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

The Design and Implementation of Planar Arrangements of Curves in CGAL

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

by

Iddo Hanniel

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

December 2000

Acknowledgments

I deeply thank Prof. Dan Halperin for his guidance and for introducing me to the field of Computational Geometry and into the CGAL project. I also thank Sigal Raab, Sariel Har-Peled, Oren Nechushtan, Eyal Flato, Lutz Kettner and the rest of the CGAL members who contributed to this work in helpful discussions and comments.

I wish to thank my mother, and of course Shlomit, my wife, for her support.

Contents

Chapter 1

Introduction

This thesis is concerned with a fundamental structure in Computational Geometry — an *arrangement* of curves in the plane. The heart of this work is a generic and robust software package for arrangements of general curves, and an application of this package to adaptive point location in arrangements of parametric curves. We start with some basic definitions.

A *planar map* is a planar embedding of a planar graph G such that each arc of G is embedded as a bounded curve, the image of a node of G is a *vertex*, and the image of an arc is an *edge*. We require that each edge be a bounded x-monotone curve[1]. A *face* of the planar map is a maximal connected region of the plane that does not contain any vertex or edge. Given a finite collection $\mathcal{C}$ of (possibly intersecting and not necessarily x-monotone) curves in the plane, we construct a collection $\mathcal{C}''$ of curves in two steps: First we decompose each curve in $\mathcal{C}$ into maximal x-monotone curves, thus obtaining a collection $\mathcal{C}'$. We then decompose each curve in $\mathcal{C}'$ into maximal connected pieces not intersecting any other curve in $\mathcal{C}'$. This way we obtain the collection $\mathcal{C}''$ of x-monotone curves that do not intersect in their interior. The *arrangement* $\mathcal{A}(\mathcal{C})$ of the curves in $\mathcal{C}$ is the planar map induced by the curves in $\mathcal{C}''$.

Arrangements of lines in the plane, as well as arrangements of hyperplanes in higher dimensions have been extensively studied in computational geometry [15], with applications to a variety of problems. An example of a problem that can be mapped to the construction of planar arrangements of lines is finding the smallest triangle defined by three out of a given set of n points in the plane (the mapping is through a "duality transform" [15]).

[1] We define an x-monotone curve to be a curve that is either a vertical line segment or a Jordan arc such that any vertical line intersects it in at most one point.

The construction of the line arrangement in the dual plane enables to solve this problem in $O(n^2)$ which is an improvement over the naive $O(n^3)$ solution. This result can be generalized to finding the minimum-volume simplex spanned by $d+1$ out of n points in E^d, which can be found in $O(n^d)$, see [15, Chapter 12.4]. Many more applications for arrangements can be found in [15, Part III]. Motivated by such applications, a study of data structures and algorithms for arrangements has been carried out. Typical basic problems in the algorithmic study of arrangements are: constructing a data structure that enables easy traversal of the entire arrangement, efficiently finding the face a query point is located in, efficiently finding the *zone* of a curve[2], to name a few.

Arrangements of general curves have also been studied producing combinatorial results (e.g., the zone theorem) and algorithmic ones (e.g., algorithms for constructing a face in an arrangement) [1, 27, 47]. These algorithms can be applied to many problems in "physical world" application domains. For example, the problem of motion planning of a robot with two degrees of freedom (under reasonable assumptions that the shape of the robot and of the obstacles consists of algebraic arcs of some constant degree), can be solved using an arrangement of algebraic curves. The so-called *configuration space* of the problem — the space of parametric representations of all possible positions of the robot, can be viewed as an arrangement of arcs in 2D. The problem is reduced to finding a single face in the arrangement of arcs — the face that contains the start and end configurations. If such a face exists then any path in it is a solution to the motion planning problem (see [28] and [47] for more details).

As noted above, this thesis deals with an implementation of an arrangements package. Implementors of geometric algorithms and data structures encounter several difficulties. Geometric algorithms are usually described assuming an infinite-precision real arithmetic model of computation. In this model, arithmetic operations, assignments and comparisons on real numbers take constant time. A program implemented using a naive substitution of floating-point arithmetic for real arithmetic can fail, since geometric primitives depend on sign-evaluation and may not be reliable if evaluated approximately. In [35], an example is given where using a floating point implementation of the orientation predicate causes a convex hull algorithm to

[2]The zone of a curve γ in a planar map is the collection of the faces of the map that intersect γ.

give a wrong result. In many cases the program will go into an infinite loop or crash. For geometric algorithms that use low degree predicates (e.g., orientation and lexicographical comparison of input points), the infinite-precision real model can be realized using multiprecision integer or rational software packages (e.g., Gmpz [25] and LEDA's rational number type [34]). Given a constant bound on the input bit length the predicates take $O(1)$ time to compute. However, the underlying constant may be very large.

A strategy to reduce the cost of exact computation is the use of *floating point filters*, which is a general name for a variety of adaptive techniques in which expressions are evaluated with floating point arithmetic, together with a bound on the error of the computation. Exact computation is used only when the result of the floating point computation is insufficient. There are many implementations of such filters, some are specifically tailored for a given predicate (e.g., [48] and the predicates in LEDA's rational kernel [34, 35]) and some are more general and can be used for evaluating general expressions (e.g., [20] and LEDA's `real` number type [10, 34, 35]). Implementations that evaluate the error bound at compile-time are called *static filters* [20] and those that evaluate it at run-time are called *dynamic filters*.

For algorithms dealing with arrangements of curves, a procedure to compute the intersection of two curves is needed. Even for arrangements of lines this presents new problems that were not encountered in simpler algorithms that only use predicates. Computations involving points that are constructed as an intersection of two segments can double the bit length of the coordinates and therefore can be much slower that computations involving only input segment endpoints. For example, when constructing the intersection points of a set of line segments where each point is represented as a pair of multiprecision rational coordinates, each intersection operation can double the bitlength of the numerator and denominator. Constructing the convex hull of the resulting set of intersection points will then be slowed down because of the increase in the size of the coordinates representation (see [10] for a description of such a case). Furthermore, most filtering schemes cannot deal with such cases since they assume a bounded bit length on the input of the predicates.

For algorithms that compute arrangements of algebraic curves the primitive operations involve exact computation of the roots of polynomials of some constant maximum degree (e.g., for finding the intersection of two curves). To evaluate these exactly, rational arithmetic is insufficient. Expensive symbolic techniques need to be applied, which result in a large overhead.

There are not many implementations of arrangements. The Algorithm Design Manual [50] records only two implementations: *arrange* [23] — a package for maintaining arrangements of polygons, and LEDA[3] — the Library of Efficient Data-structures and Algorithms[34, 35]. LEDA provides efficient algorithms (sweep-line and randomized-incremental) to construct the planar map induced by a set of (possibly intersecting) segments. It also supports a subdivision class for planar maps of non-intersecting segments with point location queries. Both implementations are restricted to dealing with linear objects — line segments (in LEDA) and polygons (in the former). Amenta [3, 4] also points to the *pploc* package for answering vertical ray shooting queries in a set of non-intersecting line segments (which seems much more restricted than the other packages).

The MAPC library [33] is a software package for manipulating algebraic points and curves. It has been used to construct the combinatorial structure of an arrangement of algebraic curves. However, it is an algebraic package rather than an arrangement package — it deals with the representation and computation of algebraic curves and their intersections, but does not implement specific data structures and algorithms for arrangements. Neagu and Lacolle [41] provide an algorithm and an implementation that takes as input a set of Bézier curves and outputs a set of polygonal lines whose arrangement has the same combinatorial structure. Their implementation does not build the arrangement data structure.

In this work we present the design and implementation of a generic and robust software package for representing and manipulating arrangements of curves. The package is part of CGAL — the Computational Geometry Algorithms Library[4], which is a collaborative effort of several academic institutes in Europe and Israel [12] to develop a C++ software library of robust geometric data structures and algorithms. The library consists of three main parts. The first part of the library (the kernel) contains basic geometric objects such as points, vectors and lines, predicates on them such as relative positions of points, and operations such as intersections and distance calculation. The next part (the basic library) contains a collection of standard data structures and geometric algorithms, such as convex hull, (Delaunay) triangulation, planar map, polyhedral surface, smallest enclosing circle, and multidimensional query structures. This is the place of the arrangement

[3] http://www.mpi-sb.mpg.de/LEDA/leda.html
[4] http://www.cgal.org/

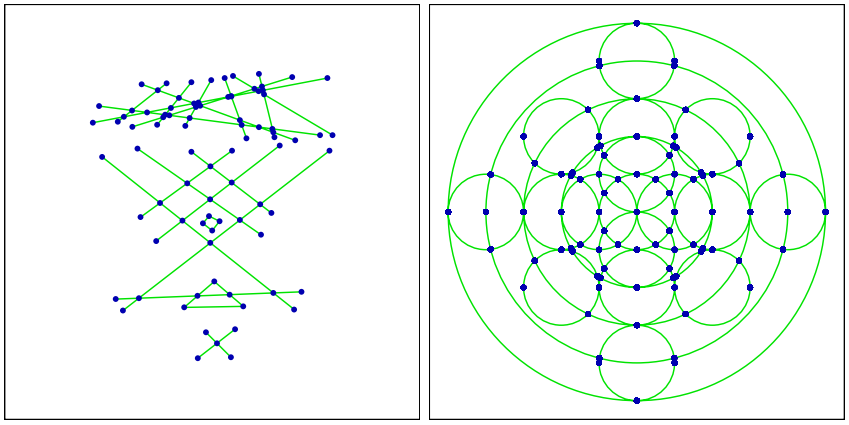


Figure 1.1: An arrangement of segments and an arrangement of circles.

package. The third part (the support library) contains interfaces to other packages, e.g., for visualization, I/O, and other support facilities.

The library's main design goals are robustness, genericity, flexibility and efficiency. To achieve these CGAL adopted the *generic programming paradigm* [5] (see [7, 16] and Section 2.1 for a description of the use of generic programming in CGAL). These goals (particularly the first three) stood before us in our implementation as well.

The arrangement package is built as a layer on top of CGAL's planar map package adding considerable functionality to it. The planar map package supports planar maps of general x-monotone curves that do not intersect in their interior. It does not assume connectivity of the map (i.e., holes inside faces are supported), and enables different strategies for efficient point location. See [18] and Section 2.3 for more details.

There are many problems related to arrangements among them: computing the combinatorial structure of the arrangement, constructing an arrangement in a way that enables efficient traversal over its components, constructing substructures in the arrangement (e.g., the zone of a curve or lower/upper envelopes[5]), or locating a point in the arrangement. Each problem has a representation that is more suited for its solution. The representation used in our

[5]If we regard each curve c_i in the arrangement as a graph of a continuous univariate function $c_i(x)$, then the *lower envelope* of the arrangement is the pointwise minimum of these functions, and the *upper envelope* is the pointwise maximum. See [27, 47] for more details.

arrangement package enables efficient traversal over the planar map induced by the arrangement, while maintaining the information of the original input curves. For example, traversal over all edges in the planar map that originate from the same curve is easily achieved. Like the planar map package, it supports holes and point location and vertical ray shooting queries. Using it, we have implemented arrangements of different curves: line segments, circle arcs (see Figure 1.1), polygonal lines and others.

We also present an application based on this package to adaptive and efficient point location in arrangements of algebraic parametric curves, provided they meet certain conditions. An example is given for arrangements of quadratic Bézier curves. The underlying idea is to approximate the Bézier curves with enclosing polygons and do the operations on these polygons. If the enclosing polygons do not enable us to solve the problem, a subdivision is performed on the polygons which gives a finer approximation of the original Bézier curves. We resort to operations on the actual Bézier curves only if we cannot resolve the queries otherwise (or if we pass some user defined threshold). In this sense our scheme resembles the floating point filter schemes that work with floating point approximations (such as interval arithmetic [8]), and resort to (slow) exact arithmetic only when the computation cannot be resolved otherwise. Indeed our work was inspired by these schemes. A similar scheme was introduced in [26] to speed-up kinetic simulations that involve polynomial functions.

In Chapter 2 we introduce the problems that we attack in our work and related work that has been done on the subject in the past. In Chapter 3 we discuss the design and implementation of the arrangement package in CGAL. The data structures and algorithms used by the adaptive point location application are presented in Chapter 4. Chapter 5 gives results from experiments conducted on that application. In Chapter 6 we present another application for generic boolean operations that demonstrates the usage of the arrangement package. Some concluding remarks are given in Chapter 7. A paper describing this thesis [30] has been recently presented in the 4th Workshop on Algorithm Engineering[6].

[6] A copy of the paper, along with its PowerPoint presentation, and the software related to it can be found in http://www.math.tau.ac.il/~hanniel/ARRG00/ .

Chapter 2

Preliminaries and Related Work

This thesis deals with the implementation of arrangements of curves. There has been a vast body of work on arrangements from a theoretical view which we will not review here. The interested reader is referred to [15] for a survey of the theoretical work on arrangements of lines and hyperplanes; algorithmic and combinatorial results on arrangements of general curves and their applications, can be found in [47]. In this chapter we describe related work that has been done in the implementation of computational geometry algorithms and data structures. We introduce CGAL — the Computational Geometry Algorithm Library, and focus on CGAL's planar map package which is the basis for the arrangements package presented in this work. We also summarize some of the results obtained in [41] for isolating intersections of convex parametric curves, which we use in our adaptive point location application.

2.1 CGAL and Generic Programming

As described in Chapter 1, the CGAL library consists of three main parts: the kernel, the basic library and the support library. The main design goals of the library are robustness genericity, flexibility, efficiency and ease-of-use. Two techniques are available in C++ for realizing generic and flexible designs: *Object-oriented programming*, using inheritance from base classes with virtual member functions, and *generic programming* [5] , using template classes and functions. These paradigms are sometimes referred to as *run-time poly-*

morphism and *static polymorphism* respectively.

The *Object-oriented programming paradigm* defines the interface explicitly with a base class that the derived implementations must follow. It also enables flexibility at runtime rather than compile time, e.g., managing polymorphic data structures such as lists of classes that are derived from the same base class. Its main disadvantages are the additional memory for each object (the so-called *virtual function table pointer*), and the additional indirection through the virtual function table for each call to a virtual member function. Since virtual functions cannot usually be made *inline* this indirection has a run-time penalty (see [36, Item 24] for an overview of the overhead due to inheritance). Examples where this overhead is noticeable are low-dimensional vector arithmetic which appears frequently in scientific programming (see for example, [49]) and traversals of combinatorial data structures.

The *generic programming paradigm* uses what is known in C++ as *templates* (or parameterized types). It provides strong type checking at compile time, does not need extra storage and allows inline member functions. There is no formal scheme in the language for expressing the requirements of template arguments, this is left to the program documentation. Geometric objects, such as points and segments, are anticipated to be small objects with simple member functions. Geometric predicates (e.g., orientation or lexicographical comparison) are also anticipated to be simple functions. Therefore, the overhead due to virtual member functions might not be negligible for these cases. Schirra [45] demonstrates the advantage of using generic programming over object-oriented programming for an interface to geometric predicates. In his case study the overhead introduced by virtual functions was greater than the overhead introduced by using exact computation with floating point filters (see Section 2.2.2). Furthermore, the use of templates enables to run the same algorithm with types from existing libraries without having to derive them from a common base class. CGAL is implemented according to the generic programming paradigm.

A good example of generic programming is the Standard Template Library, STL [5, 39, 51]. The main source of its genericity and flexibility is its separation of *concepts* from *models*. A *concept* is a set of requirements that define a class. A *model* of the concept is a class that fulfills those requirements. For example, an iterator is an abstract concept defined in terms of requirements (e.g., it should have ++ and $*$ operators, etc.) The usual C-pointer is a model of an iterator, since it fulfills those requirements. The STL algorithms are defined in terms of iterators, thus enabling to use the

same algorithm for different containers (e.g., for lists and vectors).

In CGAL all algorithms and data structures of the basic library are passed an additional template parameter. This parameter, the so-called *traits* class[1], makes it possible to use the algorithms in a flexible manner. In particular, we can let it operate on geometric types other than the ones provided by the CGAL kernel (e.g., with other geometric libraries). The traits class is a concept that defines the geometric interface to the class or function. The set of requirements defining it are given in the documentation of the class. Every class also provides a traits model that uses the CGAL types. For some classes other models are also provided (e.g., traits that use the LEDA rational kernel, see Section 2.3.1). For more detailed information on the use of generic programming in CGAL see [7].

2.2 Robustness in Geometric Computation

Computational geometry algorithms described in the literature are usually designed and proven to be correct in a computational model that assumes exact arithmetic over the real numbers where each operation takes constant time — the "real RAM" model of computation. Implementing such algorithms with floating point arithmetic can cause the program to crash, loop forever, or simply compute a wrong result. This is because of the special nature of geometric algorithms. Computational geometry algorithms generally operate on numerical data and construct combinatorial structures from it .If we take the convex hull problem as an example, then we have numeric data, the points' coordinates, and a combinatorial result, the convex hull polygon which is an ordered sequence of a subset of the input points. If we use floating point arithmetic then the result might not be convex, or there might be points left out of the result polygon (see [35, Chapter 9]).

The numerical computations of a geometric algorithm are basically of two types: *predicates* and *constructions*. Predicates are associated with branching decisions and determine the flow of the algorithm, whereas constructions are needed to produce the output data. From an implementation point of view, predicates are usually implemented as functions that take a few geometric objects as input and return a boolean or a multi-state type (typically an *enum* type), whereas constructions are implemented as functions that re-

[1] In CGAL, the term "traits class" is used in a more general sense than associating related type information to built-in types, which is the original usage [40].

turn another geometric object. Thus approximations in the evaluation of predicates may produce an incorrect branching of the algorithm and lead to catastrophic consequences. Approximations on the evaluation of constructions might give acceptable results, as long as their maximal absolute error does not exceed the resolution required by the application. However, passing the approximated results of a construction as input to a predicate in a cascaded computation can lead again to grave consequences. Furthermore, in such input the error evaluation is harder since we need to take into account the accumulated error.

2.2.1 Exact Arithmetic

A straightforward solution to the problems incurred by using floating point arithmetic is to use exact arithmetic, a solution that was advocated in a series of papers by Yap [53, 54] and others. Many geometric algorithms can be written in terms of integer (with homogeneous coordinates) or rational (with Cartesian coordinates) arithmetic. However, for all but the simplest geometric computations the required bitlength of the integer will exceed the built-in machine precision. This can be solved by using multiprecision arithmetic software packages. The major disadvantage of using exact arithmetic is its large performance cost. In the literature, slow down factors of 1000 to 10000 have been reported for computation of Delaunay triangulations with rational arithmetic compared to floating point arithmetic (for inputs that did not crash when using floating point arithmetic) [31]. Since then, a number of techniques for more efficient exact computation have been developed and explored and modern rational software packages have reduced the slow down factor. Schirra [45] gives some comparisons of such packages for convex hull algorithms.

There are several software packages that supply multiprecision integers. Among them are the *gnu multiprecision package* [25] and LEDA's *integer* and *rational* number types [34, 35]. The CGAL kernels (both Cartesian and homogeneous) are template-parameterized with the number type class. This enables the CGAL library to use with its kernel any number type, provided that it has certain functions for it [12]. In particular these functions have been implemented for the above multiprecision packages and they can therefore be run within CGAL's kernel.

The packages described above are restricted to rational arithmetic. However, for some geometric applications computation of roots is required. For

example, when implementing an arrangement of circles a square root operation is required to implement the intersection of two curves. For these, more sophisticated tools are needed. LEDA's `real` number type [10, 11] is such a tool. It operates on algebraic numbers given by expressions involving arbitrary kth roots of integral degree k. It is exact in the sense that all comparisons and signs are computed exactly as in the Real RAM model of computation. This is achieved by using arbitrary-precision floating point arithmetic (in this case LEDA's `bigfloat` number type), and *separation bounds* for zero detection. The separation bound of an expression E is a positive number q such that if $|E| < q$ then E is zero. Thus, if we can compute q, we can repeat evaluating E with increasing precision until we can verify that either $|E| > 0$ and we can evaluate its sign, or $|E| < q$ in which case E is zero, see [10] for more details. The representation of the number is an "expression DAG", where the leaves correspond to integers and each internal node is labeled with an operation. Another package that supports root operations is the `Real/Expr` package [42, 43, 54] which uses similar methods. In [10], a comparison of these two packages is performed on an implementation of the single source shortest path algorithm for a grid with $\sqrt{2}$-length diagonals (hence the need for square roots).

2.2.2 Floating Point Filters

As mentioned above, floating point arithmetic is fast but unreliable while multiprecision arithmetic is exact but slow. However, in many cases the evaluation of the predicate with floating point arithmetic will yield correct results. This leads to *floating point filters* [14, 20] where we compute error bounds on the floating point computations. Only computations that are subject to precision errors are reevaluated by exact arithmetic or tighter approximation. This way, we earn the speed of floating point arithmetic and degrade to slow exact arithmetic only when essential.

Floating point filtering is a general name for a variety of solutions that implement predicates in an adaptive scheme. There are several categories of filters:

- Static filters — compute error bounds at compile-time and need specific information on the input data to be available, for example, whether all input data are integers of a bounded range. Since the error bound is computed at compile-time, the only additional operation at run-time

is a comparison with the error bound, and therefore it is very efficient at run-time.

- Semi-dynamic filters — bounds on the sizes of the operands are computed at run-time. Some factors in the error bound, that depend on the expression only, are still computed at compile-time. Semi-dynamic filters give better error bounds than static filters.

- Fully-dynamic filters — the error bound is completely determined at run-time.

An example of a static filter is the LN package [20, 21]. There the authors implement an expression pre-compiler that computes the error-bounds of any given expression at compile-time, and produces the filtered code for it. A less general example for a static filter is CGAL's `Fixed_precision_nt` number type [12]. This number type assumes the input data is bounded by 24 bits and uses the internal `float` type as the inner representation. It implements the 2D and 3D orientation and in-sphere predicates based on this knowledge. If the static filter fails a more refined semi-dynamic filter is used before resorting to exact computation.

Semi-dynamic filters are used in `rat_leda` — LEDA's rational geometry kernel[2]. The predicates in this kernel compute part of the error bounds based on the expression of the predicate, and the other part is computed at compile time based on the actual input data. Another example of semi-dynamic filters are the orientation and in-circle predicates provided by Shewchuk [48] for points with Cartesian double coordinates (not necessarily integral). These predicates use a different implementation of multiprecision floating-point arithmetic, which enable to implement a multi-layered filter, where each layer uses the results obtained by the previous layers. The implementation is fine-tuned for the given predicates and the results are very efficient.

Fully-dynamic filters are implemented in the `leda_real` number type [10, 35]. The error bound is completely determined at run-time and there is no need to derive error bounds when implementing the predicates. Interval arithmetic [2, 8] can also be used as a fully dynamic filter: First, all computations in a predicate are done with interval arithmetic. Only if the computed intervals do not allow for reliable comparisons (e.g., comparing two values

[2]The reader should not confuse the *leda_rational* number type with LEDA's rational geometry kernel. The latter uses a different representation and employs floating point filters.

that are represented as overlapping intervals), the computation is repeated in a second step using exact arithmetic. In CGAL this technique is used in the predicates for the number type `Filtered_exact<CN,EN>`. The number type `CN` is used to store the coordinates. Inside the predicate, the numerical data is first converted into the `Interval_nt` number type [12] which is an interval with endpoints of type `double`. Then the predicate is evaluated using interval arithmetic. If during this evaluation, a comparison operation cannot be reliably resolved, an exception is thrown (using the C++ exception handling mechanism), which is then caught by a code block that repeats the computation using the exact number type `EN`. The interval arithmetic operations are implemented using the built-in rounding modes (towards plus and minus infinity) which makes them very efficient [45]. This scheme can be implemented for any predicate function given by the user using a script that converts a regular predicate function to a filtered one.

Unlike algorithms that perform their predicates directly on the input data (e.g., convex hull and Delaunay triangulations), in the construction of arrangements predicates can get intersection points as their input. This presents a difficulty to most of the filters described above. In order to enable the use of filters we must either store the input data from which the point was constructed, or compute the construction exactly and perform the filtered predicates on the exact result. A number type that can be used in cascaded computation is `leda_real`, since it stores the expression DAG. LEDA's rational kernel can also be used since it computes the intersection points exactly, and performs the filtered predicates on the result. However, some of the implementations mentioned above cannot be used for arrangements without modifications. For example, in the `Filtered_exact` predicates described above, the non-exact input number type is converted into the inner interval representation and the operations are performed on this representation. If an intersection point is constructed it is not returned in the interval representation and so if there is an error it will not be accumulated. Furthermore, if the non-exact evaluation fails, the exact evaluation does not have the original data to work on.

Even in the cases where the filters do work for cascaded computations some care should be taken in the design of the algorithm. An unbounded cascaded computation (i.e., using the result of a construction as input to another construction and so on) is also generally a bad idea. It can cause the expression tree to grow (in representations such as `leda_real`), and in cases where normalization is not performed will cause an "explosion" of the

bitlength. Therefore, the design should try to bound the depth of computations, e.g., by storing the original input data and performing the constructions on it. Normalizing of rational (or homogeneous) coordinates should also be used in order to reduce the numbers' bitlength. If normalization is not performed then every operation may double the bitlength of the numbers which causes a considerable slowdown in the computations.

2.3 Planar Maps in CGAL

In this section we describe CGAL's planar map package. The data structure for representing planar maps in CGAL supports traversal over faces, edges and vertices of the map, traversal over the boundary edges of a face and around the edges incident to a vertex and efficient point location. The design is flexible enabling the user to define his/her own special curves as long as they support the traits class requirements.

The representation is based on the *Doubly Connected Edge List* (DCEL) structure [13, Chapter 2]. This representation belongs to a family of edge-based data structures in which each edge is represented as a pair of opposite *halfedges*. The representation supports inner components (holes) inside the faces making it general and suitable for a wide range of applications (e.g., geographical information systems). A detailed description of the design and implementation of the planar maps package in CGAL can be found in [18].

2.3.1 Geometric Traits

The documentation of the planar map class gives the precise requirements that every traits class should obey. We have formulated the requirements so they make as little assumptions on the curves as possible (for example, linearity of the curves is not assumed). This enables the users to define their own traits classes for different kinds of curves that they need for their applications. The only restriction is that they obey the predefined interface.

The planar map traits class defines the two basic geometric objects of the map: the point (`Point`) and the x-monotone curve (`X_curve`). In addition four types of predicates are needed[3]:

- Access to the endpoints of the x-monotone curves.

[3]The full list of requirements can be found in [12].

- Comparison predicates between points (e.g., comparing their x-coordinates).
- Comparison predicates between points and x-curves (e.g., whether a point is above a curve).
- Predicates between curves (e.g., comparing the y-coordinate of two curves at a given x-coordinate).

The interface of the four types of predicates satisfies the geometric needs of the planar map. We shall see in Section 3.3 that for arrangements additional predicates and constructors are needed (e.g., construction of the intersection of two curves).

2.3.2 Point Location Strategies

The planar maps package provides three implementations of point location algorithms and a mechanism (the so-called *strategy pattern* [22]) for the users to implement their own point location algorithm. The *abstract* strategy class `Pm_point_location_base<Planar_map>` is a pure virtual class declaring the interface between the algorithm and the planar map. The planar map keeps a reference to the strategy. The *concrete* strategy is derived from the abstract class and implements the algorithm interface.

We have derived the following strategies: `Pm_default_point_location`, `Pm_naive_point_location` and `Pm_walk_along_line_point_location`, which is an improvement over the naive one. Each strategy is preferable in different situations. The *default* class implements the fully dynamic randomized incremental algorithm introduced by Mulmuley [37, 46]. The *naive* algorithm goes over all the vertices and halfedges of the planar map. Namely, in order to find the edge directly above a query point q (i.e., the answer to a *vertical ray-shooting query*), we go over all edges of the map and find the one that is above q and has the smallest vertical distance from it. Therefore, the time complexity of a query with the naive class is linear in the complexity of the planar map. The *walk* algorithm implements a walk over the zone of a vertical ray emanating from the query point. This decreases the number of edges visited during the query, thus improving the time complexity.

There are several trade-offs between the strategies. The main trade-off between the default strategy and the two other strategies, is between time and storage. The naive and walk algorithms generally need more time but

almost no additional storage. Another trade-off that is relevant to our application is when the user has knowledge of the combinatorial structure of the map. Combinatorial operations such as `split_edge`, `insert_at_vertices` and `remove_edge`, take an additional $O(log^2(n))$ expected time with the default strategy, since the internal structure has to be updated; for the walk and naive strategies no such overhead is incurred. For algorithms that make extensive use of those operations this can be significant. Furthermore, since the default algorithm assumes a random sequence of insertions, some applications may create insertion sequences with highly unbalanced search structures. For more information on the point location strategies see [18].

2.4 Computing the Combinatorial Structure of an Arrangement

Neagu and Lacolle [41] describe an algorithm for computing the combinatorial structure of arrangements of curves that satisfy certain conditions (see below). In their work they prove that given an arrangement of such curves, a combinatorially equivalent arrangement of polygonal lines exists and can be constructed via a subdivision process. The main idea of their approach is to avoid the use of algebraic methods on the curves themselves, performing operations only on the polygons bounding the curves. They establish sufficient conditions for equivalence of the arrangements and provide an algorithm and and implementation that compute a polyline arrangement that is equivalent to an arrangement of Bézier curves. In Section 2.4.1 we describe the family of curves they deal with and the terminology used, and in Section 2.4.2 we describe some of the theoretical results they obtained and which we use in our work.

2.4.1 Conditions for Curves

Neagu and Lacolle restrict their study to a family of piecewise convex curves that fulfill certain requirements. The requirements from a curve C are:

C1. The polygonal line $P = P_0P_1...P_m$, named the *control polygon* of the curve C, is simple and convex.

C2. P_0 and P_m are the extremities of C.

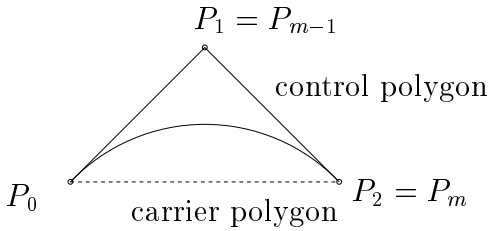


Figure 2.1: Control and carrier polygons that bound C and satisfy the conditions C1–6.

C3. The line P_0P_1 is tangent to C in P_0 and the line $P_{m-1}P_m$ is tangent to C in P_m.

C4. The curve is included in the convex hull of its control polygon.

C5. The curve and its control polygon satisfy the variation diminishing property (i.e., any line that intersects C at k points intersects the control polygon in at least k points, see [17]).

C6. There exists a subdivision algorithm through which the control polygon converges to the curve.

Since the control polygons are simple and convex, the variation diminishing property implies that these curves are convex.

As Neagu and Lacolle point out, these conditions can be satisfied by many well-known parametric curves: Bézier and rational Bézier curves, B-splines and NURBS. Furthermore, theoretically, these conditions are satisfied by *any* convex well-behaved curve: we can sample points on the curve and compute tangents to the curve at these points. The control polygon will consist of these points and the intersection points of adjacent tangents (see Figure 2.2). It is easy to see that the first three conditions are met. The fourth and fifth conditions are met because of convexity. To subdivide the control polygon, we sample a point p on the curve between two adjacent points p_i and p_{i+1} and "cut the corner" by adding the intersection points of the tangent at p with the tangents at p_i and p_{i+1} (see Figure 2.2). Note that, unlike the case of Bézier curves (and other similar curves), for this subdivision process, knowledge of the control polygon is generally insufficient and a knowledge of the original curve is also needed to sample a point on it. Our application is

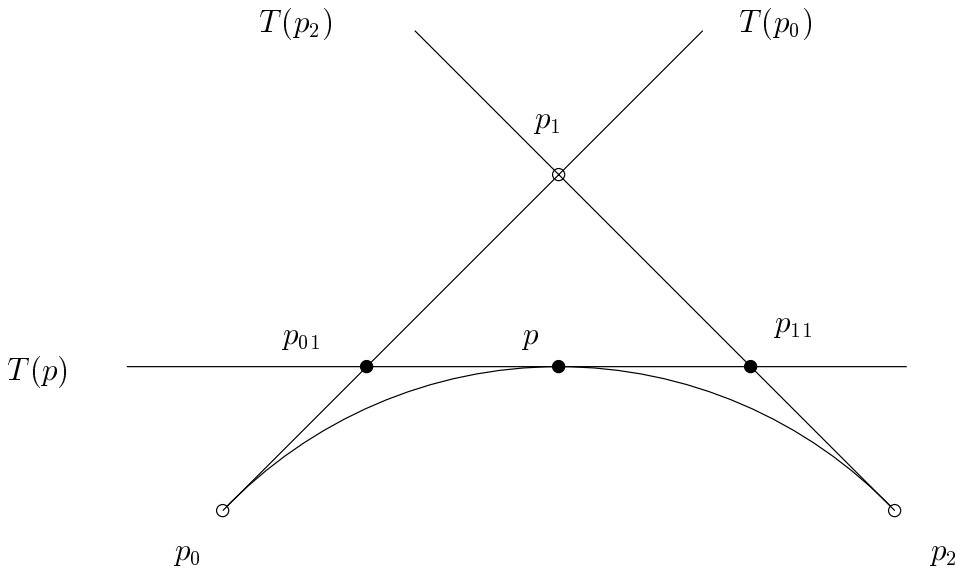


Figure 2.2: Subdividing a convex curve by adding a point and its tangent: the point p on the curve is added between p_1 and p_2, the intersection points of its tangent $T(p)$ with the tangents $T(p_0)$ and $T(p_2)$ are added to the new control polygon instead of p_1.

designed so that it can be extended to use such subdivision schemes as well (see Chapter 4).

The above subdivision process assumes we can sample points on the curve and find the tangent at this point. For algebraic *parametric* curves this is trivial (we sample the parameter of the equations and of the derivatives). However, this can also be applied to algebraic curves in implicit form, provided that we can sample points on the curve and compute tangents at these points. It should be noted that applying this subdivision scheme to algebraic curves in implicit form requires symbolic computation of the points on the curve (and of the tangent vector). It is therefore not certain that this scheme is worthwhile for non-parametric curves. For example, for circles, finding a point on the curve amounts to evaluating an expression with a square root and so does finding the intersection point of two circles.

We follow the terminology introduced in [41]. The *carrier polygon* of a curve is the segment between the two endpoints of the control polygon (see Figure 2.1). Therefore, these two polygonal lines, the control polygon

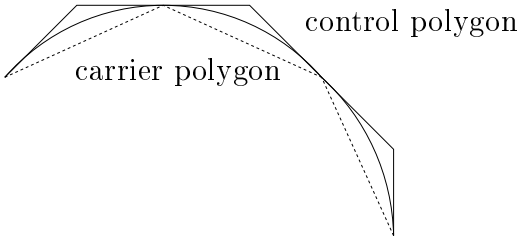


Figure 2.3: A union of control and carrier polygons.

and the carrier polygon bound the curve between them. After a subdivision has taken place, the union of the subcurves is the original curve. In some cases when we talk of the carrier (resp. control) polygon of a curve we will mean the union of the carrier (resp. control) polygons of its subcurves (see Figure 2.3). It is easy to see that these also bound the original curve.

2.4.2 Isolation of Intersection Points

Given two curves satisfying the above conditions, the paper [41] gives sufficient conditions on the control and carrier polygons of two subcurves which guarantee that the subcurves intersect exactly once. We use these conditions in the algorithm described in Section 4.1.2.

The two conditions are:

I1. The control and carrier polygon of the first subcurve C_1, each intersect the control and carrier polygon of the second subcurve C_2, exactly once (four intersections altogether). Informally, this condition guarantees that the two subcurves actually intersect. However, as shown by a counter example, subcurves that meet this condition can still intersect in more than one point.

I2. Let s_1 and s_2 be the segments of the control polygons of C_1 and C_2 respectively, which intersect each other (recall that by the first condition there is only one such intersection). Then the endpoints of C_2 are on opposite sides of the line supporting s_1, and similarly the endpoints of C_1 are on opposite sides of the line supporting s_2.

In our application we verify these conditions in order to isolate the *intersection polygons* — the areas in which the vertices of the curve arrangements

lie (see Section 4.1.2).

Chapter 3

Arrangements in CGAL

In this chapter we present CGAL's two-dimensional arrangement package — a generic and robust software package for arrangements of general curves. The arrangement layer is built on top of the planar map layer. Given a set $\mathcal{C}$ of (not necessarily x-monotone and possibly intersecting) curves, we wish to construct the planar map induced by $\mathcal{C}$. However, we wish to do so without loss of information. For example, we want to be able to trace all edges in the planar map originating from the same curve. Furthermore, we want the users to be able to control the process of decomposing the curves into subcurves, in particular splitting the curves into x-monotone curves to be inserted into the planar map. This is achieved with a special data structure that we call a *curve hierarchy tree* which is described in Section 3.1.

For some algorithms, it is not needed to build the whole planar map induced by the arrangement. For example, the lower envelope of an arrangement of x-monotone curves which intersect each other at most a constant number s times, can be found in near linear time even if the complexity of the induced planar map is quadratic [27, 47]. Therefore, building the planar map induced by the arrangement is not always desired. We enable the users to disable (or postpone) the building of the planar map.

The package does *not* assume general position. In particular it supports x-degenerate input (e.g., vertical line segments), non x-monotone curves and overlapping curves. x-degenerate input is treated with a symbolic shear transform. A point p with the same x-coordinate as another point q but with larger y-coordinate is considered "to the right" of q, see [13, Section 6.3] and [18]. Non x-monotone curves are partitioned into x-monotone subcurves and then inserted into the planar map. If two curves overlap, this fact is

traced by the traits intersection function, before the insertion into the planar map. Thus, given a halfedge in the planar map, the user can traverse all the overlapping subcurves that correspond to the same pair of halfedges.

3.1 Hierarchy Tree

When constructing an arrangement we decompose each curve C in two steps obtaining the collections C' and C''. We can regard these collections as levels in a hierarchy of curves where the union of the subcurves in each level is the original curve C. We store these collections in a special structure — a *hierarchy tree*. This structure usually consists of three levels, although in some cases it consists of less (e.g., when inserting an x-monotone curve) or more (when the users define their own *split functions*, see for example Section 3.7). The levels are:

- Curve node level: the root of the tree — holds the original curve.
- Subcurve node level: inner nodes of the tree — holds the subcurves of the original curve. In the default mode these are the x-monotone subcurves comprising the original curve.
- Edge node level: leaves of the tree — hold the subcurves corresponding to the edges of the planar map induced by the arrangement. These nodes will be built only in *update mode* (by default the arrangement is in update mode).

Figure 3.1 shows an example of a simple arrangement and its corresponding curve hierarchy.

The hierarchy tree enables us to intersect the curves without loss of information. The original curve and the intermediate subcurves are stored in the tree and the user can traverse them. Furthermore, the users can define their own hierarchy by passing their own intersection sequence. We make use of a this feature in the application described in Chapter 4.

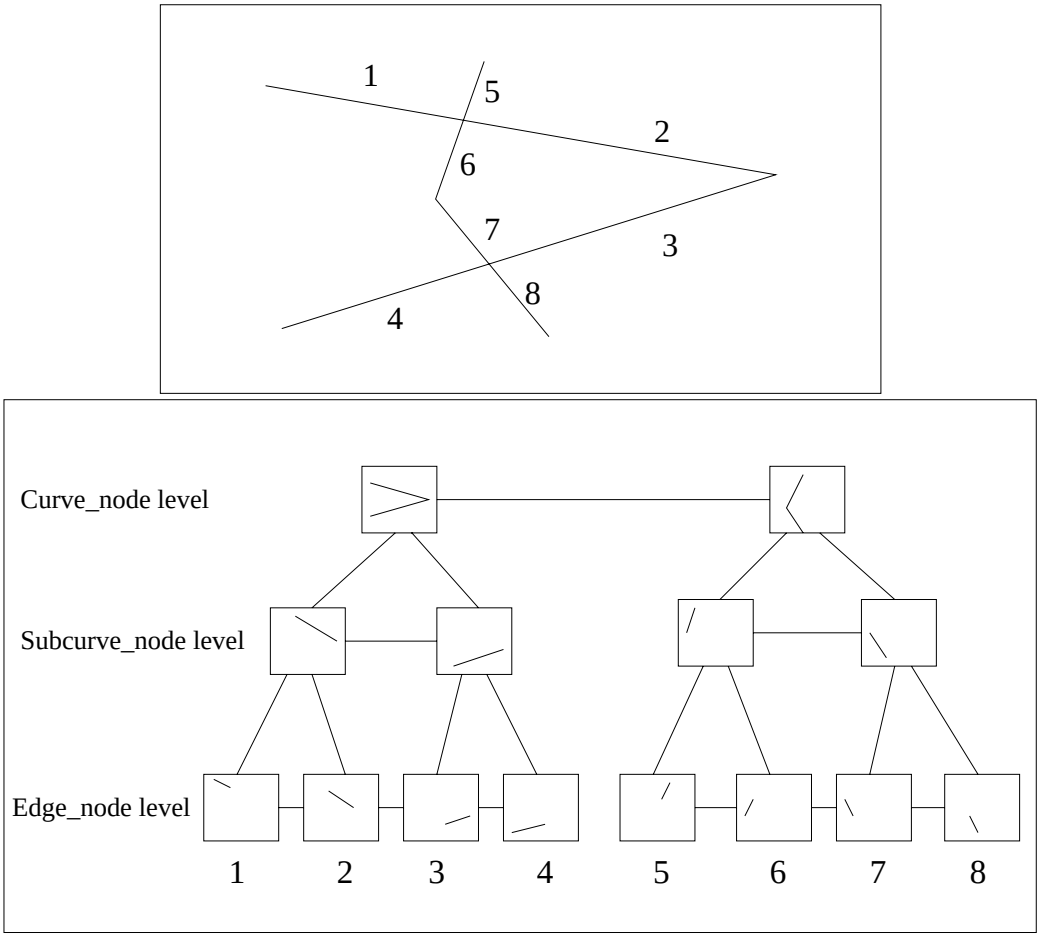


Figure 3.1: A simple arrangement of two polylines, and its corresponding hierarchy tree (the edges are numbered according to their order in the tree).

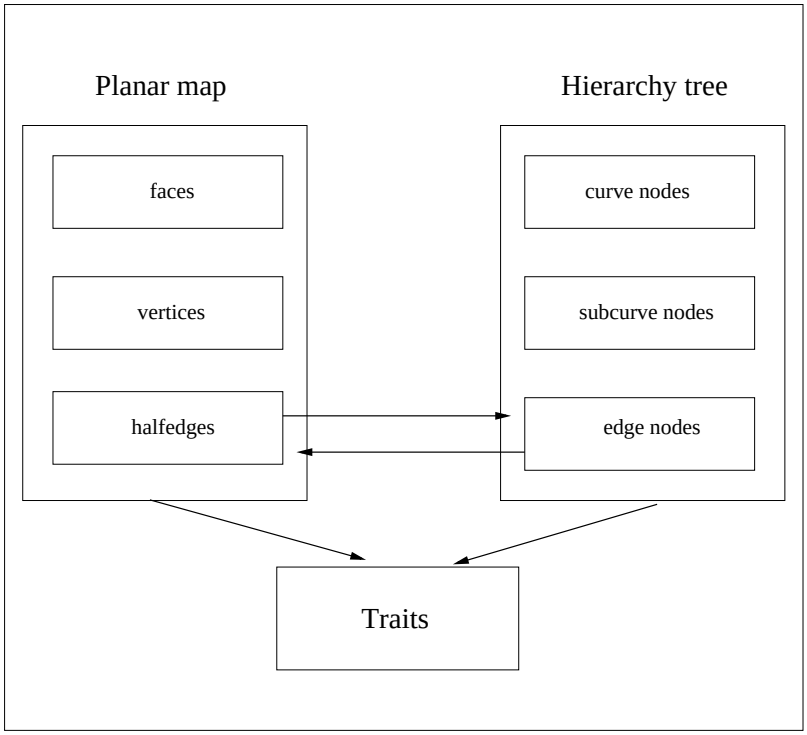


Figure 3.2: The relations between the classes in the arrangement.

3.2 Design and Implementation Details

3.2.1 General Design

Since the arrangement layer is built on top of the planar map layer, it has the features of CGAL's planar maps. These include traversal over vertices, halfedges and faces of the planar map, as well as circulating over halfedges incident to a face or to a vertex. The flexible *point location* mechanism introduced in the planar map package [18] is also maintained in the arrangement layer. In addition to the planar map, the arrangement holds a hierarchy tree and a traits class that is used by the other classes for their geometric computations. Figure 3.2 describes the relations between the different classes inside `Arrangement_2` — the arrangement class.

The `Arrangement_2` class is passed three template parameters: `Dcel`, `Traits` and `Base_node`. The first is needed for the planar map within the

arrangement, and has some additional requirements over the regular requirements from a planar map Dcel. As in the planar map, the Dcel enables addition of attributes (e.g., color of a face) by the users. The `Base_node` template parameter serves as a base class for the nodes of the hierarchy tree. It also has a minimal set of requirements that the users can extend to add attributes. Section 3.2.2 describes the use of the `Base_node` class in the hierarchy tree implementation. As in all CGAL algorithms and data structures the `Traits` parameter is used by the class as an interface to geometric types and computations; Section 3.3 describes the arrangement traits requirements and concrete implementations for them.

3.2.2 Hierarchy Tree Design

The hierarchy structure was required to store all the relevant geometry and enable efficient traversal over each level of the tree, and between the levels. The structure was also required to be flexible, enabling the advanced users to define their own hierarchy of curves, and to add attributes to the nodes of the tree. Implementing a design that meets these requirements (while maintaining an "STL-like" interface) was a difficult task. In the following paragraphs we describe our design.

As mentioned earlier, the hierarchy consists of three types of levels: the `curve_node` level, the `subcurve_node` level and the `edge_node` level. The curve node is the root of the curve hierarchy and access to any level of the tree should be enabled from it. The subcurve nodes are the inner nodes of the tree, holding references both to their parent nodes and to their children. The edge nodes are the leaves of the tree, similar to the subcurve nodes. However, they also store a reference to their corresponding halfedge (i.e., the halfedge that has the same source and target points as the edge) in the planar map. Furthermore, they store references to other edges which overlap them.

Since the users can define their own curve hierarchy, the implementation could not assume a constant number of levels in the tree. A variable depth tree structure was required, with its root and leaf nodes corresponding to the curve and edge nodes described above. The class hierarchy that implements this tree is depicted in Figure 3.3 (the figure follows the conventions of [22], where a black arrow represents aggregation and a white arrow represents derivation). It is a variation of the *composite* pattern described in [22]. In this pattern all nodes of the tree derive from the same base class and refer to their parent and children nodes with references to this base class.

In our implementation we wanted to give the users the ability to add their own attributes to the hierarchy nodes. Therefore, the base node is passed as a template parameter to the arrangement and the nodes of the tree derive from it (see Figure 3.3). Thus, if the users add an additional attribute to the base node, it is inherited by the tree nodes. The `subcurve_node` is the basic node type of the hierarchy, it stores references to its parent, children and neighbors and methods to access them. The `curve_node` and `edge_node` derive from the subcurve node type with additional methods. The edge node adds a reference to its corresponding halfedge in the planar map. The curve node, the root of the tree, adds references to all the levels of the tree enabling traversal over each level.

The ability to efficiently traverse each level of the tree was a requirement from the hierarchy structure. We used CGAL's `In_place_list` [12, Part 3] to achieve this goal. This class implements a doubly connected list data type with an STL interface, that manages its items in-place. The advantage of this type over STL's `list` type is that it is able to handle the stored elements by reference instead of copying them. This enables to erase an element from the sequence even when we only know its address and no iterator to it (unlike the standard list). This simplifies mutually pointered data structures like the DCEL for planar maps (indeed we use this structure there). This class is ideal for our purpose since it gives us an STL iterator interface for each level, while enabling traversal through the references from the other levels in the hierarchy. The usage of the `In_place_list` structure requires the item type to provide the two necessary pointers `next_link` and `prev_link`. One possibility to obtain these pointers is to inherit them from the base class `In_place_list_base` as we have done (see Figure 3.3).

3.2.3 Implementation of Algorithms

There are several algorithms for constructing an arrangement. Each algorithm may be appropriate for different inputs and different scenarios. For example, a sweep algorithm [13, Chapter 2] may be the algorithm of choice for a static arrangement of curves (i.e., when all curves are given in advance). A randomized incremental algorithm that uses a trapezoidal decomposition [37] may be suitable for a dynamic or semi-dynamic construction. Both of these algorithms are output sensitive. The sweep algorithm runs in $O((n+h)\log n)$ time, where n is the complexity of the input, i.e, the number of curves, and h is the number of intersection points in the arrangement. The random-

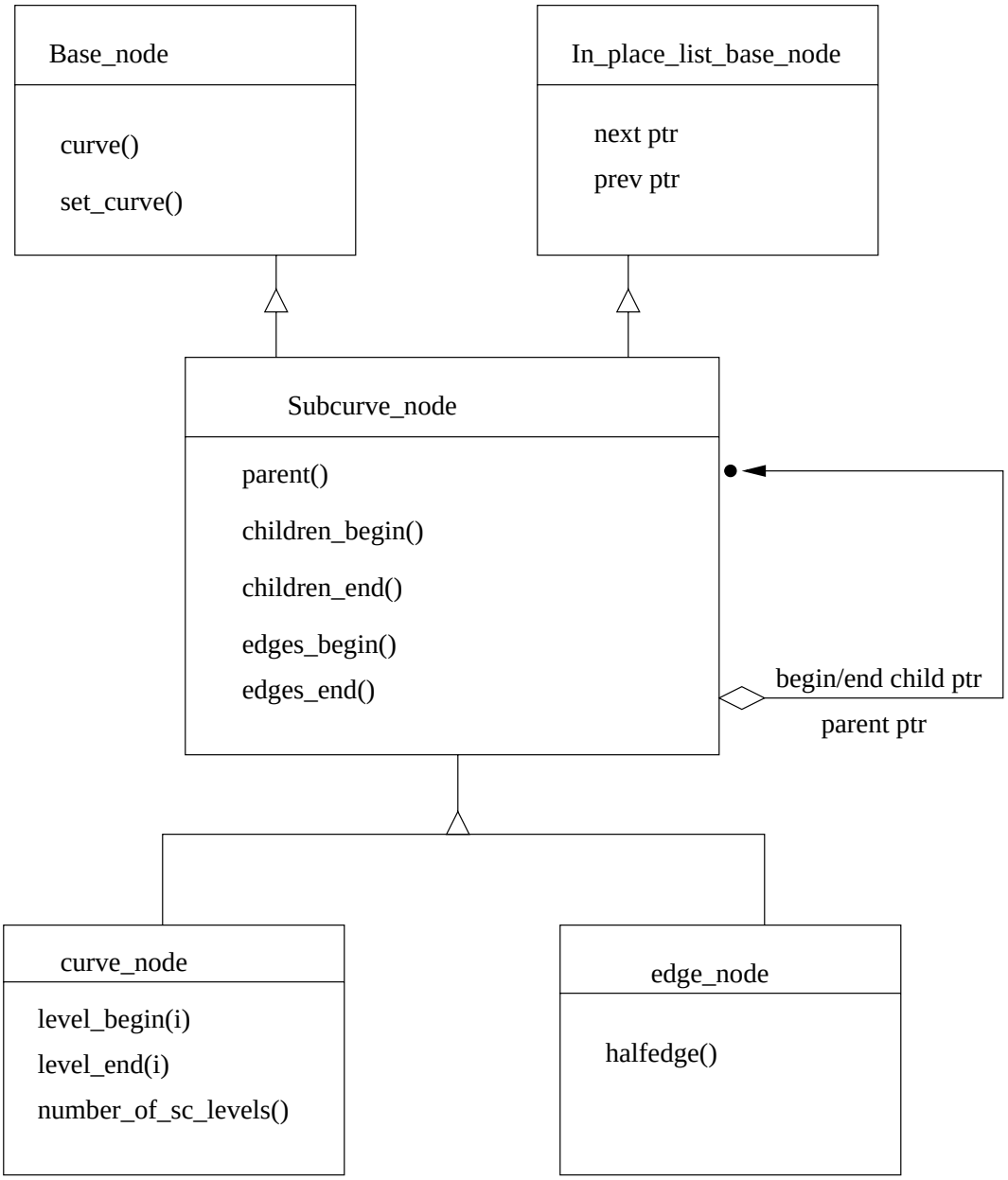


Figure 3.3: The design diagram of the hierarchy tree implementation.

ized incremental algorithm runs in $O(n \log n + h)$ expected time. For dense arrangements with complexity near $\Theta(n^2)$ a simpler incremental algorithm (whose efficiency stems from the *zone theorem* [13, 19]) may be the best solution in practice. It is simple to implement and does not require additional overhead for managing complex data structures. For arrangements of lines this algorithm runs in $O(n^2)$ time.

Since our arrangement is fully dynamic, we need to implement an incremental insertion algorithm. Incremental insertion of curves into the arrangement depends on point location, in order to find the face of the source point of the curve. In addition it depends on efficiently acquiring the intersection points of the curve with the face's boundary. In our implementation the point location algorithm is defined by the point location strategy of the user [18]. Furthermore, the availability of a trapezoidal decomposition also depends on the point-location strategy used. We therefore implemented the simpler incremental algorithm: Given a curve, we locate its start point using the strategy and traverse the zone of the curve, computing the intersection points of the curve and the face, splitting the edges of the map at these points and inserting the new edges as we progress. Acquiring the intersection points of the curve with the face can also be done using the internal trapezoidal map maintained by the fast point location strategy.[1] Thus the theoretical complexity of the insertion algorithm depends on the point location strategy used.

There are several algorithms related to arrangements of curves. Some of the algorithms query the complete arrangement, while others do not need the full construction of the arrangement. Examples of the latter are computing the lower (upper) envelope of an arrangement and constructing the face where a point is located. In these algorithms the curves are cut into x-monotone subcurves before the algorithm is performed. Currently we have not implemented these algorithms, however our implementation enables insertion of curves into the hierarchy without construction of the full planar map. This will enable us to incorporate these algorithms into our package in the future. In order to disable (or postpone) the insertion of the curves into the planar map, the users can call the `set_update` method with a `false` parameter. Calling the method with `true` enables update mode again and the curves are inserted into the planar map.

[1] In order to enable this, the interface of the point-location strategy should be slightly modified. Currently this is not yet implemented; we plan to do it in the future.

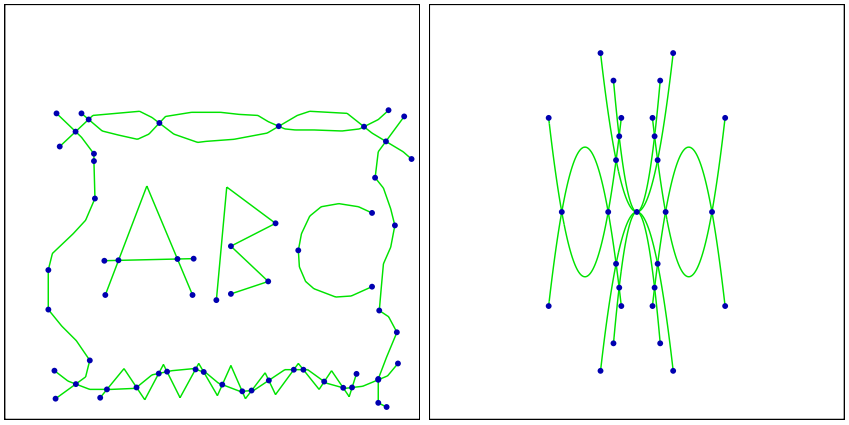


Figure 3.4: An arrangement of polylines and an arrangement of canonical-parabola arcs.

3.3 Geometric Traits

There are several functions and predicates required by the arrangement class that are not required in planar maps. In our implementation this means that the requirements from the arrangement traits class are a superset of those of the planar map traits described in Section 2.3.1 (in the terminology of [5], they are a *refinement* of the planar map traits and can be used by the planar map class as well).

The main functions that are added are for handling intersections of x-monotone curves and for detecting and handling non x-monotone curves and splitting them into x-monotone subcurves. For the latter a `Curve` type is required since in the planar map only x-monotone curves are handled. A predicate that detects whether two curves overlap is also required for dealing with degenerate cases (see Section 3.4). The full set of requirements is given in the arrangement documentation.

We have implemented several traits classes for different curves. As for the planar map, traits for line segments were implemented using both the CGAL kernel and the LEDA rational kernel. Another traits class is the circle arc traits class — `Arr_circles_real_traits`, which uses square root operations; therefore in order to guarantee robustness, a number type package that supports square roots should be used for it (e.g., `leda_reals` [10]). All of the above traits classes are available and documented in the CGAL-2.1

distribution. We have also implemented two traits classes for polygonal lines (represented as an STL container of CGAL and LEDA points respectively); we make use of these in the adaptive point location application (see Chapter 4). Another traits class that we implemented is for canonical parabolic arcs (namely, the graph of the function $y = ax^2 + bx + c$ over some closed interval), these also need square root operations. We are currently working on a traits class which deals with both circles and line segments. Since we already have a separate traits class for both curve types, we implement the traits by dispatching the functions to the relevant class, and reimplement only functions that have interactions between circle arcs and segments. Figure 1.1 in Chapter 1 shows examples of arrangements constructed with the segment and circle traits. Figure 3.4 shows examples of arrangements constructed with the polyline and canonical-parabola traits. The programs for generating them can be found at `http://www.math.tau.ac.il/~hanniel/ARRG00/`.

3.4 Degeneracies

Our arrangement package does not make any general position assumptions on the input. In particular the input curves can be x-degenerate, non x-monotone and the curves can be tangent to each other or even overlap. We next describe how we deal with these cases.

As noted above, the planar map package can take care of x-degenerate input sets (e.g., vertical segments). This is done with a symbolic shear transform (see [13, Section 6.3] and [18]). Since the arrangement package is built on top of it, this is already taken care of.

Non x-monotone curves (e.g., circles) can be used in our arrangement package. They are decomposed into x-monotone subcurves before they are inserted into the planar map. This requires from the users to supply a function in the traits class (see Section 3.3) that performs this decomposition. The original non x-monotone curve is stored in the hierarchy tree and can therefore be recovered if needed.

In order to deal with tangent curves the user must supply an appropriate implementation in the traits class. For example, in our implementation of `Arr_circles_real_traits` when two circular arcs are tangent at a point p, p is considered an intersection point (see for example, Figure 1.1 in Chapter 1). However, the users might know their input to be without such a degeneracy. For example, they might have performed a controled perturbation on the

input before inserting the curves into the arrangement; cf. [29, 44]. In that case they can implement a faster traits class based on that knowledge, which does not take degeneracies into consideration.

Overlapping curves raise a difficulty to our implementation. The planar map package that is at the basis of the arrangement implementation cannot deal with overlapping curves. Therefore, overlaps are facilitated in the hierarchy tree. This is done in the following way: for every halfedge there is a reference to its corresponding edge node in the hierarchy tree. We define a special *overlap circulator* type[2] which enables the user to traverse the list of overlapping edge nodes that correspond to the same halfedge. In order to implement this list we made use of the two pointers to the *begin_child* and *past_end_child*, that appear in the *subcurve_node* class and are not needed in the edges (since the edges are leaves of the tree). The interface of the "nearest intersection" function in the traits class was defined to return *two* intersection points which correspond to the source and target of the overlapping subcurve. If the intersection is at a single point then the returned points are identical.

3.5 Robustness

Robustness issues arise frequently in an arrangement implementation. One of the main reasons is that intersections of curves generally cannot be performed exactly using floating point arithmetic. In an arrangement, intersections are basic operations and the resulting intersection points are used in other computations. Therefore, if they are not computed exactly the intersection points will invalidate the arrangement. For example, the result of an intersection of two line segments, computed with floating point arithmetic, can be a point that is on neither of the segments. Other reasons for robustness problems can arise from degenerate situations. For example, an endpoint of one curve lying on another curve. If the predicate is evaluated with floating point arithmetic, the result might be that the curves do not touch or that they fully intersect.

The traits mechanism provides a convenient way to deal with most robustness problems. Given a traits class that implements robust predicates and constructions, our package is robust. This encapsulates robustness problems

[2] A circulator is a concept for iterating over circular sequences, see [7, 12, 32].

to implementing robust traits classes. Our general approach when implementing a traits class is therefore to use exact arithmetic packages to ensure robust predicates.

However, when used for arrangements, this approach can incur severe difficulties. Consider for example, a standard implementation of a rational number type (e.g., `leda_rational` or CGAL's `Quotient`) which is used for a segment arrangement. A number is represented as a pair of multi-precision integers (e.g., `leda_integer`) for the numerator and denominator. In every arithmetic operation (addition, subtraction, multiplication and division) between two numbers the numerators and denominators are multiplied. If no normalization is performed then the result's bitlength is doubled. When using the result as input for a new computation, the bitlength is doubled yet again. This "explosion" of the bitlength results in a major slowdown of the computations since the computation time for exact arithmetic is dependent on the bitlength. Our experiments show that even for a ten by ten grid of segments this slowdown is unbearable. One solution can be to perform normalization on the results of the intersection computation. This solution, however, should be tailored for a specific rational number type. Indeed, we use such a scheme in our `Arr_leda_segment_exact_traits`[3] class. However, such a scheme is not general and cannot be used for other exact number types, e.g., `leda_reals`, which use a different representation for exact computation. Since CGAL's kernel is parameterized with an exact number type and we have no apriori knowledge of its representation, we cannot use such a scheme with a traits class that uses CGAL's kernel.

The way we deal with this problem is to use the information that is stored in the arrangement hierarchy tree to bound the depth of computation. Instead of computing the intersection between two subcurves we perform the intersection on the original curves on which the subcurves lie. This avoids the explosion described above since all the intersection points are now obtained by operations on the original input.

3.6 Example Code

The following example code demonstrates the use of arrangements in CGAL (this and other example programs can be found in [12] and in the CGAL-2.1

[3]The reader should not be confused by the name of the class, this is a CGAL traits class that uses LEDA's rational kernel and is not part of the LEDA library.

distribution). It constructs an arrangement of two circles with a radius of 5 units (see Figure 3.5) and performs a vertical ray-shooting query. In order to achieve robust intersection computations it uses the `leda_real` number type (although for this simple example the built-in double number type will also run correctly).

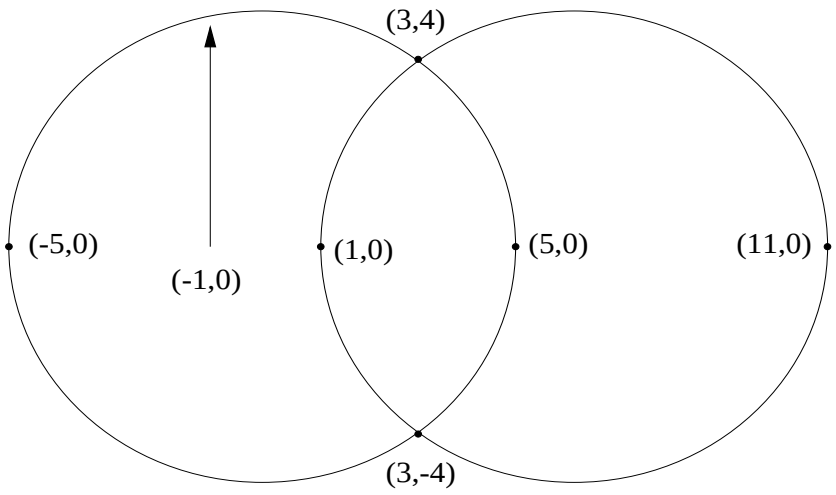


Figure 3.5: The arrangement generated by the example program.

```
1   #include <CGAL/basic.h>
2   #include <CGAL/Arr_2_bases.h>
3   #include <CGAL/Arr_2_default_dcel.h>
4   #include <CGAL/Arr_circles_real_traits.h>
5   #include <CGAL/Arrangement_2.h>
6   #include <CGAL/leda_real.h>
7
8   typedef leda_real                                   NT;
9   typedef CGAL::Arr_circles_real_traits<NT>           Traits;
10
11  typedef Traits::Point                               Point;
12  typedef Traits::X_curve                             X_curve;
13  typedef Traits::Curve                               Curve;
14
15  typedef CGAL::Arr_base_node<Curve>                   Base_node;
```

```
16  typedef CGAL::Arr_2_default_dcel<Traits>                Dcel;
17  typedef CGAL::Arrangement_2<Dcel,Traits,Base_node >     Arr_2;
18
19  using namespace std;
20
21  int main() {
22     Arr_2 arr;
23
24     //2 ccw circles with squared radius 25 and center (0,0) and (6,0)
25     Arr_2::Curve_iterator cit=arr.insert(Curve(0,0,25));
26     cit=arr.insert(Curve(6,0,25));
27
28     //upward vertical ray shooting
29     Arr_2::Locate_type lt;
30     Arr_2::Halfedge_handle e;
31     e=arr.vertical_ray_shoot(Point(-1,0),lt,true);
32
33     CGAL_assertion(e->source()->point()==Point(3,4));
34     CGAL_assertion(e->target()->point()==Point(-5,0));
35
36     return 0;
37  }
```

Lines $8-17$ are custom `typedef`s to get shorter names. Since CGAL is highly templated, avoiding these `typedef`s will result in very long and unreadable type names. Note that we use the `leda_real` number type for our predicates (for this simple example the built-in `double` type would have been sufficient, but for other circles it would cause robustness problems). In line $25-26$ we insert two counter clockwise circles with squared radius 25 units, centered at points $(0,0)$ and $(6,0)$ respectively. These induce the arrangement depicted in Figure 3.5. In lines $29-31$ we perform an upward vertical ray shooting query from the point $(-1,0)$. The return value is the halfedge of the arrangement directly above the point. The `lt` variable will store the type of object that the ray intersected, in our case an `EDGE`. If the query point were $(3,0)$, `lt` would have been `VERTEX`. In lines $33-34$ we assert that the returned halfedge is correct.

3.7 More Technical Details

As mentioned in Section 3.2.2, the requirement to enable the users to define their own hierarchy guided us in our design. The hierarchy is defined by a sequence of *splitting functions* that split a curve into subcurves. For example, for an arrangement of polyarcs, such a sequence can start with a function that splits the polyarc into single arcs, and then a function that splits each arc into x-monotone arcs. The interface for such a user-defined hierarchy is therefore an insertion function which is passed a sequence of split functions. The common mechanism for implementing this would be to pass the insertion function a sequence of *function pointers*. Using templates and following the STL, we can make the insertion function generic by passing the sequence as a pair of iterators — the `begin` and `past-end` iterators, referring to the function pointers. Thus we can pass the function pointers in any container that has iterators conforming to the STL `iterator` concept (e.g., STL containers, or C arrays). A call to this insertion function can look like:

```
void split_1(Curve, list<Curve>&);
void split_2(Curve, list<Curve>&);

vector<SPLIT_FUNC*>  sf;
sf.push_back(&split_1);
sf.push_back(&split_2);

Arrangement_2 arr;
Curve cv(...); //some curve to be inserted

arr.insert(cv,sf.begin(),sf.end());
```

The C++ template mechanism combined with inheritance allows for even more flexibility. Rather than passing a sequence of pointers to functions we can pass a sequence of pointers to *function objects*. Function objects [5] are classes that act like function pointers. This is achieved by overloading the C++ `operator()`. These classes are used extensively in the STL, for example, each STL algorithm that requires a partial order to be defined on its types, has two versions — one that assumes the `operator<` is defined for the types, and one that is passed a function object that defines the partial order (see [5] for more information). One of the main advantages of function objects over function pointers is that they can store a state (i.e., they can

store variables). This enables to pass the function additional parameters, enabling for greater flexibility. In our case we want a sequence of function objects each corresponding to a different function, with all the functions having the same interface. This can be achieved using inheritance — we define a base class for the function objects with the `operator()` defined as `virtual`. The derived function objects override the virtual operator to implement the split functions. This usage of function objects with inheritance is sometimes referred to as *action classes* [52].

The following fragment of code shows how to insert a curve into an arrangement using function objects with parameters. We make use of similar code in the adaptive point location application we describe in Chapter 4.

```
struct Split_base {
  Split_base();
  virtual void operator()(Curve,list<Curve>&);
}

struct Split_with_param : public Split_base {
  Split_with_param(double p) : param(p) {}
  virtual void operator()(Curve,list<Curve>&); //uses param
private:
  double param;
}

Split_base split_1;
Split_with_param split_2(0.5); //use 0.5 as parameter

vector<Split_base*>  sf;
sf.push_back(&split_1);
sf.push_back(&split_2);

Arrangement_2 arr;
Curve cv(...); //some curve to be inserted

arr.insert(cv,sf.begin(),sf.end());
```

Chapter 4

Approximating Curves by Polygonal Lines

The application described below addresses the following problem: Given a set $\mathcal{C}$ of curves satisfying the conditions C1–6 presented in Section 2.4.1, construct a data structure for efficient point location and vertical ray shooting queries. Given a query point q the vertical ray shooting algorithm will return the curve $C \in \mathcal{C}$ that is above q and has the minimal vertical distance to q. The point location algorithm will return a circular list of curves that are on the outer boundary of the face of the arrangement where q is located and circular lists of curves for the boundaries of the holes (if they exist) of the face. We present algorithms that perform these queries using an adaptive scheme. The algorithms we describe are static (i.e., all the curves are given in the initialization step).

The general scheme of our algorithms is to bound each curve by a *bounding polygon*, and perform all operations on these polygons rather than on the curves themselves. If the bounding polygon's approximation of the curve is not good enough, it is refined by a subdivision process. This scheme is similar to the one presented by Neagu and Lacolle [41]. There, the authors use such a scheme to compute an arrangement of polygonal lines that is topologically equivalent to the arrangement of curves. We make use of some of their results in our algorithms. Like them, we have also implemented our algorithm on Bézier curves which satisfy the given conditions and have an easy-to-implement subdivision scheme.

However, there are several significant differences between our work and the work presented in [41]. In their work they find sufficient conditions for

equivalence of arrangements of piecewise Bézier curves (or any other curves that satisfy conditions C1–6) and arrangements of polygonal lines. They further prove that such an arrangement exists and implement an algorithm which checks for these conditions, and outputs the set of polygonal lines that induces an equivalent arrangement. The novelty of their approach is that the operations are not done on the curves themselves, rather they are done on the polygons that bound the curves. We construct an efficient data structure for point location in the arrangement. Furthermore, our algorithm is local and adaptive whereas theirs is global. That is, we only compute the faces of the arrangements that are needed by a query. Another difference is that we do not make an assumption that the arrangements are connected like they do, i.e., we support holes inside the faces of the arrangement; this in turn raises considerable algorithmic difficulties. We also implement a heuristic that traces degenerate cases and deals with them, therefore the input is not assumed to be non-degenerate (see the next paragraph and Section 4.1.3). There is also a difference from a system point of view. Our application constructs a framework for future work and can easily be extended to other curves (and other representations) by changing the traits class. It is based on CGAL's arrangement package (see Chapter 3) and can therefore benefit from improvements in that package.

While we deal with degenerate cases (e.g., two curves that are tangent at a common point) by a heuristic approach, we do not give a complete treatment of these degeneracies. In some of these cases if the users require the exact answer they need to resort to exact computations on the original curves. In our implementation we employ a user-defined *threshold function* (for our program we used an ε that should be smaller than the bounding polygon's diameter; ε is a parameter that can be changed by the user). When the threshold is passed, the program terminates the subdivision process and returns to the user the region where the method fails. The user can then apply symbolic methods on that region. Section 4.1.3 describes possible degeneracies and ways of dealing with them.

We can also deal with curves that are a union of completely convex curves (each satisfying the conditions $C1-6$ from Section 2.4.1). This can be done by partitioning the original curve into completely convex subcurves (see Section 3.1). In our implementation we have chosen to deal with quadratic Bézier curves. These are easily generated randomly and are convex by definition (their control polygon is a triangle). The subdivision algorithm for these curves is the de Casteljau algorithm (with parameter 1/2, see [17])

which makes the computations easier. Extensions to other curves can be done in our application by changing the traits class (see Section 4.2.2).

4.1 The Algorithms

Several algorithms have been implemented for point location and vertical ray shooting queries in the CGAL arrangement package. This was done via the point location strategy mechanism (see Section 2.3.2). These algorithms work on any curve that conforms to the traits class requirements (see Section 2.3.1). In particular, they work on arrangements of line segments and polygonal lines. We can thus use these implementations to perform queries on the arrangement of the bounding polygons of the curve in $\mathcal{C}$. We use the results to answer the queries on the original curves. The following sections describe the algorithms we use for vertical ray shooting and point location. In a standard planar map, represented say by a DCEL, an answer to a vertical ray shooting query is easily transformed into an answer to a point location query. In our adaptive setting however, point location queries are much more involved than vertical ray shooting queries as we explain below.

4.1.1 Vertical Ray Shooting

Given a query point q in the plane, we wish to find the curve $C_i \in \mathcal{C}$ that is directly above q (i.e., is above q and has the minimum vertical distance to it).

Sufficient Conditions

Given a query point q and a curve $C_i \in \mathcal{C}$ directly above q, we need to find sufficient conditions on the bounding polygons of the curve (i.e., the polygon that consists of the control and carrier polygon) which guarantee that C_i is the result of the query from q.

The first condition is that the bounding polygon directly above q (i.e., the result of the vertical ray shooting query on the arrangement of bounding polygons) does not intersect any other polygon. A local scheme that goes over the border of the bounding polygon and checks for intersections with other segments is not sufficient. Polygons that fully contain the bounding polygon or are fully contained in it will not be detected by such a scheme. This can

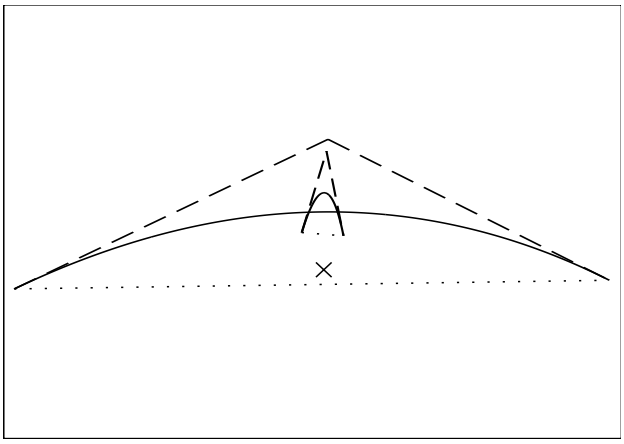


Figure 4.1: A global scheme for intersection detection is needed: the intersection of the large triangle with the small one will not be detected by a local intersection detection scheme, and the vertical ray shooting query will return the small curve instead of the large one.

cause the algorithm to return a wrong result (see Figure 4.1). We therefore need a global scheme to detect intersections between bounding polygons. This is achieved using an *intersection graph* $IG = (V, E)$: every node $v \in V$ of the graph represents a bounding polygon of a subcurve in the arrangement and two nodes are connected by an edge $e \in E$ if the bounding polygons they represent intersect. The intersection graph is initialized at the construction of the arrangement. When a subdivision is performed only the nodes that are adjacent to the node that corresponds to the subdivided polygon need to be updated.

The second condition is that both the control and carrier polygons of the bounding polygon are above the query point. If this condition is not met then we cannot guarantee that the original curve is above the query point (see Figure 4.2). This condition also assures that q is outside of the bounding polygon. If q were inside we would not be able to decide whether the original curve is above q or below it.

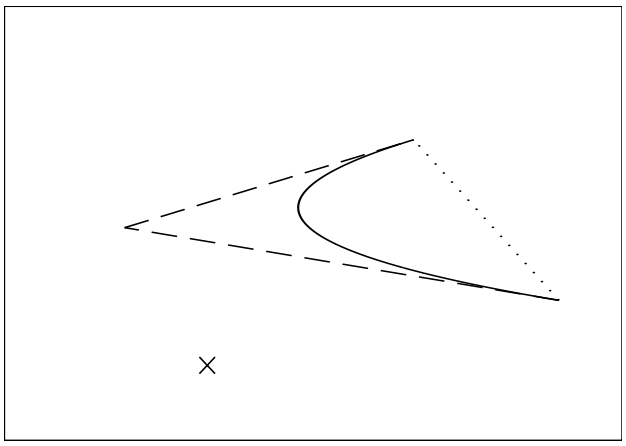


Figure 4.2: The second condition is necessary although the first condition is met: the carrier polygon is not above the query point and neither is the original curve.

The Algorithm

Our algorithm performs a vertical ray shooting query in the bounding polygon arrangement. It then checks for the sufficient conditions on the result bounding polygon *bp* and if they are not met performs a subdivision on *bp* and on the bounding polygons that intersect it (i.e., the adjacent nodes in the intersection graph). In every subdivision of a polygon the intersection graph is updated. The vertical ray shooting algorithm looks like this:

```
do {
  find the bounding polygon bp above q;
  if conditions 1 and 2 are met break;
  subdivide bp;
  find the bounding polygon bp above q;
  if conditions 1 and 2 are met break;
  subdivide the neighbors of bp in the intersection graph;
}
return bp;
```

In the algorithm we perform two steps inside the iteration, first subdividing `bp`, and if the conditions are still not met, we subdivide the bounding polygons that intersect it. The bounding polygons that intersect `bp` are easily

found since they correspond to the adjacent vertices of `bp` in the intersection graph IG. We do not perform all the subdivisions in one step because our experience shows that in many cases it is sufficient to subdivide only `bp` in order to get a valid result. After a subdivision is performed we need to repeat the search for the bounding polygon by performing a vertical ray shooting query in the polygon arrangement, since the subdivision has changed the arrangement.

4.1.2 Point Location

Given a query point q in the plane, our algorithm will find the face f of the curve arrangement where q is located. The output of the algorithm will be the ordered lists of curves (a list for the outer boundary and lists for inner boundaries if they exist) that contribute to the boundary of the face. By convention, the outer list will be ordered counterclockwise and the inner lists will be ordered clockwise.

Boundary Polygons and Intersection Polygons

In order to find the face f where q is located, we need to isolate the curves on f's boundary, and their intersection points. We define a *boundary polygon* of a face f to be a bounding polygon of a single curve (here we use bounding polygon in a general sense, i.e., it can also be a sequence of bounding polygons), such that the portion of the curve that is bounded by it is on a single edge of f, and no other curve intersects this portion. See Figure 4.3 for an illustration. We define an *intersection polygon* to be a polygon that contains a single intersection of two curves (this polygon is an intersection of two bounding polygons that meet certain conditions — see the next paragraph). An intersection polygon corresponds to a vertex of f, and the boundary polygon between two intersection polygons corresponds to an edge of f.

Sufficient Conditions for Intersection Polygons In order to isolate the intersection points of two curves, we use the results obtained by Neagu and Lacolle [41], that were described in Section 2.4.2. In their paper they give sufficient conditions on the control and carrier polygons of two subcurves which guarantee that the subcurves intersect exactly once. Therefore, if the two subcurves meet these conditions then their intersection is inside

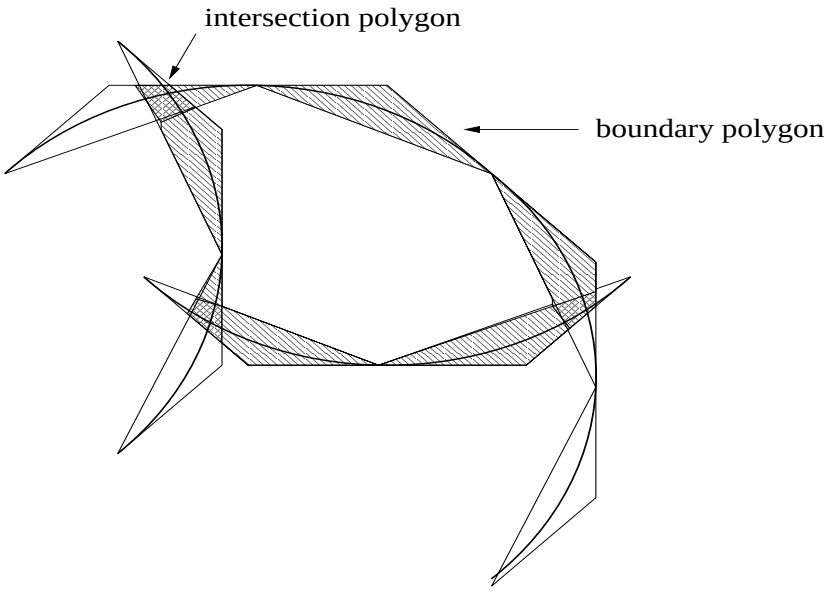


Figure 4.3: A face with boundary and intersection polygons.

the polygon that is the intersection of their two bounding polygons — the intersection polygon.

In our algorithm we check for these conditions after we verify, using the intersection graph, that $\mathcal{C}_1$ and $\mathcal{C}_2$ intersect only each other. Therefore, if the conditions are met then the polygon that is the intersection of the two bounding polygons is the intersection polygon we are looking for.

Finding a Boundary Polygon After vertical ray shooting has been successfully performed from q, we have at least one portion of an edge that will appear on f's boundary, namely the portion of the curve that is bounded by the result polygon of the vertical ray shooting query. The case where the answer to the ray shooting query is empty, i.e., q is in the unbounded face, is dealt like a face with holes (see concluding paragraphs of this section). Let bp be the bounding polygon that was found in the ray shooting query. We know that there are no other bounding polygons that intersect bp (this is the condition in the ray shooting query), therefore the subcurve bounded by bp is on f's boundary. If the neighboring bounding polygons of bp (which can be found by traversing the hierarchy tree in the polygon arrangement) are

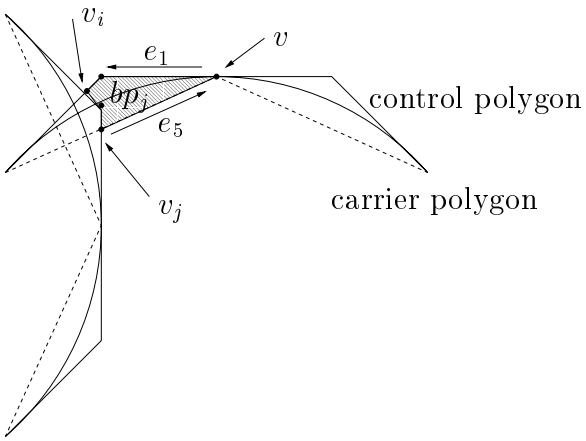


Figure 4.4: Finding the tail of a boundary polygon.

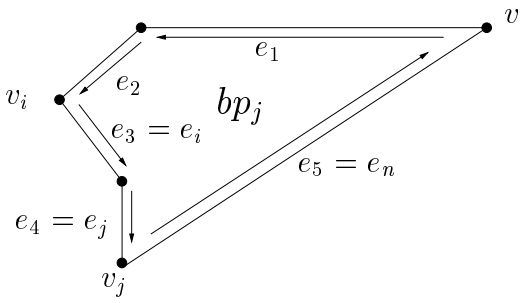


Figure 4.5: An enlargement of the face bp_j of Figure 4.4.

also non-intersecting with any other bounding polygon, then their bounded subcurves are also on f's boundary. We can continue traversing the bounding polygons until we reach a bounding polygon bp_i that is not isolated (i.e., is intersected by another bounding polygon). In principle, we can now check for the intersection-polygon conditions (defined above) in bp_i and the polygon intersecting it. However, in our implementation we add an intermediate step that helps us eliminate certain cases before we check those conditions.

Let bp_j be the polygon (that corresponds to a face in the arrangement of bounding polygons), that is the connected part of the bounding polygon which is incident to the previous isolated polygon (see Figure 4.4 and 4.5). We know that one of its vertices v, which is incident to the isolated bounding

polygon, is on the boundary of f. Therefore in a sufficiently small neighborhood of v the subcurve "going out" of v is also on the edge e of f. In order for bp_j to be part of a boundary polygon, we must verify that the portion of the curve that is bounded by it is on e, i.e., that it is not intersected by any other curve. Let $(e_1, e_2, ..., e_n)$ be the counterclockwise sequence of halfedges around bp_j, where v is the source vertex of e_1 and the target vertex of e_n, Figure 4.5 depicts such a case where there are five edges (i.e., $n = 5$). If e_1 belongs to the control polygon then e_n belongs to the carrier polygon and vice versa. Without loss of generality, let us assume that e_1 belongs to the control polygon (as in Figure 4.5). Let e_i be the first halfedge, when moving counterclockwise starting at e_1, that does not belong to the control polygon. Let e_j be the first halfedge, when moving *clockwise* starting at e_n (i.e., when moving "backwards" against the direction of the halfedges), that does not belong to the carrier polygon (note that e_i can be equal to e_j). We denote by v_i the *source* vertex of e_i and by v_j the *target* vertex of e_j. If the halfedges of the sequence $(e_i, ..., e_j)$ all belong to the control polygon of a single curve, or if they all belong to the carrier polygon of a single curve, then the curve bounded by bp_j (in Figure 4.4 this is the portion of the Bézier curve that is in the dark area in) is not intersected by any other curve. This can be demonstrated using Figure 4.5. If there were another curve c intersecting the curve bounded by bp_j, then its bounding polygons would have had an intersection region with bp_j. Since v is an isolated vertex it cannot be contained inside these bounding polygons and the intersection could only take place to the left of v. But since e_i and e_j are the first edges encountered when going left of v and all the sequence $(e_i, ..., e_j)$ belongs to the same control or carrier polygon, then c can only be to the left of $(e_i, ..., e_j)$. Therefore c does not intersect the curve bounded by bp_j. We call this area of the boundary polygon, that is not intersected by any other curve and is incident to a possible intersection polygon, a *tail* of the boundary polygon (the dark area in Figure 4.4).

If the face (or faces) "on the other side" of $(e_i, ..., e_j)$ (i.e., the neighboring faces that share the edges of $(e_i, ..., e_j)$ with bp_j) is an intersection polygon (as verified by the conditions of the previous paragraph) then we have one "side" of an edge e in f, namely the portion of e that is incident to the vertex created by the intersection. If the neighboring face is not an intersection polygon, then a subdivision on the bounding polygons will take place (see next paragraph).

The Algorithm

Finding the Boundary of a Simply Connected Face If the face f containing q is simply connected, i.e., it has no holes, then the problem is easier and we consider this case first. The first step in the point location query is performing a vertical ray shooting query. By this we achieve two goals: (i) after the ray shooting, the query point q is guaranteed to be outside of any bounding polygon. and (ii) we have an isolated bounding polygon bp_s such that its endpoints are guaranteed to be on the boundary of f (i.e., it is part of a boundary polygon). We can now consider bp_s as a starting point (an "anchor") from which we can traverse the boundary of f, verifying the conditions defined above.

Let f_q be the face in the original arrangement of curves where q is located. We denote by $\hat{f}_q$ the face that contains q in the current arrangement of bounding polygons, where current applies to the arrangement of bounding polygons in the subdivision stage we are currently in. After we have the starting point, we traverse the outer boundary of $\hat{f}_q$ from the starting point bp_s in a counterclockwise order. For every halfedge h in our path we check if the bounding polygon bp_h that is incident to it is part of the boundary polygon. If not, we subdivide bp_h and the bounding polygons that intersect it. We then start the process again from bp_s. If bp_h is a tail of a boundary polygon, then we check if the two intersecting bounding polygons conform to the conditions of an intersection polygon. If the conditions are not met, we perform a subdivision and return to bp_s. If the conditions are met, we continue the process from the next halfedge on the boundary of $\hat{f}_q$ that does not belong to the current curve. The process terminates when we arrive again at the original starting point (the "anchor").

The algorithm can be summarized as follows:

```
find the anchor halfedge ah by shooting a vertical ray from q,
  assign it to h;
  let the halfedge sp=ah;
  let the face f=ah->face();

do {

  //finding the boundary and intersection polygon
  while (true) {
    do {
      while h is on an isolated boundary polygon, advance h around f;
      if the bounding polygon of h is not a tail of a boundary polygon {
        perform a subdivision;
        h=sp; //start the process again from sp
      }
    } until h is on a tail of a boundary polygon;

    if the intersection of the bounding polygons is an intersection polygon {
      break;
    }
    else {
      perform a subdivision;
      h=sp; //start the process again from sp
    }
  }

  add the curve h is on, to the list of curves on the boundary of f;
  advance h to the first halfedge on the next curve;
  sp=h;

} until h reaches the anchor halfedge ah;
```

Dealing with Holes The algorithm described above assumes that there are no holes in f. Therefore, the vertical ray shooting query will give us a starting point on the outer boundary of f. However, if f contains holes, we want to isolate the halfedge cycles around them as well. Furthermore, the vertical ray shooting might return us a curve on a hole and not on the outer boundary. For such cases we need a way to find a starting point for our algorithm without using a vertical ray shooting query. Note that we still need the vertical ray shooting query to guarantee that q is not inside a bounding polygon.

Given that q is not inside a bounding polygon, we would like to find an isolated bounding polygon on the boundary of every hole of $\hat{f}_q$, which will be the "anchor" for our algorithm. If a vertex v of $\hat{f}_q$ is an endpoint of a bounding polygon (i.e., an endpoint of the carrier polygon), then v is on a curve cv of f's boundary. If v is isolated, i.e., it is incident only to bounding polygons of cv, then there exists a neighborhood of v in which cv is not intersected by any other bounding polygon. Therefore, if we subdivide the bounding polygons incident to v we will eventually find a bounding polygon which is in that neighborhood, and is thus isolated. We can then use this bounding polygon as an anchor.

However, there are configurations in which there are no isolated vertices that are polygon endpoints, as demonstrated in Figure 4.6. Still, we can use the following scheme: we find the lowest vertex in the hole's boundary and subdivide its incident bounding polygons, we then resume our search for an isolated vertex. The lowest point p_{lowest} of the boundary is either an endpoint of a curve (in which case we are done) or in the interior of a curve. In the latter case, there is a neighborhood of p_{lowest} that does not intersect any other curve (assuming no degenerate cases which we will discuss in Section 4.1.3). Hence, after sufficiently many subdivision steps there will be an isolated vertex in that neighborhood. The case of an outer boundary is similar, either p_{lowest} is in the interior of a curve or it is an intersection of curves. In either case, there is a neighborhood of p_{lowest} that does not intersect any other curve (again, assuming no degeneracies).

Consequently, in order to find an anchor we go over the halfedges around the hole looking for an isolated bounding polygon, if we do not find one, we look for an isolated vertex and subdivide the bounding polygon incident to it until we have an isolated polygon. If an isolated vertex was not found, we find the lowest vertex on the hole, subdivide its incident bounding polygons

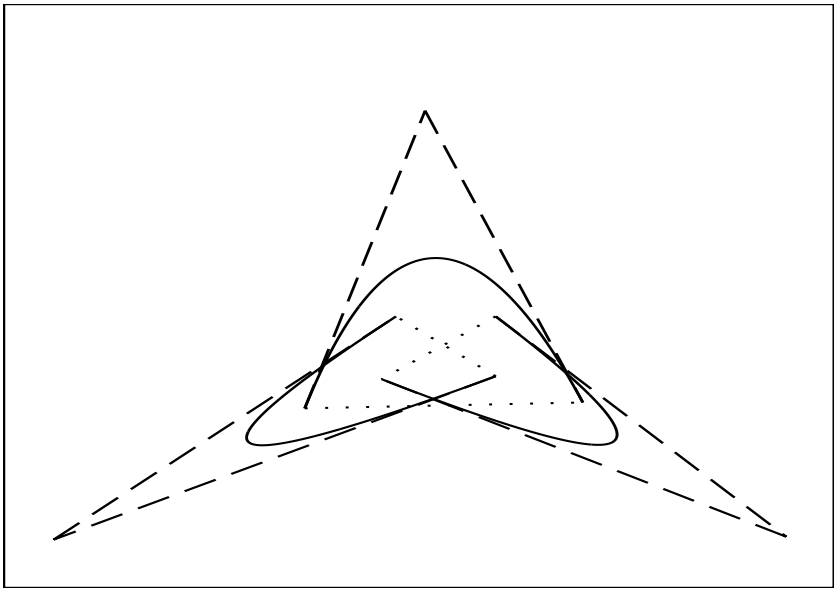


Figure 4.6: A hole that has no endpoint vertices on its boundary.

and resume the search.

4.1.3 Dealing with Degeneracies

Our application is intended to avoid the usage of expensive symbolic computations. However, in some degenerate cases this is unavoidable. In the following paragraphs we will describe some of these cases, and possible ways of dealing with them. We will also describe degenerate cases that arise in the bounding polygons' arrangement and how we deal with them.

Vertical Ray Shooting Degeneracies

The algorithm described in Section 4.1.1 is quite simple. Informally, what we do is subdivide the curve until the query point q is not inside any bounding polygon, and has an isolated bounding polygon above it (both its control and carrier polygon, see Section 4.1.1).

If q is *on* a curve, then we will go on subdividing, since we will always be inside a bounding polygon. Another special case is when q is located exactly beneath an intersection point p of two curves, i.e., it has the same

x-coordinate as p. In this case we will never be under an isolated polygon since the polygon around p will always be an intersection polygon.

Both cases cannot be dealt with with our scheme, and must be resolved by symbolic computation. We have provided in our application a practical way for the user to facilitate this. In the traits class (see Section 4.2.2) the user can define a `passed_threshold` function on a bounding polygon. For example, a function that returns *true* when the bounding polygons diameter is smaller than some small user-defined ε (this is what we use in our current implementation). When the function returns *true*, the subdivision process terminates and the subdivided curves are inserted into a *degeneracy list*. The user can thus query the degeneracy list with symbolic means to determine whether q is on the curve or if it is underneath an intersection of two curves.

Point Location Degeneracies

The algorithm for point location starts with a vertical ray shooting query. If we run into a degeneracy at the ray shooting query we cannot continue, since we might still be inside a bounding polygon. In such a case we terminate the query and return a message describing the situation. The users can then query the degeneracy list. If q is not in one of the degenerate positions described above, the point location query can be resumed with a lower threshold (or no threshold at all).

If the ray shooting query is successful, we can continue the point location query. The conditions described by Neagu and Lacolle [41] assumes general position of the curves, in particular they assume that the curves intersect transversally (i.e., no two curves are tangent to each other and no endpoint of a curve lies in the interior of another curve) and that no three curves intersect at a point. Since our algorithm relies on the conditions given in that paper we cannot deal with these cases directly either. In these cases the subdivision process will terminate when the threshold is passed, and the users can query the degeneracy list. The result of the query is not guaranteed to be totally correct in these cases, i.e., the combinatorial structure can be wrong in some places. Consider for example three curves that intersect at a point. Since we have a threshold ε we do not pass in our subdivision, we cannot distinguish between the case where the three curves intersect at one point and the case where they only pairwise intersect but their intersection points are less than ε away from each other. Similar examples can be found for tangent curves and for endpoints that "touch" another curve.

4.1.4 Degeneracies in the Polygons

There are certain degenerate cases that can occur in the arrangement of bounding polygons. Since the arrangement package underlying our implementation deals with x-degenerate input (e.g., vertical segments) we do not have to take care of such cases. The two main cases we do have to deal with are when two polygons overlap (assuming the original curves do not overlap), and when two curves meet at an endpoint of a bounding polygon.

If two polygons overlap while the curves bounded by them do not, then after sufficiently many subdivision steps the polygons will no longer overlap. Since the arrangement package underlying our implementation deals with overlaps (see Section 3.4) then we do not have to deal with this directly. The conditions for ray shooting and point location fail if there is an overlap. Therefore, as long as there is an overlap our algorithm will keep on subdividing until there is no longer an overlap or until the threshold has been reached.

A special case to be taken into consideration is when the bounding polygons intersect at their endpoints, i.e., when the intersection polygon degenerates to a single point (see Figure 4.1.4). We consider this case separately. If we identify that the intersection point is at an endpoint we would like to verify that the topology of the bounding polygons around the vertex is equivalent to the topology of the curves around the point. We do this by ensuring that edges around the vertex are ordered in consecutive pairs where each pair consists of one edge from a control polygon and one from a carrier polygon, both originating from the same subcurve. If this is not the case we subdivide the bounding polygons.

One degenerate case that is not dealt with in our application is the case of linear interpolation, i.e., when the points of the control polygon are collinear. In this case the bounding polygon degenerates to a line segment. The reason we cannot deal with this situation is that the algorithm for finding the tail of a boundary polygon (see Figure 4.4) assumes we can traverse the inside of the boundary polygon. This might be solved by special treatment for this specific case, however we leave it for future work.

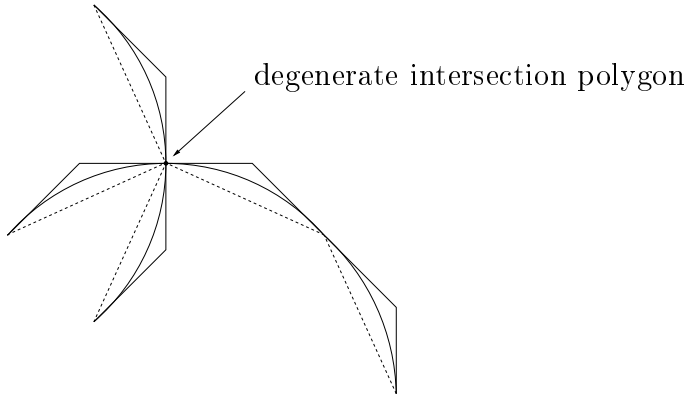


Figure 4.7: A degenerate case where the bounding polygons intersect at an endpoint, the intersection polygon degenerates to a single point.

4.2 The Implementation

We have implemented the algorithms described in this chapter, using the `Arrangement_2` class described in Chapter 3. The class `Adaptive_arr` derives from the class `Arrangement_2`. Its traits class (described in Section 4.2.2) is a superset of the polyline traits for `Arrangement_2`. In its constructor the class gets a sequence of curves and inserts their bounding polygons into the arrangement. It then initializes the intersection graph (in our current implementation we do this with a naive algorithm). When a query is performed the subdivisions take place using the `replace` function which is a protected member function of `Arrangement_2`. This function enables to replace a part of the hierarchy tree without changing the other parts. Since this function can be easily misused and cause inconsistencies in the arrangement, it is not given for public use. An advanced user who wants to use it can do it in a class that is derived from `Arrangement_2`, as we have done. Using the `replace` function enables us to perform a subdivision only on part of the original curve without having to remove the whole curve and reinsert it. In the subdivision process the intersection graph is also updated.

Our application uses the `Pm_walk_along_line_point_location` strategy. Currently this strategy gives the best experimental results (over a more efficient point location structure). This can be partially explained by the fact that our application makes extensive use of the deletion and splitting

functions which are slower for the randomized incremental (default) point location strategy (see [18] for a comparison of the strategies). However, our algorithm does not make any assumptions on the strategy used, therefore if the default strategy will become more efficient in the future it can be used merely by changing one line of code.

4.2.1 Data Structures

The `Adaptive_arr` class stores four classes: the hierarchy tree, the planar map and traits classes inherited from the `Arrangement_2` class, and the intersection graph. In addition, the *curve_node* level of the hierarchy tree stores references to the original curves (this is needed for subdivision schemes that need knowledge of the original curve — see Section 2.4.1). For the intersection graph we use a LEDA UGRAPH class [34, 35] which is a parameterized undirected graph class. The nodes of the intersection graph store references to subcurves in the hierarchy tree. Figure 4.8 depicts the classes and their inter-relationships. In order to store references inside the hierarchy tree, we supply an additional `Info` type in the `Base_node` template parameter that is passed to the `Adaptive_arr` class.

The hierarchy tree used inside `Adaptive_arr` is not the default one. In some cases we are interested in the whole bounding polygon, in others we are interested only in the control or carrier polygon. We also need to cut the polygonal lines into x-monotone segments before we insert them into the planar map. To answer all of these needs we designed a hierarchy of three levels (apart from the curve level which stores the original bounding polygon, and the edge level which stores the segments corresponding to edges in the planar map). The levels are (Figure 4.9 illustrates this hierarchy).

- bounding polygons (subcurve level 0 in Figure 4.9) — in this level we store the whole bounding polygon, oriented counterclockwise. The orientation is important since we assume in some of our procedures that the interior of the bounding polygon is to its left. This level also refers to and is referred to by the intersection graph.

- control polygons and carrier polygons (subcurve level 1 in Figure 4.9) — in this level we store the control and carrier polygons that correspond to the bounding polygon of the previous level.

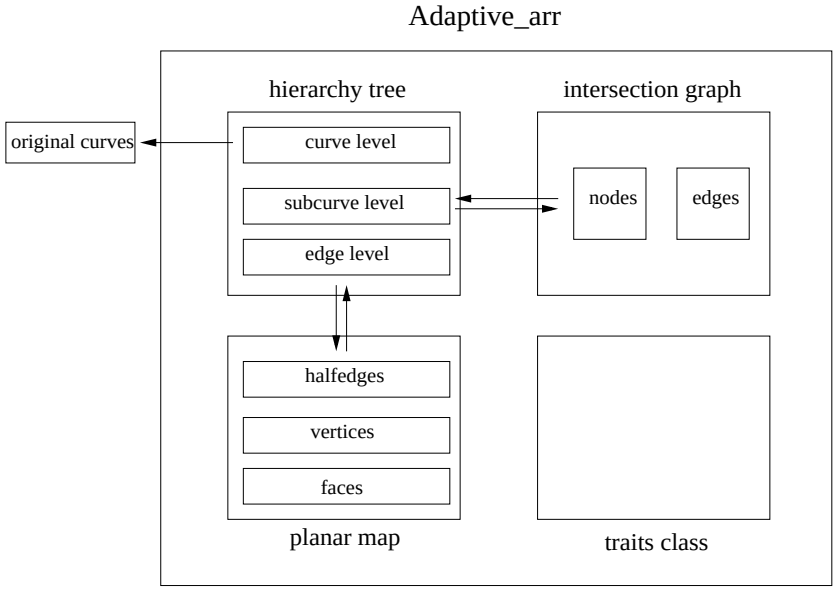


Figure 4.8: A diagram of the classes and inter-relationships in *Adaptive_arr*.

- line segments (subcurve level 2 in Figure 4.9) — in this level we partition the control polygons into line segments. This enables easier handling of the curves and guarantees that the curves inserted into the arrangement are x-monotone.

4.2.2 The Traits Class

As described in Section 2.1 a traits class enables flexibility in the design. In our application, the traits class enables us to use different implementations of polygonal lines, and different curves with varying subdivision schemes. In addition, the threshold function (see Section 4.1.3) is defined in the traits class. We have implemented a traits class for quadratic Bézier curves, using LEDA's rational kernel for the polyline representation. We have also implemented a traits class for quadratic Bézier curves which is template-parameterized by CGAL's kernel.

The traits class for the `Adaptive_arr` class should be in particular a traits class for polygonal lines, since the operations inside `Adaptive_arr` are done on the polygonal lines. Indeed, in our implementations of traits

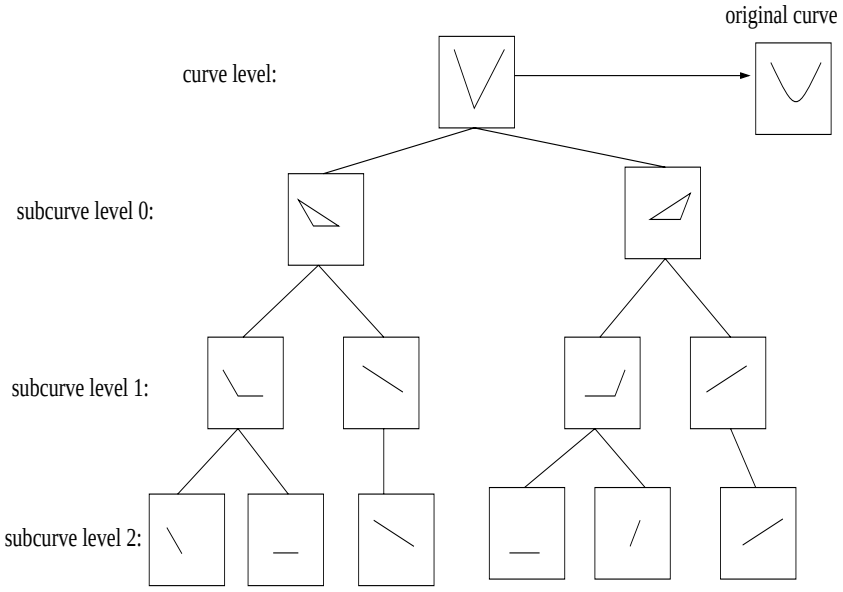


Figure 4.9: An example of a hierarchy tree in *Adaptive_arr*.

classes, we derived our traits classes from the `Arr_leda_polyline_traits` and `Arr_polyline_traits` classes. For our application, additional functionality is needed. First, an additional type `O_curve` is to be defined in the traits class. This type corresponds to the representation of an original curve that is inserted into the arrangement. For Bézier curves which are defined by their control polygons, the representation of the `O_curve` is the same as the `Curve`, however it is required for implementations that use the original curve for their subdivision schemes.

Except for the additional type `O_curve` the following functions are required by the traits class:

- Functions that manipulate control polygons:
  - *Curve to_control_polygon(O_curve oc);* — creates an initial control polygon for *oc* which will be the root of the hierarchy tree.
  - *void subdivide(Curve cv, O_curve oc, list < Curve > & divided_lst);* — subdivides the polyline *cv* and stores the resulting subcurves in *divided_lst*, where *oc* is passed to enable subdivision schemes that might need it.

- Functions that are needed to verify the conditions of Sections 4.1.1 and 4.1.2:
  - *bool ch_intersect(Curve cv1, Curve cv2);* — returns *true* if the convex hulls of the polylines $cv1$ and $cv2$ intersect.
  - *bool ch_fully_contains(Curve cv1, Curve cv2);* — returns *true* if the convex hull of $cv1$ fully contains the convex hull of $cv2$ (for the traits class that is used for quadratic Bézier curves this is done by checking if all three vertices of $cv2$ are inside $cv1$).
  - *int curve_intersections(Curve cv1, Curve cv2);* — returns the number of intersections between $cv1$ and $cv2$.
  - *void intersection_segments(Curve cv1, Curve cv2, Point& i11, Point& i12, Point& i21, Point& i22);* — given that $cv1$ and $cv2$ are intersecting polylines, this function returns the endpoints of the two segments on $cv1$ and $cv2$ that intersect. *i11* and *i12* are the source and target points of the segment lying on $cv1$ and *i21* and *i22* are the source and target points of the segment lying on $cv2$.
  - *bool rightturn (Point p1, Point p2, Point p3);* — returns *true* if the points form a right turn. Although this predicate exists in CGAL, we do not want to restrict ourselves to the CGAL kernel and therefore we give it as a requirement of the traits class (in traits classes that use the CGAL kernel this function calls the CGAL predicate).
- Functions to create the initial hierarchy tree (they need to be static so they can be referenced by a function pointer)
  - *static void first_split(Curve cv, list < Curve > & l);* — returns the bounding polygon oriented counterclockwise. This function creates the first of the subcurve levels in the hierarchy tree (subcurve level 0 in Figure 4.9).
  - *static void split_cntrl(Curve cv, list < Curve > & l);* — returns the control polygons and carrier polygons (subcurve level 1 in Figure 4.9). l is a list of the control polygons (represented as polylines). The matching carrier polygons are defined as the segment that has the last point of cv as its source and the first point as its target.

  - *static void split_sgmnts(Curve cv, list < Curve > & l);* — splits *cv* into segments (subcurve level 2 in Figure 4.9). The polyline in the carrier polygon is partitioned into its segments.

- The threshold function:

  - *bool passed_threshold(Curve cv);* — the function determines if we should stop the subdivision of *cv*.

4.3 Example Program

The following example demonstrates the use of the vertical ray shooting and point location queries inside a program (include files have been omitted for clarity). We use the class `Adaptive_leda_traits` which is a traits class for quadratic Bézier curves that implements polylines as standard `vector`s of the `leda_rat_point` type. `Adaptive_base_node` is a base class for the hierarchy tree that holds additional information which enables to refer from the hierarchy to the intersection graph (see Section 4.2). The program inserts one quadratic Bézier curve (i.e., a control polygon of three points) and performs a vertical ray-shooting and a point-location query. The experimental results from Chapter 5 were obtained with similar programs.

```
typedef CGAL::Adaptive_leda_traits                          Traits;
typedef Traits::Point                                       Point;
typedef Traits::Curve                                       Curve;
typedef Traits::O_curve                                     O_curve;
typedef Traits::X_curve                                     X_curve;

typedef CGAL::Pm_dcel< CGAL::Arr_2_vertex_base< Point >,
                       CGAL::Arr_2_halfedge_base<Adaptive_base_node >,
                       CGAL::Arr_2_face_base >  Dcel;

typedef CGAL::Adaptive_arr<Dcel,Traits,Adaptive_base_node > Ad_arr;

int main()
{
    O_curve c;
    c.push_back(Point(0,0));
```

```
    c.push_back(Point(2,2));
    c.push_back(Point(4,0));

    std::list<O_curve> in_lst; //the list of input curves
    in_lst.push_back(c);
    Ad_arr arr(in_lst.begin(),in_lst.end());

    std::list<Curve> deg_lst; //list where the degeneracies will be stored

    Point q(1,1); //query point

    //vertical ray shooting query
    Ad_arr::Halfedge_iterator hit=arr.vertical_ray_shoot(q,deg_lst);
    //finding the original curve of the result
    O_curve oc = arr.o_curve(hit->edge_node()->curve_node());
    CGAL_assertion(oc == c);

    //point location query
    deg_lst.clear();
    hit=arr.locate(q,deg_lst);

    return 0;
}
```

Chapter 5

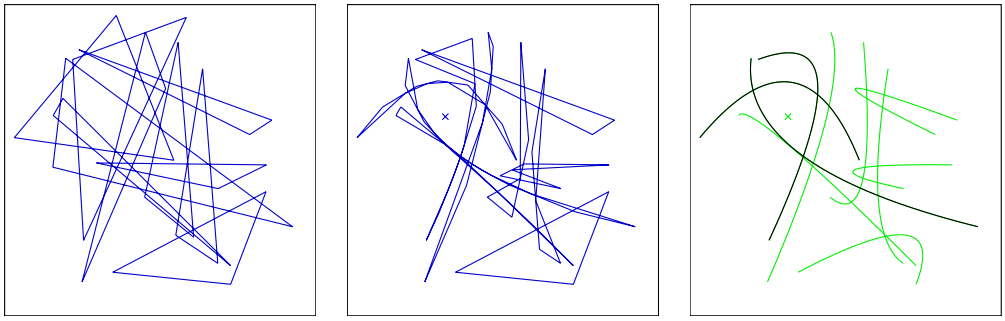
Experiments in Adaptive Point Location

We have conducted several experiments on the adaptive point location application described in this thesis [1]. In this chapter we describe these experiments and their implications. Our main goals in implementing the package were robustness, genericity and flexibility. The package is not an optimized code yet, work is currently underway to speed-up the performance of the package.

5.1 Adaptive Point Location Queries and Initialization

The adaptive point location application is built over the arrangement package using polyline traits. There are many variables which influence the performance of our adaptive point location application. The number of curves and the size of the output (the returned face) are the obvious ones. Apart from them, the performance of our adaptive scheme is influenced by the density of the arrangement — if the curves are dense then more subdivisions need to be performed in order to isolate them. Another parameter is the distance of the query point from the curve, the closer the point is to a curve the larger the number of subdivisions we may need to perform. An important parameter is the number of queries we perform and their relative location. Since our

[1] All experiments were done on a Pentium-II 450Mhz with 528MB RAM memory, under Linux.



(a) (b) (c)

Figure 5.1: An arrangement of 10 Bézier curves and its corresponding polyline arrangement before (a) and after (b) the first point location; (c) displays in bold line the curves that bound the face containing the query point.

algorithm is adaptive, a query in a face that has already been queried should be faster than the first query in that face. Similarly, a query close to a face that has already been queried is also anticipated to be faster since some of the subdivisions have already been performed for the neighboring face.

We have constructed test inputs of ten random sample sets of quadratic Bézier curves (using CGAL's random triangle generator). Figure 5.1 shows such a random set with 10 curves; it also depicts the corresponding segment arrangement before and after the first point location query. The curves of the located face are colored darker in the Bézier arrangement. We have run the test inputs on 50 random points distributed evenly in a circle centered at the origin with a radius of 100 units (the curves are distributed in a circle of radius 400). Figure 5.2(a) shows the average time per query as a function of the number of queries already performed. We can see clearly that the time reduces considerably as we make more queries.

There is a trade-off between the initialization step (in which the initial intersection graph is constructed) and the rest of the algorithm. If we subdivide the initial control polygon in the initialization step we prolong this step; however, since the resulting bounding polygon is closer to the curve the query will take less time. We timed the initialization step and Figure 5.2(b) shows the first point location query (as Figure 5.2(a) shows, this is the significant query) as a function of the number of curves. Again, as can be expected, the initialization and query time grow as the number of curves increases. We

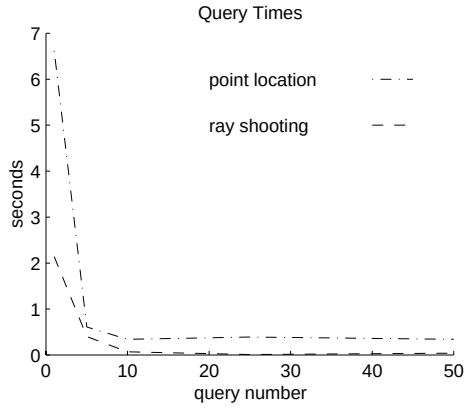


Figure 5.2: The reduction in the average query time as a function of the number of queries.

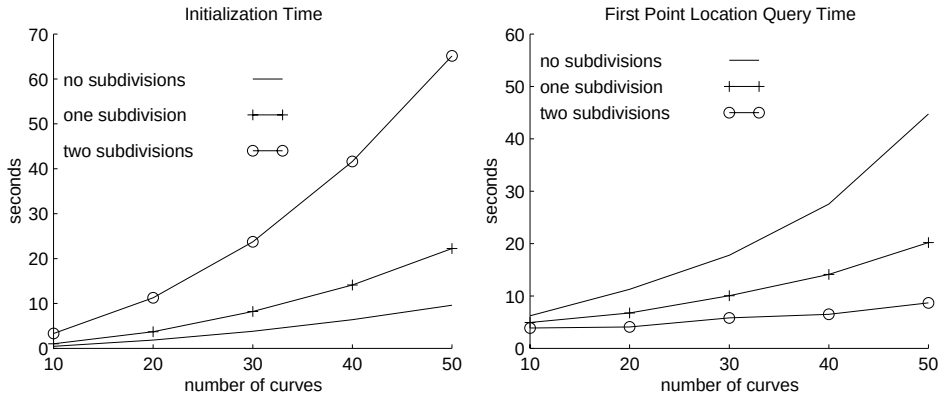


Figure 5.3: The trade-off between subdivision at the initialization step and at the queries.

repeated the procedure, performing one and two subdivisions at the initialization step. Again, as can be expected, the time for the point location query reduces while the initialization time increases as more subdivision steps are performed at the preprocess step. Implementing this change amounted to changing a few lines in a single function of the traits class, therefore the users can experiment with this trade-off for their needs.

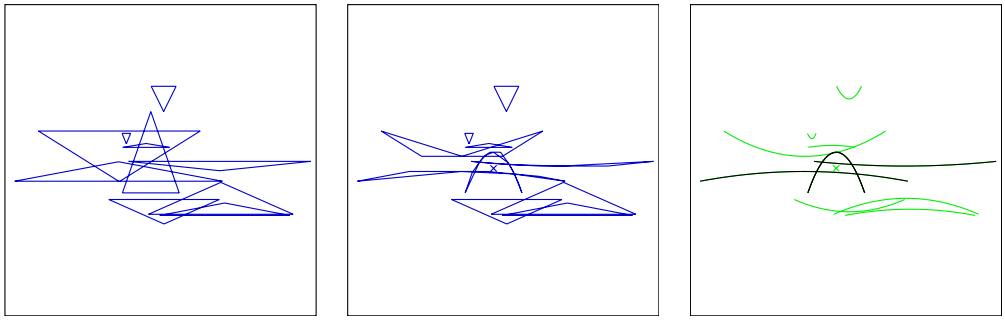
5.2 Comparison with Arrangement of Canonical Parabolas

Constructing arrangements of Bézier curves with exact arithmetic is a difficult task. Computing the intersection point of two quadratic Bézier curves is equivalent to finding the roots of a degree-4 polynomial (i.e., solving the quartic equation). Although there are known analytical solutions to the quartic equation (see, for example, [9]), we do not know how to implement them using algebraic number packages such as `leda_real`. The reason is that the solution requires finding the cubic root of a complex number; this in turn requires trigonometric functions, which these packages do not support.

In order to compare the method described in Chapter 4 with exact algebraic methods we compared it to the restricted case of arrangements of canonical-parabola arcs. As mentioned in Chapter 3 we have constructed a traits class for canonical-parabola arcs, which can be used with algebraic number types (in our experiments we used `leda_real` which is the fastest implementation of such a package known to us [10]). This traits class is easier to implement since finding the intersection point amounts to solving a quadratic equation. Still, the comparison can give us an idea of the advantages and disadvantages of our method compared to algebraic methods.

For the experiment we conducted we prepared random inputs of control polygons for canonical parabolas. The control polygons were symmetric triangles with a vertical symmetry axis (see Figure 5.4). These were the input for the adaptive point location application. The coefficients of the canonical parabolas were computed from the control points[2].

[2]This was done in a special constructor in the traits class. Given the control points $p0, p1, p2$ the coefficients a, b, c of the parabola equation $y = ax + by + c$ were computed using the following expressions: $a = (p0_y - p1_y)/2(p0_x - p1_x)^2)$; $b = -2ap1_x$; $c = (b^2/4a) + (p0_y + p1_y)/2$.



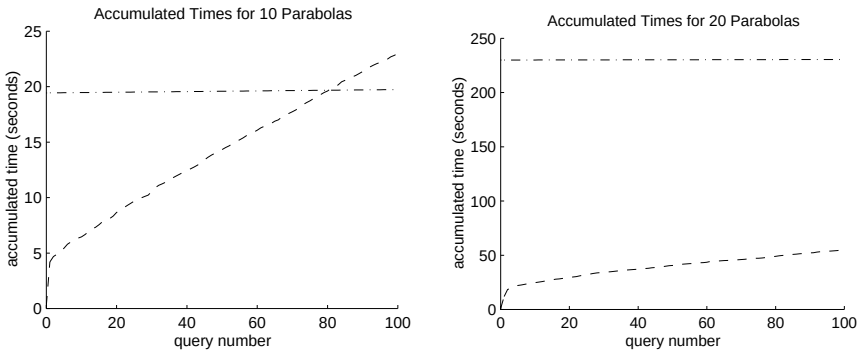
(a) (b) (c)

Figure 5.4: An arrangement of 10 canonical-parabola arcs and its corresponding arrangement of polylines before and after the first point location.

We timed the construction of the arrangement and then conducted a series of 100 point location queries. The accumulated time, i.e., the time passed from the beginning of the construction to the end of the query, for ten and twenty curves (on average over the inputs) are depicted in Figure 5.5. It shows that constructing the non-adaptive arrangement is very slow, whereas its queries are very fast (negligible compared to the construction time). On the other hand, the adaptive arrangement's construction is quite fast, while the queries show the same behavior already shown in Figure 5.2 above. Although the adaptive queries are slower than those of the non-adaptive arrangement, they are preferable in many cases because of the latter's long construction time. For small arrangements (see Figure 5.5(a)) with a sufficiently large number of queries, the non-adaptive arrangements are preferable. We anticipate that for arrangements of general (non-canonical) Bézier curves the non-adaptive operations will be much slower because of the complexity of the predicates. No such change is expected in the adaptive point location since the operations are the same as the ones used for canonical parabolas.

After the construction of the canonical parabola arrangement the point location query is very fast. In order to investigate this behavior we designed a special traits class `Arr_statistic_traits` which gave us information about the use of the predicates[3].

[3]Implementing concrete concepts for profiling is a common practice when using generic programming, see for example [38]. In our case we extended the practice to a generic traits



(a) (b)

Figure 5.5: A comparison of the accumulated time for construction and queries of arrangements of canonical-parabola arcs, using the canonical-parabola traits (dash-dotted) and the adaptive scheme presented in this work (dashed): (a) Arrangement of 10 parabolas and (b) arrangement of 20 parabolas.

The reason for the fast operation of the point location query is that once the topological structure of the arrangement has been created, the point location query is reduced to a series of predicates of the type "is the point above or below the curve?".This query is implemented in the traits `curve_get_point_status` function. The implementation of this query is to substitute the coordinates of the query point in the parabola equation and evaluate its sign. Since the query point is not close to most (if not all) of the canonic parabolas in the arrangement, and its expression tree is not very deep, this predicate can easily be resolved using the floating point filter inside `leda_real`. This was verified by the statistics that showed the only predicates used in the query were `curve_get_point_status` and `compare_x`. The `compare_x` functions in the query were never between two points sharing the same x-coordinate, therefore the use of floating point in them was sufficient.

Construction of arrangements is always much slower than a point location query (since every insertion of a curve starts with a point location, the construction of an arrangement of n curves is slower by a factor of at least n than a point location query). However, in this case there are additional reasons for the slowdown of the construction time. Profiling the construction showed that most of the time was spent in the intersection predicate

class (a "meta-traits" class) that gets a traits class as a parameter and profiles it.

`find_nearest_intersection_to_right`, that finds the nearest intersection of two x-monotone curves to the right of a given point p. This predicate is called when inserting a curve c to find the next intersection point of c with the face it is in. The implementation of this predicate is to find the intersection points of the two parabolas, check which of these intersection points are on the parabolic arcs, and compare their position to p. The expression for the intersection points amounts to solving the quadratic equation $(a_1 - a_2)x^2 + (b_1 - b_2)x + (c_1 - c_2) = 0$ to find the points' x-coordinates and substitute the result in one of the equations to get the points' y-coordinates. This results in a considerably deeper expression tree compared to the "below/above" predicate. As shown in [10], deep expression trees slow down the computation considerably. Furthermore, in many cases the given point p is one of the two intersection points. These cases are especially slow since the coordinates are equal. In such cases the computation has to reach the separation bound (see Section 2.2.1) and cannot be resolved by the floating point filter.

On the other hand in our adaptive scheme we construct a polygonal lines arrangement. This enables us to use the traits class that is implemented with LEDA's fast rational kernel. Furthermore, since our scheme is adaptive, we do not perform a lot of computations on parts of the arrangement that are far from the query point. The graphs in Figure 5.5 also show the behavior described in Figure 5.2 — as more queries are performed in the same region, the query time is reduced. However, there is always an overhead in the query time compared to the canonical parabola queries. This is because the adaptive point location algorithm traverses over the boundary of the located face to verify the sufficient conditions on the boundary and intersection polygons. We may be able to improve the algorithm by coloring faces in the polygon arrangement after a point location query has been performed. If a query point is inside such a face, we will not need to check its boundary and intersection polygons. Of course, this improvement also requires maintenance of the coloring, when the arrangement changes.

It should be noted that the choice of input representation influenced the results of these experiments. Namely, since the input was represented as control polygons, an additional computation was needed to transform it to the canonical parabola representation. This created expression trees in the `leda_real`s that were larger than the ones we would have gotten had we used a different representation (e.g., representing the input as the coefficients of the parabola and the source and target points). However, in applications of

parametric algebraic curves this is the natural representation.

We have shown that our scheme compares favorably to an implementation using `leda_real`s even for the restricted case of canonical parabolas. This demonstrates the strength of our implementation.

Chapter 6

Another Application: Boolean Operations

We have tested the software packages described in this thesis on a number of programs and applications. In this chapter we present an example of such an application that emphasizes the advantages of our design and demonstrates its use. We show how our arrangement package can be used to perform boolean operations (such as intersection) on closed curves (such as polygons). We demonstrate how the flexibility of the design can be applied, and how robustness is maintained. The function that performs these operations is a good example of code that uses our package[1].

Given a set of N polygons in the plane we wish to find the regions that are the intersection of all polygons in the set. The way we do this is to label each face in the subdivision induced by the polygons with a *covering number*. The covering number represents the number of original polygons that cover the face. The faces that have covering number N constitute the intersection. The first step of this process is therefore to find the subdivision induced by the polygons, i.e., compute the polygon arrangement. Then we proceed to label each face in the arrangement with its covering number.

Consider the simple case where no edge overlaps occur in the arrangement. Given a face f with covering number c_f, a neighboring face f' (i.e., a face sharing an edge with f) will either have a covering number $c_f + 1$ or $c_f - 1$. If in crossing the shared edge from f to f' we "go out" of a polygon, then the

[1]The code, and graphic programs using it can be found in http://www.math.tau.ac.il/~hanniel/ARRG00/.

covering number is decreased, if we "go into" a polygon then the covering number is increased. This presents a simple recursive labeling scheme: for each neighboring face f' if the neighboring face is not yet labeled, label it with $c_f + 1$ or $c_f - 1$, and recursively perform the operation on f'. We begin the recursion with the unbounded face that has, by definition, a zero covering number.

To implement this function we add a `counter` attribute to the faces of the arrangement. This counter corresponds to the covering number. We initialize it with -1 which will represent "unlabeled" in our function (thus we can distinguish between labeled and non-labeled faces in our function). The definitions for this special arrangement are:

```
struct Face_with_counter : public Arr_2_face_base {
  Face_with_counter() : Arr_2_face_base(), counter(-1) {}
  int counter;
};

//a DCEL with the Face_with_counter
typedef Pm_dcel<Arr_2_vertex_base<Point>,
                Arr_2_halfedge_base<Base_node >,
                Face_with_counter
                >                   Dcel;

typedef Arrangement_2<Dcel,Traits,Base_node > Arr_2;
```

We then implement the recursive function `covering_DFS` in the following manner:

```
1   void covering_DFS(Face_handle f) {
2     Ccb_halfedge_circulator start,circ;
3
4     if (f->does_outer_ccb_exist()) {
5       start = circ = f->outer_ccb();
6       do {
7         if (circ->twin()->face()->counter == -1) {
8           int diff = face_diff(circ);
9           circ->twin()->face()->counter = (f->counter + diff);
10          covering_DFS(circ->twin()->face());
11        }
12      } while (++circ != start);
13    }
14
15    Holes_iterator hit = f->holes_begin();
16    for (; hit!=f->holes_end(); ++hit) {
17      start = circ = (*hit);
18        do {
19          if (circ->twin()->face()->counter == -1) {
20            int diff = face_diff(circ);
21            circ->twin()->face()->counter = (f->counter + diff);
22            covering_DFS(circ->twin()->face());
23          }
24        } while (++circ != start);
25    }
26  }
```

Lines 4–13 perform the recursive function for neighboring faces that share an edge from the *outer* boundary of the face (if it exists — for the outer face it does not exist), and lines 15–25 do the same for the *inner* boundaries. If the neighboring face[2] is unlabeled (lines 7 and 19) then its `counter` should be updated. `diff` (lines 8 and 20) is defined to be either 1 or -1 by the function `face_diff`, and the neighbors counter is updated accordingly. Lines 10 and 22 call the function recursively.

[2]The sequence `circ->twin()->face()` gives us the neighboring face, on the "other side" of the halfedge `circ`.

The face_diff function defines whether we are passing from a bounded side of the polygon to an unbounded side. In order to perform it we assume that all the polygons were inserted in counterclockwise order, this can be easily verified when inserting the polygons into the arrangement (and the polygon can be reversed if necessary). Assuming this, we are inside a polygon if the curve we are crossing (the curve on the edge) has the same orientation as the halfedge we are crossing (i.e., the halfedge that belongs to the face we are in). This is checked by comparing the source and target vertices of the halfedge with the source and target points of the subcurve underlying it. It should be noted that this works because we store the subcurves in the hierarchy tree in the original orientation that they were inserted into the arrangement, otherwise this test could not have been performed. The face_diff function is as follows:

```
int face_diff( Ccb_halfedge_const_circulator circ) {
  Traits t;
  if (circ->source()->point() ==
      t.curve_source(circ->edge_node()->curve()) )
    return -1;     //we're inside, going outside
  else
    return 1;
}
```

The above function is sufficient for the cases where there are no edge overlaps. However, in the case where some of the polygons boundaries can overlap we may have a difference that is greater than 1 less than -1 or even zero (for example two polygons that intersect only at a segment). In order to account for these degenerate cases a more sophisticated face_diff function should be implemented. The following function goes over all the overlapping curves on the halfedge that is crossed, performs the test described above for each curve, and accumulates the difference accordingly.

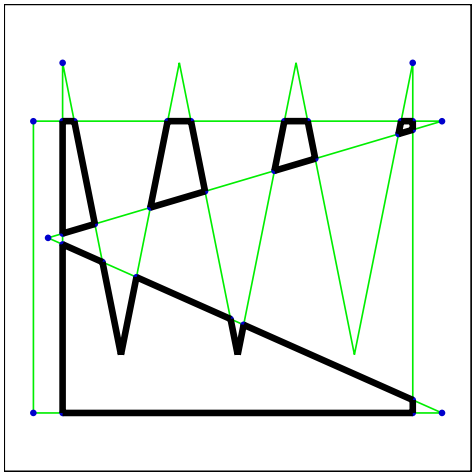


Figure 6.1: An intersection of two polygons computed using the Boolean Operations application. The boundary of the intersection is in bold line. Note the overlapping segment at the bottom of the figure.

```
//generalized face_diff function, to acount for overlaps.
int face_diff (Ccb_halfedge_const_circulator circ) {
  Traits t;
  int diff = 0;
  Arr_2::Overlap_circulator oc = circ->overlap_edges();
  do {
    if (circ->source()->point() == t.curve_source(oc->curve()) )
    diff--;     //we're inside, going outside
  else
    diff++;
  } while (++oc != circ->overlap_edges());

  return diff;
}
```

Figure 6.1 shows the intersection of two polygons that was created using the function `covering_DFS`[3]. It calls the function with the arrangement of

[3]It was generated by the `Polygon_intersect` program that can be found in http://www.math.tau.ac.il/~hanniel/ARRG00/.

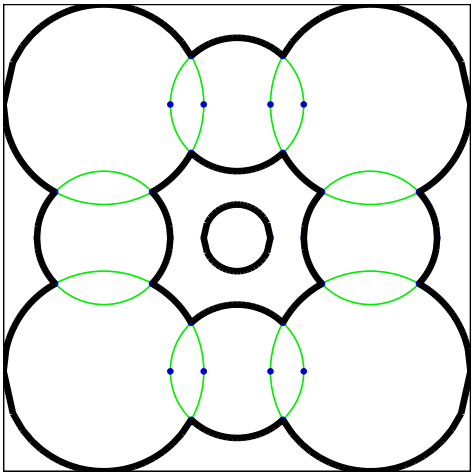


Figure 6.2: A union of a set of circles. The boundary of the union is in bold line.

the two polygons, and then traverses the faces of the arrangement coloring the faces that are covered by the two polygons. Notice that the polygons in Figure 6.1 have an overlapping segment, this is handled by the application in the way described above.

The function `covering_DFS` is not restricted to intersections. The same function can be used, for example, for computing the boundary of the union of polygons: instead of coloring the boundary of the faces with N-covering, we color those with non-zero covering. Furthermore, the function can be used for any "well behaved" closed curve that has an appropriate traits class, and follows the convention that it is oriented with its bounded side to the left. Figure 6.2 shows the union of a set of circles that was created in this way[4]. Finding the intersection of polygons with holes in them can also be performed by inserting the "hole" polygons into the arrangement ordered clockwise instead of counter clockwise. The `covering_DFS` function, without any modifications, will then decrease the counter when going "into" a hole.

This application demonstrates some of the strengths of our implementation. Using our package the application was easily programmed, dealing with degeneracies in a robust manner. Moreover, our traits-based design

[4]It was generated by the `Circle_union` program that can be found in http://www.math.tau.ac.il/~hanniel/ARRG00/.

enables the application to be used for a variety of closed curves and curve implementations, as we have done for polygons and circles.

Chapter 7

Conclusions

We presented a robust, generic and flexible software package for 2D arrangements of general curves. Special care was taken to ensure robustness and to deal with degeneracies. We introduced the curve hierarchy tree structure and a software design which implements it, that adds functionality to the arrangement enabling users to decompose the curves of the arrangement without loss of information. Generic programming techniques were used for dealing with robustness issues and to make the package general (enabling different curves) and extensible.

We also presented an application based on the arrangement package for adaptive point location in arrangements of piecewise convex parametric algebraic curves. The idea of the application is to perform the queries on the bounding polygons of the curves. If the bounding polygons do not enable us to solve the queries, subdivisions are performed on the polygons, giving a finer approximation of the original curves. We have implemented this application for quadratic Bézier curves, and presented some experimental results.

A generic planar map overlay implementation is currently being developed on top of our arrangement package. In this implementation references to the original maps will be kept enabling, for example, a hierarchy of overlays. Such functionality cannot be achieved in more limited overlay applications such as the one described in Chapter 6. Another package that has recently been implemented based on our arrangement package is for *snap rounding* arrangements of segments [24]. The new package coarsens a given arrangement of segments so that it could be maintained with standard computer arithmetic (e.g., machine integer).

The idea of a curve hierarchy structure has also been adopted for CGAL's

triangulation package [6]. The triangulation developers have extended the constrained triangulation to deal with possibly intersecting constraints. They are also currently working on *conforming triangulations* — constrained triangulations in which additional vertices are added on the constraints such that each constraint is cut into subconstraints which are edges in the Delaunay triangulation of the set of vertices. In both cases the input constraint is split into subconstraints adding new vertices. In order to be able to go back from the edges of the resulting triangulation to the original input constraints they use a (limited) variant of the curve hierarchy introduced in this work.

The work described in this thesis is a framework that can be extended and further improved. Work on improving and speeding-up the arrangement package is currently underway in two main directions: improving the internal algorithms (e.g., the algorithms for inserting new curves into the arrangement) and implementing new traits classes (e.g., traits classes that make use of filtering schemes, and for additional types of curves). In the near future an implementation of a traits class for conic section arcs is planned.

The adaptive point location application will also benefit from the improvements described above. It can also be improved in other ways. The initialization of the intersection graph is currently done using a naive algorithm that compares all pairs of bounding polygons, less naive algorithms can be implemented. The non-efficient initialization step is the main reason we have made our algorithm static. Adding a new bounding polygon bp to the arrangement requires an update of the whole intersection graph, checking each bounding polygon whether it intersects bp. If the intersection graph is maintained efficiently then the adaptive algorithm can be made dynamic, namely allow insertions and deletions of curves.

The adaptive point location application can also be applied to other parametric curves such as splines and NURBS. Non-convex parametric curves can also be used with our application, by cutting them into convex subcurves. Doing so is not always a trivial task and might require the use of exact algebraic arithmetic. However, the computation required for the task is usually less complicated than finding the intersection of two curves. Implementing traits classes for these curves is also left for future work.

Bibliography

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

[2] G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, 1983.

[3] N. Amenta. Directory of computational geometry software. http://www.geom.umn.edu/software/cglist/.

[4] N. Amenta. Computational geometry software. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 52, pages 951–960. CRC Press LLC, Boca Raton, FL, 1997.

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

[6] J. D. Boissonnat, O. Devillers, M. Teillaud, and M. Yvinec. Triangulations in cgal. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 11—18, 2000.

[7] H. Brönniman, L. Kettner, S. Schirra, and R. Veltkamp. Applications of the generic programming paradigm in the design of CGAL. Technical Report MPI-I-98-1-030, Max-Planck-Insitut für Informatik, 66123 Saarbrücken, Germany, 1998.

[8] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient arithmetic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165—174, 1998.

[9] R. S. Burington. *Handbook of Mathematical Tables and Formulas.* McGraw-Hill, 3rd edition, 1962.

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

[11] C. Burnikel, J. Könnemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995.

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

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

[14] O. Devillers and F. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete and Computational Geometry*, 20:523–547, 1998.

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

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

[17] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design.* Academic Press, 3rd edition, 1993.

[18] E. Flato, D. Halperin, I. Hanniel, and O. Nechushtan. The design and implementation of planar maps in CGAL. In *Proc. of the 3rd Workshop of Algorithm Engineering*, volume 1668 of *Lecture Notes Comput. Sci.*, pages 154–168. Springer-Verlag, 1999.

[19] S. Fortune. Progress in computational geometry. In R. Martin, editor, *Directions in Geometric Computing*, pages 81–128. Information Geometers, 1993.

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

[21] S. Fortune and C. V. Wyk. *LN User Manual.* AT&T Bell Laboratories, 1993.

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

[23] M. Goldwasser. An implementation for maintaining arrangements of polygons. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C32–C33, 1995.

[24] M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.

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

[26] L. Guibas and M. Karavelas. Interval methods for kinetic simulations. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 255—264, 1999.

[27] 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.

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

[29] D. Halperin and C. R. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *Comput. Geom. Theory Appl.*, 10:273–287, 1998.

[30] I. Hanniel and D. Halperin. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *Proc. of the 4th Workshop of Algorithm Engineering*, 2000.

[31] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulations using rational arithmetic. *ACM Trans. Graph.*, 10(1):71–91, Jan. 1991.

[32] L. Kettner. Designing a data structure for polyhedral surfaces. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 146–154, 1998.

[33] J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: a library for efficient manipulation of algebraic points and curves. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 360—369, 1999. http://www.cs.unc.edu/~geom/MAPC/.

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

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

[36] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley, 1996.

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

[38] D. R. Musser. Measuring computing times and operation counts of generic algorithms. http://www.cs.rpi.edu/ musser/gp/timing.html.

[39] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.

[40] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[41] M. Neagu and B. Lacolle. Computing the combinatorial structure of arrangements of curves using polygonal approximations. Manuscript. A preliminary version appeared in Proc. 14th European Workshop on Computational Geometry, 1998.

[42] K. Ouchi. Real/Expr: Implementation of exact computation. M.Sc. thesis, New York University, 1997.

[43] K. Ouchi and C. Yap. Real/Expr home page. http://cs.nyu.edu/exact/realexpr/.

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

[45] S. Schirra. A case study on the cost of geometric computing. In *Proc. of ALENEX*, 1999.

[46] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1(1):51–64, 1991.

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

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

[49] J. Siek and A. Lumsdaine. Generic programming for high performance numerical linear algebra, 1998. http://www.lsc.nd.edu/~jsiek/mtl_scitools/.

[50] S. Skiena. *The Algorithm Design Manual*. Telos/Springer-Verlag, 1997. http://www.cs.sunysb.edu/~algorith/index.html.

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

[52] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[53] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997.

[54] C. K. Yap and T. Dubé. The exact computation paradigm. In D. Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.