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

Robust and Efficient Construction of Planar Minkowski Sums

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

by

Eyal Flato

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

August 2000

I deeply thank Prof. Dan Halperin for supervising this research and contributing many useful ideas.

I thanks Pankaj K. Agarwal for discussions and valuable comments on this work. I also wish to thank Iddo Hanniel, Sigal Raab, Oren Nechushtan, Eti Ezra, Eli Pecker, Sariel Har-Peled and Michal Ozery for helpful discussions concerning the problems studied in this thesis. I wish to thank the whole CGAL team and especially CGAL members in Tel-Aviv University.

Contents

1 Introduction **5**

1.1 Fundamental Complexity Bounds 5

1.2 Related Work 7

1.3 The CGAL Library 10

1.4 Thesis Outline 11

2 Preliminaries **15**

2.1 Planar Arrangements 15

2.2 Vertical Decomposition 17

2.2.1 Robot Motion Planning 17

2.3 Planar Maps and Arrangements in CGAL 18

3 Minkowski Sum Algorithms **23**

3.1 Minkowski Sum of Two Convex Polygons 24

3.2 Polygons Union Algorithms 25

3.2.1 Arrangement Algorithm 25

3.2.2 Incremental Union Algorithm 26

3.2.3 Divide and Conquer Algorithm 27

3.3 Input Sets 27

3.4 Experiments 28

3.4.1 Test Platform and Frame Program 28

3.4.2 Results 31

3.4.3 Order of Insertion 34

4 Polygon Decomposition **37**

4.1 The Decomposition Algorithms 39

4.1.1 Triangulation 39

4.1.2 Convex Decomposition without Steiner Points 40

4.1.3 Convex Decomposition with Steiner Points 40

4.2 A First Round of Experiments 41

4.3 Revisiting the More Efficient Algorithms 44

4.3.1 Nonoptimality of Min-Convex Decompositions 44

4.3.2 Mixed Objective Functions 45

4.3.3 Improving the AB and KD methods 48

5 Conclusions **51**

A Handling Degeneracies in the Union Algorithms **53**

A.1 Handling Degeneracies in the Arrangement Union Algorithm 53

A.2 Handling Degeneracies in the Incremental Union Algorithm 57

B Proof of Theorem 3.2.1: Construction Time of Arrangements of Convex Polygons **61**

C Polygons Decomposition Minimizing the Mixed Objective Function **65**

C.1 Minimum Length Decomposition 66

C.2 Constrained Minimum Length Decomposition 68

Chapter 1

Introduction

Given two sets P and Q in $\mathbb{R}^2$, their *Minkowski sum* (or vector sum), denoted by $P \oplus Q$, is the set $\{p + q \mid p \in P, q \in Q\}$. Minkowski sums are used in a wide range of applications, including robot motion planning [34], assembly planning [20], and computer-aided design and manufacturing (CAD/CAM) [13].

Consider for example an obstacle P and a robot Q that moves by translation. We can choose a reference point r rigidly attached to Q and suppose that Q is placed such that the reference point coincides with the origin. If we let Q' denote a copy of Q rotated by 180°, then $P \oplus Q'$ is the locus of placements of the point r where $P \cap Q \neq \emptyset$. In the study of motion planning this sum is called a *configuration space obstacle* because Q collides with P when translated along a path π exactly when the point r, moved along π, intersects $P \oplus Q'$. See Figure 1.1.

1.1 Fundamental Complexity Bounds

Motivated by these applications, there has been much work on obtaining sharp bounds on the size of the Minkowski sum of two sets in two and three dimensions, and on developing fast algorithms for computing Minkowski sums. It is well known that if P is a polygonal set with m vertices and Q is another polygonal set with n vertices, then $P \oplus Q$ is a portion of the arrangement of $O(mn)$ segments, where each segment is the Minkowski sum of a vertex of P and an edge of Q, or vice-versa. Therefore the size of $P \oplus Q$ is $O(m^2n^2)$ and it can be computed within that time; this bound is tight in the worst case [27] (see Figure 1.2). If both P and Q are convex, then $P \oplus Q$ is a convex polygon with at most $m + n$ vertices (see Figure 1.3), and it can be computed in $O(m + n)$ time [34].

If only P is convex, then a result of Kedem et al. [29] implies that $P \oplus Q$ has $\Theta(mn)$ vertices (see Figure 1.4). Their proof relies on special properties of a set of *pseudodiscs*. We say that a collection of planar regions each bounded by a closed Jordan curve is a collection

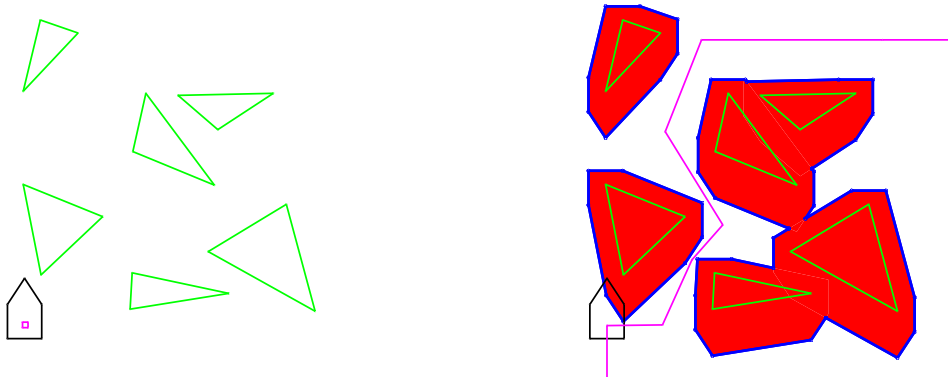


Figure 1.1: Robot and obstacles: a reference point is rigidly attached to the robot on the left-hand side. The configuration space obstacles and a free translational path for the robot on the right-hand side.

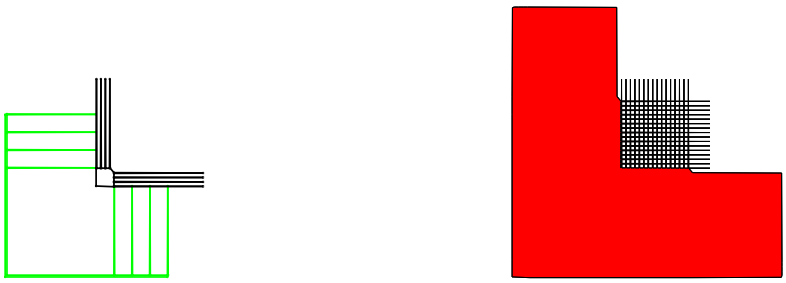


Figure 1.2: Fork input: P and Q are polygons with m and n vertices respectively each having horizontal and vertical teeth. The complexity of $P \oplus Q$ is $\Theta(m^2n^2)$.

Figure 1.3: Convex input: P and Q are convex polygons with m and n vertices. The complexity of $P \oplus Q$ is $\Theta(m+n)$.

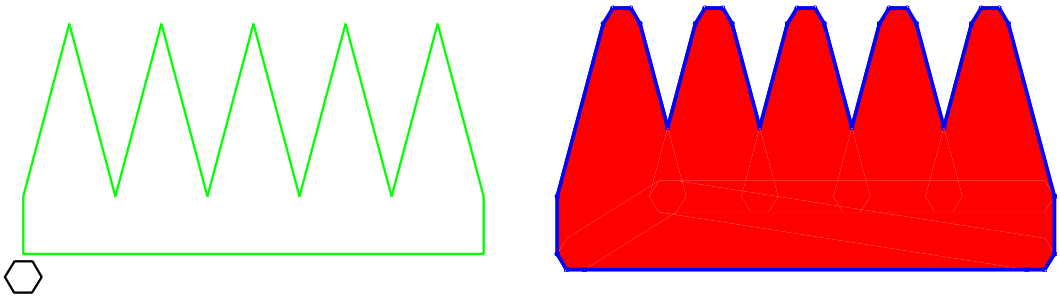


Figure 1.4: Comb input: P is a convex polygon with m vertices and Q is a comb-like polygon with n vertices. The complexity of $P \oplus Q$ is $\Theta(mn)$.

of pseudodiscs, if the boundary curves of every pair in the collection intersect at most twice. Kedem et al. [29] prove that the number of intersection points (namely vertices on the boundary where two curves intersect) on the union boundary of n pseudodiscs is $O(n)$. If P, Q_1 and Q_2 are convex polygons then $P \oplus Q_1$ and $P \oplus Q_2$ are proved to be pseudodiscs. Q can be decomposed into $O(n)$ convex subpolygons such that $P \oplus Q = \bigcup P \oplus Q_i$. The boundary of this union includes $O(n)$ intersection points among the subsums $P \oplus Q_i$ and a total of $O(mn)$ vertices. Such a Minkowski sum can be computed in $O(mn \log(mn))$ time [35].

1.2 Related Work

More Complexity Bounds

In the previous section we presented the well known combinatorial bounds on the size of the Minkowski sum of polygonal sets. In motion-planning applications, one is often interested in computing only a single connected component of the complement of $P \oplus Q$ [40]. Har-Peled et al. [24] showed that the complexity of a single face of the complement of $P \oplus Q$ is $\Theta(mn\alpha(n))$ in the worst case where m and n are the number of vertices of P and Q respectively (without loss of generality $n < m$), and $\alpha(\cdot)$ is the functional inverse of Ackermann's function [43].

Barrera [6] showed that the Minkowski sum of two monotone polygons can be computed in $O(n^2 \log n)$ time for two polygons with a total of n edges. He also proved that computing the Minkowski sum of two polygons is at least as hard as *sorting* $X + Y$ [7]. Sorting $X + Y$ is the problem of sorting the set of numbers $\{x + y | x \in X, y \in Y\}$ for two sets X, Y of n numbers each. The best known time bound for solving this problem is $O(n^2 \log n)$ and it is an open problem whether it can be improved. There is a set of other problems that are "sorting $X + Y$ hard" (for example: the polygon containment problem for two rectilinearly convex polygons[1]).

De Berg and van der Stappen [11] report on results concerning the relation between the

[1] A polygon is *rectilinearly convex* if its intersection with any horizontal or vertical line is connected.

fatness of the Minkowski sum of two sets and the fatness of the sets. The fatness of an object is determined by the emptiest ball centered inside the object and not fully containing it in its interior. Using this measure they show that the fatness of $A \oplus B$ is at least as large as $\min(fatness(A), fatness(B))$, where A and B are connected closed and bounded sets in $\mathbb{R}^d$.

Discrete Approximations

Hartquist et al. [25] suggest a computing strategy for applications that use offsets, sweeps and Minkowski operations based on the *ray-representation* method. This method involves clipping a given input to a grid of rays and applying the mathematical definitions and operators (such as Minkowski sum) on the resulting discrete set. The authors aim to solve motion planning, process-modeling and visualization problems and they present a hardware design for those applications.

Kavraki [28] uses the Fast Fourier Transform (FFT) algorithm on the bitmaps of a robot and obstacles to find the corresponding configuration-space obstacles for the robot translating among the obstacles. This method approximates the configuration space obstacles. The method is inherently parallel and can benefit from existing experience and special hardware for computing the FFT.

Applications

The translational robot motion problem planning is a convenient case study for Minkowski sum algorithms, and therefore detailed and given as an example in the rest of this thesis. There are many more applications in which the Minkowski sum operation is a useful tool. Some examples are listed here.

The following problem arises in mechanical assembly planning. An *assembly* is a collection of non-overlapping rigid parts. Given an assembly, identify a subassembly (i.e., a subset of the parts) that can be removed as a rigid object without colliding with the rest of the assembly. This is the *assembly partitioning* problem. A simple instance of the problem is where the given parts $P_1, P_2, \ldots, P_n$ are polygons in the plane, and we would like to find a removal path consisting of two consecutive translations that will separate a subset of the parts from the rest of them (notice that finding the subset is part of the problem).

Halperin and Wilson [22] use Minkowski sums to compute the configuration space obstacle $P_{ij} = P_i \oplus -P_j$ for every ordered pair of parts (P_i, P_j) using the origin as the common reference point for all the parts. For a point q in the plane, if $q \in P_{ij}$ then if P_j is placed with its reference point in q it will collide with P_i. Therefore, a path through q cannot be used to separate a subset of parts that contains P_j but not P_i. In the arrangement $\mathcal{A}$ of all the sums P_{ij}, every face introduces a set of constrains of the form: "P_j cannot be moved through this face without P_i" (according to the Minkowski sums P_{ij} in which it is contained). Given a path γ in the plane we define a directed graph $G = G(\gamma)$ whose nodes

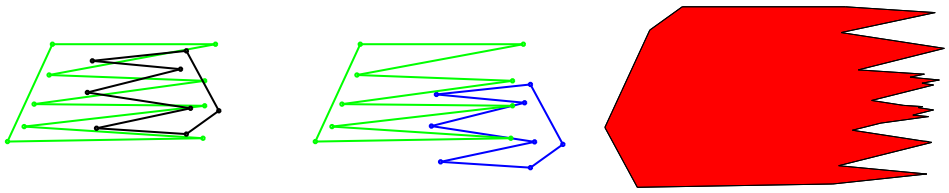


Figure 1.5: Minimal distance separation

correspond to the parts $P_1, P_2, \ldots, P_n$. There is a directed edge in G between the nodes corresponding to P_i and P_j if such a constraint appears in one of the faces of $\mathcal{A}$ that are intersected by γ. It is easily verified that if G is not strongly connected then there is a subassembly that can be removed along γ and this subassembly is a strongly connected subgraph of G. In [22] an $O(n^2N^6)$ algorithm is given to solve this problem (where N is the total number of vertices in the input).

The approach described above makes extensive use of planar Minkowski sums, and therefore calls for an efficient construction of such sums.

Geographic Information Systems (GIS) are increasingly studied in computational geometry. There are some problems in GIS that are closely related to our work. One of them is the *buffer searching* problem in which we would like to find geographic features that are within a given buffer distance from a polygonal feature. Boolean operations on planar objects are frequently used in GIS and therefore have efficient implementations in many geographic systems. Instead of measuring the distance between each feature and the query polygon, we can execute the buffer searching by first computing the Minkowski sum of a circle and the query polygon. Then, we intersect the resulting planar subdivision with the geographic database. The latter is a boolean operation that can be carried out efficiently.

The following question was posed by Marc van Kreveld as a *cartographic generalization* problem [45]: Given two polygons find the minimum length translation of one polygon relative to the other that will make the two polygons interior disjoint; see Figure 1.5. Assuming that in their original placement P and Q intersect and that the reference point of Q is at the origin O, it is not difficult to see that the minimum translation of Q relative to P is described by the point on the boundary of $P \oplus Q'$ which is closest to O where Q' is a copy of Q rotated 180°.

Given two polygons P and Q in the plane, another widely studied problem is to find whether P can be contained inside Q. This problem is known as the polygon containment problem [9]. If we restrict it to a translational problem (namely the orientation of P is fixed) it can be solved as follows: consider the complement of Q as an obstacle for the robot P and try to place P such that it does not penetrate the obstacle. Practically, let B be the bounding box of Q and let $\overline{Q} = B \setminus Q$. The free placements for P inside Q can be found by computing $P' \oplus \overline{Q}$ were P' is P rotated by 180°. See Figure 1.6 for an example.

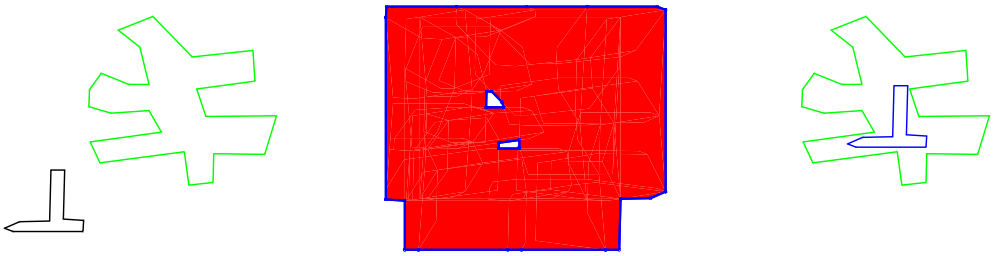


Figure 1.6: Polygon containment: the input polygons P and Q are displayed on the left-hand side, $P' \oplus \overline{Q}$ is in the middle, and a possible placement for P inside Q is on the right-hand side.

Three and Higher Dimensions

In planar motion planning, if beside translating we allow the robot to rotate then the configuration space is 3-dimensional [21]. For a given rotation angle ϕ_0 the Minkowski sum $P_{\phi_0} \oplus Q$, where P_{ϕ_0} is P rotated by ϕ_0 degrees, is the translational configuration-space obstacle for the robot in a fixed rotation angle. The entire configuration space includes beside the translational axis, a rotation axis ϕ. Each horizontal slice of this space (the plane $\phi = \phi_0$) contains the sum $P_{\phi_0} \oplus Q$. This observation is used in several approximate solutions to motion planning problems; see, e.g., [3] and [34, Section 6.5.1].

The Minkowski operations in higher dimensions are defined similarly. A summary of the known results on computing the Minkowski sum of two sets in three and higher dimensions can be found in a recent survey by Agarwal and Sharir [2].

This thesis is concerned with the 2-dimensional case where both the input and the result are planar.

1.3 The CGAL Library

We devised and implemented a package for computing the Minkowski sum of two polygonal sets based on the CGAL software library [1, 16]. CGAL — Computational Geometry Algorithms Library — is a software library developed by several research groups in Europe and Israel. The package supplies a robust, efficient, and flexible implementation of computational geometry algorithms and data structures. CGAL consist of the *kernel* which supplies geometric primitives and data types, the *basic library* which contains a large collection of basic algorithms and data structures (for example triangulations, planar maps), and a *support library* for I/O, debugging and visualization. CGAL is developed following the generic programming paradigm known from the Standard Template Library (STL) for C++ [5, 44]. Our Minkowski-sum package employs CGAL's *planar maps* [17] and *arrangements* [23] packages and follows CGAL's look-and-feel of generic programming (planar maps and arrangements are subdivisions induced by geometric objects; see Chapter 2 for details).

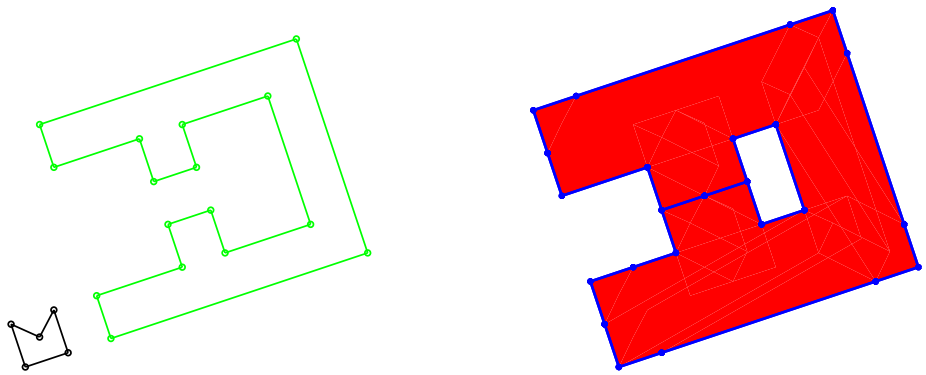


Figure 1.7: Tight passage: the desired target placement for the small polygon is inside the inner room defined by the larger polygon (left-hand side). In the configuration space (right-hand side) the only possible path to achieve this target passes through the line segment emanating into the hole in the sum.

We are currently using our software to solve translational motion planning problems in the plane. We are able to compute collision-free paths even in environments cluttered with obstacles, where the robot could only reach a destination placement by moving through tight passages, practically moving in contact with the obstacle boundaries. See Figure 1.7 for an example. This is in contrast with most existing motion planning software for which tight or narrow passages constitute a significant hurdle.

The CGAL library provides a robust implementation of basic geometric structures (e.g., planar maps) that can handle degenerate inputs (without assuming "general position"). Furthermore, we are able to choose different number types and geometric predicates to be used by the implementation. In our implementation we use rational numbers and filtered geometric predicates from LEDA — the library of efficient data structures and algorithms [37].

Transforming a geometric algorithm from theory to practical implementation raises several issues (like arithmetic precision and the treatment of degenerate inputs) which we collectively refer to as *robustness issues*. Our implementation handles robustness issues by applying exact number types and floating point filters and by directly handling degenerate input. We refer the reader to recent surveys on this topic [42, 46] for further information.

1.4 Thesis Outline

The thesis presents a general scheme for computing the Minkowski sum of two polygonal sets and describes the different steps of the computation. We describe the software package

which implements those steps and report on experimental results.

Computing the Minkowski sum of two polygonal sets P and Q can be done as follows: (1) decompose P and Q into s and t convex subpolygons respectively, (2) compute the Minkowski sum of each pair of subpolygons of P and Q resulting in the set R of $s \cdot t$ subsums, and (3) construct the union of those subsums; the result is represented as a planar map.

In the next chapter we introduce some related basic definitions and algorithms in computational geometry. We present the concept of an arrangement of curves, the vertical decomposition of an arrangement and point location algorithms. We then describe the implementation of those data structures and algorithms in CGAL.

In Chapter 3 we concentrate on the last steps in the computation of a Minkowski sum of two polygonal sets. Based on the decomposition of the input polygonal sets into convex subpolygons, after describing how to compute the Minkowski sum of two convex polygons, we present three algorithms for computing the union of the set of Minkowski subsums. The first is the *arrangement* algorithm, in which we construct the arrangement induced by the edges of the polygons in R. Then we traverse the arrangement and mark each face, edge and vertex as inside the union, on its boundary or outside the union. The construction of the arrangement takes randomized expected time $O(I + k \log k)$ (where k is the number of edges in R and I is the overall number of intersections between (edges of) polygons in R). The traversal stage takes $O(I + k)$ time. The second algorithm is the *incremental union* algorithm, in which we maintain the partial union in a planar map by inserting the polygons of R one after the other. After each insertion we remove the redundant edges from the map. We could only give a naive bound on the running time of this algorithm, which in the worst case is higher than the worst-case running time of the arrangement algorithm. Practically however the incremental union algorithm works much better than the arrangement algorithm on most problem instances. The third algorithm is the *divide-and-conquer* algorithm. This algorithm is a combination of both previous algorithms. First we use the *incremental union* algorithm to compute t maps representing the Minkowski sums of P and each convex subpolygon of Q. Then we compute the union of pairs of maps using the *arrangement* algorithm, obtaining $t/2$ new maps. We continue to compute union of pairs of intermediate maps $\log t$ times until we end up with one map describing the Minkowski sum of P and Q. We report on our experiments with these three algorithms as well as on other factors that can affect the computation such as the order of insertion of subsums.

In the theoretical study of Minkowski sum computation (e.g., [29]), the choice of decomposition is often irrelevant (as long as we decompose the polygons into convex subpolygons) because it does not affect the worst-case *asymptotic* running time of the algorithms. In practice however, different decompositions can induce a large difference in running time of the Minkowski sum algorithms. In Chapter 4 we examine different methods for decomposing polygons by their suitability for efficient construction of Minkowski sums. We study and experiment with various well-known decompositions as well as with several new decomposi-

tion schemes. We report on our experiments with the various decompositions and different input polygons. Among our findings are that in general: (i) triangulations are too costly (although they can be produced quickly, they considerably slow down the Minkowski-sum computation), (ii) what constitutes a good decomposition for one of the input polygons depends on the other input polygon — consequently, we develop a procedure for simultaneously decomposing the two polygons such that a "mixed" objective function is minimized, (iii) there are optimal decomposition algorithms that significantly expedite the Minkowski-sum computation, but the decomposition itself is expensive to compute — in such cases simple heuristics that approximate the optimal decomposition perform very well.

We give concluding remarks and suggest directions for further research in Chapter 5.

Chapter 2

Preliminaries

In this chapter we present some of the tools and terminology that we will be using throughout the thesis.

2.1 Planar Arrangements

An *arrangement* of curves in the plane is the subdivision of the plane induced by these curves. Consider, for example, the arrangement induced by a collection of n line segments in the plane. The line segments partition the plane into vertices, edges and faces. A *vertex* is an endpoint of a segment or an intersection point of two (or more) segments, an *edge* is a maximal connected portion of an original segment that does not meet any vertex, and a *face* is a maximal connected region of the plane not meeting any edge or vertex; see Figure 2.1 for an illustration. Such an arrangement has at most $O(n^2)$ vertices, edges and faces, and this bound is tight. The features of the arrangement are also called *cells*. A vertex is a 0-dimensional cell, an edge is a 1-dimensional cell and a face is a 2-dimensional cell. A *subcell* of a k-dimensional cell c_1 is a $(k-1)$-dimensional cell c_2 that is on the boundary of c_1. If c_2 is a subcell of c_1 then c_1 and c_2 are considered *incident*.

Let Γ be a collection of n curves in the plane, and let $\mathcal{A}(\Gamma)$ be the arrangement of Γ. Given another curve π we define the *zone* of π to be the set of faces of $\mathcal{A}(\Gamma)$ that are intersected by π (Figure 2.2). If the curves of Γ are x-monotone Jordan arcs such that each pair intersects in at most s points then the complexity of the zone of a curve π which intersects any curve of Γ in at most some constant number of points is $\Theta(\lambda_{s+2}(n))$, where $\lambda_s(n)$ is the maximum length of a Davenport-Shinzel sequence of order s on n symbols [43]. This result implies that for arrangements of line segments, the complexity of the zone of another segment is $\Theta(n\alpha(n))$ where $\alpha(\cdot)$ is the extremely slowly growing functional inverse of Ackermann's function.

Arrangements are also defined for higher dimensional objects. See [2, 19] for more

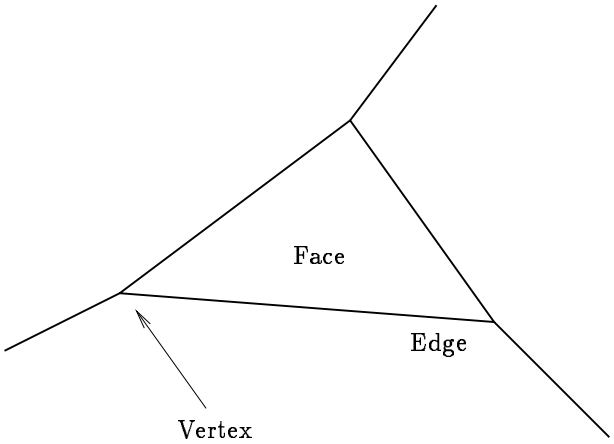


Figure 2.1: A vertex, an edge and a face in an arrangement of segments

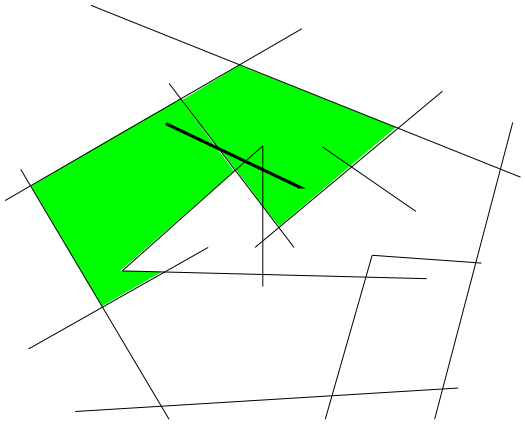


Figure 2.2: An arrangement of segments. The shaded area is the zone of the segment in bold line.

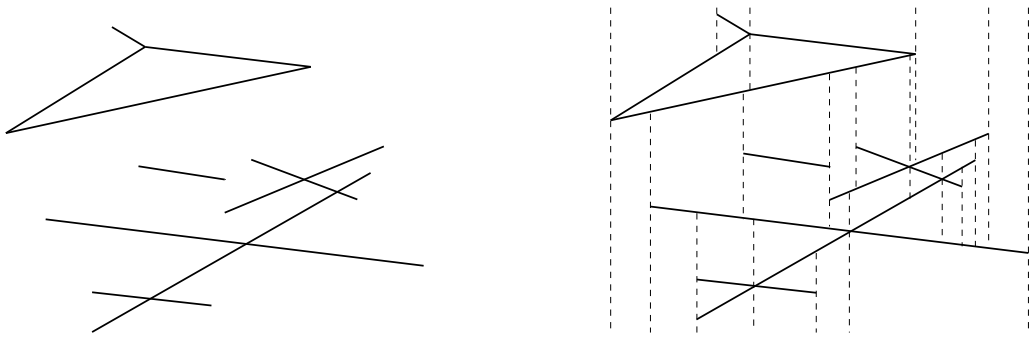


Figure 2.3: An arrangement of segments on the left-hand side and its vertical decomposition on the right-hand side

details on arrangements in two and higher dimensions and other important substructures (e.g., envelopes) of them. However, in this work we will focus on planar arrangements.

2.2 Vertical Decomposition

Let S be a set of n line segments in the plane. In the previous section we defined the arrangement $\mathcal{A}(S)$ of S. $\mathcal{A}(S)$ is a subdivision of the plane into regions that can, unfortunately, have complex shapes. Hence, it is convenient to further refine this subdivision. The *vertical decomposition* (also known as the *trapezoidal decomposition*) is a planar subdivision D such that from each vertex of $\mathcal{A}(S)$ we extend a vertical attachment. Each vertical attachment extends upwards and downwards until it hits another edge or vertex of $\mathcal{A}(S)$ and if no such feature exists, then it extends to infinity (Figure 2.3). D is a refinement of the original subdivision $\mathcal{A}(S)$: every face of D lies completely in one face of $\mathcal{A}(S)$. The faces of D are called *vertical trapezoids* even though they can also be triangles or unbounded trapezoids. Let I be the number of intersection points among the segments of S then the complexity of $\mathcal{A}(S)$ is $O(I+n)$. The complexity of the subdivision D is $O(I+n)$ and it can be constructed in expected $O(I + n\log n)$ time using a randomized incremental algorithm [39]. During an incremental construction of D we can also build a search structure such that the trapezoid in which a query point lies can be found in expected $O(\log n)$ time. This operation is called *point location*. In Section 2.3 we briefly describe three different strategies to answer a point location query in arrangements and planar maps as they are implemented in CGAL.

2.2.1 Robot Motion Planning

Vertical decompositions are commonly used in theory and practice. They are used to simplify the subdivision induced by an arrangement of curves with a fairly low overhead in time and storage. The ability to get a simple and efficient search structure on the

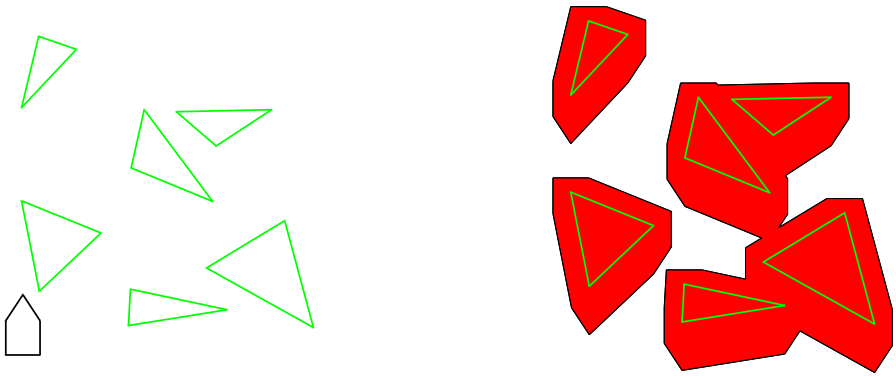


Figure 2.4: Robot and obstacles: on the left-hand side the workspace which includes the robot (on the bottom left) and the obstacles and on the right-hand side the configuration-space obstacles

vertical decomposition is another reason for its popularity. One of the applications of vertical decomposition is robot motion planning which is relevant to this work. We describe methods for constructing the Minkowski sum of polygonal sets. Planning a motion of a polygonal robot translating among polygonal obstacles (Figure 2.4) can be carried out as follows: (i) First, fix a reference point r on the robot and construct the configuration-space obstacles C by computing the Minkowski sum of the robot rotated by 180° and the obstacles (Figure 2.4). (ii) Plan a path in the free portion FP of the configuration space. Moving r along the computed path while the robot is rigidly attached to r gives a collision free motion plan for the robot in the workspace. We discuss the first step in detail in the sequel. The second step can be easily accomplished using a trapezoidal decomposition as we explain next.

Let C' be the trapezoidal decomposition of FP. We would like to compute a path in FP for a point from p_{start} to p_{goal}. If p_{start} and p_{goal} lie in the same trapezoid we can simply move along a straight line segment between them. Otherwise, we construct a road map through the free space. We set a node in the center of each free trapezoid and in the center of each of its vertical walls. We connect each node in the center of a trapezoid with all the nodes on the trapezoid's boundary. This gives us a planar graph embedded entirely in FP (Figure 2.5) . Constructing the road map and finding a path in it from the trapezoid that contains p_{start} to the trapezoid that contains p_{goal} is easily achieved in time linear in the complexity of the road map, after the trapezoidal decomposition had been constructed.

2.3 Planar Maps and Arrangements in CGAL

CGAL — Computational Geometry Algorithms Library — is a software library developed by several research groups in Europe and Israel. The library provides a robust, efficient, and

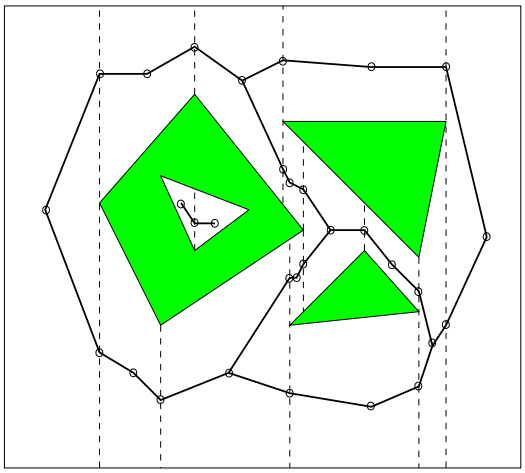


Figure 2.5: Road map constructed using a vertical decomposition of the free space

flexible implementation of computational geometry algorithms and data structures [1, 16]. The planar-map [17] and arrangement software packages are part of CGAL's basic library.

Given a set S of non-intersecting x-monotone curves in the plane, the planar-map package contains data structures and algorithms to dynamically maintain the planar subdivision induced by the curves of S. Furthermore, the planar-map package allows for flexibility in choosing the curve type and supports the use of robust number types for computations. The package does not assume *general position*, namely it handles degenerate inputs. Beside the insertion and removal operations, it supports several useful services: traversal of the map features, point location and trapezoidal decomposition. These capabilities are implemented as follows (the full details are given in [17] and [23]):

Geometric Traits The geometric traits class is an abstract interface of predicates and functions that wraps the access of an algorithm to the geometric (rather than combinatorial) inner representation. In the planar-map package the traits class is defined as the minimal geometric interface which will enable a construction and handling of a geometric map. The traits class defines the basic objects of the map: the point and the x-monotone curve. In addition it defines predicates for comparing points, accessing curves' endpoints, comparing points and curves (e.g., whether a point is above, below or on a given curve), and comparing curves (e.g., compare the y-coordinate of two curves at a given x-coordinate). The traits class implicitly defines the geometric representation and robustness handling methods.

Doubly Connected Edge List (DCEL) The DCEL [12, Chapter 2] is the fundamental data structure used by the planar map. This representation belongs to a family of edge-based data structures in which each edge is represented as a pair of opposite *halfedges*. Each halfedge e points to its source and target vertices: $source(e)$

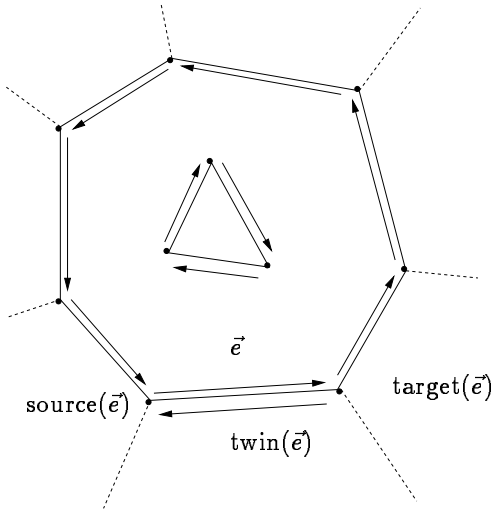


Figure 2.6: Source and target vertices, and twin halfedges in a face with a hole

and *target*(e), to its twin (opposite) halfedge: *twin*(e), to the next and previous halfedges on the Connected Component of the Boundary (CCB) of its face: *next*(e) and *previous*(e), and to the face on its left: *face*(e). Each face f points to a halfedge on its outer boundary (if it exists) and to a list of holes: *outer*(f) and *holes*(f). Each vertex v points to a halfedge from it: *halfedge*(v). See an example of a DCEL in Figure 2.6. We can use the bidirectional pointers (previous and next) to traverse a CCB of a face , to jump to neighboring faces (twin pointer) and to explore all the faces and halfedges that are incident to a vertex. The implementation enables keeping extra data in each feature of the DCEL.

Topological Layer and Geometric Layer The topological layer is responsible for maintaining the combinatorial data by using the DCEL as the storage class. We can update the map by using *insert*, *remove* and *split* operations. The topological layer also enables us to traverse the features of the planar map by combinatorial connectivity using a set of iterators: (i) halfedges around vertex iterator, (ii) connected component of the boundary of a face (CCB) iterator, (iii) holes iterator. The geometric layer is an embedding of the topological layer in the plane using the geometric traits. The topological layer can also be used for non-planar subdivisions (e.g., terrains, subdivisions on a sphere).

Point Location Strategy The planar map package supports point location queries. Using the point location as a *strategy*[1] enables the users to implement their own point location algorithms. The planar-map package supplies three point location strategies: (i) a naive algorithm — goes over all the edges in the map to find the location of

[1] The use of "strategy" here refers to the *strategy pattern* [18].

the query point; (ii) an efficient algorithm — Mulmuley's randomized incremental algorithm that uses a vertical decomposition and a search structure to answer a point location query in expected $O(\log^2 n)$ time [38] (we get an additional logarithmic factor over the regular trapezoidal decomposition because here the search structure is fully dynamic); and (iii) a "walk" algorithm that is an improvement over the naive one; it finds the point's location by walking along a ray from "infinity" towards the query point, traversing only the zone of this ray rather than the entire map.

The arrangement package [23] uses the same technology but handles curves that are not necessarily x-monotone and that are allowed to intersect. The arrangement class keeps a hierarchy graph by which we can get the original curve from which a halfedge in the map was created.

We use the planar maps of CGAL along with the implementation of rational numbers from LEDA[2] — the Library of Efficient Data-structures and Algorithms [36, 37]. LEDA provides a set of algorithms and data structures from graph theory and computational geometry. The library includes an implementation of exact number types (e.g., rational numbers, real numbers). LEDA's rational number holds two varying length integers: a numerator and a denominator to represents an exact rational. In addition, it includes a floating point approximation (floating-point filter) to save computing time by resorting to expensive computation only when the correct answer cannot be determined with the approximation only. LEDA also provides a graphic user interface for interactive applications.

[2] http://www.mpi-sb.mpg.de/LEDA/leda.html

Chapter 3

Minkowski Sum Algorithms

We devised and implemented three algorithms for computing the Minkowski sum of two polygonal sets based on the CGAL software library [1, 16]. Our main goal was to produce a *robust* and exact implementation. This goal was achieved by employing the CGAL *planar map* package (described in Chapter 2) while using exact number types.

The input to our algorithms are two polygonal sets P and Q, with m and n vertices respectively. P and Q are sets of simple polygons that are not necessarily pairwise disjoint. All our algorithms consist of the following three steps:

Step 1: Decompose P into the convex subpolygons $P_1, P_2, \ldots, P_s$ and Q into the convex subpolygons $Q_1, Q_2, \ldots, Q_t$. The number of vertices of a polygon X is denoted by $|X|$.

Step 2: For each $i \in [1..s]$ and for each $j \in [1..t]$ compute the Minkowski *subsum* $P_i \oplus Q_j$ which we denote by R_{ij}. We denote by R the set $\{R_{ij} \mid i \in [1..s], j \in [1..t]\}$.

Step 3: Construct the union of all the polygons in R, computed in Step 2; the output is represented as a planar map.

We discovered that the choice of the decomposition in Step 1 can have a dramatic effect on the running time of the Minkowski-sum algorithms. We postpone the discussion of the decomposition process to Chapter 4. We can assume, for this chapter, that the input polygonal sets are decomposed into convex subpolygons. Throughout the experiments that we describe in this chapter we use the same decomposition which has proved very efficient for our purposes. It is based on a heuristic method proposed in [10] which we have slightly improved and we refer to as the small-side angle-bisector decomposition; for details see Chapter 4.

In the next section we describe the computation of the Minkowski sum of two convex polygon (Step 2). We survey the polygons union algorithms (Step 3) that we implemented in Section 3.2. The input sets on which we performed the experiments are listed in Section 3.3 and experimental comparison between the algorithms is given in Section 3.4.

3.1 Minkowski Sum of Two Convex Polygons

The second phase of the Minkowski-sum computation is constructing the subsums $R_{ij} := P_i \oplus Q_j$. We review a simple, well-known linear-time algorithm to compute this Minkowski sum [12, Chapter 13][1].

Given two convex polygons P and Q with m and n vertices respectively, we will compute $P \oplus Q$. For a given direction $\vec{d}$ an extreme point in direction $\vec{d}$ on $P \oplus Q$ is the vector sum of extreme points in direction $\vec{d}$ on P and Q. For a convex polygon, if we change the direction $\vec{d}$ in a counterclockwise manner then we get a sequence of extreme points that contains all the vertices of the polygon ordered exactly as they are ordered on the polygon's counterclockwise boundary. The following algorithm scans the directions in counterclockwise order and uses the above observation to simultaneously traverse both polygons and find extreme points in each direction.

1. Let $v_1, \ldots, v_m$ and $w_1, \ldots, w_n$ be the vertices of P and Q, ordered in counter-clockwise order, with v_1 and w_1 being the vertices with smallest y-coordinate (and smallest x-coordinate in case of ties)
2. $i \leftarrow 1$; $j \leftarrow 1$
3. $v_{m+1} \leftarrow v_1$; $w_{n+1} \leftarrow w_1$
4. **repeat**
5. Add $v_i + w_j$ as a vertex to $P \oplus Q$
6. **if** $\mathit{angle}(v_i v_{i+1}) < \mathit{angle}(w_j w_{j+1})$ **then** $i \leftarrow i + 1$
7. **else if** $\mathit{angle}(v_i v_{i+1}) > \mathit{angle}(w_j w_{j+1})$ **then** $j \leftarrow j + 1$
8. **else** $i \leftarrow i + 1$; $j \leftarrow j + 1$
9. **if** $i > m + 1$ **then** $i \leftarrow m + 1$
10. **if** $j > n + 1$ **then** $j \leftarrow n + 1$
11. **until** $i = m + 1$ and $j = n + 1$

We use the notation $\mathit{angle}(ab)$ to denote the angle that the vector ab makes with the positive x-axis.

This algorithm runs in linear time $O(m + n)$, because at each iteration of the loop in lines 4 – 11 either i or j are incremented. The vertices of $P \oplus Q$ are vector sums of pairs of vertices from P and Q that are extreme at a common direction. Since the input polygons are convex the angle test ensures that all extreme pairs are found.

[1] The algorithm as it appears in the book might run into an infinite loop. The algorithm that we present contains the necessary fixes.

3.2 Polygons Union Algorithms

The Minkowski sum of P and Q is the union of the polygons in R. Let k denote the overall number of edges of the polygons in R, and let I denote the overall number of intersections between (edges of) polygons in R. We present three different algorithms for performing Step 3, computing the union of the polygons in R, which we refer to as the *arrangement* algorithm, the *incremental union* algorithm and the *divide-and-conquer* algorithm.

3.2.1 Arrangement Algorithm

The algorithm constructs the arrangement $\mathcal{A}(R)$ induced by the polygons in R (we refer to this arrangement as the *underlying arrangement* of the Minkowski sum) by adding the (boundaries of the) polygons of R one by one in a random order and by maintaining the vertical decomposition of the arrangement of the polygons added so far. Each polygon is chosen with equal probability at each step. Once we have constructed the arrangement, we traverse all its cells (vertices, edges or faces) and we mark a cell as belonging to the Minkowski sum if it is contained inside at least one polygon of R.

Theorem 3.2.1 *The construction of $\mathcal{A}(R)$ takes randomized expected time $O(I + k \log k)$.*

Proof: See Appendix B. □

The traversal stage takes $O(I+k)$ time. The number of vertices in R is $k = O(mn)$ therefore the expected construction time using the arrangement algorithm is $O(I + mn \log(mn))$ and the traversal stage takes $O(I + mn)$.

As mentioned earlier, a distinctive feature of our work is our ability to handle degenerate input such as "tight passages" (Figure 1.7). To do this we need to pay special attention to various "boundary conditions".

Let Union(R) be the union of the polygons of R. Union(R) can be represented by a subset of the features of the arrangement $\mathcal{A}(R)$. Therefore, we would like to compute for each feature in the underlying arrangement (face, edge or vertex) whether it is inside the union or not.

During the insertion of polygons into the arrangement we count for each *halfedge* on how many boundaries of polygons of R it lies. Using this count we can find for each face in how many polygons it is contained by traversing the arrangement once in a breadth-first manner. Then, a face is inside the union if and only if it lies in at least one of the polygons of R.

Next, we would like to know which edges are on the boundary of the union. Trivially we would check for each edge e its two incident faces. If one of those faces is inside the union and the other is not then we would mark e to be on the boundary of the union. But this

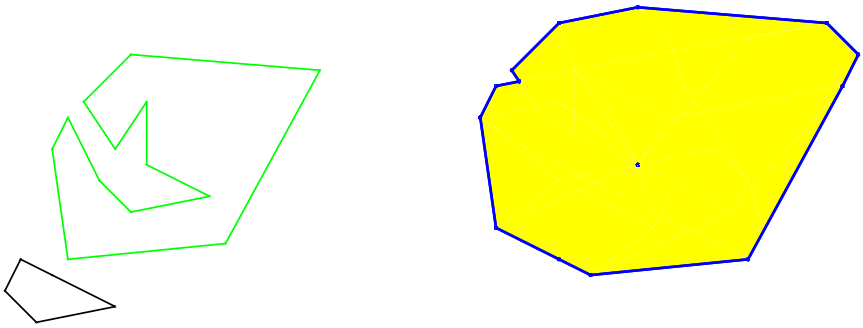


Figure 3.1: Semi-free vertex: the small polygon can fit into the cavity in the larger polygon (left-hand side) as indicated by a singular point in the middle of the Minkowski sum of the larger polygon with a copy of the small one rotated by 180° (right-hand side)

is insufficient since there are edges that are on the boundary of one or more polygons of R but are not contained in any of those polygons (in motion planning such an edge represents a tight passage for the robot through the obstacles — see Figure 1.7). This edge is on the boundary of the union but the faces on its sides are both inside the union (in some cases such an edge might not be connected to the rest of the boundary of the union). We refer to such an edge as a *semi-free* edge. If the number of polygons that contain a halfedge on their boundary equals to the number of polygons that contain the face bounded by this halfedge in the arrangement then we mark this halfedge to be on the boundary.

Finally, we should check the vertices. A vertex is on the boundary of Union(R) if it is a target or a source vertex of an edge that is on the boundary of Union(R), or it can be disconnected from the rest of the boundary (in motion planning it is a semi-free vertex which represents a location in which a robot can be placed but from which it cannot move in any direction — see Figure 3.1). We provide the full technical details of this process in Appendix A.1.

3.2.2 Incremental Union Algorithm

In this algorithm we incrementally construct the union of the polygons in R by adding the polygons one after the other in random order. We maintain the planar map representing the partial union of polygons in R. For each $r \in R$ we insert the edges of r into the map and then remove redundant edges from the map. Redundant edges are edges that are completely contained in (at least) one of the polygons of R inserted so far. We do this using the *"coloring"* procedure described below. All the operations of this procedure can be carried out efficiently using the planar map package. We could only give a naive bound on the running time of this algorithm, which in the worst case is higher than the worst-case running time of the arrangement algorithm. Practically however the incremental union algorithm works much better than the arrangement algorithm on most problem instances.

To compute the exact union and boundary we should identify in each insertion step

of the incremental algorithm what features are redundant and can therefore be removed. While the arrangement algorithm uses a post-processing stage, in this algorithm we execute a coloring procedure after each insertion of a polygon into the map. After the insertion of a polygon r, the coloring procedure marks all the faces that are contained in r to be inside the union. In addition we can remove all the edges and vertices that lie completely inside r since they do not contribute to the union any more. A special treatment should be given to the edges and vertices on the boundary of r. Some of those edges may be a part of the boundary of the partial union as they separate between a face that is inside the union and a face outside the union. Again, as in the arrangement algorithm we have cases in which an edge (or a vertex) is surrounded by faces that are in the partial union but the edge (or the vertex) is on the union's boundary.

We can remove an edge from the boundary of r if it does not overlap with part of the union's boundary before we added r and both its adjacent faces are inside the union. A vertex that is on the boundary of r can be removed if it was first inserted with r and all the faces that are adjacent to it are inside the union. The full details of the coloring procedure are explained in Appendix A.2.

3.2.3 Divide and Conquer Algorithm

While the incremental algorithm removes the redundant edges from the union map, the arrangement algorithm handles all the edges of the polygons of R during the entire process. On the other hand, with the incremental algorithm we may have very complex faces in our planar map. Handling these faces is highly time consuming (we discuss these issues in Section 3.4.2). The divide-and-conquer algorithm is a combination of both previous algorithms, attempting to overcome the shortcomings of both. First we use the *incremental union* algorithm to compute the Minkowski sums of P and Q_j for each $1 \leq j \leq t$. This results in t polygonal sets (each represented as a planar map) $S_1, \ldots, S_t$, where S_j's complexity is $O(n|Q_j|)$ [29]. In the second stage we compute the union of pairs of maps from $S_1, \ldots, S_t$ using the *arrangement* algorithm, obtaining $t/2$ new maps. We continue to compute union of pairs of maps $\log t$ times until we end up with one map describing the Minkowski sum of P and Q.

This algorithm is just one way of applying a divide-and-conquer scheme for computing the union of polygons. We found this method efficient because it balances the use of the previous two union algorithms.

3.3 Input Sets

The input data we used is described in Table 3.1 and in Figures 3.2 through 3.10. The Minkowski sum of a *comb* and a convex polygon has complexity $\Theta(mn)$ (see Figure 3.2), while the *fork* input results in $\Theta(m^2n^2)$ Minkowski sum complexity (see Figure 3.4). The

input name	description	figure
comb	P is a 'comb' with $(m-3)/2$ teeth, and Q is a convex polygon with n vertices, of which $n-2$ lie on the top boundary	Figure 3.2
star	P and Q are star-shaped polygons	Figure 3.3
fork	P and Q each consists of two orthogonal sets of 'teeth' such that their Minkowski sum has $\Theta(n^2m^2)$ complexity	Figure 3.4
random	P and Q are random looking polygons [15]	Figure 3.5
concave chain	P and Q consist of concave chains with $m-1$ and $n-1$ vertices, respectively	Figure 3.6
mixed chain	P consists of chains of different type of vertices: convex vertices, concave vertices and comb-like vertices (alternating from concave to convex repeatedly), Q is a star-shaped polygon	Figure 3.7
knife	P is shaped as a long triangle with short and even comb teeth along its base and Q consists of horizontal and vertical teeth	Figure 3.8
countries borders	This is real-life data consisting of the polygonal description of the borders of several countries across the world	Figure 3.9
robot and obstacles	Q is a star-shaped robot and P consist of triangular obstacles which are randomly placed inside a square	Figure 3.10

Table 3.1: Input data

rest of the input data results in Minkowski sum complexity that is between $\Theta(m+n)$ and $\Theta(m^2n^2)$. The 'intermediate' inputs (star, random, countries) are interesting in that there are many different ways to decompose them into convex subpolygons — this is the topic of the next chapter.

3.4 Experiments

We present experimental results of applying the algorithms described in Section 3.2 to the collection of input pairs of polygonal sets that are listed in the previous section.

3.4.1 Test Platform and Frame Program

Our implementation of the Minkowski sum package is based on the CGAL (version 2.0) and LEDA (version 4.0) libraries. Our package works with Linux (g++ compiler) as well as with

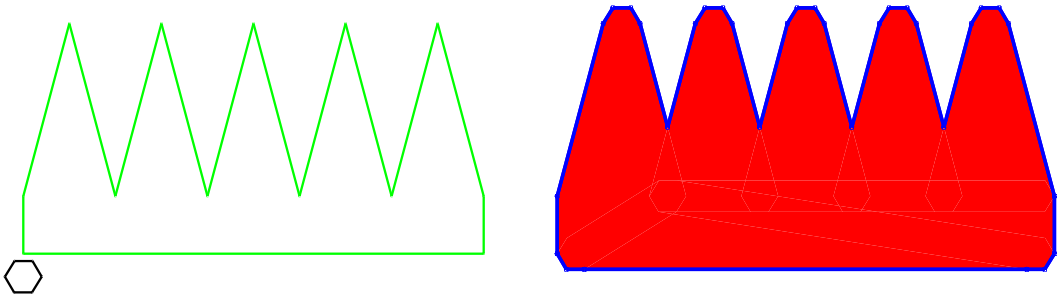


Figure 3.2: Comb input on the left-hand side and the Minkowski sum on the right-hand side

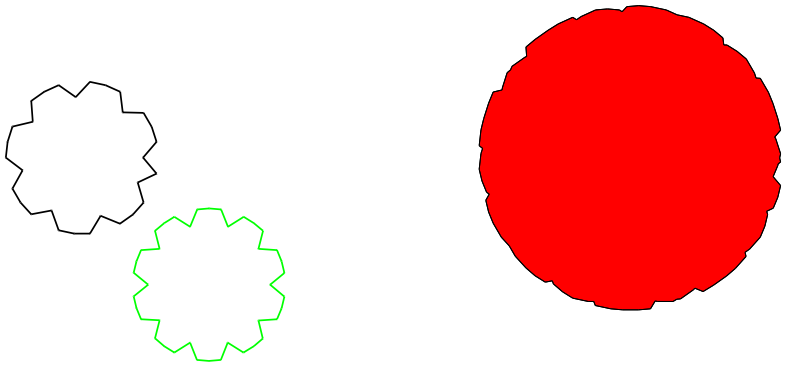


Figure 3.3: Star input

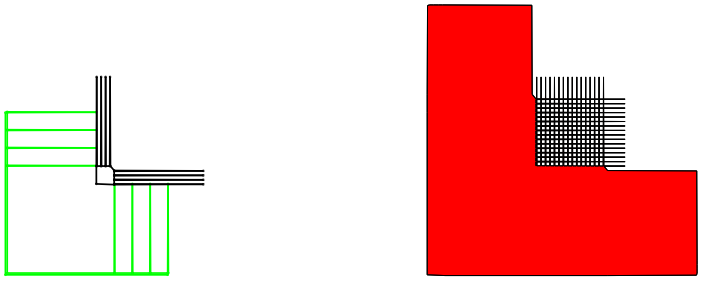


Figure 3.4: Fork input

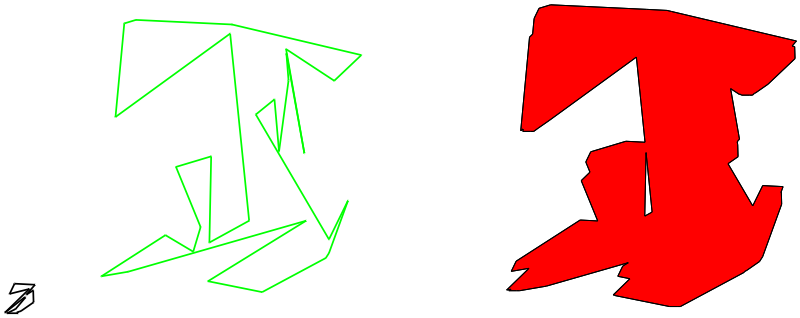


Figure 3.5: Random looking polygons input

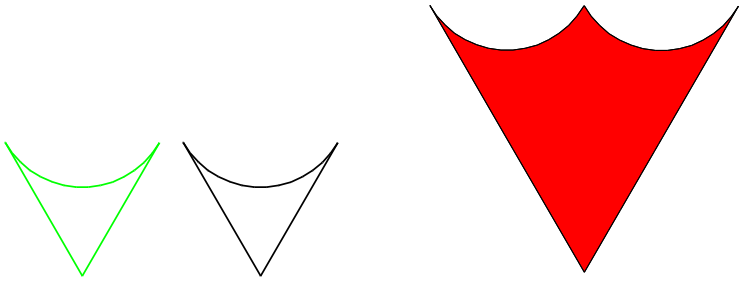


Figure 3.6: Concave chains input

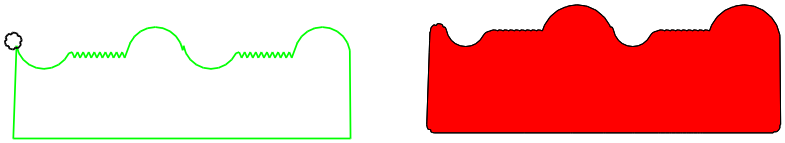


Figure 3.7: Mixed chain input

Figure 3.8: Knife input

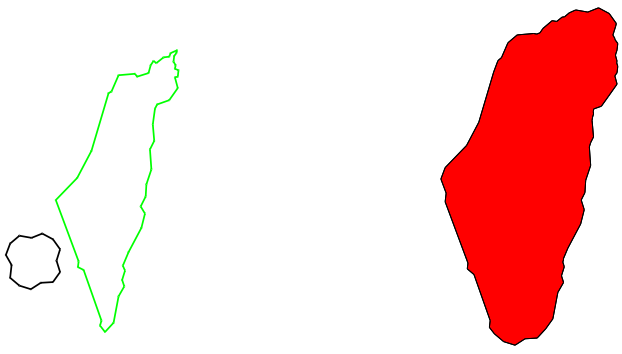


Figure 3.9: Countries borders input

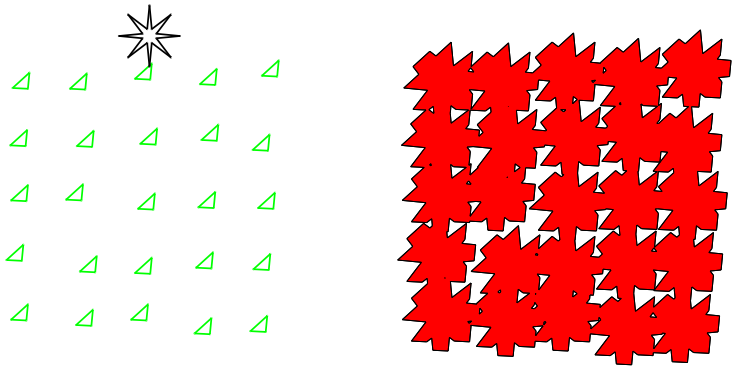


Figure 3.10: Robot and obstacles input

WinNT (Visual C++ 6.0 compiler). The tests were performed under WinNT workstation on a 500 MHz PentiumIII machine with 128 Mb of RAM.

We implemented an interactive program (Figure 3.11) which constructs Minkowski sums, computes configuration space obstacles, and solves polygon containment and polygon separation problems. The software also enables to choose the decomposition method and union algorithm and presents the resulting Minkowski sum and underlying arrangement. The software is available from `http://www.math.tau.ac.il/~flato/`.

We measured the running times for the various algorithms with different input data.

3.4.2 Results

The results of running the three union algorithms are presented in Figure 3.12. We can see that for polygonal sets for which the Minkowski sum is complex (e.g., the fork input) the arrangement algorithm performs better. When the sum's complexity is relatively small (e.g., the star input) the incremental algorithm has the best running times. The divide-and-conquer algorithm's performance is mostly between the other two algorithms and is likely to be closer to the best algorithm.

The complexity of the Minkowski sum varies for the different input data. For the two input sets P and Q we denote by V_{PQ} the total number of vertices in the arrangement of the polygons of R, and M_{PQ} the number of vertices on the boundary of $P \oplus Q$. Figure 3.13 presents the ratio $C_{PQ} = \frac{M_{PQ}}{V_{PQ}}$. From comparing this ratio to the running times of the three algorithms (Figure 3.12) it is clear that for small C_{PQ} the incremental algorithm performs best but as the ratio grows the arrangement algorithm overtakes so that for inputs like the fork and comb it performs better than the other algorithms. The incremental algorithm maintains the union of the polygons of R removing redundant edges and vertices. Smaller C_{PQ} indicates that there are many vertices and edges in the arrangement of R that do not contribute to the boundary of the Minkowski sum. Handling those features during the union process (as the arrangement algorithm does) is therefore costly. When C_{PQ} is large,

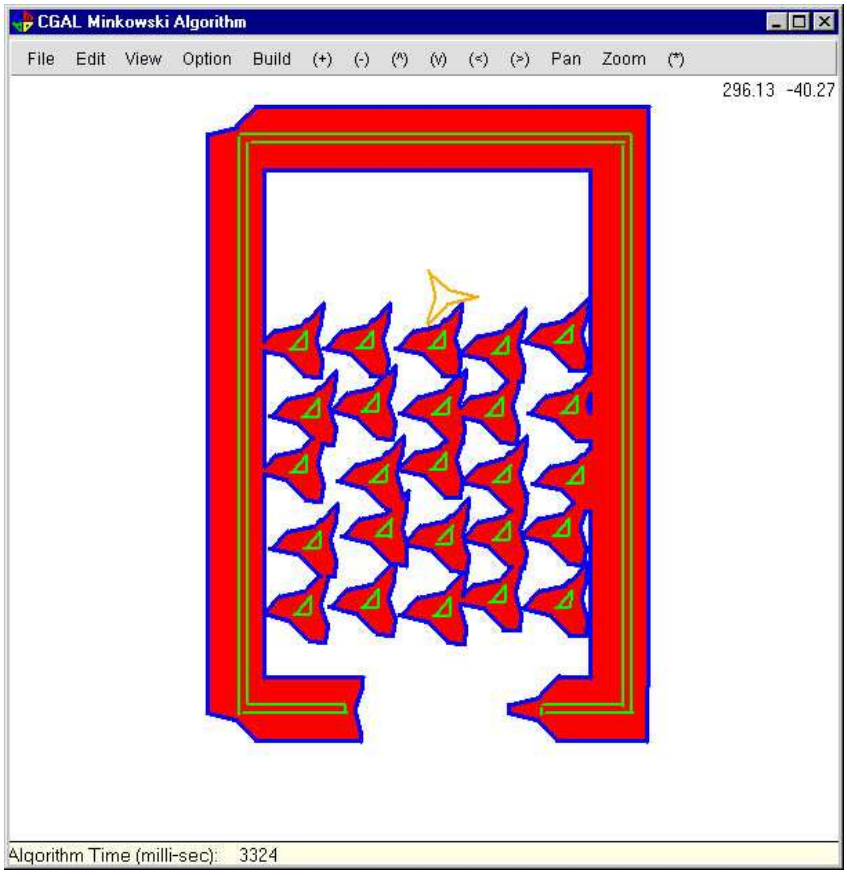


Figure 3.11: The Minkowski-sum application — a screenshot

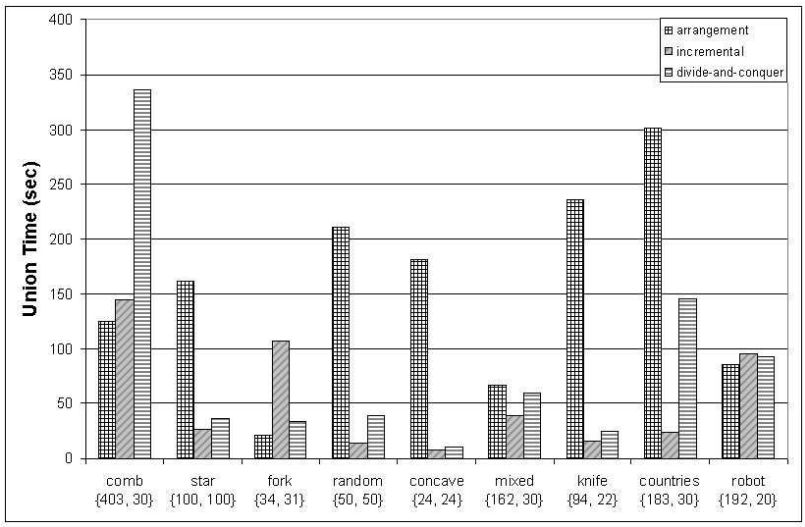


Figure 3.12: Running times in seconds for computing the Minkowski sum of different input data with all three union algorithms. The sizes of P and Q are in parenthesis.

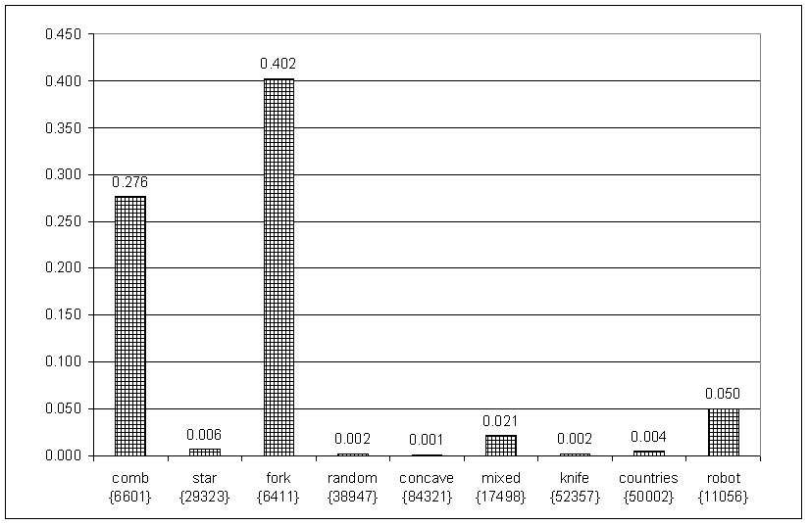


Figure 3.13: C_{PQ}: the ratio between the number of vertices in the Minkowski sum of different input data compared with the number of vertices of the underlying arrangement. The values V_{PQ} are in parenthesis. We measured C_{PQ} for the same inputs we used to produce the results in Figure 3.12.

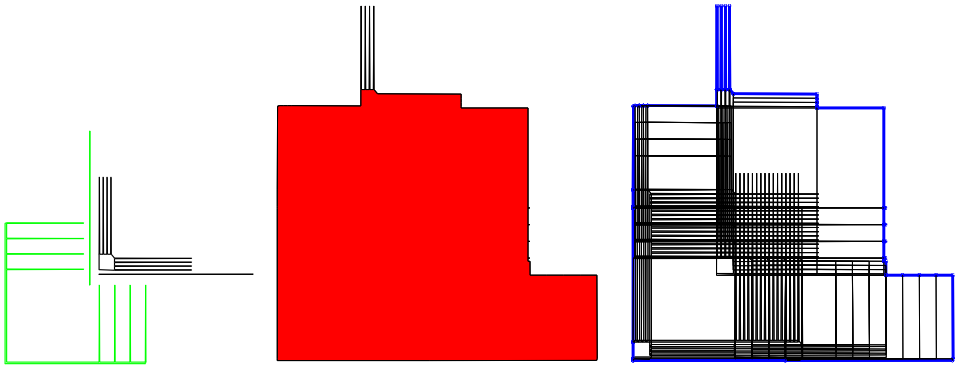


Figure 3.14: Covered fork: input sets on the left-hand side (the first set includes the larger fork polygon and a long vertical triangle and the second set includes the smaller fork polygon and a long horizontal triangle), the Minkowski sum in the middle, and the underlying arrangement on the right-hand side

removing the unnecessary edges and vertices almost does not help and in the extreme cases results in poor running times. Removing the redundant edges from the map sometimes results in very complex faces. In our implementation of the planar map handling these complex faces can take longer time. Such complex faces are likely to be created when the Minkowski sum is relatively complex (large C_{PQ}).

3.4.3 Order of Insertion

Another factor that affects the running time of the union algorithms is the order in which the polygons of R are inserted into the planar map. Consider for example the *covered fork* input data (suggested to us by R. Wenger). It consists of two fork polygons whose Minkowski sum has complexity $\Theta(m^2n^2)$ and two long triangles whose Minkowski sum is a large hexagon that covers (contains) the grid created by the sum of the fork polygons. Therefore, The Minkowski sum in this case has $\Theta(m+n)$ vertices while the underlying arrangement has $\Theta(m^2n^2)$ vertices. See Figure 3.14. If we use the incremental algorithm and insert the large hexagon first, we can avoid handling the (complex) grid-like planar map and we get output-sensitive running time. This example shows that an algorithm that inserts the subsum polygons of R in random order cannot be output sensitive.

If we insert the polygons of R into the map in descending order of *fatness* (we use here a very simple measure of fatness — the area divided by the diameter squared) we will get the desired output-sensitivity effect in this special case. The results are given in Figure 3.15. This permutation, however, does not always result in better running times. Consider for example Figure 3.16 where all the thinner polygons of R are intersecting the fatter polygons. We can see in the results that for this input (*fat grid*) the union time when using the fatness permutation is about two times slower than when using a random permutation.

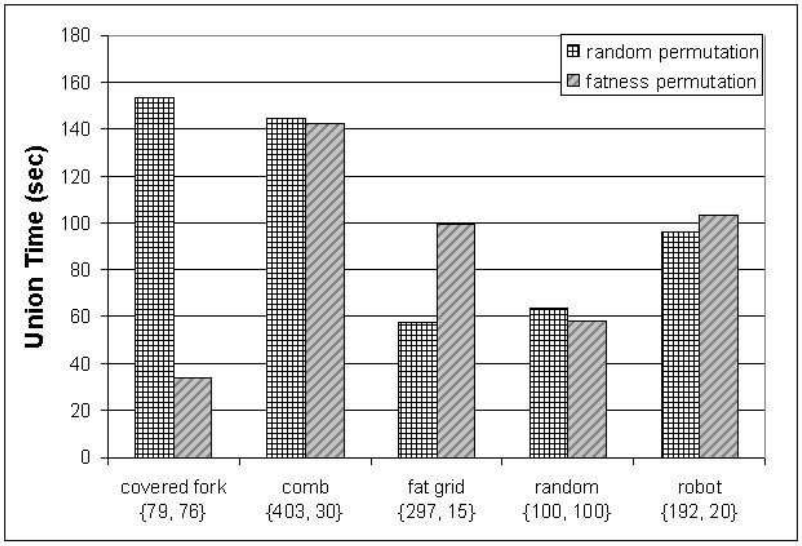


Figure 3.15: Running times in seconds for computing the Minkowski sum of different input data using the incremental algorithm with both random and fatness permutations on the polygons of R. The sizes of P and Q are in parenthesis.

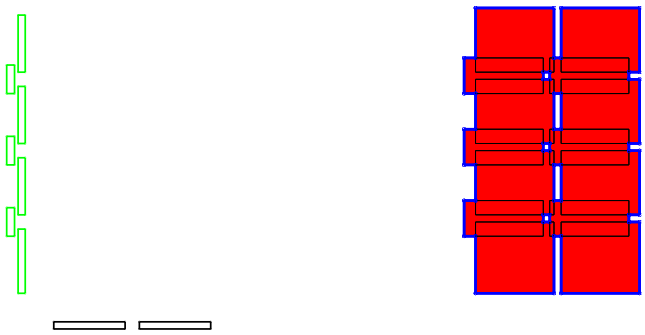


Figure 3.16: Fat grid input

Chapter 4

Polygon Decomposition

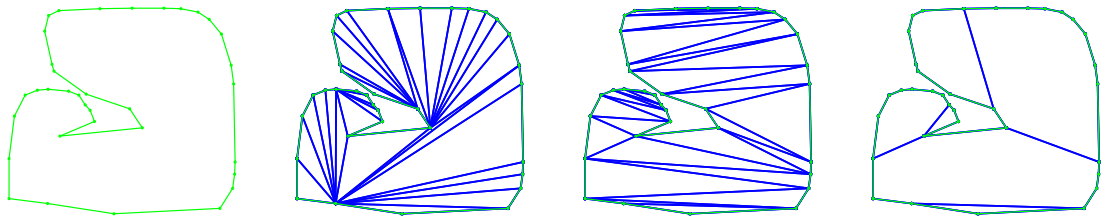
In this chapter we examine different methods for decomposing polygons by their suitability for efficient construction of Minkowski sums. We study and experiment with various well-known decompositions as well as with several new decomposition schemes. Some of the presented algorithms are optimal while others approximate an optimal solution or use various heuristics.

In the theoretical study of Minkowski-sum computation (e.g., [29]), the choice of decomposition is often irrelevant (as long as we decompose the polygons into convex subpolygons) because it does not affect the worst-case *asymptotic* running time of the algorithms. In practice however, different decompositions can induce a large difference in running time of the Minkowski-sum algorithms (see Figure 4.1 for an example). The decomposition can affect the running time of algorithms for computing Minkowski sums in several ways: some of them are global to all algorithms that decompose the input polygons into convex polygons, while some others are specific to certain algorithms or even to specific implementations. We examine these various factors and report our findings below.

Polygon decomposition has been extensively studied in computational geometry; it is beyond the scope of this thesis to give a survey of results in this area and we refer the reader to the survey papers by Keil [33] and Bern [8], and the references therein. As we proceed, we will provide details on specific decomposition methods that we will be using.

We apply several optimization criteria to the decompositions that we employ. In the context of Minkowski sums, it is natural to look for decompositions that minimize the number of convex subpolygons. As we show in the sequel, we are also interested in decompositions with minimal maximum vertex degree of the decomposition graph, as well as several other criteria.

We report on our experiments with the various decompositions and different input polygons. As mentioned in the Introduction, among our findings are that in general: (i) triangulations are too costly, (ii) what constitutes a good decomposition for one of the input polygons depends on the other input polygon — consequently, we develop a procedure for



	P's decomposition		
	naive triang.	min Σd_i^2 triang.	min convex
Σd_i^2	754	530	192
# of convex subpolygons in P	33	33	6
time (mSec) to compute $P \oplus Q$	2133	1603	120

Figure 4.1: Different decomposition methods applied to the polygon P (leftmost in the figure), from left to right: naive triangulation, minimum Σd_i^2 triangulation and minimum convex decomposition (the details are given in Section 4.1). We can see in the table for each decomposition the sum of squares of degrees, the number of convex subpolygons, and the time in milliseconds to compute the Minkowski sum of the polygon with a small convex polygon, Q, with 4 vertices.

simultaneously decomposing the two polygons such that a "mixed" objective function is minimized, (iii) there are optimal decomposition algorithms that significantly expedite the Minkowski-sum computation, but the decomposition itself is expensive to compute — in such cases simple heuristics that approximate the optimal decomposition perform very well.

In the next section we describe the different decomposition algorithms we have implemented. We present a first set of experimental results in Section 4.2 and filter out the methods that turn out to be inefficient. In Section 4.3 we focus on the decomposition schemes that are not only fast to compute but also help to compute the Minkowski sum efficiently.

We use the notation from Chapter 3. For simplicity of the exposition we assume here that the input data for the Minkowski algorithm are two *simple polygons* P and Q. In practice we use the same decomposition schemes that are presented here for general polygonal sets, mostly without changing them at all. However this is not always possible. For example, Keil's optimal minimum convex decomposition algorithm will not work on polygons with holes[1]. Furthermore, the problem of decomposing a polygon with holes to convex subpolygons is proven to be NP-Hard whether Steiner points are allowed or not; see [31]. Other algorithms that we use (e.g., AB algorithm) can be applied to general polygons without changes. We discuss these decomposition algorithms in the following sections.

[1] In such cases we can apply a first decomposition step that connects the holes to the outer boundary and then use the algorithm on the simple subpolygons. This is a practical huristic that does not guarantee an optimal solution.

4.1 The Decomposition Algorithms

We briefly describe here the different algorithms that we have implemented for decomposing the input polygons into convex subpolygons. We used both decomposition with or without Steiner points. Some of the techniques are optimal and some use heuristics to optimize certain objective functions. The running time of the decomposition stage is significant only when we search for the optimal solution and use dynamic programming; in all other cases the running time of this stage is negligible even when we implemented a naive solution. Therefore we only mention the running time for the 'heavy' decomposition algorithms. In what follows P is a polygon with n vertices $p_1, \ldots, p_n$, r of which are reflex.

4.1.1 Triangulation

Greedy triangulation. This procedure searches for a pair of vertices p_i, p_j such that the segment p_ip_j is a diagonal, namely it lies inside the polygon. It adds such a diagonal, splits the polygon into two subpolygons by this diagonal, and triangulates each subpolygon recursively. The procedure stops when the polygon becomes a triangle. See Figure 4.1 for an illustration.

In some of the following decompositions we are concerned with the degrees of vertices in the decomposition (namely the number of diagonals incident to a vertex). Our motivation for considering the degree comes from an observation on the way our planar map structures perform in practice: we noted that the existence of high degree vertices makes maintaining the maps slower. The DCEL structure that is used for maintaining the planar map has, from each vertex, a pointer to one of its incident halfedges. We can traverse the halfedges around a vertex by using the adjacency pointers of the halfedges. If a vertex v_i has d incident halfedges then finding the location of a new edge around v_i will take $O(d)$ traversal steps. To avoid the overhead of a search structure for each vertex the planar-maps implementation does not include such a structure. Therefore, since we build the planar map incrementally, if the degree of v_i in the final map is d_i then we performed $\Sigma_1^{d_i} O(i) = O(d_i^2)$ traversal steps on this vertex. Trying to minimize this time over all the vertices we can either try to minimize the maximum degree or the sum of squares of degrees, Σd_i^2. Now, high degree vertices in the decomposition result in high degree vertices in the underlying arrangement, and therefore we try to avoid them. We can apply the same minimization criteria to the vertices of the decomposition.

Optimal triangulation — minimizing the maximum degree. Using dynamic programming we compute a triangulation of the polygon where the maximum degree of a vertex $MAX(d_i)$ is minimal. The algorithm is described in [26], and runs in $O(n^3)$ time.

Optimal triangulation — minimizing Σd_i^2 . We adapted the minimal-maximum-degree algorithm to find the triangulation with minimum Σd_i^2 where d_i is the degree of vertex v_i of the polygon. See Figure 4.1. In the min-max degree triangulation the dynamic programming scheme apply recursively the triangulation algorithm on smaller parts of the polygon and

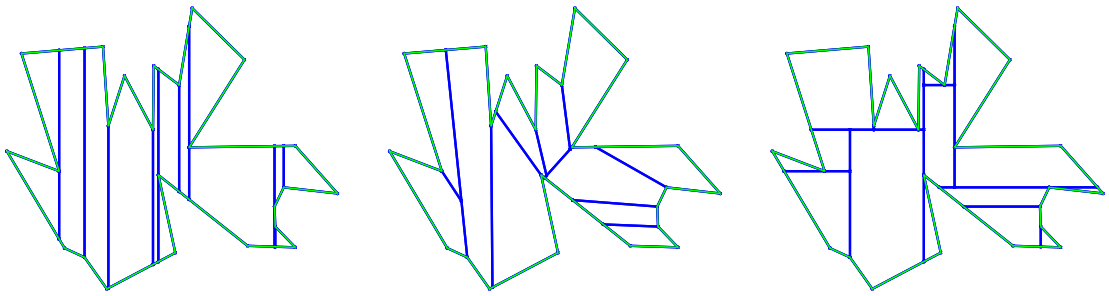


Figure 4.2: From left to right: Slab decomposition, angle bisector (AB) decomposition, and KD decomposition

computes the maximum degree over all the returned triangulations. The modification that is used to give the minimum Σd_i^2 is done only in the final step. We compute the sum of squares of degrees instead of maximum degree. Since both Σd_i^2 and $MAX(d_i)$ are global properties of the decomposition that can be updated in constant time at each step of the dynamic programming algorithm — the rest of the algorithm and its analysis remain the same.

4.1.2 Convex Decomposition without Steiner Points

Greedy convex decomposition. The same as the greedy triangulation algorithm except that it stops as soon as the polygon does not have a reflex vertex.

Minimum number of convex subpolygons (min-convex). We apply the algorithm of Keil [30] which computes a decomposition of a polygon into the minimum number of convex subpolygons without introducing new vertices (Steiner points). The running time of the algorithm is $O(r^2 n \log n)$. This algorithm uses dynamic programming. See Figure 4.1. This result was recently improved to $O(n + r^2 \min\{r^2, n\})$ [32].

Minimum Σd_i^2 convex decomposition. We modified Keil's algorithm so that it will compute decompositions that minimize Σd_i^2, the sum of squares of vertex degree. Like the modification of the min-max degree triangulation, in this case we also modify the dynamic programming scheme by simply replacing the cost function of the decomposition. Instead of computing the number of polygons (as the original min-convex decomposition algorithm does) we compute a different global property, namely the sum of squares of degrees. We can compute Σd_i^2 in constant time given the values Σd_i^2 of the decompositions of two subpolygons.

4.1.3 Convex Decomposition with Steiner Points

Slab decomposition. Given a direction $\vec{e}$, from each reflex vertex of the polygon we extend a segment in directions $\vec{e}$ and $-\vec{e}$ inside the polygon until it hits the polygon boundary. The result is a decomposition of the polygon into convex slabs. If $\vec{e}$ is vertical then this is the

well-known vertical decomposition of the polygon. See Figure 4.2. The obvious advantage of this decomposition is its simplicity.

Angle "bisector" decomposition (AB). In this algorithm we extend the internal angle "bisector" from each reflex vertex until we first hit the polygon's boundary or a diagonal that we have already extended from another vertex[2]. See Figure 4.2. This decomposition (suggested by Chazelle and Dobkin [10]) gives a 2-approximation to the optimal convex decomposition: If P has r reflex vertices then every decomposition of P must include at least $\lceil r/2 \rceil + 1$ subpolygons, since every reflex vertex should be eliminated by at least one diagonal incident to it and each diagonal can eliminate at most 2 reflex vertices. The AB decomposition method extends one diagonal from each reflex vertex until P is decomposed into at most $r + 1$ convex subpolygons.

KD decomposition. This algorithm is inspired by the KD-tree method to partition a set of points in the plane [12]. First we divide the polygon by extending vertical rays inside the polygon from a reflex vertex horizontally in the middle (the number of vertices to the left of a vertex v, namely having smaller x-coordinate than v's, is denoted v_l and the number of vertices to the right of v is denoted v_r. We look for a reflex vertex v for which $\max\{v_l, v_r\}$ is minimal). Then we divide each of the subpolygons by extending an horizontal line from a vertex vertically in the middle. We continue dividing the subpolygons that way (alternating between horizontal and vertical division) until no reflex vertices remain. See Figure 4.2. By this method we try to lower the *stabbing number* of the subdivision (namely, the maximum number of subpolygons in the subdivision intersected by any line) — see the discussion in Section 4.3.2 below. The decomposition is similar to the quad-tree based approximation algorithms for computing the minimum-length Steiner triangulations [14].

4.2 A First Round of Experiments

We present experimental results of applying the decompositions described in the previous section to a collection of input pairs of polygons. We summarize the results and draw conclusions that lead us to focus on a smaller set of decomposition methods (which we study further in the next section). The implementation and test platform details are given in Chapter 3.

We ran the union algorithms (arrangement and incremental-union) with all nine decomposition methods on the input data described in Section 3.3. The running times for the computation of the Minkowski sum for four input examples are summarized in Figures 4.3 through 4.6.

It is obvious from the experimental results that triangulations result in poor union

[2]It is not necessary to compute exactly the direction of the angle bisector, it suffice to find a segment that will eliminate the reflex vertex from which it is extended. Let v be a reflex vertex and let u (w) be the previous (resp. next) vertex on the boundary of the polygon then a segment at the direction $\overrightarrow{uv} + \overrightarrow{wv}$ divides the angle $\angle uvw$ into two angles with less than 180° each.

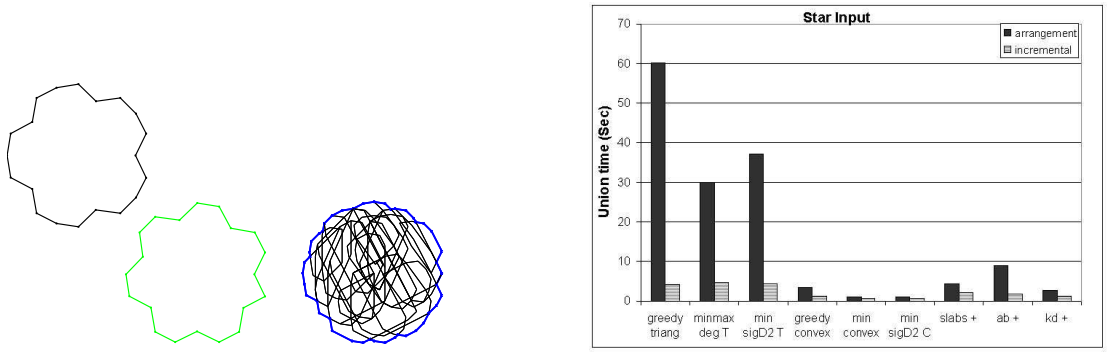


Figure 4.3: Star input: The input (on the left-hand side) consists of two star-shaped polygons. The underlying arrangement of the Minkowski sum is shown in the middle. Running times in seconds for different decomposition methods (for two star polygons with 20 vertices each) are presented in the graph on the right-hand side.

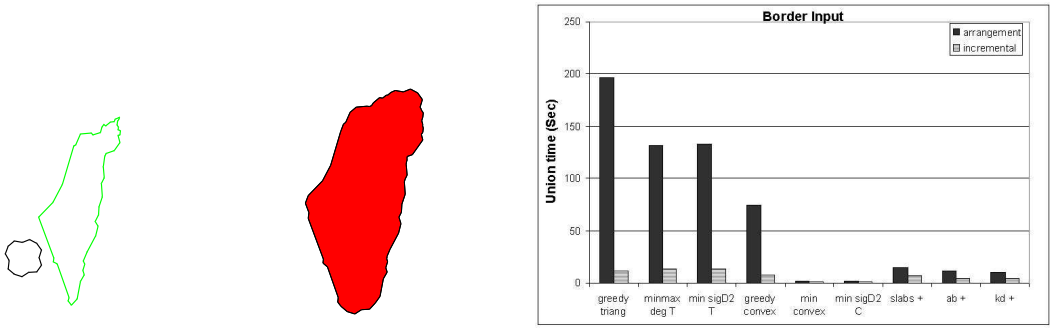


Figure 4.4: Border input: The input (an example on the left-hand side) consists of a border of a country and a star shaped polygon. The Minkowski sum is shown in the middle, and running times in seconds for different decomposition methods (for the border of Israel with 50 vertices and a star shaped polygon with 15 vertices) are shown in the graph on the right-hand side.

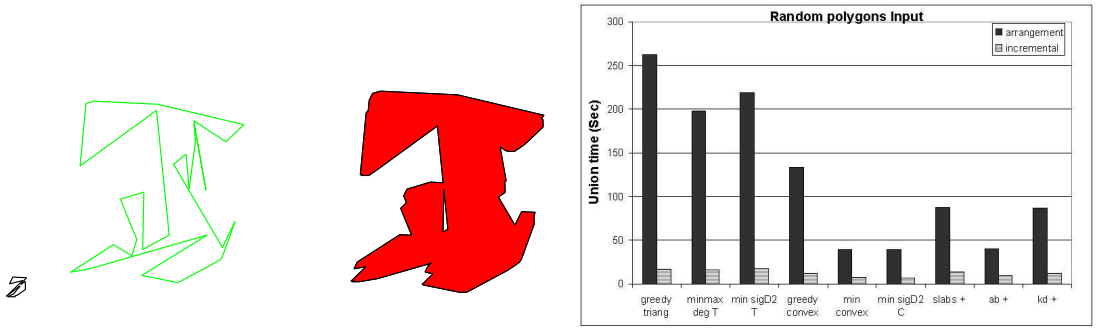


Figure 4.5: Random polygons input: The input (an example on the left-hand side) consists of two random looking polygons. The Minkowski sum is shown in the middle, and running times in seconds for different decomposition methods (for two random looking polygons with 30 vertices each) are shown in the graph on the right-hand side.

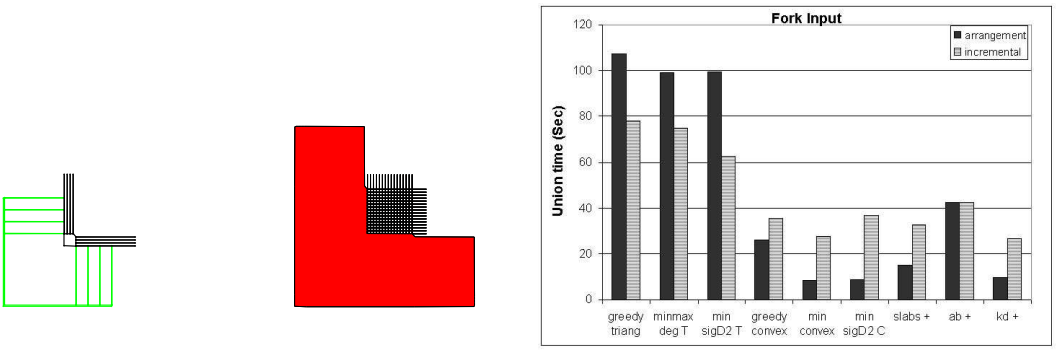


Figure 4.6: Fork input: The input (on the left-hand side) consists of two orthogonal fork polygons. The Minkowski sum is shown in the middle, and running times in seconds for different decomposition methods (for two fork polygons with 8 teeth each) are shown in the graph on the right-hand side.

running times (the left three pairs of columns in the histograms of Figures 4.3 through 4.6). By triangulating the polygons, we create $(n-1)(m-1)$ hexagons in R with potentially $\Omega(m^2n^2)$ intersections between the edges of these polygons. We get those poor results since the performance of the union algorithms strongly depends on the number of vertices in the arrangement of the hexagon edges. Minimizing the maximum degree or the sum of squares of degrees in a triangulation is a slow computation that results in better union performance (compared to the naive triangulation) but is still much worse than other simple convex-decomposition techniques.

In most cases the arrangement union algorithm runs much slower than the incremental union approach. By removing redundant edges from the partial sum during the insertion of polygons, we reduce the number of intersections of new polygons and the current planar map features. The fork input is an exception since the complexity of the union is roughly the same as the complexity of the underlying arrangement and the edges that we remove in the incremental algorithm do not significantly reduce the complexity of the planar map; see Figure 4.6. More details on the comparison between the arrangement union algorithm and the incremental union algorithm are given in Chapter 3.

The min-convex algorithm almost always gives the best union computation time but constructing this optimal decomposition may be expensive — see Figure 4.7. Minimizing the sum of squares of degrees in a convex decomposition rarely results in a decomposition that is different from the min-convex decomposition.

This first round of experiments helped us to filter out inefficient methods. In the next section we focus on the better decomposition algorithms (i.e., minimum convex, slab, angle "bisector", KD), we further study them and attempt to improve their performance.

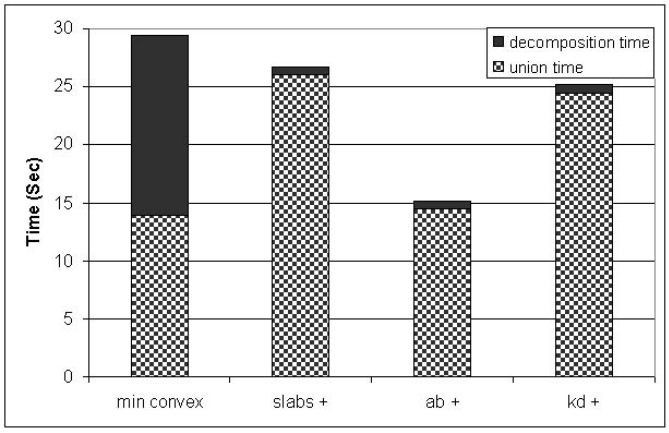


Figure 4.7: When using the min-convex decomposition the union computation time is the smallest but it becomes inefficient when considering the decomposition time as well (running times in seconds for two star polygons with 100 vertices each)

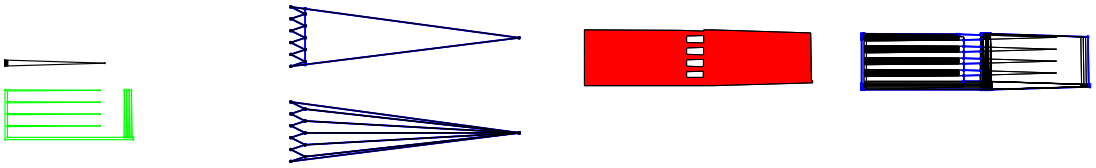
4.3 Revisiting the More Efficient Algorithms

In this section we focus our attention on the algorithms that were found to be efficient in the first round of experiments. As already mentioned, we measure efficiency by combining the running times of the decomposition step together with the union step. We present an experiment that shows that, contrary to the impression that the first round of results may give, minimizing the number of convex subpolygons in the decomposition does not always lead to better Minkowski-sum computation time.

We also show in this section that in certain instances the decision how to decompose the input polygon P may change depending on the other polygon Q, namely for the same P and different Q's we should decompose P differently based on properties of the other polygon. This leads us to propose a "mixed" objective function for the simultaneous optimal decomposition of the two input polygons. We present an optimization procedure for this mixed function. Finally, we take the two most effective decomposition algorithms (AB and KD) — not only are they efficient, they are also very simple and therefore easy to modify — and we try to improve them by adding various heuristics.

4.3.1 Nonoptimality of Min-Convex Decompositions

Minimizing the number of convex parts of P and Q can be expensive to compute, but it does not always yield the best running time of the Minkowski-sum construction. In some cases other factors are important as well. Consider for example the knife input data. P is a long triangle with j teeth along its base and Q is composed of horizontal and



	$j+1$ **long decomposition**	$j+2$ **short decomposition**
number of vertices	23448	9379
running time (sec)	71.7	25.6

Figure 4.8: Knife input: The input polygons are on the left-hand side. Two types of decompositions of P (enlarged) are shown second left: on top, $j+2$ subpolygons with short diagonals length, and below minimum convex decomposition with $j+1$ subpolygons with long diagonals. Third from the left is the Minkowski sum of P and Q. The underlying arrangement (using the short decomposition of P) is shown on the right-hand side. The table below presents the number of vertices in the underlying arrangement and the running time for both decompositions (P has 20 teeth and 42 vertices and Q has 34 vertices).

vertical teeth. See Figure 4.8. P can be decomposed into $j+1$ convex parts by extending diagonals from the teeth in the base to the apex of the polygon. Alternatively, we can decompose it into $j+2$ convex subpolygons with short diagonals (this is the "minimal length AB" decomposition described below in Section 4.3.3). If we fix the decomposition of Q, the latter decomposition of P results in considerably faster Minkowski-sum running time, despite having more subpolygons, because the Minkowski sum of the long subpolygons in the first decomposition with the subpolygons of Q results in many intersections between the edges of polygons in R. In the first decomposition we have $j+1$ long subpolygons while in the latter we have $j+2$ subpolygons when only one of them is a "long" subpolygon and the rest are $j+1$ small subpolygons.

We can also see a similar behavior in real-life data. Computing the Minkowski sum of the countries borders with star polygons mostly worked faster while using the KD-decomposition than with the AB technique; the KD decomposition always generates at least as many subpolygons as the AB decomposition.

4.3.2 Mixed Objective Functions

Good decomposition techniques that handle P and Q separately might not be sufficient because what constitutes a good decomposition of P depends on Q. We measured the running time for computing the Minkowski sum of a knife polygon P (Figure 4.8 — the knife polygon is second left) and a random polygon Q (Figure 3.5). We scaled Q differently in each test. We fixed the decomposition of Q and decomposed the knife polygon P once with the short $j+2$ "minimal length AB" decomposition and then with the long $j+1$ minimum convex decomposition. The results are presented in Figure 4.9. We can see that

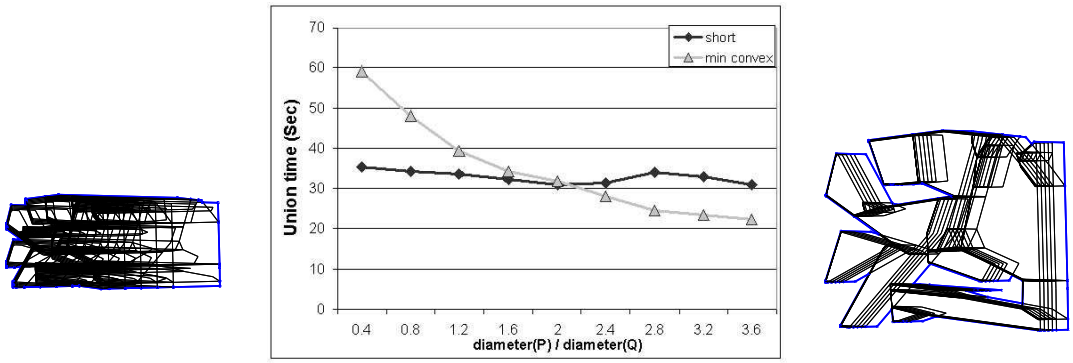


Figure 4.9: Minkowski sum of a knife, P, with 22 vertices and a random polygon, Q, with 40 vertices using the arrangement union algorithm. On the left-hand side the underlying arrangement of the sum with the smallest random polygon and on the right-hand side the underlying arrangement of the sum with the largest random polygon. As Q grows, the number of vertices I in the underlying arrangement is dropping from (about) 15000 to 5000 for the "long" decomposition of P, and from 10000 to 8000 for the "short" decomposition.

for small Q's the short decomposition of the knife P with more subpolygons performs better but as Q grows the long decomposition of P with fewer subpolygons wins.

These experiments imply that a more careful strategy would be to simultaneously decompose the two input polygons, or at least take into consideration properties of one polygon when decomposing the other.

The running time of the arrangement union algorithm is $O(I + k \log k)$, where k is the number of edges of the polygons in R and I is the overall number of intersections between (edges of) polygons in R (see Section 3.2). The value of k depends on the complexity of the convex decompositions of P and Q. Hence, we want to keep this complexity small. It is harder to optimize the value of I. Intuitively, we want each edge of R to intersect as few polygons of R as possible. If we consider the standard rigid-motion invariant measure μ on lines in the plane [41] and use $L(C)$ to denote the set of lines intersecting a set C, then for any polygon R_{ij}, $\mu(L(R_{ij}))$ is the perimeter of R_{ij}. This suggests that we want to minimize the total lengths of the diagonals in the convex decompositions of P and Q. (Aronov and Fortune [4] use this approach to show that minimizing the length of a triangulation can decrease the complexity of the average case ray shooting query.) But we want to minimize the two criteria simultaneously, and let the decomposition of one polygon govern the decomposition of the other.

We can see supporting experimental results for segments in Figure 4.10. In these experiments we randomly chose a set T of points inside a square in $\mathbb{R}^2$ and connected pairs of them by a set S of random segments (for each segment we randomly chose its two endpoints from T). Then we measured the average number of intersections per segment as a function

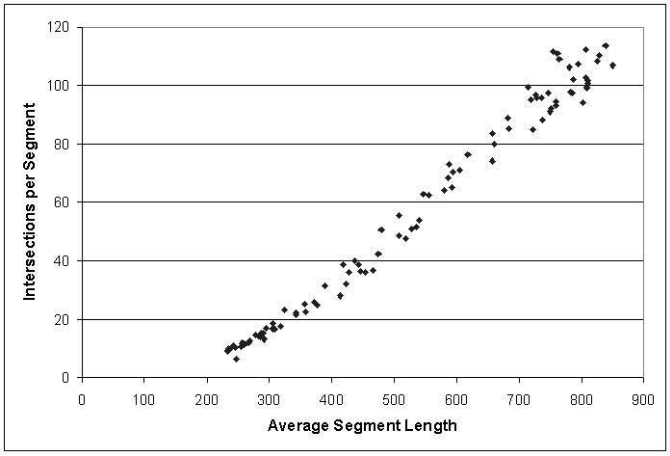


Figure 4.10: Average number of intersections per segment as a function of the average segment length. The configuration contains 125 randomly chosen points in a square $[0,1000] \times [0,1000]$ in $\mathbb{R}^2$ and 500 randomly chosen segments connecting pairs of these points.

of the average length of a segment. To get different average length of the segments, at each round we chose each segment by taking the longest (or shortest) segment out of l randomly chosen segments, where l is a small integer varying between 1 and 15. The average number of intersections is $\frac{I}{|S|}$ where I is the total number of intersections in the arrangement $\mathcal{A}(S)$. We performed 5 experiments for each value of l between 1 and 15, each plotted point in the graph in Figure 4.10 represents such an experiment. The values of l are not shown in the graph — it was used to generate sets of segments with different average lengths. In the case of the arrangement $\mathcal{A}(R)$ of the polygons of R we have $O(mn)$ endpoints and k segments, where $\min(m,n) \leq k \leq 6mn$. For the presented results we took $|S| = 4|T|$. As the results show, the intersection count per segment grows linearly (or close to linearly) with the average length of a segment.

Therefore, we assume that the expected number of intersection of a segment in the arrangement $\mathcal{A}(R)$ of the polygons of R is proportional to the total length of edges of $\mathcal{A}(R)$ which we denote by $\pi_{\mathcal{A}(R)}$. The intuition behind the mixed objective function which we propose next, is that minimizing $\pi_{\mathcal{A}(R)}$ will lead to minimizing I.

Let $P_1, P_2, ..., P_{k_P}$ be the convex subpolygons into which P is decomposed. Let π_{P_i} be the perimeter of P_i. Similarly define $Q_1, Q_2, ..., Q_{k_Q}$ and π_{Q_j}. If $\pi_{R_{ij}}$ is the perimeter of R_{ij} (the Minkowski sum of P_i and Q_j) then

$$\pi_{R_{ij}} = \pi_{P_i} + \pi_{Q_j}$$

Summing over all (i,j) we get

$$\pi_{\mathcal{A}(R)} = \Sigma_{ij}\pi_{R_{ij}} = \Sigma_{ij}(\pi_{P_i} + \pi_{Q_j}) = k_Q(\Sigma_i \pi_{P_i}) + k_P(\Sigma_j \pi_{Q_j})$$

Let π_P denote the perimeter of P and Δ_P denote the sum of the lengths of the diagonals in P. Similarly define π_Q and Δ_Q. Let $D_{P,Q}$ be the decomposition of P and Q. Then

$$c(D_{P,Q}) = \pi_{\mathcal{A}(R)} = k_Q(2\Delta_P + \pi_P) + k_P(2\Delta_Q + \pi_Q).$$

The function $c(D_{P,Q})$ is a cost function of a simultaneous convex decomposition of P and Q. Our empirical results showed that this cost function approximates the running time of the arrangement algorithm. We want to find a decomposition that minimizes this cost function. Let $c^* = \min_{D_{P,Q}} c(D_{P,Q})$.

If we do not allow Steiner points, we can modify the dynamic-programming algorithm by Keil [30] to compute c^* in $O(n^2 r_P^4 + m^2 r_Q^4)$ as follows. We define an auxiliary cost function $\hat{c}(P,i)$, which is the minimum total length of diagonals in a convex decomposition of P into at most i convex polygons. Then

$$c^* = \min_{i,j}[j(2\hat{c}(P,i) + \pi_P) + i(2\hat{c}(Q,j) + \pi_Q)].$$

Since the number of convex subpolygons in any minimal convex decomposition of a simple polygon is at most twice the number of the reflex vertices in it, the values i and j are at most $2r_P$ and $2r_Q$, respectively, where r_P (resp. r_Q) is the number of reflex vertices in P (resp. Q). One can compute $\hat{c}(P,i)$ by modifying Keil's algorithm [30] — the modified algorithm as well as the algorithm for computing c^* are described in detail in Appendix C.

Since the running time of this procedure is too high to be practical, we did not implement it nor did we make any serious attempt to improve the running time. We regard this algorithm as a first step towards developing efficient algorithms for approximating mixed objective functions.

If we allow Steiner points, then it is an open question whether an optimal decomposition can be computed in polynomial time. Currently, we do not even have a constant-factor approximation algorithm. The difficulty arises because unlike the minimum-size decomposition for which an optimal algorithm is known [10], no constant-factor approximation is known for minimum-length convex decomposition of a simple polygon if Steiner points are allowed [31].

4.3.3 Improving the AB and KD methods

It seems from most of the tests that in general the AB and KD decomposition algorithms work better than the other heuristics. We next describe our attempts to improve these algorithms.

Minimal length angle "bisector" decomposition. In each step we handle one reflex vertex. For a reflex vertex we look for one or two diagonals that will eliminate it. We choose the shortest combination among the eliminators we have found. As we can see in Figure 4.12, the *minimal length AB* decomposition performs better than the naive AB even though it generally creates more subpolygons.

While the AB decomposition performs very well, in some cases (concave chains, countries borders) the KD algorithm performs better. We developed the KD-decomposition technique aiming to minimize the stabbing number of the decomposition of the input polygons (which in turn, as discussed above, we expect to reduce the overall number I of intersections in the underlying arrangement $\mathcal{A}(R)$ of the polygons of R). This method however often generates too many convex parts. We tried to combine these two algorithms as follows.

Angle "bisector" and KD decomposition (AB+KD). In this algorithm we extend a "bisector" from each reflex vertex that both its neighbors are convex vertices. We apply the KD decomposition algorithm for the remaining non-convex polygons. By this method we aim to lower the stabbing number without creating redundant convex polygons in the sections of the polygons that are not bounded by concave chains). We tested these algorithms on polygons with different number of convex vertices, vertices in concave chains and "tooth vertices". We can see from the results in Figure 4.11 that AB+KD performs best when the numbers of vertices in concave chains and tooth vertices are the same. When there are more tooth vertices than vertices in concave chains, then the AB decomposition performs better.

Next, we tried to further decrease the number of convex subpolygons generated by the decomposition algorithm. Instead of emanating a diagonal from any reflex vertex, we first tested whether we can eliminate two reflex vertices with one diagonal (let's call such a diagonal a 2-*reflex eliminator*). All the methods listed below generate at most the same number of subpolygons generated by the AB algorithm but practically the number is likely to be smaller.

Improved angle "bisector" decomposition. For a reflex vertex, we look for 2-reflex eliminators. If we cannot find such a diagonal we continue as in the standard AB algorithm.

Reflex angle "bisector" decomposition. In this method we work harder trying to find 2-reflex eliminator diagonals. In each step we go over all reflex vertices trying to find an eliminator. When there are no more 2-reflex eliminators, we continue with the standard AB algorithm on the rest of the reflex vertices.

Small side angle "bisector" decomposition. As in the *reflex AB* decomposition, we are looking for 2-reflex eliminators. Such an eliminator decomposes the polygon into two parts, one on each of its side. Among the candidate eliminators we choose the one that has the minimal number of reflex vertices on one of its sides. Vertices on different sides of the added diagonal cannot be connected by another diagonal because it will intersect the added diagonal. By choosing this diagonal we are trying to "block" the minimal number of reflex vertices from being connected (and eliminated) by another 2-reflex eliminator diagonal.

Experimental results are shown in Figure 4.12. These latter improvements to the AB decomposition seem to have the largest effect on the union running time, while keeping the decomposition method very simple to understand and implement. Note that the *small side AB* heuristic results in 20% faster union time than the *improved AB* and *reflex AB* decompositions, and 50% faster than the standard *angle "bisector"* method.

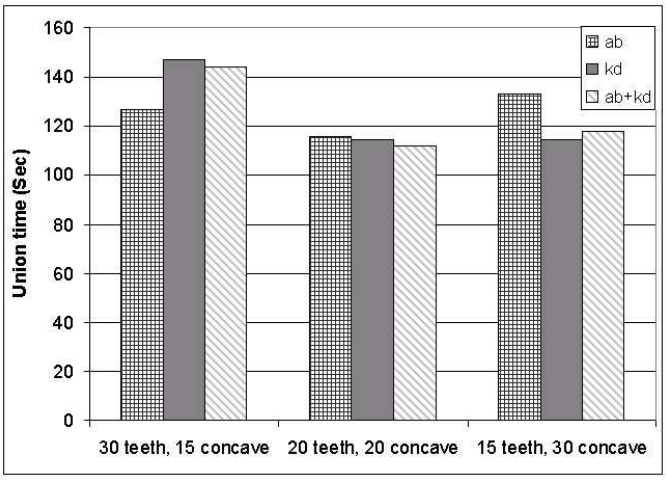


Figure 4.11: Running times in seconds for computing the Minkowski sum of the chain input using AB, KD, and AB+KD decompositions

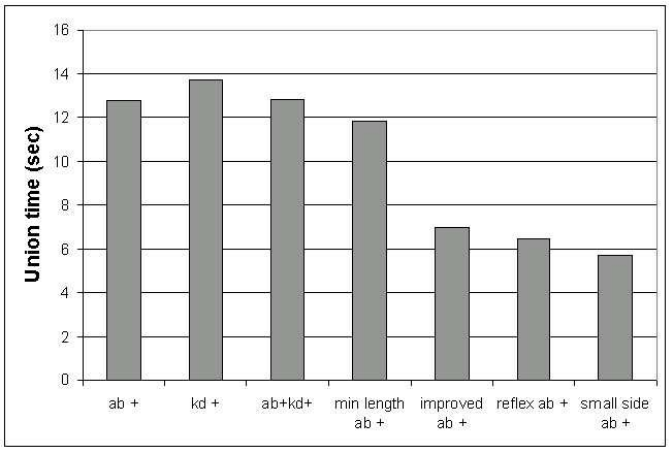


Figure 4.12: Average union running times in seconds for star inputs with the improved decomposition algorithms

Chapter 5

Conclusions

We presented a general scheme for computing the Minkowski sum of polygonal sets. We concentrated on improving the efficiency of this scheme by attacking its two main steps: polygon decomposition and computing the union of polygons.

We implemented three union algorithms which overcome all possible degeneracies. Using exact number types and special handling for geometric degeneracies we obtained a robust and exact implementation that could handle all kinds of polygonal inputs. Our program finds the subdivision of the plane that represents the Minkowski sum of two given polygonal sets and reports the boundary cases where edges or vertices are "semi-free". We compared the efficiency of the algorithms on several inputs. The experimental results imply that if the Minkowski sum is complex compared to the input polygons the arrangement union algorithm gives good results, but if the output is likely to be simpler, the incremental union algorithm is very efficient. If we cannot predict the complexity of a Minkowski sum, the divide-and-conquer algorithm is a good choice, since in the experiments it mostly results in running times that were between the running times of the other two algorithms and in many cases closer to the faster one.

Furthermore, we measured the effect of the decomposition method on the efficiency of the overall process. We implemented over a dozen of decomposition algorithms, among them triangulations, optimal decompositions for different criteria, approximations and heuristics. We examined several criteria that affect the running time of the Minkowski-sum algorithm. The most effective optimization is minimizing the number of convex subpolygons. Thus, triangulations which are widely used in the theoretical literature are not practical for the Minkowski-sum algorithms. We further found that minimizing the number of subpolygons is not always sufficient. Since we deal with two polygonal sets that are participating in the algorithm we found that it is smarter to decompose the polygons simultaneously minimizing a cost function which takes into account the decomposition of both input set. Optimal decompositions for this function and also simpler cost functions like the overall number of convex subpolygons were practically too slow. In some cases the decomposition step of the Minkowski algorithm took more time than the union step. Therefore, we developed

some heuristics that approximate very well a cost function and run much faster than their exact counterparts. Allowing Steiner points, the angle "bisector" decomposition gives a 2-approximation for the minimal number of convex subpolygons. The AB decomposition with simple practical modifications (small-side AB decomposition) is a decomposition that is easy to implement, very fast to execute and gives excellent results in the Minkowski-sum algorithm.

We propose several direction for further research:

1. Use the presented scheme and the practical improvement that we proposed with real-life applications such as motion planning and GIS and examine the effect of different decompositions for those special types of input data.

2. Use the flexibility of CGAL for applying the Minkowski sum algorithms to input sets defined by non-linear curves, for (an easy) example sets whose boundary is composed of line segments and circular arcs.

3. Further improve the AB decomposition algorithms to give better theoretical approximation and better running times.

4. The Minkowski sums of modest-size input sets can be huge. For example, the Minkowski sum of two polygons with about 100 vertices each can be a polygonal region with millions of vertices. We are currently looking for algorithms that approximate the Minkowski sum with fewer edges and vertices. We are specifically looking for conservative approximations (that contain the exact Minkwoski sum) as this is desirable in certain applications such as robot motion planning and assembly planning.

5. We tested the efficiency of the Minkowski-sum algorithm with different convex decomposition methods, but the algorithm will still give a correct answer if we will have a covering of the input polygons by convex polygons. Can one further improve the efficiency of the Minkowski sum program using coverings instead of decompositions.

6. We anticipate that our observations regarding the three union algorithms will be helpful for solving other, similar problems, and in particular for computing the *minimization diagram* [43] induced by lower envelopes in three-dimensional space. It is a challenging task to cast our union algorithms in a general software framework from which the solution for specific problems could be derived with little effort.

Appendix A

Handling Degeneracies in the Union Algorithms

In Chapter 3 we described the union algorithms that we use to construct the Minkowski sum of two polygonal sets out of a set R of convex polygons which are the subsums of pairs of convex subpolygons of the input sets. The result of each union algorithm is a decomposition of the plane into regions that are inside the union and regions that are outside the union. In addition we would like to compute the boundary of the union. This boundary does not always consist of just the edges of the decomposition that separate between inside regions and outside regions. In some cases we can have a boundary edge or a boundary vertex that is surrounded by regions that are contained entirely in the union. These features are not always connected to the 'regular' boundary. In robot motion planning, when the union represents the configuration space obstacle of a polygonal robot moving among polygonal obstacles, the special features represent a tight passage or a singular placement for the robot among the obstacles; see for example Figure 1.7 and Figure 3.1. In this setting, a point on the boundary of the union is called a *semi-free* location. In this appendix we will detail the classification of the special features for the arrangement and incremental union algorithms that we introduced in Chapter 3. We use the notation set up in Chapter 3.

We believe that the technical details that we describe here can be useful in solving degenerate cases in other problems such as computing the minimization diagram representing the lower envelope of surfaces in three dimensions.

A.1 Handling Degeneracies in the Arrangement Union Algorithm

Recall that in the arrangement algorithm we are constructing the arrangement $\mathcal{A}(R)$ of the polygons of R. The union of the polygons of R, Union(R), can be represented by a

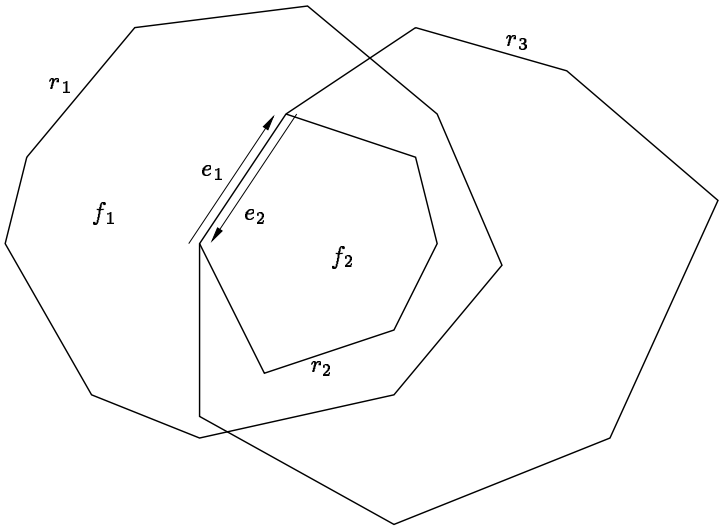


Figure A.1: Example of the *inside count* calculation: the arrangement of $R = \{r_1, r_2, r_3\}$ is drawn in the figure. $BC(e_2) = 2$ because it is on the directed boundary of r_2 and r_3. $BC(e_1) = 0$ because it doesn't lie on a directed boundary of a polygon of R. The inside count of f_1 is 1 since f_1 is contained in r_1. Therefore, when we traverse from f_1 to f_2 through the halfedges e_1 and e_2 we get: $IC(f_2) = IC(f_1) - BC(e_1) + BC(e_2) = 1 - 0 + 2 = 3$. The face f_2 is indeed contained in the three polygons of R.

subset of the features of $\mathcal{A}(R)$. Therefore, we would like to compute for each feature in the underlying arrangement (face, edge or vertex) whether it is inside the union or not. To do this we keep information in the halfedges and apply an update operation after each insertion of a polygon into the arrangement. After inserting all the polygons of R, we traverse the arrangement once and mark the features that are in the union.

We keep for each halfedge e an integer value *boundary count*, denoted $BC(e)$, which counts for each (directed) halfedge in the underlying arrangement on how many boundaries of polygons in R it lies. For a polygon $r \in R$ with vertices $v_1, v_2, \ldots, v_{k_r}$ given in counterclockwise order, r's *directed boundary* is the following sequence of directed edges: $\overrightarrow{v_1v_2}, \overrightarrow{v_2v_3}, \ldots, \overrightarrow{v_{k_r-1}v_{k_r}}, \overrightarrow{v_{k_r}v_1}$ (these directed edges are not yet halfedges of the arrangement). We consider a halfedge e of the arrangement to be on the boundary of r if its direction is the same as the direction of the directed boundary of r. Each halfedge e is initialized with $BC(e) := 0$. After a new polygon r is inserted into the arrangement we increment the *boundary count* for all the halfedges on its boundary.

In the final traversal phase we visit the faces of $\mathcal{A}(R)$ in a BFS order (a face is reached through one of its neighbors). We keep for each face f its *inside count*, denoted $IC(f)$, which is the number of polygons of R in which it lies. We start from the unbounded face whose *inside count* is zero.

If the faces f_1 and f_2 are neighbors sharing an edge which is represented by two halfedges e_1 and e_2 ($twin(e_1) = e_2$) then f_2 lies in the same set of polygons that f_1 lies minus the polygons that we leave by moving from e_1 to e_2 plus the polygons that we enter by crossing this edge. We get the following crossing rule:

Lemma A.1.1 $IC(f_2) = IC(f_1) - BC(e_1) + BC(e_2)$.

See Figure A.1 for an example.

A face f of the arrangement is inside Union(R) if and only if $IC(f) > 0$ (it lies in at least one of the polygons of R). By applying the equation of Lemma A.1.1 during the traversal phase we can compute the *inside count* for all the faces of $\mathcal{A}(R)$.

Next, we would like to know which edges are on the boundary of the union. Trivially we would check for each edge e its two incident faces. If one of those faces is inside the union and the other is not then we would mark e to be on the boundary of the union. But this is insufficient since, as mentioned earlier, there are edges that are on the boundary of one or more polygons of R but are not contained in any of these polygons. Such an edge is on the boundary of the union but the faces on its sides are both inside the union (in some cases such an edge might not be connected to the rest of the boundary of the union). We refer to such an edge as a *semi-free* edge.

Lemma A.1.2 *Let e be an edge in $\mathcal{A}(R)$ represented by the halfedge e_1, then e is semi-free if and only if* $[IC(face(e_1)) - BC(e_1)] = 0$.

Lemma A.1.2 says that if a halfedge e_1 is on the boundary of $BC(e_1)$ polygons in R that contain $face(e_1)$ then e_1 is on the boundary of Union(R) if and only if the *inside count* of $face(e_1)$ is the same as $BC(e_1)$. The halfedge e_1 is contained in (or on the boundary of) all the polygons that contain $face(e_1)$. Also, all the polygons of R that e_1 is on their directed boundary surely contain $face(e_1)$. Therefore for every halfedge we get $IC(face(e_1)) \geq BC(e_1)$. If $IC(face(e_1)) > BC(e_1)$ then we have a polygon $r \in R$ that contains $face(e_1)$ but does not have e_1 on its boundary. In this case e_1 is contained in the interior of r. When there is no such polygon, we get $IC(face(e_1)) = BC(e_1)$ and then e_1 is on the boundary of Union(R). This observation works for the regular cases as well as the "tight passage" cases.

Finally, we should check the vertices. A vertex is on the boundary of Union(R) if it is a source or a target vertex of an edge that is on the boundary of Union(R), or it can be disconnected from the rest of the boundary as explained above. We should know for each pair (v, f) of a vertex v and an incident face f on how many boundaries of polygons of R that contain f, v lies. Let's call this number the *slice count* of (v, f). Since each vertex can have many incident faces, the most appropriate place to keep this information is the halfedge on the boundary of f that is targeted at v. Note that there is exactly one halfedge $e_{(v,f)}$ for which $face(e_{(v,f)}) = f$ and $target(e_{(v,f)}) = v$. We denote the *slice count* of the pair (v, f) by $SC(e_{(v,f)})$. Therefore, we get the following lemma:

Lemma A.1.3 *v is semi-free if and only if there is a face f such that* $[IC(face(e_{(v,f)})) - SC(e_{(v,f)})] = 0$.

We maintain the *slice count* during the insertion of polygons of R into the map: If e_1 and e_2 are adjacent halfedges on the boundary of a polygon in R, sharing a vertex v, we increase $SC(e)$ for each halfedge e that lies clockwise between e_1 and e_2 around v, and such that $target(e) = v$.

The following schema summarizes the details given in this section. Beside the regular pointers and data, the features of the arrangement contain the following information:

feature	property	type	description
face	IC	integer	inside count
	inside	Boolean	is inside Union(R)
halfedge	BC	integer	boundary count
	SC	integer	slice count
	boundary	Boolean	is on the boundary of Union(R)
vertex	boundary	Boolean	is on the boundary of Union(R)

Algorithm ARRANGEMENTUNION(R)

Input: A set $R = \{r_1, r_2, \ldots\}$ of convex polygons
Output: The Arrangement $\mathcal{A}(R)$ in which the faces of the union have their *inside* property set to true and the edges and vertices on its boundary have their *boundary* property set to true.

1. **for** $j \leftarrow 1$ to $size(R)$ **do**
2. $\quad \mathcal{A}$.insert(r_j, BE) // insert the edges of r_j into the arrangement
   BE is an ordered list of the halfedges of $\mathcal{A}$ on the directed boundary of r_j. BE_i is the ith item of BE.
3. $\quad$ **for** $i \leftarrow 1$ to $size(BE)$ **do**
4. $\quad\quad BC(BE_i) \leftarrow BC(BE_i) + 1$
5. $\quad\quad$ **foreach** halfedge h around the vertex $target(BE_i)$ **do**
6. $\quad\quad\quad$ **if** h is clockwise between BE_i and BE_{i+1} **then** $SC(h) \leftarrow SC(h) + 1$
7. $\quad$ **end for**
8. **end for**
9. Traverse $\mathcal{A}$ in BFS order and calculate $IC(f)$ for each face f using $BC(e)$ of the haldeges
10. **foreach** face f in $\mathcal{A}$ **do**

11. **if** $IC(f) > 0$ **then** $inside(f) \leftarrow true$
12. **foreach** halfedge e in $\mathcal{A}$ **do**
13. **if** $IC(face(e)) = BC(e)$ **then** $boundary(e) \leftarrow true$
14. **foreach** halfedge e in $\mathcal{A}$ **do**
15. **if** $IC(face(e)) = SC(target(e))$ **then** $boundary(target(e)) \leftarrow true$
16. **return** $\mathcal{A}$

Complexity Analysis

In line 2 we insert a polygon into the arrangement. In lines 3–7 we traverse the arrangement features of the inserted polygons, namely the halfedges of its directed boundary and the edges that it intersects (lines 5–6). The time we spend on the additional traversal is proportional to the insertion time. If the set R is randomly ordered then we get a total arrangement construction time of $O(I + k \log k)$ (see Appendix B). The traversal in BFS order in line 9 can be carried out in $O(I + k)$ time using the adjacency pointers of the arrangement. In the next lines (10–15) we visit each face once and each halfedge twice. The total time complexity of the algorithm is therefore $O(I + k \log k)$. We use $O(I + k)$ space for storing the arrangement.

A.2 Handling Degeneracies in the Incremental Union Algorithm

In the incremental algorithm we also insert one polygon of R at a time. However, the coloring procedure is executed after each insertion of a polygon into the map. We keep for each face f a boolean value $inside(f)$ that is set if $f \subseteq \text{Union}(R)$. For each halfedge or vertex we keep a boolean value *boundary* that is set if they are on the boundary of Union(R). For each halfedge e we have an additional boolean value $mark(e)$ that is set if e is on the directed boundary of a polygon of R. After inserting a polygon r into the map[1], we set $mark(e)$ for all the edges that are on r's directed boundary.

Let $\mathcal{P}_0$ be an empty planar map. For each polygon r_i of R we perform two steps: (i) insert r_i into the planar map $\mathcal{P}_{i-1}$ obtaining the map $\mathcal{P}'_i$, and (ii) remove map features of $\mathcal{P}'_i$ that do not contribute to the union boundary. The result is the planar map $\mathcal{P}_i$.

To find all the redundant features of $\mathcal{P}'_i$ it suffices to check only the features of $\mathcal{P}'_i$ in (or on the boundary of) r_i.

[1] The implementation of the incremental union algorithm uses an extension of the planar map class that supports intersections.

We denote the set of halfedges of $\mathcal{P}_i'$ on the directed boundary of r_i by $BE(r_i)$. For every halfedge $e \in BE(r_i)$ we set $mark(e)$. By using the incidence pointers of the map we can traverse all the faces, halfedges and vertices that are inside r_i. We denote them by Faces(r_i), Halfedges(r_i) and Vertices(r_i) accordingly. First, we set $inside(f)$ for each face f in Faces(r_i). We can also remove the features Halfedges(r_i) and Vertices(r_i) since they are contained inside a polygon of the union and do not contribute to the union's boundary. Therefore, it remains to count carefully on $BE(r_i)$.

We need to distinguish edges and vertices of the boundary of r_i that contribute to the union boundary (in this stage the union refers to the union of the first i polygons of R). Let e be an edge of $\mathcal{P}_i'$ on the boundary of r_i. We denote the directed halfedge that has the interior of r_i on its left by e_1 and its twin by e_2. Thus $e_1 \in BE(r_i)$. The following lemma is trivial:

Lemma A.2.1 *If* $inside(face(e_1)) \neq inside(face(e_2))$ *then* e *is on the boundary.*

In such a case we do not remove e from the map and we set $boundary(e)$ because it is on the boundary of the union of the polygons inserted so far.

In the rest of the cases an edge e has faces that are inside the union on both its sides. If it has been first inserted to the map when we inserted r_i then it can be removed since both its sides were contained in the union before inserting r_i. If e was part of $\mathcal{P}_{i-1}$ then it is possible that e is on the boundary of two (or more) polygons of R but it is not contained in the union. We can identify it by checking whether the halfedges e_1 and e_2 on e have both $mark(e_1)$ and $mark(e_2)$ set. Therefore,

Lemma A.2.2 *In the map* $\mathcal{P}_i'$, *if* $inside(face(e_1)) = inside(face(e_2))$ *and* $mark(e_2)$ *is set then* e *is on the boundary.*

Finally, we want to identify the vertices on the boundary. Vertices that are endpoints of edges of the boundary of the union are kept with their incident edges. The other vertices that we will keep are those that were not first inserted when we insert r_i. A vertex v was not first inserted with r_i if its degree in $\mathcal{P}_i'$ is greater than two. A boundary vertex that is not connected to other parts of the boundary (see an example in Figure 3.1) will have its degree in $\mathcal{P}_i'$ greater than two and none of its incident halfedges on the boundary (i.e. the halfedges do not satisfy the terms of Lemmas A.2.1 and A.2.2)[2].

The whole process can be carried out by traversing once all the features of the map $\mathcal{P}_i'$ that are inside or on the boundary of r_i. After marking the faces and removing the redundant halfedges we get the planar map $\mathcal{P}_i$ which will be used by the next iteration of the algorithm.

[2]Our planar map representation cannot handle vertices that are not connected to halfedges. Therefore, if a vertex is found to be on the boundary but none of its incident halfedges are on the boundary then we keep a halfedge from $BE(r_i)$ with it. The halfedge that we keep will not be part of the resulting union.

The following schema summarizes the details of this section. Beside the regular pointers and data, the features of the planar map contain the following information:

feature	property	type	description
face	inside	Boolean	is inside Union(R)
halfedge	mark	Boolean	is part of a directed boundary of a polygon
	boundary	Boolean	is on the boundary of a partial union
vertex	boundary	Boolean	is on the boundary of a partial union

Algorithm IncrementalUnion(R)

Input: A set $R = \{r_1, r_2, \ldots\}$ of convex polygons
Output: The planar map $\mathcal{P}$ representing Union(R).

1. $\mathcal{P} \leftarrow$ empty map
2. **for** $j \leftarrow 1$ to $size(R)$ **do**
3. $\quad \mathcal{P}$.insert(r_j, BE) // insert the edges of r_j into the arrangement
   BE is an ordered list of the halfedges of $\mathcal{A}$ on the directed boundary of r_j. BE_i is the ith item of BE.
4. $\quad$ **for** $i \leftarrow 1$ to $size(BE)$ **do**
5. $\quad\quad mark(BE_i) \leftarrow true$
6. $\quad$ **for** $i \leftarrow 1$ to $size(BE)$ **do**
7. $\quad\quad$ **if** $degree(target(BE_i)) > 2$ **then** $boundary(target(BE_i)) \leftarrow true$
8. $\quad$ **foreach** halfedge e of $\mathcal{P}$ inside r_j **do**
9. $\quad\quad \mathcal{P}$.remove(e)
10. $\quad$ **let** f be the face of $\mathcal{P}$ inside r_j
11. $\quad inside(f) \leftarrow true$
12. $\quad$ **for** $i \leftarrow 1$ to $size(BE)$ **do**
13. $\quad\quad$ **let** $DoRemove \leftarrow true$
14. $\quad\quad$ **if** $inside(face(BE_i)) <> inside(face(twin(BE_i)))$ **then**
15. $\quad\quad\quad DoRemove \leftarrow false$
16. $\quad\quad$ **if** $mark(twin(BE_i))$ **then**
17. $\quad\quad\quad DoRemove \leftarrow false$

18. **if** $boundary(target(BE_i))$ **and** $degree(target(BE_i)) = 1$ **then**
19. $DoRemove \leftarrow false$
20. **if** $DoRemove$ **then** $\mathcal{P}$.remove(BE_i) **else** $boundary(BE_i) \leftarrow true$
21. **end for**
22. **end for**
23. **return** $\mathcal{P}$

Complexity Analysis

In each step we insert a polygon r_j into the map (line 3) and traverse the halfedges that we added additional $O(1)$ times (lines 4–7 and 12–21). In lines 8–9 we remove the edges that are inside r_j. Since every edge is removed only once, we can charge each removal of an edge to an edge of the arrangement $\mathcal{A}(R)$ and therefore we have $O(I + k)$ removals. Using Mulmuley's dynamic search structure [39] we can give an upper bound of $O(k^2 \log^2 k)$ for the expected time of the construction. As mentioned in Chapter 3 this bound is worse than that of the arrangement union algorithm but in practice the incremental union algorithm often performs much better than the arrangement algorithm. We use $O(I + k)$ space for the planar map.

Appendix B

Proof of Theorem 3.2.1: Construction Time of Arrangements of Convex Polygons

Given a set R of convex polygons we wish to construct the arrangement $\mathcal{A}(R)$. To do this efficiently we construct the trapezoidal decomposition $H(R)$ induced by the edges of R by a randomized incremental algorithm. $H(R)$ is a refinement of $\mathcal{A}(R)$ having the same asymptotic complexity. The running time of the algorithm presented in Section 3.2 is $O(I + k \log k)$ where k is the number of edges of the polygons in R and I is the number of intersections among the edges of the polygons of R.

We follow the notation of Mulmuley [39]. Each polygon of R can be decomposed into two x-monotone chains. Let $N = \{s_1, s_2, \ldots, s_n\}$ be the set of n x-monotone chains composing the polygons of R, with a total of $\overline{n}$ segments. Let κ be the number of intersections between pairs of chains from N. We will randomly permute N and insert one chain at a time. Assume $s_1, s_2, \ldots, s_n$ is a random permutation. We will construct $H(N)$ incrementally storing the result and the intermediate decomposition in a doubly connected edge list (DCEL).

Let $N^i = \{s_1, s_2, \ldots, s_i\}$ and $H(N^i)$ be the trapezoidal decomposition of N^i. We will add s_{i+1} to the decomposition to get $H(N^{i+1})$. Let's assume that we are given the trapezoid in $H(N^i)$ that contains the leftmost endpoint of s_{i+1}. We can insert the chain s_{i+1} by visiting all the trapezoids of $H(N^i)$ that intersect s_{i+1}. We can traverse the faces of $H(N^i)$ that intersect s_{i+1} by the adjacency pointers of the DCEL. Traversing a face f along s_{i+1} takes $O(T(f) + m(s_{i+1}, f))$ time where $T(f)$ is the complexity of the face f in $H(N^i)$ and $m(s_{i+1}, f))$ is the number of segments of the chain s_{i+1} that are inside f. We need to split every such traversed face by vertical lines from intersection points of s_{i+1} with the upper and lower sides of f and from endpoints of segments of s_{i+1} inside f. The overall traversal and splitting procedure takes $O(|s_{i+1}| + \sum_f T(f))$ for all the faces $f \in H(N^i)$ such that $f \cap s_{i+1} \neq \emptyset$. To get $H(N^{i+1})$ we should contract the vertical attachments in

$H(N^i)$ that are intersected by s_{i+1}. Contracting a vertical attachment merges the faces adjacent to it into one face. The vertical attachment that was contracted now ends at its intersection with s_{i+1}. The merging takes $O(1)$ time for each contracted attachment. Hence, the overall insertion step will take $O(|s_{i+1}| + \sum_f T(f))$ for all the faces $f \in H(N^i)$ such that $f \cap s_{i+1} \neq \emptyset$.

We also need to locate the face of $H(N^i)$ in which the leftmost endpoint of s_{i+1} lies. We maintain a conflict list $L(f)$ for every face of $H(N^i)$. $L(f)$ will contain all the leftmost endpoints of the chains of $N \setminus N^i$. We denote the number of points in $L(f)$ by $l(f)$. With these lists we can determine in which face an endpoint of a chain lies in $O(1)$. To maintain the conflict lists we have to pay $O(l(f))$ time for every face that we change during the insertion. That gives a total insertion time for s_{i+1} of $O(|s_{i+1}| + \sum_f [T(f) + l(f)])$ for all the faces $f \in H(N^i)$ such that $f \cap s_{i+1} \neq \emptyset$.

The cost of inserting the $(i+1)$st chain can be rewritten as $O(|s_{i+1}| + \sum_g [T(g) + l(g)])$ where g ranges over all trapezoids in $H(N^{i+1})$ adjacent to s_{i+1}. Each chain in N^{i+1} is equally likely to be involved in the $(i+1)$st insertion. Hence, this conditional expected cost is proportional to

$$C_{i+1} = \frac{1}{i+1} \sum_{s \in N^{i+1}} \{|s_{i+1}| + \sum_g [T(g) + l(g)]\}$$

where g ranges over all trapezoids in $H(N^{i+1})$ adjacent to s_{i+1}. Each trapezoid is adjacent to at most four chains in N^{i+1}. We denote the number of segments in the chains of N^{i+1} by $\overline{n}_{i+1}$ and the number of intersections between them by κ_{i+1}. We get

$$C_{i+1} \leq \frac{4}{i+1} \{\overline{n}_{i+1} + \sum_g T(g) + \sum_g l(g)\}$$

where g ranges over all trapezoids in $H(N^{i+1})$.

$$\sum_g T(g) = O(|H(N^{i+1})|) \quad \text{and therefore} \quad \sum_g T(g) = O(\kappa_{i+1} + \overline{n}_{i+1}) \ .$$

where $|H(N^{i+1})|$ is the complexity of $H(N^{i+1})$.

The conflict lists in $H(N^{i+1})$ contain all the leftmost endpoints that were not yet inserted. Therefore:

$$\sum_g l(g) = n - i \ .$$

Thus we get

$$C_{i+1} = O(\frac{\kappa_{i+1} + \overline{n}_{i+1} + n - i}{i+1}) \ .$$

Let $0 < j \leq n$ be a fixed integer. For any fixed intersection v between chains of N let I_v be a $0-1$ random variable such that $I_v = 1$ if and only if v occurs in $\mathcal{A}(N^j)$. Clearly

$\kappa_j = \sum I_v$ where v ranges over all intersections among chains of N. v occurs in $\mathcal{A}(N^j)$ if and only if both chains that cause this intersection are in N^j. This happens with probability $O(j^2/n^2)$. The expected value of I_v is $O(j^2/n^2)$. Therefore, by linearity of the expectation we get $\kappa_j = O(\kappa j^2/n^2)$.

We also define a random variable I_i to count the number of segments that the ith chain contributes to N^j. $I_i = |s_i|$ when $s_i \in N^j$ and 0 otherwise. s_i is in N^j with probability j/n and therefore the expected value of I_i is $|s_i|j/n$. Clearly $\overline{n}_j = \sum I_i$. By linearity of expectation we get $\overline{n}_j = O(\overline{n}j/n)$.

Therefore the expected value of C_{i+1} is (we set $j := i+1$):

$$O(\frac{\kappa j^2/n^2 + \overline{n}j/n + n - j + 1}{j})$$

The expected cost of the whole algorithm is then

$$O(\sum_{j=1}^{n} \frac{\kappa j^2/n^2 + \overline{n}j/n + n - j + 1}{j}) = O(\frac{\kappa}{n^2}\sum_{j=1}^{n} j + \frac{\overline{n}}{n}\sum_{j=1}^{n} 1 + n\sum_{j=1}^{n}\frac{1}{j} - \sum_{j=1}^{n} 1 + \sum_{j=1}^{n}\frac{1}{j}) =$$

$$= O(\kappa + \overline{n} + n\log n)$$

Using this algorithm we can incrementally construct the arrangement of the polygons of R by randomly permuting the polygons in R and then inserting them as pairs of monotone chains. After each insertion we can update the additional data that is used for the calculation of the union of R (see Appendix A). Let k be the number of edges of polygons of R and I be the number of intersections among them. Clearly the number of polygons in R is $O(k)$ then the overall time complexity will be $O(I + k\log k)$.

Appendix C

Polygons Decomposition Minimizing the Mixed Objective Function

In Section 4.3.2 we developed a mixed objective function for the decomposition of the two input polygons to the Minkowski sum computation. In this Appendix we describe an algorithm based on the optimal convex decomposition of Keil [30] for decomposing the input polygons simultaneously minimizing the mixed objective function[1]. Here we do not allow Steiner points.

The running time of the arrangement union algorithm is $O(I + k \log k)$, where k is the number of edges of the polygons in R and I is the overall number of intersections between (edges of) polygons in R (see Section 3.2). The value of k depends on the complexity of the convex decompositions of P and Q. Hence, we want to keep this complexity small. Furthermore, we want to reduce the value of I. Intuitively, we want each edge of R to intersect as few polygons of R as possible. If we consider the standard rigid-motion invariant measure μ on lines in the plane [41] and use $L(C)$ to denote the set of lines intersecting a set C, then for any polygon R_{ij}, $\mu(L(R_{ij}))$ is the perimeter of R_{ij}. This suggests that we want to minimize the total lengths of the diagonals in the convex decompositions of P and Q. But we want to minimize the two criteria simultaneously, and let the decomposition of one polygon govern the decomposition of the other.

First we recall the notation of Section 4.3.2. Given two simple polygons P and Q with m and n vertices respectively. Let $D_{P,Q}$ be a decomposition of P and Q into convex subpolygons. Let $P_1, P_2, ..., P_{k_P}$ be the convex subpolygons into which P is decomposed. Let π_{P_i} be the perimeter of P_i. Let π_P denote the perimeter of P and Δ_P denote the sum of

[1] The algorithm described here was proposed to us by Pankaj K. Agarwal.

the lengths of the diagonals in P. Similarly define $Q_1, Q_2, ..., Q_{k_Q}$, π_{Q_j}, π_Q and Δ_Q. Then

$$c(D_{P,Q}) = k_Q(2\Delta_P + \pi_P) + k_P(2\Delta_Q + \pi_Q).$$

The function $c(D_{P,Q})$ is a cost function of a simultaneous convex decomposition of P and Q. Let $c^* = \min_{D_{P,Q}} c(D_{P,Q})$. We present an algorithm that finds a decomposition that meets this minimum.

We define an auxiliary cost function $\hat{c}(P, a)$, which is the minimum total length of diagonals in a convex decomposition of P into at most a convex polygons. Then

$$c^* = \min_{a,b}[b(2\hat{c}(P, a) + \pi_P) + a(2\hat{c}(Q, b) + \pi_Q)].$$

Since the number of convex subpolygons in any minimal convex decomposition of a simple polygon is at most twice the number of the reflex vertices in it, the values a and b are at most $2r_P$ and $2r_Q$ respectively, where r_P (resp. r_Q) is the number of reflex vertices in P (resp. Q). Assuming that we know the decompositions of P and Q that achieve $\hat{c}(P, a)$ and $\hat{c}(Q, b)$, respectively, for every a and b, we can compute c^* and find the decomposition in $O(r_P r_Q)$ time.

The single issue that we need to resolve is how to compute $\hat{c}(P, a)$. In the following section we describe a dynamic-programing algorithm to compute the minimum length decomposition of a polygon (based on [30]) and in Section C.2 we describe how to modify this algorithm for computing $\hat{c}(P, a)$.

C.1 Minimum Length Decomposition

Let $v_1, v_2, \ldots, v_m$ be the vertices of P given in clockwise order. We call a pair (i, j) valid if v_i is visible from v_j and at least one of the two vertices is a reflex vertex. If two vertices are visible from each other and they are not a valid pair then they must both be convex. A diagonal that connects two convex vertices is redundant in any optimal convex decomposition because it can be removed and the two convex subpolygons on its sides can be merged into a convex polygon. Therefore, for the construction of an optimal decomposition, we can consider only the diagonals that connect two vertices that are a valid pair. For a valid pair (i, j), let P_{ij} be the polygonal chain from vertex v_i to v_j. $P_{1n} = P$ is also a valid chain. Let $d(i, j)$ be the length of the diagonal (i, j).

Let $f(i, j)$ denote the cost of the minimum length decomposition of P_{ij}; $f(i, j)$ only counts the length of diagonals added. For a convex decomposition of P_{ij}, let C_{ij} be the *base* convex polygon that contains the edge (i, j). Let (i, k) and (l, j) be the first and the last edges of C_{ij}; see Figure C.1. The pair (i, k) should be a valid pair unless $k = i + 1$. Similarly, the pair (l, j) should be a valid pair unless $l = j - 1$.

We define a function $F(i, j; k, l)$ as follows: $F(i, j; k, l)$ is the cost of a minimum weight decomposition under the constraints that (i, k) is the first edge of C_{ij} and (l, j) is the last

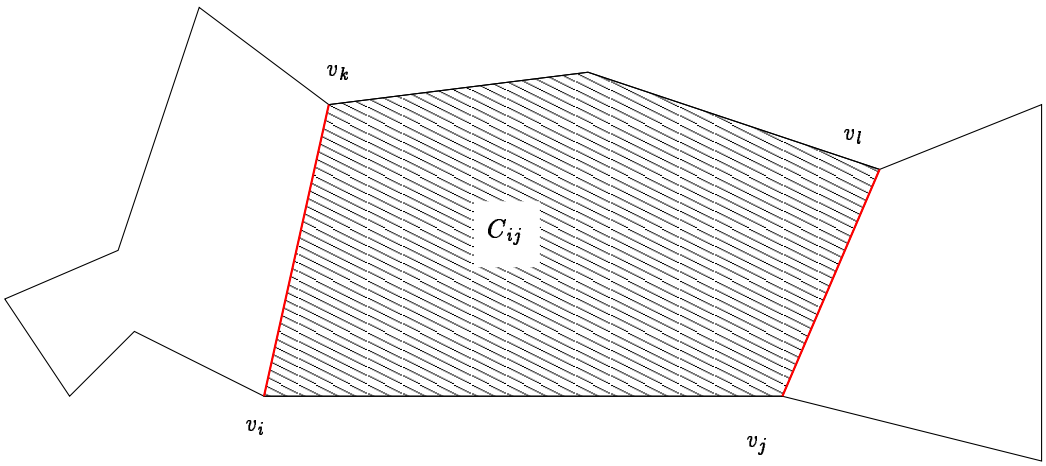


Figure C.1: The base subpolygon C_{ij} of P_{ij} with (i,k) as its first edge and (l,j) as its last edge

edge of C_{ij}. If $k \neq i+1$, then (i,k) has to be a valid pair, and a similar condition holds for the pair (l,j).

If the angle (j,i,k) or (l,j,i) is greater than 180°, we set $F(i,j;k,l)$ to infinity, as it is not a valid convex decomposition. Then

$$f(i,j) = \min_{k,l} F(i,j;k,l)$$

We need to compute $F(i,j;k,l)$ for at most $m^2 r_P^2$ pairs because if i is reflex (convex), then k is any (resp. reflex) vertex. The same condition holds for l and j.

We can compute the values of F using the following recursive formula:

$$F(i,j;k,l) = d(l,j) + f(l,j) + \min_g F(i,l;k,g)\ , \tag{C.1}$$

where the minimum is taken over all vertices g such that (g,l) is a valid pair (or $g = l-1$) and the angle $(g,l,j) \leq 180^\circ$. The recurrence uses a minimum decomposition of P_{lj} along with a minimum decomposition of P_{il} for which the first edge of C_{il} is (i,k) and the last edge is (g,l). g is chosen only if the polygon C_{il} can merge with the triangle T_{ilj}. Since both T_{ilj} and C_{il} are convex it is sufficient to verify that the angles $(g,l,j), (j,i,k) \leq 180^\circ$. See Figure C.2 for an illustration.

To complete the algorithm we first need to find all valid pairs and then compute $F(i,j;k,l)$ for them. The result of the algorithm will be $f(1,m)$. We will compute $F(i,j;k,l)$ in ascending order of the difference $j-i$ using Formula C.1.

Theorem C.1.1 *The minimum length convex decomposition of P can be computed in $O(m^2 r_P^2)$ time.*

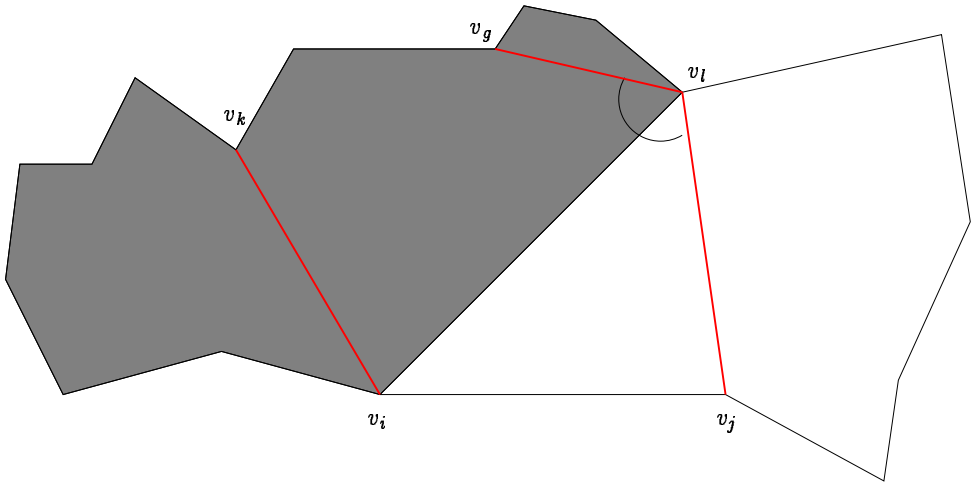


Figure C.2: The recurrence: we compute $F(i,j;k,l)$ using a minimum decomposition of P_{lj} along with a minimum decomposition of P_{il} (shaded) for which the first edge of C_{il} is (i,k) and the last edge is (g,l). The triangle T_{ilj} can merge with the base subpolygon C_{il}.

Proof: For each pair (i,j) we can compute whether v_i is visible from v_j in $O(m)$ time. A potentially valid pair is a pair (i,j) for which at least one of v_i or v_j is reflex. Computing visibility for all potentially valid pairs will therefore take $O(m^2 r_P)$. Sorting all valid pairs in ascending order of the difference between the indices will take an additional $O(m r_P \log m)$ time.

For a fixed quadruple i, j, k, and l, let $g(i,j,k,l)$ denote the index of g that minimizes the recurrence C.1. For a fixed triple i,k, and l, $g(i,j,k,l)$ increases monotonically with j because as we increase j, the angle (j,l,i) can only decrease and more pairs (g,l) become relevant; see Figure C.3. When we use the recurrence C.1 we should only compute the minimum over all relevant g's that are greater than $g(i,j',k,l)$ where j' is the largest index for which (i,j') and (l,j') are valid pairs and $j' < j$. Thus, the amortized time spent in computing each $F(i,j;k,l)$ is $O(1)$. The overall running time of the algorithm is therefore $O(m^2 r_P^2)$. □

C.2 Constrained Minimum Length Decomposition

We slightly change the above algorithm to compute $\hat{c}(P,a)$. We define $F(s,i,j;k,l)$ to be the minimum length convex decomposition of P_{ij} into at most s convex subpolygons, under the constraint that (i,k) is the first edge of the base polygon C_{ij} and that (l,j) is the last edge of C_{ij}. If the angle (j,i,k) or (l,j,i) is greater than 180° or if P_{ij} cannot

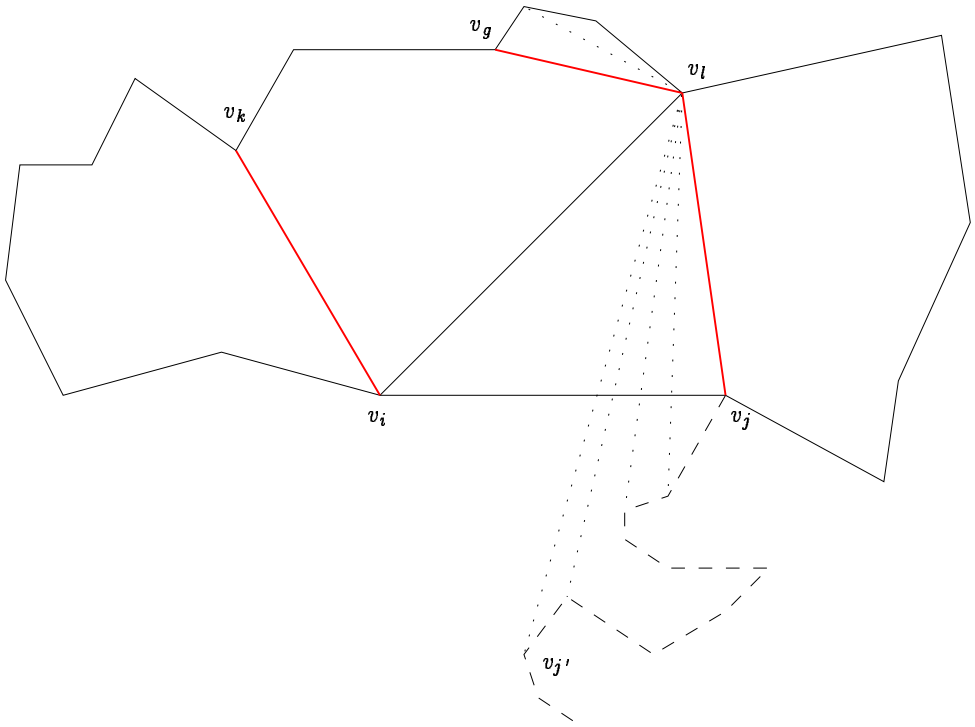


Figure C.3: When j increases the angle (j,l,i) decreases and more valid pairs (g,l) become relevant

be decomposed into at most s convex subpolygons, we set the cost to infinity. We define $f(s,i,j)$ to be the cost of any convex decomposition of P_{ij} with at most s subpolygons.

The recurrence is now given by:

$$F(s,i,j;k,l) = d(l,j) + \min_{u \le s}\{f(u,l,j) + \min_{g} F(s-u,i,l;k,g)\}\,, \tag{C.2}$$

where the minimum is taken over all vertices g such that (g,l) is a valid pair (or $g = l-1$) and the angle $(g,l,j) \le 180°$.

Theorem C.2.1 *The minimum length convex decomposition of P into at most s subpolygons (for every $1 \le s \le 2r_p$) can be computed in $O(m^2 r_P^4)$ time.*

Proof: We use the arguments from the proof of Theorem C.1.1. We now compute $O(m^2 r_P^3)$ entries. The monotonicity condition described above still holds, i.e., for a fixed quadruple s, i, k, l, the value of g increases monotonically with j. So each entry can be computed in $O(r_P)$ amortized time since $s \le 2r_P$, giving a total of $O(m^2 r_P^4)$ time. □

Theorem C.2.2 *A decomposition D_{PQ} of the polygons P and Q that minimizes the mixed function $c(D_{PQ})$ can be computed in time $O(m^2 r_P^4 + n^2 r_Q^4)$.*

Proof: Using the above algorithm we can compute $\hat{c}(P, a) = f(a, 1, m)$ for P for every $a \leq 2r_P$ in $O(m^2 r_P^4)$ time and $\hat{c}(Q, b) = f(b, 1, n)$ for Q for every $b \leq 2r_Q$ in $O(n^2 r_Q^4)$. We need an additional $O(r_P r_Q)$ time to compute c^*, which is subsumed by the other factors of the running time. □

Bibliography

[1] *The CGAL User Manual, Version 2.0*, 1999. http://www.cs.ruu.nl/CGAL.

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

[3] H. Alt, R. Fleischer, M. Kaufmann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Approximate motion planning and the complexity of the boundary of the union of simple geometric figures. *Algorithmica*, 8:391–406, 1992.

[4] B. Aronov and S. Fortune. Average-case ray shooting and minimum weight triangulations. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 203–211, 1997.

[5] M. Austern. *Generic Programming and the STL — Using and Extending the C++ Standard Template Library.* Addison-Wesley, 1999.

[6] A. H. Barrera. Computing the Minkowski sum of monotone polygons. In *Abstracts 12th European Workshop Comput. Geom.*, pages 113–116, Münster, Germany, 1996.

[7] A. H. Barrera. Finding an $o(n^2 \log n)$ algorithm is sometimes hard. In *Proc. 8th Canad. Conf. Comput. Geom.*, pages 289–294. Carleton University Press, Ottawa, Canada, 1996.

[8] M. Bern. Triangulations. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 22, pages 413–428. CRC Press LLC, Boca Raton, FL, 1997.

[9] B. Chazelle. The polygon containment problem. In F. P. Preparata, editor, *Computational Geometry*, volume 1 of *Adv. Comput. Res.*, pages 1–33. JAI Press, London, England, 1983.

[10] B. Chazelle and D. P. Dobkin. Optimal convex decompositions. In G. T. Toussaint, editor, *Computational Geometry*, pages 63–133. North-Holland, Amsterdam, Netherlands, 1985.

[11] M. de Berg and A. van der Stappen. On the fatness of minkowski sums. Technical Report UU-CS-1999-39, Dept. of Computer Science, Utrecht University, 1999.

[12] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[13] G. Elber and M.-S. Kim, editors. *Special Issue of Computer Aided Design: Offsets, Sweeps and Minkowski Sums*, volume 31. 1999.

[14] D. Eppstein. Approximating the minimum weight Steiner triangulation. *Discrete Comput. Geom.*, 11:163–191, 1994.

[15] E. Ezra and E. Flato. Generating random polygons. In preparation, 2000.

[16] A. Fabri, G. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the Computational Geometry Algorithms Library. Technical Report MPI-I-98-1-007, MPI Inform., 1998. To appear in *Software—Practice and Experience*.

[17] E. Flato, D. Halperin, I. Hanniel, and O. Nechushtan. The design and implementation of planar maps in CGAL. In J. Vitter and C. Zaroliagis, editors, *Proceedings of the 3rd Workshop on Algorithm Engineering*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 154–168. Springer-Verlag, 1999. Full version: http://www.math.tau.ac.il/~flato/WaeHtml/index.htm.

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

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

[20] D. Halperin, J.-C. Latombe, and R. H. Wilson. A general framework for assembly planning: The motion space approach. *Algorithmica*, 26:577–601, 2000.

[21] D. Halperin and M. Sharir. A near-quadratic algorithm for planning the motion of a polygon in a polygonal environment. *Discrete Comput. Geom.*, 16:121–134, 1996.

[22] D. Halperin and R. H. Wilson. Assembly partitioning along simple paths: the case of multiple translations. *Advanced Robotics*, 11:127–145, 1997.

[23] I. Hanniel. The design and implementation of planar arrangements of curves in CGAL. Master's thesis, Dept. Comput. Sci., Tel-Aviv Univ., 2000. Forthcoming.

[24] 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.*, pages 91–96, 1995.

[25] E. Hartquist, J. Menon, K. Suresh, H. Voelcker, and J. Zagajac. A computing strategy for applications involving offsets, sweeps, and Minkowski operations. *Comput. Aided Design*, 31(4):175–183, 1999. Special Issue on Offsets, Sweeps and Minkowski Sums.

[26] G. Kant and H. L. Bodlaender. Triangulating planar graphs while minimizing the maximum degree. In *Proc. 3rd Scand. Workshop Algorithm Theory*, volume 621 of *Lecture Notes Comput. Sci.*, pages 258–271. Springer-Verlag, 1992.

[27] A. Kaul, M. A. O'Connor, and V. Srinivasan. Computing Minkowski sums of regular polygons. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 74–77, 1991.

[28] L. E. Kavraki. Computation of configuration-space obstacles using the Fast Fourier Transform. *IEEE Trans. Robot. Autom.*, 11:408–413, 1995.

[29] K. Kedem, R. Livne, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete Comput. Geom.*, 1:59–71, 1986.

[30] J. M. Keil. Decomposing a polygon into simpler components. *SIAM J. Comput.*, 14:799–817, 1985.

[31] J. M. Keil and J.-R. Sack. Minimum decompositions of polygonal objects. In G. T. Toussaint, editor, *Computational Geometry*, pages 197–216. North-Holland, Amsterdam, Netherlands, 1985.

[32] J. M. Keil and J. Snoeyink. On the time bound for convex decomposition of simple polygons. In *Proc. 10th Canad. Conf. Comput. Geom.*, 1998.

[33] M. Keil. Polygon decomposition. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1999.

[34] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.

[35] D. Leven and M. Sharir. Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams. *Discrete Comput. Geom.*, 2:9–31, 1987.

[36] K. Mehlhorn, S. Näher, C. Uhrig, and M. Seel. *The LEDA User Manual, Version 4.0*. Max-Planck-Insitut für Informatik, 66123 Saarbrücken, Germany, 1999.

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

[38] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbolic Comput.*, 10(3-4):253–280, 1990.

[39] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[40] R. Pollack, M. Sharir, and S. Sifrony. Separating two simple polygons by a sequence of translations. *Discrete Comput. Geom.*, 3:123–136, 1988.

[41] L. Santaló. *Integral Probability and Geometric Probability*, volume 1 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, 1979.

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

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

[44] A. Stepanov and M. Lee. The standard template library, Oct. 1995. http://www.cs.rpi.edu/~musser/doc.ps.

[45] M. van Kreveld. Twelve computational geometry problems from cartographic generalization. Manuscript. Presented at the Dagstuhl meeting on Computational Geometry. March, 1999.

[46] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 35, pages 653–668. CRC Press LLC, Boca Raton, FL, 1997.