Advanced Programming Techniques Applied to CGAL's Arrangement Package*

Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin

School of Computer Science
Tel-Aviv University, Israel
{wein, efif, baruchzu, danha}@post.tau.ac.il

ABSTRACT

Arrangements of planar curves are fundamental structures in computational geometry. Recently, the arrangement package of CGAL, the Computational Geometry Algorithms Library, has been redesigned and re-implemented exploiting several advanced programming techniques. The resulting software package, which constructs and maintains planar arrangements, is easier to use, to extend, and to adapt to a variety of applications, is more efficient space- and time-wise, and is more robust. The implementation is complete in the sense that it handles degenerate input, and it produces exact results. In this paper we describe how various programming techniques were used to accomplish specific tasks within the context of Computational Geometry in general and Arrangements in particular. A large set of benchmarks assured the successful applications of the adverted programming techniques. The results of a small sample are reported at the end of this article.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Patterns; D.1.5 [**Object-oriented Programming**]

General Terms

Computational geometry, CGAL, arrangements, generic programming, design patterns

1. INTRODUCTION

Given a set $\mathcal{C}$ of planar curves, the *arrangement* $\mathcal{A}(\mathcal{C})$ is the subdivision of the plane induced by the curves in $\mathcal{C}$ into maximally connected cells. The cells can be 0-dimensional (*vertices*), 1-dimensional (*edges*), or 2-dimensional (*faces*). The *planar map* of $\mathcal{A}(\mathcal{C})$ is the embedding of the arrangement as a planar graph, such that each arrangement vertex corresponds to a planar point, and each edge corresponds to a planar subcurve of one of the curves in $\mathcal{C}$. Arrangements and planar maps are ubiquitous in computational geometry, and have numerous applications (e.g., [13, 22]), so many potential users in the academia and in the industry may benefit from a generic implementation of a complete software package that constructs and maintains planar arrangements.

CGAL [1], the Computational Geometry Algorithms Library, is the product of a collaborative effort of several sites in Europe and Israel, aiming to provide a generic and robust, yet efficient, implementation of widely used geometric data structures and algorithms. The library consists of a geometric *kernel* [17, 24], which in turn consists of constant-size non-modifiable geometric primitive objects (such as points, line segments, triangles, etc.) and predicates and operations on these objects. On top of the kernel layer, the library consists of a collection of modules, which provide implementations of many fundamental geometric data structures and algorithms. The arrangement package is a part of this layer.

The software described in this paper rigorously adapts, as does CGAL in general, the *generic programming* paradigm [6], making extensive use of C++ class-templates and function-templates. The generic-programming paradigm uses a formal hierarchy of abstract requirements on data types referred to as *concepts*, and a set of components that conform precisely to the specified requirements, referred to as *models*.

In software engineering, *design patterns* are frequently used to supply standard solutions to common problems recurring in software design. Design patterns supply a systematic high-level approach that focuses on the relations between classes and objects, rather than the specification of individual components. See the book by Gamma *et al.* [20] for a catalog of the most common design patterns.

While relations between objects in a design pattern are usually realized in terms of abstract data types and polymorphism, design patterns can successfully be applied in generic programming as well, as we show in this paper. A good example are the point-location algorithms supplied by the arrangement package. One of the most important operations on arrangements is answering the *point-location* query: Given a query point q, find the arrangement cell that contains q. We supply several point-location algorithms, and

*Work reported in this paper has been supported in part by IST Programme of the EU as a Shared-corst RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes) and by the Hermann Minkowski – Minerva Center for Geometry at Tel Aviv University.

enable package users to employ the algorithm best suited for their application. To this end, we use the ***strategy*** design-pattern, which defines a family of algorithms, each implemented by a separate class, and we make them interchangeable. The four point-location classes are: `Arr_naive_point_location`, which locates the query point naïvely, by exhaustively scanning all arrangement cells; `Arr_walk_along_a_line_point_location`, which simulates a traversal along an imaginary vertical ray emanating from infinity and directed toward the query point; `Arr_landmarks_point_location`, which uses a set of "landmark" points, whose locations in the arrangement are known. Given a query point, it uses a nearest-neighbor search structure (e.g., KD-tree) to find the nearest landmark and then it traverses the straight line segment connecting this landmark to the query point. Finally, the `Arr_trapezoidal_ric_point_ location` implements Mulmuley's point-location algorithm [29], which is based on the vertical decomposition of the arrangement into pseudo-trapezoids. The last two strategies are more efficient. However, they require preprocessing and consume more space, as they maintain auxiliary data-structures. The first two strategies do not require any extra data and operate directly on their associated arrangements.

In classic object-oriented programming, the point-location process can be realized with an abstract base class that provides a pure virtual function, `locate(q)`, which accepts a point q and results with the arrangement cell containing it. All concrete point-location classes inherit from the base class, and all arrangement algorithms that issue point-location queries use a pointer to an abstract base object, which actually refers to one of the concrete point-location classes. When using generic programming, we rely less on inheritance or virtual functions. Instead, we define a concept named *ArrangementPointLocation_2*, such that all models of this concept must supply a `locate()` function. All the various point-location classes model this concept. Note that the concept definition has no trace in the actual C++ code, so from a syntactical point of view, these classes are completely unrelated. Any generic algorithm that issues point-location queries is implemented as a template parameterized by a point-location class, which is a model of the *ArrangementPointLocation_2* concept.

In the rest of the paper we show how additional design patterns are exploited in the CGAL arrangement package in conjunction with generic programming techniques. The application of combinations of advanced programming techniques is argued to be synergistic. Not only does it make the implementation more generic, it also improves the quality of the software in all measured aspects.

1.1 Related Work

In the classic computational geometry literature two assumptions are usually made to simplify the design and analysis of geometric algorithms: First, inputs are in "general position". That is, degenerate cases (e.g., three curves intersecting at a common point) in the input are precluded. Secondly, operations on real numbers yield accurate results (the "real RAM" model, which also assumes that each basic operation takes constant time). Unfortunately, these assumptions do not hold in practice. Thus, an algorithm implemented from a textbook may yield incorrect results, get into an infinite loop, or just crash, while running on a degenerate, or nearly degenerate, input (see [26, 32] for examples). This is one of the problems addressed successfully by CGAL in general and by the CGAL arrangement package described here in particular.

The need for robust software implementation of computational geometry algorithms has driven many researchers to develop variants of the classic algorithms that are less susceptible to degenerate inputs over the last decade. At the same time, advances in computer algebra enabled the development of efficient software libraries that offer exact arithmetic manipulations on unbounded integers, rational numbers (e.g., GMP — Gnu's multi-precision library [4]) and even algebraic numbers (the CORE [2] library and the numerical facilities of LEDA [5]). These exact *number types* serve as fundamental building-blocks in the robust implementation of many geometric algorithms [37].

Keyser *et al.* [12, 27] implemented an arrangement-construction module for algebraic curves as part of the MAPC and ESOLID libraries. However, their implementations make some general position assumptions. The LEDA library [5, 28] includes geometric facilities that allow the construction and maintenance of arrangements of line segments.

CGAL's arrangement package was the first complete software-implementation, designed for constructing arrangements of arbitrary planar curves and supporting operations and queries on such arrangements. More details on the design and implementation of the previous versions of the package can be found in [18, 23]. Many users (e.g., [11, 14, 21, 25, 31]) have employed the arrangement package to develop a variety of applications.

In this paper we show how concurrent applications of advanced programming techniques improve the quality of the CGAL arrangement software-package, achieving a software design according to the generic-programming paradigm that is more modular and easy to use, and an implementation, which is more extensible, adaptable, and efficient.

1.2 Outline

The rest of this paper is organized as follows: Section 2 provides the required background on CGAL's arrangement package, introducing key terms and presenting its architecture. The four succeeding sections describe the applications of four different design patterns within the generic programming paradigm, namely ***adapter***, ***decorator***, ***observer***, and ***visitor***. These sections detail the pattern intent, their impact, and implementation in the context of the arrangement package. In Section 7 we highlight the performance of our methods on various benchmarks. Finally, concluding remarks and future-research suggestions are given in Section 8.

2. THE ARCHITECTURE

The `Arrangement_2<Traits,Dcel>`[1] class-template represents the planar embedding of a set of (weakly) x-monotone[2] planar curves that are pairwise disjoint in their interiors. It

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

[2] A continuous planar curve C is *weakly x-monotone*, if every vertical line intersects it at most once, or it is a vertical

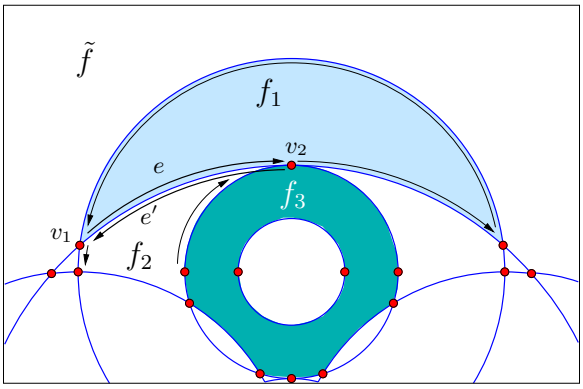


Figure 1: A portion of an arrangement of circles with some of the DCEL records that represent it. $\tilde{f}$ is the unbounded face. The halfedge e (and its twin e') correspond to a circular arc that connects the vertices v_1 and v_2 and separates the face f_1 from f_2. The predecessors and successors of e and e' are also shown. Note that e together with its predecessor and successor halfedges form a closed chain representing the inner boundary of f_1 (lightly shaded). Also note that the face f_3 (darkly shaded) has a more complicated structure, as it contains a hole.

provides the necessary capabilities for maintaining the planar graph, while associating geometric data with the vertices, edges and faces of the graph. The arrangement is represented using a *doubly-connected edge list* (DCEL) — a data structure that enables efficient maintenance of two-dimensional subdivisions.

The DCEL data-structure represents each curve using a pair of directed *halfedges*, one directed from the left endpoint of the curve to its right endpoint, and the other (its *twin* halfedge) going in the opposite direction. The DCEL consists of containers of *vertices* (associated with planar points), *halfedges* and *faces*, where halfedges are used to separate faces and to connect vertices. We store a pointer from each halfedge to the face lying to its left. Moreover, halfedges are connected in circular lists and form chains, such that all edges of a chain are incident to the same face and wind in a counterclockwise direction along its inner boundary (see Figure 1 for an illustration). A non simply-connected face stores a container of *holes*, where each hole is represented by an arbitrary halfedge on the clockwise-oriented chain that forms its outer boundary. The full details concerning the DCEL are omitted here; see [13, Section 2.2] for further details and examples.

The `Arrangement_2` class-template should be instantiated with two objects as follows. (i) A traits class, which provides the geometric functionality, and is tailored to handle a specific family of curves. It encapsulates implementation details, such as the number type used, the coordinate representation, and the geometric or algebraic computation methods. (ii) A DCEL class, which represents the underlying topological data structure, and defaults to `Arr_default_dcel`. Users may extend this default DCEL implementation, as explained in Section 3.1, or even supply their own DCEL class, written from scratch.

segment.

The two template parameters enable the separation between the topological and geometric aspects of the planar subdivision. This separation is advantageous as it allows users to employ the package with their own representation of any special family of curves, without having any expertise in computational geometry. They should only be capable of supplying the traits methods, which mainly involves algebraic computations. Indeed, several of the package users are not familiar with computational-geometry techniques and algorithms. The separation is enabled by the modular design and conveniently implemented within the generic-programming paradigm. It is a key aspect of the package, has been forced since its early stages, and heightened by the new design.

The interface of `Arrangement_2` consists of various methods that enable the traversal of the arrangement. For example, the class supplies iterators for its vertices, halfedges and faces. The value types of these iterators are `Vertex_handle`, `Halfedge_handle` and `Face_handle`, respectively. The handle classes themselves supply methods for local traversals. For example, it is possible to visit all halfedges incident to a specific vertex using its `Vertex_handle`, or to iterate over all halfedges along the boundary of a face using its `Face_handle`.

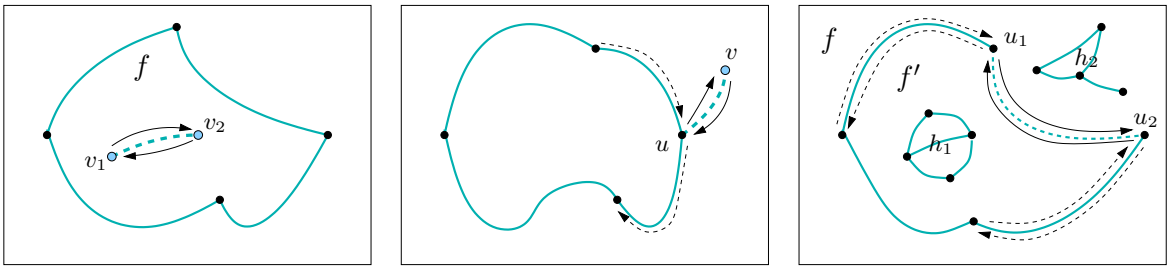
Alongside with the traversal methods, the arrangement class also supports several methods that modify the arrangement, the most important ones being the specialized insertion functions. The functions `insert_at_face_interior(C,f)`, `insert_from_left_vertex(C,u)`. (the symmetric function `insert_from_right_vertex(C,u)`,) and `insert_at_vertices(C,u1,u2)` can be used to create an edge that correspond to an x-monotone curve C whose interior is disjoint from existing edges and vertices, depending on whether the curve endpoints are associated with existing arrangement vertices; see Figure 2 for an illustration of the various cases. Note that these insertion functions hardly involve any geometric operations, if at all. They accept topologically related parameters, and use them to operate directly on the DCEL records, thus saving algebraic operations, which are especially expensive when higher-degree curves are involved. Other modification methods enable users to split an edge into two, to merge two adjacent edges, and to remove an edge from the arrangement.

An important guideline in the design is to decouple the arrangement representation from the various algorithms that operate on it. Thus, non-trivial algorithms that involve geometric operations are implemented as free (global) functions. For example, we offer a free `insert()` function for the *incremental* insertion of general curves[3] computing their *zone* (see Section 6.2), and another free `insert()` function for the *aggregated* insertion of sets of general curves, using a sweep-line algorithm. Another important operation implemented as a free function is the computation of the *overlay* of two arrangements (see [13, Chapter 2] and Section 6.1 below).

2.1 The Traits Class

As mentioned in the previous subsection, the `Arrangement_2` class-template is parameterized with a *traits* class that defines the abstract interface between the arrangement data

[3] A general curve may not necessarily be x-monotone, can intersect the existing arrangement curves, and its insertion location is unknown *a priori*.



(a) (b) (c)

Figure 2: The various insertion procedures. The inserted x-monotone curve is drawn with a light dashed line, surrounded by two solid arrows that represent the pair of twin halfedges added to the DCEL. Existing vertices are shown as black dots while new vertices are shown as light dots. Existing halfedges that are affected by the insertion operations are drawn as dashed arrows. (a) Inserting a subcurve inside the face f. (b) Inserting a subcurve whose one endpoint corresponds to the existing vertex u. (c) Inserting a subcurve whose both endpoints correspond to the existing vertices u_1 and u_2.

structure and the geometric primitives they use. The name "traits class" was given by Myers [30] for a concept of a class that should support certain predefined methods, passed as a parameter to another class template. In our case, the geometric traits-class defines the family of curves handled. Moreover, details such as the number type used to represent coordinate values, the type of coordinate system used (i.e., Cartesian or homogeneous), the algebraic methods used, and extraneous data stored with the geometric objects, if present, are all determined by the traits class and encapsulated within it.

The traits-class concept is factored into a hierarchy of refined concepts listed in the next paragraph. The refinement hierarchy is generated according to the identified minimal requirements from the traits imposed by different algorithms that operate on arrangements, thus alleviating the production of traits classes, and increasing the usability of the algorithms.

Every model of the traits-class concept must define two types of objects, namely `X_monotone_curve_2` and `Point_2`. The former represents an x-monotone curve, and the latter is the type of the endpoints of the curves, representing a point in the plane. The basic *ArrangementBasicTraits_2* concept lists the minimal set of predicates on objects of these two types sufficient to enable the operations provided by the `Arrangement_2` class-template itself, and the insertion of x-monotone curves that are also non-intersecting in their interiors. Among these predicate are the *point-status* predicate: given an x-monotone curve C and a point p, determine whether p is above, below, or lies on C; and the *compare-to-right* predicate: given two x-monotone curves C_1, C_2 that share a common left endpoint p, determine the relative position of the two curves immediately to the right of p. The set of predicates defined by the *ArrangementBasicTraits_2* concept is also sufficient for answering point-location queries by various strategies, as detailed in the previous section.[4]

The construction of an arrangement of general curves requires the refined *ArrangementTraits_2* concept. In addition to the point and x-monotone curve types, a model of the refined concept must define a third type that represents a general (not necessarily x-monotone) curve in the plane, named `Curve_2`. An intersection point of the curves is of type `Point_2`. In addition, it has to support geometric constructions, such as subdividing a given curve into simple x-monotone subcurves, computing the intersections between two given x-monotone curves, splitting an x-monotone curve into two subcurves at a given point in its interior, and merging two contiguous x-monotone portions of the same curve into a single x-monotone curve.

All traits-class operations are implemented as function objects (*functors*) according to CGAL's guidelines. This allows for the extension of the primitive types above without the need to redefine the methods that operate on them (see [24] for details on the extensible kernel). For a detailed specification of the various concept requirements see [36].

We include several traits classes with the public distribution of CGAL (see Figure 3) as follows. Traits classes for line segments[5], a traits class that operates on continuous piecewise linear curves, namely polylines [23], and a traits class that handles segments of planar algebraic curves of degree 2, namely conic arcs (e.g., ellipses, hyperbolas, or parabolas) [35].

EXACUS [3] is an ongoing project that aims to provide a set of libraries for efficient and exact algorithms for curves and surfaces. In particular, it includes CGAL-compatible traits-classes for computing arrangements of planar algebraic curves of degree 2 (conics) [10], 3 (cubics) [15] and 4 (quartics) [9]. Another traits class for conics was developed as part of an initial attempt to provide a CGAL kernel that supports curved objects [16].

[4]The only exception is the "landmarks" strategy, which requires a traits class that models the refined *ArrangementLandmarkTraits_2* concept. For lack of space, we omit the details here.

[5]The "non-caching" classes shown in Figure 3, which model the *ArrangementBasicTraits_2* and *ArrangementTraits_2* concepts respectively, directly operate on the kernel segments. Their implementation is simple, yet may lead to a cascaded representation of intersection points with exponentially long bit-length, which in turn may drastically increase the time consumption of arithmetic operations. The class `Arr_segment_traits_2` avoids this cascading problem by storing extra data with each segment. It achieves faster running times when arrangements with relatively many intersection points are constructed. However, it uses more space.

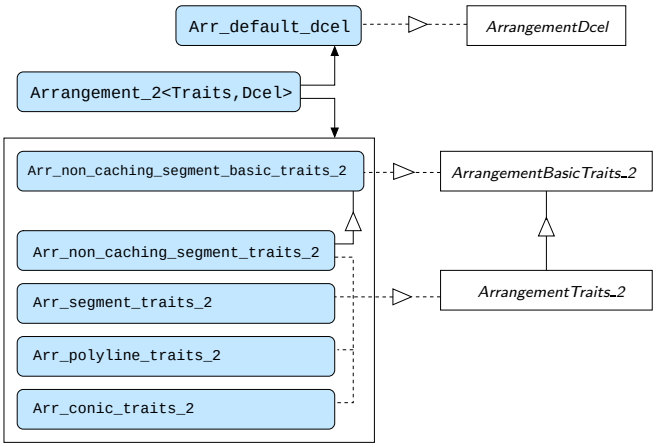


Figure 3: The main `Arrangement_2` class and its template parameters. Arrows designate pointers, solid lines directed through a triangle mark an inheritance or a refinement relation, and directed dotted lines directed through a triangle designate "is a model of" relation.

3. ADAPTERS

The ***adapter*** design-pattern "*converts the interface of a class into another interface clients expect. Adapters let classes work together that could not otherwise, because of incompatible interfaces*" (Gamma *et al.* [20]).

Adapters manifest themselves in a few places in the arrangement module, the first being a mediator between the arrangement class operations and the traits-class primitive operations. This traits adapter add geometric predicates to the traits class, based on the primitive operations provided by a model of the *ArrangementBasicTraits_2* concept. For lack of space we omit the technical details, which can be found in [19].

3.1 The DCEL Face Extender

Another application of an adapter is exhibited in the mechanism to conveniently extend the topological face-feature of the DCEL. While it is possible to store extra (non-geometric) data with the curves or points by extending their types respectively (see more details in Section 4.1), it is also possible to extend the vertex, halfedge, or face types of the DCEL through inheritance. Many times it is desired to associate extra data just with the arrangement faces. For example, when an arrangement represents the subdivision of a country into regions associated with their population density. In this case, there is no alternative other than to extend the DCEL face. As this technique is might be difficult for inexperienced users, we provide the class-template `Face_extended_dcel<FaceData>`, which extends each face in the `Arr_default_dcel` class with a `FaceData` object.

3.2 Boost Graph Adapters

The BOOST graph library (BGL; see [33]) is a generic library of graph algorithms and data structures designed in the same spirit as STL. It supports graph algorithms, and as our arrangements are embedded as planar graphs, it is only natural to extend the DCEL with the interface that the BGL expects, and gain the ability to perform the operations that the BGL supports, such as shortest-path computation. We adapt an `Arrangement_2` instance to a BOOST graph by providing a set of free functions that operate on the arrangement features and conform with the relevant BGL concepts.

We mention that besides the straightforward adaptation, which associates a vertex with each DCEL vertex and an edge with each DCEL edge, we also offer a *dual* adapter, which associates a graph vertex with each DCEL face, such that two vertices are connected, iff there is a DCEL halfedge that separates the two corresponding faces. These representations are useful for many applications, such as answering motion-planning queries (see e.g., [25]).

4. DECORATORS

The ***decorator*** design-pattern "*attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality*"(Gamma *et al.* [20]).

In traditional object-oriented programming, attaching additional functionality to an entire hierarchy of classes, all inheriting from a common (perhaps virtual) base class, referred to as the component class, requires the introduction of a decorator class that inherits from the base class and stores a pointer to a virtual component object. When applying one of the methods to the decorator, it first calls the component method, and then performs the supplementary operations. In the arrangement package we apply the decorator design-pattern when we attach auxiliary data to the geometric entities defined by a specific traits class.[6]

4.1 Meta-Traits Classes

We offer several traits-class decorators, which we refer to as *meta-traits* classes. Recall that the traits classes do not have a common base class, but they all model the *ArrangementTraits_2* concept. The meta-traits decorators are parameterized by such a traits class. They inherit some of the base-traits class functors, while overriding others exploiting the auxiliary data maintained with the geometric objects.

The `Arr_consolidated_curve_data_traits_2<BaseTraits,Data>` class inherits its `Curve_2` and `X_monotone_curve_2` types from the respective types of the base-traits class, while extending the curve with an additional *data* field, and the x-monotone curve with a container of data fields. It relies on the geometric operations supplied by the base-traits, and only needs to maintain the extra data fields. When subdividing a curve into x-monotone subcurves, its data field is copied to the resulting subcurves. Similarly, when splitting an x-monotone curve, its data container is duplicated and stored with the two resulting subcurves. When two x-monotone curves overlap, the union of their data containers is computed and stored at the resulting overlapping subcurve.

The `Arr_merged_curve_data_traits_2<BaseTraits,Data,Merge>` class operates similarly, except that it extends the `X_monotone_curve_2` type with just a single data field. When an overlap occurs, it uses the `Merge` functor, given as a template parameter, to merge the data fields of the two overlapping

[6]This is a straightforward alternative to extending the DCEL vertices and halfedges (see Section 3.1).

x-monotone curves, and stores the result with the resulting overlapping subcurve.

4.2 Arrangements with History

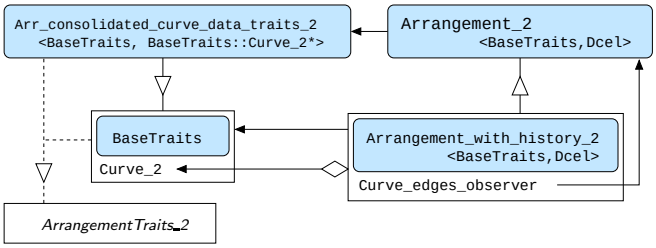


Figure 4: The `Arrangement_with_history_2` decorator. An arrow with a rhombus-shaped tail mean that a class stores a container of objects of the pointed type.

Another major component of the CGAL arrangement package is the `Arrangement_with_history_2<BaseTraits,Dcel>` class-template, which maintains a planar arrangement of general curves, while maintaining its construction history. The input curves that induce the arrangement are split into x-monotone subcurves that are pairwise disjoint in their interior. These subcurves are associated with the arrangement halfedges. In particular, each edge stores a pointer to the input curve associated with it, (or a container of pointers in case the edge is associated with an overlapping section of several curves), while each subcurve stores the set of edges it induces. Users can traverse through the origin curves of each arrangement edge, or iterate on all edges induced by a given input curve.

The `Arrangement_with_history_2` class is not more than a simple decorator for the `Arrangement_2`, as shown in Figure 4. It inherits from an arrangement class that is parameterized by the consolidated curve-data traits (see Section 4.1), where the extra data type is a pointer to a `BaseTraits::Curve_2` object. Thus, the pointers from each edge to its origin curve(s) are automatically maintained. The cross-pointers between input curves and arrangement edges are maintained using an *observer* (see the next section) that keeps track of each change that involves an arrangement edge.

Tracing back the curve (or curves) that induced an arrangement edge is essential in a variety of applications that use arrangements, such as robot motion planning (see e.g., [25]).

5. OBSERVERS

The ***observer*** design-pattern "*defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically*" (Gamma *et al.* [20]).

Observers play a significant role in the new design of the arrangement package. They serve many different needs with a single unified approach, as multiple observers can be attached to the same arrangement instance. An important set of observer classes is the one employed by some of the *point-location* strategies that maintain auxiliary data-structures (see Section 1). Another important reason for supporting observers of arrangements is to allow users to introduce their own observer classes. This is not just a convenience, but crucial to the usability of the package, as it might be the only way for providing certain output — data that should be bound with the topological features of the arrangement and is available only during construction. This is explained in Subsection 5.3. In the following subsections we give a detailed description of the notification mechanism implemented via the observer design-pattern.

5.1 The Notification Mechanism

The `Arr_observer<Arrangement>` class-template is parameterized with an arrangement class. It stores a pointer to an arrangement object, and is capable of receiving notifications just before a structural change occurs in the arrangement and immediately after such a change takes place. Hence, each notification is comprised of a pair of "before" and "after" functions. The `Arr_observer<Arrangement>` class-template serves as a base class for other observer classes and defines a set of virtual notification functions, giving them all a default empty implementation. Naturally, one of the objectives is to minimize the observer interface, that is, identifying the minimal set of event points, while capturing all possible changes that arrangements can undergo.

The set of notification functions can be divided into three categories as follows (see [36] for a detailed specification): (i) Notifiers of changes that affect the entire topological structure. Such changes occur when the arrangement is cleared or when it is assigned with the contents of another arrangement. (ii) Notifiers of a local change to the topological structure. Among these changes are the creation of a new vertex, the splitting of an edge, and the formation of a new hole inside a face. (iii) Notifiers of a global change initiated by a free function, and called by the free function (e.g., incremental and aggregate insert; see Section 2). It is required that no point-location queries (or any other queries for that matter) are issued between the calls to the "before" and "after" functions of this pair.[7]

Each arrangement object stores a list of pointers to `Arr_observer` objects, and whenever one of the structural changes listed in the first two categories above is about to take place, the arrangement object performs a *forward* traversal of this list and invokes the appropriate function of each observer. After the change has taken place the observer list is traversed in a *backward* manner (from tail to head) and the appropriate notification function is invoked for each observer. This allows for the nesting of observer objects. The observer list is not made public, and can only be accessed by the `Arr_observer` class. A free function may choose to trigger a similar notification, which falls under the third category above.

A pointer to a valid arrangement object must be supplied to the constructor of an `Arr_observer` object. The newly created observer object adds itself to the observer list of the arrangement. From that moment on, it starts receiving notifications whenever the associated arrangement object changes. In case the new observer is attached to a non-empty arrangement, its constructor may extract the relevant data from the

[7]This constraint can improve the efficiency of the maintenance of auxiliary data-structures for the relevant point-location strategies, as explained in the next subsection.

non-empty arrangement using various traversal methods offered by the public interface of the `Arrangement_2` class, and update any internal data stored in the observer.

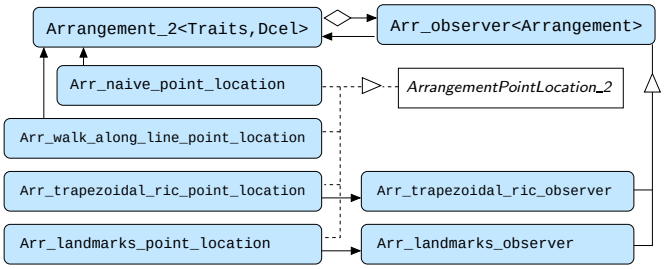


Figure 5: The point-location classes and the notification mechanism.

5.2 Point-Location Observers

As mentioned in Section 1, the *landmarks* and the *trapezoidal* point-location classes maintain auxiliary data structures. These strategies are characterized by very efficient query time but less efficient preprocessing time and space. Naturally, these strategies exhibit better overall performance when the number of updates to the arrangement is relatively small compared to the number of issued queries. Nevertheless, when the arrangement is modified the classes that implement these point-location strategies must keep their auxiliary data structure synchronized with their attached arrangement-instance.

To this end, the landmarks point-location class and the trapezoidal point-location class define the nested observer classes that inherit from `Arr_observer`, and are used to receive notifications whenever the arrangement is modified (see Figure 3). For example, a variant of the landmarks strategy uses the arrangement vertices as landmarks, so whenever a new vertex is created (by the insertion of a new edge or by the splitting of an existing edge), it should be inserted to the nearest-neighbor search structure maintained by the landmarks class. The usage of the notification mechanism makes it possible to associate several point-location objects with the same arrangement simultaneously.

5.3 User-defined Observers

In addition to the point-location observer classes, users can inherit their own observer classes from `Arr_observer` and use the notification mechanism for a variety of purposes, such as dynamically maintaining the extra data they store with the arrangement features. Assume, for example, users associate some additional data records with the arrangement faces (see Section 3.1). In this case their application needs to be notified whenever a new face is created (split from another face) or deleted (merged with another face), and receive a handle to the edge whose insertion (or deletion, respectively) causes this change. An appropriately written observer is ideal for this purpose.

6. VISITORS

The ***visitor*** design-pattern "*represents an operation to be performed on an object or on the elements of an object structure. Visitors allow the definition of new operations without changing the classes of the elements on which they operate*" (Gamma *et al.* [20]).

Arrangements have numerous applications, and different applications may require distinct and unrelated operations to be performed on arrangements. Each of these operations may treat different elements of the arrangement data-structure differently using a subset of related operations. Implementing all these operations within the arrangement class and distributing all the operation subsets across the various elements of the arrangement data structure leads to a "polluted" system that is hard to understand, use, and maintain. The BGL, for example, uses visitors [33, Section 12.3] to overcome this problem when extending its graph algorithms.

In the arrangement package we use visitors to implement geometric algorithms that are based on a common algorithmic infrastructure. We have identified two main sets of algorithms: Algorithms based on the sweep-line framework and algorithms based on the zone-computation framework. Thus, we provide two class-templates, namely `Sweep_line_2` and `Arrangement_zone_2`, which implement these two fundamental algorithmic procedures common to the two families of algorithms. Specific algorithms are implemented as visitor classes that receive notifications of the events handled by the basic procedure and can construct their output structures accordingly. The main benefit we gain from this design is a centralized, reusable and easy to maintain code. Moreover, users may add their own sweep-based (or zone-based) algorithms, as the implementation of such an algorithm reduces to implementing an appropriate visitor class.

6.1 The Generic Sweep-Line Algorithm

Sweeping the plane with a line is one of the most fundamental paradigms in Computational Geometry. The famous *sweep-line* algorithm of Bentley and Ottmann [8] was originally formulated for sets of non-vertical line segments, with the "general position" assumptions that no three segments intersect at a common point and no two segments overlap. An imaginary vertical line is swept over the input set from left to right, transforming the static two-dimensional problem into a dynamic one-dimensional one. At each time during the sweep a subset of the input segments intersect this vertical line in a certain order. The order of the segments may change as the line moves along the x-axis, implying a change in the topology of the arrangement, only at a finite number of *event points*, namely intersection points of two segments and left endpoints or right endpoints of segments. The event points, namely segment endpoints and all intersection points that have already been discovered, are stored in a dynamic event queue, named the *X-structure*, in an xy-lexicographic order, while the ordered sequence of segments intersecting the imaginary vertical line is stored in a dynamic structure called the *Y-structure*. Both structures are maintained as balanced binary trees.

The `Sweep_line_2<Traits,Event,Subcurve,Visitor>` class-template implements a generic sweep-line algorithm that can handle any set of arbitrary x-monotone curves [34], containing all possible kinds of degeneracies [13, Section 2.1], [28, Section 10.7], using a small set of geometric predicates and constructions involving the curves. The `Traits` parameter should be instantiated with a model of the *ArrangementTraits_2* concept (see Section 2.1). The `Visitor` parameter should be a model of the *SweepLineVisitor_2* concept, whose

functionality is explained in details next.

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

During the sweep-line process the event objects in the X-structure are sorted lexicographically and the subcurve objects are stored in the Y-structure. The `Sweep_line_2` class performs only the very basic operations of maintaining the X-structure and the Y-structure, while the visitor class is responsible for producing the actual output of the algorithm. Whenever the sweep-line class handles an event, it sends a notification to its visitor, with the relevant `Event` object and the `Subcurve` objects incident to it.[8] This way the sweep-line visitor is capable of attaching auxiliary data members and adding functionality to the event and subcurve objects. It can also construct the output accordingly.

It should be mentioned that Bartuschka *et al.* [7] made an initial attempt to provide a generic sweep-line algorithm in the LEDA library. They offer a class that couples a sweep-traits class with a visitor. However, in their implementation the traits class is responsible for performing the entire sweep-line algorithm, whereas our class performs the sweep-line process by itself, and only requires a traits class that supplies a small set of geometric primitives.

A simple sweep-line visitor class is used for reporting all intersection points induced by a set of input curves.[9] This visitor does not require storing any auxiliary data structures with events or with subcurves. The default `Event_base` and `Subcurve_base` types are used to instantiate the sweep-line template. The visitor simply reports an event point p, if it has more than a single incident subcurve.

As mentioned above, a key operation implemented with the aid of a sweep-line visitor is the construction of a DCEL that corresponds to the arrangement of a set of input curves. The visitor class in this case is more complicated, as it needs to store extra data with the subcurves and the events as follows. The event class is extended by a handle of a DCEL vertex that corresponds to the event point. As long as the vertex has not been created yet, the handle is invalid. The subcurve class is extended by a pointer to an event-object point that corresponds to the left endpoint of the subcurve. When processing an event point p, it is possible to go over all subcurves such that p is their right endpoint (so they lie to the left of p) and use this auxiliary data to insert the subcurves into the arrangement using one of the specialized insertion methods (see Section 2). In fact, additional information, stored with each subcurve, helps performing the insertion in the most efficient manner, utilizing all available geometric and topological information. For lack of space, we omit the related technical details here.

Another operation closely related to the construction of a DCEL structure from scratch is the aggregated insertion of new curves into an *existing* arrangement and efficiently updating an existing DCEL structure. In this case we have to sweep over a consolidated set of curves comprised of all subcurves associated with existing DCEL edges, and the set of new curves $\mathcal{C}$. Our goal is to discover the intersections involving the new curves and to update the existing DCEL accordingly. We first define a meta-traits class that extends the x-monotone curve type (see [19] for details) with a pointer to a corresponding DCEL halfedge (this pointer will be null for the newly inserted curves).[10] This way we can easily identify events that involve only existing subcurves, which can be ignored, and handle only those events involving the newly inserted curves. When handling such events, we should insert new edge pairs into the DCEL, representing the subcurves of $\mathcal{C}$. In addition, if we locate an intersection between a new curve and an existing subcurve in the DCEL, we should split the corresponding edges at the intersection point to form two halfedge pairs. This operation is elementary and takes constant time.

A fundamental operation that is straightforwardly implemented using a sweep-line visitor is the overlay of two arrangements, given as a "blue" DCEL and a "red" DCEL. The major added difficulty over the previously mentioned visitors is the need to update face structure and face information. Let us assume that each of the input-arrangement faces is associated with some data object (see Section 3.1). If we put our arrangements one on top of the other, we get an arrangement, whose faces correspond to overlapping regions of the blue and red faces. We would like to construct an output DCEL whose faces are associated with the corresponding pairs of blue and red data objects. We do so by sweeping through a consolidated set of "blue" and "red" subcurves. As explained above, it is convenient to use a meta-traits class that extends the x-monotone curves with a color identifier (`BLUE` or `RED` in our case) and a halfedge pointer. This way we can ignore "monochromatic" intersections and compute only the red–blue intersection points (or overlaps). The overlay visitor is parameterized by an overlay-traits class, which defines the merge operations between "red" and "blue" DCEL features.

6.2 Zone-Computation Visitors

Many applications can make use of the following operation: Given an arrangement $\mathcal{A}$ and an x-monotone curve C, compute the *zone* of C in $\mathcal{A}$. That is, identify all arrangement cells that the curve crosses. The zone can be computed by locating the left endpoint of C in the arrangement and then

[8] The visitor accepts two iterators defining the range of incident subcurves in the Y-structure, so it may also access the neighboring subcurves from above and below.

[9] This operation is indirectly related to arrangements, as it is implemented using the sweep-line framework.

[10] It is also possible for the visitor to extend the `Subcurve` type, but if we attach the auxiliary data at the traits-class level we can benefit from giving more efficient implementations of some traits-class functions. For example, we do not have to compute intersections between two existing DCEL subcurves.

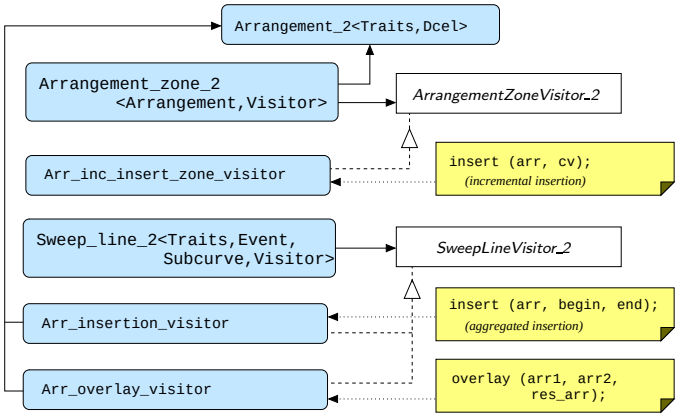


Figure 6: The free functions that are implemented with the aid of visitor classes.

"walking" along the curve to the right endpoint, keeping track of the vertices, edges and faces crossed on the way (see for example [13, Section 8.3] for the computation of the zone of a line in an arrangement of lines).

The primary usage for the zone-computation algorithm is the incremental insertion of an x-monotone curve into the arrangement. However, it is sometimes necessary to compute the zone of a curve in an arrangement without actually inserting it. In other cases, the entire zone is not required: Suppose we wish to check whether a given curve passes through an existing arrangement vertex. If such a vertex exists, the process can be terminated as soon as the vertex is located.

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

The `Arrangement_zone_2<Arrangement,Visitor>` class-template implements a generic zone-computation algorithm. It is parameterized by an arrangement class and by a visitor class. Given a curve C, the zone visitor is notified whenever a maximal subcurve $\hat{C}$ of C is found. The interior of every reported subcurve does not coincide with any arrangement vertex or edge and lies within a face f. The arrangement features that define the subcurve endpoints are also reported. A similar notification is issued whenever a subcurve $\hat{C}$ that overlaps an arrangement edge is detected. In both cases, the visitor returns a pair comprised of a halfedge handle and a Boolean flag as a result. In case the visitor inserts the subcurve $\hat{C}$ into the arrangement, it returns a handle to the newly created edge. Otherwise, it returns an invalid handle. The Boolean value indicates whether the zone-computation process should terminate — this is convenient for gaining efficiency in some applications.

The visitor class `Arr_inc_insert_zone_visitor_2` performs the incremental insertion of an x-monotone curve. It implements the two functions described above to insert the generated subcurves by splitting the halfedges intersected by the curve and using the specialized insertion functions. Other zone visitors are even easier to implement.

7. EXPERIMENTS

A user of the package has to select the appropriate component in many categories (e.g., number type, geometric kernel, traits class, end point-location strategy). For each selection the user is offered many options. The use of generic programming enables this flexibility. However, it induces a vast number of configurations that must be tested, verified, and tuned. We have developed a benchmarking toolkit that automatically generates all the required configurations and measures the performance of each configuration on a set of inputs. Naturally, we had to restrict ourselves and publish just the most efficient configurations for each traits class. Table 7 indicates the time (in seconds) it took to construct arrangements of various curve types *using exact computations*. For each traits class we have an input file containing many degeneracies (denoted *Degn.*) and a randomly generated input file (denoted *Rand.*). The results, produced by experiments conducted on a Pentium 1.8 GHz, clearly show the major improvement in performance that the package has undergone from the last public release of CGAL (version 3.1) to the current internal release (version 3.2).

Table 1: Time consumption in seconds of the construction of arrangements of various curve types. The number of input curves and the dimensions of the resulting arrangements are also shown.

Name	C	V	E	F	3.1	3.2
Segments						
Degn.	104	1504	2704	1202	0.170	0.083
Rand.	100	1129	1958	831	0.160	0.041
Polylines						
Degn.	10	112	204	94	0.081	0.020
Rand.	10	1508	2923	1417	0.769	0.223
Conics						
Degn.	41	507	1042	537	2.970	0.647
Rand.	30	677	1303	628	118.0	18.2

The reimplemented package is at least twice as efficient as the old version (CGAL 3.1) in all cases, and as much as six times more efficient in some cases. The main contribution to the improvement is due to the reduction in the number of calls to geometric operations (provided by the traits class). The effect of this reduction increases with the increase in time consumption of the geometric operation. Thus, construction of arrangements of conic arcs exhibits the largest improvement. Figure 7 shows the arrangement of the CGAL logo. It consists of 34 circles and 425 line segments. It took 1.14 seconds to construct the arrangement on the 1.8 GHz Pentium PC using the aggregate insertion method.

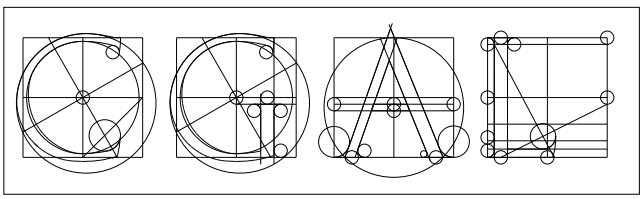


Figure 7: The arrangement of the CGAL logo.

8. CONCLUSIONS

We show how our arrangement package can be used with various components and different underlying algorithms that can be plugged in using the appropriate traits classes. Users may select the configuration that is most suitable for their application from the variety offered in CGAL or in its accompanying software libraries, or implement their own traits class. Switching between different traits classes typically involves just a minor change of a few lines of code.

We have shown how careful software design based on the generic-programming paradigm makes it easier to adapt existing traits classes or even to develop new ones. We believe that similar techniques can be employed in other software packages from other disciplines as well.

9. REFERENCES

[1] The CGAL project homepage. http://www.cgal.org/.

[2] The CORE library homepage. http://www.cs.nyu.edu/exact/core/.

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

[4] The GNU MP bignum library. http://www.swox.com/gmp/.

[5] The LEDA homepage. http://www.algorithmic-solutions.com/enleda.htm.

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

[7] U. Bartuschka, S. Näher, and M. Seel. A generic plane sweep framework, 2000. http://www.mpi-inf.mpg.de/~seel/Generic_sweep/index.html.

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

[9] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, and N. Wolpert. EXACUS: Efficient and exact algorithms for curves and surfaces. to appear in proc. 13th Europ. Sympos. Alg. (ESA), 2005.

[10] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In *Proc. 10th Europ. Sympos. Alg. (ESA)*, pages 174–186, 2002.

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

[12] T. Culver, J. Keyser, M. Foskey, S. Krishnan, and D. Manocha. ESOLID — a system for exact boundary evaluation. *Computer-Aided Design*, 36, 2003.

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

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

[15] A. Eigenwillig, E. S. L. Kettner, and N. Wolpert. Complete, exact and efficient computations with cubic curves. In *Proc. 20th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 409–418, 2004.

[16] I. Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, and E. P. Tsigaridas. Towards an open curved kernel. In *Proc. 20th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 438–446, 2004.

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

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

[19] E. Fogel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proc. 12th Europ. Sympos. Alg. (ESA)*, pages 664–676, 2004.

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

[21] B. Gerkey. Visibility-based pursuit-evasion for searchers with limited field of view. Presented in the 2nd CGAL User Workshop (2004).

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

[23] I. Hanniel and D. Halperin. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *Proc. 4th Workshop Alg. Eng. (WAE)*, pages 171–182, 2000.

[24] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proc. 5th Workshop Alg. Eng. (WAE)*, pages 79–90, 2001.

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

[26] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th Europ. Sympos. Alg. (ESA)*, pages 702–713, 2004.

[27] 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. (SCG)*, pages 360–369, 1999.

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

[29] K. Mulmuley. A fast planar partition algorithm, I. *Journal of Symbolic Computation*, 10(3-4):253–280, 1990.

[30] N. Myers. Traits: A new and useful template technique. *C++ Gems*, 17, 1995.

[31] V. Rogol. Maximizing the area of an axially-symmetric polygon inscribed by a simple polygon. Master's thesis, Technion, Haifa, Israel, 2003.

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

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

[34] J. Snoeyink and J. Hershberger. Sweeping arrangements of curves. In *Proc. 5th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 354–363, 1989.

[35] R. Wein. High-level filtering for arrangements of conic arcs. In *Proc. 10th Europ. Sympos. Alg. (ESA)*, pages 884–895, 2002.

[36] R. Wein and E. Fogel. The new design of CGAL's arrangement package. Technical report, Tel-Aviv University, 2005. http://www.cs.tau.ac.il/ efif/applications/Arr_new_design.pdf.

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