The Design and Implementation of Planar Maps in CGAL

Eyal Flato, Dan Halperin, Iddo Hanniel, Oren Nechushtan and Eti Ezra
School of Computer Science
Tel Aviv University, Tel Aviv 69978, Israel

Planar maps are fundamental structures in computational geometry. They are used to represent the subdivision of the plane into regions and have numerous applications. We describe the planar map package of CGAL—a Computational Geometry Algorithms Library. We discuss its modular design and implementation. In particular we introduce the two main classes of the design—*planar maps* and *topological maps* that enable the convenient separation between geometry and topology. The modular design is implemented using a generic programming approach. By switching a template parameter—the *geometric traits* class, one can use the same code for planar maps of different objects such as line segments or circular arcs. More flexibility is achieved by choosing a point location algorithm out of three implemented algorithms or plugging in an algorithm implemented by the user. The user of the planar maps package can benefit both from its flexibility and robustness. We present several examples of geometric traits classes and point location algorithms which demonstrate the possibility to adapt the general package to specific needs.

1. INTRODUCTION

Planar maps are used to represent the subdivision of the plane into regions. They are ubiquitous in computational geometry and have numerous applications. Planar maps by themselves can be used to represent geographic maps (or "themed" layers of such maps), the workspace of a robot, the two-dimensional view of a three-dimensional scene, and more. They also serve as fundamental structures on which more involved geometric data structures are constructed.

We start with some basic definitions. A *planar map* is a plane embedding of a planar graph G such that each arc of G is embedded as a bounded curve. The image of a node of G is a *vertex*, and the image of an arc is an *edge*. Each *face* of the map

Work reported in this paper has been supported in part by ESPRIT IV LTR Projects No. 21957 (CGAL) and No. 28155 (GALIA), by the USA-Israel Binational Science Foundation, by The Israel Science Foundation founded by the Israel Academy of Sciences and Humanities (Center for Geometric Computing and its Applications), by a Franco-Israeli research grant "factory of the future" (monitored by AFIRST/France and The Israeli Ministry of Science), and by the Hermann Minkowski – Minerva Center for Geometry at Tel Aviv University.

is a maximally connected region of the plane that does not meet any vertex or edge of the map. The edges of the map are pairwise interior disjoint. We distinguish between planar maps and *arrangements*. An arrangement is a subdivision of the plane induced by a collection of curves that are possibly intersecting. Two of the authors [Hanniel and Halperin 2000] have recently developed an arrangement package on top of the package described in this paper; see Section 7. In this paper we deal with collections of curves that do not intersect in their interior.

We describe the design and implementation of a software package for constructing and efficiently searching in planar maps. Our package is part of CGAL—a Computational Geometry Algorithms Library [CGAL00 2000; Fabri et al. 1996; Fabri et al. 2000], which is a collaborative effort by several academic institutes in Europe and Israel to develop a C++ software library of geometric data structures and algorithms. The library main design goals are robustness, genericity and efficiency. To achieve these goals CGAL adopted the *generic programming paradigm* [Austern 1999; Brönniman et al. 1998].

Our CGAL planar map package is designed for general curves. We tried to define the minimal geometric interface that will enable the construction and handling of a geometric map. Packaging those predicates and functions under one *geometric traits* class [Brönniman et al. 1998; Fabri et al. 2000] helped us achieve the following goals: flexibility in choosing the geometric kernel and number representation, ability to have several strategies for robustness, and extendibility to maps of objects other than line segments. Our package can be used with any family of curves as long as the user supplies a small set of operations for the family.

We are not aware of any publicly available software for representing planar maps of general curves and which supports efficient access to regions of the map as ours does. The MAPC library [Keyser et al. 1999] has been applied to construct the arrangement of algebraic curves. However, this package is a library for manipulating algebraic points and curves and not a planar map implementation. Using the traits mechanism, incorporating the MAPC representations into our package is possible. LEDA, the Library of Efficient Data Structures and Algorithms [Mehlhorn et al. 1997; Melhorn and Näher 1999], provides tools for constructing planar maps of straight line segments, and for efficient point location on such maps.

In the next section we review the basic terminology that we will be using in the paper and present a sketch of the design of the package. Section 3 presents the two key classes of the package *planar maps* and *topological maps*, which enable the convenient separation between geometry and topology of the maps. The notion of geometric traits is explained in Section 4 together with examples of specific traits that we have implemented. In Section 5 we present the point location interface along with the three point-location algorithms that we supply with the package. We present experimental results in Section 6. Concluding remarks and directions for further work are given in Section 7.

2. PRELIMINARIES AND DESIGN OVERVIEW

In this section we provide more details on the terminology and tools that we will be using throughout the paper. We also give a brief overview of the package's structure.

Our package consists of an implementation of algorithms and data structures for

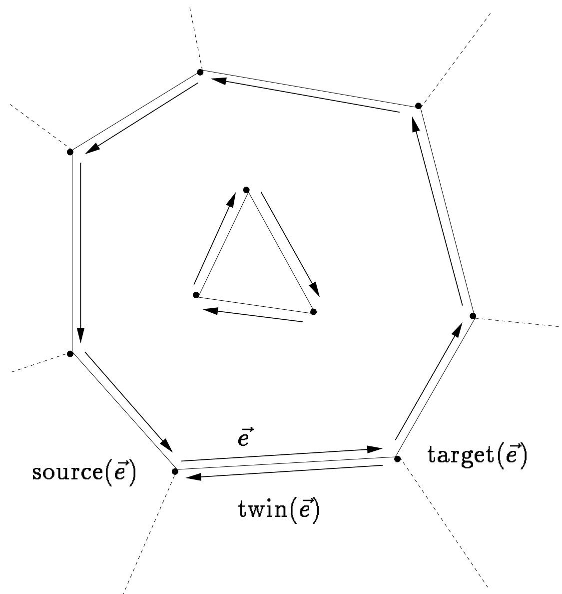

Fig. 1. DCEL — source and target vertices, and a twin halfedge in a face with a hole

constructing and accessing planar maps. The data structures support traversal over faces, edges and vertices of the map, traversal over a face and around a vertex, and efficient point location.

The basic representation that we use is the *Doubly Connected Edge List* (DCEL) structure. This representation belongs to a family of edge-based data structures in which each edge is represented as a pair of opposite *halfedges*; see e.g., [de Berg et al. 1997; Kettner 1999; Mäntylä 1988; Weiler 1985]. Consider Figure 1 for an illustration. A halfedge $\vec{e}$ connects two vertices—its endpoints: it is directed from its *source* endpoint to its *target* endpoint. Its *twin* halfedge, twin($\vec{e}$), connects the same endpoints in the reverse order. We regard a halfedge as bounding the face on its left-hand side. The DCEL is augmented to support inner components (holes) inside the faces making it more general and more suitable for a wide range of applications (e.g., geographical information systems) than a representation that requires that each face of the map be simply connected. For more details on our variant of the DCEL see [de Berg et al. 1997, Chapter 2] and the CGAL documentation [CGAL00 2000].

We do not restrict the planar maps that we support to consist of straight line segments; see for example Figure 8 which depicts a map of circular arcs constructed by our software. Our package can be used with any collection of bounded x-monotone curves. We define an x-monotone curve (x-curve for short) to be a curve that is either a vertical line segment or a Jordan arc such that any vertical line intersects it in at most one point. The requirement that the curves be x-monotone was aimed to simplify the algorithms and data structures that we implemented at this fundamental level. In a more general package for two-dimensional arrangements built on top of the package that we describe here, we relaxed this constraint; see Section 7.

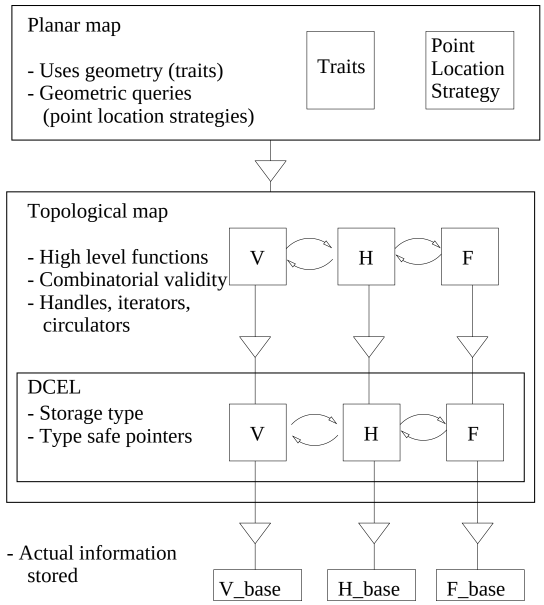

Fig. 2. Design overview. V, H, and F stand for vertex, halfedge, and face, respectively.

The package is composed of two main classes: `Topological_map` and `Planar_map_2`. The two classes are described in detail in Section 3, together with a discussion of what distinguishes between them and how they interact with each other. We first explain the choice of names for the classes. Most algorithms and data structures in computational geometry need to work in tandem on two types of information: (i) numeric (or geometric) and (ii) topological (or combinatorial). For example, when we represent a simple polygon, supplying only the coordinates of its vertices (numeric data) will not suffice for most purposes, and we also have to specify how the vertices connect to form the polygon. We refer to the latter as *topological* information. Sometimes this type of data is called *combinatorial*. Since the basic non-numeric data that we need in order to manipulate maps regard adjacency and incidence of map features, we chose to name the corresponding class `Topological_map`, a title which we believe reflects the role of the class fairly well. The same is true for our reference to the numeric and algebraic data—point coordinates and curve equations—as *geometric*.

This duality of information (geometric and topological) is the source of much of the difficulty in implementing geometric algorithms (see, e.g., [Guibas 1996]). Separating these two types of data makes the package modular and convenient to extend, as we show in the subsequent sections.

Before we delve into the underlying structures and algorithms, we give a brief overview of the package. Figure 2 depicts an outline of the package. The design follows CGAL's polyhedron design introduced in [Kettner 1999]. The bottom layer holds base classes for vertices, halfedges and faces. Their responsibilities are the actual storage of the incidence relations between the map features, the geometry and other attributes. The middle layer is the topological map. This layer uses the DCEL as its data structure. The DCEL is used as a container that stores the objects of the bottom layer and adds functionality for manipulating them (for example, for traversing all the halfedges around a face). The top layer, the planar map layer, adds geometry to the topological map layer. This is accomplished using a `Traits` template parameter (Section 4). We provide high-level functionality (e.g., simple insertion operation) based on the geometric properties of the objects. Geometric queries — point location and vertical ray shooting — are also introduced in this class (see Section 5).

We defer further explanation of the terminology that we use in Figure 2 to the following sections.

3. TOPOLOGICAL MAPS AND PLANAR MAPS

The planar map package is implemented with two major classes: the `Topological_map` (`TPM`) class, and the geometric `Planar_map_2` (`PM`) class which is derived from `TPM`. The topological class consists of vertices, halfedges, and faces which we refer to as *t-vertices*, *t-halfedges*, and *t-faces*. It supports traversal of t-halfedges along a t-face boundary, traversal of t-halfedges around a t-vertex, and finding a neighboring t-face. The geometric class consists of numeric and algebraic information like point coordinates and curve equations. We maintain this information in the planar map objects: *pm-vertices*, *pm-halfedges* and *pm-faces*. This class supports geometric functionality such as point location.

The planar map is an embedding of the topological map in the plane. The basic design of the topological map restricts it to topological spaces homeomorphic to the plane, and there it can only handle finite geometric objects. We also have additional machinery (which is not described in this paper) which enables the handling of infinite curves.

In what follows, to distinguish the different types, we will use the prefix *t-* for the topological map objects (e.g., t-vertex) and the prefix *pm-* for the planar map objects (e.g., pm-vertex). Note that the source code does not include these prefixes.

3.1 Topological Map

The topological map class is a base class for two-dimensional subdivisions. It consists of t-vertices, t-halfedges, and t-faces and an incidence relation on them. Each topological edge of the subdivision is represented by two t-halfedges with opposite orientations. A valid topological map always has one *unbounded t-face* (since we only handle finite objects). A CCB (Connected Component of the Boundary) is a cycle[1] of at least two t-halfedges. Each t-face can have one outer boundary which we refer to as its *outer CCB*, and a set of t-holes which we call the *inner CCB's*. A t-face is defined by its inner and outer CCB's. An empty map has one unbounded t-face (and no t-halfedges nor t-vertices).

A containment relationship between a t-face and its inner t-holes is a topological characteristic which distinguishes the topological map from standard graph structures, and from other edge-based structures. This enables us to derive subdivisions

[1] *cycle* is used here in a graph theoretic sense—a cyclic sequence of t-halfedges in which each t-halfedge points to its successive t-halfedge along the boundary of the face that they both bound.

with holes in them.

The topological map does not include geometric information and therefore it is suitable as the basis for different types of two-dimensional subdivisions. In this paper we present the implementation of the 2-dimensional planar maps but we can also derive other subdivisions, for example, a two-dimensional map on an xy-monotone three-dimensional surface—we regard this capability as provisory for implementing three-dimensional subdivisions induced by algebraic surfaces. It can also be used almost as it is as a representation class for polyhedral terrains or polygons with holes, by merely adding point coordinates as additional attributes to the vertices.

From each t-vertex we can retrieve all the t-halfedges that are incident to it. A t-halfedge points to its source and target t-vertices, its incident t-face, its twin t-halfedge, and the next t-halfedge on its CCB. A t-face points to a t-halfedge on its outer boundary (if it exists) and holds a list of its t-holes. Beside the incidence pointers, the `Topological_map` class supports the following update methods:

```
t-halfedge insert_at_vertices(t-halfedge, t-halfedge)
t-halfedge insert_from_vertex(t-halfedge)
t-halfedge insert_in_face_interior(t-face)
t-halfedge split_edge(t-halfedge)
t-halfedge merge_edge(t-halfedge, t-halfedge)
t-face remove_edge(t-halfedge)
bool move_hole(t-halfedge, t-face, t-face)
```

When we apply an insert function, we create the twin t-halfedges, the necessary t-vertices (for example, the missing vertex in the case of `insert_from_vertex`), and possibly a new t-face. We update the pointers of the relevant CCB's locally. (Note that the insert functions operate on the t-vertices at the targets of the input t-halfedges.) When we apply `split_edge` we create two twin t-halfedges and a t-vertex. The functions `merge_edge` and `remove_edge` delete two twin t-halfedges and possibly one or two t-vertices. When `remove_edge` removes a t-edge that separates two t-faces, a t-face is removed from the map as well. The `move_hole` function does not create or delete features from the map and only updates the holes lists of two t-faces.

The following simple function `topological_triangle()` demonstrates the use of `Topological_map`. It creates an empty map (with one t-face corresponding to the unbounded t-face) and then inserts a t-edge `e1` inside the unbounded t-face. (The term t-edge is an abstract term for brevity. Our insertion functions indicate where an edge needs to be inserted: in face interior, from a vertex, between two vertices. As the result of an insert function, a pair of t-halfedges are created, together with the necessary vertices and faces.) It then inserts a t-edge `e2` from the target t-vertex of `e1` and finally inserts a t-edge between the target t-vertices of `e2` and `e1->twin()`, closing a "topological" triangle (i.e., a closed cycle of three t-vertices without coordinates).

Note that the types that are used in the code for t-vertex, t-halfedge, and t-face are `Vertex`, `Halfedge`, and `Face` and the pointers are `Vertex_handle`, `Halfedge_handle` and `Face_handle`. As noted before, the DCEL is a container for the features of the map and therefore is templated with the base classes for these features: t-vertex, t-halfedge, and t-face.

```
using namespace CGAL;
typedef Pm_dcel<Tpm_vertex_base,Tpm_halfedge_base,Tpm_face_base> Dcel;
typedef Topological_map<Dcel> Tpm;

void topological_triangle() {
  Tpm t;
  Tpm::Face_handle uf = t.unbounded_face();
  Tpm::Halfedge_handle e1 = t.insert_in_face_interior(uf);
  Tpm::Halfedge_handle e2 = t.insert_from_vertex(e1);
  t.insert_at_vertices(e2,e1->twin());
}
```

Addition of attributes to the features of the topological map is made easy by the use of generic programming. The following example demonstrates how to add an attribute (in this case some `Point` type) to a t-vertex of a map. It creates a new vertex type `My_vertex` that derives from `Tpm_vertex_base` and adds the attribute. The new vertex is then passed as a template parameter to the `Dcel`. After the insertion of the new t-edge the information in its incident t-vertices can be updated by the user. This can be used, for example, as a representation class for polygons with holes. The `insert_with_info` function below adds such a polygon (Figure 3) to a topological map.

```
using namespace CGAL;
struct My_vertex : public Tpm_vertex_base {
   Point pt;
};
typedef Pm_dcel<My_vertex, Tpm_halfedge_base,Tpm_face_base> Dcel;
typedef Topological_map<Dcel> Tpm;

void insert_with_info() {
    Tpm t;
    Tpm::Face_handle uf = t.unbounded_face();
    Tpm::Halfedge_handle e1 = t.insert_in_face_interior(uf);
    e1->source()->pt = Point(0,0);
    e1->target()->pt = Point(10,0);
    Tpm::Halfedge_handle e2 = t.insert_from_vertex(e1);
    e2->target()->pt = Point(5,9);
    Tpm::Halfedge_handle e3 = t.insert_at_vertices(e2,e1->twin());
    Tpm::Halfedge_handle e4 = t.insert_in_face_interior(e1->face());
    e4->source()->pt = Point(3,2);
    e4->target()->pt = Point(7,2);
    Tpm::Halfedge_handle e5 = t.insert_from_vertex(e4);
    e5->target()->pt = Point(5,7);
    Tpm::Halfedge_handle e6 = t.insert_at_vertices(e5,e4->twin());
}
```

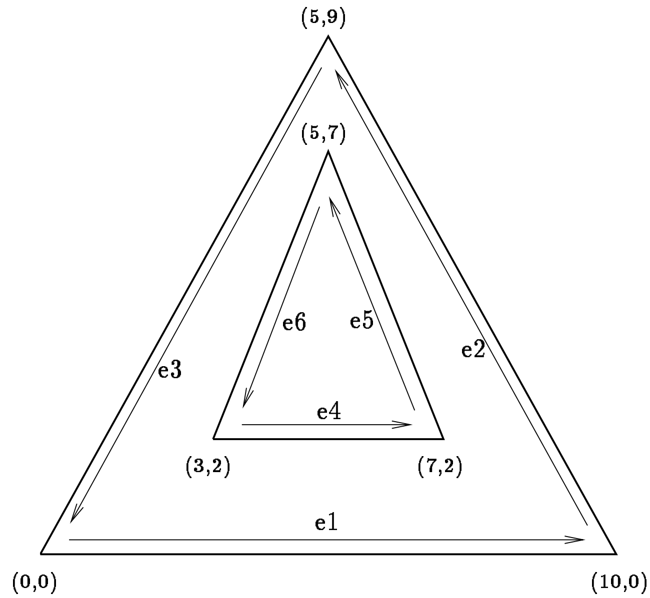

Fig. 3. Adding attributes to a topological map: A polygon with a hole built by the example function *insert_with_info*

3.2 Planar Map

The planar map is a 2-dimensional subdivision of the plane. The `Planar_map_2` class is derived from `Topological_map` and represents an embedding of a topological map in the plane such that each t-edge is embedded as a bounded x-monotone curve and each t-vertex is embedded as a planar point. In this embedding no pair of edges intersect except at their endpoints.

The geometric traits class (described in Section 4) defines the geometric objects: the point and the x-curve and supplies predicates on them. `Planar_map_2` augments the topological map it is derived from with this geometric information. The planar map is implemented using only the topological properties of the topological map and the geometric definitions and predicates of the geometric traits.

The pm-vertex class extends the t-vertex class by adding a point representation. The pm-halfedge class extends the t-halfedge class by adding an x-curve. The additional geometric information enables geometric queries as well as an easier interface that uses the geometry. Here is a list of the main methods of `Planar_map_2`:

```
pm-halfedge insert(X_curve)
pm-halfedge insert_at_vertices(X_curve, pm-vertex, pm-vertex)
pm-halfedge insert_from_vertex(X_curve, pm-vertex, bool source)
pm-halfedge insert_in_face_interior(X_curve, pm-face)
pm-halfedge split_edge(pm-halfedge, X_curve, X_curve)
pm-halfedge merge_edge(pm-halfedge, pm-halfedge, X_curve)
pm-face remove_edge(pm-halfedge)
```

```
pm-halfedge locate(point, locate_type)
pm-halfedge vertical_ray_shoot(point, locate_type, bool up)
```

Using the planar map class an x-curve can be inserted into the map without specifying where it should be placed (`PM::insert`). The planar map uses the geometry to perform point-location queries (`PM::locate`) with the x-curve's endpoints and then uses one of the specific insertion functions (`PM::insert_at_vertices`, `PM::insert_from_vertex`, or `PM::insert_in_face_interior`). The other methods are overloaded at the planar map layer from the topological layer and get a different set of parameters. We discuss the differences among the planar map methods and the topological map methods in the following subsection.

Although the `insert` function includes the functionality of the other three insertion functions, we kept all of them in the interface. If one has information about where an x-curve is located it is more efficient to execute a specific insertion function instead of the simple one (`insert`) because by that we avoid performing unnecessary point-location queries which may be highly time consuming.

The following function `geometric_triangle()` creates an empty map and then inserts the three edges of the triangle into it. It resembles the example of `topological_triangle()` of the previous section and shows the difference between the interface of the topological map and that of the planar map. We use the traits class `Pm_segment_exact_traits` that handles segments with the homogeneous point representation using the machine integer number type `long` (`Homogeneous<long>`). The DCEL that is used is the default for planar maps (`Pm_default_dcel`). The DCEL includes the point and x-curve information for each vertex and halfedge that it stores.

```
using namespace CGAL;
typedef Homogeneous<long> Rep;
typedef Pm_segment_exact_traits<Rep> Traits;
typedef Pm_default_dcel<Traits> Dcel;
typedef Planar_map_2<Dcel,Traits> Pm;

void geometric_triangle() {
   Pm p;
   Traits::Point p1(0,0), p2(1,0), p3(0,1);
   Traits::X_curve cv1(p1,p2), cv2(p2,p3), cv3(p3,p1);
   p.insert(cv1);
   p.insert(cv2);
   p.insert(cv3);
}
```

Geometric Tools Used by the Planar Map. The planar map needs several geometric operations in order to make its branching decisions. Beside access to points and curves, we organized the rest of the geometric operations in three procedures. These procedures are part of the implementation of the planar map but they are not publicly exposed:

Locate around vertex Given a pm-vertex v and an x-curve c from it, we would like to find the first pm-halfedge incident to v that is clockwise before c. Let h,

h_1 be two successive pm-halfedge emanating form v. If c is clockwise between h and h_1 then h is immediately clockwise before c around v. To find the previous pm-halfedge we iterate over all the pm-halfedges around v and apply the traits function `curve_is_between_cw` on each pair of successive pm-halfedges. We use this procedure to find the *previous* pm-halfedge of an inserted x-curve (see Section 3.3.1).

Point inside face Given a point p and a pm-face f we should find whether p is inside f. This is done using a standard algorithm. Conceptually, we shoot a ray from p vertically upwards and count the number of pm-halfedges of the boundary of f that intersect it. If the number is odd the point is inside f. In practice, we do not need to implement an intersection function (which can be expensive for some curves), we just need a function that defines if a curve is above or below a point. We then go over all the pm-halfedges on the boundary of f and use this function to count how many of them are above p (some care is needed so that the counting gives the desired answer in the presence of degeneracies).

Counterclockwise cycle Given a cycle (CCB) in a planar map we would like to find out whether it is counterclockwise ordered. Let $\bar{v}$ be the left-lower-most pm-vertex in the cycle and let h be the pm-halfedge of the cycle whose target is $\bar{v}$ and h_1 the pm-halfedge with $\bar{v}$ as its source. Then, if h is above h_1 immediately to the right of $\bar{v}$ the cycle is counterclockwise ordered.

These three procedures give an idea of how to encapsulate the geometric representation and predicates. We elaborate on this issue in Section 4.

3.3 Separating the Topological Layer from the Geometric Layer

As stated before, the separation between the topological and geometric layer was meant to enable more flexibility (e.g., to enable implementing a terrain using the same topological map class). When designing the topological map class, no geometric considerations could be used. This imposes constraints on the functions and function parameters of the topological map. For some functions (e.g., the `split_edge` function), the topological information suffices, while for others additional topological information is needed to avoid ambiguity. In the planar map layer this information can be deduced from the geometric information. In the remainder of this section we discuss the difference between the interfaces of the two layers and show algorithmic solutions for some design challenges raised by the separation.

3.3.1 *The Use of Previous Halfedge.* When inserting a new t-edge from a t-vertex v using the function `TPM::insert_from_vertex`, passing only v to the function will cause ambiguity. Figure 4 shows an example of two possible t-edges that can be inserted if we had only passed the vertex. Topologically, what defines the insertion from a t-vertex uniquely is the *previous* t-halfedge to the edge inserted. Therefore, this is what is passed in our topological function. In the geometric layer, passing the pm-vertex v is sufficient, since we can find the previous t-halfedge geometrically, using the procedure **locate around vertex**—Section 3.2.

3.3.2 *The Use of the* TPM::move_hole *Function.* When inserting a new t-edge between two t-vertices *v1* and *v2*, a new t-face might be created. Figure 5 shows

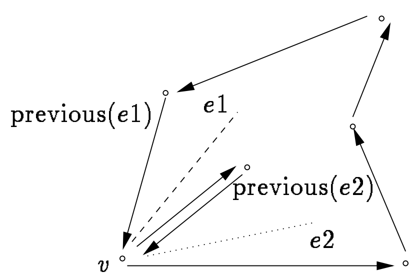

Fig. 4. Previous t-halfedges are necessary for *TPM::insert_from_vertex* : Passing only *v* as a parameter to the insertion function cannot resolve whether *e1* or *e2* is the t-edge to be inserted into the map

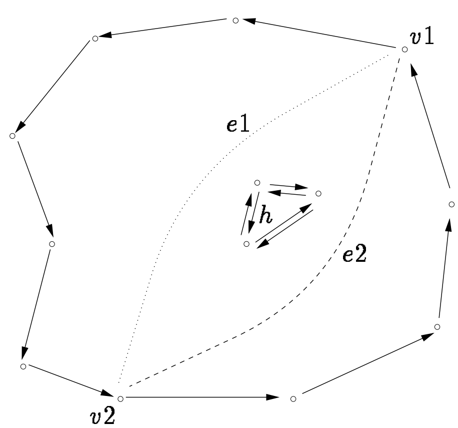

Fig. 5. The need for the *TPM::move_hole* function: After inserting a new t-edge between *v1* and *v2* the original t-face is split into two t-faces. Since we have no knowledge of the geometry of the curves in the map, we cannot determine topologically which of the two t-faces contains the hole *h*.

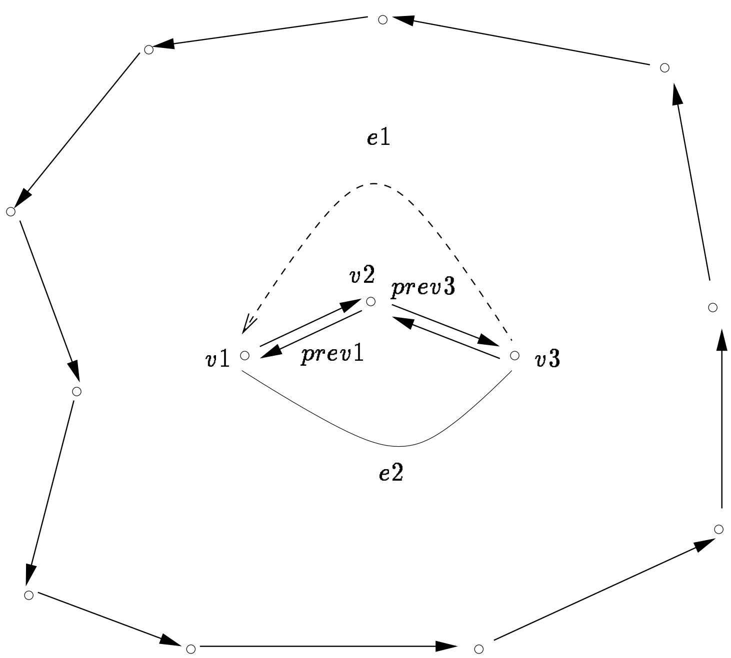

Fig. 6. After inserting a new edge between *v1* and *v3* a new face is created inside the original face. If *e1* is the new edge inserted, the new face corresponds to the ordered list of vertices *v1,v2,v3,v1*; if *e2* is inserted then it corresponds to *v1,v3,v2,v1*

an example of two possible edges that can be inserted into a topological map. If *e1* is inserted, the t-hole h should be in the right t-face, whereas if *e2* is inserted the t-hole h should be in the left t-face. The topological map has no geometric knowledge of the curves, so it cannot determine which of the two t-faces contains the t-hole. Therefore, this is done by the client—the planar map, which calls the function `TPM::move_hole` if a t-hole needs to be moved to a new t-face created after insertion.

In the planar map class we go over the holes of the original pm-face and check if they are inside the new pm-face that was created. If so the pm-hole is moved to the new pm-face using the `TPM::move_hole` function. We check whether a pm-hole is inside a pm-face by checking if one of its pm-vertices is inside the pm-face, using the procedure **point inside face**—Section 3.2.

3.3.3 *Defining the Incident Face of Inserted Halfedges.* Using `TPM::insert_at_vertices` we insert a new t-edge between two given t-vertices v_1 and v_2 that are inside a t-face f. The new inserted t-edge has two corresponding t-halfedges: h_1 is directed from v_1 to v_2 and h_2 is directed from v_2 to v_1. If h_1 and h_2 close a cycle of t-halfedges then a new t-face f_{new} should be created as a t-hole of f. After creating f_{new} we cannot determine topologically whether the cycle of t-halfedges that is the outer boundary of f_{new} is the cycle that includes h_1 (and h_2 is included in an inner cycle of f) or whether it includes h_2 (and h_1 is on an inner cycle of f).

First, similar to the solution we mentioned in Section 3.3.1, we pass the previous t-halfedges $prev_1$ and $prev_2$ to `TPM::insert_at_vertices` instead of the t-vertices themselves. The order of the parameters is also important to the topological function. The topological map always assigns f_{new} as the face of the halfedges on the cycle that is directed from the target vertex of $prev_1$ to the target vertex of $prev_2$ (in our case it is h_1).

In the planar map layer the method `PM::insert_at_vertices` does not expect to get an ordered pair of pm-vertices. Therefore, the planar map layer (as the client of the topological map) should find out whether the cycle that corresponds to the new face includes the pm-halfedge from v_1 to v_2 or its opposite pm-halfedge. The planar map must also decide this before we call `TPM::insert_at_vertices` to update the topological structure. Since h_1 and h_2 are not yet in the map we simulate a walk on the cycle that will contain h_1 and find whether its direction is counterclockwise, using the procedure **counterclockwise cycle**—Section 3.2. If so, we call `TPM::insert_at_vertices(prev1,prev2)` otherwise we call `TPM::insert_at_vertices(prev2, prev1)`.

Figure 6 shows an example of two possible t-edges that can be inserted between the t-vertices *v1* and *v3*, and the different t-faces they induce. If *e1* is inserted, the t-halfedge on the outer boundary of the new t-face corresponds to the ordered pair $(v3, v1)$, whereas if *e2* is inserted, the t-halfedge in the new t-face corresponds to the ordered pair $(v1, v3)$, assuming the convention of a t-face being to the left of a t-halfedge on its boundary.

3.3.4 *Summary of Interfaces Differences.* Although both classes, the topological map and the planar map, have the three insertion functions `insert_at_vertices`, `insert_from_vertex`, and `insert_in_face_interior`, their parameter lists are different. By passing x-curves to the planar map versions of the functions we supply the required geometric information. The planar map should also geometrically determine some topological ambiguities as described in Sections 3.3.1 and 3.3.3. We explained the need for `TPM::move_hole` in Section 3.3.2.

The rest of the updating functions (namely, `split_edge`, `merge_edge`, and `remove_edge`) are simply the same in the topological and geometric layer. Here, the planar map only adds an update of the geometric data of its features. Note that each geometric (`PM`) version of the updating functions uses a call to its topological counterpart (`TPM`).

The most important addition of the planar map layer is the method `PM::insert` that gets only an x-curve as a parameter. This method performs point location queries (described in Section 5) to locate both ends of the inserted curve and then calls one of the other update functions (`PM::insert_at_vertices`, `PM::insert_from_vertex`, or `PM::insert_in_face_interior`). Although one can call the specific insertion functions (if one has the required information at hand), the `PM::insert` method enables the user of the planar map to easily update the map without worrying about geometric or topological decisions.

4. GEOMETRIC TRAITS

The geometric traits class is an abstract interface of predicates and functions that wraps the access of the algorithms in our package to the geometric (rather than the topological) inner representation[2].

We tried to define the minimal geometric interface that will enable the construction and handling of a geometric map. Packaging those predicates and functions under one traits class helped us achieve the following goals: flexibility in choosing the geometric representation of the objects (Homogeneous, Cartesian); flexibility in choosing the geometric kernel (LEDA, CGAL, or a user-defined kernel); ability to have several strategies for robustness; extendibility to maps of objects other than line segments.

The documentation of the planar map class [CGAL00 2000] gives the precise requirements that every traits class should obey. We have formulated the requirements to minimize the assumptions on the curves. This enables the users to define their own traits classes for different kinds of x-monotone curves that they need for their applications. The only restriction is that they obey the predefined interface.

The planar map traits class defines the basic objects of the map: the point (`Point`) and the x-monotone curve (`X_curve`). In addition four types of predicates are needed (we give here the full list to demonstrate the compactness of the interface):

(i) Access to the endpoints of the x-monotone curves:

```
Point curve_source(X_curve c)
```
returns the source point of c.

```
Point curve_target(X_curve c)
```
returns the target point of c.

(ii) Comparison predicates between points:

```
int compare_x(Point p1, Point p2)
```
compares the x-coordinates of the input points.

```
int compare_y(Point p1, Point p2)
```
compares the y-coordinates of the input points.

(iii) Comparison between points and x-curves

```
int curve_get_point_status(X_curve c, Point p)
```
returns a constant describing the relative position between the x-curve and the point: p is above c, p is below c, or p is on c.

(iv) Predicates between x-curves

```
int curve_compare_at_x(X_curve c1, X_curve c2, Point p)
```
compares the y-coordinate of the curves at the x-coordinate of p.

[2] The original usage of the term "traits class" is for associating related type information to built-in types [Myers 1995]. In CGAL, this term is used in a more general sense [Brönniman et al. 1998; Fabri et al. 2000].

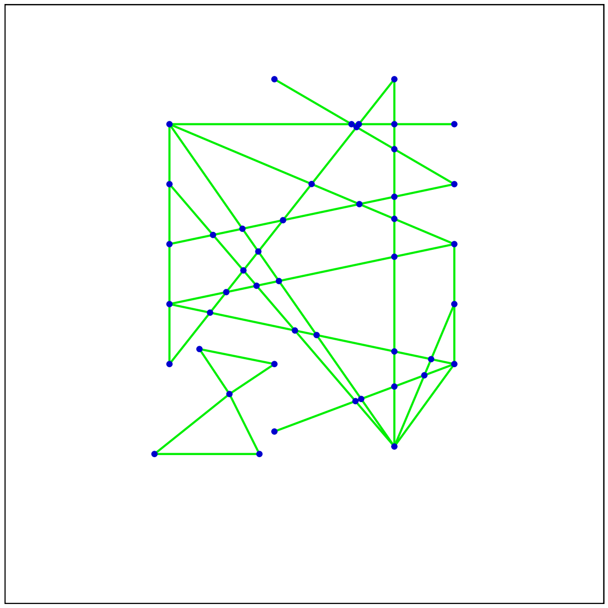

Fig. 7. A planar map of segments created with a segment traits class

`int curve_compare_at_x_left(X_curve c1, X_curve c2, Point p)`
compares the y-coordinate of the curves immediately to the left of the x-coordinate of p.

`int curve_compare_at_x_right(X_curve c1, X_curve c2, Point p)`
compares the y-coordinate of the curves immediately to the right of the x-coordinate of p.

`bool curve_is_between_cw(X_curve c, X_curve c1, X_curve c2, Point p)`
given three curves that share an endpoint p, returns true if c is between $c1$ and $c2$, when proceeding in the clockwise direction around p.

The predicates above satisfy the geometric needs of the planar map. We use the access and comparison functions (types (i) and (ii)) to update the geometric information of the planar map.

In Section 3.2 we presented the need for more elaborate geometric predicates. The abstract description in that section translates to the following concrete predicates of types (iii) and (iv). In the procedure **point inside face** we use `curve_get_status`; `curve_is_between_cw` is used in **locate around vertex**, and comparison between x-curves, e.g., `curve_compare_at_x_right`, are used by **counterclockwise cycle** procedure.

Following these interface specifications we implemented several traits classes. Here are a few examples:

`Pm_segment_exact_traits<R>` A class for planar maps of line segments that uses CGAL's kernel [Fabri et al. 1996]. The `R` template parameter enables the use of CGAL's homogeneous or Cartesian kernel. This class is robust when used with

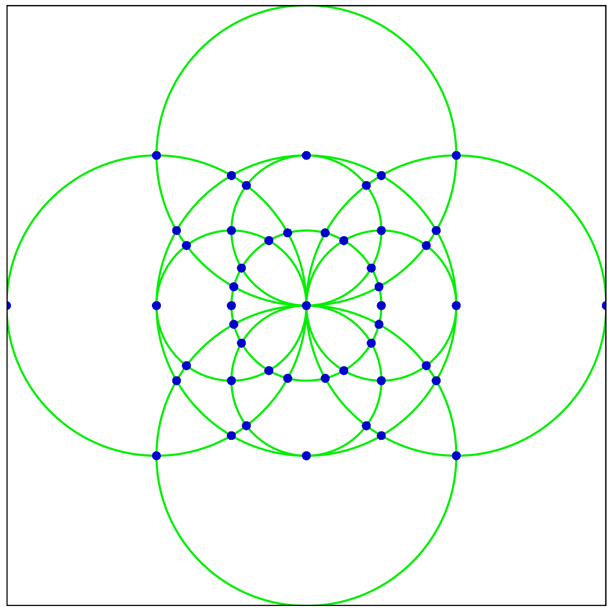

Fig. 8. A planar map of circle arcs created with the circles traits class

exact arithmetic. Figure 7 shows a planar map of segments (the source code and program can be found in the planar map demo in the CGAL distribution).

`Pm_leda_segment_exact_traits` A class for planar maps of line segments that uses LEDA's rational geometry kernel[3][Melhorn and Näher 1999]. Consequently, the predicates become faster. One of the differences that makes this traits class more efficient is the use of LEDA's primitive predicates (e.g., orientation) that are implemented using floating point "filters" [Fortune and van Wyk 1996; Schirra 1999] which speed up the exact computation. This traits class could not be made a special case of `Pm_segment_exact_traits<R>` since the usage of the more efficient LEDA rational kernel could not simply be parametrized as a representation type.

`Arr_circles_real_traits` A class for planar maps of circular arcs that uses the LEDA *real* number type (to support robust square-root predicates). Figure 8 shows an example of a planar map that uses this traits class. In a preprocessing step we cut the circles into x-monotone arcs and find the intersection points among them. Then we insert the intersected arcs into the planar map.

A planar map is initialized with a single traits class which cannot be modified later. A traits class for conics is currently under development. It will enable to use segments and circular arcs (and other conic arcs) in the same map.

[3]The reader should not confuse the *leda_rational* number type with LEDA's rational geometry kernel. The latter uses a different representation, employs floating point filters, and is used by us in the LEDA traits.

5. POINT LOCATION

When we described the planar map class in Section 3.2 we mentioned that it answers point-location queries. In this section we discuss the interface between the point-location algorithms and the planar map and list the different implementations of point-location algorithms.

Given a planar subdivision S and a point p the result of a point-location query is the feature of S in which p lies. In our case, the function `PM::locate` returns a pm-halfedge and a location type indicator. If p is on a pm-vertex then a pm-halfedge whose target is this pm-vertex is returned. When p is on a pm-halfedge h, h is returned. When p is in the interior of a pm-face f, a pm-halfedge on f's boundary is returned. It is also possible to perform a point-location query on an empty map — in such case null is returned. An extra indicator distinguishes among the different cases. The planar map also answers vertical ray-shooting queries. The function `PM::vertical_ray_shoot` returns the pm-halfedge that is immediately above (or below) the query point p.

The design of the package separates between the planar map class and the point-location algorithm. The separation is done using a mechanism called the *strategy pattern* [Gamma et al. 1995]. The strategy pattern is used to define a family of algorithms, encapsulate each one, and make them interchangeable. The strategy mechanism lets the algorithms vary independently from clients that use them.

The interface between the planar map and the point-location algorithm is defined by the abstract strategy class `Pm_point_location_base <Planar_map>`. When creating a planar map instance, an instance of a *concrete* point-location strategy is passed and kept by the planar map during its lifetime. The concrete strategy is derived from the abstract class and implements the algorithm interface.

When a point-location query is performed the planar map calls the appropriate point-location class methods. The point-location class uses the planar map interface to answer the query:

```
pm-halfedge locate(point, locate_type)
pm-halfedge vertical_ray_shoot(point, locate_type,  bool up)
```
the last parameter indicates whether we shoot the ray upwards or downwards

Since we would like to have point-location algorithms that have their own data we should be able to update these internal data structures. The planar map is calling the updating functions of the point-location class whenever the map is updated: `insert`, `split_edge`, `merge_edge`, and `remove_edge`.

We have implemented three concrete strategies: `Pm_naive_point_location`, `Pm_walk_point_location`, and `Pm_ric_point_location`:

Naive The naive algorithm tests all the edges in the map, and finds the one that is above the query point and has the smallest vertical distance from it. When the distance is zero then the query point is on a vertex or an edge, and we report the edge that we have found. Otherwise, if the query point is in the interior of a face, then the edge that we have found is on its boundary. Clearly, the time complexity of a query with the naive class is linear in the complexity of the planar map.

Walk Like the naive algorithm the walk algorithm does not need to maintain extra data structures for its operation. However the walk and naive approaches have one crucial difference. In the walk algorithm the search is limited to the *zone* in the planar map of a vertical ray emanating from the query point (the *zone* of a curve γ in a planar map is the collection of faces of the map intersected by γ). This decreases the number of edges visited during the query, thus improving the time complexity in practice. Note that the worst-case time complexity remains linear.

RIC The RIC (Randomized Incremental Construction) algorithm is an implementation of the dynamic algorithm introduced by Mulmuley [Mulmuley 1990]. Unlike the naive or walk algorithms, this approach requires the maintenance of additional data structures. The implementation consists of two structures: (i) a *trapezoidal map* and (ii) a *search structure*—the *history DAG* (Directed Acyclic Graph). We support insertions and deletions of map edges, while maintaining an efficient point-location query time and linear storage space. This is achieved by a "lazy" approach that performs an occasional rebuilding step whenever the history DAG passes predefined thresholds in size or in depth. The rebuilding step is an option that can be fine-tuned or disabled by the user. The trapezoidal map is a "uniform" collection of trapezoids, where each trapezoid corresponds to a subset of the plane bounded above and below by curves and from the sides by *vertical attachments*; see [de Berg et al. 1997, Chapter 6] for more details. The RIC algorithm is the default point-location strategy of planar maps in our package.

Each algorithm is preferable in different situations where the main trade-off between the RIC algorithm and the other two is between time and storage. The naive and walk algorithms need more time but almost no additional storage. Section 6 describes experiments with the different algorithms.

The above three algorithms demonstrate one aspect of the modularity of our package. It is possible to implement other strategies such as an algorithm that uses simple bucketing for performing point locations, which can be practically very efficient for some applications. Although the design of the planar map makes it quite easy to plug-in a point location algorithm, the algorithm might be very complicated to implement when aiming for robustness and generality. Our implementation of the RIC algorithm, for example, includes dozens of inner classes and thousands of lines of code that handle all kinds of degeneracies and robustness issues.

6. EXPERIMENTAL RESULTS

We tested the performance of the planar map package. The choice of a point-location algorithm and a traits class plays a crucial role in the performance of the package. Therefore, we experimented with the various point-location strategies and different traits classes.

We experimented with the following types of data sets. The *random* input set is built of random segments in a rectangle, intersected so that the resulting sub-segments are pairwise interior disjoint (Figure 9). The *robotics* (motion planning) input is constructed by the segments of an arrangement underlying the Minkowski sum (vector sum) of a star-shaped robot and a set of obstacles (Figure 9) produced by the program described in [Agarwal et al. 2000; Flato and Halperin 2000]. Each

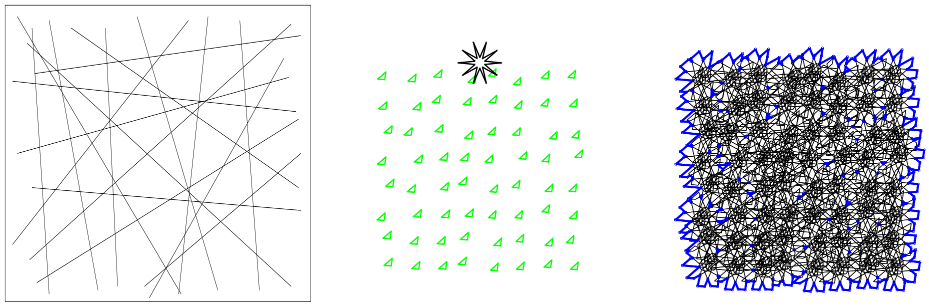

Fig. 9. Random input set (left-hand side). Robotics input set — the planar map (right-hand side) created in the process of constructing the Minkowski sum of a star shaped robot and a set of small triangular obstacles (shown in the middle).

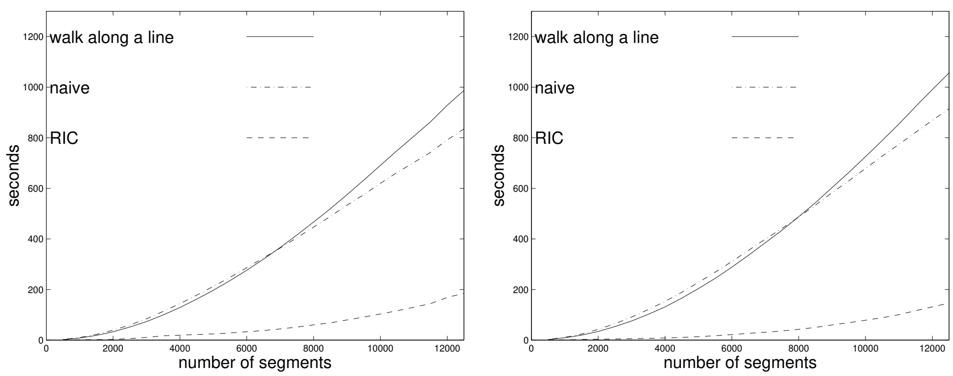

Fig. 10. Construction time for the random input (left-hand side) and for the robotics input (right-hand side) with different point-location algorithms

set has about 12000 segments. The *geographic* input (Figure 13) consists of 25000 segments describing a map of the world.

In each experiment we choose a random permutation of the input segments and add them one after the other into the map. We report the running time for intermediate stages. All the experiments were carried out on a Pentium-II 450MHz PC with 528MB RAM memory, under Linux.

In the first experiment we compared the construction time of planar maps using the three point-location algorithms: the naive, the walk and the RIC (Section 5). The traits class we use is `Pm_leda_segment_exact_traits` (Section 4). Note that the point-location query in the insertion function dominates the construction time. Figure 10 displays the results for the random and robotics inputs.

As expected, construction took the least time using the RIC algorithm. Construction took considerably longer when using the walk or naive point-location algorithms. When we compared the naive and walk methods, it initially seemed surprising that the construction time with the naive algorithm is sometimes faster than the construction time with the walk algorithm. The reason is that the first

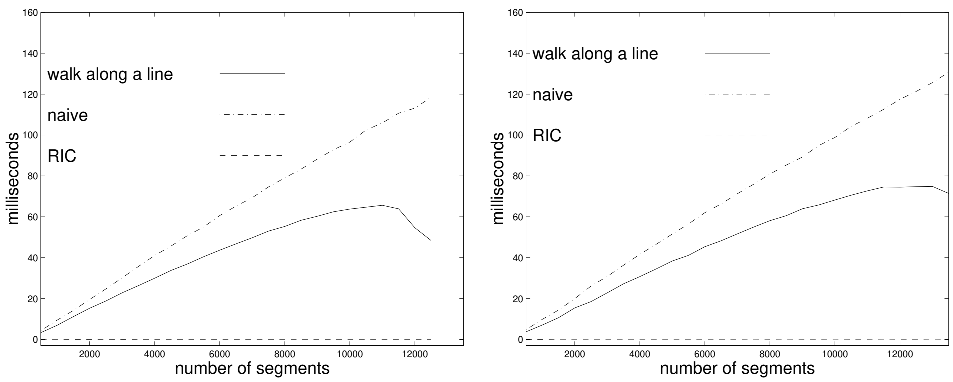

Fig. 11. Average point-location query time (in milliseconds) for the map constructed for the random input set (left-hand side) and for the robotics input set (right-hand side) using different algorithms

phase of the naive point location procedure is to check whether the query point lies exactly on one of the vertices of the planar map. Since it includes only simple comparisons, this phase is very efficient relative to the other predicates that are executed by the point-location algorithms. We also observe this phenomenon on the other input sets which all have many segments that share endpoints.

In the second experiment we performed 1000 random point-location queries in each planar-map using the three point location algorithms and we report the average time for one query. Again, we used `Pm_leda_segment_exact_traits`. The RIC algorithm always gives the best results. The running time for one query is less than 0.5 milliseconds. The naive algorithm has linear performance and it is the slowest. Answering point location queries with the walk algorithm is faster than with the naive algorithm. Moreover, we even get better results for denser maps because the zone of the line along which we walk to find the query point is made of much simpler faces—at some point most of the segments' endpoints are already in the map and most of the new edges we add split faces into simpler faces with less edges on their boundaries. This is the explanation for the drop in the running time in the walk performance graph.

The third experiment demonstrates the construction time for the random set while using different traits classes. Since the running time becomes very long for large inputs, we restrict ourselves to maps of 5000 segments. Figure 12 displays the construction time using the RIC algorithm. As expected, construction with floating-point arithmetic is the fastest. The LEDA traits perform very well compared with a Cartesian representation with a rational number type (in this case the *leda_rational* type).

We also experimented with larger sets. An example is displayed in Figure 13. Constructing the planar map of the 25000 segments of the geographic input with the RIC algorithm and LEDA traits takes 166 seconds.

From the above experiments we can draw some immediate conclusions. First, we can see that using the RIC algorithm is the better choice for incremental construc-

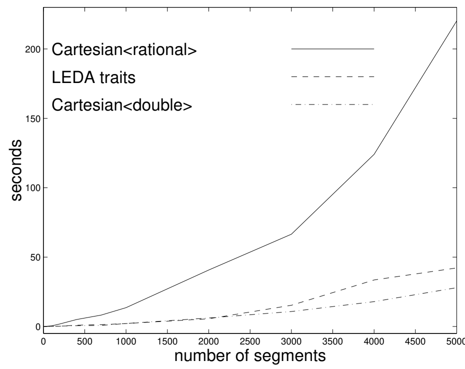

Fig. 12. Average construction time for the random input set with the RIC algorithm using different traits classes

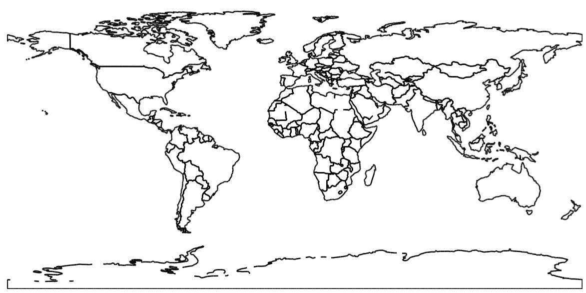

Fig. 13. The geographic input (25000 segments)

tion of a planar map. The performance depends very much on the input set and other applications may find other point location algorithms preferable. Second, if we have to construct a planar map of line segments it is preferable to use the LEDA traits—we get both robust behavior and good performance.

7. CONCLUSIONS

We have described a software package for constructing and efficiently accessing planar maps. The package can be used with general types of x-monotone curves, as long as the users supply a small set of functions (geometric traits) for their type of curves. We also supply the geometric traits for several types of curves including line segments and circular arcs. Besides its generality, our package is special in its handling of degeneracies. We relax the typical "general position" assumption made in the theoretical study of geometric algorithms. For example, we allow for any number of segments to intersect in a single point.

Two additional packages have been recently successfully developed on top of the planar map package. The first is for two-dimensional arrangements of curves [Hanniel and Halperin 2000]. An arrangement is the subdivision of the plane induced by curves that are possibly intersecting and not necessarily x-monotone. Therefore, the arrangement package requires additional functionality.

The second package is for computing Minkowski sums (or vector sums) of planar sets [Agarwal et al. 2000; Flato and Halperin 2000]. Computing the Minkowski sum of two sets is a basic and useful geometric operation in computer-aided design and manufacturing (CAD/CAM) among other applications. The planar map package turned out to be a valuable and convenient infrastructure for the Minkowski sum package. The support of degeneracy handling in the planar map package is especially significant for certain instances of Minkowski sums.

The storage taken by our data structures is linear in the number of input curves, both for the DCEL and for the most elaborate point location algorithm—the RIC, ignoring the space required to represent coordinates. However, when using certain exact number types the coordinates take the lion's share of space. We leave the determination of the constants in the storage requirement and the effect of using exact number types on the storage for a future study.

In developing the package we paid special attention to robustness and generality, often at the expense of speed. Our package could benefit from more optimization of the code to speed up the running time of the algorithms. The experiments reported in this paper are just part of a more comprehensive set of experiments that will be done to identify the more time-consuming portions of the code and will be reported in the future.

An obvious goal for future work is to develop geometric traits for more types of and more general curves. A more ambitious goal is to develop robust software for general spatial maps, namely to extend the package described here to the three-dimensional case.

ACKNOWLEDGMENTS

The authors wish to thank Sariel Har-Peled, Sigal Raab, Lutz Kettner, Hervé Brönnimann and Michal Ozery for helpful discussions concerning the package described in this paper and other contributions. The authors also thank all the people who attended the CGAL meeting in October 1997 at INRIA, Sophia Antipolis, where the planar map design was discussed. Finally, we thank the anonymous referees for many valuable comments.

REFERENCES

AGARWAL, P. K., FLATO, E., AND HALPERIN, D. 2000. Polygon decomposition for efficient construction of Minkowski sums. In *Algorithms — Proc. 8th Annu. European Sympos. Algorithms*, Volume 1879 of *Lecture Notes Comput. Sci.* (2000), pp. 20–31. Springer-Verlag.

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

BRÖNNIMAN, H., KETTNER, L., SCHIRRA, S., AND VELTKAMP, R. 1998. Applications of the generic programming paradigm in the design of CGAL. Technical Report MPI-I-98-1-030, Max-Planck-Insitut für Informatik, 66123 Saarbrücken, Germany.

CGAL00. 2000. *The CGAL User Manual, Version 2.2.* http://www.cgal.org.

DE BERG, M., VAN KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. 1997. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin.

FABRI, A., GIEZEMAN, G., KETTNER, L., SCHIRRA, S., AND SCHÖNHERR, S. 1996. The CGAL kernel: A basis for geometric computation. In M. C. LIN AND D. MANOCHA Eds., *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, Volume 1148 of *Lecture Notes Comput. Sci.* (1996), pp. 191–202. Springer-Verlag.

FABRI, A., GIEZEMAN, G., KETTNER, L., SCHIRRA, S., AND SCHÖNHERR, S. 2000. On the design of CGAL, the Computational Geometry Algorithms Library. *Software - Practice and Experience 30*, 1167–1202.

FLATO, E. AND HALPERIN, D. 2000. Robust and efficient construction of planar Minkowski sums. In *Abstracts 16th European Workshop Comput. Geom.* (Eilat, 2000), pp. 85–88.

FORTUNE, S. AND VAN WYK, C. J. 1996. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph. 15*, 3 (July), 223–248.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GUIBAS, L. J. 1996. Implementing geometric algorithms robustly. In *Proc. 1st ACM Workshop on Appl. Comput. Geom.* (May 1996), pp. 24–28.

HANNIEL, I. AND HALPERIN, D. 2000. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *Proc. of the 4th Workshop on Algorithm Engineering* (2000).

KETTNER, L. 1999. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry: Theory and Applications 13*, 65–90.

KEYSER, J., CULVER, T., MANOCHA, D., AND KRISHNAN, S. 1999. MAPC: a library for efficient manipulation of algebraic points and curves. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.* (1999), pp. 360—369.

MÄNTYLÄ, M. 1988. *An Introduction to Solid Modeling.* Computer Science Press, Rockville, MD.

MEHLHORN, K., NÄHER, S., UHRIG, C., AND SEEL, M. 1997. *The LEDA User Manual, Version 3.8.* 66123 Saarbrücken, Germany: Max-Planck-Insitut für Informatik.

MELHORN, K. AND NÄHER, S. 1999. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press.

MULMULEY, K. 1990. A fast planar partition algorithm, *I. J. Symbolic Comput. 10*, 3-4, 253–280.

MYERS, N. C. 1995. Traits: a new and useful template technique. *C++ Report*.

SCHIRRA, S. 1999. A case study on the cost of geometric computing. In *Algorithm Engineering and Experimentation (ALENEX99)*, Volume 1619 of *Lecture Notes Comput. Sci.* (Berlin, 1999), pp. 156–176. Springer-Verlag.

WEILER, K. 1985. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications 5*, 1 (Jan.), 21–40.