Code Flexibility and Program Efficiency by Genericity: Improving CGAL's Arrangements*

Efi Fogel, Ron Wein, and Dan Halperin

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

Abstract. Arrangements of planar curves are fundamental structures in computational geometry. We describe the recent developments in the arrangement package of CGAL, the Computational Geometry Algorithms Library, making it easier to use, to extend and to adapt to a variety of applications. This improved flexibility of the code does not come at the expense of efficiency as we mainly use generic-programming techniques, which make dexterous use of the compilation process. To the contrary, we expedited key operations as we demonstrate by experiments.

1 Introduction

Given a set $\mathcal{C}$ of planar curves, the *arrangement* $\mathcal{A}(\mathcal{C})$ is the subdivision of the plane induced by the curves in $\mathcal{C}$ into maximally connected cells. The cells can be 0-dimensional (*vertices*), 1-dimensional (*edges*) or 2-dimensional (*faces*). The *planar map* of $\mathcal{A}(\mathcal{C})$ is the embedding of the arrangement as a planar graph, such that each arrangement vertex corresponds to a planar point, and each edge corresponds to a planar subcurve of one of the curves in $\mathcal{C}$. Arrangements and planar maps are ubiquitous in computational geometry, and have numerous applications (see, e.g., [8, 15] for some examples), so many potential users in academia and in the industry may benefit from a generic implementation of a software package that constructs and maintains planar arrangements.

CGAL [1], the Computational Geometry Algorithms Library, is a software library, which 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* [11, 18], that in turn consists of constant-size non-modifiable geometric primitive objects (such as points, line segments, triangles etc.) and predicates and operations on these objects. On top of the kernel layer, the library consists of a collection of modules, which provide implementations of many fundamental geometric data structures and algorithms. The arrangement package is a part of this layer.

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, there are no degenerate cases (e.g., three

* This work has been supported in part by the IST Programs of the EU as Shared-cost RTD (FET Open) Projects under Contract No IST-2000-26473 (ECG — Effective Computational Geometry for Curves and Surfaces) and No IST-2001-39250 (MOVIE — Motion Planning in Virtual Environments), by The Israel Science Foundation founded by the Israel Academy of Sciences and Humanities (Center for Geometric Computing and its Applications), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.

curves intersecting at a common point) in the input. Secondly, operations on real numbers yield accurate results (the "real RAM" model, which also assumes that each basic operation takes constant time). Unfortunately, these assumptions do not hold in practice. Thus, an algorithm implemented from a textbook may yield incorrect results, get into an infinite loop or just crash while running on a degenerate, or nearly degenerate, input (see [25] for examples).

The need for robust software implementation of computational-geometry algorithms has driven many researches to develop variants of the classic algorithms that are less susceptible to degenerate inputs over the last decade. At the same time, advances in computer algebra enabled the development of efficient software libraries that offer exact arithmetic manipulations on unbounded integers, rational numbers (GMP — Gnu's multi-precision library [5]) and even algebraic numbers (the CORE [2] library and the numerical facilities of LEDA [6]). These exact *number types* serve as fundamental building-blocks in the robust implementation of many geometric algorithms.

Keyser et al. [21] implemented an arrangement-construction module for algebraic curves as part of the MAPC library. However, their implementation makes some general position assumptions. The LEDA library [6] includes geometric facilities that allow the construction and maintenance of planar maps of line segments. LEDA-based implementations of arrangements of conic curves and of cubic curves were developed under the EXACUS project [3].

CGAL's arrangement package was the first generic 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 this package can be found in [12, 17]. In this paper we summarize the recent efforts that have been put into the arrangement package and show the improvements achieved: A software design relying on the generic-programming paradigm that is more modular and easy to use, and an implementation which is more extensible, adaptable, and efficient.

The rest of this paper is organized as follows: Section 2 provides the required background on CGAL's arrangement package introducing its architecture key-points, with a special attention to the *traits* concept. In Section 3 we demonstrate the use of generic programming techniques to improve modularity and flexibility or gain functionality that did not exist in the first place. In Section 4 we describe the principle of a meta-traits class and show how it considerably simplifies the curve hierarchy of the arrangement (as introduced in [17]). We present some experimental results in Section 5. Finally, concluding remarks and future-research suggestions are given in Section 6.

2 Preliminaries

2.1 The Arrangement Module Architecture

The `Planar_map_2<Dcel,Traits>`[1] class-template represents the planar embedding of a set of x-monotone planar curves that are pairwise disjoint in their inte-

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

riors. It is derived from the `Topological_map` class, which provides the necessary combinatorial capabilities for maintaining the planar graph, while associating geometric data with the vertices, edges and faces of the graph. The planar map is represented using a *doubly-connected edge list* (DCEL for short), a data structure that enables efficient maintenance of two-dimensional subdivisions (see [8, 20]).

The `Planar_map_2` class-template should be instantiated with two parameters. A DCEL class, which represents the underlying topological data structure, and 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. The two template parameters enable the separation between the topological and geometric aspects of the planar subdivision. This separation is advantageous as it allows users to employ the package with their own special type of curves, without having any expertise in computational geometry. They should only be capable of supplying the traits methods, which mainly involve algebraic computations. Indeed, several of the package users are not familiar with computational-geometry techniques and algorithms.

The `Planar_map_with_intersections_2` class-template, should be instantiated with a `Planar_map_2` class. It inherits from the planar-map class and extends its functionality by enabling insertion of intersecting and not necessarily x-monotone curves. The main idea is to break each input curve into several x-monotone subcurves, then treat each subcurve separately. Each subcurve is in turn split at its intersection points with other curves. The result is a set of x-monotone and pairwise disjoint subcurves that induce a planar subdivision, equivalent to the arrangement of the original input curves [12]. An arrangement of a set of curves can be constructed *incrementally*, inserting the curves one by one, or *aggregately*, using a sweep-line algorithm (see, e.g., [8]). Once the arrangement is constructed, *point-location* queries on it can be answered, using different point-location strategies (see [12] for more details). Additional interface functions that modify, traverse, and display the map are available as well.

The `Arrangement_2` class-template allows the construction of a planar map with intersections, while maintaining a hierarchy of curve history. At the top level of the hierarchy stand the input curves. Each curve is subdivided into several x-monotone subcurves, forming the second level of the hierarchy. As indicated above, these x-monotone subcurves are further split such that there is no pair of subcurves that intersect in their interior. These subcurves comprise the low level of the curve hierarchy. See [17] for more details regarding the design of the `Arrangement_2` template. The curve hierarchy is essential in a variety of applications that use arrangements. Robot motion planning is one example.

2.2 The Traits Class

As mentioned in the previous subsection, the `Planar_map_2` class template is parameterized with a *traits* class that defines the abstract interface between planar maps and the geometric primitives they use. The name "traits" was given by

Myers [23] for a concept of a class that should support certain predefined methods, passed as a parameter to another class template. In our case, the geometric traits-class defines the family of curves handled. Moreover, details such as the number type used to represent coordinate values, the type of coordinate system used (Cartesian, homogeneous, polar), the algebraic methods used and whether extraneous data is stored with the geometric objects, are all determined by the traits. A class that follows the geometric traits-class concept defines two types of objects, namely `X_monotone_curve_2` and `Point_2`. The former represents an x-monotone curve, and the latter is the type of the endpoints of the curves, representing a point in the plane. In addition, the concept lists a minimal set of predicates on objects of these two types, sufficient to enable the operations provided by the `Planar_map_2` class:

1. Compare two points by their x-coordinates only or lexicographically (by their x and then by their y-coordinates).
2. 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 if p is above, below or lies on C.
3. Compare the y-values of two x-monotone curves C_1, C_2 at a given x-value in the x-range of both curves.
4. Given two x-monotone curves C_1, C_2 and one of their intersection points p, determine the relative positions of the two curves immediately to the right of p, or immediately to the left of p.[2]

In order to support the construction of an arrangement of curves (more precisely, a `Planar_map_with_intersections_2`) one should work with a refined traits class. In addition to the requirements of the planar map traits concept, it defines a third type that represents a general (not necessarily x-monotone) curve in the plane, named `Curve_2`. An intersection point of the curves is of type `Point_2`. It also lists a few more predicates and geometric constructions on the three types as follows:

1. Given a curve C, subdivide it into simple x-monotone subcurves $C_1, \ldots C_k$.
2. Given two x-monotone curves C_1, C_2 and a point p, find the next intersection point of the two curves that is lexicographically larger, or lexicographically smaller, than p. In degenerate situations, determine the overlap between the two curves.

We include several traits classes with the public distribution of CGAL: A traits class for line segments; a traits class that operates on polylines, namely continuous piecewise linear curves [16]; and a traits class that handles conic arcs — segments of planar algebraic curves of degree 2 such as ellipses, hyperbolas or parabolas [26]. There are other traits classes that were developed in other sites [13] and are not part of the public distribution. Many users (see, e.g., [7, 10, 14, 19, 24]) have employed the arrangement package to develop a variety of applications.

[2] Notice that when we deal with curved objects the intersection point may also be a tangency point, so the relative position of the curves to the right of p may be the same as it was to its left.

3 Genericity — The Name of the Game

In this section we describe how we exploited several generic-programming techniques to make our arrangement package more modular, extensible, adaptable, and efficient. We have tightened the requirements from the traits concept, and allowed for an alternative subset of requirements to be fulfilled using a tag-dispatching mechanism, enabling easier development of external traits classes. At the same time, we have also improved the performance of the built-in traits classes and extended their usability through deeper template nesting.

3.1 Flexibility by Genericity

When constructing an arrangement using the sweep-line algorithm, we sweep the input set of planar curves from left to right, so it is sufficient to find just the next intersection point of a pair of curves to the right of a given reference point, and to compare two curves to the right (and not to the left) of their intersection point.[3] It is therefore sufficient for our arrangement traits-class to provide just a subset of the requirements listed in Section 2.2. However, even if one uses the incremental construction algorithm, one may wish to implement a reduced set of traits class methods in order to avoid code duplication.

We use a *tag-dispatching* mechanism (see [4] for more details) to select the appropriate implementation that enables users to implement their traits class with an alternative or even reduced set of member functions. The traits class is in fact injected as a template parameter into a traits-class *wrapper*, which also inherits from it. The wrapper serves as a mediator between the planar-map general operations and the traits-class primitive operations. It uses the basic set of methods provided by the base traits-class to implement a wider set of methods, using tags to identify the missing or rather the existing basic methods.

The tag `Has_left_category`, for example, indicates whether the requirements for the two methods below are satisfied:

1. Given two x-monotone curves C_1, C_2 and one of their intersection points p, determine the relative positions of the two curves immediately to the left of p.
2. Given two x-monotone curves C_1, C_2 and a point p, find the next intersection point of the two curves that is lexicographically smaller than p.

The tag `Has_reflect_category` indicates whether an alternative requirement is satisfied. That is, whether functions that reflect a point or a curve about the major axes are provided. When the *has-left* tag is false and the *reflect* tag is true, the next intersection point, or a comparison to the left of a reference point, is computed with the aid of the alternative methods by reflecting the relevant

[3] At first glance, it may seem that implementing the next intersection to the left computation is a negligible effort once we implement the next intersection to the right computation. However, for some sophisticated traits classes, such as the one described in [9], it is a major endeavor.

objects, performing the desired operation to the right, and then reflecting the results back. This way the user is exempt from providing an implementation of the "left" methods. If none of these methods are provided, we resort to (somewhat less efficient) algorithms based just on the reduced set of provided methods: We locate a new reference point to the left of the original point, and check what happens to its right.

3.2 Efficiency by Genericity

In geometric computing there is a major difference between algorithms that evaluate predicates only, and algorithms that in addition construct new geometric objects. A predicate typically computes the sign of an expression used by the program control, while a constructor produces a new geometric object such as the intersection point of two segments. If we use an exact number type to ensure robustness, the newly constructed objects often have a more complex representation in comparison with the input objects (i.e. the bit-length needed for their representation is often larger). Unless the overall algorithm is carefully designed to deal with these new objects, constructions will have a severe impact on the algorithm performance.

The `Arr_segment_traits_2` class is templated with a geometric *kernel* object, that conforms to the CGAL kernel-concept [18], and supplies all the data types, predicates and constructions on linear objects. A natural candidate is CGAL's Cartesian kernel, which represents each segment using its two endpoints, while each point is represented using two rational coordinates. This simple representation is very natural, yet it may lead to a cascaded representation of intersection points with exponentially long bit-length (see Figure 1 for an illustration).[4]

To avoid this cascading problem, we introduced the `Arr_cached_segment_traits_2` class. This traits class is also templated by a geometric kernel, but uses its own internal representation of a segment: In addition to the two endpoints it also stores the coefficients of the underlying line. When a segment is split, the underlying line of the two resulting sub-segments remains the same and only their endpoints are updated. When we compute an intersection point of two segments, we use the coefficients of the corresponding underlying lines and thus overcome the undesired effect of cascading.

The *cached* segment traits-class achieves faster running times than the kernel segment traits, when arrangements with relatively many intersection points are constructed. It also allows for working with less accurate, yet computationally efficient number types, such as `Quotient<MP_Float>`[5] (see CGAL's manual at [1]). On the other hand, it uses more space and stores extra data with each segment, so constructing sparse arrangements could be more efficient with the

[4] A straightforward solution would be to normalize all computations. However, our experience shows that indiscriminate normalization considerably slows down the arrangement construction.

[5] `MP_Float` represents floating-point numbers with an unbounded mantissa, but with a bounded exponent. In some cases (see, e.g, Figure 1) the exponent may overflow.

kernel (non-cached) traits-class implementation. As our software is generic, users can easily switch between the two traits classes and check which one is more suitable for their application by changing just a few lines of code. An experimental comparison of the two types of segment traits-classes is presented in Section 5.

3.3 Succinctness by Genericity

Polylines are of particular interest, as they can be used to approximate higher-degree algebraic curves, and at the same time they are easier to deal with in comparison with conics for example.[6]

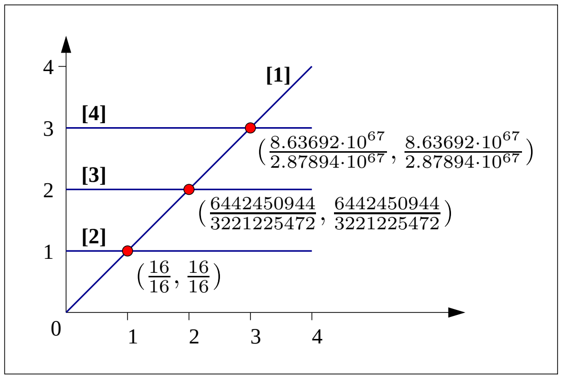

[1]
[2]
[3]
[4]
$(\frac{16}{16}, \frac{16}{16})$
$(\frac{6442450944}{3221225472}, \frac{6442450944}{3221225472})$
$(\frac{8.63692 \cdot 10^{67}}{2.87894 \cdot 10^{67}}, \frac{8.63692 \cdot 10^{67}}{2.87894 \cdot 10^{67}})$
0
1
2
3
4

Fig. 1. Cascaded computation of the coordinates of the intersection points in an arrangement of four segments. The order of insertion of the segments is indicated in brackets. Notice the exponential growth of the bitlengths of the intersection-point coordinates.

Previous releases of CGAL included a stand-alone polyline traits class. This class represented a polyline as a list of points and performed all geometric operations on this list (see [16] for more details). We have recently rewritten the polyline traits-class, making it a class template named `Arr_polyline_traits_2`, that is parametrized with a `Segment_traits`, a traits class appropriate for handling segments. The polyline is implemented as a vector of `Segment_traits::Curve_2` objects (namely of segments). The new polyline traits-class does not perform any geometric operation directly. Instead, it relies solely on the functionality of the segment traits. For example, when we wish 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 compare the point to this segment. Operations on polylines of size m therefore take $O(\log m)$ time.

Users are free to choose the underlying segment traits, giving them the ability to use the kernel (non-cached) segment traits or the cached segment traits, depending on the number of expected intersection points. Moreover, it is possible to instantiate the polyline traits template with a *data traits-class* that handles segments with some additional data (see the next section). This makes it possible to associate some data with the entire polyline and possibly different data with each of the segments of the set that comprises it.

[6] With polylines it is sufficient to use an exact rational number type.

4 The Data Meta-Traits

Additional information can be maintained either by extending the vertex, half-edge, or face types provided by the topological map through inheritance, or alternatively by extending their geometric mappings — that is, the point and curve types. The former option should be used to retain additional information related to the planar-map topology, and is done by instantiating an appropriate Dcel, which can be conveniently derived from the one provided with the Cgal distribution. The latter option can be carried out by extending the geometric types of the kernel, as the kernel is fully adaptable and extensible [18], but this indiscriminating extension could lead to an undue space consumption. Extending the curve type of the planar-map only is easy with the meta traits which we describe next.

We define a simple yet powerful meta-traits class template called *data traits*. The data traits-class is used to extend planar-map curve types with additional data. It should be instantiated with a regular traits-class, referred to as the *base traits-class*, and a class that contains all extraneous data associated with a curve.

```
template <class Base_traits, class Data>
class Arr_curve_data_traits_2 : public Base_traits {
public:
  // (1) Type definitions.
  // (2) Overridden functions.
};
```

The base traits-class must meet all the requirements detailed in Section 2.2 including the definition of the types `Point_2`, `Curve_2`, and `X_monotone_curve_2`. The data traits-class redefines its `Curve_2` and `X_monotone_curve_2` types as derived classes from the respective types in the base traits-class, with an additional *data* field.

```
  // (1) Type definitions:
  typedef Base_traits::Curve_2            Base_curve_2;
  typedef Base_traits::X_monotone_curve_2 Base_x_mon_curve_2;
  typedef Base_traits::Point_2            Point_2;

  class Curve_2 : public Base_curve_2 {
    Data m_data;        // Additional data.
  public:
    Curve_2 (const Base_curve_2& cv, const Data& dat);
    const Data& get_data () const;
    void set_data (const Data& data);
  };

  class X_monotone_curve_2 : public Base_x_mon_curve_2 {
    Data m_data;        // Additional data.
  public:
    X_monotone_curve_2 (const Base_x_mon_curve_2& cv, const Data& dat);
    const Data& get_data () const;
    void set_data (const Data& data);
  };
```

The base traits-class must support all necessary predicates and geometric constructions on curves of the specific family it handles. The data traits-class inherits from the base traits-class all the geometric predicates and some of the geometric constructions of point objects. It only has to override the two functions that deal with constructions of x-monotone curves:

- It uses the `Base_traits::curve_make_x_monotone()` function to subdivide the basic curve into basic x-monotone subcurves. It then constructs the output subcurves by copying the data from the original curve to each of the output x-monotone subcurves.
- Similarly, the `Base_traits::curve_split()` function is used to split an x-monotone curve, then its data is copied to each of the two resulting subcurves.

```
// (2) Overridden functions:
template <class Output_iterator>
void curve_make_x_monotone (const Curve_2& cv,
                            Output_iterator& x_cvs) const;

void curve_split (const X_monotone_curve_2& cv, const Point_2& p,
                  X_monotone_curve_2& c1, X_monotone_curve_2& c2) const;
```

4.1 Constructing the Arrangement Hierarchy

Using the data traits-class with the appropriate parameters, it is simple to implement the arrangement hierarchy (see Section 2.1) without any additional data structures. Given a set of input curves, we construct a planar map that represents their planar arrangement. Since we want to be able to identify the originating input curve of each half-edge in the map, each subcurve we create is extended with a pointer to the input curve it originated from, using the data-traits mechanism as follows: Suppose that we have a base traits-class that supplies all the geometric methods needed to construct a planar arrangement of curves of some specific kind (e.g., segments or conic arcs). We define a data traits-class in the following form:

```
Arr_curve_data_traits_2<Base_traits, Base_traits::Curve_2 *>  traits;
```

When constructing the arrangement, we keep all our base input-curves in a container. Each curve we insert to the arrangement is then formed of a base curve and a pointer to this base curve. Each time a subcurve is created, the pointer to the base input-curve is copied, and can be easily retrieved later.

4.2 An Additional 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.

Using the data traits we can give a straightforward solution to this problem. Suppose that the class `Polyline_traits_2` supplies all the necessary geometric type definition and methods for polylines, fulfilling the requirements of the traits concept. We define the following classes:

```
struct My_data {
  enum {ROAD, RAILROAD, RIVER} type;
  std::string        name;
};
Arr_curve_data_traits_2<Polyline_traits_2, My_data>   traits;
```

Each curve consists of a base polyline-curve (e.g., a road, a river) and a name. We construct the arrangement of all curves in our datasets overlayed on top of the regional map. Then, we can simply go over all arrangement vertices located in the desired region, and examine the half-edges around each vertex. If we find only `ROAD` or `RAILROAD` half-edges around a vertex, we can conclude it represents a crossroad. A vertex where half-edges 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.

5 Experimental Results

As mentioned above, there are many ways to represent line segments in the plane. In the first set of experiments we compared the performance of some representations. When the CGAL Cartesian kernel is used as the template parameter of the traits class (`Arr_segment_traits_2` for example), the user is still free to choose among different number types. We conducted our experiments with (i) `Quotient<MP_Float>`, (ii) `Quotient<Gmpz>`, representing the numerator and denominator as two unbounded integers, (iii) `Gmpq`, GMP's rational class, and (iv) `Leda_rational`, an efficient implementation of exact rationals. In addition, we used an external geometric kernel, called LEDA rational kernel [22], that uses floating-point filtering to speed up computations.

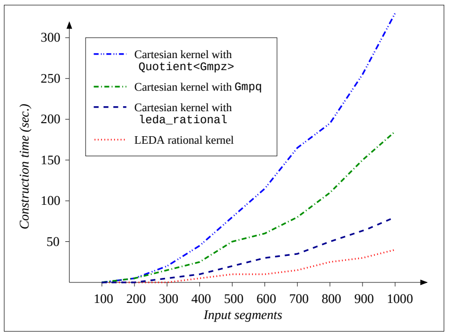
We have tested each representation on ten input sets, containing 100–1000 random input segments, having a quadratic number of intersection points. The results are summarized in Figure 2. The cached traits-class achieves better performance for all tested configurations. Moreover, it is not possible to construct an arrangement using the `Quotient<MP_Float>` number type with the kernel traits (see Section 3.2). For lack of space, we do not show experiments with sparse arrangements.

It is worth mentioning that switching from one configuration to another requires a change of just a few lines of code. In fact, we used a benchmarking toolkit that automatically generates all the required configurations and measures the performance of each configuration on a set of inputs.

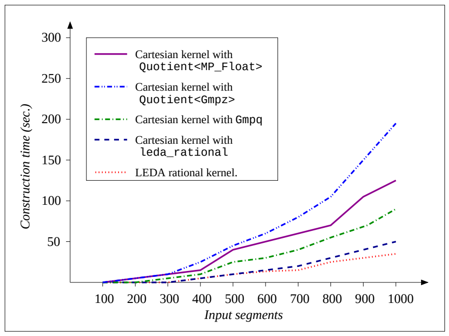
Figure 3 shows the output of the algorithm presented in Section 4.2. The input set, consisting of more than 900 polylines, represents major roads, railroads, rivers and water-canals in the Netherlands. The arrangement construction takes just a few milliseconds.

6 Conclusions and Future Work

We show how our arrangement package can be used with various components and different underlying algorithms that can be plugged in using the appropriate traits class. Users may select the configuration that is most suitable for their



(a)



(b)

Fig. 2. Construction times for arrangements of random line segments: (a) Using the kernel (non-cached) segment traits, (b) using the cached segment traits.

application from the variety offered in CGAL or in its accompanying software libraries, or implement their own traits class. Switching between different traits classes typically involves just a minor change of a few lines of code.

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

In the future we plan to augment the polyline traits-class into a traits class that handles piecewise general curves that are not necessarily linear, and provide means to extend the planar-map point geometric type in similar ways the curve data-traits extends the curve type.

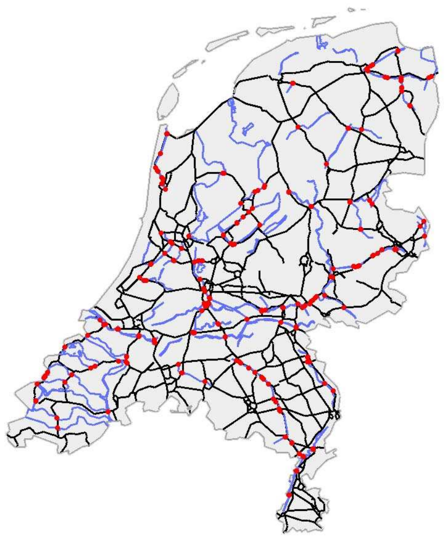


Fig. 3. Roads, railroads, rivers and water canals on the map of the Netherlands. Bridges are marked by circles.

References

1. The CGAL project homepage. `http://www.cgal.org/`.
2. The CORE library homepage. `http://www.cs.nyu.edu/exact/core/`.
3. The EXACUS homepage. `http://www.mpi-sb.mpg.de/projects/EXACUS/`.
4. Generic programming techniques. `http://www.boost.org/more/generic_programming.html`.
5. The GNU MP bignum library. `http://www.swox.com/gmp/`.
6. The LEDA homepage. `http://www.algorithmic-solutions.com/enleda.htm`.

7. D. Cohen-Or, S. Lev-Yehudi, A. Karol, and A. Tal. Inner-cover of non-convex shapes. *International Journal on Shape Modeling*, 9(2):223–238, Dec 2003.
8. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.
9. O. Devillers, A. Fronville, B. Mourrain, and M. Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. *Comput. Geom. Theory Appl.*, 22(1–3):119–142, 2002.
10. D. A. Duc, N. D. Ha, and L. T. Hang. Proposing a model to store and a method to edit spatial data in topological maps. Technical report, Ho Chi Minh University of Natural Sciences, Ho Chi Minh City, Vietnam, 2001.
11. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the Computational Geometry Algorithms Library. *Software — Practice and Experience*, 30:1167–1202, 2000.
12. E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The design and implementation of planar maps in CGAL. *The ACM Journal of Experimental Algorithmics*, 5, 2000. Also in LNCS Vol. 1668 (WAE '99), pages 154–168.
13. E. Fogel et al. An empirical comparison of software for constructing arrangements of curved arcs. Technical Report ECG-TR-361200-01, Tel-Aviv Univ., 2004.
14. B. Gerkey. Visibility-based pursuit-evasion for searchers with limited field of view. Presented in the 2nd CGAL User Workshop (2004).
15. D. Halperin. Arrangements. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. Chapman & Hall/CRC, 2nd edition, 2004.
16. I. Hanniel. The design and implementation of planar arrangements of curves in CGAL. M.Sc. thesis, School of Computer Science, Tel Aviv University, 2000.
17. I. Hanniel and D. Halperin. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *LNCS Vol. 1982 (Proc. WAE '00)*, pages 171–182. Springer-Verlag, 2000.
18. S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *LNCS Vol. 2141 (Proc. WAE '01)*, pages 79–90. Springer-Verlag, 2001.
19. S. Hirsch and D. Halperin. Hybrid motion planning: Coordinating two discs moving among polygonal obstacles in the plane. In J.-D. Boissonnat, J. Burdick, K. Goldberg, and S. Hutchinson, editors, *Algorithmic Foundations of Robotics V*, pages 239–255. Springer, 2003.
20. L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theory Appl.*, 13:65–90, 1999.
21. J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: a library for efficient manipulation of algebraic points and curves. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 360–369, 1999. `http://www.cs.unc.edu/~geom/MAPC/`.
22. K. Mehlhorn and S. Näher. LEDA*: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.
23. N. Myers. Traits: A new and useful template technique. *C++ Gems*, 17, 1995.
24. V. Rogol. Maximizing the area of an axially-symmetric polygon inscribed by a simple polygon. Master's thesis, Technion, Haifa, Israel, 2003.
25. S. Schirra. Robustness and precision issues in geometric computation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1999.
26. R. Wein. High-level filtering for arrangements of conic arcs. In *Proc. ESA 2002*, pages 884–895. Springer-Verlag, 2002.