convex path a' that shortcuts the arc a, the width integral over a' is larger than $\alpha w_{\max}$. This is done by approximating (with arbitrary precision) the curve by a polyline and separately considering each of the line segments this polyline contains. We conclude that selecting a shorter backbone path with less clearance will only decrease the quality of the corridor.

At the same time, it is not recommended to take wider turns. Consider the example depicted to the right, where the corridor C' has a longer backbone path than the corridor C extracted from the visibility graph of the dilated polygons. As both corridors are of maximal width, it is clear that its width integral is also larger. However, the curvature integral of each the corridors is proportional the sum of the angles defining the circular arcs, so it is obvious that the curvature integral of C' is larger than of C, as $\alpha'_1+\alpha'_2>\alpha_1+\alpha_2$. It is therefore clear that $L^*_\mu(C)<L^*_\mu(C')$ for each $0<\mu\le 1$.



◇◇
◇

In this chapter we have introduced a measure for the quality of corridors and studied the structure of optimal corridors amidst point obstacles and polygonal obstacles in the plane.

Our measure balances between the length and clearance of the backbone path of the corridor, and we show that optimal paths with respect to our measure are always smooth. We have also showed how the curvature of the backbone path can be taken into account. After studying the structure of an optimal corridor with respect to our quality measure, we devised an approximation algorithm for computing near-optimal corridors amidst obstacles, and showed that extracting backbone paths from the visibility-Voronoi diagram is also a good approximation strategy.

Chapter 7

Conclusions and Future Work

In this thesis we described how a complete and extendible implementation of planar arrangements (and their sub-structures) is used to develop exact techniques and robust applications that give accurate solutions to problems arising in domains such as motion planning, computer-aided design and solid modeling.

We now describe some future prospects, giving some insight to the future evolvement of the arrangement package, and suggesting more applications that can be implemented on top of the forthcoming infrastructure.

7.1 Handling Curves of a Higher Degree

The applications described in this thesis were mostly based on arrangements of curves with algebraic degree 2 at most, using the segment-traits class, the circle/segment traits-class, or the conic-traits class (see Section 2.3), which instantiates the arrangement template. We next review additional traits classes that are under development and highlight some of their applications.

7.1.1 A Filtered Traits-Class for Bézier Curves

An important family of planar curves that is extensively used in fields like computer graphics and computer-aided design, is the family of Bézier curves. A planar *Bézier curve* B is a parametric curve that is defined by a sequence of *control points* $p_0, \ldots, p_n$ as follows:

$$B(t) = (X(t), Y(t)) = \sum_{k=0}^{n} p_k \cdot \frac{n!}{k!(n-k)!} \cdot t^k (1-t)^{n-k} \ , \qquad t \in [0,1] \ . \tag{7.1}$$

The degree of the curve is therefore n — namely, $X(t)$ and $Y(t)$ are polynomials of degree n. In case the control polygons have rational coordinates, then both these polynomials have rational coefficients. In order to subdivide a Bézier curve into x-monotone subcurves, one simply has to compute the solutions of $X'(t) = 0$ that lie in the open interval $(0, 1)$. As $X'(t)$ has rational coefficients, its roots are obviously algebraic numbers. Moreover, if t_0 is such a root, substituting it into the curve equation results in a point $B(t_0)$ with algebraic coefficients. That is, an x-monotone subcurve is defined by $B(t)$ and by two algebraic parameter values $0 \leq t_1 < t_2 \leq 1$ that define its endpoints, which in turn have algebraic coefficients.

The intersection of two Bézier curves $B_1(s) = (X_1(s), Y_1(s))$ and $B_2(t) = (X_2(t), Y_2(t))$ can be computed by solving the polynomial system:

$$\begin{cases} X_1(s) = X_2(t) \\ Y_1(s) = Y_2(t) \end{cases} .$$

This system of equations can be separately solved for s and for t using resultant calculus, and the valid solutions are the parameter pairs $\langle s, t \rangle$ such that $s \in [0, 1]$ and $t \in [0, 1]$. Once again, as all the polynomials involved have rational coefficients, the parametric solutions are algebraic numbers, and the intersection points all have algebraic coordinates.

It is therefore possible to implement an arrangement-traits class for Bézier curves that uses exact algebraic computation employing the number-types supplied by the Core library (or the Leda library), and to handle all inputs in a robust manner. The main drawback in this approach is that the degree of the algebraic numbers involved can rapidly grow, yielding prohibitive running times.

At the same time, industrial systems that handle Bézier curves usually use bisection methods that are based on de Casteljau's algorithm and utilize the geometric properties of Bézier curves; see, e.g., [Pie93]. These methods are very fast and are especially suited for machine-precision arithmetic, yet they are vulnerable to instabilities when the output is degenerate or near-degenerate.

Our goal is to apply a geometric filtering scheme that combines these two approaches. Thus, we compute vertical tangency points and intersection points based only on bisection methods, employing interval arithmetic. As a result, each point of interest is bounded in some small isolating box. Comparing two such points is trivial in case their isolating boxes do not overlap. Otherwise, we can refine the approximation quality by performing additional bisection steps, or — in case we reach the precision limits of the machine — resort to exact computation. This approach can handle non-degenerate inputs very efficiently, and is robust to degenerate inputs as well, where the more computationally demanding procedures have to be employed.

The development of the Bézier curve traits-class is done jointly with Iddo Hanniel from the Technion. A paper describing the techniques we used in more details [HW07] has been recently accepted to *ACM Solid and Physical Modeling Symposium (SPM'07)*. Our traits class will also be included in the next public release of Cgal (Version 3.3). The traits class also allows the application of Boolean operations on sets defined by Bézier splines, namely smooth curves formed by sequences of low-degree Bézier curves (see, e.g., [CER01]). Figure 7.1 shows the intersection of two general polygons whose boundaries are formed by continuous sequences of Bézier curves. We also intend to continue our work and develop traits classes for related families of curves, such as splines and rational Bézier curves, that are used in various application domains.

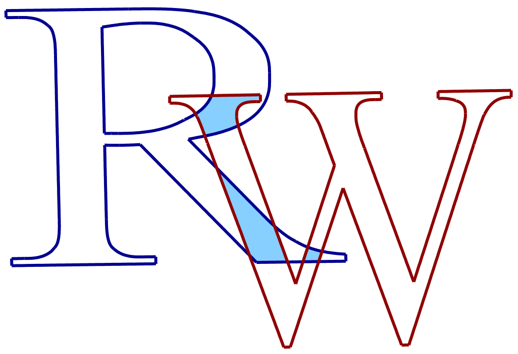


Figure 7.1: Computing the intersection (shaded) of two Times New Roman characters. The boundary of the letter R is modeled using 37 Bézier curves of degrees up to 6, while the boundary of the letter W comprises 36 Bézier curves.

7.1.2 An Emerging Curved Kernel

As already mentioned in Section 2.3.3, traits classes for algebraic curves of degrees higher than 2 — namely of curve of degree 3 [EKSW04], and special types of curves of degree 4 [BHK^{+}05] — have already been implemented. In addition, there is an ongoing effort in the Cgal community to design a concept of a *curved kernel*, which will supply primitive operations on rational algebraic curves of arbitrary degree, as the current Cgal kernels operate on linear objects. Models of the curved kernel will be parameterized with an *algebraic kernel*, which provides some basic algebraic primitives (e.g., solving a system defined by two bivariate polynomials) needed for implementing the geometric operations in the curved kernel. The curved kernel is thus decoupled from the algebraic primitives and can operate with various algebraic kernels, which in turn may employ different sets of algebraic tools.

An arrangement-traits class for handling algebraic curves, or segments of such curves, will therefore be just a thin wrapper for the curved kernel functionality, once such a kernel is introduced — as is currently the case for the segment-traits classes, which wrap the functionality of the linear kernel and adapt it to the interface required by the traits-class concepts. A fundamental problem that can be solved in an exact manner using planar arrangements of algebraic curves with rational coefficients is the motion-planning problem for a translating and rotating polygon amidst polygonal obstacles. The piano-movers' algorithm [SS83a] solves this motion-planning variant by considering the arrangement of a set of planar critical curves, whose degree is bounded by 4 (see also Section 1.1.2). In Appendix A we revisit the construction of these critical curves and apply some careful analysis to show that in case all input polygons have rational coordinates, then all critical curves have rational coefficients. This latter observation is crucial for a future implementation of the piano-movers' algorithm on top of the Cgal arrangement package.

7.2 Arrangements on Surfaces in 3D

So far we only discussed arrangements of *bounded* curves, which are usually sufficient for many applications. However, in some cases it is necessary to consider arrangements of *unbounded* curves. For example, the duality transform [CGL85] can be used for solving various problems on point sets, by constructing lines dual to the input points and examining the planar subdivision these dual lines induce on the plane.

We have recently enhanced the functionality of the Cgal arrangement package so it can handle unbounded curves as well. This enhanced capability will be included in the next public release of Cgal (version 3.3), which will also include a traits class for handling arbitrary linear objects (lines, rays and line segments).

The important property of the framework we suggest for handling arrangements of unbounded curves is that it can be extended to handle the more generic case of a 2D arrangement embedded on a surface. In Appendix B we explain how we extended the arrangement package to support unbounded curves, and show how the design principles we have used can be slightly adapted to allow the construction of arrangements on surfaces. The paradigm described in Appendix B is based on ongoing work with Eric Berberich from Max-Planck Institut für Informatik and with Efi Fogel from Tel-Aviv University. An extended abstract describing our preliminary results appeared in [BFHW07], and a more elaborate version, including some experimental results, is about to appear in [BFH$^+$07].

Supporting arrangements on generic 3D surfaces opens the door for implementing a multitude of new applications. Here we mention just a few:

- Fogel and Halperin [FH06] have recently implemented an efficient algorithm for computing Minkowski sums of convex polyhedra in 3D. In their implementation, they projected the normal diagrams of each polyhedron onto the cube $[-1, 1]^3$, such that the diagram is represented by six planar arrangements formed on the cube facets. The Minkowski sum is computed by overlaying the six arrangement pairs. Having the ability to construct a single arrangement on a sphere can simplify the computational process and help achieving additional speedups.

- Computing arrangements of circles on a sphere is a fundamental tool for developing molecular modeling applications, where each sphere models and atom and the circles correspond to intersection curves induced by pairs of atoms. Halperin and Shelton [HS98] developed a perturbation scheme that enables the construction of such arrangements using floating-point arithmetic. Their perturbation scheme works very well when one considers the var der Waals radii of the atoms, such that the intersections induced by the spheres are very sparse. However, when computing the solvent-accessible surface of a molecule, one typically needs to inflate each atom by the radius of the solvent, making the intersections denser. In some cases it may be impossible to devise a valid perturbation and one has to resort to exact computation. Cazals and Loriot [CL06] have recently have recently developed a software package that computes arrangements of circles on a sphere. However, their implementation is restricted to the case of a spherical topology, and does not generalize to arbitrary surfaces as our suggested framework.

- Arrangements on surfaces can serve as a convenient basis for a package that handles arrangements of 3D surfaces. Imagine we are given a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of surfaces in $\mathbb{R}^3$. We can consider each surface S_k separately and construct the arrangement $\mathcal{A}_k$

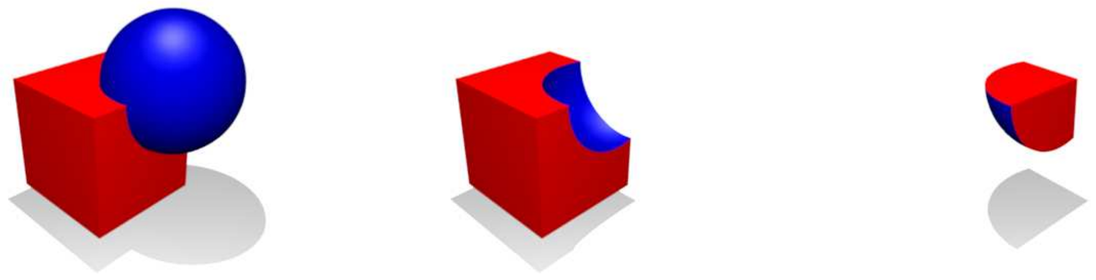


Figure 7.2: Typical operations performed by constructive geometry modelers: (a) The union of a cube and a sphere; (b) The set-difference between the cube and the sphere. (c) The intersection of the two primitives. (This figure is taken from Wikipedia, ⟨`http://en.wikipedia.org/wiki/Constructive_solid_geometry`⟩.)

of all intersection curves it forms with other surfaces in the set. The arrangements $\mathcal{A}_1, \ldots, \mathcal{A}_n$ can subsequently be connected together to properly represent the spatial arrangement $\mathcal{A}(\mathcal{S})$. Similar to what was done for 2D set-operations, it will be possible to provide a package that performs robust set-operations on general polyhedra in 3D, namely sets whose boundaries comprise surface patches.

While there already exists a Cgal package that performs set-operations of 3D Nef polyhedra [HK06], this package operates only on linear polyhedra whose facets are planar patches defined by straight-line edges. Having the ability to represent arrangement in 3D and to perform set-operations on *general* (curved) polyhedra is crucial for giving an exact solution for many fundamental problems in fields like solid modeling and motion planning.

- Computing the union of general polyhedra created by offsetting linear polyhedra, is necessary for obtaining an exact representation of the forbidden configuration space of a "free-flyer" sphere robot moving amidst polyhedral obstacles.
- As mentioned in Appendix B, we intend to provide traits classes that can handle arrangements of quadric surfaces, based on the algebraic methods proposed in [BHK$^+$05]. Operating on general polyhedra whose faces are given as quadratic surface patches plays an important role in constructive solid geometry,[1] where a complex surface or object is created by using Boolean operators to combine objects, which may be complex themselves, or are some primitive solids, such as boxes, spheres, cylinders and cones (see Figure 7.2 for an illustration). We note that all the typical primitive solids are special cases of quadric surfaces. As constructive geometry models are mainly used in applications where mathematical accuracy is important, it is highly desirable to be able to cope with such models in an accurate manner.

[1]See, e.g., ⟨`http://en.wikipedia.org/wiki/Constructive_solid_geometry`⟩.

Appendix A

Revisiting the Critical Curves in the Piano Movers' Algorithm

In the early 1980's, Schwartz and Sharir published a seminal series of articles proposing complete solutions to several motion-planning problems. In the first article [SS83a], they treated the problem of moving (translating and rotating) a line segment (a so-called *"ladder"*) amidst polygonal obstacles, and of moving a polygonal robot amidst polygonal obstacles. Both problems have three degrees of motion freedom. The "ladder" case is also discussed by Latombe [Lat91] in a somewhat more intuitive manner.

The nice property of the piano-movers' algorithm is that it enables solving the motion-planning problem using only planar geometry, without the need to perform computations in the three-dimensional configuration space. The availability of software tools for handling planar geometry makes the algorithms more feasible to implement, in comparison to algorithms that require computations in the (three dimensional) configuration space (see, e.g., the work of Avnaim *et al.* [ABF89]).

Both piano-movers' algorithms are based on the definition of planar *critical curves* and the ability to construct their arrangement, namely the subdivision into maximally-connected components they induce on the planar workspace. Bañon [Bañ90] presented an implementation for the "ladder" algorithm. However, Bañon's work uses floating-point arithmetic, making the implementation inexact. Thus, it may fail to find a path if it involves going through tight passages. If the input is not in general position and contains degeneracies, or even if it is nearly degenerate, the output may be inconsistent or inaccurate.

Here we concentrate on the algebraic definition of the critical curves and strive to achieve a compact representation for them. We give a detailed analysis of the critical curves in case of a general polygonal robot, omitted from the publicly available version of the paper [SS83a].[1] Our additional contribution is that we show that under the assumption that the input is given as polygons with rational coordinates, it is possible to represent the critical curves as algebraic curves with rational (or, equivalently, integer) coefficients. As the next releases of Cgal will include an *algebraic kernel* that can serve as a basis for an arrangement-traits

[1]This analysis was previously done by Schwartz and Sharir, but its details can only be found in a technical report [SS81] which is unfortunately not available in electronic form.

class for such algebraic curves with rational coefficients, our analysis paves the way for an exact implementation of the piano-movers' algorithm in the near future.

A.1 Motion Planning for a Ladder

The simplest form of a robot whose motion in the plane has three degrees of freedom is a line segment pq (also called a "ladder"), which is free to translate and rotate. A *configuration* c of the robot is given by $\langle x, y, \theta \rangle$, where (x, y) is the location of p and θ is the angle between the vector $\vec{pq}$ and the x-axis. A configuration c is called *free* if the ladder does not collide with any of the obstacles when placed at c. If it just touches an obstacle when placed at c, but does not penetrate any obstacle, c is called *semi-free.* If the robot placed at c penetrates some obstacle, the configuration is called *forbidden.*

In addition, we are given a set of polygonal obstacles with a total number of n vertices. Our goal is to preprocess the input scene, so we can answer motion-planning queries efficiently: Given a start and a goal configuration (both are free configurations of course), we should find collision-free motion path for the ladder, such that the robot does not penetrate any of the obstacles when moving along this path, or determine that no such path exists.

The main idea in the piano-movers algorithm is to subdivide the plane into a finite number of maximally connected cells, such that all points in the same cell share a combinatorial property which we define next. This subdivision is realized as the arrangement of a set of *critical curves*, defined by the set of configurations where a specific ladder feature (the vertex q or the interior of the ladder) is in contact with a specific obstacle feature (a vertex or an edge). Assume that we place the reference point p at (x_0, y_0). It is possible to subdivide the range $(-\pi, \pi]$ of possible orientations of the ladder into maximal sub-intervals, such that each interval is either *free*, namely the ladder is disjoint from any obstacle, or *forbidden*, as the the ladder penetrates one of the obstacles. In the latter case, we can label each boundary of every free interval with an identifier of the obstacle feature that the ladder touches at this configuration, such that the point (x_0, y_0) can be given a *characteristic label*, which is a set of labels associated with its forbidden orientation intervals. The cells are constructed in a way that guarantees that two points belonging to the same cell have the same characteristic label. Thus, it is possible to give a characteristic label to the entire cell, and to construct a graph that captures the connectivity of the free orientation intervals between neighboring cells. The task of answering a motion-planning query is thus reduced to finding a path in this connectivity graph.

We will assume that as an input we get a set of simple polygonal obstacles $P_1, \ldots, P_k$, such that each polygon is given as a sequence of points with *rational coordinates.* The length of the ladder, denoted by ℓ, is also given as part of the input and we will also assume that $\ell \in \mathbb{Q}$.

We now define a set of critical curves. First, each obstacle edge defines a critical curve which is a line segment. If an obstacle edge is defined by the endpoints $v_1 = (x_{v_1}, y_{v_1})$ and $v_2 = (x_{v_2}, y_{v_2})$, then its supporting line is given by:

$$(y_{v_1} - y_{v_2})x + (x_{v_2} - x_{v_1})y + (x_{v_1}y_{v_2} - x_{v_2}y_{v_1}) = 0 \;, \tag{A.1}$$

and is therefore an algebraic curve of degree 1 with rational coefficients. In addition, five types of critical curves are defined as follows (see Figure A.1):

Type I: For each obstacle edge E, the curve traced by p as q slides along E. This curve is a line segment parallel to E and at distance ℓ from it.

Type II: For each obstacle vertex v, the curve traced by p as q touches v and the ladder rotates around it. This curve is a circular arc centered at v and whose radius is ℓ, bounded by the two lines supporting the two obstacle edges that intersect at v.

Type III: For each obstacle edge E with a convex vertex v as its endpoint, the curve traced by p as q slides from v along E. This curve is a line segment of length ℓ that lies on the same line as E with v being its endpoint.

Type IV: For each pair of convex obstacle vertices v_1 and v_2 that are not two endpoints of a common obstacle edge and such that $d = ||v_1 - v_2|| < \ell$, the curve traced by p as the ladder slides while touching both vertices and initially q touches v_1. This curve is a line segment of length $\ell - d$ that lies on the line connecting v_1 and v_2 with v_2 as its endpoint.

Type V: For each obstacle edge E and a vertex v that is not one of E's endpoints and whose distance d from E is less than ℓ, the curve traced by p as q slides along E and the interior of the ladder touches v. This curve is the loop of the *conchoid of Nicomedes*, which is an algebraic curve of degree 4, as we will show in Section A.1.1.

We first mention that a critical curve of type I is a line segment parallel to an obstacle edge that lies at a distance ℓ from this edge. As we show in Section 3.1.2, if the edge is supported by the line $ax + by + c = 0$ (with $a, b, c \in \mathbb{Q}$) we can formulate the locus of all points at a distance ℓ from that line (recall that the signed distance of a point (x, y) from the line $ax + by + c = 0$ is given by $\frac{ax+by+c}{\sqrt{a^2+b^2}}$):

$$\frac{(ax + by + c)^2}{a^2 + b^2} = \ell^2 \ . \tag{A.2}$$

We can therefore represent a critical curve of type I as a segment of a degree 2 curve (a line-pair) with rational coefficients. We stress that the endpoints of this segment do not have rational coordinates in the general case.

A critical curve of type II is a circular arc of radius ℓ centered at an obstacle vertex $v = (x_v, y_v)$, and is therefore supported by a rational circle.

A critical curve of type III is a continuation of an obstacle edge and therefore lies on the same supporting line as this edge. A curve of type IV lies on the straight line connecting two obstacle vertices. In both cases, the supporting curve can be represented as a line with rational coefficients.

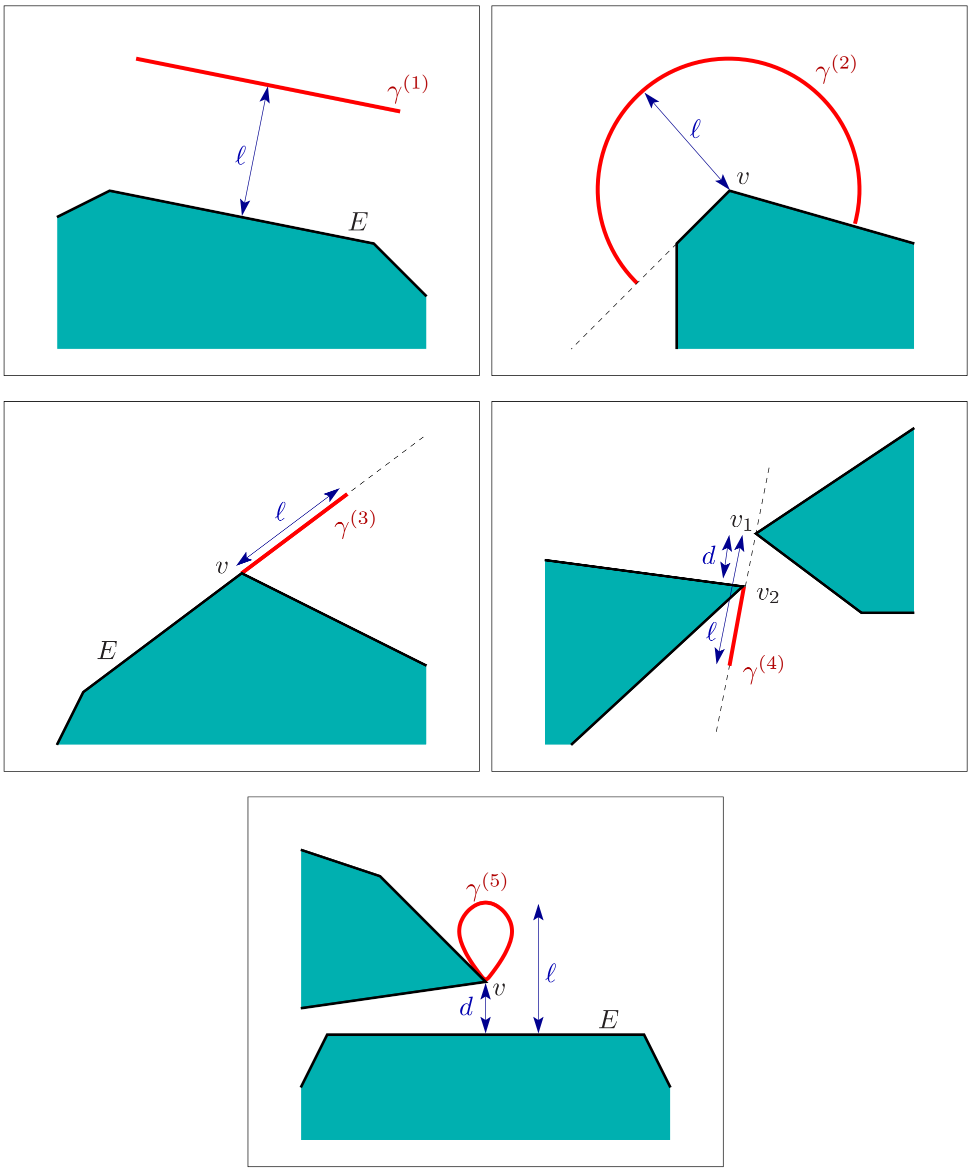


Figure A.1: The critical curves for translation and rotation of a ladder in the plane: $\gamma^{(1)}, \ldots, \gamma^{(5)}$ represent critical curves of types I–V, respectively.

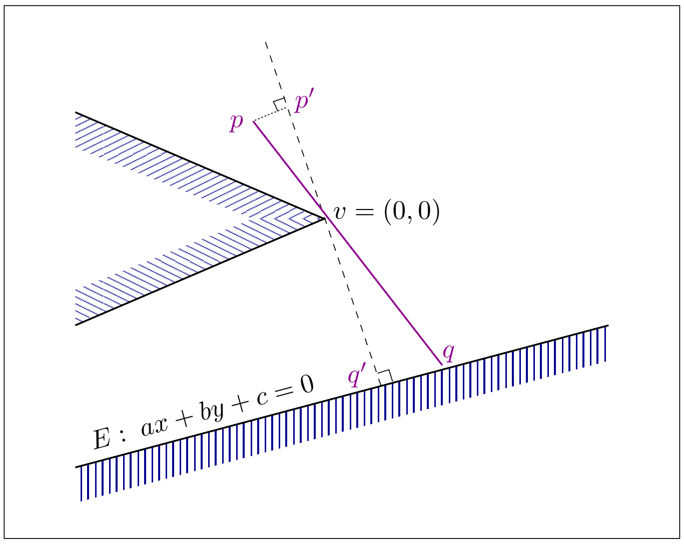


Figure A.2: Analysis of a critical curve of type V.

A.1.1 Critical Curves of Type V

We wish to determine the locus of all points (x, y) where the ladder vertex p can be located, such that its other vertex q touches the obstacle edge E, and the interior of the ladder touches the convex obstacle vertex v. Let us assume for simplicity that $v = (0, 0)$. Let $ax + by + c = 0$ be the line supporting the edge E. If we denote the x-coordinate of q by σ, we have $q = (\sigma, -\frac{a\sigma+c}{b})$, assuming without loss of generality that $b \neq 0$ (otherwise we can rotate the scene by $\frac{\pi}{2}$ and swap the roles of x and y).

The equation of the line perpendicular to E through the origin v is $bx - ay = 0$. Let p' and q' be the projection of p and q onto this line, respectively (see Figure A.2 for an illustration). We therefore have:

$$p' = \left(-\frac{a(ax+by)}{a^2+b^2}, -\frac{b(ax+by)}{a^2+b^2}\right) , \qquad q' = \left(-\frac{ac}{a^2+b^2}, -\frac{bc}{a^2+b^2}\right) .$$

The (signed) distances of these two points from the origin are given by:

$$\begin{aligned} \|p'\| &= \sqrt{\frac{a^2(ax+by)^2}{(a^2+b^2)^2} + \frac{b^2(ax+by)^2}{(a^2+b^2)^2}} = \frac{ax+by}{\sqrt{a^2+b^2}} , \\ \|q'\| &= \sqrt{\frac{a^2c^2}{(a^2+b^2)^2} + \frac{b^2c^2}{(a^2+b^2)^2}} = \frac{c}{\sqrt{a^2+b^2}} . \end{aligned}$$

We can also compute the (signed) distances of p and q from the line $bx - ay = 0$:

$$\begin{aligned} \|p - p'\| &= \frac{bx - ay}{\sqrt{a^2+b^2}} , \\ \|q - q'\| &= \frac{b\sigma + a\frac{a\sigma+c}{b}}{\sqrt{a^2+b^2}} = \frac{(a^2+b^2)\sigma + ac}{b\sqrt{a^2+b^2}} . \end{aligned}$$

Now since the triangles $\triangle vpp'$ and $\triangle vqq'$ are similar, and since x and σ have opposite signs, we have:

$$\frac{\|p - p'\|}{\|p'\|} = -\frac{\|q - q'\|}{\|q'\|} \; . \tag{A.3}$$

Substituting the expressions for these distances and reducing the $\sqrt{a^2 + b^2}$ factor, we obtain:

$$\begin{aligned}
\frac{bx - ay}{ax + by} &= -\frac{(a^2 + b^2)\sigma + ac}{bc} \; , \\
(ax + by)(a^2 + b^2)\sigma &= (a^2 + b^2)cx \; , \\
\sigma &= \frac{cx}{ax + by} \; . \qquad \text{(A.4)}
\end{aligned}$$

Having expressed σ in terms of x and y we can write the equation expressing the length of the ladder (namely, $\|p - q\|^2 = \ell^2$):

$$\begin{aligned}
(x - \sigma)^2 + \left(y + \frac{a\sigma + c}{b}\right)^2 &= \ell^2 \; , \\
b^2\left(x - \frac{cx}{ax + by}\right)^2 + \left(by + \frac{cx}{ax + by} + c\right)^2 &= b^2\ell^2 \; , \\
b^2(ax^2 + bxy - cx)^2 + (abxy + b^2y^2 + 2acx + bcy)^2 &= b^2\ell^2(ax + by)^2 \; . \qquad \text{(A.5)}
\end{aligned}$$

We found that the locus of p such that q touches E, and the interior of the ladder touches v is an algebraic curve of degree 4. In case v is not located at the origin, but at (x_v, y_v), one can simply substitute each x by $(x - x_v)$ and $(y - y_v)$ into the equation above, and the degree of the curve remains unchanged. Finally, since $x_v, y_v, a, b, c, \ell \in \mathbb{Q}$, its is clear that our curve has rational coefficients.

Our analysis generalizes the analysis given in [Lat91, SS83a], where E lies on the horizontal line $y = -d$, where d is the distance between v and E.

Observation A.1 *It is important to notice that in all cases, it is the* squared *length of the ladder ℓ takes part in the definition of a critical curve (see equations (A.2) and (A.5) above). It is therefore possible to relax the requirement that $\ell \in \mathbb{Q}$ and instead assume that only ℓ^2 is rational.*

A.2 Motion Planning for a General Polygon

The case of a general polygon which is free to translate and rotate amidst polygonal obstacles is more complicated. However, the same notions and techniques developed for the case of a ladder can be extended to deal with the general case.

As we did in the previous section, we assume that the obstacles are simple polygons, such that all obstacle vertices have rational coordinates. In addition, we are given a polygonal robot with m vertices, all of them having rational coordinates. One of the vertices will be denoted p and serve as our reference point, while the other robot vertices will be denoted

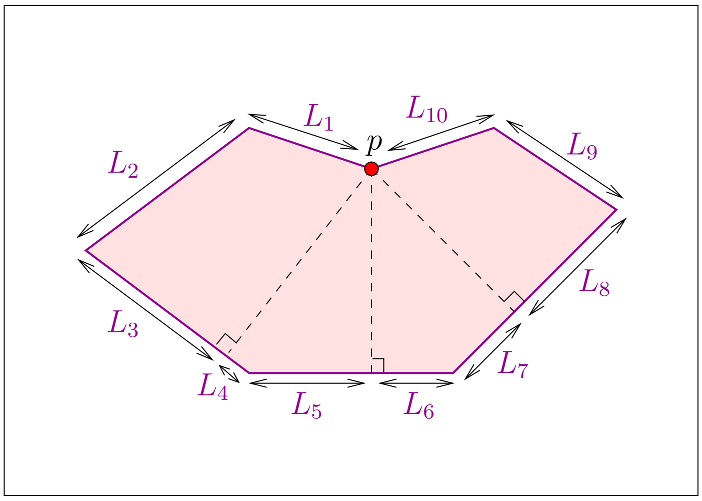


Figure A.3: The subdivision of a polygonal robot with 7 edges into 10 sub-edges with respect to the reference vertex p.

$q_1, \ldots, q_{m-1}$. A configuration $c = \langle x, y, \theta \rangle$ of the polygonal robot defines its placement uniquely, where the reference vertex p is located at (x, y) and θ is the angle the vector $\vec{pq_1}$ forms with the x-axis.

The robot has m edges, but in order to generalize the concept of a characteristic label for a cell, as developed in the previous section, we have to subdivide some of these edges into two sub-edges. A *sub-edge*[2] is a maximal continuous portion of an edge not containing the projection of the vertex p onto its supporting line in its interior. That is, for each edge $q_i q_{i+1}$ we draw a line perpendicular to the edge that passes through p. If this line intersects the interior of the segment $q_i q_{i+1}$ (that is, if both angles $\measuredangle p q_i q_{i+1}$ and $\measuredangle p q_{i+1} q_i$ are acute) we will use this intersection point to subdivide the edge into two sub-edges. Otherwise, the edge remains undivided and induces a single sub-edge. We label the sub-edges $L_1, L_2, \ldots, L_{m'}$ starting from p in a counterclockwise direction (m' is obviously bounded by $2(m-1)$, so the asymptotic complexity of the robot remains unchanged); see Figure A.3 for an illustration.

It is important to notice that even though we create new robot vertices, these vertices still have rational coordinates. Let us assume that we subdivide a robot edge L supported by the line $ax + by + c = 0$. We split this edge at the intersection point of this line with a line perpendicular to it that passes through $p = (x_p, y_p)$. As this perpendicular line is given by $bx - ay + (ay_p - bx_p) = 0$, the new vertex v' is the intersection of these two lines (recall that if L's endpoints are q_i and q_j then $a = y_{q_i} - y_{q_j}$, $b = x_{q_j} - x_{q_i}$ and $c = x_{q_i} y_{q_j} - x_{q_j} y_{q_i}$):

$$v' = \left(\frac{b(bx_p - ay_p) - ac}{a^2 + b^2} \, , \, \frac{a(ay_p - bx_p) - bc}{a^2 + b^2} \right) \, ,$$

and as a, b, c, x_p and y_p are all rational, v' is a point with rational coefficients.

As we did in the previous section for the ladder case, we define a set of critical curves based on the structure of the obstacles and the polygonal robot. When dealing with a

[2]Schwartz and Sharir used the term *half-edge*; as we reserve this term to DCEL components, we use a different term in the sequel.

polygonal robot the degenerate semi-free configurations that induce the critical curves are somewhat more complex. First, each obstacle edge defines a critical curve which is obviously a segment of a straight line with rational coefficients. In addition, the following critical curves are defined:

Type I: For each obstacle edge E and for each convex robot vertex $q \neq p$, the curve traced by p as q slides along E, such the line supporting the segment pq is perpendicular to E. This curve is a line segment parallel to E and at distance $||q - p||$ from it.

Type II: For each obstacle vertex v and for each robot vertex $q \neq p$, the curve traced by p as the robot rotates around v and q touches v. This curve is a circular arc centered at v and whose radius equals $||q - p||$.

Type III: For each obstacle edge E and two convex robot vertices $q_1, q_2 \neq p$ such that $||q_1 - q_2|| < ||E||$, the curve traced by p as q_1, q_2 slide along E. This curve is a line segment of length $||E|| - ||q_1 - q_2||$ that lies parallel to E.

Type IV: For each pair of convex obstacle edges E_1, E_2 and two convex robot vertices $q_1, q_2 \neq p$, the curve traced by p as q_1 slides along E_1 while q_2 slides along E_2. This curve is an elliptic arc — see Section A.2.1.

Type V: For each obstacle edge E, non-incident convex obstacle vertex v, convex robot vertex $q \neq p$ and robot sub-edge L, the curve traced by p as q slides along E while L touches v. These constraints define a segment of an algebraic curve of degree 4 — see Section A.2.2.

Type VI: For each pair of convex obstacle vertices v_1, v_2 and each robot sub-edge L not incident to p, and such that $||v_1 - v_2|| < ||L||$, the curve traced by p as L slides while touching both v_1 and v_2. This curve is a line segment of length $||L|| - ||v_1 - v_2||$ that lies parallel to the line connecting v_1 and v_2.

Type VII: For each pair of convex obstacle vertices v_1, v_2 and two robot sub-edges L_1, L_2, the curve traced by p as L_1 touches v_1 and L_2 touches v_2. This curve is a segment of an algebraic curve of degree 4 — see Section A.2.3.

Type VIII: For each obstacle edge E and each robot sub-edge L not incident to p, the curve traced by p as L slides along E. This curve is a line segment that lies parallel to E.

The supporting lines of critical curves of types I, III, VI and VIII can all be characterized as the locus of all points lying at some distance δ from a given line:

- A critical curve of type I is parallel to an obstacle edge E at a distance of $\delta = \|pq\|$.
- A critical curve of type III is parallel to an obstacle edge E, with δ being the distance between p and the line connecting the robot vertices q_1 and q_2.
- A critical curve of type VI is parallel to the line connecting the obstacle vertices v_1 and v_2, where δ is the distance of p from the robot sub-edge L.

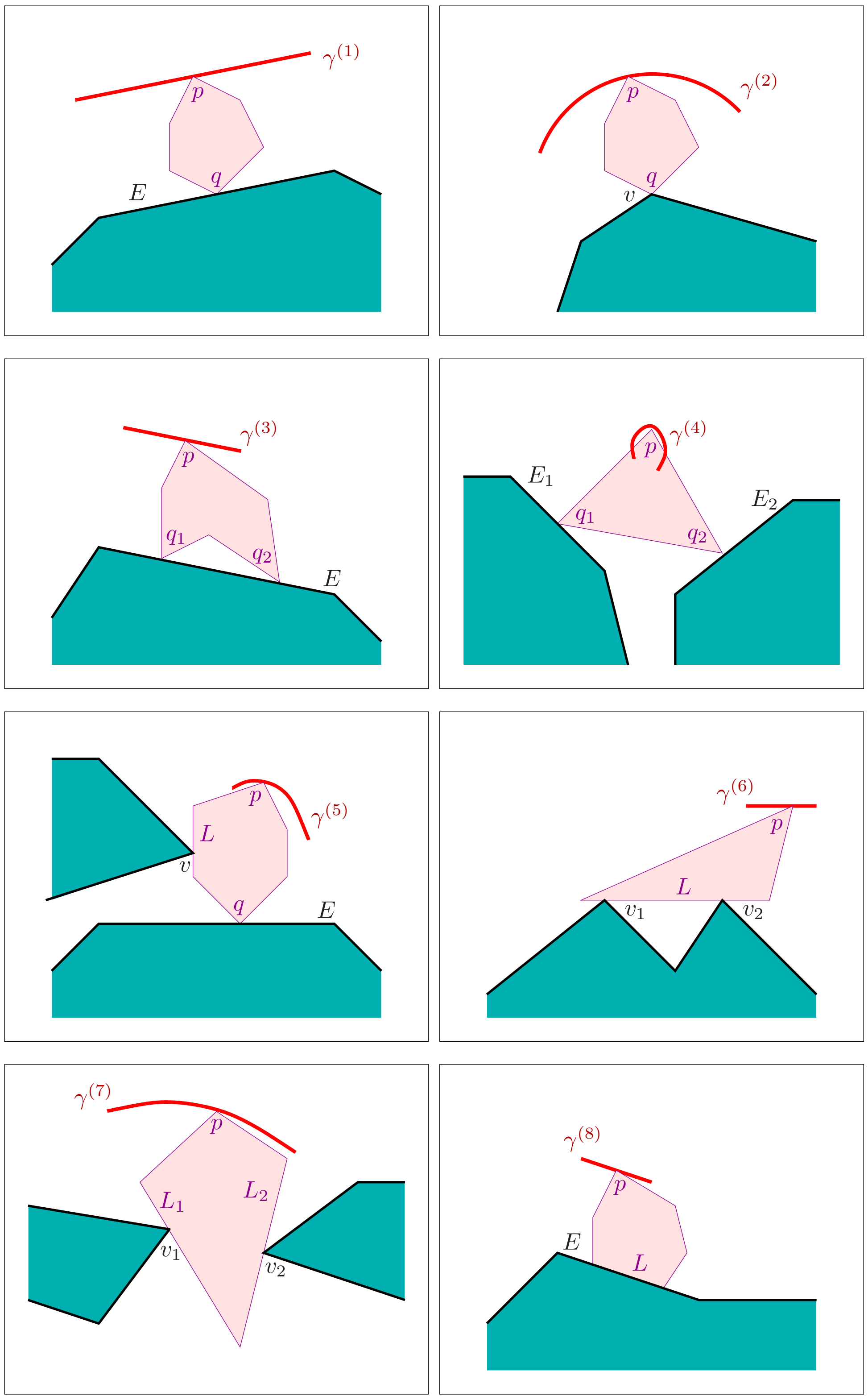


Figure A.4: The critical curves for translation and rotation of a polygon in the plane: $\gamma^{(1)}, \ldots, \gamma^{(8)}$ represent critical curves of types I–VIII, respectively.

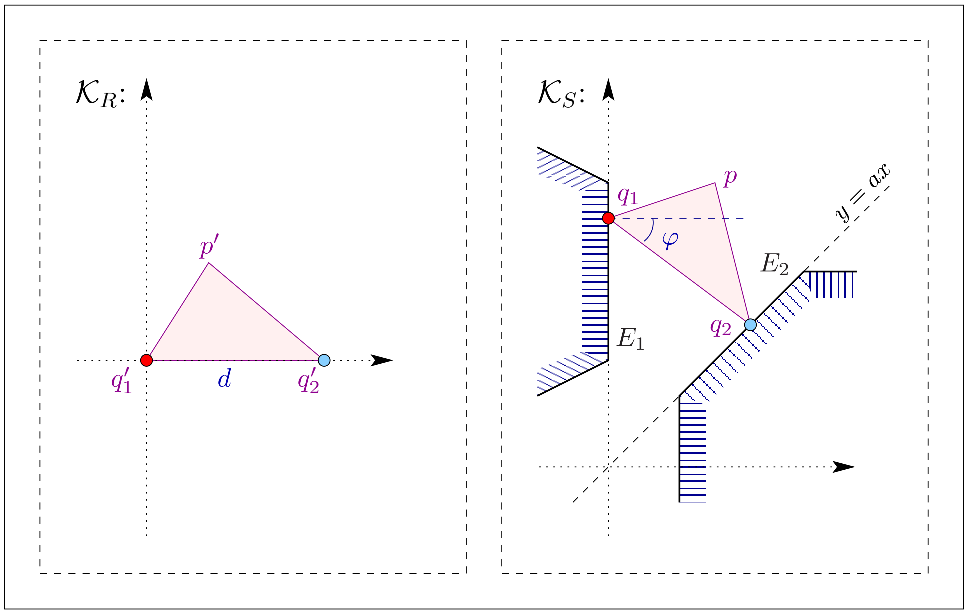


Figure A.5: Analysis of a critical curve of type IV.

- A critical curve of type VII is parallel to the obstacle edge E, lying at a distance δ that equals the distance of p from the robot sub-edge L.

In all cases, the base line defining the line supporting the critical curve is either an obstacle edge or a line connecting two obstacle vertices and therefore has rational coefficients. Moreover, the required *squared* distance δ^2 is also a rational number, as it is either the distance between the reference vertex p and another obstacle vertex, or the distance between p and a line with rational coefficients. We conclude that critical curves of types types I, III, VI and VIII are all segments of line-pairs with rational coefficients of the form of Equation (A.2) (where we use δ^2 instead of ℓ^2).

The analysis of a type II curve is very similar to the ladder case. We only have to replace the squared radius of the circle by $\|pq\|^2$, which is obviously a rational number.

We shall next analyze the algebraic representation of the critical curves of types IV, V and VII and parameterize them in terms of the robot features and the location of the obstacles. As we show next, all these critical curves are supported by algebraic curves with rational coefficients of degree 4 at most.

A.2.1 Critical Curves of Type IV

We are interested in the location (x, y) of the reference robot vertex p such that the convex robot vertex q_1 touches the obstacle edge E_1 and the convex robot vertex q_2 touches the obstacle edge E_2.

For our convenience, let us define a coordinate system $\mathcal{K}_R$ associated with the robot, such that q_1 is its origin and q_2 lies on it x-axis. If the distance between these two robot vertices is d, then the location of q_2 in $\mathcal{K}_R$ is $(0, d)$. Let us denote the location of the reference vertex

p in this coordinate system by (x_0, y_0). To avoid confusion, we shall refer to these locations as q_1', q_2' and p', respectively.

We shall also associate a second coordinate system $\mathcal{K}_S$ with the obstacle scene, such that E_1 coincides with its y-axis (that is, it lies on the line $x = 0$) and E_2 lies on a line passing through the origin (that is, it lies on the line $y = ax$ where a is a constant that depends only on the shape and placement of the obstacle in the scene). See Figure A.5 for an illustration.

Now let us define a rigid transformation $T : \mathcal{K}_R \to \mathcal{K}_S$ in the following manner: Pick a point on E_1, whose coordinates in $\mathcal{K}_S$ are given by $(0, \sigma)$ and move the robot so that q_1' coincides with it. Now rotate the robot around this point by φ until q_2' becomes in contact with E_2. We shall denote the coordinates of $q_2 = T(q_2')$ by $(\tau, a\tau)$. Let (x, y) be the location of p in $\mathcal{K}_S$ after this transformation.

It is clear that since T preserves distances, then $||q_2 - q_1|| = ||q_2' - q_1'||$, so we have:

$$\tau^2 + (a\tau - \sigma)^2 = d^2 \ . \tag{A.6}$$

Furthermore, since all angles are preserved under T, and in particular $\measuredangle p'q_1'q_2' = \measuredangle pq_1q_2$, the vector $\vec{q_1'p'}$ equals the vector $\vec{q_1p}$ rotated by $-\varphi$, where φ is the angle between the vector $\vec{q_1q_2}$ and $\mathcal{K}_S$'s x-axis. Thus:

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix} \begin{pmatrix} x \\ y - \sigma \end{pmatrix} \ . \tag{A.7}$$

But it is clear that (see Figure A.5):

$$\cos\varphi = \frac{\tau}{\sqrt{\tau^2 + (a\tau - \sigma)^2}} = \frac{\tau}{d} \ , \qquad \sin\varphi = \frac{a\tau - \sigma}{\sqrt{\tau^2 + (a\tau - \sigma)^2}} = \frac{a\tau - \sigma}{d} \ .$$

Hence (A.7) can be written as (note that we have inverted the rotation matrix):

$$\frac{1}{d} \begin{pmatrix} \tau & \sigma - a\tau \\ a\tau - \sigma & \tau \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} x \\ y - \sigma \end{pmatrix} \ . \tag{A.8}$$

We therefore obtain a linear system of equations in σ and τ:

$$\begin{cases} y_0\sigma + (x_0 - ay_0)\tau = dx \\ (d - x_0)\sigma + (ax_0 + y_0)\tau = dy \end{cases} \ . \tag{A.9}$$

For now we shall assume that the rank of this system is 2. (In the next subsection we focus on the degenerate case when its rank is 1.) Thus, the matrix of the system coefficients is not singular as $\left(x_0^2 + y_0^2 + d(ay_0 - x_0)\right) \neq 0$. To solve our linear system, we can simply invert the coefficient matrix, and compute σ and τ:

$$\begin{aligned} \begin{pmatrix} \sigma \\ \tau \end{pmatrix} &= \frac{1}{x_0^2 + y_0^2 + d(ay_0 - x_0)} \begin{pmatrix} ax_0 + y_0 & ay_0 - x_0 \\ x_0 - d & y_0 \end{pmatrix} \begin{pmatrix} dx \\ dy \end{pmatrix} = \\ &= \frac{d}{x_0^2 + y_0^2 + d(ay_0 - x_0)} \begin{pmatrix} a(x_0x + y_0y) + (y_0x - x_0y) \\ (x_0x + y_0y) - dx \end{pmatrix} \ . \end{aligned} \tag{A.10}$$

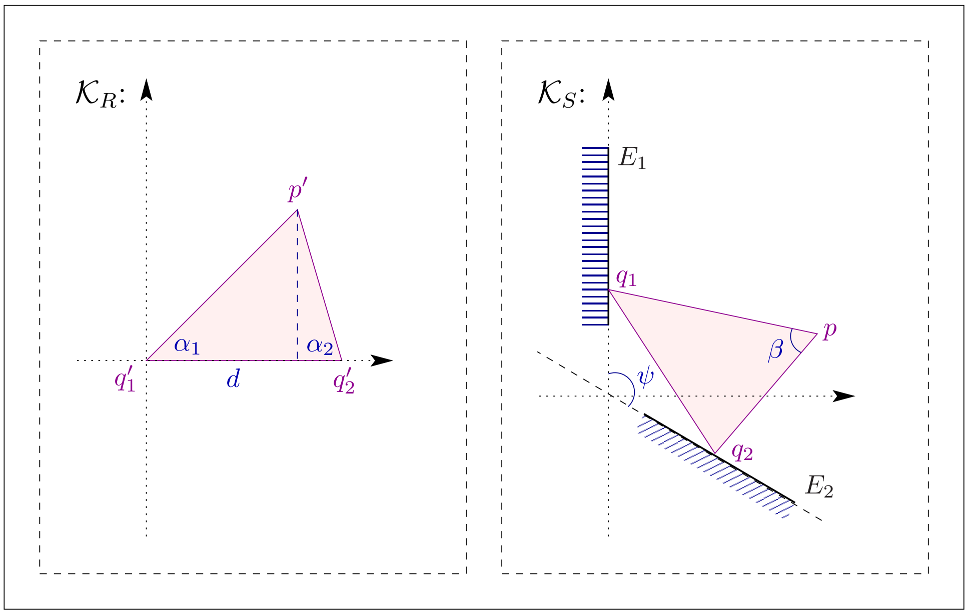


Figure A.6: Analysis of the degenerate case for a critical curve of type IV.

We can now substitute the expressions for σ and τ into Equation (A.6), and obtain (after multiplying by the denominator):

$$\begin{aligned} &d^2\big((x_0x + y_0y) - dx\big)^2 + d^2\big(a(x_0x + y_0y) - adx - a(x_0x + y_0y) - (y_0x - x_0y)\big)^2 = \\ &\qquad = d^2\big(x_0^2 + y_0^2 + d(ay_0 - x_0)\big)^2 \,. \end{aligned}$$

This expression can be simplified to yield the following equation of a conic curve — that is, an algebraic curve of degree 2:

$$\big((x_0x + y_0y) - dx\big)^2 + \big(adx + (y_0x - x_0y)\big)^2 = \big(x_0^2 + y_0^2 + d(ay_0 - x_0)\big)^2$$

$$\big((x_0 - d)^2 + (y_0 + ad)^2\big)x^2 + (x_0^2 - y_0^2)y^2 - 2d(ax_0 + y_0)xy = \big(x_0^2 + y_0^2 + d(ay_0 - x_0)\big)^2 \,. \quad \text{(A.11)}$$

It is possible to classify the conic curve defined by Equation (A.11) by looking at the sign of the following expression, involving the coefficients of x^2, y^2 and xy:

$$\begin{aligned} &4\big((x_0 - d)^2 + (y_0 + ad)^2\big)(x_0^2 - y_0^2) - \big(2d(ax_0 + y_0)\big)^2 = \\ &\qquad = 4\big(x_0^2 + y_0^2 + d(ay_0 - x_0)\big)^2 \,. \end{aligned}$$

Since $\big(x_0^2 + y_0^2 + d(ay_0 - x_0)\big) \neq 0$ (recall this is the determinant of the matrix of coefficients of the linear system (A.9)), this expression is always positive — thus we can conclude that the curve is an ellipse.

The Degenerate Case

We have already mentioned that in case that $x_0^2 + y_0^2 + d(ay_0 - x_0) = 0$, the rank of the system (A.9) is 1, so we cannot proceed and extract the expressions for σ and τ as we did in the general case.

First, let us examine when does such a degenerate case occur. Since we have $x_0^2 + y_0^2 = -d(ay_0 - x_0)$, then obviously:

$$a = -\frac{x_0(x_0 - d) + y_0^2}{dy_0} \ . \tag{A.12}$$

Since E_1 forms a right angle with the x-axis of $\mathcal{K}_S$, then the slope a of E_2's supporting line equals $\tan(\frac{\pi}{2} - \psi)$, where ψ is the angle between E_1 and E_2 (see Figure A.6). We therefore have:

$$\tan\psi = \cot(\frac{\pi}{2} - \psi) = \frac{1}{a} = -\frac{dy_0}{x_0(x_0 - d) + y_0^2} \ . \tag{A.13}$$

Now let us examine the triangle $\triangle q_1 q_2 p$, and let us denote by α_1, α_2 and β the triangle angles at q_1, q_2 and p respectively. By examining the triangle in $\mathcal{K}_R$ it is easy to see that (see Figure A.6):

$$\tan\alpha_1 = \frac{y_0}{x_0} \ , \qquad \tan\alpha_2 = \frac{y_0}{d - x_0} \ .$$

And so:

$$\tan\beta = \tan\big((\frac{\pi}{2} - \alpha_1) + (\frac{\pi}{2} - \alpha_2)\big) = \frac{\cot\alpha_1 + \cot\alpha_2}{1 - \cot\alpha_1\cot\alpha_2} = \frac{dy_0}{x_0(x_0 - d) + y_0^2} \ . \tag{A.14}$$

Since $\tan\psi = -\tan\beta$ and both angles must be less than π, we can conclude that the degenerate case occurs when $\psi = \pi - \beta$. In addition, as the rank of the equation system (A.9) is 1, the second equation in this system must equal the first equation times some constant factor. In particular, we obtain:

$$\frac{y_0}{d - x_0} = \frac{x}{y} \ , \tag{A.15}$$

so the critical curve in this degenerate case is a segment of the following line:

$$(d - x_0)x - y_0 y = 0 \ . \tag{A.16}$$

Moreover, we already know that the right term of Equation (A.15) above equals $\tan\alpha_2$, so $\tan\alpha_2 = \frac{x}{y}$, and we conclude that the angle between the vector connecting $\mathcal{K}_S$'s origin and p and the y-axis must equal α_2 (recall that $p = (x, y)$ in $\mathcal{K}_S$ — see Figure A.6).

Recall that $\psi = \pi - \beta = \alpha_1 + \alpha_2$. The critical curve therefore lies on the line cutting ψ into two angles that equal α_1 and α_2, respectively. However, in the next subsection we show that it is more convenient to represent the supporting curve as a conic curve also in the degenerate case. To this end, we multiply Equation (A.16) by the equation of a perpendicular line passing through $\mathcal{K}_S$'s origin, which is given by $y_0 x + (d - x_0)y = 0$. We therefore obtain the line-pair:

$$(d - x_0)y_0(x^2 - y^2) + \big((d - x_0)^2 - y_0^2\big)xy = 0 \ . \tag{A.17}$$

Analysis for Arbitrary Location

Up to this point, we have assumed that the robot and the obstacles are positioned in some canonical setting, in order to simplify our analysis and to find the algebraic degree of the critical curve. In order to obtain the true equation of the underlying ellipse in the original coordinate system of the workspace, we still have to apply some rigid transformations on the resulting curves. We next show that these rigid transformations provide us with the desired rational coefficients, while they obviously do not effect the algebraic degree of the curve.

So far we obtained the representation in Equation (A.11), where d is the distance between q_1 and q_2, (x_0, y_0) are the coordinates of p in the coordinate system $\mathcal{K}_R$ and a is the slope of the obstacle edge E_2 in the coordinate system $\mathcal{K}_S$.

Let us assume that the relevant robot vertices are given with the input coordinates (x_p, y_p), (x_{q_1}, y_{q_1}) and (x_{q_2}, y_{q_2}). We first note that $d = \sqrt{(x_{q_2} - x_{q_1})^2 + (y_{q_2} - y_{q_1})^2}$, so d is in general an irrational number whereas d^2 is obviously rational. To bring the vector $\vec{q_1q_2}$ to coincide with the x-axis, we have to translate the robot by $(-x_{q_1}, -y_{q_1})$ and to rotate it by $-\theta$, where θ is the angle between $\vec{q_1q_2}$ and the x-axis, using the rotation matrix $R_{-\theta}$. We thus have:

$$R_{-\theta} \;=\; \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix},$$

where:

$$\cos\theta = \frac{x_{q_2} - x_{q_1}}{d}, \qquad \sin\theta = \frac{y_{q_2} - y_{q_1}}{d}.$$

The coordinates of $p' = (x_0, y_0)$ can be therefore expressed as:

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = R_{-\theta} \begin{pmatrix} x_p - x_{q_1} \\ y_p - y_{q_1} \end{pmatrix} = \frac{1}{d} \begin{pmatrix} (x_{q_2} - x_{q_1})(x_p - x_{q_1}) + (y_{q_2} - y_{q_1})(y_p - y_{q_1}) \\ (y_{q_1} - y_{q_2})(x_p - x_{q_1}) + (x_{q_2} - x_{q_1})(y_p - y_{q_1}) \end{pmatrix}.$$

It is easy to show that if we substitute x_0 and y_0 into Equation (A.11) (or in Equation (A.17) in the degenerate case), we get an equation involving just d^2 and the input coordinates (x_p, y_p), (x_{q_1}, y_{q_1}) and (x_{q_2}, y_{q_2}).

We still have to express the slope a. Let us assume that the obstacle edges E_1 and E_2 are defined by the endpoints u_1, v_1 and u_2, v_2, respectively. The angles ω_1 and ω_2 these two edges form with the x-axis in the original coordinate system are therefore given by:

$$\cos\omega_1 = \frac{x_{v_1} - x_{u_1}}{\|E_1\|}, \qquad \sin\omega_1 = \frac{y_{v_1} - y_{u_1}}{\|E_1\|},$$
$$\cos\omega_2 = \frac{x_{v_2} - x_{u_2}}{\|E_2\|}, \qquad \sin\omega_2 = \frac{y_{v_2} - y_{u_2}}{\|E_2\|}.$$

The sine and cosine of the angle ω between the two edges are therefore:

$$\cos\omega \;=\; \cos(\omega_2 - \omega_1) = \frac{(x_{v_2} - x_{u_2})(x_{v_1} - x_{u_1}) + (y_{v_2} - y_{u_2})(y_{v_1} - y_{u_1})}{\|E_1\| \cdot \|E_2\|},$$

$$\sin\omega \;=\; \sin(\omega_2 - \omega_1) = \frac{(y_{v_2} - y_{u_2})(x_{v_1} - x_{u_1}) - (x_{v_2} - x_{u_2})(y_{v_1} - y_{u_1})}{\|E_1\| \cdot \|E_2\|}.$$

As E_1 coincides with the y-axis in $\mathcal{K}_R$, it follows that a is also a rational number, since we can express it as:

$$a = \tan(\frac{\pi}{2} - \omega) = \cot\omega = \frac{(x_{v_2} - x_{u_2})(x_{v_1} - x_{u_1}) + (y_{v_2} - y_{u_2})(y_{v_1} - y_{u_1})}{(y_{v_2} - y_{u_2})(x_{v_1} - x_{u_1}) - (x_{v_2} - x_{u_2})(y_{v_1} - y_{u_1})}.$$

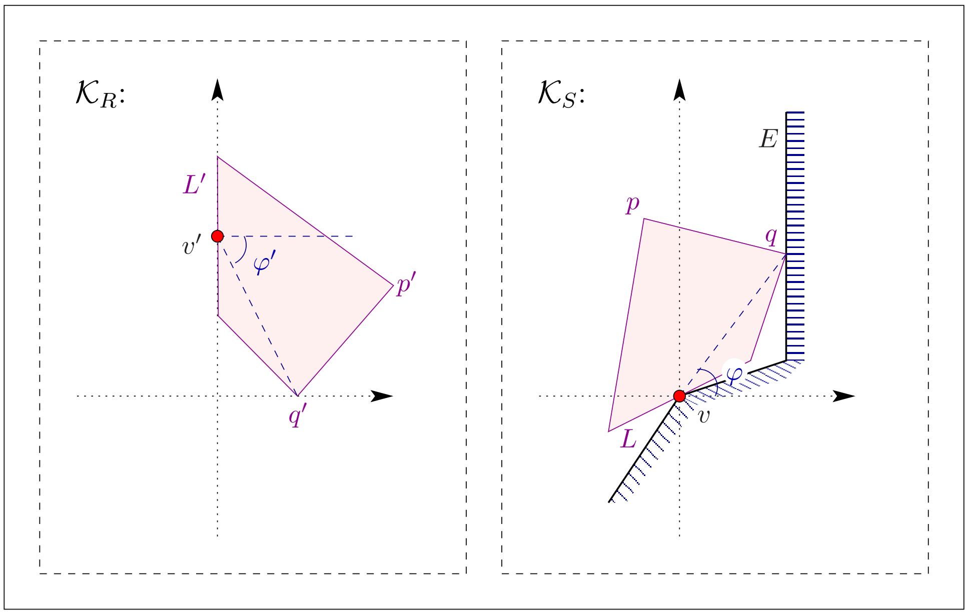


Figure A.7: Analysis of a critical curve of type V.

Up to this point, we showed that our ellipse has rational coefficients in the coordinate system $\mathcal{K}_S$. We still have to transform it back to the original coordinate system. We begin by rotating by an angle $\omega_1 - \frac{\pi}{2}$, namely by replacing x by $\hat{x}$ and y by $\hat{y}$, where:

$$\begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} = R_{\omega_1 - \frac{\pi}{2}} \begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{\|E_1\|} \begin{pmatrix} (y_{v_1} - y_{u_1})x + (x_{v_1} - x_{u_1})y \\ (y_{v_1} - y_{u_1})y - (x_{v_1} - x_{u_1})x \end{pmatrix} .$$

The length $\|E_1\|$ is irrational, but as all monomials in Equation (A.11) have an even degree (namely, we have the monomials containing x^2, y^2, xy and a free coefficient), the equation we get only involves $\|E_1\|^2$, which is a rational number. We finally translate the origin of $\mathcal{K}_R$ to the intersection point (x_t, y_t) between the lines supporting E_1 and E_2. This point clearly has rational coordinates, so the translation preserves the "rationality" of the ellipse coefficients.

A.2.2 Critical Curves of Type V

We are interested in the location (x, y) of the reference robot vertex p such that the convex robot vertex q touches the obstacle edge E and the robot sub-edge L touches the convex obstacle vertex v (where v is not an endpoint of E).

Let us first assume that q is not an endpoint of the L. We can then associate a coordinate system $\mathcal{K}_R$ with the robot, such that L coincides with its y-axis (that is, it lies on the line $x = 0$) and such that q lies on the x-axis and its coordinates are $(0, d)$, where d is the distance between q and the line containing L. Let (x_0, y_0) denote the coordinates of the vertex p in $\mathcal{K}_R$. For clarity, we shall denote these robot features L', q' and p' respectively in this coordinate system.

We also associate another coordinate system $\mathcal{K}_S$ with the obstacle, with v as its origin and with E being parallel to the y-axis, lying on a line whose equation is $x = \delta$, where δ is the distance between v and the line supporting E (see Figure A.7).

Now let us define a rigid transformation $T : \mathcal{K}_R \to \mathcal{K}_S$ in the following manner: Pick a point v' on L', whose coordinates in $\mathcal{K}_R$ are $(0, \sigma)$ and shift the robot such that it coincides with v. Then rotate the robot until q touches the obstacle edge E — at this point, its coordinates in $\mathcal{K}_S$ are (δ, τ). Let (x, y) be the location of p in $\mathcal{K}_S$ after this transformation.

Since the rigid transformation preserves distances, we have $||p - v|| = ||p' - v'||$ and $||q - v|| = ||q' - v'||$, hence we obtain:

$$x^2 + y^2 = x_0^2 + (y_0 - \sigma)^2 \ , \tag{A.18}$$

and:

$$\delta^2 + \tau^2 = d^2 + \sigma^2 \ . \tag{A.19}$$

Moreover, we can use the fact that $\measuredangle p'v'q' = \measuredangle pvq$, since the transformation T preserves angles. Hence, if we rotate the vector $\vec{v'p'}$ by $-\varphi'$ (where φ' is the angle between the vector $\vec{v'q'}$ and $\mathcal{K}_R$'s x-axis) we will get an identical vector to the one obtained by the rotation of $\vec{vp}$ by $-\varphi$ (where φ is the angle between the vector $\vec{vq}$ and $\mathcal{K}_S$'s x-axis). That is, we can write:

$$\begin{pmatrix} \cos\varphi' & \sin\varphi' \\ -\sin\varphi' & \cos\varphi' \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 - \sigma \end{pmatrix} = \begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \ . \tag{A.20}$$

The sines and cosines of φ and φ' are given by (see Figure A.7):

$$\cos\varphi' = \frac{d}{\sqrt{d^2 + \sigma^2}} \ , \qquad \sin\varphi' = \frac{-\sigma}{\sqrt{d^2 + \sigma^2}} \ ,$$

$$\cos\varphi = \frac{\delta}{\sqrt{\delta^2 + \tau^2}} \ , \qquad \sin\varphi = \frac{\tau}{\sqrt{\delta^2 + \tau^2}} \ ,$$

so Equation (A.20) can be rewritten as:

$$\frac{1}{\sqrt{d^2 + \sigma^2}} \begin{pmatrix} d & -\sigma \\ \sigma & d \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 - \sigma \end{pmatrix} = \frac{1}{\sqrt{\delta^2 + \tau^2}} \begin{pmatrix} \delta & \tau \\ -\tau & \delta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \ . \tag{A.21}$$

Using the equality (A.19) we can eliminate the denominators and obtain the following system:

$$\begin{cases} dx_0 - \sigma(y_0 - \sigma) = \delta x + \tau y \\ \sigma x_0 + d(y_0 - \sigma) = \delta y - \tau x \end{cases} \ . \tag{A.22}$$

In order to eliminate τ from this system, we can multiply the first equation by x and add to it the second equation, multiplied by y. We then get:

$$x_0(dx + \sigma y) + (y_0 - \sigma)(dy - \sigma x) = \delta(x^2 + y^2) \ . \tag{A.23}$$

We now wish to eliminate σ^2 from the equation and obtain a linear equation in σ. To this end we multiply Equation (A.18) by x and subtract Equation (A.23). We obtain:

$$(x - \delta)(x^2 + y^2) = x(x_0^2 + y_0^2 - dx_0) - dy_0 y + \sigma(dy - x_0 y - y_0 x) \ , \tag{A.24}$$

so we can now express σ as:

$$\sigma = \frac{(x-\delta)(x^2+y^2)+(x_0(d-x_0)-y_0^2)+dy_0y}{(d-x_0)y-y_0x} \ . \tag{A.25}$$

Substituting this expression back into Equation (A.18) we get:

$$x^2+y^2 = x_0^2 + \left(y_0 - \frac{(x-\delta)(x^2+y^2)+(x_0(d-x_0)-y_0^2)+dy_0y}{(d-x_0)y-y_0x}\right)^2 \ ,$$

$$\begin{aligned}(x^2+y^2)\left((d-x_0)y-y_0x\right)^2 &= x_0^2\left((d-x_0)y-y_0x\right)^2 + \\ +\big((d-x_0)y_0y-y_0^2x &- (x-\delta)(x^2+y^2)+(y_0^2-x_0(d-x_0))-dy_0y\big)^2 \ ,\end{aligned}$$

$$\begin{aligned}(x^2+y^2)\left((d-x_0)y-y_0x\right)^2 &= \\ = \underbrace{x_0^2\left((d-x_0)y-y_0x\right)^2}_{\text{I}} &+ \left((x-\delta)(x^2+y^2)+\underbrace{x_0\big((d-x_0)x+y_0y\big)}_{\text{II}}\right)^2 \ .\end{aligned} \tag{A.26}$$

One can easily notice that the terms in (A.26) that do not contain the factor x^2+y^2 all result in $\text{I}+\text{II}^2$, where:

$$\begin{aligned}\text{I}+\text{II}^2 = x_0^2(d-x_0)^2y^2 - 2x_0^2(d-x_0)y_0xy + x_0^2y_0^2x^2 + \\ +x_0^2(d-x_0)^2x^2 + 2x_0^2(d-x_0)y_0xy + x_0^2y_0^2y^2 &= \\ &= x_0^2\left((d-x_0)^2+y_0^2\right)(x^2+y^2) \ .\end{aligned}$$

It is therefore possible to reduce Equation (A.26) by the factor x^2+y^2 to obtain the equation of an algebraic curve of degree 4:

$$\begin{aligned}\left((d-x_0)y-y_0x\right)^2 &= (x-\delta)^2(x^2+y^2) + \\ &\quad +2x_0(x-\delta)\left((d-x_0)x+y_0y\right) + \\ &\quad +x_0^2\left((d-x_0)^2+y_0^2\right) \ .\end{aligned} \tag{A.27}$$

We mention that in case that q is an endpoint of L, we have $d=0$ and the coordinate system $\mathcal{K}_R$ is defined such that L lies on the y-axis and q is the origin. The equation of the critical curve in this case is obtained by substituting $d=0$ into Equation (A.27):

$$(x_0y+y_0x)^2 = (x-\delta)^2(x^2+y^2) + 2x_0(x-\delta)(y_0y-x_0x) + x_0^2(x_0^2+y_0^2) \ . \tag{A.28}$$

Analysis for Arbitrary Location

In Equation (A.27) we expressed the critical curve in terms of: (i) δ, the distance between the obstacle vertex v and the obstacle edge E, (ii) d, which is the distance between the robot vertex q and the sub-edge L, and (iii) (x_0, y_0), the coordinates of p in the coordinate system $\mathcal{K}_R$.

Let us assume that the endpoints of the robot sub-edge L are r_1 and r_2. We can write the distance of p from the line containing L as:

$$d = \frac{|(y_{r_1} - y_{r_2})x_p + (x_{r_2} - x_{r_1})y_p + (x_{r_1}y_{r_2} - x_{r_2}y_{r_1})|}{\sqrt{(y_{r_1} - y_{r_2})^2 + (x_{r_2} - x_{r_1})^2}} \ . \tag{A.29}$$

Notice that the denominator of this expression equals $\|L\|$.

Let (x_t, y_t) be the intersection point of a line perpendicular to L going through p with the line supporting the sub-edge L. As we previously showed, this point can be computed from the coordinates of p, r_1 and r_2 and has rational coordinates. To obtain the coordinate system $\mathcal{K}_R$ we first have to translate the coordinates by $(-x_t, -y_t)$ and then rotate it using the rotation matrix $R_{\frac{\pi}{2}-\theta}$, where θ is the angle between L and the x-axis:

$$R_{\frac{\pi}{2}-\theta} \ = \ \begin{pmatrix} \sin\theta & \cos\theta \\ -\cos\theta & \sin\theta \end{pmatrix} ,$$

where:

$$\cos\theta = \frac{x_{r_2} - x_{r_1}}{\|L\|} \ , \qquad \sin\theta = \frac{y_{r_2} - y_{r_1}}{\|L\|} \ .$$

We thus have:

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = R_{\frac{\pi}{2}-\theta} \begin{pmatrix} x_p - x_t \\ y_p - y_t \end{pmatrix} = \frac{1}{\|L\|} \begin{pmatrix} (y_{r_2} - y_{r_1})(x_p - x_t) + (x_{r_2} - x_{r_1})(y_p - y_t) \\ (x_{r_1} - x_{r_2})(x_p - x_t) + (y_{r_2} - y_{r_1})(y_p - y_t) \end{pmatrix} \ .$$

It is easy to show that if we substitute d, x_0 and y_0 into Equation (A.27), we get an equation involving just $\|L\|^2$ and the input coordinates (x_p, y_p), (x_{r_1}, y_{r_1}) and (x_{r_2}, y_{r_2}). It is not difficult to prove a similar result for the special case of Equation (A.28).

In an analogous manner, let us assume that the endpoints of the obstacle edge E are u_1 and u_2. The distance between v and the line containing E is:

$$\delta = \frac{|(y_{u_1} - y_{u_2})x_v + (x_{u_2} - x_{u_1})y_v + (x_{u_1}y_{u_2} - x_{u_2}y_{u_1})|}{\sqrt{(y_{u_1} - y_{u_2})^2 + (x_{u_2} - x_{u_1})^2}} \ . \tag{A.30}$$

Notice that the denominator of this expression equals $\|E\|$.

We now have to change the coordinate system $\mathcal{K}_R$ to the original coordinate system. We first have to perform rotation using the rotation matrix $R_{\omega-\frac{\pi}{2}}$, where ω is the angle between the edge E and the x-axis in the original coordinate system, and the sine and cosine of this angle obviously have $\|E\|$ as their denominator (see the case of curves of type IV).

Let us examine what happens when we substitute the rotated coordinates and the distance δ into Equation (A.27). If we consider the monomials in δ, x and y, then we have only monomials of an even degree. As a result, our substitution will yield coefficients that involve only $\|E\|^2$, which is a rational number. We finally translate the origin of $\mathcal{K}_S$ to the location of the vertex $v = (x_v, y_v)$, preserving the "rationality" of the curve coefficients as v has rational coordinates.

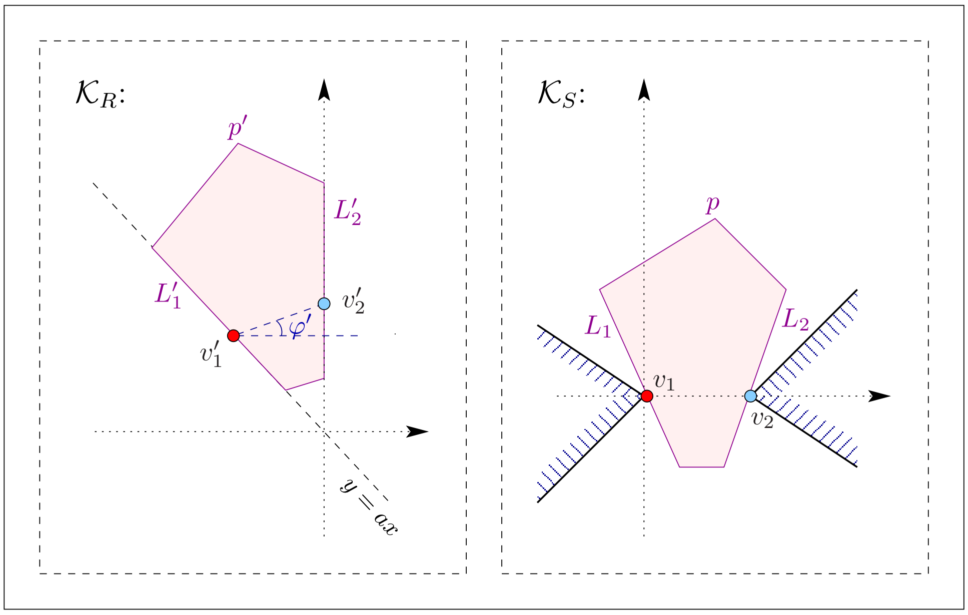


Figure A.8: Analysis of a critical curve of type VII.

A.2.3 Critical Curves of Type VII

We are interested in the location (x, y) of the reference robot vertex p such that the two robot sub-edges L_1 and L_2 touch the two convex obstacle vertices v_1 and v_2, respectively.

As we did in our analysis of the previous curve types, let us associate a coordinate system $\mathcal{K}_R$ with the robot, such that L_2 coincides with its y-axis (that is, it lies on the line $x = 0$) and such that L_1 lies on a line that passes through $\mathcal{K}_R$'s origin (that is, it lies on the line $y = ax$, where a is a constant that depends only on the shape of the robot). Let (x_0, y_0) denote the coordinates of the vertex p in $\mathcal{K}_R$. For clarity, we shall denote the relevant robot features in this coordinate system by L'_1, L'_2 and p'.

We also associate another coordinate system $\mathcal{K}_S$ with the obstacles, such that v_1 coincides with its origin and v_2 lies on the x-axis. Thus, if d is the distance between the two vertices, v_2 is located at $(0, d)$ in $\mathcal{K}_S$ (see Figure A.8).

Now let us define a rigid transformation $T : \mathcal{K}_R \rightarrow \mathcal{K}_S$ in the following manner: Pick a point v'_1 on L'_1, whose coordinates in $\mathcal{K}_R$ are $(\tau, a\tau)$ and shift the robot such that it coincides with v_1. Then rotate the robot around that point until L_2 touches the obstacle vertex v_2. Let $v'_2 = (0, \sigma)$ be the point of L'_1 which is mapped by T to v_2, and let (x, y) be the location of p in $\mathcal{K}_S$ after this transformation.

Since the rigid transformation preserves distances, we have $||p - v_1|| = ||p' - v'_1||$ and $||v_2 - v_1|| = ||v'_2 - v'_1||$, hence we obtain:

$$x^2 + y^2 = (x_0 - \tau)^2 + (y_0 - a\tau)^2 \ , \tag{A.31}$$

and:

$$d^2 = \tau^2 + (\sigma - a\tau)^2 \ . \tag{A.32}$$

Since the transformation T is angle preserving, we know that $\measuredangle p'v_1'v_2' = \measuredangle pv_1v_2$. Hence, if we rotate the vector $\vec{v_1'p'}$ by $-\varphi'$ (where φ' is the angle between the vector $\vec{v_1'v_2'}$ and $\mathcal{K}_R$'s x-axis) we will get an identical vector to $\vec{v_1p}$ (notice that there is no need to further rotate this vector, since $\vec{v_1v_2}$ is the direction of the x-axis). That is, we can write:

$$\begin{pmatrix} \cos\varphi' & \sin\varphi' \\ -\sin\varphi' & \cos\varphi' \end{pmatrix} \begin{pmatrix} x_0 - \tau \\ y_0 - a\tau \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} . \tag{A.33}$$

As the sine and cosine of φ' are given by (recall that $d = \sqrt{\tau^2 + (\sigma - a\tau)^2}$):

$$\cos\varphi' = \frac{-\tau}{\sqrt{\tau^2 + (\sigma - a\tau)^2}} = \frac{-\tau}{d} , \qquad \sin\varphi' = \frac{\sigma - a\tau}{\sqrt{\tau^2 + (\sigma - a\tau)^2}} = \frac{\sigma - a\tau}{d} ,$$

Equation (A.33) can be rewritten as:

$$\begin{pmatrix} -\tau & \sigma - a\tau \\ a\tau - \sigma & -\tau \end{pmatrix} \begin{pmatrix} x_0 - \tau \\ y_0 - a\tau \end{pmatrix} = d \begin{pmatrix} x \\ y \end{pmatrix} . \tag{A.34}$$

Let us now multiply Equation (A.34) by the matrix $\begin{pmatrix} x_0 - \tau & y_0 - a\tau \\ a\tau - y_0 & x_0 - \tau \end{pmatrix}$. Notice that this matrix and the matrix that appears on the left-hand side of Equation (A.34) both correspond to rotational transformations, hence their multiplication is commutative. It is easy to verify that $(a\tau - y_0)(x_0 - \tau) + (x_0 - \tau)(y_0 - a\tau) = 0$, and using Equation (A.31) we obtain the following:

$$\begin{pmatrix} -\tau & \sigma - a\tau \\ a\tau - \sigma & -\tau \end{pmatrix} \begin{pmatrix} x^2 + y^2 \\ 0 \end{pmatrix} = d \begin{pmatrix} x_0 - \tau & y_0 - a\tau \\ a\tau - y_0 & x_0 - \tau \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} , \tag{A.35}$$

hence we obtain the following system of linear equations:

$$\begin{cases} -\tau(x^2 + y^2) = d(x_0x + y_0) - d(x + ay)\tau \\ (a\tau - \sigma)(x^2 + y^2) = d(x_0y - y_0x) + d(ax - y)\tau \end{cases} . \tag{A.36}$$

Notice that σ does not appear in the first equation, and we can readily obtain an expression for τ:

$$\tau = \frac{d(x_0x + y_0y)}{d(x + ay) - (x^2 + y^2)} . \tag{A.37}$$

Substituting τ back into Equation (A.31) we get:

$$x^2 + y^2 = \left(x_0 - \frac{d(x_0x + y_0y)}{d(x + ay) - (x^2 + y^2)}\right)^2 + \left(y_0 - \frac{ad(x_0x + y_0y)}{d(x + ay) - (x^2 + y^2)}\right)^2 ,$$

$$\begin{aligned} &(x^2 + y^2)\left(d(x + ay) - (x^2 + y^2)\right)^2 = \\ &= \left(\underbrace{d(ax_0 - y_0)y}_{\text{I}} - x_0(x^2 + y^2)\right)^2 + \left(\underbrace{d(y_0 - ax_0)x}_{\text{II}} - y_0(x^2 + y^2)\right)^2 . \end{aligned} \tag{A.38}$$

As we did in the previous section for curves of type V, we shall examine all the terms of Equation (A.38) that do not contain the factor $x^2 + y^2$. It is easy to see that they all result in $\mathrm{I}^2 + \mathrm{II}^2$, where:

$$\mathrm{I}^2 + \mathrm{II}^2 = \left(d(ax_0 - y_0)y\right)^2 + \left(d(y_0 - ax_0)x\right)^2 = d^2(ax_0 - y_0)^2(x^2 + y^2) \ .$$

It is therefore possible to reduce Equation (A.38) by the factor x^2+y^2 and obtain an equation of an algebraic curve of degree 4:

$$\begin{aligned} \left(d(x + ay) - (x^2 + y^2)\right)^2 \ = \ & (x_0^2 + y_0^2)^2(x^2 + y^2) + \\ & +2d(ax_0 - y_0)(y_0x - x_0y) + \\ & +d^2(ax_0 - y_0)^2 \ . \end{aligned} \tag{A.39}$$

Analysis for Arbitrary Location

We shall now transform the curve given in Equation (A.39) to the original coordinate system, using similar techniques to those we used for curves of type IV and V.

We first note that the slope a depends on the tangent of the angle between the two supporting lines of L_1 and L_2, and is a rational expression that involves the coordinates of the endpoints of the two sub-edges, as given in the reference coordinate system for the robot (see Equation (A.30) for a similar expression).

Moreover, if we denote the intersection between the supporting lines lines of L_1 and L_2 by (x_t, y_t), we can express x_0 and y_0 using the the following expression, where r_1 and r_2 denote the endpoints of L_1 and θ is the angle its supporting line forms with the x-axis:

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = R_{\frac{\pi}{2}-\theta} \begin{pmatrix} x_p - x_t \\ y_p - y_t \end{pmatrix} = \frac{1}{\|L_1\|} \begin{pmatrix} (y_{r_2} - y_{r_1})(x_p - x_t) + (x_{r_2} - x_{r_1})(y_p - y_t) \\ (x_{r_1} - x_{r_2})(x_p - x_t) + (y_{r_2} - y_{r_1})(y_p - y_t) \end{pmatrix} \ .$$

Note that all monomials in x_0 and y_0 of Equation (A.39) are of an even degree, so if we substitute the expressions for x_0 and y_0 we obtained above into this equation, our curve equation will only contain $\|L_1\|^2$, which is a rational number.

We finally apply a rigid transformation to convert $\mathcal{K}_S$ to our original coordinate system. To do this, we first rotate by ω, which is the angle between the vector $v_1\vec{v}_2$ and the x-axis in the original coordinate system. Clearly:

$$\cos\omega = \frac{x_{v_2} - x_{v_1}}{\sqrt{(x_{v_2} - x_{v_1})^2 + (y_{v_2} - y_{v_1})^2}} = \frac{x_{v_2} - x_{v_1}}{d} \ , \qquad \sin\omega = \frac{y_{v_2} - y_{v_1}}{d} \ .$$

As all monomials in x, y and d in Equation (A.39) are of an even degree, this results in coefficients that only involve d^2, which is a rational number. We finally translate the origin to v_1, and obtain a curve of degree 4 with rational coefficients.

Appendix B

Sweeping Curves and Maintaining 2D Arrangements on Surfaces

In this appendix we describe how the classes and algorithms in the arrangement package of CGAL, which we have overviewed in Chapter 2, can be extended to represent more diverse topologies. We first describe the extension for unbounded curves, then go one step further and describe how we can construct and maintain arrangement of arbitrary curves embedded on a 3D parametric surface. Namely, given a parametric surface S in $\mathbb{R}^3$ and a set $\mathcal{C}$ of curves that all lie on this surface, we compute the subdivision these curves induce on S.

The ability to construct and maintain arrangement of curves defined on a surface in a robust manner makes it possible to develop software solutions in diverse application fields. See Section 7.2 for more details.

A *parametric surface* S is a surface defined by a parametric equation involving two parameters u and v, namely:

$$f_S(u, v) = (x(u, v),\, y(u, v),\, z(u, v)) \ . \tag{B.1}$$

Thus, $f_S : \mathbb{P} \longrightarrow \mathbb{R}^3$ and $S = f_S(\mathbb{P})$, where $\mathbb{P}$ is a continuous and simply connected two-dimensional parameter space. We use the surface parameterization in order to sweep over a set of curves embedded on the surface in the two-dimensional uv-plane, rather than considering the three-dimensional image of $\mathbb{P}$.

B.1 The Augmented Sweep-Line Algorithm

Our goal is to generalize the Bentley–Ottmann algorithm [BO79], as described in Section 2.4.1, and perform the sweep procedure over a parametric surface $S = f_S(\mathbb{P})$ in its parameter space. However, to conveniently do so, we must consider a subspace of $\mathbb{P}$. Sweeping over the entire parameter space raises, in general, several difficulties either when the parameter space is unbounded, or when there is no inverse mapping from the surface to the parameter space. We eliminate these difficulties by cutting out portions of the parameter space and symbolically keeping track of these modifications.

We next formally define three aspects that require special attention when generalizing the sweep procedure. In all cases, S is a parametric surface defined over $\mathbb{P}$ in the uv-plane. We give the definitions using the u-parameter; the definitions with respect to the v-parameter are similar.

Definition B.1 (Infinite boundary) *Let $\hat{u}$ be one of the values defining the u-range of $\mathbb{P}$ ($\hat{u}$ may be finite or $\hat{u} = \pm\infty$). We say that the surface has an* infinite boundary *in u if:*

$$\forall v \quad \lim_{u \to \hat{u}} f_S(u, v) = \pm\infty \ .$$

Definition B.2 (Curve of discontinuity) *If u is defined over a bounded parameter range $[u_{\min}, u_{\max})$ such that:*

$$\forall v \quad \lim_{u \to u_{\max}} f_S(u, v) = f_S(u_{\min}, v) \ ,$$

then the curve defined by $f_S(u_{\min}, v)$ forms a curve of discontinuity *in u on the surface S.*

Definition B.3 (Singularity point) *We say that a point $f_S(u_0, v_0) = p_0 \in S$ is a* singularity point *in u, if u_0 is either the maximum or the minimum of the u-parameter range in $\mathbb{P}$ (i.e., either $u_0 = u_{\min}$ or $u_0 = u_{\max}$), and for each $\delta > 0$ we have:*

$$\forall v \quad \exists u \quad \| f_S(u, v) - p_0 \| < \delta \ .$$

Consider, for example, the xy-plane, which can be parameterized by $\mathbb{P} = \mathbb{R}^2$ and $f_S = (u, v, 0)$. An alternative representation is $\mathbb{P} = (-\frac{\pi}{2}, \frac{\pi}{2}) \times (-\frac{\pi}{2}, \frac{\pi}{2})$ and $f_S = (\tan u, \tan v, 0)$. In both cases, the surface has an infinite boundary in the minimal and the maximal values of u *and* in the minimal and maximal values of v.

Let us examine the canonical 3D cylinder of radius r, given by the implicit representation $x^2 + y^2 = r^2$. It can be parameterized for $\mathbb{P} = [-\pi, \pi) \times \mathbb{R}$ such that $f_S(u, v) = (r \cos u, r \sin u, v)$. We have $f_S(-\pi, v) = (-r, 0, v) = \lim_{u \to \pi} f_S(u, v)$, so in this case the cylinder contains a line of discontinuity that is parallel to the z-axis and passes through $(-r, 0, 0)$.

The unit sphere, which can be parameterized over $\mathbb{P} = [-\pi, \pi) \times [-\frac{\pi}{2}, \frac{\pi}{2}]$ using $f_S(u, v) = (\cos u \sin v, \sin u \sin v, \cos v)$, contains a semicircle of discontinuity that connects the two poles $(0, 0, -1)$ and $(0, 0, 1)$ through $(-1, 0, 0)$. In addition, the two poles are singularity points in v: for instance, for each u if we take $v > \arcsin(1 - \frac{\delta}{2})$ we get that the distance of $f_S(u, v)$ from the north pole $(0, 0, 1)$ is $2(1 - \sin v) < \delta$.

Given a surface containing curves of discontinuity and singularity points we modify the parameter space as follows: In case of discontinuity in u (similarly, in v), we consider the open u-range $(u_{\min} + \varepsilon, u_{\max} - \varepsilon)$ for an infinitesimally small $\varepsilon > 0$. In case of a singularity point in $u_{\min}$ we augment the u-parameter range to be lower bounded by $u_{\min} + \varepsilon$ (or upper bounded by $u_{\max} - \varepsilon$ in case of a singularity point in $u_{\max}$), for an infinitesimally small $\varepsilon > 0$; we handle singularities in v in a similar fashion. As a result, we obtain an augmented parameter space $\tilde{\mathbb{P}}$, for which it is possible to define the inverse mapping $f_S^{-1} : \mathbb{R}^3 \longrightarrow \tilde{\mathbb{P}}$.

It is now possible to apply an augmented sweep-line algorithm to our parametric surface, where we perform a plane sweep over $\tilde{\mathbb{P}}$. Let $\tilde{S}$ denote the image of the augmented parameter space, namely $f_S(\tilde{\mathbb{P}})$. Given a set $\mathcal{C}$ of curves defined on S, we start by computing $C' = C \cap \tilde{S}$ for each $C \in \mathcal{C}$, and by subdividing C' into u-monotone subcurves. We refer to the resulting subcurves as *sweepable curves*. Note that in particular, the interior of a sweepable curve cannot intersect a curve of discontinuity or contain a singularity point. However, the curve-ends may be incident to the modified surface boundaries.

We start the sweep with the curve $f_S(u_0, v)$, for some initial fixed u-value u_0 (e.g., $u_0 = u_{\min} + \varepsilon$ in the example of the cylinder). We now sweep the curve over the surface $\tilde{S}$. For each u-value u', a subset of the sweepable curves induced by $\mathcal{C}$ intersect the sweep-curve $f_S(u', v)$, at the points $p_1, \ldots, p_k \in S$. The status structure stores these curves ordered in ascending v of $f_S^{-1}(p_1), \ldots, f_S^{-1}(p_k)$. Similarly, when we detect an intersection point p, we insert it into the event queue, considering the lexicographic uv-order of $f_S^{-1}(p)$. For the proper maintenance of the algorithm the event queue must contain — along with events that represent regular curve endpoints and intersection points — events associated with curve-ends incident to the surface boundaries.

In the next subsection we give an overview of our augmented sweep-line algorithm for the case of unbounded planar curves. In Section B.1.2 we show that this algorithm is actually a special case of sweeping over the augmented parameter space, and generalize it to the case of general orientable parametric surfaces. We give special attention to the detection and handling of curve-ends that are incident to the modified surface boundaries.

B.1.1 Sweeping Unbounded Curves

The main difficulty in applying the Bentley–Ottmann sweep algorithm on a set of unbounded planar curves lies in the initialization step. When dealing with bounded curve segments, we insert all curve endpoints into a event queue, where they are sorted by an ascending xy-lexicographic order. As unbounded curves do not have valid endpoints, we have to augment the algorithm.

One approach that comes to mind is to extend the definition of a point and allow *points at infinity*. To maintain the order of the event queue, we have to support lexicographical comparison on these extended points. This is quite straightforward in some cases, but more intricate in others. For instance, if we have to compare two points lying at $x = -\infty$, we need to compare the vertical position of the *curves* that induce these points at $x = -\infty$. This means that a point at infinity is implicitly associated with a curve, and the lexicographic comparison of two points may involve computations with their underlying curves. A similar approach was taken in Exacus [BEH+05] to make the sweep-line algorithm of Cgal (version 3.1) operate on unbounded algebraic curves.

The main advantage of this approach is that the sweep-line algorithm remains the same and needs no alteration. On the other hand, the burden of comparisons at infinity falls on the traits class, which is undesirable in our case. Recall that we have a single generic implementation for the sweep-line algorithm, which can be instantiated with different traits classes (see Section 2.4.1). It therefore makes more sense to handle the infinite curve-ends

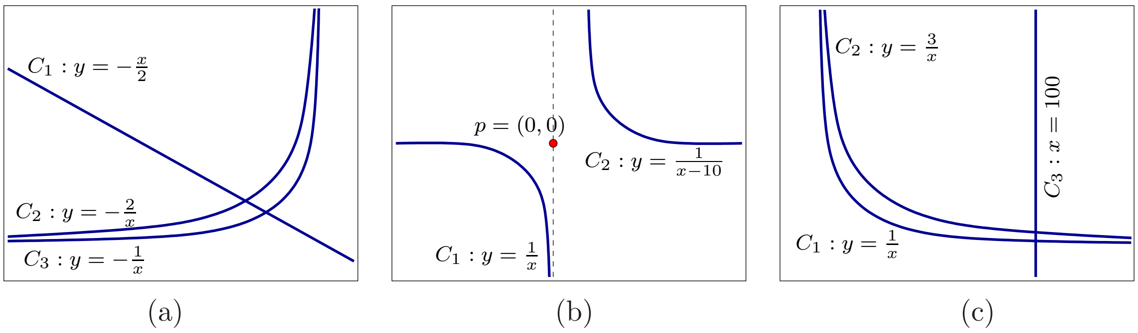


Figure B.1: Comparing unbounded curve-ends: (a) Comparing curve-ends at $x = -\infty$. Note that C_1 is obviously above C_2 and C_3, but C_2 is considered to by above C_3 as there exists x_0 such that for each $x < x_0$, $C_2(x) > C_3(x)$. (b) Comparing the horizontal positions of a curve-end and a point. The vertical curve that passes through p is to the right of C_1's right end and to the left of C_2 left end. (c) Comparing the horizontal positions of curve-ends at $y = \pm\infty$. Note that C_1's left end is to the left of C_2's left end and both are to the left of C_3.

at a single centralized place and slightly modify the sweep-line algorithm, allowing a simpler interface and logic for the various traits classes.

Let us revise the terminology we used when we introduced the arrangement-traits concepts in Section 2.3.1. Instead of talking about endpoints of an x-monotone curve, we refer to the two *curve-ends*. A curve-end may be unbounded or bounded, and only in the latter case we have a valid endpoint. In addition, we require any model of the *ArrangementBasicTraits_2* concept to support the following predicates involving unbounded curve-ends:

Infinity type: Given a curve-end (the left end or the right end of the x-monotone curve C), determine whether it lies at $x = \pm\infty$. If the curve-end has a finite x-coordinate, determine whether it lies at $y = \pm\infty$, or whether it also has a finite y-coordinate.[1]

Compare y at infinity: Given the unbounded left ends of C_1 and C_2, both defined at $x = -\infty$, compare the vertical positions of C_1 and C_2 at $x = -\infty$ (similarly for unbounded right ends at $x = \infty$).

Compare x at infinity (I): Given a curve C whose end lies at $y = \pm\infty$ and a point p, compare the horizontal position of C's end and the vertical line that contains p.

Compare x at infinity (II): Given two curves C_1 and C_2 whose ends lie at $y = \pm\infty$, compare the horizontal positions of the two curve-ends.

The last three comparison operations should consider the asymptotic behavior of the curve-ends, as shown in Figure B.1. Thus, two curve-ends are equal at infinity only in case of overlap between two curves. Naturally, there are traits classes that do not support unbounded curves and need not implement the extra predicates. To this end, each traits class should define a tag that signals whether it supports unbounded curves or not. The extra predicates listed above are only required when this tag is defined as a *true* tag.

[1]For example, the left end of the hyperbolic branch $y = \frac{1}{x}, x > 0$ has a bounded x-coordinate but lies at $y = \infty$. Typically, vertical lines and vertical asymptotes share this behavior.

Having defined the additional traits-class operations, we are ready to modify the sweep-line algorithm to handle infinite curves. First, we store extra information with the events: an event may be associated with a (finite) point, or it may be associated with an unbounded curve-end and lie at $x = \pm\infty$ or at $y = \pm\infty$. We begin the sweep process by constructing events that represent all unbounded curve-ends and all finite endpoints. To sort these events we use a simple procedure based on the basic comparison functions implemented by the arrangement-traits class: if the two events are associated with finite points, we simply compare these points; if one event lies at $x = -\infty$ and the other is a finite point, then the first event is obviously smaller; if both events lie at $x = -\infty$ we compare their associates curve-ends there, etc.

Once the event queue is initialized the main loop of the sweep-line process begins. Note that the first events that are extracted from the event queue are all associated with unbounded left curve-ends, and are already given in the correct increasing vertical order of these curves at $x = -\infty$. Thus, as long as we handle events at $x = -\infty$, we can insert the corresponding curve at the top of the status line, *without having to perform additional geometric comparisons*. Similarly, when the sweep-line process advances to some finite x-coordinate and handles an event that lies at $y = -\infty$ (or $y = +\infty$) that represents an unbounded curve-end, we already know that the curve lies below (above) all other curves currently intersecting the sweep line. Thus, we simply insert the corresponding curve at the bottom (top) end of the status line *without performing any geometric comparisons*. We note that intersection points are always finite, therefore we do not modify the way we handle intersection events.

Consider the example depicted in Figure B.2(a), where we sweep over the lines $\ell_1 : y = \frac{x}{2}$, $\ell_2 : y = 0$ and over the two branches of the hyperbola $y = -\frac{1}{x}$, denoted h_1 and h_2, respectively. The order of the events in this case is: $\min(\ell_1)$, $\min(\ell_2)$, $\min(h_1)$, $\max(h_1)$, $\min(h_2)$, $\max(h_2)$, $\max(\ell_2)$, $\max(\ell_1)$, where $\min(c)$ and $\max(c)$ refer to the minimal and maximal ends of the curve c, respectively. When the intersection point at the origin is discovered, it is inserted into the event queue between $\max(h_1)$ and $\min(h_2)$.

To summarize, adapting the sweep-line process to handle unbounded curves helps keeping the interface of the traits class simple, so the correct handling of unbounded curve-ends is done in a single centralized place, namely in the modified sweep-line procedure. In addition, we benefit from reducing the number of geometric operations the algorithm needs to perform. As most traits classes employ the exact computation paradigm and perform exact geometric computations, these geometric operations can be quite costly, so reducing their number is essential in order to control the running-time of the application.

B.1.2 Sweeping on General Surfaces

The additional advantage, and perhaps the most significant one, of the approach we have just described, is that it enables an elegant generalization of the sweep-line procedure for sweeping over curves embedded on a surface in $\mathbb{R}^3$.

So far we swept over the parameter space $\mathbb{P} = \mathbb{R}^2$, and treated curve-ends that coincide with the infinite boundaries symbolically. It is not difficult to verify that the same set of geometry-traits operations required in the planar case also applies when sweeping over a set

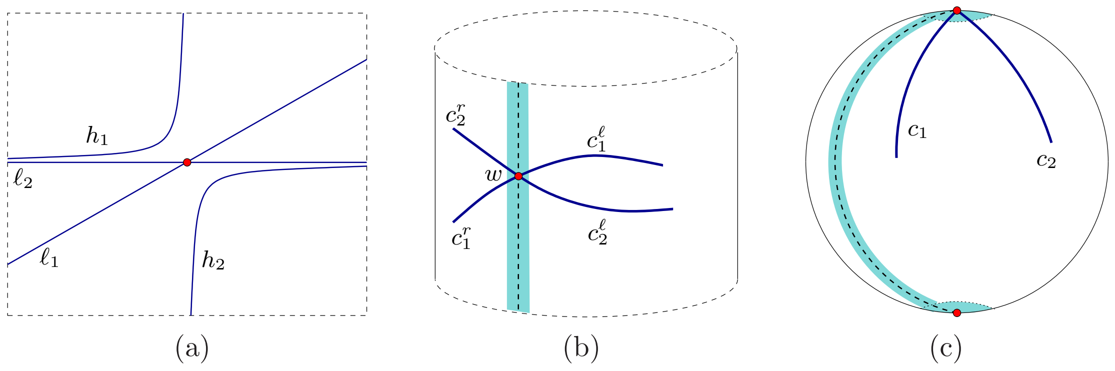


Figure B.2: Comparing curve-ends with boundary conditions: (a) Comparing at infinity. (b) Comparing near the line of discontinuity. (c) Comparing near a singularity point.

of curves on a surface. However, we have to re-interpret the geometric predicates as if they are given in the uv-plane.

As mentioned before, we begin by subdividing each input curve into sweepable subcurves. Note that each sweepable subcurve C in defined over a continuous range of u-values in $\tilde{\mathbb{P}}$, and we consider the curve as the graph of the function $v = C(u)$ where $f_S(u, v) \in C$. A sweepable curve-end may have *boundary conditions*. In the previous subsection we have already encountered curves with unbounded ends, and we say that the boundary condition in x (or in y) of such a curve-end is of type *minus infinity* or *plus infinity*. In the general case, we may also encounter curve-ends whose boundary condition is *leaving discontinuity* (or *approaching discontinuity*), or *leaving singularity* (or *approaching singularity*). By *leaving* a singularity we mean that a singularity point is defined by the minimal value of the u-range (or the v-range) of the parameter space, so we view its incident curves as if they begin in an ε-environment after the singularity. Similarly, by *approaching* a singularity we mean that a singularity point is defined by the maximal value of the u-range (or the the v-range).

The rest of the geometry-traits primitives involve only sweepable curves and regular points, namely points that do not coincide with the boundaries of the augmented surface $\tilde{S}$. We next list the new formulations of the traits-class operations (compare with Section 2.3.1 and with Section B.1.1 above):

Compare u: Given two points $p_1, p_2 \in \tilde{S}$ compare the u-values of $f_S^{-1}(p_1)$ and $f_S^{-1}(p_2)$.

Compare uv: Compare $f_S^{-1}(p_1)$ and $f_S^{-1}(p_2)$ lexicographically, by their u-values, and in case of equality by their v-values.

Boundary type: Given a curve-end (the minimal or the maximal end of a sweepable curve C), determine whether it has a boundary condition in u (infinity, discontinuity or singularity). If it has no boundary condition in u, determine if the curve-end has a boundary condition in v.

Min/max endpoint: Return the uv-lexicographically smaller, or the uv-lexicographically larger, endpoint of a given sweepable curve, with the precondition that the corresponding curve-end is *not* unbounded.

Compare v at u: Given a sweepable curve C and a point $p_0 = f_S(u_0, v_0)$ such that u_0 is in the u-range of C, compare v_0 and $C(u_0)$.

Compare v to right: Given two sweepable curves C_1 and C_2 that intersect at a given point $p_0 = f_S(u_0, v_0)$ (note that $u_0 > u_{min}$), compare $C_1(u')$ and $C_2(v')$, for $u' > u_0$ such that for any other intersection point $p_1 = f_S(u_1, v_1)$ of the curves we have $u' < u_1$.

Intersect: Compute the intersection points of two given sweepable curves C_1 and C_2.

Compare v at boundary: Given two curve-ends of C_1 and C_2, both having a boundary condition at $\hat{u}$ ($\hat{u}$ may also be $\pm\infty$), compare $\lim_{u \to \hat{u}} C_1(u)$ and $\lim_{u \to \hat{u}} C_2(u)$.

Compare u at boundary (I): Given a curve-end of a sweepable curve C with a boundary condition at $\hat{v}$ and a point $p_0 = f_S(x_0, y_0)$, compare $\lim_{v \to \hat{v}} C_1^{-1}(v)$ and u_0.

Compare u at boundary (II): Given two curve-ends of the sweepable curves C_1 and C_2, having boundary conditions at $\hat{v}_1$ and at $\hat{v}_2$, respectively, compare $\lim_{v \to \hat{v}_1} C_1^{-1}(v)$ and $\lim_{v \to \hat{v}_2} C_2^{-1}(v)$.

We note that even though the traits-class operations refer to the parameter space, it is possible to carry out the necessary geometric and algebraic operations wherever it is most convenient — e.g., on the surface itself, or on some projected image of S. These operations need not be done in parameter-space coordinates.

Let us consider the canonical cylinder depicted in Figure B.2(b). As we remove the line of discontinuity, we have $\tilde{\mathbb{P}} = (-\pi, \pi) \times \mathbb{R}$ (where $f_S(u, v) = (r \cos u, r \sin u, v)$). In this case, all curve-ends may start right after the line of discontinuity or may end right before this line. The two curves C_1 and C_2 are split at the line of discontinuity, forming the sweepable curves c_1^ℓ, c_1^r and c_2^ℓ, c_2^r, respectively. Yet when we compare the curve-ends we consider an ε-neighborhood around the line of discontinuity (shaded). Thus, c_1^ℓ is above c_2^ℓ after the line of discontinuity (when the sweep starts), and c_1^r lies below c_2^r immediately before this line (when the sweep ends) — meaning that we actually consider four distinct events. Observe that if we wish to implement a sweep-line visitor (see Section 2.4.1) that detects all intersection points induced by a set of curves on a sphere, we can easily compute the intersection points that occur on the line of discontinuity and are not detected as such by the sweep-line procedure, by simply examining the adjacencies between curves incident to the line of discontinuity.

Let us now consider the case of a pole. By the definition of a singularity point on a parametric surface (see Definition B.3), had we not removed singularity points from the surface, a curve that coincides with a singularity point would have always been contained in the status structure. However, by removing an infinitesimally small neighborhood around the pole we make sure that all sweepable curves enter and leave the status structure during the sweep process.

Consider for instance Figure B.2(c). We symbolically handle curve-ends that are incident to a singularity point (the north pole of a sphere in this case): c_1 lies to the left of c_2, as we compare the ends of sweepable curves in an ε-neighborhood below the north pole (shaded). Note that this means that we have a different event for every curve-end that coincides with a pole. Once again, a sweep-line visitor can properly process these events, and report them as a single intersection point, if necessary.

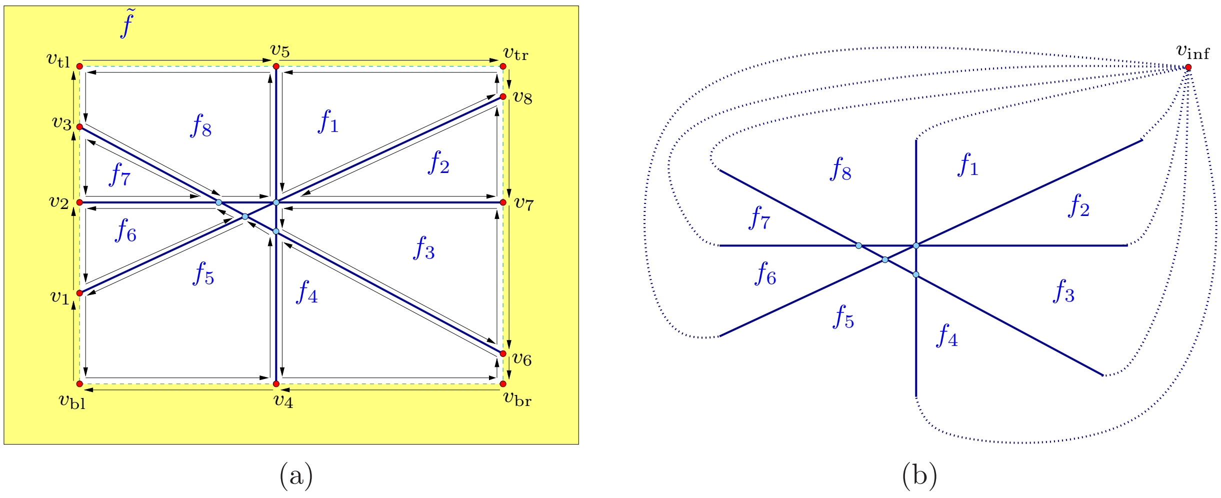


Figure B.3: Possible DCEL representations of an arrangement of four lines: (a) Using an implicit bounding rectangle. The face denoted $\tilde{f}$ (lightly shaded) is the fictitious face, which lies outside the imaginary rectangle (dashed) that bounds the actual arrangement. The vertices $v_1, \ldots v_8$ lie at infinity, while v_{bl}, v_{tl}, v_{br} and v_{tr} are fictitious vertices that represent the four corners of the imaginary bounding rectangle. (b) Using a single vertex at infinity v_{inf}, with all unbounded curve-ends being incident to this vertex.

B.2 Constructing Arrangements on Surfaces

As explained in Section 2.4.1, it is possible to devise various algorithms that are all based on the sweep-line framework by implementing an appropriate *visitor* class and instantiating the `Sweep_line_2` class-template accordingly. So far we have just considered a sweep-line visitor that computes the intersection points induced by a set of curves. However, the augmented sweep-line algorithm can be used for constructing and maintaining arrangements on surfaces, if only we can supply the appropriate visitor class. Note that as the modifications of the original sweep-line algorithm involve curve-ends with boundary conditions, we only need augment the visitor that handles bounded curves (see Section 2.4.1) so that it properly inserts such curve-ends into the DCEL.

B.2.1 The Topology-Traits Concept

It is straightforward to represent an arrangement of *bounded* planar curves using the DCEL structure, where the DCEL contains a single unbounded face. However, already when moving to unbounded curves we should consider alternative representations of the arrangement. Figure B.3(a) demonstrates one possibility, where we use an implicit bounding rectangle embedded in the DCEL structure. First of all, we allow vertices to be located *at infinity*, where a vertex at infinity is associated with an unbounded curve-end rather than being associated with a point (as finite vertices are). The vertices at infinity lie on the edges of the imaginary boundary rectangle and split them into pairs of fictitious halfedges. These edges are called *fictitious* as they do not represent any concrete planar curve. Each face whose boundary contains at least one fictitious edge is considered to be unbounded. In addition, the DCEL also contains one exterior face, which contains the bounding rectangle as a hole in its interior. It is *fictitious*, as it does not represent any two-dimensional portion of the plane bounded by the input curves, and we do not regard it as part of the arrangement.

It is possible to choose a different representation of a planar arrangement of unbounded curves that uses a single vertex at infinity v_{inf}, such that all unbounded curve-ends are incident to this vertex; see Figure B.3(b) for an illustration. Note that in the former representation, an empty arrangement is represented by four fictitious edges that form an empty bounding rectangle. These edges are eventually split, as unbounded curves are inserted into the arrangement. Using the representation of a vertex at infinity, an empty arrangement contains just a single unbounded face, where v_{inf} is created upon the insertion of the first unbounded curve. To insert an additional unbounded curve into the arrangement, we have to locate its unbounded end(s) around v_{inf}. We mention that all *finite* vertices, edges, and faces are identically represented in both options.

We extend the classical definition of the DCEL structure, by introducing vertices at infinity and fictitious features. We also have to extend the representation of a face to support more general surfaces. Consider the portion of the canonical 3D cylinder $x^2 + y^2 = r^2$ sandwiched between the planes $z = 0$ and $z = 1$. This portion forms a *perimetric face* whose outer boundary comprises two connected components. However, extending the DCEL structure to support the representation of such faces is fairly straightforward. So far we have supported DCEL faces with multiple holes, also known as *inner components* of the face boundary (see Section 2.2); we now also have to support faces with multiple *outer components*.

Aiming for modularity, we wish to decouple the implementation of the basic arrangement operations (e.g., inserting a new edge associated with a subcurve, removing an edge, etc.) from the actual representation of the arrangement. We do this by introducing the concept of a *topology-traits class*, which encapsulates the topology of the surface on which the arrangement is embedded, and determines the underlying DCEL representation of the arrangement. It does so by supplying predicates and operations related to curve-ends with boundary conditions. For example, it is responsible for initializing a DCEL structure that represents an empty arrangement, and for locating the DCEL feature that represents a given curve-end. In case no such DCEL feature exists (e.g., when inserting the first unbounded curve and v_{inf} is not yet created), the topology-traits class is responsible for creating it.

Using the topology-traits primitives, the sweep-line visitor is capable of constructing the arrangement of a set of curves on a surface. When the visitor is notified on a subcurve with boundary conditions, it queries the topology-traits class to obtain the DCEL feature containing the curve-end, then inserts the subcurve accordingly. For example, if we sweep over the cylinder depicted in Figure B.2(b), a vertex w is created on the line of discontinuity when we insert c_1^ℓ into the arrangement. The topology-traits class keeps track of this vertex, so it will associate w as the minimal end of c_2^ℓ and as the maximal ends of c_1^r and c_2^r. Similarly, in the example shown in Figure B.2(c), the north pole will eventually be represented as a single DCEL vertex, with c_1 and c_2 incident to it.

According to our software design, future releases of the CGAL arrangement package will contain a class template named `Arrangement_on_surface_2<GeomTraits,TopTraits>`. This class is parameterized with a geometry-traits class, which encapsulates the geometry of the curves the arrangement handles (so far we referred to it simply as a *traits class*), and a topology-traits class that defines the topology of the surface on which the arrangement is embedded. We mention that the two parameters are not entirely decoupled, as the geometry-traits class needs to be aware of the topology of the surfaces on which its curves are defined.

In order to maintain backward compatibility, we will still supply the class-template `Arrangement_2<GeomTraits,Dcel>`, which represents a planar arrangement by instantiating the `Arrangement_on_surface_2` template with its geometry-traits class and the with a default topology-traits class. The topology-traits class may be one that handles only a bounded planar topology (when the geometry-traits class defines the relevant tag as a *false* tag — see Section B.1.1), or a topology-traits class that handles unbounded curves as well, using an imaginary bounding rectangle.

B.2.2 Implementation Details

The arrangement package included in the next version of CGAL (the forthcoming Version 3.3) will support planar arrangements of unbounded curves. We use the imaginary bounding rectangle approach, described in the previous subsection (see also Figure B.3(a)), to represent arrangements of such curves. We note that this representation is somewhat similar to the usage of an *infimaximal frame* (namely, a dynamic bounding box) proposed by Mehlhorn and Seel [MS03]. However, our representation has several advantages over the infimaximal-frame approach: First, it requires less algebraic operations, which can be very costly when using exact computing with non-linear curves.[2] Secondly, our approach makes much of the logic of handling the curve-ends centralized in one place and saves code duplication for different types of curves. In addition to a new geometry-traits class for linear curves in the plane (lines, rays and line segments), the conic-traits class and the traits class for rational functions (see Section 2.3.3) will be extended to handle unbounded curves as well.

In addition to the topology-traits classes for curves on the plane, we intend to provide two additional topology-traits classes. The first handles a spherical topology, and will come with a geometry-traits class for handling arcs of great circles. A favorable property of these curves is that they can be handled using only exact rational arithmetic, yet they have useful applications; see, e.g., [GHH$^+$03].

We also intend to provide geometry-traits class and a topology-traits class that handle arrangements of curves on a quadric surface, where the curves correspond to the intersections of this quadric with other quadric surfaces. A *quadric surface* is defined by the zero-set of a trivariate polynomial of degree 2 at most. The geometry-traits class will be based on the work of Berberich *et al.* [BHK$^+$05], who implemented a (planar) geometry-traits class that can handle the projection of intersection curves of two quadric surfaces onto the plane. These projected intersections are algebraic curves of degree 4, and it is possible to implement the operations on the 3D curves by considering their projected images. We note that while some

[2]The infimaximal approach is general and can be applied to arbitrary planar curves. The paper [MS03] however describes experiments with linear objects only.

quadric surfaces, such as hyperbolic paraboloids[3] and planes (degenerate forms of quadric surfaces) are xy-monotone, other surfaces such as ellipsoids or parabolic paraboloids are not, and we should consider their *upper* part and their a *lower* part separately. The quadric-topology traits-class will encapsulate this separation and will also perform the "stitching" of the curves that reside on the upper part of the surface and the ones that lie on its lower part, in order to form a single arrangement on the entire surface.

[3]See, e.g., ⟨`http://mathworld.wolfram.com/HyperbolicParaboloid.html`⟩.

Bibliography

[ABF89] F. Avnaim, J.-D. Boissonnat, and B. Faverjon. A practical exact motion planning algorithm for polygonal object amidst polygonal obstacles. In *Geometry and Robotics*, volume **391** of *LNCS*, pages 67–86. Springer, 1989.

[ACA01] E. U. Acar, H. Choset, and P. N. Atkar. Complete sensor-based coverage with extended-range detectors: A hierarchical decomposition in terms of critical points and Voronoi diagrams. In *Proc. IEEE/RSJ Internat. Conf. Intell. Robot. Sys. (IROS)*, pages 1305–1311, 2001.

[AFH02] 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.

[AK00] F. Aurenhammer and R. Klein. Voronoi diagrams. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 201–290. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[AS00a] 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.

[AS00b] P. K. Agarwal and M. Sharir. Pipes, cigars, and kreplach: The union of Minkowski sums in three dimensions. *Discrete and Computational Geometry*, **24**(4):645–685, 2000.

[Ata85] M. J. Atallah. Some dynamic computational geometry problems. *Computers and Mathematics with Applications*, **11**(12):1171–1181, 1985.

[Aus98] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Addison-Wesley, 1998.

[Bañ90] J. Bañon. Implementation and extension of the ladder algorithm. In *Proc. IEEE Internat. Conf. Robot. Auto. (ICRA)*, pages 1548–1553, 1990.

[BBP01] 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.

[BEH$^+$02] 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 Europ. Sympos. Alg. (ESA)*, volume **2461** of *LNCS*, pages 174–186. Springer, 2002.

[BEH$^+$05] 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 Europ. Sympos. Alg. (ESA)*, volume **3669** of *LNCS*, pages 155–166. Springer, 2005.

[BFH$^+$07] E. Berberich, E. Fogel, D. Halperin, K. Mehlhorn, and R. Wein. Sweeping and maintaining two-dimensional arrangements on surfaces: A first step. In *Proc. 15th Europ. Sympos. Alg. (ESA)*, October 2007. To appear.

[BFHW07] E. Berberich, E. Fogel, D. Halperin, and R. Wein. Sweeping and maintaining two-dimensional arrangements on surfaces. In *Proc. 23rd Europ. Workshop Comp. Geom. (EWCG)*, pages 223–226, 2007.

[BFM$^+$01] C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. In *Proc. 9th Europ. Sympos. Alg. (ESA)*, volume **2161** of *LNCS*, pages 254–265. Springer, 2001.

[BHK$^+$05] 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 Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 99–106, 2005.

[BK05] S. Bereg and D. G. Kirkpatrick. Curvature-bounded traversals of narrow corridors. In *Proc. 21st Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 278–287, 2005.

[BL03] J.-D. Boissonnat and S. Lazard. A polynomial-time algorithm for computing a shortest path of bounded curvature amidst moderate obstacles. *Internat. J. Computational Geometry and Applications*, **13**(3):189–229, 2003.

[BMK$^+$03] E. L. J. Bohez, N. T. H. Minh, B. Kiatsrithanakorn, P. Natasukon, H. Ruei-Yun, and L. T. Son. The stencil buffer sweep plane algorithm for 5-axis CNC tool path verification. *Computer-Aided Design*, **35**(12):1129–1142, October 2003.

[BMS97] U. Bartuschka, K. Mehlhorn, and S.Näher. A robust and efficient implementation of a sweep line algorithm for the straight line segment intersection problem. In *Proc. 1st Workshop Alg. Eng. (WAE)*, pages 124–135, 1997.

[BNS00] U. Bartuschka, S. Näher, and M. Seel. A generic plane sweep framework, 2000. Unpublished manuscript.

[BO79] J. L. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Computers*, **28**(9):643–647, 1979.

[BPR96] S. Basu, R. Pollack, and M.-F. Roy. Computing roadmaps of semi-algebraic sets. In *Proc. 28th Annu. ACM Sympos. Theory Comput. (STOC)*, pages 168–173, 1996.

[BSM03] M. Balasubramaniam, S. E. Sarma, and K. Marciniak. Collision-free finishing toolpaths from visibility data. *Computer-Aided Design*, **35**(4):359–374, April 2003.

[But06] Z. Butler. Corridor planning for natural agents. In *Proc. IEEE Internat. Conf. Robot. Auto. (ICRA)*, pages 499–504, 2006.

[BZ88] B. K. Bhattacharya and J. Zorbas. Solving the two-dimensional findpath problem using a line-triangle representation of the robot. *J. Algorithms*, **9**:449–469, 1988.

[Can86] J. F. Canny. Collision detection for moving polyhedra. *IEEE Trans. Pattern Analysis and Machine Intelligence*, **8**(2):200–209, March 1986.

[Can87] J. F. Canny. *The Complexity of Robot Motion Planning.* ACM – MIT Press Doctoral Dissertation Award Series. MIT Press, Cambridge, MA, 1987.

[Can88] J. F. Canny. Some algebraic and geometric computations in PSPACE. In *Proc. 20th Annu. ACM Sympos. Theory Comput. (STOC)*, pages 460–469, 1988.

[CD85] B. Chazelle and D. P. Dobkin. Optimal convex decompositions. In G. T. Toussaint, editor, *Computational Geometry*, pages 63–133. North-Holland, Amsterdam, The Netherlands, 1985.

[CEGS91] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. A singly-exponential stratification scheme for real semi-algebraic varieties and its applications. *Theoretical Computer Science*, **84**:77–105, 1991.

[CER01] E. Cohen, G. Elber, and R. F. Riesenfeld. *Geometric Modeling with Splines: An Introduction.* A. K. Peters, 2001.

[CGL85] B. Chazelle, L. J. Guibas, and D.-T. Lee. The power of geometric duality. *BIT*, **25**:76–90, 1985.

[CL06] F. Cazals and S. Loriot. Computing the exact arrangement of circles on a sphere, with applications in structural biology. Technical Report 6049, INRIA Sophia-Antipolis, December 2006.

[CLRS01] T. E. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2nd edition, 2001.

[Col75] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proc. 2nd GI Conf. Automat. Theory Form. Lang.*, volume **33** of *LNCS*, pages 134–183. Springer, 1975.

[dBvKOS00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer, Berlin, Germany, 2nd edition, 2000.

[DEKW01] C. A. Duncan, A. Efrat, S. G. Kobourov, and C. Wenk. Drawing with fat edges. In *Proc. 9th Intern. Sympos. Graph Drawing*, pages 162–177, 2001.

[DFMT02] O. Devillers, A. Fronville, B. Mourrain, and M. Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. *Computational Geometry: Theory and Applications*, **22**(1–3):119–142, 2002.

[DLLP03] L. Dupont, D. Lazard, S. Lazard, and S. Petitjean. Near-optimal parameterization of the intersection of quadrics. In *Proc. 19th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 246–255, 2003.

[EK06] I. Z. Emiris and M. I. Karavelas. The predicates of the Apollonius diagram: Algorithmic analysis and implementation. *Computational Geometry: Theory and Applications*, **33**(1–2):18–57, 2006.

[EKP$^+$04] I. Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, and E. P. Tsigaridas. Towards an open curved kernel. In *Proc. 20th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 438–446, 2004.

[EKSW04] A. Eigenwillig, L. Kettner, E. Schömer, and N. Wolpert. Complete, exact and efficient computations with cubic curves. In *Proc. 20th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 409–418, 2004.

[Elb95] G. Elber. Freeform surface region optimization for 3-axis and 5-axis milling. *Computer-Aided Design*, **27**(6):465–470, June 1995.

[ET04] I. Z. Emiris and E. P. Tsigaridas. Computing with real algebraic numbers of small degree. In *Proc. 12th Europ. Sympos. Alg. (ESA)*, volume **3221** of *LNCS*, pages 652–663. Springer, 2004.

[FGK$^+$00] 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**:1167–1202, 2000.

[FH06] 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)*, pages 3–15, 2006.

[FHK$^+$06] E. Fogel, D. Halperin, L. Kettner, M. Teillaud, R. Wein, and N. Wolpert. Arrangements. In J.-D. Boissonnat and M. Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*, chapter **1**. Springer, 2006.

[Fla00] E. Flato. Robust and efficient construction of planar Minkowski sums. M.Sc. thesis, School of Computer Science, Tel-Aviv University, 2000. `http://www.cs.tau.ac.il/CGAL/Theses/flato/thesis/`.

[For04] S. Fortune. Voronoi diagrams and Delaunay triangulations. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter **23**, pages 513–528. Chapman & Hall/CRC, 2nd edition, 2004.

[FP06] A. Fabri and S. Pion. A generic lazy evaluation scheme for exact geometric computations. In *Proc. 2nd Workshop on Library-Centric Software Design (LCSD)*, 2006.
`http://sms.cs.chalmers.se/bibliography/proceedings/2006-LCSD.pdf`.

[FWH04] E. Fogel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proc. 12th Europ. Sympos. Alg. (ESA)*, volume **3221** of *LNCS*, pages 664–676. Springer, 2004.

[FWZH06] E. Fogel, R. Wein, B. Zukerman, and D. Halperin. 2D regularized boolean set-operations. In CGAL Editorial Board, editor, CGAL-*3.2 User and Reference Manual*. 2006. `http://www.cgal.org/Manual/3.2/doc_html/cgal_manual/Boolean_set_operations_2/Chapter_main.html`.

[GHH+03] M. Granados, P. Hachenberger, S. Hert, L. Kettner, K. Mehlhorn, and M. Seel. Boolean operations on 3D selective Nef complexes: Data structure, algorithms, and implementation. In *Proc. 11th Europ. Sympos. Alg. (ESA)*, volume **2832** of *LNCS*, pages 174–186. Springer, 2003.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GM91] S. K. Ghosh and D. M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. on Computing*, **20**(5):888–910, 1991.

[GO03] R. Geraerts and M. H. Overmars. A comparative study of probabilistic roadmap planners. In J.-D. Boissonnat, J. Burdick, K. Goldberg, and S. Hutchinson, editors, *Algorithmic Foundations of Robotics V*, pages 43–57. Springer, 2003.

[GO04] R. Geraerts and M. H. Overmars. Clearance based path optimization for motion planning. In *Proc. IEEE Internat. Conf. Robot. Auto. (ICRA)*, pages 2386–2392, 2004.

[Gra97] A. Gray. *Modern Differential Geometry of Curves and Surfaces with Mathematica*, chapter *Logarithmic Spirals*, pages 40–42. CRC Press, Boca Raton, FL, 2nd edition, 1997.

[Gre83] D. H. Greene. The decomposition of polygons into convex parts. In F. P. Preparata, editor, *Computational Geometry*, volume **1** of *Adv. Comput. Res.*, pages 235–259. JAI Press, Greenwich, CT, 1983.

[GRS83] L. J. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 100–111, 1983.

[GS87] L. J. Guibas and R. Seidel. Computing convolutions by reciprocal search. *Discrete and Computational Geometry*, **2**:175–193, 1987.

[Hal04] 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.

[Han00] I. Hanniel. The design and implementation of planar arrangements of curves in CGAL. M.Sc. thesis, School of Computer Science, Tel-Aviv University, 2000.

[Hel91] M. Held. *On the Computational Geometry of Pocket Machining*, volume **500** of *LNCS*. Springer, 1991.

[Her89] J. Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Information Processing Letters*, **33**:169–174, 1989.

[Her06] S. Hert. 2D polygon partitioning. In CGAL Editorial Board, editor, CGAL-*3.2 User and Reference Manual*. 2006. `http://www.cgal.org/Manual/3.2/doc_html/cgal_manual/Partition_2/Chapter_main.html`.

[HH03] 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.

[HHK$^+$01] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proc. 5th Internat. Workshop Alg. Eng. (WAE)*, volume **2141** of *LNCS*, pages 79–90. Springer, 2001.

[HK06] P. Hachenberger and L. Kettner. 3D Boolean operations on Nef polyhedra. In CGAL Editorial Board, editor, CGAL-*3.2 User and Reference Manual*. 2006. `http://www.cgal.org/Manual/3.2/doc_html/cgal_manual/Nef_3/Chapter_main.html`.

[HKL04] D. Halperin, L. E. Kavraki, and J.-C. Latombe. Robotics. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter **48**, pages 1065–1094. Chapman & Hall/CRC, 2nd edition, 2004.

[HL02] S. Hirsch and E. Leiserowitz. Exact construction of Minkowski sums of polygons and a disc with application to motion planning. Technical Report ECG-TR-181205-01, Tel-Aviv University, 2002.

[HM83] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th Internat. Conf. Found. Comput. Theory*, volume **158** of *LNCS*, pages 207–218. Springer, 1983.

[Hof04] C. M. Hoffmann. Solid modeling. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter **56**, pages 1257–1278. Chapman & Hall/CRC, 2nd edition, 2004.

[HPCA$^+$95] S. Har-Peled, T. M. Chan, B. Aronov, D. Halperin, and J. Snoeyink. The complexity of a single face of a Minkowski sum. In *Proc. 7th Canad. Conf. Comput. Geom. (CCCG)*, pages 91–96, 1995.

[HS86] S. Hart and M. Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica*, **6**:151–177, 1986.

[HS96] D. Halperin and M. Sharir. A near-quadratic algorithm for planning the motion of a polygon in a polygonal environment. *Discrete and Computational Geometry*, **16**:121–134, 1996.

[HS98] D. Halperin and C. R. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *Computational Geometry: Theory and Applications*, **10**(4):273–288, 1998.

[HSA01] S. Ho, S. Sarma, and Y. Adachi. Real-time interference analysis between a tool and an environment. *Computer-Aided Design*, **33**(13):935–947, November 2001.

[HW07] I. Hanniel and R. Wein. An exact, complete and efficient computation of arrangements of Bézier curves. In *Proc. ACM Solid Phys. Model. Sympos. (SPM)*, pages 253–263, June 2007.

[IEH$^+$04] O. Ilushin, G. Elber, D. Halperin, R. Wein, and M.-S. Kim. Precise global collision detection in multi-axis machining. *Computer-Aided Design*, **37**(9):909–920, August 2004.

[JHDS89] R. B. Jerard, S. Z. Hussaini, R. L. Drysdale III, and B. Schaudt. Approximate methods for simulation and verification of numerically controlled machining programs. *The Visual Computer*, **5**(6):329–348, 1989.

[Kar04] M. I. Karavelas. A robust and efficient implementation for the segment Voronoi diagram. In *Proc. Internat. Sympos. Voronoi Diag.*, pages 51–62, 2004.

[KF95] A. Kaul and R. T. Farouki. Computing Minkowski sums of plane curves. *Internat. J. Computational Geometry and Applications*, **5**(4):413–432, 1995.

[Kha86] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Internat. J. Robotics Research*, **5**(1):90–98, 1986.

[KLPS86] K. Kedem, R. Livne, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete and Computational Geometry*, **1**:59–70, 1986.

[KLPY99] V. Karamcheti, C. Li, I. Pechtchanski, and C. K. Yap. A core library for robust numeric and geometric computation. In *Proc. 15th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 351–359, 1999.

[KMP$^+$04] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. K. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th Europ. Sympos. Alg. (ESA)*, volume **3221** of *LNCS*, pages 702–713. Springer, 2004.

[KO04a] A. Kamphuis and M. H. Overmars. Finding paths for coherent groups using clearance. In R. Boulic and D. K. Pai, editors, *Eurographics/ACM SIGGRAPH Sympos. Computer Animation*, pages 1–10, 2004.

[KO04b] A. Kamphuis and M. H. Overmars. Motion planning for coherent groups of entities. In *Proc. IEEE Internat. Conf. Robot. Auto. (ICRA)*, pages 3815–3822, 2004.

[Kol04] V. Koltun. Almost tight upper bounds for vertical decompositions in four dimensions. *J. of the ACM*, **51**:699–730, 2004.

[KOS91] A. Kaul, M. A. O'Connor, and V. Srinivasan. Computing Minkowski sums of regular polygons. In *Proc. 3rd Canad. Conf. Comput. Geom. (CCCG)*, pages 74–77, 1991.

[KPOL05] A. Kamphuis, J. Pettre, M. H. Overmars, and J.-P. Laumond. Path finding for the animation of walking characters. In *Proc. Eurographics/ACM SIGGRAPH Sympos. Comp. Animat.*, pages 8–9, 2005.

[KR03] B. Kim and J. Rossignac. Collision prediction for polyhedra under screw motions. In *Proc. 8th ACM Sympos. Solid Model. Appl.*, pages 4–10, 2003.

[Kra99] S. G. Krantz. *Handbook of Complex Variables*. Birkhäuser, Boston, MA, 1999.

[KS02] M. Keil and J. Snoeyink. On the time bound for convex decomposition of simple polygons. *Internat. J. Computational Geometry and Applications*, **12**(3):181–192, 2002.

[KŠLO96] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high dimensional configuration spaces. *IEEE Trans. Robotics and Automation*, **12**:566–580, 1996.

[KVLM03] Y. J. Kim, G. Varadhan, M. C. Lin, and D. Manocha. Fast swept volume approximation of complex polyhedral models. In *Proc. 8th ACM Sympos. Solid Model. Appl.*, pages 11–22, 2003.

[LA95] Y. H. Liu and S. Arimoto. Finding the shortest path of a disc among polygonal obstacles using a radius-independent graph. *IEEE Trans. Robotics and Automation*, **11**:682–691, 1995.

[Lat91] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.

[Lat99] J.-C. Latombe. Motion planning: A journey of robots, molecules, digital actors, and other artifacts. *Internat. J. Robotics Research*, **18**(11):1119–1128, 1999. Special Issue on Robotics at the Millennium — Part I.

[LC95] Y.-S. Lee and T.-C. Chang. 2-phase approach to global tool interference avoidance in 5-axis machining. *Computer-Aided Design*, **27**(10):715–729, October 1995.

[LD81] D.-T. Lee and R. L. Drysdale III. Generalization of Voronoi diagrams in the plane. *SIAM J. on Computing*, **10**(1):73–87, 1981.

[LDK03] B. Lauwers, P. Dejonghe, and J. P. Kruth. Optimal and collision free tool posture in 5-axis machining through the tight integration of tool path generation and machine simulation. *Computer-Aided Design*, **35**(5):421–432, April 2003.

[LKE98] I.-K. Lee, M.-S. Kim, and G. Elber. Polynomial/rational approximation of Minkowski sum boundary curves. *Graphical Models and Image Processing*, **60**(2):136–165, 1998.

[LM04] M. C. Lin and D. Manocha. Collision and proximity queries. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter **35**, pages 787–807. Chapman & Hall/CRC, 2nd edition, 2004.

[LP83] T. Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Trans. Computers*, **C-32**(2):108–120, February 1983.

[LPW79] T. Lozano-Pérez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, **22**(10):560–570, October 1979.

[LS87] D. Leven and M. Sharir. Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams. *Discrete and Computational Geometry*, **2**:9–31, 1987.

[LY01] C. Li and C. K. Yap. New constructive root bound for algebraic expressions. In *Proc. 12th ACM-SIAM Symp. Disc. Alg. (SODA)*, pages 496–505, 2001.

[Mae99] T. Maekawa. An overview of offset curves and surfaces. *Computer-Aided Design*, **31**(3):165–173, March 1999.

[Mey06] M. Meyerovitch. Robust, generic and efficient construction of envelopes of surfaces in three-dimensional space. In *Proc. 14th Europ. Sympos. Alg. (ESA)*, volume **4168** of *LNCS*, pages 792–803. Springer, 2006.

[Mig82] M. Mignotte. Identification of algebraic numbers. *J. Algorithms*, **3**(3):197–204, 1982.

[Mit04] J. S. B. Mitchell. Shortest paths and networks. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 27, pages 607–642. Chapman & Hall/CRC, 2nd edition, 2004.

[MN00] K. Mehlhorn and S. Näher. LEDA*: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[MP91] J. S. B. Mitchell and C. H. Papadimitriou. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *J. of the ACM*, **38**(1):18–73, 1991.

[MS03] K. Mehlhorn and M. Seel. Infimaximal frames: A technique for making lines look like segments. *Internat. J. Computational Geometry and Applications*, **13**(3):241–255, 2003.

[Mul90] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbolic Computation*, **10**(3–4):253–280, 1990.

[Mye97] N. Myers. A new and useful template technique: "Traits". In S. B. Lippman, editor, *C++ Gems*, volume **5** of *SIGS Reference Library*, pages 451–458. 1997.

[Nee97] T. Needham. *Visual Complex Analysis.* Oxford University Press, Oxford, UK, 1997.

[NKMO04] D. Nieuwenhuisen, A. Kamphuis, M. Mooijekind, and M. H. Overmars. Automatic construction of roadmaps for path planning in games. In *Proc. Internat. Conf. Computer Games: Artificial Intelligence, Design and Education*, pages 285–292, 2004.

[NO04a] D. Nieuwenhuisen and M. H. Overmars. Motion planning for camera movements. In *Proc. IEEE Internat. Conf. Robot. Auto. (ICRA)*, pages 3870–3876, 2004.

[NO04b] D. Nieuwenhuisen and M. H. Overmars. Useful cycles in probabilistic roadmap graphs. In *Proc. IEEE Internat. Conf. Robot. Auto. (ICRA)*, pages 446–452, 2004.

[ÓSY83] C. Ó'Dúnlaing, M. Sharir, and C. K. Yap. Retraction: A new approach to motion-planning. In *Proc. 15th Annu. ACM Sympos. Theory Comput. (STOC)*, pages 207–220, 1983.

[Ove05] M. H. Overmars. Path planning for games. In *Proc. 3rd Internat. Game Design Tech. Workshop (GDTW)*, pages 29–33, 2005.

[ÓY85] C. Ó'Dúnlaing and C. K. Yap. A "retraction" method for planning the motion of a disc. *J. Algorithms*, **6**:104–111, 1985.

[Pie93] L. Piegl. *Fundamental Developments of Computer Aided Geometric Design.* Academic Press, 1993.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer, New York, NY, 1985.

[PV96] M. Pocchiola and G. Vegter. The visibility complex. *Internat. J. of Computational Geometry and Applications*, **6**(3):279–308, 1996.

[Ram96] G. D. Ramkumar. An algorithm to compute the Minkowski sum outer-face of two simple polygons. In *Proc. 12th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 234–241, 1996.

[Rei87] J. Reif. Complexity of the generalized movers problem. In J. Hopcroft, J. Schwartz, and M. Sharir, editors, *Planning, Geometry and Complexity of Robot Motion*, pages 267–281. Ablex Publishing, Norwood, NJ, 1987.

[RKLM04] S. Redon, Y. J. Kim, M. C. Lin, and D. Manocha. Fast continuous collision detection for articulated models. In *Proc. 9th ACM Sympos. Solid Model. Appl.*, pages 145–156, 2004.

[Roh91] H. Rohnert. Moving a disc between polygons. *Algorithmica*, **6**:182–191, 1991.

[RS94] J. Reif and M. Sharir. Motion planning in the presence of moving obstacles. *J. Association for Computing Machinery*, **41**(4):764–790, 1994.

[RW95] J. Reif and H. Wang. Social potential fields: A distributed behavioral control for autonomous robots. In K. Goldberg, D. Halperin, J.-C. Latombe, and R. Wilson, editors, *Internat. Workshop on Algorithmic Foundations of Robotics*, pages 431–459. A. K. Peters, 1995.

[SA95] M. Sharir and P. Agarwal. *Davenport–Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, 1995.

[Sch00] 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, 2000.

[SH89] J. Snoeyink and J. Hershberger. Sweeping arrangements of curves. In *Proc. 5th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 354–363, 1989.

[Sha04] M. Sharir. Algorithmic motion planning. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter **47**, pages 1037–1064. Chapman & Hall/CRC, 2nd edition, 2004.

[SLL02] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library, User guide and reference manual.* Addison-Wesley, 2002.

[SM88] K. Sutner and W. Maass. Motion planning among time dependent obstacles. *Acta Informatica*, **26**:93–122, 1988.

[SS81] J. T. Schwartz and M. Sharir. On the "piano movers" problem I: The case of a two-dimensional rigid polygonal body moving amidst polygonal barriers. Technical Report 39, Department of Computer Science, New-York University, October 1981.

[SS83a] J. T. Schwartz and M. Sharir. On the "piano movers" problem I: The case of a two-dimensional rigid polygonal body moving amidst polygonal barriers. *Commun. Pure and Applied Mathematics*, **36**:345–398, 1983.

[SS83b] J. T. Schwartz and M. Sharir. On the "piano movers" problem II: General techniques for computing topological properties of real algebraic manifolds. *Advances in Applied Mathematics*, **4**:298–351, 1983.

[SS83c] J. T. Schwartz and M. Sharir. On the "piano movers" problem III: Coordinating the motion of several independent bodies: The special case of circular bodies moving amidst polygonal barriers. *Internat. J. Robotics Research*, **2**(3):46–75, 1983.

[SS91] M. Sharir and S. Sifrony. Coordinated motion planning for two independent robots. *Ann. Mathematics and Artificial Intelligence*, **3**:107–130, 1991.

[vdS94] A. F. van der Stappen. The complexity of the free space for motion planning amidst fat obstacles. *J. Intelligent and Robotic Systems*, **11**:21–44, 1994.

[Ver94] S. Verma. Simulation of numerically controlled machines. M.Sc. thesis, Computer Science Department, The University of Utah, 1994.

[Wei02a] R. Wein. High-level filtering for arrangements of conic arcs. M.Sc. thesis, School of Computer Science, Tel-Aviv University, 2002.

[Wei02b] R. Wein. High-level filtering for arrangements of conic arcs. In *Proc. 10th Europ. Sympos. Alg. (ESA)*, volume **2461** of *LNCS*, pages 884–895. Springer, 2002.

[Wei05] R. Wein. Efficient implementation of red-black trees with split and catenate operations. Technical report, Tel-Aviv University, 2005. `http://www.cs.tau.ac.il/~wein/publications/pdfs/rb_tree.pdf`.

[Wei06a] R. Wein. 2D envelopes. In Cgal Editorial Board, editor, Cgal-*3.3 User and Reference Manual*. 2006. `http://www.cgal.org/Manual/3.3/doc_html/cgal_manual/Envelope_2/Chapter_main.html`.

[Wei06b] R. Wein. 2D Minkowski sums. In Cgal Editorial Board, editor, Cgal-*3.3 User and Reference Manual*. 2006. `http://www.cgal.org/Manual/3.3/doc_html/cgal_manual/Minkowski_sum_2/Chapter_main.html`.

[Wei06c] R. Wein. Exact and efficient construction of planar Minkowski sums using the convolution method. In *Proc. 14th Europ. Sympos. Alg. (ESA)*, volume **4186** of *LNCS*, pages 829–840. Springer, 2006.

[Wei07] R. Wein. Exact and approximate construction of offset polygons. *Computer-Aided Design*, **39**(6):518–527, June 2007.

[WFZH05] R. Wein, E. Fogel, B. Zukerman, and D. Halperin. Advanced programming techniques applied to CGAL's arrangement package. In *Proc. 1st Workshop on Library-Centric Software Design (LCSD)*, pages 24–33, 2005. `www.cs.rpi.edu/research/pdf/06-12.pdf`.

[WFZH07] R. Wein, E. Fogel, B. Zukerman, and D. Halperin. Advanced programming techniques applied to CGAL's arrangement package. *Computational Geometry: Theory and Applications*, **38**(1–2):37–63, 2007.

[WH04] 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.

[WIEH04] R. Wein, O. Ilushin, G. Elber, and D. Halperin. Continuous path verification in multi-axis NC-machining. In *Proc. 20th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 86–95, 2004.

[WIEH05] R. Wein, O. Ilushin, G. Elber, and D. Halperin. Continuous path verification in multi-axis NC-machining. *Internat. J. Computational Geometry and Applications*, **15**(4):351–377, 2005.

[WvdBH05] R. Wein, J. P. van den Berg, and D. Halperin. The visibility-Voronoi complex and its applications. In *Proc. 21st Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 63–72, 2005.

[WvdBH06] R. Wein, J. P. van den Berg, and D. Halperin. Planning near-optimal corridors amidst obstacles. In *Proc. 7th Internat. Workshop Alg. Found. Robot. (WAFR)*, 2006. To appear.

[WvdBH07] R. Wein, J. P. van den Berg, and D. Halperin. The visibility-Voronoi complex and its applications. *Computational Geometry: Theory and Applications*, **36**(1):66–87, 2007.

[WZ06] R. Wein and B. Zukerman. Exact and efficient construction of planar arrangements of circular arcs and line segments with applications. Technical Report ACS-TR-121200-01, Tel-Aviv University, March 2006.

[Yap04] 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.

[YD95] C. K. Yap and T. Dubé. The exact computation paradigm. In D. Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume **4** of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.