Available online at www.sciencedirect.com
ScienceDirect
Computational Geometry 38 (2007) 37–63
Computational Geometry
Theory and Applications
www.elsevier.com/locate/comgeo

Advanced programming techniques applied to CGAL's arrangement package ☆

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

School of Computer Science, Tel-Aviv University, Israel

Received 8 February 2006; received in revised form 24 August 2006; accepted 15 November 2006

Available online 4 April 2007

Communicated by B. Gärtner and R. Veltkamp

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. It is more efficient space- and time-wise, and 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. These tasks are exemplified by several applications, whose robust implementation is based on the arrangement package. Together with a set of benchmarks they assured the successful application of the various programming techniques.

Keywords: CGAL; Arrangements; Generic programming; Design patterns; Exact computation; Robustness

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 of dimensions 0 (*vertices*), 1 (*edges*), or 2 (*faces*). The *planar map* of $\mathcal{A}(\mathcal{C})$ is the embedding of the arrangement as a planar graph, such that each arrangement vertex corresponds to a planar point, and each edge corresponds to a planar subcurve of one of the curves in $\mathcal{C}$, whose interior is disjoint from all other subcurves. Arrangements and planar maps are ubiquitous in computational geometry, and have numerous applications

☆ This work has been supported in part by the IST Programme of the EU as Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS—Algorithms for Complex Shapes), by the Israel Science Foundation (grant no. 236/06), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.
* Corresponding author.
E-mail addresses: wein@tau.ac.il (R. Wein), efif@tau.ac.il (E. Fogel), baruchzu@tau.ac.il (B. Zukerman), danha@tau.ac.il (D. Halperin).

0925-7721/$ – see front matter
doi:10.1016/j.comgeo.2006.11.007

[12,23]. Hence, many potential users in academia and in industry may benefit from a generic implementation of a complete software package that constructs and maintains planar arrangements.

CGAL, the Computational Geometry Algorithms Library,[1] is the product of a collaborative effort of several sites in Europe and Israel, aiming to provide a generic and robust, yet efficient, implementation of widely used geometric data structures and algorithms. The library consists of a geometric *kernel* [17,27], which in turn consists of types of constant-size non-modifiable geometric primitive objects (such as points, line segments, triangles, etc.) and predicates and operations on objects of these types. On top of the kernel layer, the library consists of a collection of modules, which provide implementation 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 [3], making extensive use of C++ class-templates and function-templates. The generic-programming paradigm uses a formal hierarchy of abstract requirements on data types referred to as *concepts*, and a set of components that conform precisely to the specified requirements, referred to as *models*. Concepts correspond to template parameters, and models correspond to classes used to instantiate them.

In software engineering, *design patterns* are frequently used to supply standard solutions to common problems recurring in software design. Design patterns supply a systematic high-level approach that focuses on the relations between classes and objects, rather than designing individual components tailored for a specific programming task. See the book by Gamma et al. [22] 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 implementations of the point-location algorithms bundled with 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 the implementation of several point-location algorithms, and 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, letting the algorithm in use vary according to the client choice.

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

In the rest of the paper we provide a full overview of the CGAL arrangement package. We describe the main classes it comprises and show how additional design patterns are exploited in the 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 many measurable aspects, as shown in this paper through various examples and experiments.

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 input (e.g., three curves intersecting at a common point) is precluded. Secondly, operations on real numbers yield accurate results (the "real RAM" model [39], 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

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

an infinite loop, or just crash, while running on a degenerate, or nearly degenerate, input (see [31,41] for examples). This is one of the problems addressed successfully by CGAL in general and by the CGAL arrangement package in particular.

Advances in computer algebra enabled the development of efficient software libraries that offer exact arithmetic manipulations on unbounded integers, rational numbers (GMP—Gnu's multi-precision library[2]), and even algebraic numbers (the CORE library[3] [29] and the numerical facilities of LEDA[4] [34, Chapter 4]). These *exact-number types* serve as fundamental building-blocks in the robust implementation of many geometric algorithms; see [48] for a review.

Both closely related MAPC[5] [33] and ESOLID[6] [32] libraries consist of an arrangement-construction module for algebraic curves. However, these implementations make some general-position assumptions on the input curves. The LEDA library [34] includes geometric facilities that allow the robust construction and maintenance of planar maps of line segments that may contain degeneracies. However, the resulting planar maps are represented as simple graphs that cannot fully represent the topological structure of the arrangement. For example, it is impossible to encode the containment relations between disconnected components of the graph (i.e., to keep track of the holes contained in a face; see Section 2.1). LEDA-based implementation of arrangements of conic curves and of cubic curves were developed under the EXACUS project[7] [6].

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 this package can be found in [19,25]. Many users have employed the arrangement package to develop a variety of applications; see, for example, [11,14,40]. We have also used it ourselves in various applications [2,20,28,47].

In this paper we show how concurrent applications of advanced programming techniques improve the quality of the CGAL arrangement software-package, achieving a software package designed according to the generic-programming paradigm that is more modular and easy to use than previous versions, and at the same time an implementation that is more extensible, adaptable, and efficient. We remark that in the ensuing sections we assume some familiarity of the reader with advanced programming paradigms and techniques such as generic programming [3], object-oriented programming, and design patterns [22].

1.2. Outline

The rest of this paper is organized as follows: Section 2 describes the interface of the main components of the CGAL arrangement package, introducing key terms and presenting the architecture of the package. Special attention is devoted to the *traits* concept, one of the central components of the package.

In the four succeeding sections we describe the main features of the package that allow users to conveniently extend it for the special needs of their applications. Several examples of applications that use the various new features of the package are provided as well. In Section 3 we explain how an arrangement instance can be interpreted as a graph so we can apply the graph algorithms offered by the BOOST library[8] [42] on it. Section 4 describes the notification mechanism that allows external classes to be notified on changes in the structure of a particular arrangement instance. In Section 5 we review the two major algorithmic frameworks used in the arrangement package: the sweep-line framework and the zone framework, and explain how they are extended and used in the package. Section 6 describes how users can associate auxiliary information to the curves that induce the arrangement. In Section 7 we highlight the performance of our methods on various benchmarks. Finally, concluding remarks and future-work directions are given in Section 8.

[2] http://www.swox.com/gmp/.
[3] http://www.cs.nyu.edu/exact/core/.
[4] http://www.algorithmic-solutions.com/enleda.htm.
[5] http://www.cs.unc.edu/~geom/MAPC/.
[6] http://www.cs.unc.edu/~geom/ESOLID/.
[7] http://www.mpi-sb.mpg.de/projects/EXACUS/.
[8] http://www.boost.org/libs/graph/doc/index.html.

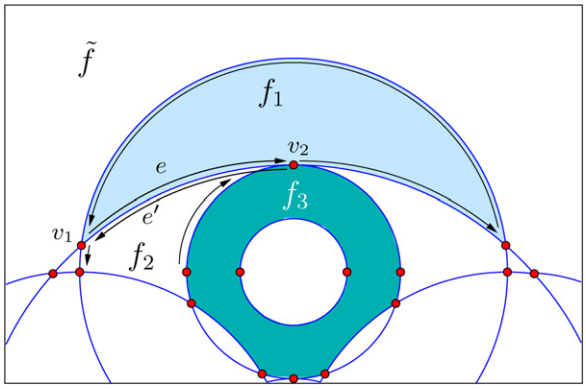


Fig. 1. A portion of an arrangement of circles with some of the DCEL records that represent it. $\tilde{f}$ is the unbounded face. The halfedge e (and its twin e') correspond to a circular arc that connects the vertices v_1 and v_2 and separates the face f_1 from f_2. The predecessors and successors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

2. The architecture

2.1. The main component

The `Arrangement_2` class-template[9] represents the planar embedding of a set of weakly x-monotone[10] planar curves that are pairwise disjoint in their interiors. It 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.

There are several variants of the DCEL data-structure. In the structure we use, each curve is represented using a pair of directed *halfedges*, one directed from the xy-lexicographically smaller endpoint of the curve to its larger endpoint, and the other (its *twin* halfedge) going in the opposite direction. The DCEL structure consists of containers of *vertices* (associated with planar points), *halfedges*, and *faces*, where halfedges are used to separate faces and to connect vertices. We store a pointer from each halfedge to the face lying to its left. Moreover, halfedges are connected in circular lists and form chains, such that all edges of a chain are incident to the same face and wind in a counterclockwise direction along its outer boundary (see Fig. 1 for an illustration). A non simply-connected face stores a non-empty container of *holes*, where each hole is represented by an arbitrary halfedge on the clockwise-oriented chain that forms its boundary. The full details concerning the DCEL structure are omitted here; see [12, Section 2.2] for further details and examples. We also extend the DCEL data-structure, allowing *isolated vertices* to be located in the interior of a face.[11]

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

- A traits class, which provides the geometric functionality, and is tailored to handle a specific family of curves. It encapsulates implementation details, such as the number type used, the coordinate representation, and the geometric or algebraic computation methods; see Section 2.2 for more details.
- A DCEL class, which represents the underlying topological data structure, and defaults to `Arr_default_dcel<Traits>`. It associates a point with each DCEL vertex and an x-monotone curve with each halfedge pair, where the geometric types of the point and the x-monotone curve are defined by the traits class. Users may

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

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

[11] Unlike other CGAL packages, the arrangement package does not base its DCEL representation on the *halfedge data-structure* [30] as this structure does not currently support holes and isolated vertices.

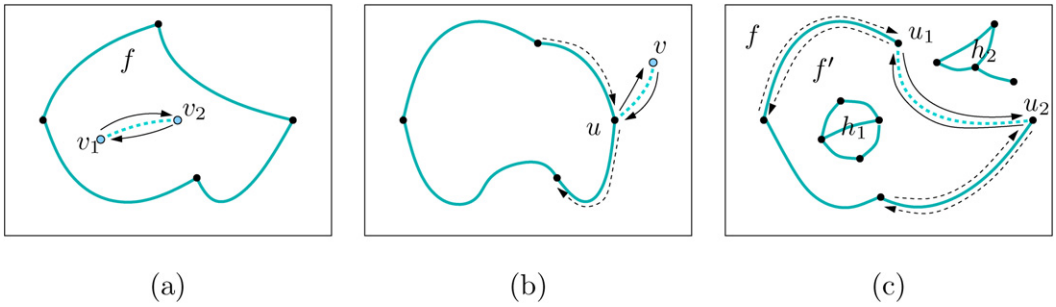


Fig. 2. The various insertion procedures. The inserted x-monotone curve is drawn with a light dashed line, surrounded by two solid arrows that represent the twin halfedges added to the DCEL. Existing vertices are shown as black dots while new vertices are shown as light dots. Existing halfedges that are affected by the insertion operations are drawn as dashed arrows. (a) Inserting a subcurve into the interior of face f, which becomes a hole of this face. (b) Inserting a subcurve, one endpoint of which corresponds to the existing vertex u. (c) Inserting a subcurve, both endpoints of which correspond to the existing vertices u_1 and u_2. In this case, the new pair of halfedges close a new face f', where the hole h_1, which used to belong to f, now becomes an enclave in this new face.

extend the default DCEL implementation, in order to attach additional data to the DCEL records, as explained in Section 2.3, or even supply their own DCEL class written from scratch.

The two template parameters enable the separation between the topological and geometric aspects of the planar subdivision. This separation is advantageous, as it allows users with limited expertise in computational geometry to employ the package with their own representation of any special family of curves. They must however supply the relevant traits-class types and methods, which mainly involve algebraic computation. The separation is enabled by the modular design and conveniently implemented within the generic-programming paradigm. It is a key aspect of the package, as well as of other central CGAL components, such as the various triangulation packages [9] and convex-hull algorithms (see [1] for more details), 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 arrangement features. For example, the class supplies iterators over its vertices, halfedges, or faces. The classes `Vertex`, `Halfedge`, and `Face`, nested in the `Arrangement_2` class, supply in turn methods for local traversals. For example, it is possible to visit all halfedges incident to a specific vertex, or traverse all the halfedges along the outer boundary of a given face.

Alongside with the traversal methods, the arrangement class also supports several methods that modify the arrangement, the most important ones being the special insertion functions. The functions `insert_in_face_interior(C,f)`, `insert_from_left_vertex(C,u)` (with a symmetric function named `insert_from_right_vertex(C,u)`) and `insert_at_vertices(C,u1,u2)` create an edge that corresponds to an x-monotone curve C, whose interior is disjoint from existing edges and vertices. The choice of which one to use depends on whether the curve endpoints are associated with existing non-isolated arrangement vertices: (i) If both curve endpoints do not correspond to any existing vertex, `insert_in_face_interior()` is used to generate a new hole inside an existing face (see Fig. 2(a)). (ii) If exactly one endpoint corresponds to an existing DCEL vertex, one of the functions `insert_from_left_vertex()` or `insert_from_right_vertex()` is called, depending on which endpoint is associated with an existing vertex. It forms an "antenna" emanating from an existing connected component (see Fig. 2(b)). (iii) Otherwise, both endpoints correspond to existing vertices, and `insert_at_vertices()` is called to connect these vertices using a pair of twin halfedges (see Fig. 2(c)). These functions hardly involve any geometric operations, if at all.[12] They accept topologically related parameters, and use them to operate directly on the DCEL records, thus avoiding algebraic operations, which are especially expensive when high-degree curves are involved.

Other modification methods included in the arrangement class enable users to split an edge into two, to merge two edges incident to a common vertex, and to remove an edge from the arrangement. It is also possible to insert a point in the interior of a given face, creating an isolated vertex that corresponds to this point, or to remove an isolated vertex from the arrangement.

[12] Unless, of course, we force checking preconditions. In this case the precondition evaluation involves geometric computation.

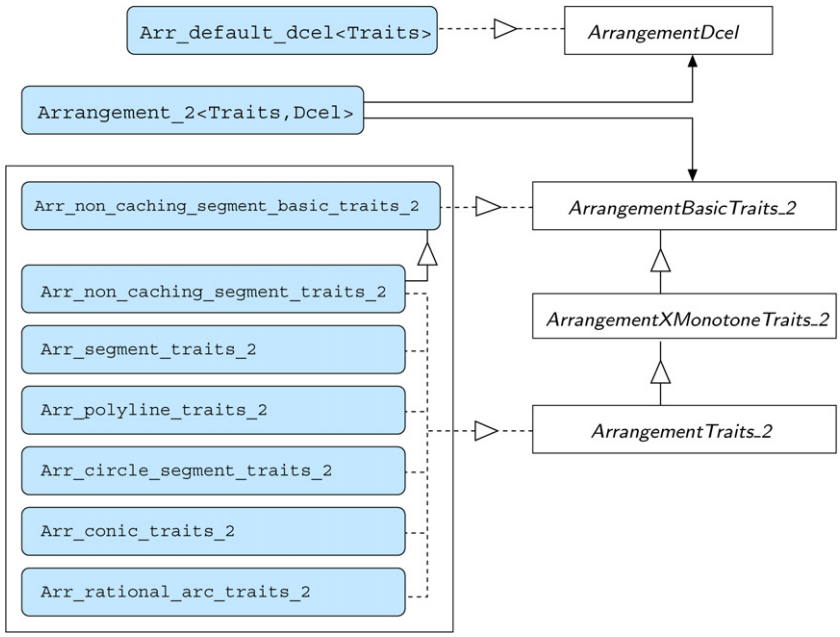


Fig. 3. The main `Arrangement_2` class and its template parameters, including some of the traits classes provided by the package. A plain arrow designates a reference to a concept associated with a template parameter, solid lines directed through a triangle mark an inheritance or a refinement relation, and dashed lines directed through a triangle designates "is a model of" relation.

An important guideline in the design is to decouple the arrangement representation from the various algorithms that operate on it. Thus, the `Arrangement_2` class provides only a restricted set of methods for locally modifying the arrangement. Non-trivial algorithms that involve geometric operations are implemented as free (global) functions that make use of the interface of the arrangement class. This way we keep the fundamental arrangement class more lightweight and allow for greater modularity in the design and implementation of algorithms that operate on planar arrangements.

For example, the package offers a free function named `insert_curve()` that inserts a general curve into the arrangement. This curve may not necessarily be x-monotone, can intersect the existing arrangement curves, and its insertion location is unknown *a priori*. The general idea is to subdivide the input curve into several x-monotone subcurves, then to treat each subcurve separately. We locate the arrangement feature that contains the left endpoint of each x-monotone subcurve, and then split the subcurve at its intersection points with the arrangement features; the resulting subcurves are finally inserted into the arrangement using one of the special insertion methods listed above. Note that the point-location operation is provided by an auxiliary point-location class and not by `Arrangement_2` itself. The arrangement class is also not capable of computing intersections and splitting curves—such methods are employed by the `insert_curve()` function, and should be provided by the instantiated traits class.

In the same spirit, the arrangement package offers free functions for *incremental* or *aggregated* insertion of curves. The incremental version inserts one curve at a time by computing its *zone* [12, Section 8.3], as described above; see also Section 5.3. The aggregated version inserts a set of general curves, using a *sweep-line algorithm* [12, Section 2.1]; see also Section 5.1.

Other algorithms that operate on planar arrangements, such as the computation of the *overlay* of two arrangements [12, Section 2.3] (see Section 5.2) are implemented as free functions as well.

2.2. The arrangement-traits concepts

As mentioned in the previous section, the `Arrangement_2` class-template is parameterized by a geometric *traits* class that defines the abstract interface between the arrangement data-structure and the geometric primitives it uses. The name "traits" was given by Myers [37] for a concept, a model of which supports certain predefined methods that have a common denominator. In our case, a geometric traits class defines the family of curves handled. Moreover,

details such as the number type used to represent coordinates, the type of coordinate system used (i.e., Cartesian or homogeneous), the algebraic methods used, and auxiliary data stored with the geometric objects, if present, are all determined by the traits class and are encapsulated within it.

The traits concept is factored into a hierarchy of refined concepts. The refinement hierarchy is defined according to the identified minimal requirements imposed by different algorithms that operate on arrangements, thus alleviating the production of traits classes, and increasing the usability of the algorithms.

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

(1) Compare two points by their x-coordinates only, or lexicographically, by their x and then by their y-coordinates.
(2) Return the lexicographically smaller (left), or the lexicographically larger (right), endpoint of a given x-monotone curve.
(3) Determine whether a weakly x-monotone curve is a vertical segment.
(4) Given an x-monotone curve C and a point $p = (x_0, y_0)$ such that x_0 is in the x-range of C (namely x_0 lies between the x-coordinates of C's endpoints), determine whether p is above, below, or lies on C.
(5) Given two x-monotone curves C_1 and C_2 that share a common left endpoint p, determine the relative position of the two curves immediately to the right of p. The traits class can also provide a symmetric comparison method, namely to the left of a common right endpoint. The latter is an optional requirement with ramifications in case it is not fulfilled; see Section 2.2.1.

The set of predicates listed above is also sufficient for answering point-location queries by the various point-location strategies, with the exception of the "landmarks" strategy, which requires a traits class that models the refined concept *ArrangementLandmarksTraits_2*. This is described in Section 4.2.

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

(1) Compute the intersections between two given x-monotone curves C_1 and C_2, sorted in increasing lexicographical order. Each intersection is represented by an intersection point and its *geometric multiplicity*,[13] if the multiplicity is defined and known, or by an x-monotone curve representing an overlapping portion of C_1 and C_2.
The introduction of multiplicity of intersection points enables the arrangement-construction algorithms to exploit the geometric knowledge they may have, in order to avoid costly calls to other traits-class functions, since in many cases the order of incident curves to the right of a common intersection point can be deduced from their order to the left of the point and the intersection multiplicity.[14]
(2) Split a given x-monotone curve C at a given point p, which lies in C's interior, into two subcurves.

The construction of an arrangement of *general* curves requires a model of the further refined concept *ArrangementTraits_2*. In addition to the point and x-monotone curve types, a model of the refined concept must define a third type that represents a general (not necessarily x-monotone) curve in the plane, named `Curve_2`. It

[13] See, e.g., http://en.wikipedia.org/wiki/Intersection_number for an exact definition.
[14] This is quite clear when we have two curves intersecting at a point, as they swap their relative order if and only if the multiplicity of intersection is odd. See [7] for a generalization to the case of multiple curves intersecting at a common point.

also has to supply a method that subdivides a given curve into simple x-monotone subcurves, and possibly isolated points.[15]

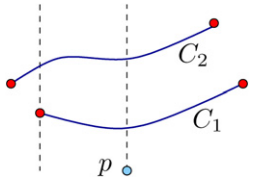
All traits-class operations are implemented as function objects (*functors*) according to CGAL's guidelines. This allows extending the geometric types above, without the need to redefine the methods that operate on them (see [27] for details on the extensible kernel). For a detailed specification of the various concept requirements see [46].

2.2.1. The traits-class adapter

In addition to the primitive operations involving curves and points listed above, the arrangement class needs to perform several other geometric operations that can be implemented by using the traits-class primitives. For example, in order to determine whether a point p is in the x-range of an x-monotone curve C, we simply have to compare p to C's endpoints, and check whether it lies to the right of the left endpoint and to the left of the right endpoint.

The generic-programming paradigm dictates that concepts should be as tight as possible. Indeed, the requirements listed by the geometric traits concepts are made minimal. They include only the utterly essential types and methods, and fully specify all the preconditions on the input data, as these may simplify the implementation of models of this concept even further. Another important reason for reaching the minimal requirements is to avoid computing the same algebraic entity in different ways. The importance of this is amplified in the context of geometry, because the presence of more than a single way can lead to artificial degenerate conditions. To this end, `Arrangement_2` and its peripheral classes use a traits-class adapter that implements additional functors based on the functors supplied by the traits class. The traits class is injected as a template parameter into the traits-class adapter, which inherits from it.

Traits concepts with minimal interface are important in our case not just for programming convenience—they also have geometric implications. In previous versions of CGAL the traits classes had to include the following predicate: given two interior disjoint x-monotone curves C_1, C_2 and a point p, whose x-coordinate lies in the x-ranges of both curves, determine the y-order of the two curves with respect to a vertical line passing through p. This predicate is required by the simple point-location algorithms (see Section 4.2) in order to locate the arrangement edge lying right above the query point. Computing this predicate is trivial for line segments, but it involves algebraic expressions of high order when handling algebraic curves of degree 2 or higher. In fact, some algebraic methods employed by the traits classes mentioned in Section 2.2.4 cannot handle such a predicate in an exact manner at all.



The traits-class adapter handles this predicate in a very simple manner. Recall that C_1 and C_2 are interior disjoint, but their x-ranges overlap, as they both contain the x-coordinate of p. Thus, we compare the left endpoints of both curves, and select the rightmost one; assume without loss of generality it is the left endpoint of C_1 (see the illustration above). Then, we simply find the position of this point with respect to C_2. If this point lies on C_2, then the two curves share a common endpoint, and we just compare them to the right of this point.

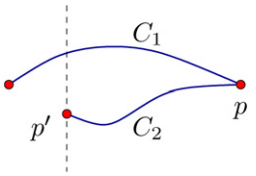
Reducing the requirements from the geometric traits-class and minimizing the number of invocations of traits-class functions has a dramatic effect on the performance of our arrangement operations, as we report in Section 7.

In some cases, the traits-class adapter uses a *tag-dispatching* mechanism to select the appropriate implementation of a traits-class functor. Tag dispatching is a technique that uses function overloading to dispatch a function *at compile time*, based on properties of the types the function accepts.[16] This mechanism enables users to implement their traits class with a reduced set of functors. The traits class must define the two tags listed below that affect the operation of

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

[16] See, e.g., http://www.boost.org/more/generic_programming.html for more details.

the adapter.



- The tag `Has_left_category` indicates whether the traits class supports the comparison of two x-monotone curves C_1 and C_2 to the *left* of a common right endpoint p. This predicate is required by some point-location strategies and by the zone-computation algorithm, and while it may sometimes be fairly easy for the traits-class implementer to provide it, it can be computed using other primitive traits-class predicates. Thus, when the *has-left* tag is false, the traits-class adapter resorts to the following (somewhat less efficient) algorithm: It locates a new reference point p' to the left of p, namely the lexicographically largest left endpoint of C_1 and C_2, and determines the relative position of the curves to its right.
- The tag `Has_merge_category` indicates whether a model of the *ArrangementXMonotoneTraits_2* supports the merge of x-monotone curves. If the tag is true, the traits class must provide the two following operations:
  (1) Determine whether two x-monotone curves C_1 and C_2 that share a common endpoint are *mergeable*—that is, whether they can be merged into a single continuous x-monotone curve representable by the traits class.
  (2) Merge two mergeable x-monotone curves C_1 and C_2 into a single continuous x-monotone curve.

  The merger operation is used to eliminate redundant features in the arrangement. For example, if we have a T-shaped structure formed by two line segments, and the vertical segment forming the "leg" is removed, then it is possible to merge the two horizontal sub-segments. When the *has-merge* tag is false, the adapter simply declares any pair of curves as non-mergeable. The only effect on the arrangement is that we cannot remove redundant vertices (of degree two) following the deletion of edges.

2.2.2. Segment-traits classes

The arrangement package provides two traits classes that handle line segments. The `Arr_segment_traits_2<Kernel>` class-template is parameterized by a geometric *kernel*, that conforms to the CGAL-kernel concept [17]. We note that the `Segment_2` type defined by most CGAL kernels is represented only by its two endpoints. When a segment is split several times, the bit-length of the coordinates needed to represent its endpoints may grow exponentially (see [21] for a discussion), which may significantly slow down the computation. Our traits class therefore represents a segment by its supporting line and its two endpoints. When the traits class computes an intersection point of two line segments, it uses the coefficients of their supporting lines. When a segment is split at an intersection point, the underlying line of the two resulting sub-segments remains the same, and only their endpoints are updated. The `Arr_segment_traits_2<Kernel>` thus overcomes the undesired effect of cascading intersection-point representation described above.

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

2.2.3. Polyline-traits classes

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

Previous releases of CGAL included a stand-alone polyline-traits class, which represented a *polyline* as a list of points, and performed all geometric operations on this list [24]. The new arrangement package introduces the `Arr_polyline_traits_2<SegmentTraits>` class-template, which must be instantiated with a geometric traits class that is able to handle line segments. A polyline curve is represented as a vector of `Segment_Traits::X_monotone_curve_2` objects (namely segments). The new polyline-traits class does not perform any geometric operations directly. Instead, it solely relies on the functionality of the instantiated segment-traits class. For example, when we need to determine the position of a point with respect to an x-monotone polyline, we use binary search to locate the relevant segment that contains the point in its x-range, then we compute the position of the point with respect to this segment. Thus, operations on x-monotone polylines of size m typically take $O(\log m)$ time.

Users are free to choose the underlying segment-traits class based on the number of expected intersection points (see discussion above in Section 2.2.2). Moreover, it is possible to instantiate the polyline-traits class-template with a traits class that handles segments with some additional data attached to them (see Section 6.1). This makes it possible to associate different data objects with the different segments that comprise a polyline.

2.2.4. Traits classes for non-linear curves

The arrangement package includes several traits classes that handle non-linear curves: a traits class for circular arcs, another for general conic arcs (bounded segments of algebraic curves of degree 2), and a traits class for arcs of graphs of rational functions. As these traits classes handle more complex objects, they need to employ more sophisticated algebraic methods in order to ensure the exactness and robustness of the computations they carry out; see, e.g., [44] for more details.

Additional traits classes that are also compatible with the arrangement-traits concept have been developed by other groups of researchers. Among these we can list traits classes for circular arcs and for conic arcs developed by Emiris et al. [16], extending the predicates described by Devillers et al. [13], and traits classes for conic curves [7], cubic curves [15], and special types of quartic curves [8] that were developed as part of the EXACUS project [6]. The work of [16] is available in CGAL Version 3.2, under the circular kernel package. The EXACUS-based traits classes are planned to be integrated into future versions of CGAL, as the basis for a curved kernel that handles algebraic curves of arbitrary degree.

2.3. Extending the DCEL

As mentioned in Section 2.1, the `Arrangement_2` is parameterized by a DCEL class, which is by default `Arr_default_dcel<Traits>`. This default DCEL model simply associates a point with each DCEL vertex and an x-monotone curve with each halfedge pair. However, it is sometimes necessary to extend the topological features of the DCEL. While it is possible to store auxiliary data with the curves or points by extending their respective types (see more details in Section 6.1), it is also possible to extend the vertex, halfedge, or face types of the DCEL through a mechanism based on inheritance. Many times it is desired to associate extra data with the arrangement faces only. 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 there is no concrete geometric entity that corresponds to an arrangement face.

We note that similar mechanisms of extending geometric features with auxiliary attributes can be found in other components in CGAL, such as the triangulation packages [9] and the halfedge data-structure [30]. However, as the feature-extension technique is somewhat cumbersome and might be difficult for inexperienced users, the arrangement package provides two DCEL adapters that give it a more intuitive interface. The class-template `Arr_face_extended_dcel<Traits,FaceData>` extends each face in the `Arr_default_dcel` class with a `FaceData` object, and provides interface for accessing and modifying this object. Similarly, the package also contains a more general adapter named `Arr_extended_dcel<Traits,VData,HData,FData>` that extends the DCEL vertex, halfedge, and face records with fields of types `VData`, `HData` and `FData`, respectively.

3. Adapting to BOOST graphs

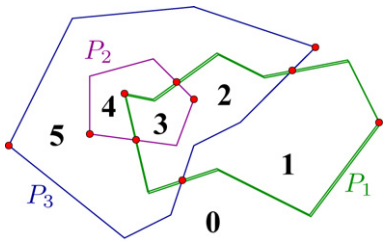
The BOOST graph library (BGL) [42] is a generic library of graph algorithms and data structures designed in the same spirit as the STL [3]. It supports graph algorithms, and as our arrangements are embedded as planar graphs, it is

only natural to augment the DCEL with the interface that the BGL expects, and gain the ability to perform the operations that the BGL supports, such as shortest-path computations. We adapt `Arrangement_2` instances to BOOST graphs by specializing the `boost::graph_traits` template for the `Arrangement_2` class and providing a set of free functions for traversing the arrangement features that conform with the interface prescribed by the BGL.

In addition to the straightforward adaptation, which associates a vertex with each DCEL vertex and an edge with each DCEL halfedge, we also offer a *dual* adapter, which associates a graph vertex with each DCEL face, such that two vertices are connected, if and only if there is a DCEL halfedge that separates the two corresponding faces. Using this dual adapter it is possible, for example, to perform breadth-first or depth-first traversals on arrangement faces.

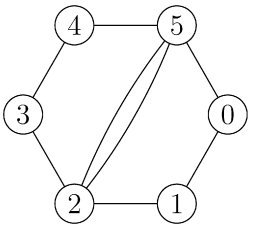
This dual representation is very useful for many applications, such as answering motion-planning queries (see, e.g., [28]). Assume that we have an arrangement of line segments and circular arcs that represents the configuration space of a disk robot moving amidst polygonal obstacles in the plane. We extend the DCEL by attaching a Boolean flag to each arrangement face, indicating whether it is *free* or *forbidden*. As the BGL enables the application of filters on graph vertices, we can perform a breadth-first traversal starting at a given free face and filter out faces that are marked as forbidden, considering only the free arrangement faces. This way we can efficiently answer motion-planning queries.

3.1. Example: Cumulative polygon operations



Assume that we are given a set $P_1, P_2, \ldots, P_n$ of simple polygons, where each polygon is represented by a closed polyline (see Section 2.2.3) with counterclockwise orientation that forms its boundary. Namely, the polyline winds in a counterclockwise direction around the interior of the polygon. Using this representation we can compute the arrangement of the boundary polylines. The distinct property of the planar subdivision represented by this arrangement is that if two points $p, q \in \mathbb{R}^2$ belong to the same arrangement face, they are covered by the same subset of polygons.

Let $N(f)$ denote the number of polygons that cover the face f. We extend the face type of the DCEL (see Section 2.3) with a counter: an unsigned integer that stores the value of $N(f)$. This representation enables the computation of several cumulative set operations on our input polygons. It immediately follows that the union of polygons is given by $\bigcup_{i=1}^{n} P_i = \{f \mid N(f) > 0\}$, their intersection is simply $\bigcap_{i=1}^{n} P_i = \{f \mid N(f) = n\}$, and their cumulative symmetric difference is defined as $\bigoplus_{i=1}^{n} P_i = \{f \mid N(f) \text{ is odd}\}$.[17]



[17] For simplicity, we describe here *regularized* set operations. The regularized result is obtained by taking the closure of the interior of the ordinary result, thus eliminating its low-dimensional features (i.e., "antennas" and isolated vertices). We note however that using an appropriately extended DCEL class, it is not difficult to compute the low-dimensional features of the result as well; see [18, Appendix A] for the details.

When we know all input polygons in advance, it is possible to construct the arrangement of their boundary polylines using the aggregated insertion function (see Section 5.1 for more details). Once the arrangement is constructed, it is possible to compute $N(f)$ for each face applying a variant of the breadth-first search (BFS) algorithm on the dual-graph representation of our arrangement (as described above), starting from the graph vertex that corresponds to the unbounded face f_0. We maintain the distance variable, and each time we visit a face, we update its counter accordingly.[18] As mentioned above, extracting the regularized result of a cumulative set operation on the polygons set is trivial given the extended face information $N(f)$.

4. Notifying external classes on changes

Some arrangement-based algorithms and applications should be bound to a specific arrangement instance and receive notifications on various topological changes this arrangement undergoes. This is not just a convenience, but crucial to the usability of the package, as it might be the only way for providing the external algorithm with a certain input, such as data that should be associated with the topological features of the arrangement, and is available only during construction; see Section 4.3 for an example.

In previous versions of the arrangement package [19] it was possible to provide the various insertion functions with a pointer to a *change-notification* class. If the pointer pointed to an operative notification object, the arrangement would invoke notification functions implemented by this class—for example, it would call `on_split_face()` when a face was split into two. The main disadvantage of this approach was that only a single change-notification class was supported. Indeed, this class might have informed several other objects on the changes, but this would have reduced the reusability of such notification classes. In addition, the trapezoidal RIC point-location strategy (see more details in Section 4.2) used a specialized notification mechanism, which forced a point-location instance to be tightly coupled with an arrangement instance.

In Version 3.2 we introduce a new notification mechanism that is based on arrangement 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. [22]). Using this new mechanism it is possible to attach any number of observer instances to a specific arrangement, such that all attached observers get notified on local and global changes the arrangement undergoes.

In the following subsections we give a detailed description of the notification mechanism, describe how it is used by different point-location strategies, and give an example of a user-defined observer.

4.1. The notification mechanism

The `Arr_observer<Arrangement>` class-template is parameterized by an arrangement type. 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 comprises of a pair of "before" and "after" functions (e.g., `before_split_face()` and `after_split_face()`). 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. The interface of the base class is designed to capture all possible changes that arrangements can undergo, with a minimal set of topological events.

The set of functions can be subdivided into three categories as follows.

(1) 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.
(2) Notifiers of a *local* change to the topological structure. Among these changes are the creation of a new vertex or an edge, the splitting of an edge or a face, the formation of a new hole inside a face, the removal of an edge, etc.

[18] Special attention should be exercised in the presence of degeneracies, such as overlapping boundary curves. For simplicity of presentation, we assume here that no such degeneracies exist. The reader is referred to [18, Appendix A] for a review of the complete solution.

(3) Notifiers of a *global* change initiated by a free (global) function, and called by the free function (e.g., incremental or aggregated insert; see Section 2). This category consists of a single pair of notifiers, neither of them is called by methods of the `Arrangement_2` class-template itself. 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.[19]

See [46] for a detailed specification of the arrangement observer class sketched above.

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 invokes the appropriate function of each of its observers. It also does so immediately after the change has taken place. In addition, a free function may choose to trigger a similar notification, which falls under the third category above.

The observer list of an arrangement object is not made public, and can only be accessed by the `Arr_observer` class. 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 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. This is necessary, for example, in case of the point-location strategies that maintain auxiliary data structures described in the next subsection.

4.2. Point-location observers

Several types of queries on arrangements are supported by the package, the most important one being the *point-location* query: given a point, find the arrangement cell that contains it. Typically, the result of the point-location query is one of the arrangement faces, but in degenerate situations the query point can lie on an edge, or it may coincide with a vertex. Since the arrangement representation is decoupled from the algorithms that operate on it, the `Arrangement_2` class does not support point-location queries directly. Instead, the package provides a set of classes that are capable of answering such queries, all are models of the concept *ArrangementPointLocation*. Each class employs a different algorithm or *strategy* for answering queries. A model of this concept must define the `locate()` function that accepts an input query point and returns an object representing the arrangement cell that contains this point (a polymorphic `CGAL::Object` instance that can either be a `Face_handle`, a `Halfedge_handle`, or a `Vertex_handle`).

The following models of the concept *ArrangementPointLocation* are included in the arrangement package. Each employs a different point-location strategy.

- `Arr_naive_point_location` locates the query point naïvely, by exhaustively scanning all arrangement cells.
- `Arr_walk_along_a_line_point_location` simulates a reverse traversal along an imaginary vertical ray emanating from the query point toward infinity. It starts from the unbounded face of the arrangement and moves downward toward the query point until it locates the arrangement cell containing it.
- `Arr_landmarks_point_location<Generator>` uses an auxiliary generator class to create a set of "landmark" points, whose location in the arrangement is known. Given a query point, it uses a nearest-neighbor search structure (e.g., KD-tree) to find the nearest landmark, and then traverses the straight-line segment connecting this landmark to the query point.[20] See [26] for more details.

[19] This constraint can improve the efficiency of the maintenance of auxiliary data structures for the relevant point-location strategies, which have to update their data structures according to the changes the arrangement undergoes (see the next section for more details). Since no point-location queries are issued between the invocation of `before_global_change()` and `after_global_change()`, it is not necessary to perform an update each time a local topological change occurs, and it is possible to postpone the updates until after the global operation is completed.

[20] The "landmarks" strategy, requires that the arrangement is instantiated with a traits class that models the *ArrangementLandmarksTraits_2* concept, which adds two requirements to the basic *ArrangementBasicTraits_2* concept: (i) Approximating the coordinates of a given point p using

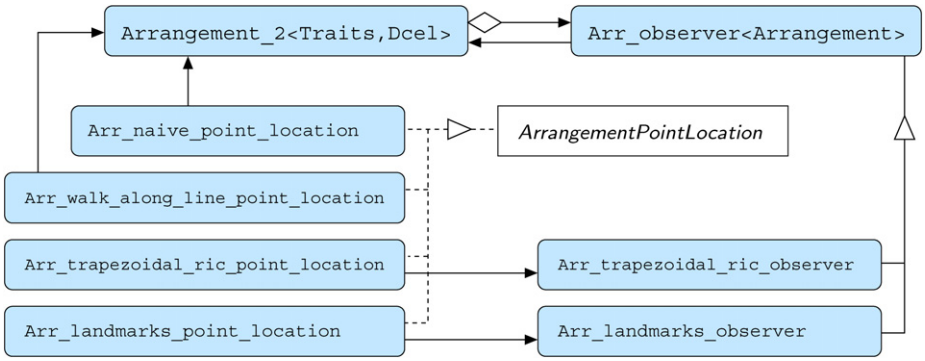


Fig. 4. The point-location classes and the notification mechanism. Solid lines directed through a triangle mark an inheritance relation, dashed lines directed through a triangle designate "is a model of" relation, a line with an arrow head or a similar line with a rhombus-shaped tail indicate that the source class stores a reference to an object of the target type or a container of objects of the target type, respectively.

- `Arr_trapezoidal_ric_point_location` implements Mulmuley's point-location algorithm [36], 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 the DCEL that represents the arrangement.

Each of the "landmarks" point-location class and the trapezoidal point-location class defines a nested observer class that inherits from `Arr_observer`, and is used to receive notifications whenever the arrangement is modified (see Fig. 4). For example, the default generator employed by 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 into the nearest-neighbor search structure maintained by the respective landmark class. The usage of the notification mechanism makes it possible to associate several point-location objects with the same arrangement simultaneously.

The "landmarks" and the trapezoidal point-location strategies are both characterized by very efficient query time at the account of time-consuming preprocessing. Naturally, these strategies exhibit better overall performance when the number of arrangement updates is relatively small compared to the number of issued queries. For a report on extensive experiments with the various point-location strategies see [26].

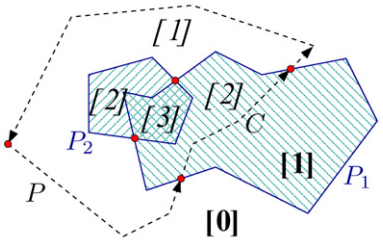
4.3. Example: Online cumulative polygon operations

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 extra data they store with the arrangement features. Let us revisit the cumulative set-operation example introduced in Section 3. However, we now assume that the polygon set is not static, namely that new input polygons arrive as time progresses. In this case we incrementally insert each new polygon into the arrangement upon arrival and efficiently update the face counters accordingly.

We use an observer that stores a face handle f_{in} and another container of face handles, denoted $\mathcal{F}_{\text{out}}$. Each time we handle a new polygon P^*, we set f_{in} to be an invalid handle and reset the container $\mathcal{F}_{\text{out}}$. As the boundary curve of the polygon is inserted into the arrangement, existing faces are split into two. Our observer is notified each time such an event occurs. It receives handles for the two new faces f_1, f_2 resulting from the split and a handle for the halfedge e that causes the split. Recall that the polygon boundary is properly oriented, such that its interior always lie to the left of its boundary curve. Thus, we can easily identify the face lying to the left of e. Assume, without loss of generality, it is f_1, and set $f_{\text{in}} \longleftarrow f_1$. (Note that f_{in} is updated each time a face-split event takes place. As we explain next, we only need a handle to *some* face that lies inside the new polygon.) The other face f_2 is not contained in the new

double-precision arithmetic; and (ii) constructing an x-monotone curve that connects two given points p and q, where p represents a landmark point and q is the query point. Most traits classes included in the arrangement package are models of this refined concept.

polygon and is inserted into $\mathcal{F}_{\text{out}}$.



When the insertion process is over, we can easily update the face counters using the information accumulated by the observer. We perform a filtered breadth-first search on the graph dual to the arrangement (Section 3), starting from f_{in}. Whenever we visit a face f, we increment its counter $N(f)$. The faces of $\mathcal{F}_{\text{out}}$ are used to filter the breadth-first traversal: as they completely surround the boundary of P^*, we are guaranteed to visit only the faces covered by this new polygon. The figure above illustrates the insertion process of a new polygon into an arrangement induced by two polygons. The final counters (after the insertion of P^*) appear in brackets, and those incremented during the traversal are slanted. In this case, the two other arrangement faces form $\mathcal{F}_{\text{out}}$.

5. Major algorithmic frameworks

We have identified two main algorithmic frameworks related to arrangements of planar curves: the sweep-line framework and the zone-computation framework. Each framework serves as the foundation of a family of concrete algorithms. For instance, the implementation of operations like aggregated insertion of a set of curves into an arrangement and the overlay computation of two arrangements is based on the sweep-line framework, while the incremental insertion of a single curve into an arrangement involves the computation of the zone of this curve.

We provide two class-templates, namely `Sweep_line_2` and `Arrangement_zone_2`, that implement these two fundamental algorithms common to the two families of concrete algorithms, respectively. Each class template is parameterized, among the other, by a visitor class—a model of the appropriate visitor concept. The concrete algorithms are realized through sweep-line visitors or through zone-computation 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. [22]). The visitor classes receive notifications of the events handled by the basic procedure and can construct their output structures accordingly. We gain a centralized, reusable, and easy to maintain code. For example, we use sweep-line visitors to obtain a variety of different results: computing all intersection points induced by a set of curves, constructing the arrangements of these curves, inserting the curves into an existing arrangement, etc. Moreover, users may introduce their own sweep-based or zone-based algorithms, as implementing such an algorithm reduces to implementing an appropriate visitor class.

There is a certain similarity between observers and visitors, as typically each of their methods is triggered as a response to a certain event—a member of a pre-determined list of events. The main difference between them is that observers define a one-to-many mapping between objects, while visitors define a one-to-one mapping. Recall that a single arrangement may register many observers, but it is only natural to relate a single visitor to a specific algorithmic framework in order to realize a certain concrete algorithm. Consequently, observers must be derived from a common base class, and their methods must be virtual. Recall that arrangement observers must be derived from `Arr_observer`. In contrast, while visitors must model certain concepts, they can, and typically are, syntactically unrelated. The requirements of the visitor concepts of the two algorithmic frameworks are described in the next sections. We mention that the BOOST Graph Library, for example, uses visitors [42, Section 12.3] to support user-defined extensions to its fundamental graph algorithms.

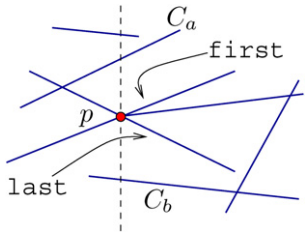
5.1. The generic sweep-line algorithm

Sweeping the plane with a line is one of the most fundamental algorithmic frameworks in computational geometry. The famous *sweep-line* algorithm of Bentley and Ottmann [5] was originally formulated for sets of non-vertical

line segments, with the "general position" assumptions that no three segments intersect at a common point and no two segments overlap. An imaginary vertical line is swept over the plane from left to right, transforming the static two-dimensional problem into a dynamic one-dimensional one. At each time during the sweep a subset of the input segments intersect this vertical line in a certain order. The subset of segments and their order along the sweep line may change as the line moves along the x-axis, 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 an xy-lexicographic order in a dynamic event queue, named the *X-structure*. 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, such as red-black trees, that enable their efficient maintenance. In particular, we use an advanced implementation of red-black trees [45] that offers extended functionality over other alternatives such as STL maps.

The `Sweep_line_2<Traits,Event,Subcurve,Visitor>` class-template implements a generic sweep-line algorithm that can handle any set of arbitrary x-monotone curves [43], containing all possible kinds of degeneracies (see [12, Section 2.1] and [34, Section 10.7] for the treatment of degeneracies induced by line segments), using a small set of geometric predicates and constructions involving the curves. The `Traits` parameter must be instantiated with a model of the *ArrangementXMonotoneTraits_2* concept (see Section 2.2). The `Visitor` parameter must be instantiated with a model of the *SweepLineVisitor_2* concept, whose functionality is explained in details next.

The `Sweep_line_2` class-template uses two auxiliary classes: `Event_base`, which stores a `Point_2` object that represents the coordinates of an event point, and `Subcurve_base`, associated with a portion of an x-monotone curve (represented as an `X_monotone_curve_2` object), whose interior is disjoint from all other subcurves at the current location of the sweep line (it may intersect yet undiscovered subcurves as the sweep line advances). These two auxiliary classes also store additional data members needed internally by the sweep-line algorithm, which are not exposed to external users. The `Sweep_line_2` parameters `Event` and `Subcurve` are instantiated with these two types by default. Users may however extend these types with data required by their visitor class by inheriting an event class and a subcurve class from the respective base classes, and using these extended classes to instantiate 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 in the same order as the lexicographic order of their intersection with the imaginary sweep-line. The `Sweep_line_2` class performs only the operations required to maintain the 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 point p, it sends a notification to its visitor, with the relevant `Event` object and the `Subcurve` objects incident to it. The latter is specified by a pair of iterators that define the subcurve range. Using this information, the visitor can access not only the subcurves incident to p, but also the neighboring subcurves from above and below. In the example depicted above, the event point p is sent to the visitor with the iterator range `[first, last]`, which defines the three subcurves that share p as a common left endpoint; the subcurves C_a and C_b lying above and below p can be accessed by dereferencing the expressions `--first` and `++last`, respectively. The sweep-line visitor is capable of attaching auxiliary data members and adding functionality to the event and subcurve objects. It can also construct its output accordingly.

It should be mentioned that Bartuschka et al. [4] designed and implemented a generic sweep-line algorithm in the LEDA library. They offer a class that couples a sweep-traits class with a visitor. However, in their implementation the traits class is responsible for performing almost the entire sweep-line algorithm, whereas our class performs the bare sweep-line procedure, and requires only a traits class that supplies a small set of geometric primitives. Hence, our approach provides a more modular framework that is easier to extend.

A simple sweep-line visitor class is used for reporting all intersection points induced by a set of input curves.[21] This visitor does not require storing any auxiliary data structures with events or with subcurves. The default `Event_base` and `Subcurve_base` types are sufficient and used to instantiate the sweep-line class-template. The visitor simply reports an event point p, if it has more than a single incident subcurve.

As mentioned above, a key operation implemented with the aid of a sweep-line visitor is the construction of a DCEL that corresponds to the arrangement induced by a set of input curves. The visitor class in this case is more complicated, as it needs to store extra data with the subcurves and the events as follows. The event class is extended by a handle for a DCEL vertex that corresponds to the event point. As long as the vertex has not been created yet, the handle is invalid. The subcurve class is extended with a pointer to an event point that corresponds to the left endpoint of the subcurve. When processing an event point p, it is possible to go over all subcurves such that p is their right endpoint (they lie to the left of p) and use this auxiliary data to insert the subcurves into the arrangement using one of the special 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. 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 the plane and account for the set $\mathcal{C}$ of new curves as well as the consolidated set of all subcurves associated with the existing DCEL halfedges. Our goal is to discover the intersections involving the new curves, and to update the existing DCEL accordingly. We first extend the x-monotone curve type defined by the traits class (see [21] for details) with a handle for one of its corresponding halfedge twins (this handle is invalid for the newly inserted curves). It is also possible to extend the `Subcurve` type of the visitor, but attaching the auxiliary data at the traits-class level enables a more efficient implementation of the traits-class methods. For example, it is possible to avoid the computation of intersections between two curves that correspond to valid halfedges. This way we can easily identify events in which only existing subcurves are involved, ignore them, and handle only those events in which newly inserted curves are involved.

5.2. Overlaying arrangements

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

The major added difficulty over the previously mentioned visitors is the need to construct a DCEL that properly represents the overlay of two potentially extended input arrangements. That is, the features of each one of the two DCEL data-structures that represent the two respective input arrangements could have been extended with additional data (see Section 2.3). If we put our arrangements one on top of the other, we get an arrangement whose faces correspond to overlapping regions of the blue and red faces. An edge in the overlaid arrangement may be a blue edge, a red edge, or an overlap of two differently colored edges. An overlay vertex may be a blue vertex, a red vertex, a coincidence of two differently colored vertices, or it may correspond to a blue–red intersection. In each case, the data associated with the overlaid DCEL feature should be computed from the red and blue DCEL features that induce it. To this end, the overlay visitor is parameterized by an overlay-traits type, which defines the merger operations between various DCEL features, achieving maximum genericity and flexibility for the users.

Let us recall the application of computing the cumulative set operations on polygon sets presented in Sections 3 and 4.3 to exemplify the use of an overlay traits. The polygon arrangement is represented using an extended DCEL class, such that each face f stores the number $N(f)$ of polygons that cover it. Assume that we are given a "blue"

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

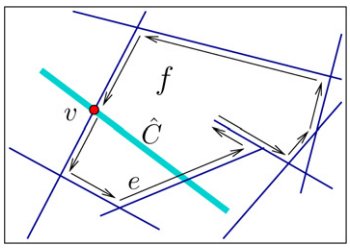
arrangement and a "red" arrangement, representing two polygon sets $\mathcal{S}_b$ and $\mathcal{S}_r$ respectively, and we wish to compute the arrangement that represents the consolidated set $\mathcal{S}_b \cup \mathcal{S}_r$. We overlay the two arrangements, using an overlay-traits class that operates as follows. Given an overlay face f_o induced by the intersection of a "blue" face f_b and a "red" face f_r, the overlay-traits class simply sets $N(f_o) \longleftarrow N(f_b) + N(f_r)$. It is straightforward to see that the resulting arrangement represents $\mathcal{S}_b \cup \mathcal{S}_r$. Cumulative set operations on this consolidated set can be easily computed as described in Section 3.

5.3. Zone-computation visitors

Many applications can make use of the following operation: Given an arrangement $\mathcal{A}$ and an x-monotone curve C, compute the *zone* of C in $\mathcal{A}$. That is, identify all arrangement cells that the curve crosses. The zone can be computed by locating the left endpoint of C in the arrangement, and then "walking" along the curve towards the right endpoint, keeping track of the vertices, edges, and faces crossed on the way (see, for example, [12, Section 8.3] for the computation of the zone of a line in an arrangement of lines).

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

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



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

The visitor class `Arr_inc_insert_zone_visitor` performs the incremental insertion of an x-monotone curve (see Fig. 5). It implements the functions described above to insert the generated subcurves by splitting the halfedges intersected by the curve and using the special insertion functions. Other zone visitors, such as a visitor that determines whether a query curve intersects with the curves of an arrangement, are even easier to implement.

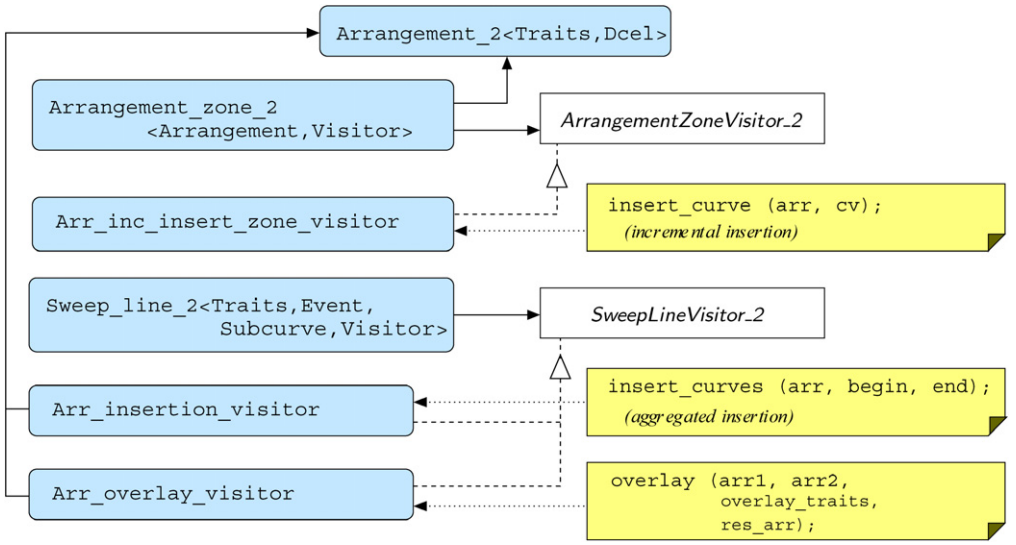


Fig. 5. The free functions (drawn as rectangles with a folded corner) that are implemented with the aid of visitor classes. Dotted arrows indicate the usage of a visitor by a free function. The other line types are explained in Figs. 3 and 4.

6. Adding information to curves

It is sometimes necessary to attach auxiliary (perhaps non-geometric) information to the curves that induce a specific arrangement instance. For example, in a geographical information system our curves may be polylines that represent streets in a specific city, and we would like to attach the street names to these curves. Section 2.3 explains how to extend the DCEL features. In particular, it is possible to extend the DCEL vertex and halfedge types. However, in many instances it is more natural to extend their geometric mappings (i.e., point and x-monotone curve, respectively) defined by the traits class at hand, as the auxiliary data is of geometric nature. Extending the x-monotone curve type is also more space efficient than extending the halfedge type, as there are two twin halfedges associated with every x-monotone curve.

In this section we present decorators that allow the extension of curve objects. 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. [22]). In our context, we use a traits-class decorator. Recall that the traits classes do not have a common base class, but they are all models of a traits concept. Indeed, the traits-class decorator must be instantiated with a model of the *ArrangementTraits_2* concept, referred to as the base-traits class. The decorator inherits some of the base-traits functors, while overriding others exploiting the auxiliary data maintained with the geometric objects.

An important application of traits-class decorators is in the `Arrangement_with_history_2` class-template, which is in turn a decorator that operates on an arrangement class, and maintains its construction history, as we describe in Section 6.2. In previous versions of the library [25] this construction history was stored in an auxiliary data structure, which needed constant maintenance by the arrangement class. By using the proper decorators, in conjunction with an arrangement observer, it is possible to obtain the construction history with only negligible overhead.

6.1. The traits-class decorators

We use traits-class decorators to extend the geometric entities defined by the traits class with additional, possibly non-geometric, data. An alternative way to achieve this is to extend the geometric types of the kernel, as the kernel is fully adaptable and extensible [27]. However, this indiscriminating extension may lead to an undue space-consumption, as every geometric object is extended, regardless of its use. It also requires nontrivial knowledge about the kernel structure and the techniques to extend it.

The `Arr_curve_data_traits_2<BaseTraits,XData,Merge,CData,Convert>` class template enables the extension of the curve types defined by a geometric base-traits class, which must be a model of the

ArrangementTraits_2 concept. It inherits its `Curve_2` from the curve type defined in the base-traits class and extends it with an additional data field of type `CData`. The `X_monotone_curve_2` type is also inherited from the corresponding type in the base-traits class and is extended with a data field of type `XData`. The class also supplies constructors and methods to access the data fields of its extended curve types.

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

- When a curve is subdivided into x-monotone subcurves, its `CData` field is converted using the `Convert` functor and propagated to all subcurves. By default, the `CData` and `XData` types are the same, and the data field is simply copied to the x-monotone subcurves.
- When we split an x-monotone curve into two, its data field is duplicated and stored with both resulting subcurves.
- When two x-monotone curves overlap, their data fields are merged using the `Merge` functor and the result is stored with the resulting subcurve that represents the overlap.
- We allow the merger of two x-monotone curves, only if they are geometrically mergeable (as determined by the base-traits class) *and* their data fields are equivalent.

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

An example

Suppose that we are given a few sets of data for some country: A geographical map of the country divided into regions, the national road and railroad network, and the water routes. Roads, railroads, and rivers are represented as polylines and have attributes (e.g., a name). We wish to obtain all crossroads and all bridges in some region of this country. To this end, we define the following classes:

```
stuct My_data {
   enum {ROAD, RAILROAD, RIVER}   type;
   std::string                    name;
};
Arr_consolidated_curve_data_traits_2<Polyline_traits_2,
                                     My_data>           traits;
```

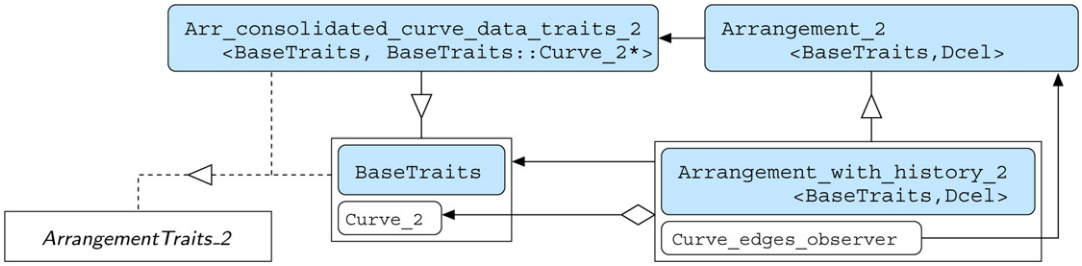


Fig. 6. The `Arrangement_with_history_2` decorator. The unfilled rounded rectangles indicate nested classes. The line types are explained in Figs. 3 and 4.

Each curve consists of a base-polyline (e.g., a road, a river) and a name. We construct the arrangement of all curves in our data sets overlaid on top of the regional map. Then, we can simply iterate over all arrangement vertices located in the desired region, and filter out those of degree less than four. If we encounter a vertex whose incoming halfedges are only of types `ROAD` or `RAILROAD`, we conclude it represents a crossroad. A vertex where halfedges of types `ROAD` (or `RAILROAD`) and `RIVER` meet represents a bridge. In any case, we can easily retrieve the names of the intersecting roads or rivers and present them as part of the output. The figure above shows the main roads and railroad in the Netherlands (drawn as dark curves), with the rivers and main water routes (drawn as lighter curves). The bridges, as computed by the application, are marked by small dots.

6.2. Curves with history

Another major component of the CGAL arrangement package is the class-template `Arrangement_with_history_2<BaseTraits,Dcel>`, which represents a planar arrangement of general curves, and maintains the *construction history* of this arrangement. Recall that the input curves that induce the arrangement are eventually split into x-monotone subcurves that are pairwise disjoint in their interior, and these subcurves are associated with the arrangement halfedges. While in the `Arrangement_2` class we lose track of the connections between input curves and the subcurves they induce, in the `Arrangement_with_history_2` class each halfedge stores a pointer to the input curve associated with it, or a container of curve pointers in case the edge is associated with an overlapping section of several curves. The arrangement decorator also stores the set of input curves, where each curve keeps the set of edges it induces. Users can traverse through the original curves of each arrangement halfedge, or iterate over all halfedges induced by a given input curve.

The `Arrangement_with_history_2` class is nothing more than a simple decorator for `Arrangement_2`, as shown in Fig. 6. It inherits from an arrangement class that is instantiated with the consolidated curve-data traits (Section 6.1), 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 halfedges are maintained using an *observer* (Section 4) that keeps track of each change that involves an arrangement halfedge and updates its underlying curve(s) accordingly.

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., [28]. It is possible, for example, to efficiently remove a curve from the arrangement by deleting all edges it induces.

6.3. Example: Cumulative set operations on fully-dynamic polygon sets

Let us revisit the cumulative set-operation example. In Section 3 we devised a solution for static polygon sets, and generalized it toward dynamic polygon sets in Section 4.3. We now generalize it further, and assume that our polygon sets are fully dynamic. Namely, polygons may also be removed after they are inserted into a set. In order to efficiently maintain the arrangement that encodes the topological structure of a polygon set, we use an `Arrangement_with_history_2` object instantiated with the polyline-traits class and a DCEL, whose faces are extended with an unsigned integer, which represents the counter $N(f)$:

```
typedef Arr_segment_traits_2<Kernel>                Segment_traits_2;
typedef Arr_polyline_traits_2<Segment_traits_2>     Polyline_traits_2;
typedef Arr_face_extended_dcel<unsigned int>        Extended_dcel;

Arrangement_with_history_2<Polyline_traits_2,
                           Extended_dcel>           arr;
```

Recall that each polygon is represented by a single curve, so each time we insert a polygon boundary into the arrangement, we receive a curve handle that serves as a polygon identifier. When we wish to remove a polygon $\hat{P}$, we simply call `arr.remove_curve(ch)`, where `ch` is the curve handle for this polygon. However, in order to keep the face counters up-to-date, we perform a simple preprocessing stage prior to the removal of the curve. Given the curve handle of the polygon, it is possible to traverse all arrangement edges induced by $\hat{P}$. We keep track of the set $\mathcal{F}_{\text{out}}$ of faces lying to the right of these edges, and a face f_{in} that lies to the left of one of the edges (contained in $\hat{P}$). In an analogous manner to the insertion operation described in Section 4.3, we perform a filtered breadth-first traversal starting from f_{in}, with the faces of $\mathcal{F}_{\text{out}}$ serving as our traversal filters, and decrement the counter $N(f)$ of every face f we encounter. Notice that the following curve-removal operation causes faces to merge, however the counters of two merged faces are always equal due to the performed preprocessing.

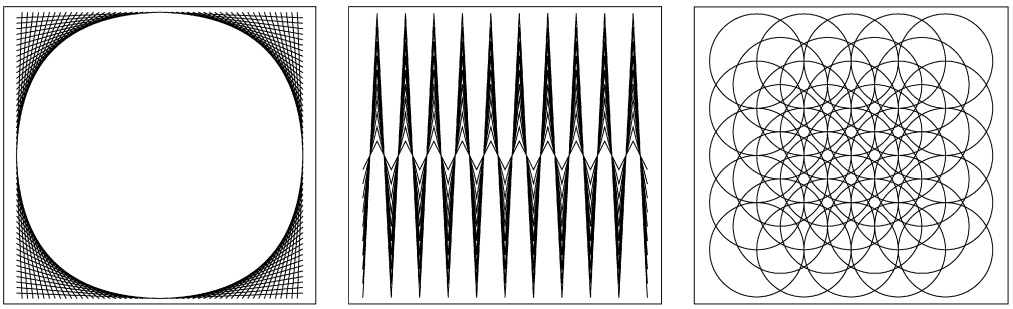
Using similar ideas it is not difficult to compute cumulative set operations of entities, which are more complicated, such as sets of polygons that may have polygonal holes. In this case the vertices of each polygonal hole are ordered in a *clockwise* direction around the polygon. Moreover, using a different arrangement-traits class, we can handle *general polygons*, whose edges are realized as non-linear arcs. In Section 7 we report on experimental results we obtained on general polygons, whose boundaries comprise of line segments and circular arcs.

7. Experimental results

In this section we show that in addition to the improved design of the new arrangement package and its more flexible and extensible interface, it is also more efficient and achieves significantly faster running time in comparison to previous versions.

We compared the performance of the arrangement module in the previous version of CGAL (3.1) and the module distributed with the new version (3.2). We tested the segment-traits class, the polyline-traits class, and the traits class for conic arcs (Section 2.2), all bundled in the public release of CGAL. For each traits class we used at least one structured data set that contains many degeneracies, and at least one data set that contains random curves in general position.

- The `onebig_n` input files contain n line segments forming four "fan-line" grids, each induced by $\frac{n}{4}$ segments, as depicted in Fig. 7(a). The `rnd_segs_n` files contain n segments, whose endpoints were randomly selected from a $[0, 10000] \times [0, 10000]$ integer grid.



(a) (b) (c)

Fig. 7. Structured inputs used in the benchmarks: (a) the `onebig_100` segment input, (b) the `sines_10` polyline input, (c) the `packed_41` circle input.

Table 1
Comparing Version 3.1 with Version 3.2; **V**, **E**, and **F** indicate the number of vertices, edges, and faces respectively; times are measured in seconds

Input file	Arrangement size			Version **3.1**		Version **3.2**	
	V	**E**	**F**	inc.	agg.	inc.	agg.
`onebig_100`	1504	2704	1202	3.416	0.122	0.292	0.043
`onebig_250`	8056	15376	7322	44.468	0.797	2.872	0.234
`rnd_segs_500`	33589	65678	32091	9.276	3.440	2.796	1.085
`rnd_segs_1000`	114376	225752	111380	30.257	13.010	9.678	3.872
`sines_10`	40	210	172	0.223	0.013	0.031	0.014
`rnd_plns_20`	4136	8109	3975	1.375	0.504	0.518	0.245
`packed_41`	507	1042	537	2.056	1.120	0.478	0.376
`rnd_elps_30`	677	1303	628	102.854	44.142	7.733	7.238
`rnd_elps_50`	1877	3663	1788	299.007	121.394	21.353	20.588

- The `sines_10` input file contains 10 sine-shaped polylines, each one comprises 20 line segments, as depicted in Fig. 7(b). The `rnd_plns_20` file contain 20 polylines, each comprises 10 line segments, whose endpoints were randomly selected from a $[0, 10000] \times [0, 10000]$ integer grid.
- The `packed_41` input file contains two layers of 5×5 and 4×4 circles, whose centers are located on a grid, resulting in a highly degenerate structure; see Fig. 7(c). The `rnd_elps_`*n* files contain random ellipses with integer radii, whose centers are randomly selected from a $[0, 30] \times [0, 30]$ integer grid.

We use the most efficient configurations in Version 3.1. Namely, the segment-traits class is instantiated with a kernel that uses the GMP exact rational number-type (`Cartesian<Gmpq>`), and such a segment-traits class is used to instantiate the polyline-traits class. The conic-traits class uses the number types supplied by the CORE library, which are capable of carrying out certified computations with algebraic numbers. In each case, we construct the arrangement incrementally, using the walk-along-a-line point-location strategy (Section 4.2), or aggregately, using a sweep-line process that inserts all input curves simultaneously.

The running times we obtained in our experiments are summarized in Table 1. The running times reported in this section where obtained on a 3 GHz Pentium IV machine with 2 Gb of RAM, running a Linux operating system. The software was compiled using the Gnu C++ compiler (`g++` Version 3.3.2).

As mentioned throughout this paper, we minimized the number of geometric predicates and constructions (calls to traits-class functors) invoked by the various algorithms in the new version of the arrangement package. This minimization is the main reason for the considerable speedups achieved in Version 3.2. Table 2 lists the number of calls to the various traits-class operations for three selected data-sets, as obtained in Version 3.1 and in Version 3.2. We mention that the set of traits-class operations in Version 3.1 was slightly different than it is in the new version: an x-monotone curve used to have a source and a target point, instead of minimal and maximal endpoints; we used to compare the intersections one by one, instead of a single call that computes all intersections; and finally, some operations that are now computed by the traits-class adapter used to be part of the traits class (e.g., comparing two x-monotone curves over a given point—see Section 2.2.1 for more details).

Recall that in Version 3.2 the main algorithmic frameworks, namely the sweep-line framework and the zone-computation framework, are implemented as two centralized classes (see Section 5). This code centralization enabled us to improve the core algorithms, so they better utilize combinatorial and topological information and invoke fewer geometric operations.

Minimizing the number of calls to traits-class functions has also paved the way to use faster geometric kernels. CGAL includes a predefined filtered kernel [27], named `Exact_predicates_exact_constructions_kernel`, which applies arithmetic filters based on interval arithmetic [10], in order to filter out computations with the exact `Gmpq` type. Only when a filter fails, in case of a degenerate (or near degenerate) situation, does the kernel resort to exact computing, in order to arrive at a correct result. In Version 3.2 of the arrangement package major efforts were made to exploit combinatorial information to avoid redundant geometric predicates. Redundant tests, such as comparing two points, which are in fact equal, cause filter failures; therefore, they render the filtered kernel inefficient. On the other hand, the new version successfully benefits from using filtered computations in case of non-degenerate input, as our experiments show.

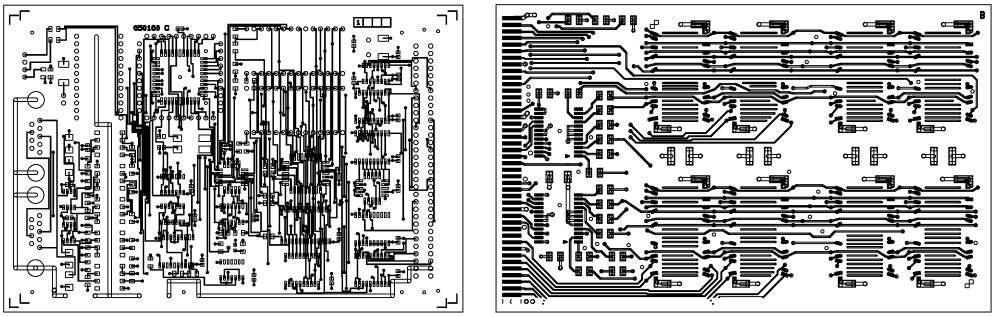
Table 2
The number of calls to traits-class operations in Version 3.1 and in Version 3.2

Operation name	onebig_250		random_500		packed_41	
	inc.	agg.	inc.	agg.	inc.	agg.
Version 3.1						
compare x	2,553,663	171,871	1,349,760	762,873	21,500	13,157
compare xy	3,136,876	228,904	905,196	924,150	9,312	13,092
get source	11,420,454	50,964	1,497,407	219,733	20,091	3,834
get target	9,278,696	43,154	1,002,356	185,587	12,892	3,300
in x-range	20,218,305	29,533	707,079	148,241	5,560	2,778
is vertical	171,791	132,994	658,521	612,800	11,932	9,731
point status	196,529	246	72,768	3,213	1,378	128
compare at x	14,595	115,169	92,017	506,437	1,738	9,573
compare to left	17,019	0	90,048	0	2,115	0
compare to right	28,061	40,948	137,614	182,519	2,348	3,789
next intersection	2,102,007	22,695	241,247	99,149	2,970	1,598
do overlap	22,696	18,794	130,642	80,076	2,254	1,454
Version 3.2						
compare x	2,937,242	0	718,021	0	8,497	0
compare xy	8,608,206	75,856	573,217	301,419	5,858	4,788
min vertex	2,750,792	248	558,780	500	6,518	82
max vertex	2,735,012	248	466,876	500	5,293	82
is vertical	1,163	0	52,265	0	1,515	0
point status	24,529	1,749	84,440	3,593	1,619	44
compare to left	62	0	7,684	0	760	0
compare to right	7,590	1	5,892	0	625	164
intersect	47,843	7,811	90,632	33,864	1,586	602

Table 3
Comparing different kernels in Version 3.2; times are measured in seconds

Input file	Using Gmpq		Filtered kernel		Lazy kernel		Using double	
	inc.	agg.	inc.	agg.	inc.	agg.	inc.	agg.
onebig_100	0.292	0.043	0.209	0.083	0.131	0.099	0.049	0.010
onebig_250	2.872	0.234	2.055	0.459	1.152	0.531	0.597	0.064
rnd_segs_500	2.796	1.085	1.428	0.480	0.657	0.374	0.373	0.266
rnd_segs_1000	9.678	3.872	4.953	1.787	2.348	1.425	1.318	0.991
sines_10	0.031	0.014	0.013	0.005	0.007	0.003	0.004	0.002
rnd_plns_20	0.518	0.245	0.389	0.075	0.215	0.059	N/A	0.037

Table 3 summarizes the benchmarks we conducted using different CGAL kernels for instantiating the segment-traits and the polyline-traits classes. The first two columns correspond to the Cartesian<Gmpq> kernel and are copied from Table 1. The next two columns correspond to the predefined filtered kernel described above. We also give results for the *lazy* geometric kernel [38], an emerging CGAL kernel that uses novel filtering techniques; this kernel is currently available only in the internal CGAL releases, but will probably replace the predefined filtered kernel in the forthcoming CGAL release (Version 3.3). Finally, we give results obtained using machine-precision floating-point arithmetic, as exercised by the Simple_cartesian<double> kernel. The minimized number of traits-class invocations makes it possible to use floating-point arithmetic in many input scenarios and obtain topologically correct results. This was not feasible in previous versions. We also note that with non-degenerate inputs, using certified computation incurs a running-time overhead of only 40% compared to the machine-precision arithmetic when using aggregated construction, and an overhead of 75% when using incremental construction. Users can easily experiment and select the best kernel suited for their applications, as switching between different kernels boils down to instantiating the Arr_segment_traits_2<Kernel> class-template with a different Kernel type.



(a) (b)

Fig. 8. Input files representing PCBs that contain generalized polygons: (a) the `VLSI_1` input and (b) the `VLSI_2` input. (Files in courtesy of Maniabarco Inc.)

Table 4
Computing the union of generalized polygons and circles taken from VLSI models; **V**, **E**, and **F** indicate the number of vertices, edges, and faces respectively

Input file	Num. of polygons	Num. of circles	Union size			Time (sec).
			V	**E**	**F**	
`VLSI_1`	2593	645	13130	13130	614	3.50
`VLSI_2`	22406	294	14698	14698	357	24.70

We finally give some results that demonstrate the industrial strength of the new arrangement package. Here we use two VLSI models that represent printed circuit boards (PCBs), the components of which have been dilated by a small radius; see Fig. 8 for an illustration. The models therefore contain a large number of generalized polygons (representing dilated segments or dilated polygons) and circles (representing dilated points or dilated circles), where the boundary of a general polygon comprises line segments and circular arcs. We have to compute the union of the dilated components, and it is possible to do so in an aggregated manner, following the algorithm described in Section 3. In this case we also perform a cleanup phase during which we remove each arrangement edge whose incident faces are both contained in the union, in order to simplify the representation of the union. The result can be expressed as a set of disjoint generalized polygons that contain holes. We use the `Arr_circle_segment_2` traits class that is capable of handling line segments and circular arcs *in an exact manner*; its operations are optimized and use arithmetic filtering schemes, so it achieves very fast running times. The input size, the size of the union, and the running time for the two VLSI models are given in Table 4.

8. Conclusions and future work

We have shown how careful software design, combined with a careful handling of the geometric and the topological data-structures, yield a software package that is robust and complete, and at the same time efficient and easy to extend. We have conducted a large set of benchmarks, some of them summarized in this paper, which prove the efficiency of our package. The ability to extend and customize the arrangement package for the usage of various applications is also demonstrated throughout the paper.

We continue the development of the arrangement package so that it also supports arrangements of unbounded curves, which may induce planar subdivisions that contain several unbounded faces. This extended functionality will be available in the forthcoming public release of CGAL (Version 3.3). In this release we also introduce new traits classes that use certified computations for handling new families of curves—e.g., a traits class for planar Bézier curves.

Another direction we intend to pursue is the representation of 3-dimensional structures using planar arrangement. A significant progress has already been made in developing a CGAL package that computes the lower envelope of a set of bounded surfaces using a divide-and-conquer approach. This package makes extensive use of most of the

new features introduced in the two-dimensional arrangement package described in this paper in order to achieve an efficient implementation of the algorithm [35].

Acknowledgements

The following developers have contributed to previous versions of the arrangement package: Esther Ezra, Eyal Flato, Iddo Hanniel, Idit Haran, Shai Hirsch, Eugene Lipovetsky, Oren Nechushtan, Sigal Raab and Tali Zvi.

The new design of the arrangement package has greatly benefited from many discussions with other CGAL developers. In particular we wish to thank Eric Berberich, Arno Eigenwillig, Andreas Fabri, Susan Hert, Michael Hemmer, Lutz Kettner, Sylvain Pion and Monique Teillaud. We would also like to thank Andreas Fabri and Sylvain Pion for reviewing the revised arrangement package.

We also thank the anonymous referees for many helpful comments and suggestions for improving this paper.

References

[1] CGAL-3.2 User and Reference Manual, 2006; http://www.cgal.org/Manual/3.2/doc_html/cgal_manual/index.html.
[2] P.K. Agarwal, E. Flato, D. Halperin, Polygon decomposition for efficient construction of Minkowski sums, Computational Geometry: Theory and Applications 21 (2002) 39–61.
[3] M.H. Austern, Generic Programming and the STL: Using and Extending the C++ Standard Template Library, Addison-Wesley, 1998.
[4] U. Bartuschka, S. Näher, M. Seel, A generic plane sweep framework, 2000, unpublished manuscript.
[5] J.L. Bentley, T. Ottmann, Algorithms for reporting and counting geometric intersections, IEEE Transactions on Computers 28 (9) (1979) 643–647.
[6] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, N. Wolpert, EXACUS: Efficient and exact algorithms for curves and surfaces, in: Proc. 13th Europ. Sympos. Alg. (ESA), in: Lecture Notes in Comput. Sci., vol. 3669, Springer, Berlin, 2005, pp. 155–166.
[7] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, E. Schömer, A computational basis for conic arcs and Boolean operations on conic polygons, in: Proc. 10th Europ. Sympos. Alg. (ESA), in: Lecture Notes in Comput. Sci., vol. 2461, Springer, Berlin, 2002, pp. 174–186.
[8] E. Berberich, M. Hemmer, L. Kettner, E. Schömer, N. Wolpert, An exact, complete and efficient implementation for computing planar maps of quadric intersection curves, in: Proc. 21st Annu. ACM Sympos. Comput. Geom. (SCG), 2005, pp. 99–106.
[9] J.-D. Boissonnat, O. Devillers, M. Teillaud, M. Yvinec, Triangulations in CGAL, in: Proc. 16th Annu. ACM Sympos. Comput. Geom., 2000, pp. 11–18.
[10] H. Brönnimann, C. Burnikel, S. Pion, Interval arithmetic yields efficient dynamic filters for computational geometry, Discrete Applied Mathematics 109 (1–2) (2001) 25–47.
[11] D. Cohen-Or, S. Lev-Yehudi, A. Karol, A. Tal, Inner-cover of non-convex shapes, International Journal on Shape Modeling 9 (2) (2003) 223–238.
[12] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry: Algorithms and Applications, second ed., Springer, Berlin, 2000.
[13] O. Devillers, A. Fronville, B. Mourrain, M. Teillaud, Algebraic methods and arithmetic filtering for exact predicates on circle arcs, Computational Geometry: Theory and Applications 22 (1–3) (2002) 119–142.
[14] D.A. Duc, N.D. Ha, 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; http://www.cgal.org/UserWorkshop/2002/ndha_abstract.pdf.
[15] A. Eigenwillig, L. Kettner, E. Schömer, N. Wolpert, Complete, exact and efficient computations with cubic curves, in: Proc. 20th Annu. ACM Sympos. Comput. Geom. (SCG), 2004, pp. 409–418.
[16] I.Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, E.P. Tsigaridas, Towards an open curved kernel, in: Proc. 20th Annu. ACM Sympos. Comput. Geom. (SCG), 2004, pp. 438–446.
[17] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, S. Schönherr, On the design of CGAL, the Computational Geometry Algorithms Library, Software—Practice and Experience 30 (2000) 1167–1202.
[18] E. Flato, Robust and efficient construction of planar Minkowski sums, M.Sc. thesis, School of Computer Science, Tel-Aviv University, 2000; http://www.cs.tau.ac.il/CGAL/Theses/flato/thesis/.
[19] E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, E. Ezra, The design and implementation of planar maps in CGAL, The ACM Journal of Experimental Algorithmics 5 (2000) 1–23.
[20] E. Fogel, D. Halperin, Exact and efficient construction of Minkowski sums of convex polyhedra with applications, in: Proc. 8th Workshop Alg. Eng. Exper. (Alenex'06), 2006, pp. 3–15.
[21] E. Fogel, R. Wein, D. Halperin, Code flexibility and program efficiency by genericity: Improving CGAL's arrangements, in: Proc. 12th Europ. Sympos. Alg. (ESA), in: Lecture Notes in Comput. Sci., vol. 3221, Springer, Berlin, 2004, pp. 664–676.
[22] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns—Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[23] D. Halperin, Arrangements, in: J.E. Goodman, J. O'Rourke (Eds.), Handbook of Discrete and Computational Geometry, second ed., Chapman & Hall/CRC, 2004, pp. 529–562. Chapter 24.
[24] I. Hanniel, The design and implementation of planar arrangements of curves in CGAL, MSc thesis, School of Computer Science, Tel Aviv University, 2000.

[25] I. Hanniel, D. Halperin, Two-dimensional arrangements in CGAL and adaptive point location for parametric curves, in: Proc. 4th Internat. Workshop Alg. Eng. (WAE), in: Lecture Notes in Comput. Sci., vol. 1982, Springer, Berlin, 2000, pp. 171–182.
[26] I. Haran, D. Halperin, An experimental study of point location in general planar arrangements, in: Proc. 8th Workshop Alg. Eng. Exper. (Alenex'06), 2006, pp. 16–25.
[27] S. Hert, M. Hoffmann, L. Kettner, S. Pion, M. Seel, An adaptable and extensible geometry kernel, in: Proc. 5th Internat. Workshop Alg. Eng. (WAE), in: Lecture Notes in Comput. Sci., vol. 2141, Springer, Berlin, 2001, pp. 79–90.
[28] S. Hirsch, D. Halperin, Hybrid motion planning: Coordinating two discs moving among polygonal obstacles in the plane, in: J.-D. Boissonnat, J. Burdick, K. Goldberg, S. Hutchinson (Eds.), Algorithmic Foundations of Robotics V, Springer, Berlin, 2003, pp. 239–255.
[29] V. Karamcheti, C. Li, I. Pechtchanski, C. Yap, A core library for robust numeric and geometric computation, in: Proc. 15th Annu. ACM Sympos. Comput. Geom. (SCG), 1999, pp. 351–359.
[30] L. Kettner, Using generic programming for designing a data structure for polyhedral surfaces, Computational Geometry: Theory and Applications 13 (1) (1999) 65–90.
[31] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, C. Yap, Classroom examples of robustness problems in geometric computations, in: Proc. 12th Europ. Sympos. Alg. (ESA), in: Lecture Notes in Comput. Sci., vol. 3221, Springer, Berlin, 2004, pp. 702–713.
[32] J. Keyser, T. Culver, M. Foskey, S. Krishnan, D. Manocha, ESOLID—a system for exact boundary evaluation, Computer-Aided Design 36 (2) (2004) 175–193.
[33] J. Keyser, T. Culver, D. Manocha, S. Krishnan, Efficient and exact manipulation of algebraic points and curves, Computer-Aided Design 32 (11) (2000) 649–662.
[34] K. Mehlhorn, S. Näher, LEDA: A Platform for Combinatorial and Geometric Computing, Cambridge University Press, Cambridge, UK, 2000.
[35] M. Meyerovitch, Robust, generic and efficient construction of envelopes of surfaces in three-dimensional space, in: Proc. 14th Europ. Sympos. Alg. (ESA), in: Lecture Notes in Comput. Sci., vol. 4168, Springer, Berlin, 2006, pp. 792–803.
[36] K. Mulmuley, A fast planar partition algorithm, I, Journal of Symbolic Computation 10 (3–4) (1990) 253–280.
[37] N. Myers, A new and useful template technique: "Traits", in: S.B. Lippman (Ed.), C++ Gems, SIGS Reference Library, vol. 5, 1997, pp. 451–458.
[38] A. Fabri, S. Pion, A generic lazy evaluation scheme for exact geometric computations, in: Proc. 2nd Workshop on Library-Centric Software Design (LCSD), 2006, http://sms.cs.chalmers.se/bibliography/proceedings/2006-LCSD.pdf.
[39] F.P. Preparata, M.I. Shamos, Computational Geometry: An Introduction, Springer, New York, 1985.
[40] V. Rogol, Maximizing the area of an axially-symmetric polygon inscribed by a simple polygon, MSc thesis, Technion, Haifa, Israel, 2003.
[41] S. Schirra, Robustness and precision issues in geometric computation, in: J.-R. Sack, J. Urrutia (Eds.), Handbook of Computational Geometry, Elsevier Science Publishers B.V., North-Holland, Amsterdam, 1999, pp. 597–632.
[42] J.G. Siek, L.-Q. Lee, A. Lumsdaine, The Boost Graph Library, User Guide and Reference Manual, Addison-Wesley, 2002.
[43] J. Snoeyink, J. Hershberger, Sweeping arrangements of curves, in: Proc. 5th Annu. ACM Sympos. Comput. Geom. (SCG), 1989, pp. 354–363.
[44] R. Wein, High-level filtering for arrangements of conic arcs, in: Proc. 10th Europ. Sympos. Alg. (ESA), in: Lecture Notes in Comput. Sci., vol. 2461, Springer, Berlin, 2002, pp. 884–895.
[45] R. Wein, Efficient implementation of red-black trees with split and catenate operations, Technical report, Tel-Aviv University, 2005; http://www.cs.tau.ac.il/~wein/publications/pdfs/rb_tree.pdf.
[46] R. Wein, E. Fogel, The new design of CGAL's arrangement package, Technical report, Tel-Aviv University, 2005; http://www.cs.tau.ac.il/~wein/publications/pdfs/Arr_new_design.pdf.
[47] R. Wein, J.P. van den Berg, D. Halperin, The visibility-Voronoi complex and its applications, Computational Geometry: Theory and Applications 36 (1) (2007) 66–87.
[48] C.K. Yap, Robust geometric computation, in: J.E. Goodman, J. O'Rourke (Eds.), Handbook of Discrete and Computational Geometry, second ed., Chapman & Hall/CRC, 2004, pp. 927–952. Chapter 41.