Research Report: Implementation of Planar Nef Polyhedra

Michael Seel

MPI–I–2001–1-003 August 2001

Author's Address

Michael Seel
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg
66123 Saarbrücken
seel@mpi-sb.mpg.de

Acknowledgements

Partially supported by ESPRIT project 28155 (GALIA).

Abstract

A planar Nef polyhedron is any set that can be obtained from the open halfspaces by a finite number of set complement and set intersection operations. The set of Nef polyhedra is closed under the Boolean set operations. We describe a data structure that realizes two-dimensional Nef polyhedra and offers a large set of binary and unary set operations. The underlying set operations are realized by an efficient and complete algorithm for the overlay of two Nef polyhedra. The algorithm is efficient in the sense that its running time is bounded by the size of the inputs plus the size of the output times a logarithmic factor. The algorithm is complete in the sense that it can handle all inputs and requires no general position assumption. The second part of the algorithmic interface considers point location and ray shooting in planar subdivisions.

The implementation follows the generic programming paradigm in C++ and CGAL. Several concept interfaces are defined that allow the adaptation of the software by the means of traits classes. The described project is part of the CGAL library version 2.3.

Keywords

Nef polyhedron, Arrangement, Plane Sweep, Bounding Box

Introduction

This report collects all implementation documents concerning the package *Nef_polyhedron_2* of CGAL. It is a complete literate programming document consisting of specification and implementation descriptions. The chapter contain the major implementation modules used to realize the Nef polyhedra in CGAL.

A planar Nef polyhedron is any set that can be obtained from the open halfspaces by a finite number of set complement and set intersection operations. The set of Nef polyhedra is closed under the Boolean set operations. We describe a data structure that realizes two-dimensional Nef polyhedra and offers a large set of binary and unary set operations. The underlying set operations are realized by an efficient and complete algorithm for the overlay of two Nef polyhedra. The algorithm is efficient in the sense that its running time is bounded by the size of the inputs plus the size of the output times a logarithmic factor. The algorithm is complete in the sense that it can handle all inputs and requires no general position assumption. The second part of the algorithmic interface considers point location and ray shooting in planar subdivisions.

The implementation follows the generic programming paradigm in C++ and CGAL. Several concept interfaces are defined that allow the adaptation of the software by the means of traits classes. The described project is part of the CGAL library version 2.3.

The framing module is that of chapter 1. It is based on three algorithmic parts the geometry (presented in the chapters 7 and 8, where the former uses the polynomials of chapter 6), the binary overlay of plane maps (based on the module of chapter 3 that in turn uses the module of chapter 2), and the point location module (based on the module of chapter 5 that in turn uses the module of chapter 4). The generic sweep class of chapter 10 is just a standard interface for our sweep modules presented in the chapters 2 and 4.

Contents

1 Planar Nef Polyhedra

1.1 The Interface Specification

1.1.1 Nef Polyhedra in the Plane (Nef_polyhedron_2)

1. Definition

An instance of data type *Nef_polyhedron_2*<*T*> is a subset of the plane that is the result of forming complements and intersections starting from a finite set H of halfspaces. *Nef_polyhedron_2* is closed under all binary set operations *intersection*, *union*, *difference*, *complement* and under the topological operations *boundary*, *closure*, and *interior*.

The template parameter T is specified via an extended kernel concept. T must be a model of the concept *ExtendedKernelTraits_2*.

2. Types

Nef_polyhedron_2<*T*>::*Line*	the oriented lines modeling halfplanes
Nef_polyhedron_2<*T*>::*Point*	the affine points of the plane.
Nef_polyhedron_2<*T*>::*Direction*	directions in our plane.
Nef_polyhedron_2<*T*>::*Aff_transformation*	affine transformations of the plane.

Nef_polyhedron_2<*T*>::*Boundary* { EXCLUDED, INCLUDED }

construction selection.

Nef_polyhedron_2<*T*>::*Content* { EMPTY, COMPLETE }

construction selection

3. Creation

Nef_polyhedron_2<*T*> *N*(*Content plane* = *EMPTY*);

creates an instance N of type *Nef_polyhedron_2*<*T*> and initializes it to the empty set if *plane* == *EMPTY* and to the whole plane if *plane* == *COMPLETE*.

Nef_polyhedron_2<*T*> *N*(*Line l*, *Boundary line* = *INCLUDED*);

creates a Nef polyhedron N containing the halfplane left of l including l if *line* == *INCLUDED*, excluding l if *line* == *EXCLUDED*.

template <*class Forward_iterator*>

Nef_polyhedron_2<T> N(*Forward_iterator it*, *Forward_iterator end*, *Boundary b* = *INCLUDED*);

creates a Nef polyhedron N from the simple polygon P spanned by the list of points in the iterator range $[it, end)$ and including its boundary if b = *INCLUDED* and excluding the boundary otherwise. *Forward_iterator* has to be an iterator with value type *Point*. This construction expects that P is simple. The degenerate cases where P contains no point, one point or spans just one segment (two points) are correctly handled. In all degenerate cases there's only one unbounded face adjacent to the degenerate polygon. If b == *INCLUDED* then N is just the boundary. If b == *EXCLUDED* then N is the whole plane without the boundary.

4. Operations

void	N.clear(*Content plane* = *EMPTY*)	makes N the empty set if *plane* == *EMPTY* and the full plane if *plane* == *COMPLETE*.
bool	N.is_empty()	returns true if N is empty, false otherwise.
bool	N.is_plane()	returns true if N is the whole plane, false otherwise.

Constructive Operations

Nef_polyhedron_2<T>	N.complement()	returns the complement of N in the plane.
Nef_polyhedron_2<T>	N.interior()	returns the interior of N.
Nef_polyhedron_2<T>	N.closure()	returns the closure of N.
Nef_polyhedron_2<T>	N.boundary()	returns the boundary of N.
Nef_polyhedron_2<T>	N.regularization()	returns the regularized polyhedron (closure of interior).
Nef_polyhedron_2<T>	N.intersection(*Nef_polyhedron_2<T> N1*)	returns $N \cap N1$.
Nef_polyhedron_2<T>	N.join(*Nef_polyhedron_2<T> N1*)	returns $N \cup N1$.
Nef_polyhedron_2<T>	N.difference(*Nef_polyhedron_2<T> N1*)	returns $N - N1$.
Nef_polyhedron_2<T>	N.symmetric_difference(*Nef_polyhedron_2<T> N1*)	returns the symmectric difference $N - T \cup T - N$.
Nef_polyhedron_2<T>	N.transform(*Aff_transformation t*)	returns $t(N)$.

Additionally there are operators $*, +, -, \hat{}, !$ which implement the binary operations *intersection*, *union*, *difference*, *symmetric difference*, and the unary operation *complement* respectively. There are also the corresponding modification operations $*=, +=, -=, \hat{}=$.

There are also comparison operations like $<, \leq, >, \geq, ==, !=$ which implement the relations subset, subset or equal, superset, superset or equal, equality, inequality, respectively.

Exploration - Point location - Ray shooting

As Nef polyhedra are the result of forming complements and intersections starting from a set H of halfspaces that are defined by oriented lines in the plane, they can be represented by an attributed plane map $M = (V, E, F)$. For topological queries within M the following types and operations allow exploration access to this structure.

5. Types

Nef_polyhedron_2<T>::Explorer

a decorator to examine the underlying plane map. See the manual page of *Explorer*.

Nef_polyhedron_2<T>::Object_handle

a generic handle to an object of the underlying plane map. The kind of object (*vertex*, *halfedge*, *face*) can be determined and the object can be assigned to a corresponding handle by the three functions:
bool assign(*Vertex_const_handle*& *h*, *Object_handle*)
bool assign(*Halfedge_const_handle*& *h*, *Object_handle*)
bool assign(*Face_const_handle*& *h*, *Object_handle*)
where each function returns *true* iff the assignment to *h* was done.

Nef_polyhedron_2<T>::Location_mode { DEFAULT, NAIVE, LMWT }

selection flag for the point location mode.

6. Operations

bool *N*.contains(*Object_handle h*)

returns true iff the object *h* is contained in the set represented by *N*.

bool *N*.contained_in_boundary(*Object_handle h*)

returns true iff the object *h* is contained in the 1-skeleton of *N*.

Object_handle *N*.locate(*Point p*, *Location_mode m* = *DEFAULT*)

returns a generic handle *h* to an object (face, halfedge, vertex) of the underlying plane map that contains the point *p* in its relative interior. The point *p* is contained in the set represented by *N* if *N.contains*(*h*) is true. The location mode flag *m* allows one to choose between different point location strategies.

Object_handle *N*.ray_shoot(*Point p*, *Direction d*, *Location_mode m* = *DEFAULT*)

returns a handle *h* with *N.contains*(*h*) that can be converted to a *Vertex_/Halfedge_/Face_const_handle* as described above. The object returned is intersected by the ray starting in *p* with direction *d* and has minimal distance to *p*. The operation returns the null handle *NULL* if the ray shoot along *d* does not hit any object *h* of *N* with *N.contains*(*h*). The location mode flag *m* allows one to choose between different point location strategies.

Object_handle *N*.ray_shoot_to_boundary(*Point p*, *Direction d*, *Location_mode m* = *DEFAULT*)

returns a handle *h* that can be converted to a *Vertex_/Halfedge_const_handle* as described above. The object returned is part of the 1-skeleton of *N*, intersected by the ray starting in *p* with direction *d* and has minimal distance to *p*. The operation returns the null handle *NULL* if the ray shoot along *d* does not hit any 1-skeleton object *h* of *N*. The location mode flag *m* allows one to choose between different point location strategies.

Explorer *N*.explorer()

returns a decorator object which allows read-only access of the underlying plane map. See the manual page *Explorer* for its usage.

Input and Output

A Nef polyhedron *N* can be visualized in a *Window_stream W*. The output operator is defined in the file *CGAL/IO/Nef_polyhedron_2_Window_stream.h*.

7. Implementation

Nef polyhedra are implemented on top of a halfedge data structure and use linear space in the number of vertices, edges and facets. Operations like *empty* take constant time. The operations *clear*, *complement*, *interior*, *closure*, *boundary*, *regularization*, input and output take linear time. All binary set operations and comparison operations take time $O(n \log n)$ where n is the size of the output plus the size of the input.

The point location and ray shooting operations are implemented in two flavors. The *NAIVE* operations run in linear query time without any preprocessing, the *DEFAULT* operations (equals *LMWT*) run in sub-linear query time, but preprocessing is triggered with the first operation. Preprocessing takes time $O(N^2)$, the sub-linear point location time is either logarithmic when LEDA's persistent dictionaries are present or if not then the point location time is worst-case linear, but experiments show often sublinear runtimes. Ray shooting equals point location plus a walk in the constrained triangulation overlayed on the plane map representation. The cost of the walk is proportional to the number of triangles passed in direction d until an obstacle is met. In a minimum weight triangulation of the obstacles (the plane map representing the polyhedron) the theory provides a $O(\sqrt{n})$ bound for the number of steps. Our locally minimum weight triangulation approximates the minimum weight triangulation only heuristically (the calculation of the minimum weight triangulation is conjectured to be NP hard). Thus we have no runtime guarantee but a strong experimental motivation for its approximation.

8. Example

Nef polyhedra are parameterized by a so-called extended geometric kernel. There are three kernels, one based on a homogeneous representation of extended points called *Extended_homogeneous<RT>* where *RT* is a ring type providing additionally a *gcd* operation and one based on a cartesian representation of extended points called *Extended_cartesian<NT>* where *NT* is a field type, and finally *Filtered_extended_homogeneous<RT>* (an optimized version of the first).

The member types of *Nef_polyhedron_2< Extended_homogeneous<NT> >* map to corresponding types of the CGAL geometry kernel (e.g. *Nef_polyhedron::Line* equals *CGAL::Homogeneous<leda_integer>::Line_2* in the example below).

```
#include <CGAL/basic.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_2.h>

using namespace CGAL;
typedef  Extended_homogeneous<leda_integer> Extended_kernel;
typedef  Nef_polyhedron_2<Extended_kernel>  Nef_polyhedron;
typedef  Nef_polyhedron::Line               Line;

int main()
{
  Nef_polyhedron N1(Line(1,0,0));
  Nef_polyhedron N2(Line(0,1,0), Nef_polyhedron::EXCLUDED);
  Nef_polyhedron N3 = N1 * N2; // line (*)
  return 0;
}
```

After line (*) *N3* is the intersection of *N1* and *N2*.

1.1.2 Topological plane map exploration (Topological_explorer)

1. Definition

An instance *D* of the data type *Topological_explorer* is a decorator for interfacing the topological structure of a plane map *P* (read-only).

A plane map *P* consists of a triple (V,E,F) of vertices, edges, and faces. We collectively call them objects. An edge *e* is a pair of vertices (v,w) with incidence operations $v = source(e)$, $w = target(e)$. The list of all edges with source v is called the adjacency list $A(v)$.

Edges are paired into twins. For each edge $e = (v,w)$ there's an edge $twin(e) = (w,v)$ and $twin(twin(e)) == e$[1].

An edge $e = (v,w)$ knows two adjacent edges $en = next(e)$ and $ep = previous(e)$ where $source(en) = w$, $previous(en) = e$ and $target(ep) = v$ and $next(ep) = e$. By this symmetric $previous - next$ relationship all edges are partitioned into face cycles. Two edges e and e' are in the same face cycle if $e = next^*(e')$. All edges e in the same face cycle have the same incident face $f = face(e)$. The cyclic order on the adjacency list of a vertex $v = source(e)$ is given by $cyclic\_adj\_succ(e) = twin(previous(e))$ and $cyclic\_adj\_pred(e) = next(twin(e))$.

A vertex v is embedded via coordinates $point(v)$. By the embedding of its source and target an edge corresponds to a segment. P has the property that the embedding is always *order-preserving*. This means a ray fixed in $point(v)$ of a vertex v and swept around counterclockwise meets the embeddings of $target(e)$ $(e \in A(v))$ in the cyclic order defined by the list order of A.

The embedded face cycles partition the plane into maximal connected subsets of points. Each such set corresponds to a face. A face is bounded by its incident face cycles. For all the edges in the non-trivial face cycles it holds that the face is left of the edges. There can also be trivial face cycles in form of isolated vertices in the interior of a face. Each such vertex v knows its surrounding face $f = face(v)$.

Plane maps are attributed by a Boolean value, for each object $u \in V \cup E \cup F$ we attribute an information $mark(u)$ of type *bool*.

2. Types

Topological_explorer::*Plane_map*	The underlying plane map type
Topological_explorer::*Point*	The point type of vertices.
Topological_explorer::*Mark*	All objects (vertices, edges, faces) are attributed by a *Mark* object.
Topological_explorer::*Size_type*	The size type.

Local types are handles, iterators and circulators of the following kind: *Vertex_const_handle*, *Vertex_const_iterator*, *Halfedge_const_handle*, *Halfedge_const_iterator*, *Face_const_handle*, *Face_const_iterator*. Additionally the following circulators are defined.

Topological_explorer::*Halfedge_around_vertex_const_circulator*

circulating the outgoing halfedges in $A(v)$.

Topological_explorer::*Halfedge_around_face_const_circulator*

circulating the halfedges in the face cycle of a face f.

Topological_explorer::*Hole_const_iterator* iterating all holes of a face f. The type is convertible to *Halfedge_const_handle*.

Topological_explorer::*Isolated_vertex_const_iterator*

iterating all isolated vertices of a face f. The type generalizes *Vertex_const_handle*.

3. Operations

Vertex_const_handle *D*.source(*Halfedge_const_handle e*)

returns the source of e.

Vertex_const_handle *D*.target(*Halfedge_const_handle e*)

returns the target of e.

Halfedge_const_handle *D*.twin(*Halfedge_const_handle e*)

returns the twin of e.

bool *D*.is_isolated(*Vertex_const_handle v*)

returns *true* iff $A(v) = \emptyset$.

[1]The existence of the edge pairs makes P a bidirected graph, the *twin* links make P a map.

Halfedge_const_handle *D*.first_out_edge(*Vertex_const_handle v*)

returns one halfedge with source v. It's the starting point for the circular iteration over the halfedges with source v. *Precondition*: $!is\_isolated(v)$.

Halfedge_const_handle *D*.last_out_edge(*Vertex_const_handle v*)

returns a the halfedge with source v that is the last in the circular iteration before encountering $first\_out\_edge(v)$ again. *Precondition*: $!is\_isolated(v)$.

Halfedge_const_handle *D*.cyclic_adj_succ(*Halfedge_const_handle e*)

returns the edge after e in the cyclic ordered adjacency list of $source(e)$.

Halfedge_const_handle *D*.cyclic_adj_pred(*Halfedge_const_handle e*)

returns the edge before e in the cyclic ordered adjacency list of $source(e)$.

Halfedge_const_handle *D*.next(*Halfedge_const_handle e*)

returns the next edge in the face cycle containing e.

Halfedge_const_handle *D*.previous(*Halfedge_const_handle e*)

returns the previous edge in the face cycle containing e.

Face_const_handle *D*.face(*Halfedge_const_handle e*)

returns the face incident to e.

Face_const_handle *D*.face(*Vertex_const_handle v*)

returns the face incident to v. *Precondition*: $is\_isolated(v)$.

Halfedge_const_handle *D*.halfedge(*Face_const_handle f*)

returns a halfedge in the bounding face cycle of f ($Halfedge\_const\_handle(\ )$ if there is no bounding face cycle).

Iteration

Halfedge_around_vertex_const_circulator *D*.out_edges(*Vertex_const_handle v*)

returns a circulator for the cyclic adjacency list of v.

Halfedge_around_face_const_circulator *D*.face_cycle(*Face_const_handle f*)

returns a circulator for the outer face cycle of f.

Hole_const_iterator *D*.holes_begin(*Face_const_handle f*)

returns an iterator for all holes in the interior of f. A *Hole_iterator* can be assigned to a *Halfedge_around_face_const_circulator*.

Hole_const_iterator *D*.holes_end(*Face_const_handle f*)

returns the past-the-end iterator of f.

Isolated_vertex_const_iterator *D*.isolated_vertices_begin(*Face_const_handle f*)

returns an iterator for all isolated vertices in the interior of f.

Isolated_vertex_const_iterator *D*.isolated_vertices_end(*Face_const_handle f*)

returns the past the end iterator of f.

Associated Information

The type *Mark* is the general attribute of an object. The type *GenPtr* is equal to type *void∗*.

const Point& *D*.point(*Vertex_const_handle v*)

returns the embedding of v.

const Mark& *D*.mark(*Vertex_const_handle v*)

returns the mark of v.

const Mark& *D*.mark(*Halfedge_const_handle e*)

returns the mark of e.

const Mark& *D*.mark(*Face_const_handle f*)

returns the mark of f.

Statistics and Integrity

Size_type *D*.number_of_vertices( ) returns the number of vertices.

Size_type *D*.number_of_halfedges( ) returns the number of halfedges.

Size_type *D*.number_of_edges( ) returns the number of halfedge pairs.

Size_type *D*.number_of_faces( ) returns the number of faces.

Size_type *D*.number_of_face_cycles( ) returns the number of face cycles.

Size_type *D*.number_of_connected_components( )

calculates the number of connected components of P.

void *D*.print_statistics(*std*::*ostream*& *os* = *std*::*cout*)

print the statistics of P: the number of vertices, edges, and faces.

void *D*.check_integrity_and_topological_planarity(*bool faces* = *true*)

checks the link structure and the genus of P.

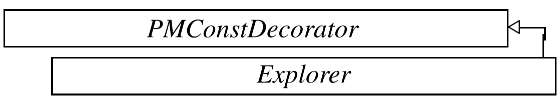
1.1.3 Plane map exploration (Explorer)

1. Definition

An instance E of the data type *Explorer* is a decorator to explore the structure of the plane map underlying the Nef polyhedron. It inherits all topological adjacency exploration operations from *PMConstDecorator*. *Explorer* additionally allows one to explore the geometric embedding.

The position of each vertex is given by a so-called extended point, which is either a standard affine point or the tip of a ray touching an infinimaximal square frame centered at the origin. A vertex v is called a *standard* vertex if its embedding is a *standard* point and *non-standard* if its embedding is a *non-standard* point. By the straightline embedding of their source and target vertices, edges correspond to either affine segments, rays or lines or are part of the bounding frame.

2. Generalization



3. Types

Explorer::*Topological_explorer*

The base class.

Explorer::*Point* the point type of finite vertices.

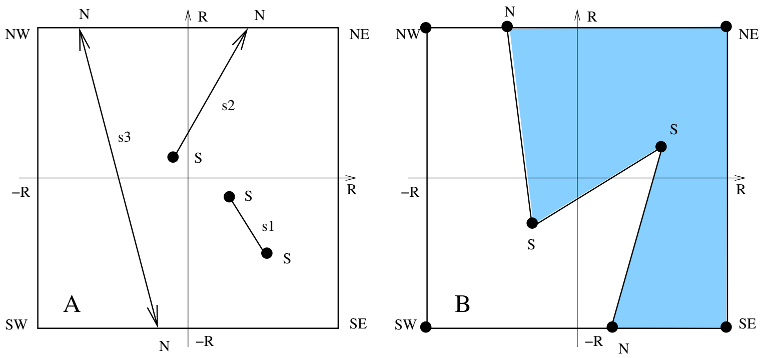


Figure 1.1: Extended geometry: standard vertices are marked by S, non-standard vertices are marked by N. **A**: The possible embeddings of edges: an affine segment s1, an affine ray s2, an affine line s3. **B**: A plane map embedded by extended geometry: note that the frame is arbitrarily large, the 6 vertices on the frame are at infinity, the two faces represent a geometrically unbounded area, however they are topologically closed by the frame edges. No standard point can be placed outside the frame.

Explorer::*Ray* the ray type of vertices on the frame.

Iterators, handles, and circulators are inherited from *Topological_explorer*.

4. Creation

Explorer is copy constructable and assignable. An object can be obtained via the *Nef_polyhedron_2*::*explorer*( ) method of *Nef_polyhedron_2*.

5. Operations

bool *E*.is_standard(*Vertex_const_handle v*)

returns true iff *v*'s position is a standard point.

Point *E*.point(*Vertex_const_handle v*)

returns the standard point that is the embedding of *v*. *Precondition*: *E.is_standard*(*v*).

Ray *E*.ray(*Vertex_const_handle v*)

returns the ray defining the non-standard point on the frame. *Precondition*: !*E.is_standard*(*v*).

bool *E*.is_frame_edge(*Halfedge_const_handle e*)

returns true iff *e* is part of the infinimaximal frame.

1.2 Motivation

Nef polyhedra are the most general model for rectilinearly bounded subsets of affine space. Their definition is surprisingly simple whereas the operations that are supported without leaving the model are versatile. Nef's model of polyhedra does not impose topological restrictions on the sets that can be modeled like manifold or regularized models do. This implies of course that the abstract representation of the underlying theory has to cope with general topological complexity. The main reason for us to offer a data type Nef polyhedron is that many other models that are standard concepts in the field are covered by Nef's model:

- A *convex polytope* is defined as the convex hull of a nonempty finite set of points. Convex polytopes are thus compact closed and manifold sets. [Grü67]
- An *elementary polyhedron* is defined as the union of a finite number of convex polytopes. [Grü67]
- A *polyhedral set* is defined as the intersection of a finite number of closed half-spaces. Such sets are closed and convex but need not to be compact. [Grü67]
- The set of all points belonging to the simplices of a *simplicial complex* is normally called a (rectilinear) polyhedron. [Lef71]

This list shows that a system modeling Nef polyhedra enables a user to calculate in many interesting domains.

1.3 Previous Work

We cite the main publications from the field and present its development. We will first give an outline, then we deepen the notions that are interesting with respect to our research. Other notions are solely linked to the literature.

The theory of Nef polyhedra was first published in W. Nef's book "Beiträge zur Theorie der Polyeder mit Anwendungen in der Computergraphik" [Nef78]. The book presents a mathematically sound theory of a general kind of polyhedra in arbitrary dimension and provides a great intuition about the generality of the elaborated concepts. The algorithmic part does not specialize in dimension. It should be clear that by only realizing a fixed low-dimensional data type the corresponding algorithms and data structures can be streamlined in runtime and space requirements.

In the following considerations we concentrate mainly on the presented data structure and the realization of a binary set intersection operation which poses the most requirements[2] on the underlying data structures and algorithmic modules.

On the workshop on computational geometry in Würzburg [BN88] a refined elaboration of the book concepts was given. The paper introduces the later so-called *Würzburg structure* which is the set of all low-dimensional[3] faces of a polyhedron. Each such face is stored in form of its local pyramid. Thus, the data structure is essentially a collection of pyramids. Each pyramid is realized by a selective arrangement of hyperplanes. This representation of pyramids is *not lean* in low dimensions and can be improved. Moreover, no incidence relation is coded into the collection. The intersection operation of two polyhedra is defined on top of this structure and is based on two techniques: recursion in dimension and superposition of two local pyramids. Neither space nor runtime bounds are given for the algorithmic description which apart from that is clearly structured. The paper solves some subproblems by introducing a symbolic parameter to obtain a symbolic hyperplane at infinity. However the transfer from the affine to the symbolic objects is part of the algorithmic flow and not encapsulated into a geometric kernel as we proceed in the report [SM00].

H. Bieri's introduction [Bie95] was a step to market Nef polyhedra to a wider audience. It provides a summary of the book and introduces all notions in a more tutorial-style picture. It also describes a small test system realizing binary set and simple topological operations written in PASCAL based on the so-called *extended Würzburg structure*. The predicate "extended" stems from the

[2] The calculation of closure, interior, or complement is much simpler due to the fact that the faces of the input polyhedron are part of the faces of the output polyhedron in these cases.

[3] not full-dimensional

addition of incidence links to the Würzburg structure[4].

The article [Bie94] describes the realization of simple topological and set operations on top of the Würzburg structure. It has an introductory part and an algorithmic part. The key operation of interest, the intersection of two Nef polyhedra, is defined in pseudo code in a dimension recursive manner (up to dimension three but a possible generalization to higher dimensions is sketched). The main phase of the operation is based on a spatial sweep approach but the presentation is rather condensed, mainly mathematical, and lacks runtime and space bounds. In fact, the described procedure up to dimension two is quadratic and we will improve this bound to the optimal plane sweep bound (of segment intersection).

A follow up to the previous publication is the article [Bie96] that mainly closes open details of algorithmic considerations of the previous articles. Its main impact is the introduction of the *reduced Würzburg structure*. H. Bieri shows that it suffices to store the pyramids[5] of faces that are minimal elements of the incidence relation (a partial order defined via closure relation). As a consequence this reduces the space requirements of the pyramid collection but requires additional algorithmic processing when collecting all faces. The proposition has a strong impact on the representation of Nef polyhedra when we consider dimension three or higher. In space the above result implies that the pyramids of all vertices suffice to completely describe a compact polyhedron.

The paper [Bie98] embeds Nef polyhedra in the field of solid modeling by offering conversion routines between previously defined data structures for Nef polyhedra: the reduced Würzburg structure, selective cellular complexes based on hyperplane arrangements, CSG-trees based on half-spaces, and binary space partitions. Again there are no time and space bounds.

J.R. Rossignac and M.A. O'Connor [RO90] have introduced *Selective Geometric Complexes* (SGC). An SGC consists of a cellular complex (the topological structure) and the corresponding geometric support spaces. Geometrically SGCs are pretty general. They use *real algebraic varieties* as the geometric elements that support cells. Such varieties can be decomposed into finite sets of connected smooth manifolds (so-called *extents*). An SGC is therefore a collection of mutually disjoint cells such that (1) each cell is a relatively open subset of an extent, (2) for each cell of the complex its boundary (a set of cells) is also part of the complex, and (3) each cell has a Boolean selection flag. The dimension of a cell is determined by the dimension of its underlying extent. The point set modeled by such a complex is the union of all selected cells.

One important concept is the incidence relation on cells. In SGCs it is defined in terms of a *boundary* and a *star* relation stemming from the corresponding concepts of simplicial complexes. Moreover due to the possibly curved geometry of extents the notion of *neighborhood* (orientation) is introduced to disambiguate degenerate boundary conditions.

The description of SGCs is still abstract and leaves room for refinement concerning the realization of the necessary data structure concepts[6]. The paper presents the central ideas and abstract definition of SGCs and its notions. It sketches binary operations based on the data type separated into phases. (We use this approach later in our implementation). On the other hand the paper omits many concrete considerations of the algorithmic subtasks (boundary evaluations, merging complexes, etc.) including runtime and space complexity. The problem of unbounded structures is not an issue.

How do SGCs relate to Nef polyhedra? The support spaces of Nef polyhedra are flats. Therefore, varieties and extents are no separate concepts. Many geometric ambiguities do not occur. The theory of Nef polyhedra as described by Nef and Bieri varies in the way how the exterior of Nef polyhedra

[4] However, the extension is only based on an untyped list.
[5] Pyramids represent faces.
[6] A promised follow-up paper never appeared.

is modeled. In their later papers the exterior is a *non-proper* face and thereby the ambient space is completely partitioned. With SGCs the exterior is no cell. We want to stress the following similarity. Assume we realize SGCs geometrically restricted to flats. Then, the simplification algorithm as part of the algorithmic description of the binary operations on SGCs produces cells that are the connected components of proper Nef faces.

K. Dobrindt, K. Mehlhorn, and M. Yvinec [DMY93] describe an efficient algorithm for the intersection of a convex polyhedron and a Nef polyhedron in three-dimensional space. The presentation uses a *local graph* data structure modeling the local view that we introduce below. The local graph data structure is used as a vehicle to project the three dimensional topological neighborhood (the local pyramid) that defines Nef facets into the surface of a sphere centered at a point of interest. Their idea greatly simplifies the representation of local pyramids in three dimensions and allows a space efficient representation thereof (linear as opposed to the possible quadratic space of the original proposed arrangements). The corresponding algorithm was not implemented.

K. Mehlhorn and S. Naeher have introduced the notion of planar *generalized* polygons and implemented them in LEDA [MN99, Section 10.8]. A generalized polygon is a point set bounded by possibly several (weakly) simple polygonal chains. This topological restriction implies that generalized polygons are the same as regularized compact Nef polyhedra.

V. Ferrucci [Fer95] presented two implementation efforts to realize a data type modeling Nef polyhedra. His first approach is based on selective simplicial complexes and restricts the model to bounded Nef polyhedra. All simplices are rectilinearly embedded into the affine subspace spanned by their vertices and represent relatively open convex sets. All simplices (including subsimplices) of the complex are selectable by a Boolean flag. The point set of such a simplicial complex is the union of the embeddings of the selected simplices. In this approach Nef faces are only present implicitly as a union of simplices. The simplicial complex is thus a conforming simplicial subdivision of the bounded Nef polyhedron. In the second part of the paper Ferrucci proves that binary space partitions can be used to realize general Nef polyhedra. Both representations do not provide runtime or space qualification.

Several algorithmic descriptions from the above list either assume general position of their inputs to avoid degeneracies or even require what is called regular intersection. In some cases, the generally present robustness problems are tackled by transformations of the underlying coordinate system to avoid degenerate inputs and minimize robustness problems. The possibility of that approach was presented in [NS90].

Our approach differs from the previous work in several aspects. We elaborate on the planar case which is of course easier than the higher-dimensional case but leaves room for optimization via specialization. We solve the problem of degeneracy and robustness. We use standard data structures of our field to represent Nef polyhedra. We generalize an optimized plane sweep framework that can handle all degenerate cases for the binary operations and meet the optimal time and space bounds. One of our main contributions is the introduction of an extended geometric kernel that encapsulates the necessary geometric predicates to run the operations of the data type. In the report [SM00] we show the implementation of the kernel in two flavors, one which is simple to implement and one which is tuned by filter methods and is both robust and fast.

Quite some effort is put into the user interface of the software. Our data type offers the user to get her hand on faces by handles and explore the incidence of an object within the geometric structure. Our data type is based on a space efficient implementation of plane maps. It incorporates an intuitive exploration interface and allows further attribution of the objects. Our data type could be considered a flavor of the extended Würzburg structure where we shift the incidence into the center of our attention. Faces are not represented by their pyramids but the pyramids can be infered from the incidence relation and the extent of any face can be explored via incident lower-dimensional faces at a cost linear in its size. It is our conviction that the original Würzburg design is reasonable in higher-dimensions but means a loss of strength in the planar case. We also add functionality. Exploration of a geometric complex needs point location and ray shooting operations to link a user's geometric question into the geometric complex and its incidence. Finally our approach unifies the handling of special cases. By the introduction of infimaximal frames (technical report [SM00]) and their integration into the geometric complex we enclose the geometric complex into a symbolic box. Exploration and maintainance of the structures become much easier and more homogeneous as the faces of minimal dimension are always vertices. The latter has interesting consequences as Bieri has shown that the local pyramids of minimal faces describe the polyhedron completely.

In this project we realize a data type *Nef_polyhedron_2*. We present a software project clearly separated into different modules responsible for the different aspects of its realization. We will start with the theory, derive an abstract representation, map it to an abstract data type and add the algorithmic components.

In Section 1.4 we present the abstract knowledge about Nef polyhedra and introduce the notions that we use. Afterwards, in Section 1.5 we present the necessary software components of our design. We introduce the geometric and topological modules and their interaction. Then, in Section 1.6 we present the concrete software design of our main interface data type and describe the interaction of different modules to implement the geometric methods of that data type. To fill the details of the representation we append the three additional implementation projects: the extended geometry in the Chapters 7 and 8, the binary operations in Chapter 3, and the point location in Chapter 5.

1.4 The Theory

We start with the formal definition of Nef polyhedra.

Definition 1 (Nef Polyhedra [Nef78]): A set $P \subseteq \mathbb{R}^n$ is a *Nef polyhedron* if P is the result of a recursive application of set intersection and set complement starting from open half-spaces.

This definition supports the claim that they are the most general framework to handle polyhedral sets. As set union, set difference, and symmetric set difference can be reduced to intersection and complement all these set operations are closed in the model.

H. Bieri later gave alternative definitions for Nef polyhedra and proved their equivalence.

Fact 1 (Bieri [Bie95]): The original definition is equivalent to any of the following conditions

1. P corresponds to the root of a CSG-tree with closed half-spaces as leaf primitives and intersection, union and difference as internal nodes.
2. There exist two finite families $F = \{f_1, \dots, f_n\}$ and $G = \{g_1, \dots, g_m\}$ of relatively open subsets of $\mathbb{R}^n$ such that $P = \bigcup_i f_i$ and $\mathsf{cpl}\, P = \bigcup_j g_j$.
3. There exists a set of hyperplanes H such that P is the union of some cells of the arrangement $\mathcal{A}(H)$.

(1) gives a link to constructive solid geometry. (2) links Nef polyhedra to cellular complexes and (3) to hyperplane arrangements. When studying the original theory actually the third equivalence is the key observation. Many propositions about the point set P can be reduced to an examination of the minimal building blocks of the polyhedron: the cells of the arrangement built by the hyperplanes that define the polyhedron.

The elegance of the definition is carried forward to the notion of faces.

Definition 2 (Local pyramids and Faces): Let $K \subseteq \mathbb{R}^n, x \in \mathbb{R}^n$. We call K a *cone with apex* 0 if $K = \mathbb{R}^+ K$ and cone with apex x if $K = x + \mathbb{R}^+(K - x)$. A cone which is also a polyhedron is called a *pyramid*.

Now let $P \subseteq \mathbb{R}^n$ be a polyhedron and $x \in \mathbb{R}^n$. There is a neighborhood $U_0(x)$ such that the pyramid $Q := x + \mathbb{R}^+((P \cap U(x)) - x)$ is the same for all neighborhoods $U(x) \subseteq U_0(x)$. Q is called the *local pyramid* of P in x and denoted P^x.

A *face* s of P is then a maximal non-empty subset of $\mathbb{R}^n$ such that all of its points have the same local pyramid Q, i.e., $s = \{x \in \mathbb{R}^n : P^x = Q\}$. In this case Q is also denoted P^s. The dimensions of a face is the dimension of its affine hull $\mathsf{dim}(s) := \mathsf{dim}(\mathsf{aff}\, s)$.

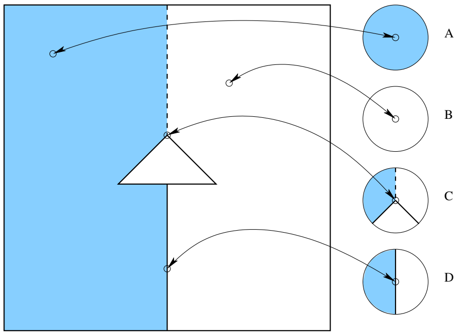


Figure 1.2: The polyhedron P consists of the colored face, the triangle boundary and the vertical segment below the triangle. Some local views of P are: (A) the full plane, (B) the empty set, (C) a radial cake of sectors, (D) a half-space including its boundary.

Note that this notion of a face partitions $\mathbb{R}^n$ into faces of different dimension. Faces as defined by Nef do not have to be connected. There are only two full-dimensional faces possible whose local pyramids are the space itself or the empty set. All lower-dimensional faces form the *boundary* of the polyhedron. As usual we call zero-dimensional faces *vertices* and one-dimensional faces *edges*. In the plane we call the full-dimensional faces *2-faces* or just *faces* when the meaning is clear from the context.

Definition 3 (Incidence): Two faces s and t (in their general sense) are *incident* if $s \subseteq \mathsf{clos}\, t$.

We will treat incidence as a bidirectional relation. We say that s is *downward-incident* to t and conversely that t is *upward-incident* to s.

We now list some facts about faces. The proofs for these facts can be found in Nef's book [Nef78]. We append the chapter and theorem numbers separated by a semicolon. All faces of a polyhedron are polyhedra [6;1.1]. Faces are relatively open sets [6;2]. The linear subspace of all apices of the pyramid associated with a face is the affine hull of the face [6;2]. $x \in P^x$ iff $x \in P$ [3;7]. Let s be a face of the polyhedron P and let t be a face of s. Then, t is the union of some faces of P [6;12]. A face of P is either a subset of P or disjoint from P [6;4]. For two faces s and t of P either $s \subseteq \mathsf{clos}(t)$ or $s \cap \mathsf{clos}(t) = \emptyset$ [6;9,10].

Remember that Nef edges and Nef 2-faces are not necessarily connected. Note also that some connected components of edges are not necessarily bounded by a vertex. How do the local pyramids of any point x in the plane look like? We can represent them by the intersection of a small enough neighborhood disc centered at x with its pyramid P^x. The disc is partitioned into sectors by radial segments. We assign a mark to the center, to any radial segment, and to the sectors in between the radial segments such that the corresponding point set is marked if it belongs to the local pyramid of x. We also call such a disc together with its marks the *local view* of x (cf. Figure 1.2).

Lemma 1.4.1: The local view of a point x has the following properties:

1. the mark of any radial segment is different from one of the two sectors incident to it.

2. x is contained in a 2-face of P iff the local view is a disc that contains no radial segments and is marked as the center.

3. x is contained in an edge of P iff the local view contains exactly two radial segments that are part of a line through the center; the center and the two radial segments are marked equally but their common mark is different from at least one sector of the disc.

4. x is a vertex of P iff the mark of the center is different from at least one radial segment or one sector of the disc and the local view is not that of item 3.

Proof. (1) follows from the fact that radial segments are part of the boundary of an open or closed half-plane and a point on this boundary has the local view of that half-plane. (2) refers to the two trivial pyramids: the empty set and the full plane. (3) follows from the fact that edges are part of the boundary of open and closed half-spaces. (4) if all marks are equal then x would have a totally marked or non-marked disk neighborhood. But that's the neighborhood of a 2-face. □

To determine the local view of a point x with respect to a polyhedron P is called to *qualify* x with respect to P.

1.5 The Data Structure

We want to store Nef polyhedra in an intuitive way and want to use generally known data structures. Faces should be objects whose extent and topological neighborhood (incidence) can be explored. We revert the role of pyramids and incidence of the extended Würzburg structure. In our approach incidence is the key concept, pyramids can be derived from it.

Starting from the last item of Fact 1 we have to model (parts) of an arrangement of lines in the plane. Moreover we want to model the Nef faces including their incidences. 2-faces of a polyhedron are in general two dimensional sets of points bounded by chains of segments that do not have to be simple, can be circularly closed, or open. The latter happens when 2-faces extend to infinity. Then the chain of segments has rays as its first and last element. A simple line bounding a 2-face can be seen as two oppositely oriented rays starting in the same point.

Therefore, concerning the geometry of edges we need to model straight line objects like segments, rays and lines, which are the result of intersection operations of half-spaces. To simplify the treatment of unbounded structures we use the concept of extended points and infimaximal frames which we have formally introduce in the report [SM00] and which builds the geometric layer of our design. Consider an axis-parallel squared box centered at the origin. Any ray is pruned by this box in a so-called non-standard point (corresponding to the ray-tip). The assignment of ray-tips to box segments becomes topologically constant when we grow the framing box above a certain size. The frame is called infimaximal because it is always large enough to enclose all concrete geometric objects like points and segments in its interior that appear in the execution of our algorithms. Extended points are defined to be standard points and the non-standard points corresponding to ray-tips. They allow us to represent segments, rays and lines by a pair of such points and we get rid of the infinite extent of the unbounded structures. Adding such a frame to bound the plane, all unbounded faces become symbolically bounded structures, their boundary becomes a cyclic structure.

To realize Nef faces including their incidence relations we need a cellular subdivision of the plane. We use a plane map data type (bidirected, embedded graphs). The incidence relations between the objects of a plane map (vertices, edges, and faces) reflect the topology of the local pyramids of the planar Nef polyhedron. Most newer textbook recommend the use of doubly connected edge lists (DCEL) or equivalently half edge data structures (HDS) [PS85, dBvKOS97] when they discuss the implementation of plane maps. There are already standard implementations of *plane maps* like the CGAL HDS or embedded bidirected graphs in LEDA (similar design). We use an extended version of the CGAL HDS structure as the topological bottom layer of our Nef polyhedra. See the paper of L. Kettner [Ket99] for an excellent review of different plane map representation and for the description of the adaptability of the CGAL HDS. To be more flexible we insert a decorator interface that homogeneously defines the functionality offered by the HDS. Geometrically we embed the vertices by means of extended points. Segments, rays, and lines are uniformly treated by the straight line embedding of edges incident to such vertices. We additionally add one face cycle of edges whose embedding corresponds to an infimaximal frame:

Construction 1: Consider a Nef polyhedron P and the connected components of the faces of dimension zero to two. Assign plane map objects of corresponding dimension to each component and match the Nef incidence concept with the plane map incidence concept where possible. All objects corresponding to unbounded connected components of Nef edges and unbounded components of Nef 2-faces have incomplete incidence structures: edges miss end vertices and faces miss closed face cycles. To cure this, we add an infimaximal frame consisting of four uedges and four corner vertices. For all plane map edges e that correspond to a Nef edge component extending to infinity along a ray r we do the following: if r is pruned by one of the corner vertices on the frame structure link e to that vertex; otherwise e obtains an additional terminating vertex in the relative interior of a uedge e' that is part of the infimaximal frame where r is meets the frame. e' is split into two uedges by this vertex insertion. (For Nef edges that represent lines we do this at both ends). After all such edges are linked to the frame (respecting the embedding such that the adjacency list are order-preserving), all plane map faces corresponding to an unbounded component of a Nef face are cyclically bounded and their incidences structure can be completed. We call all edges and vertices that are part of the frame and the face outside of the frame that completes the subdivision combinatorially *infimaximal frame objects*.

To mark set membership, all objects of the plane map are *selectable*. All objects (vertices, edges, faces) obtain a marker labeling set inclusion or exclusion. The markers allow us to obtain the local pyramids associated to the plane map objects. The local view of a vertex is defined by its own marker, the markers of the edges in its adjacency list and markers of the faces in between these edges representing the neighborhood disc as explained above. The local view of an edge is defined by its mark and the two marks of its two incident faces. Finally the mark of a face maps to the trivial local view: the whole plane or the empty set depending on its selection flag. The selection markers of infimaximal frame objects have no geometric meaning and therefore those objects are always kept unselected.

As plane maps are implemented by bidirected graphs the incidence relation between edges and faces is encoded in an oriented fashion. Unoriented edges are implemented as pairs of oppositely directed halfedges where each such halfedge is incident to exactly one face.

Definition 4 (Data type): A Nef polyhedron P is stored as a selective plane map (V,E,F) according to Construction 1. The objects of the plane map (vertices, edges, and faces) correspond to the connected components of the Nef faces of corresponding dimension and additionally to the infimaximal frame objects.

Each vertex $v \in V$ is embedded via an extended point $point(v)$. Each object $o \in V \cup E \cup F$ is contained in P iff the selection mark $mark(o) == true$.

The feasibility of this definition can be seen as follows. Interpret P as a set valued function ϕ on (open) half-spaces $H_1, \ldots, H_r$ that are combined by the operations $\cap$ and cpl. Let h_i be the line bounding the half-space H_i. Now consider the arrangement build by the lines $\{h_i\}_i$ enclosed in a large enough frame. Interpret the arrangement $A(h_1, \ldots, h_r)$ inside the frame as a collection of relatively open convex cells of dimension 0 to 2. Consider any cell c. Any point of c is either in $\phi(H_1, \ldots, H_r)$ or not. Mark all cells correspondingly. It should be clear that $A(h_1, \ldots, h_r)$ can be represented by a plane map as described above. Now start a simplification process. Consider all edges e (besides the ones on the frame box). If e and the two faces incident to it have the same mark remove the edge and unify the faces (if not equal). Afterwards we iterate over all vertices and check if any vertex incident to two edges that are supported by the same line has the same local view as the points in the relative interior of the two incident edges. If this is the case, we unify the two edges and remove the vertex. Finally, we remove all vertices that are isolated and whose selection mark equals the one of the surrounding face. The final complex is just the data type described. When the simplification iteration terminates all points on vertices and edges have a local view that makes them low-dimensional faces of the Nef polyhedron.

The above algorithm is called *simplification* and is described in more detail when considering binary operations. There, we also give a runtime bound. Note that the local view properties of the vertices, edges, and faces of the plane map after the simplification process are a necessary condition of Definition 4.

Lemma 1.5.1: The representation of Definition 4 is unique.

Proof. Assume that there are two different plane maps $M(V, E, F)$ and $M'(V', E', F')$ representing the same Nef polyhedron P. If P is the complete plane or the empty set then the plane maps consist of just one face inside the frame box and F and F' and their selection markers have to be equal. So assume otherwise. Then neither P nor $\mathsf{cpl}\,P$ are empty. The boundary of P and $\mathsf{cpl}\,P$ consists of vertices and edges. Assume that M and M' have a vertex at a point x but the local views differ. Then, the represented point sets of M and M' differ and thus P cannot be represented by both. If there is a point x where M has a vertex v but M' has not then obviously the local views are different too. Note that edges are terminated by vertices, and thus edges that are in M but not in M' already imply differences in the local view of their end vertices. The same holds when the edges are equal but the selection markers are not. Finally, note that due to the fact that the 1-skeleton of M and M' is equal, so are the faces (they are defined that way). Different selection markers on faces imply different local views in the vertices that are part of their closure. As a consequence M and M' have to be equal. □

Selective plane maps are the basic structure used to store Nef polyhedra. Remember that the edges and faces of a plane map are the connected components of the Nef edges and Nef 2-faces. For performance reasons we do not maintain the relationship between plane map objects and the corresponding abstract Nef faces which are defined as collections of the plane map objects with the same implicitly stored local pyramids. The *size* of a Nef polyhedron is the size of its underlying plane map (which is the number of vertices, edges, and faces).

For the binary operations we follow an approach as presented by Rossignac et al. [RO90]. In our case the approach is based on a generic plane sweep framework. A binary operation is basically split into three phases: subdivision – selection – simplification. The implementation is presented in the module *PM_overlayer*. An unary (topological) operation can be subdivided into two phases: selection – simplification.

We shortly present the abstract algorithmic ideas and the runtimes.

subdivision means for two plane maps $P_i (i = 0, 1)$ to create a plane map P with a minimal number of objects (vertices, edges, faces) such that each object of P is supported by exactly one object of P_i for $i = 0, 1$. The subdivision is realized by a plane sweep of the objects of the 1-skeleta of P_i followed by face creation and support determination. The time is dominated by the time for the sweep phase which is $O(n \log n)$ where n is the size of the resulting subdivision.

selection with respect to the binary set operations means selecting the cells of the subdivision according to the logic of the underlying Boolean operation. With respect to unary topological set operations selection happens according to the logic of the topological unary operation. Selection is in both cases linear in n.

simplification means unification of subsets of cells that have the same local view. This phase has a quasi linear runtime due to the usage of a union-find data structure. Runtime is $O(n\alpha(kn,n))$[7] for some small constant k.

Theorem 1.5.1: The result of a binary set operation (intersection, union, difference, symmetric difference) of two Nef polyhedra P_0 and P_1 can be calculated in time $O(n \log n)$ where n is the size of the overlay of P_0 and P_1. The result of an unary set operation (complement, boundary, interior, closure, regularization) can be constructed in time $O(n\alpha(kn,n))$ where n is the size of the input structure.

The correctness and time bounds of our binary operations are based on three parts:

- The Sections 1.6.3 and 1.6.4 show the high-level composition of unary and binary set operations decomposed into phases.
- Chapter 3 provides the algorithmic modules for the subdivision, selection, and simplification phase. The correctness and resource argumentation is purely based on affine concepts and therefore our readers can trust their standard geometric intuition when verifying the correctness of those modules.
- The report [SM00] on the other hand shows that infimaximal frames allow us to use these algorithmic modules together with our extended objects.

The latter two observations together imply the correctness of the above theorem. The runtime lemmata of Chapter 3 imply the time bounds.

Now for our additional functionality like point location and ray shooting queries we need more than just the naked plane map structure described above. Looking at the literature for ray shooting there is the notion of segment walks which can be done easiest in convex subdivisions of the plane of bounded complexity [MMS94]. Our goal is to refine the basic plane map by such a structure. Note that this structure implies again faces, edges, and vertices, but of a much simpler fashion. However we do not want to lose the original character of the plane map. We might still be interested in the original face cycles. We store the refinement separate from the plane map. One solution to the segment walk

[7] $\alpha(kn,n)$ is the extremly slow growing inverse of the Ackermann function used in the analysis of union-find data structures.

problem is to use a constrained Delaunay triangulation of the edges and vertices of the HDS, and use any efficient point location structure for the location of the ray shooting start point. Our approach is presented in Section 5.1.2.

1.6 Top Level Implementation

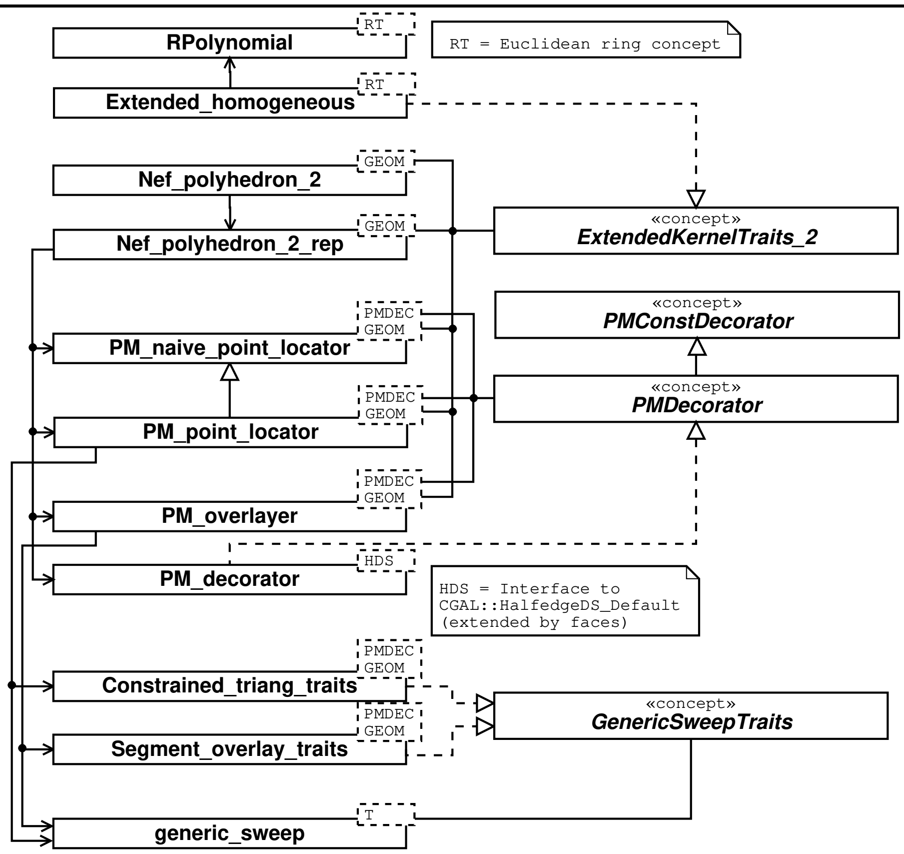


Figure 1.3: An UML diagram of the Nef polyhedron module. The main modules are the geometry *Extended_homogenous*<*RT*>, the plane map decorator *PM_decorator*<*HDS*>, the two algorithmic modules *PM_overlayer*<*PMDEC*, *GEOM*> and *PM_point_locator*<*PMDEC*, *GEOM*>.

The whole implementation scheme is depicted in Figure 1.3. The main classes map to the abstract layers described above: geometry, plane maps, binary overlay, and point location.

1.6.1 The Polyhedron Class

⟨*nef polyhedron definition*⟩≡

```
template <typename T> class Nef_polyhedron_2;
template <typename T> class Nef_polyhedron_2_rep;

template <typename T>
```

```
std::ostream& operator<<(std::ostream&, const Nef_polyhedron_2<T>&);
template <typename T>
std::istream& operator>>(std::istream&, Nef_polyhedron_2<T>&);
```

Our data type *Nef_polyhedron_2*<*T*> is implemented as a smart pointer data type. Content such as a plane map object and a pointer to an optional point location object is stored in the class *Nef_polyhedron_2_rep*<*T*>. The traits template parameter *T* is specified by the concept *ExtendedKernelTraits_2* as presented on page 328.

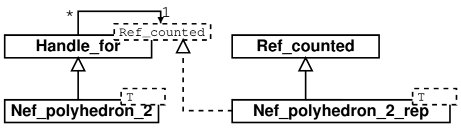


Figure 1.4: The smart pointer realization of data type *Nef_polyhedron_2*.

Within the scope of *Nef_polyhedron_2_rep*<*T*> all auxiliary classes are instantiated. The plane map type is based on the CGAL HDS and uses two traits classes *HDS_traits* and *HDS_items*. The former carries attributes the latter carries the (fixed) models for vertices, edges, and faces.

⟨*nef rep types*⟩≡
```
struct HDS_traits {
  typedef typename T::Point_2 Point;
  typedef bool                Mark;
};
typedef CGAL_HALFEDGEDS_DEFAULT<HDS_traits,HDS_items> Plane_map;
typedef CGAL::PM_const_decorator<Plane_map>           Const_decorator;
typedef CGAL::PM_decorator<Plane_map>                 Decorator;
typedef CGAL::PM_naive_point_locator<Decorator,T>     Slocator;
typedef CGAL::PM_point_locator<Decorator,T>           Locator;
typedef CGAL::PM_overlayer<Decorator,T>               Overlayer;
```

For Microsoft we use a hardwired Halfedge data structure specially adapted to its weaknesses.

⟨*nef rep types msc*⟩≡
```
struct HDS_traits {
  typedef typename T::Point_2 Point;
  typedef bool                Mark;
};
typedef CGAL::HalfedgeDS_default_MSC<HDS_traits>  Plane_map;
typedef CGAL::PM_const_decorator<Plane_map>       Const_decorator;
typedef CGAL::PM_decorator<Plane_map>             Decorator;
typedef CGAL::PM_naive_point_locator<Decorator,T> Slocator;
typedef CGAL::PM_point_locator<Decorator,T>       Locator;
typedef CGAL::PM_overlayer<Decorator,T>           Overlayer;
```

⟨*nef polyhedron definition*⟩+≡
```
template <typename T>
class Nef_polyhedron_2_rep : public Ref_counted
```

```
{ typedef Nef_polyhedron_2_rep<T> Self;
  friend class Nef_polyhedron_2<T>;
#ifndef CGAL_SIMPLE_HDS
  ⟨nef rep types⟩
#else
  ⟨nef rep types msc⟩
#endif
  //typedef CGAL::PM_transformer<Decorator,T> Transformer;
  Plane_map pm_; Locator* pl_;
  void init_locator()
  { if ( !pl_ ) pl_ = new Locator(pm_); }
  void clear_locator()
  { if ( pl_ ) delete pl_; pl_=0; }
public:
  Nef_polyhedron_2_rep() : Ref_counted(), pm_(), pl_(0) {}
  Nef_polyhedron_2_rep(const Self& R) : Ref_counted(), pm_(), pl_(0) {}
  ~Nef_polyhedron_2_rep() { pm_.clear(); clear_locator(); }
};
```

The class *Nef_polyhedron_2*<*T*> has a static member object *EK* of type *T* which allows us to interface a kernel object.

⟨*nef polyhedron definition*⟩+≡

```
template <typename T>
class Nef_polyhedron_2 : public Handle_for< Nef_polyhedron_2_rep<T> >
{
public:
typedef T Extended_kernel;
static  T EK; // static extended kernel
  ⟨nef interface types⟩
protected:
  ⟨nef protected members⟩
public:
  ⟨nef interface operations⟩
}; // end of Nef_polyhedron_2
template <typename T>
T Nef_polyhedron_2<T>::EK;
```

In the class *Nef_polyhedron_2*<*T*> all geometric types are obtained from the geometric traits class *T*. *T* contains affine types that are part of the interface but also the extended types that are used in the infimaximal framework. The *Standard*-prefixed types from within *T* become the interface types of *Nef_polyhedron_2*<*T*>. The non-prefixed[8] types within *T* become the extended types within *Nef_polyhedron_2*<*T*>.

[8] T is used as a traits class in our generic geometry based modules like *PM_overlayer*<*T*>. Therefore, the extended types conform to a simpler naming scheme within *T*.

⟨*nef interface types*⟩≡
```
typedef Nef_polyhedron_2<T> Self;
typedef Handle_for< Nef_polyhedron_2_rep<T> > Base;
typedef typename T::Point_2   Extended_point;
typedef typename T::Segment_2 Extended_segment;
typedef typename T::Standard_line_2 Line;
typedef typename T::Standard_point_2 Point;
typedef typename T::Standard_direction_2 Direction;
typedef typename T::Standard_aff_transformation_2  Aff_transformation;
```

⟨*nef interface types*⟩+≡
```
typedef bool Mark;
enum Boundary { EXCLUDED=0, INCLUDED=1 };
enum Content { EMPTY=0, COMPLETE=1 };
```

We import the type *Plane_map* and all decorator types like *Decorator*, *Overlayer*, *Locator*, *Slocator* from the representation type *Nef_polyhedron_2_rep*. Additionally, we import handles and iterators from *Decorator* .

⟨*nef protected members*⟩≡
⟨*boolean classes*⟩
```
typedef Nef_polyhedron_2_rep<T>           Nef_rep;
typedef typename Nef_rep::Plane_map       Plane_map;
typedef typename Nef_rep::Decorator       Decorator;
typedef typename Nef_rep::Const_decorator Const_decorator;
typedef typename Nef_rep::Overlayer       Overlayer;
//typedef typename Nef_rep::T               Transformer;
typedef typename Nef_rep::Slocator        Slocator;
typedef typename Nef_rep::Locator         Locator;
Plane_map& pm() { return ptr->pm_; }
const Plane_map& pm() const { return ptr->pm_; }
friend std::ostream& operator<< CGAL_NULL_TMPL_ARGS
    (std::ostream& os, const Nef_polyhedron_2<T>& NP);
friend std::istream& operator>> CGAL_NULL_TMPL_ARGS
    (std::istream& is, Nef_polyhedron_2<T>& NP);
typedef typename Decorator::Vertex_handle         Vertex_handle;
typedef typename Decorator::Halfedge_handle       Halfedge_handle;
typedef typename Decorator::Face_handle           Face_handle;
typedef typename Decorator::Vertex_const_handle   Vertex_const_handle;
typedef typename Decorator::Halfedge_const_handle Halfedge_const_handle;
typedef typename Decorator::Face_const_handle     Face_const_handle;
typedef typename Decorator::Vertex_iterator       Vertex_iterator;
typedef typename Decorator::Halfedge_iterator     Halfedge_iterator;
typedef typename Decorator::Face_iterator         Face_iterator;
typedef typename Const_decorator::Vertex_const_iterator
                                                  Vertex_const_iterator;
typedef typename Const_decorator::Halfedge_const_iterator
```

```
                                                  Halfedge_const_iterator;
typedef typename Const_decorator::Face_const_iterator
                                                  Face_const_iterator;

struct Except_frame_box_edges {
Decorator D_; Face_handle f_;
Except_frame_box_edges(Plane_map& P) : D_(P), f_(D_.faces_begin()) {}
bool operator()(Halfedge_handle e) const
{ return D_.face(e)==f_ || D_.face(D_.twin(e))==f_; }
};
```

1.6.2 Creating Polyhedra

We provide the construction methods for basic polyhedra. These are the empty set, the whole plane, open and closed half-planes, and construction of simple polygonal chains (Jordan Curves) where the modeled point set can be the bounded or unbounded part for the plane and the set can be open or closed.

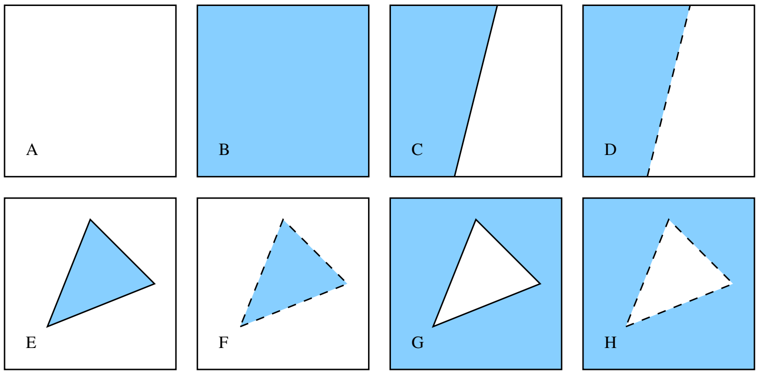


Figure 1.5: Elementary Nef polyhedra: (A) the empty set, (B) the whole plane, (C/D) a closed/open half-plane, (E/F) a closed/open bounded polygon, (G/H) the plane with a closed/open polygonal hole.

The construction of simple Nef polyhedra is reduced to the overlay of a list of extended segments, the creation of the 2-faces, followed by setting the attribute marks that code set inclusion. The first task is implemented in our overlayer module *PM_overlayer*<>::$create(s,e,DA)$ where $S = tuple[s,e]$ is the set of segments and *DA* is a data accessor that allows us to link plane map edges to the segments in S.

Finally, we only have to take care of the correct marks of the plane map objects with respect to the construction information from our constructor interface. Note that by using the overlayer module we obtain all output properties of the plane map created from that module.

All Nef polyhedra obtain an infimaximal frame embedded by four extended segments. We encapsulate this into the following operation. See the extended geometry module for the definition of this frame.

⟨*nef protected members*⟩+≡
```
typedef std::list<Extended_segment>      ES_list;
typedef typename ES_list::const_iterator ES_iterator;

void fill_with_frame_segs(ES_list& L) const
{ L.push_back(Extended_segment(EK.SW(),EK.NW()));
  L.push_back(Extended_segment(EK.SW(),EK.SE()));
  L.push_back(Extended_segment(EK.NW(),EK.NE()));
  L.push_back(Extended_segment(EK.SE(),EK.NE()));
}
```

We want to establish a link between a particular extended segment and its corresponding edge in the plane map. Our overlay module allows us to get a grip on this relation by means of a data accessor that is passed to the overlay algorithm. The class *Link_to_iterator* is a model for that data accessor concept. An object D of this type can store an iterator referencing a segment. Passed to the *PM_overlayer*<>::*create*(...) method it stores the corresponding edge after the executed overlay as its member *D_e*. *Link_to_iterator* also initializes all marks of the newly created skeleton objects.

We show more details. We allow also degenerate segments. We associate a halfedge or vertex to an input segment $s = *it$ referenced by an iterator *it*. If the segment s is trivial it will be associated to a vertex, else it is associated to a halfedge. The data accessor initializes the marks of the newly created skeleton objects to m (defined in the construction). For the concept of the data accessor see the manual page of *PM_overlayer*<>::*create*(...).

⟨*nef protected members*⟩+≡
```
struct Link_to_iterator {
  const Decorator& D;
  Halfedge_handle _e;
  Vertex_handle   _v;
  ES_iterator     _it;
  Mark            _m;
  Link_to_iterator(const Decorator& d, ES_iterator it, Mark m) :
    D(d), _e(), _v(), _it(it), _m(m) {}

  void supporting_segment(Halfedge_handle e, ES_iterator it)
  { if ( it == _it ) _e = e; D.mark(e) = _m; }
  void trivial_segment(Vertex_handle v, ES_iterator it)
  { if ( it == _it ) _v = v; D.mark(v) = _m; }
  void starting_segment(Vertex_handle v, ES_iterator)
  { D.mark(v) = _m; }
  void passing_segment(Vertex_handle v, ES_iterator)
  { D.mark(v) = _m; }
  void ending_segment(Vertex_handle v, ES_iterator)
  { D.mark(v) = _m; }

};
```

We add the box edges to L, overlay them and set the second face (inside the frame box) to the *plane* mark. We know from the output properties of *Overlayer* that the first face object is always the one surrounding the frame.

⟨*nef interface operations*⟩≡
```
Nef_polyhedron_2(Content plane = EMPTY) : Base(Nef_rep())
{
  ES_list L;
  fill_with_frame_segs(L);
  Overlayer D(pm());
  Link_to_iterator I(D, --L.end(), false);
  D.create(L.begin(),L.end(),I);
  D.mark(++D.faces_begin()) = bool(plane);
}
```

We come to the construction of a half-plane. A user describes an open or closed half-plane by an oriented line l. To create a plane map representing it we overlay a frame box plus an extended segment splitting the box into two faces along l. The overlayer module *PM_overlayer*<>::*create*(...) creates the plane map (including faces) out of the list of extended segments passed to it where no object is selected (concerning membership). The data accessor *Link_to_iterator I* obtains the edge *I._e* of *pm*( ) corresponding to $--$*L.end*( ) (the iterator pointing to the extended segment which is the line) during the *create* phase. We can use that edge to mark its adjacent face and the edge itself according to the *line* flag.

⟨*nef interface operations*⟩+≡
```
Nef_polyhedron_2(const Line& l, Boundary line = INCLUDED) : Base(Nef_rep())
{
  ES_list L;
  fill_with_frame_segs(L);
  Extended_point ep1 = EK.construct_opposite_point(l);
  Extended_point ep2 = EK.construct_point(l);
  L.push_back(EK.construct_segment(ep1,ep2));
  Overlayer D(pm());
  Link_to_iterator I(D, --L.end(), false);
  D.create(L.begin(),L.end(),I);
  Halfedge_handle el = I._e;
  if ( D.point(D.target(el)) != EK.target(L.back()) )
    el = D.twin(el);
  D.mark(D.face(el)) = true;
  D.mark(el) = bool(line);
}
```

The construction of a simple polygon defined by an iterator range of standard affine points (the value type of *Forward_iterator* is *Point*) follows the same idea, however we also accept degenerate polygons.

The iterator range of points can confront us with the following cases: (1) the list is empty, (2) the list has only one point, (3) the list contains at least two points spanning line segments where the following cases and problems can occur: (a) the segment(s) has (have) affine dimension 1 (the hull is a segment). (b) the segments enclose a simple polygon. (c) the segments enclose no simple polygon (touching or intersecting inside). We cover one point, two points spanning a segment and n points spanning a simple polygon (which we do not check). The construction of the plane map still succeeds when P is not simple as the overlayer module just constructs the planar subdivision implied by the segments in the iterator range. However, the face marks will in general be incorrect.

⟨*nef interface operations*⟩+≡
```
template <class Forward_iterator>
Nef_polyhedron_2(Forward_iterator it, Forward_iterator end,
  Boundary b = INCLUDED) : Base(Nef_rep())
{
  ES_list L;
  fill_with_frame_segs(L);
  bool empty = false;
  if (it != end)
    ⟨fill segment list L⟩
  else empty = true;
  Overlayer D(pm());
  Link_to_iterator I(D, --L.end(), true);
  D.create(L.begin(),L.end(),I);
  ⟨mark face and boundary⟩
}
```

We fill L with the segments cyclically spanned by the points in the input iterator range.

⟨*fill segment list L*⟩≡
```
{
  Extended_point ef, ep = ef = EK.construct_point(*it);
  Forward_iterator itl=it; ++itl;
  if (itl == end) // case only one point
    L.push_back(EK.construct_segment(ep,ep));
  else { // at least one segment
    while( itl != end ) {
      Extended_point en = EK.construct_point(*itl);
      L.push_back(EK.construct_segment(ep,en));
      ep = en; ++itl;
    }
    L.push_back(EK.construct_segment(ep,ef));
  }
}
```

We create the marks via the object stored in the *Link_to_iterator* object I. The object is determined by the last segment s in L. If that segment is trivial then $I.\_v$ stores the corresponding vertex. If s is non-trivial then $I.\_e$ contains the edge supported by the segment. We only have to extract the correct halfedge of the edge twins. Then, we can mark the face and the boundary accordingly.

⟨*mark face and boundary*⟩≡
```
if ( empty ) {
  D.mark(++D.faces_begin()) = !bool(b); return; }
if ( EK.is_degenerate(L.back()) ) {
  D.mark(D.face(I._v)) = !bool(b); D.mark(I._v) = b;
} else {
  Halfedge_handle el = I._e;
  if ( D.point(D.target(el)) != EK.target(L.back()) )
    el = D.twin(el);
  D.set_marks_in_face_cycle(el,bool(b));
  if ( D.number_of_faces() > 2 ) D.mark(D.face(el)) = true;
```

```
  else                          D.mark(D.face(el)) = !bool(b);
}
clear_outer_face_cycle_marks();
```

Now the standard copy construction, assignment and destruction. The first two are delegated to the *Handle* type.

⟨*nef interface operations*⟩+≡

```
Nef_polyhedron_2(const Nef_polyhedron_2<T>& N1) : Base(N1) {}
Nef_polyhedron_2& operator=(const Nef_polyhedron_2<T>& N1)
{ Base::operator=(N1); return (*this); }
~Nef_polyhedron_2() {}
#ifndef _MSC_VER
template <class Forward_iterator>
Nef_polyhedron_2(Forward_iterator first, Forward_iterator beyond,
  double p=0.5) : Base(Nef_rep())
{
  ES_list L; fill_with_frame_segs(L);
  while ( first != beyond ) {
    Extended_point ep1 = EK.construct_opposite_point(*first);
    Extended_point ep2 = EK.construct_point(*first);
    L.push_back(EK.construct_segment(ep1,ep2)); ++first;
  }
  Overlayer D(pm());
  Link_to_iterator I(D, --L.end(), false);
  D.create(L.begin(),L.end(),I);
  Vertex_iterator v; Halfedge_iterator e; Face_iterator f;
  for (v = D.vertices_begin(); v != D.vertices_end(); ++v)
    D.mark(v) = ( default_random.get_double() < p ? true : false );
  for (e = D.halfedges_begin(); e != D.halfedges_end(); ++(++e))
    D.mark(e) = ( default_random.get_double() < p ? true : false );
  for (f = D.faces_begin(); f != D.faces_end(); ++f)
    D.mark(f) = ( default_random.get_double() < p ? true : false );
  D.simplify(Except_frame_box_edges(pm()));
  clear_outer_face_cycle_marks();
}
#endif
```

The construction from a plane map H is a clone action. We create a representation object which has the same topology, geometry and attributes as H.

⟨*nef interface operations*⟩+≡

```
protected:
Nef_polyhedron_2(const Plane_map& H, bool clone=true) : Base(Nef_rep())
{ if (clone) {
    Decorator D(pm()); // a decorator working on the rep plane map
    D.clone(H);        // cloning H into pm()
  }
}
void clone_rep() { *this = Nef_polyhedron_2<T>(pm()); }
```

And now for the standard operations. We expect that the underlying plane map structure is simplified as described in the *PM_overlayer*<> module. Then for the simple configurations *empty* and *plane* the plane map has to be just the frame of four vertices, four uedges and two faces.

⟨*nef interface operations*⟩+≡
```
public:
void clear(Content plane = EMPTY)
{ *this = Nef_polyhedron_2(plane); }
bool is_empty() const
{ Const_decorator D(pm());
  Face_const_iterator f = D.faces_begin();
  return (D.number_of_vertices()==4 &&
          D.number_of_edges()==4 &&
          D.number_of_faces()==2 &&
          D.mark(++f) == false);
}
bool is_plane() const
{ Const_decorator D(pm());
  Face_const_iterator f = D.faces_begin();
  return (D.number_of_vertices()==4 &&
          D.number_of_edges()==4 &&
          D.number_of_faces()==2 &&
          D.mark(++f) == true);
}
```

1.6.3 Unary Operations

For the unary operations on Nef polyhedra we implement several class-modifying methods. They can be chained together to larger units. We thereby save cloning operations if the representation object is only referenced by one handle. Note that the first line of any modifying operation has to check if the representation object is shared by several handles. If the representation object is shared, the plane map has to be cloned before modification. We first implement the three operations **cpl**, **int**, and **bd**.

As our planar Nef polyhedra completely partition the plane, the complement operation is easy to implement by an inversion (Boolean flip) of the selection markers. Note that this conforms to the result [Nef78, theorem 6;14] in which Nef showed that the low-dimensional faces of P and $\mathsf{cpl}\,P$ (in their common boundary) are the same and the full-dimensional faces that are part of P or $\mathsf{cpl}\,P$ obviously exchange their role (if non-empty). The local pyramid in any point x of the plane is inverted by the Boolean flips according to [Nef78, theorem 3;15] $(\mathsf{cpl}\,P)^x = \mathsf{cpl}\,P^x$. Due to the special role[9] of the outer face and the edges and vertices that are part of the frame box we keep all objects of the frame unmarked (operation *clear_outer_face_cycle_marks*). Note that just by flipping we do not spoil the local views properties as specified in Lemma 1.4.1. Thus, we do not have to simplify here.

⟨*nef interface operations*⟩+≡
```
void extract_complement()
{
  if ( ptr->is_shared() ) clone_rep();
```

[9] No affine object can be placed on or outside the infimaximal frame.

```
    Overlayer D(pm());
    Vertex_iterator v, vend = D.vertices_end();
    for(v = D.vertices_begin(); v != vend; ++v)      D.mark(v) = !D.mark(v);
    Halfedge_iterator e, eend = D.halfedges_end();
    for(e = D.halfedges_begin(); e != eend; ++(++e)) D.mark(e) = !D.mark(e);
    Face_iterator f, fend = D.faces_end();
    for(f = D.faces_begin(); f != fend; ++f)         D.mark(f) = !D.mark(f);
    clear_outer_face_cycle_marks();
  }
```

The interior $\mathsf{int}\, P$ of a point set P is the set of all points where an open ball of infinitesimal radius (a neighborhood) is contained in the point set. This is not the case for all low-dimensional faces of the plane map. Accordingly (see also [Nef78, theorem 3;19]), we have to keep all selected full-dimensional faces of P and all objects of the 1-skeleton have to be deselected. Afterwards the simplification operation minimizes the structure and makes it again consistent with Definition 4. For example, marked isolated vertices within non-marked faces and marked edges within such faces are first unmarked and then deleted in the *simplify*( ) operation. The simplification operation has to exempt edges of the infimaximal frame. An object *Except_frame_box_edges*(P) has a function operator method *bool operator*( )$(Halfedge_handle\ e)$that returns true iff the edge e of the plane map P is part of the frame box. Thereby within *simplify* a removal of edges is only executed on edges that partition the interior of the frame box.

⟨*nef interface operations*⟩+≡
```
  void extract_interior()
  {
    if ( ptr->is_shared() ) clone_rep();
    Overlayer D(pm());
    Vertex_iterator v, vend = D.vertices_end();
    for(v = D.vertices_begin(); v != vend; ++v)      D.mark(v) = false;
    Halfedge_iterator e, eend = D.halfedges_end();
    for(e = D.halfedges_begin(); e != eend; ++(++e)) D.mark(e) = false;
    D.simplify(Except_frame_box_edges(pm()));
  }
```

The boundary of a point set P is defined to be the intersection $\mathsf{clos}\, P \cap \mathsf{clos}\,\mathsf{cpl}\, P$. This is the set of all points that have a nonempty neighborhood with $\mathsf{int}\, P$ or $\mathsf{ext}\, P$. Any point x on the 1-skeleton of the plane map has this property due to the properties of its local pyramid P^x. The boundary $\mathsf{bd}\, P$ is thus obtained by selecting all low-dimensional objects and then deselecting all 2-faces. Finally we cope with the frame and simplify the structure.

⟨*nef interface operations*⟩+≡
```
  void extract_boundary()
  {
    if ( ptr->is_shared() ) clone_rep();
    Overlayer D(pm());
    Vertex_iterator v, vend = D.vertices_end();
    for(v = D.vertices_begin(); v != vend; ++v)      D.mark(v) = true;
    Halfedge_iterator e, eend = D.halfedges_end();
    for(e = D.halfedges_begin(); e != eend; ++(++e)) D.mark(e) = true;
    Face_iterator f, fend = D.faces_end();
```

```
    for(f = D.faces_begin(); f != fend; ++f)        D.mark(f) = false;
    clear_outer_face_cycle_marks();
    D.simplify(Except_frame_box_edges(pm()));
  }
```

Finally, we use the above operations for *closure* and *regularization*. The closure of P can be reduced to the operations *interior* and *complement* as $\mathsf{clos}\, P = \mathsf{cpl}\,\mathsf{int}\,\mathsf{cpl}\, P$. The regularization of P is defined as $\mathsf{clos}\,\mathsf{int}\, P$.

⟨*nef interface operations*⟩+≡

```
  void extract_closure()
  {
    extract_complement();
    extract_interior();
    extract_complement();
  }
  void extract_regularization()
  {
    extract_interior();
    extract_closure();
  }
```

The constructive interface methods are just mapped to the corresponding extract methods.

⟨*nef interface operations*⟩+≡

```
  Nef_polyhedron_2<T> complement() const
  { Nef_polyhedron_2<T> res = *this;
    res.extract_complement();
    return res;
  }
```

All other operations like *interior*( ), *closure*( ), *boundary*( ), and *regularization*( ) are implemented accordingly.

⟨*nef interface operations*⟩+≡

```
  Nef_polyhedron_2<T> interior() const
  { Nef_polyhedron_2<T> res = *this;
    res.extract_interior();
    return res;
  }
  Nef_polyhedron_2<T> closure() const
  { Nef_polyhedron_2<T> res = *this;
    res.extract_closure();
    return res;
  }
  Nef_polyhedron_2<T> boundary() const
  { Nef_polyhedron_2<T> res = *this;
    res.extract_boundary();
    return res;
  }
```

```
Nef_polyhedron_2<T> regularization() const
{ Nef_polyhedron_2<T> res = *this;
  res.extract_regularization();
  return res;
}
```

1.6.4 Binary Set Operations

We follow Rossignac and O'Connor [RO90] and split the binary operations into three phases subdivision – selection – simplification. The subdivision phase creates the overlay of the two input structures. This overlay has the property that each object (vertex, edge, face) has exactly one object from each input structure that supports it. After the subdivison each object of the resulting plane map knows its mark in each of the two input structures and can thereby be qualified with respect to each input structure. The binary set operation is then reduced to a Boolean predicate on these marks. The resulting structure can be a planar partition that is not a legal Nef polyhedron due to the fact that plane map boundary objects can have local views that contradict Lemma 1.4.1. Violations are fixed in the simplification phase without changing the represented point set. This simplification makes the plane map representation minimal with respect to the number of its objects and again consistent with Definition 4.

The above scheme refers to the theory as presented by Nef who showed the following general lemma.

Lemma 1.6.1: Let P_0, P_1 be polyhedra. Then every face of $P = P_0 \cap P_1$ is the union of intersections of faces of P_0 and P_1.

The proof follows [Nef78, Satz 6;16]. As a consequence the simplification just unions objects within P to form the connected components of Nef faces.

To implement the binary set operations we use functors[10] that carry the underlying Boolean logic.

⟨*boolean classes*⟩≡
```
struct AND { bool operator()(bool b1, bool b2)  const { return b1&&b2; }  };
struct OR { bool operator()(bool b1, bool b2)   const { return b1||b2; }  };
struct DIFF { bool operator()(bool b1, bool b2) const { return b1&&!b2; } };
struct XOR { bool operator()(bool b1, bool b2)  const
                                       { return (b1&&!b2)||(!b1&&b2); } };
```

⟨*nef interface operations*⟩+≡
```
Nef_polyhedron_2<T> intersection(const Nef_polyhedron_2<T>& N1) const
{ Nef_polyhedron_2<T> res(pm(),false); // empty, no frame
  Overlayer D(res.pm());
  D.subdivide(pm(),N1.pm());
  AND _and; D.select(_and);
  res.clear_outer_face_cycle_marks();
  D.simplify(Except_frame_box_edges(res.pm()));
  return res;
}
```

[10] a short form for function objects.

Join, difference, and symmetric difference follow similar schemes based on *OR*, *DIFF*, and *XOR*.

⟨*nef interface operations*⟩+≡
```
Nef_polyhedron_2<T> join(const Nef_polyhedron_2<T>& N1) const
{ Nef_polyhedron_2<T> res(pm(),false); // empty, no frame
  Overlayer D(res.pm());
  D.subdivide(pm(),N1.pm());
  OR _or; D.select(_or);
  res.clear_outer_face_cycle_marks();
  D.simplify(Except_frame_box_edges(res.pm()));
  return res;
}
Nef_polyhedron_2<T> difference(const Nef_polyhedron_2<T>& N1) const
{ Nef_polyhedron_2<T> res(pm(),false); // empty, no frame
  Overlayer D(res.pm());
  D.subdivide(pm(),N1.pm());
  DIFF _diff; D.select(_diff);
  res.clear_outer_face_cycle_marks();
  D.simplify(Except_frame_box_edges(res.pm()));
  return res;
}
Nef_polyhedron_2<T> symmetric_difference(
  const Nef_polyhedron_2<T>& N1) const
{ Nef_polyhedron_2<T> res(pm(),false); // empty, no frame
  Overlayer D(res.pm());
  D.subdivide(pm(),N1.pm());
  XOR _xor; D.select(_xor);
  res.clear_outer_face_cycle_marks();
  D.simplify(Except_frame_box_edges(res.pm()));
  return res;
}
#if 0
Nef_polyhedron_2<T> transform(const Aff_transformation& t) const
{ Nef_polyhedron_2<T> res(pm()); // cloned
  Transformer PMT(res.pm());
  PMT.transform(t);
  return res;
}
#endif
```

The first face object is the one outside the bounding frame. Thus it's only hole face cycle consists of the edges of the frame.

⟨*nef protected members*⟩+≡
```
void clear_outer_face_cycle_marks()
{ // unset all frame marks
  Decorator D(pm());
  Face_iterator f = D.faces_begin();
  D.mark(f) = false;
```

```
    Halfedge_handle e = D.holes_begin(f);
    D.set_marks_in_face_cycle(e, false);
  }
```

⟨*nef interface operations*⟩+≡

```
Nef_polyhedron_2<T>  operator*(const Nef_polyhedron_2<T>& N1) const
{ return intersection(N1); }
Nef_polyhedron_2<T>  operator+(const Nef_polyhedron_2<T>& N1) const
{ return join(N1); }
Nef_polyhedron_2<T>  operator-(const Nef_polyhedron_2<T>& N1) const
{ return difference(N1); }
Nef_polyhedron_2<T>  operator^(const Nef_polyhedron_2<T>& N1) const
{ return symmetric_difference(N1); }
Nef_polyhedron_2<T>  operator!() const
{ return complement(); }
Nef_polyhedron_2<T>& operator*=(const Nef_polyhedron_2<T>& N1)
{ this = intersection(N1); return *this; }
Nef_polyhedron_2<T>& operator+=(const Nef_polyhedron_2<T>& N1)
{ this = join(N1); return *this; }
Nef_polyhedron_2<T>& operator-=(const Nef_polyhedron_2<T>& N1)
{ this = difference(N1); return *this; }
Nef_polyhedron_2<T>& operator^=(const Nef_polyhedron_2<T>& N1)
{ this = symmetric_difference(N1); return *this; }
```

1.6.5 Binary Comparison Operations

All set comparison operations are reduced to binary operations followed by an empty-set test. For two Nef polyhedra P_1, P_2 it holds

$$\begin{aligned} P_1 = P_2 &\Leftrightarrow \mathit{symmetric\_difference}(P_1,P_2) = \emptyset \\ P_1 \subseteq P_2 &\Leftrightarrow \mathit{difference}(P_1,P_2) = \emptyset \\ P_1 \subsetneq P_2 &\Leftrightarrow \mathit{difference}(P_1,P_2) = \emptyset \ \wedge \ \mathit{difference}(P_2,P_1) \neq \emptyset \end{aligned}$$

In our specification $\subseteq$ is *operator* $\leq$, and $\subsetneq$ is *operator* $<$. The other operations are symmetric.

⟨*nef interface operations*⟩+≡

```
bool operator==(const Nef_polyhedron_2<T>& N1) const
{ return symmetric_difference(N1).is_empty(); }
bool operator!=(const Nef_polyhedron_2<T>& N1) const
{ return !operator==(N1); }
bool operator<=(const Nef_polyhedron_2<T>& N1) const
{ return difference(N1).is_empty(); }
bool operator<(const Nef_polyhedron_2<T>& N1) const
{ return difference(N1).is_empty() && !N1.difference(*this).is_empty(); }
```

⟨*nef interface operations*⟩+≡

```
bool operator>=(const Nef_polyhedron_2<T>& N1) const
{ return N1.difference(*this).is_empty(); }

bool operator>(const Nef_polyhedron_2<T>& N1) const
{ return N1.difference(*this).is_empty() && !difference(N1).is_empty(); }
```

1.6.6 Point location and Ray shooting

Let P be the plane map underlying our Nef polyhedron stored in the $*this$ object. The result of a point location query with an affine point p is the object of P whose embedding contains p. Ray shooting queries come in two flavors. One variant starts the ray shot in a point p and determines the closest object of P in direction d that is in the set (determined by the selection mark). The other variant determines the closest 1-skeleton object in direction d. The point location and ray shooting functionality is taken from the two point location classes *PM_point_locator*<> and *PM_naive_point_locator*<>. All operations can choose between the two approaches by a mode flag m. The default is location as implemented by *PM_point_locator*<>. That class uses a further subdivision of P by a locally minimized weight constrained triangulations (LMWT) to allow so-called segment walks. The LMWT is calculated on demand, when the first point location or ray shooting operation is called. The naive point location method is based on a global examination of all objects of P to find the one that contains p.

⟨*nef interface operations*⟩+≡

```
typedef Const_decorator Topological_explorer;
typedef CGAL::PM_explorer<Const_decorator,T> Explorer;
typedef typename Locator::Object_handle Object_handle;
enum Location_mode { DEFAULT, NAIVE, LMWT };

void init_locator() const { ptr->init_locator(); }
const Locator& locator() const
{ assert(ptr->pl_); return *(ptr->pl_); }
```

The type *Object_handle* is a polymorphic handle that can reference vertices, edges, and faces. Conversion is done by an assign operation similarly to the polymorphic CGAL type *Object*.

⟨*nef interface operations*⟩+≡

```
bool contains(Object_handle h) const
{ Slocator PL(pm()); return PL.mark(h); }

bool contained_in_boundary(Object_handle h) const
{ Vertex_const_handle v;
  Halfedge_const_handle e;
  return  ( CGAL::assign(v,h) || CGAL::assign(e,h) );
}
```

The chosen point location method depends on m. In the non-naive case the locator object is initialized by a call to *init_locator*( ). The corresponding locator object is stored in the representation object for further usage in iterated queries and can be accessed by the *locator*( ) method. The naive locate operation requires a segment as input that intersects the 1-skeleton of the plane map.

⟨*nef interface operations*⟩+≡
```
Object_handle locate(const Point& p, Location_mode m = DEFAULT) const
{
  if (m == DEFAULT || m == LMWT) {
    ptr->init_locator();
    Extended_point ep = EK.construct_point(p);
    return locator().locate(ep);
  } else if (m == NAIVE) {
    Slocator PL(pm(),EK);
    Extended_segment s(EK.construct_point(p),
                       PL.point(PL.vertices_begin()));
    return PL.locate(s);
  }
  CGAL_assertion_msg(0,"location mode not implemented.");
  return Object_handle();
}
```

The ray shooting operation determines the closest object of P that is marked and hit by a ray shot from the point p in direction d. The search is delegated to the corresponding member of the locator object. The class *INSET* is the predicate class that stops the ray shot when a marked object is hit. See the manual page of the locator classes for its concept.

⟨*nef interface operations*⟩+≡
```
struct INSET {
  const Const_decorator& D;
  INSET(const Const_decorator& Di) : D(Di) {}
  bool operator()(Vertex_const_handle v) const { return D.mark(v); }
  bool operator()(Halfedge_const_handle e) const { return D.mark(e); }
  bool operator()(Face_const_handle f) const { return D.mark(f); }
};

Object_handle ray_shoot(const Point& p, const Direction& d,
                        Location_mode m = DEFAULT) const
{
  if (m == DEFAULT || m == LMWT) {
    ptr->init_locator();
    Extended_point ep = EK.construct_point(p),
                   eq = EK.construct_point(p,d);
    return locator().ray_shoot(EK.construct_segment(ep,eq),
                               INSET(locator()));
  } else if (m == NAIVE) {
    Slocator PL(pm(),EK);
    Extended_point ep = EK.construct_point(p),
                   eq = EK.construct_point(p,d);
    return PL.ray_shoot(EK.construct_segment(ep,eq),INSET(PL));
  }
```

```
    CGAL_assertion_msg(0,"location mode not implemented.");
    return Object_handle();
  }
```

A similar implementation is used for *ray_shoot_to_boundary*. Note that we only use a different predicate *INSKEL*.

⟨*nef interface operations*⟩+≡

```
  struct INSKEL {
    bool operator()(Vertex_const_handle) const { return true; }
    bool operator()(Halfedge_const_handle) const { return true; }
    bool operator()(Face_const_handle) const { return false; }
  };

  Object_handle ray_shoot_to_boundary(const Point& p, const Direction& d,
                Location_mode m = DEFAULT) const
  {
    if (m == DEFAULT || m == LMWT) {
      ptr->init_locator();
      Extended_point ep = EK.construct_point(p),
                     eq = EK.construct_point(p,d);
      return locator().ray_shoot(EK.construct_segment(ep,eq),INSKEL());
    } else if (m == NAIVE) {
      Slocator PL(pm(),EK);
      Extended_point ep = EK.construct_point(p),
                     eq = EK.construct_point(p,d);
      return PL.ray_shoot(EK.construct_segment(ep,eq),INSKEL());
    }
    CGAL_assertion_msg(0,"location mode not implemented.");
    return Object_handle();
  }
```

To examine the plane map underlying the Nef polyhedron the user can obtain a decorator object that has read-only access to *pm*( ). Thus, modifications can only take place via the interface operations of *Nef_polydron_2*.

⟨*nef interface operations*⟩+≡

```
  Explorer explorer() const { return Explorer(pm(),EK); }
```

1.6.7 The file wrapper

⟨*Nef_polyhedron_2.h*⟩≡

```
  ⟨CGAL Header1⟩
  // file          : include/CGAL/Nef_polyhedron_2.h
  ⟨CGAL Header2⟩
  #ifndef CGAL_NEF_POLYHEDRON_2_H
  #define CGAL_NEF_POLYHEDRON_2_H

  #if defined(_MSC_VER) || defined(__BORLANDC__)
```

```
#define CGAL_SIMPLE_HDS
#endif
#include <CGAL/basic.h>
#include <CGAL/Handle_for.h>
#include <CGAL/Random.h>
#ifndef CGAL_SIMPLE_HDS
#include <CGAL/Nef_2/HDS_items.h>
#include <CGAL/HalfedgeDS_default.h>
#else
#include <CGAL/Nef_2/HalfedgeDS_default_MSC.h>
#endif
#include <CGAL/Nef_2/PM_explorer.h>
#include <CGAL/Nef_2/PM_decorator.h>
#include <CGAL/Nef_2/PM_io_parser.h>
#include <CGAL/Nef_2/PM_overlayer.h>
//#include <CGAL/Nef_2/PM_transformer.h>
#include <CGAL/Nef_2/PM_point_locator.h>
#include <vector>
#include <list>
#undef _DEBUG
#define _DEBUG 11
#include <CGAL/Nef_2/debug.h>
CGAL_BEGIN_NAMESPACE
⟨nef polyhedron definition⟩
⟨nef polyhedron input and output⟩
CGAL_END_NAMESPACE
#undef CGAL_SIMPLE_HDS
#endif //CGAL_NEF_POLYHEDRON_2_H
```

1.6.8 Visualization

At last we provide a drawing routine for Nef polyhedra in a LEDA window. We want to draw faces, edges, vertices in this order, where faces are maximally connected point sets bounded by one outer face cycle and maybe by several inner hole cycles. We draw objects which are in our point set black and objects which are not in our pointset by a light color. Note that we face the following problem. Our window represents a rectangular view to our square frame, which is large enough to make the topology on this boundary constant. Imagine making our frame big enough and then shrinking it slowly down to zero. For each ray with slope not equal to one and not containing the origin there is a value R when the ray tip on the frame leaves its correct frame segment. If we want to prevent topological difficulties when drawing the polyhedron we have to keep our frame radius above the minimum R_m. Thus visualization determines this R_m and sets the internal evaluation parameter to this value. Then all points on the frame and segments containing such points have fixed coordinates and can be drawn. For the concrete technical details of drawing the objects see the class *PM_visualizor*<>.

⟨*window stream output*⟩≡
```
static long frame_default = 100;
static bool show_triangulation = false;

template <typename T>
CGAL::Window_stream& operator<<(CGAL::Window_stream& ws,
const Nef_polyhedron_2<T>& P)
{
  typedef Nef_polyhedron_2<T> Polyhedron;
  typedef typename T::RT RT;
  typedef typename T::Standard_RT Standard_RT;
  typedef typename Polyhedron::Topological_explorer TExplorer;
  typedef typename Polyhedron::Point            Point;
  typedef typename Polyhedron::Line             Line;
  typedef CGAL::PM_BooleColor<TExplorer> BooleColor;
  typedef CGAL::PM_visualizor<TExplorer,T,BooleColor> Visualizor;

  TExplorer D = P.explorer();
  const T& E = Nef_polyhedron_2<T>::EK;

  Standard_RT frame_radius = frame_default;
  E.determine_frame_radius(D.points_begin(),D.points_end(),frame_radius);
  RT::set_R(frame_radius);
  Visualizor PMV(ws,D); PMV.draw_map();
  ⟨draw the refining constrained triangulation⟩
  return ws;
}
```

Drawing of the constrained triangulation is done depending on the static variable *show_triangulation*. Of course such animation requires the necessary preprocessing with the locator object.

⟨*draw the refining constrained triangulation*⟩≡
```
if (show_triangulation) {
  P.init_locator();
  Visualizor V(ws,P.locator().triangulation());
  V.draw_skeleton(CGAL::BLUE);
}
```

The header just wraps the above operations.

⟨*Nef_polyhedron_2_Window_stream.h*⟩≡
```
⟨CGAL Header1⟩
// file          : include/CGAL/IO/Nef_polyhedron_2_Window_stream.h
⟨CGAL Header2⟩
#ifndef NEF_POLYHEDRON_2_WINDOW_STREAM_H
#define NEF_POLYHEDRON_2_WINDOW_STREAM_H

#include <CGAL/Nef_polyhedron_2.h>
#include <CGAL/Nef_2/PM_visualizor.h>

CGAL_BEGIN_NAMESPACE

⟨window stream output⟩

CGAL_END_NAMESPACE

#endif // NEF_POLYHEDRON_2_WINDOW_STREAM_H
```

1.6.9 Input and Output

Standard input and output is done by the plane map I/O class *PM_io_parser*. We tag the output with an idendifier to be able to check correct coordinate representation when we read it as input.

⟨*nef polyhedron input and output*⟩≡
```
template <typename T>
std::ostream& operator<<
 (std::ostream& os, const Nef_polyhedron_2<T>& NP)
{
  os << "Nef_polyhedron_2<" << NP.EK.output_identifier() << ">\n";
  typedef typename Nef_polyhedron_2<T>::Decorator Decorator;
  CGAL::PM_io_parser<Decorator> O(os, NP.pm()); O.print();
  return os;
}
template <typename T>
std::istream& operator>>
  (std::istream& is, Nef_polyhedron_2<T>& NP)
{
  typedef typename Nef_polyhedron_2<T>::Decorator Decorator;
  CGAL::PM_io_parser<Decorator> I(is, NP.pm());
  if (I.check_sep("Nef_polyhedron_2<") &&
      I.check_sep(NP.EK.output_identifier()) &&
      I.check_sep(">")) I.read();
  else {
    std::cerr << "Nef_polyhedron_2 input corrupted." << std::endl;
    NP = Nef_polyhedron_2<T>();
  }
  typename Nef_polyhedron_2<T>::Topological_explorer D(NP.explorer());
  D.check_integrity_and_topological_planarity();
  return is;
}
```

1.6.10 Hiding extended geometry

The plane map explorer provides an interface that masks the properties of our extended kernel by reintroducing purely affine objects. We want to provide a simple interface for users who are not interested in the detailed features of extended objects. The methods of the class *PM_explorer*<> allow queries to the category of vertices and edges such as “Is a vertex embedded in the affine space or on the frame box?” or “Is an edge part of the affine structure or part of the frame box”.

⟨*PM_explorer.h*⟩≡
```
⟨CGAL Header1⟩
// file          : include/CGAL/Nef_2/PM_explorer.h
⟨CGAL Header2⟩
#ifndef CGAL_PM_EXPLORER_H
#define CGAL_PM_EXPLORER_H
#include <CGAL/basic.h>
#include <CGAL/Nef_2/PM_const_decorator.h>
```

```
CGAL_BEGIN_NAMESPACE
⟨PM explorer⟩
CGAL_END_NAMESPACE
#endif // CGAL_PM_EXPLORER_H
```

⟨*PM explorer*⟩≡

```
template <typename PMCDEC, typename GEOM>
class PM_explorer : public PMCDEC
{ typedef PMCDEC Base;
  typedef PM_explorer<PMCDEC,GEOM> Self;
  const GEOM* pK;
public:
typedef PMCDEC Topological_explorer;
typedef typename PMCDEC::Plane_map Plane_map;
typedef GEOM Geometry;
typedef typename GEOM::Standard_point_2 Point;
typedef typename GEOM::Standard_ray_2   Ray;
⟨local typedefs of explorer handles⟩
PM_explorer(const Self& E) : Base(E), pK(E.pK) {}
Self& operator=(const Self& E)
{ Base::operator=(E); pK=E.pK; return *this; }
PM_explorer(const Plane_map& P, const Geometry& k = Geometry()) :
  Base(P), pK(&k) {}
bool is_standard(Vertex_const_handle v) const
{ return pK->is_standard(Base::point(v)); }
Point point(Vertex_const_handle v) const
{ return pK->standard_point(Base::point(v)); }
Ray ray(Vertex_const_handle v) const
{ return pK->standard_ray(Base::point(v)); }
bool is_frame_edge(Halfedge_const_handle e) const
{ return ( face(e) == faces_begin() ||
           face(twin(e)) == faces_begin() ); }
}; // PM_explorer<PMCDEC,GEOM>
```

⟨*local typedefs of explorer handles*⟩≡

```
typedef typename Base::Vertex_const_handle Vertex_const_handle;
typedef typename Base::Halfedge_const_handle Halfedge_const_handle;
typedef typename Base::Face_const_handle Face_const_handle;
typedef typename Base::Vertex_const_iterator Vertex_const_iterator;
typedef typename Base::Halfedge_const_iterator Halfedge_const_iterator;
typedef typename Base::Face_const_iterator Face_const_iterator;
typedef typename Base::Halfedge_around_face_const_circulator
                       Halfedge_around_face_const_circulator;
typedef typename Base::Halfedge_around_vertex_const_circulator
                       Halfedge_around_vertex_const_circulator;
typedef typename Base::Isolated_vertex_const_iterator
```

```
                       Isolated_vertex_const_iterator;
typedef typename Base::Hole_const_iterator
                       Hole_const_iterator;
```

1.7 A Demo Program

The basic idea of our demo is simple. Store some basic polyhedra in a history list and let the user interactively extend this list by the binary and unary operations. The point location and ray shooting capabilities of the structure can be triggered by mouse clicks into the drawing window.

⟨*Nef_polyhedron_2-demo.C*⟩≡

```
#include <CGAL/basic.h>
#ifdef CGAL_USE_LEDA
#include "xpms/nef.xpm"
#include <LEDA/pixmaps/button32/eye.xpm>
#include <LEDA/pixmaps/button32/draw.xpm>
#include <CGAL/leda_integer.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Filtered_extended_homogeneous.h>
#include <CGAL/IO/Filtered_extended_homogeneous_Window_stream.h>
#include <CGAL/Nef_polyhedron_2.h>
#include <CGAL/IO/Nef_polyhedron_2_Window_stream.h>

template <>
struct ring_or_field<leda_integer> {
  typedef ring_with_gcd kind;
  typedef leda_integer RT;
  static RT gcd(const RT& r1, const RT& r2)
  { return ::gcd(r1,r2); }
};

#define FILTERED_KERNEL
#ifndef FILTERED_KERNEL
typedef CGAL::Extended_homogeneous<leda_integer> EKernel;
#else
typedef CGAL::Filtered_extended_homogeneous<leda_integer> EKernel;
#endif

#if defined(_MSC_VER) || defined(__BORLANDC__)
#define WIN32CONFIG
#endif
```

⟨*getting types in global scope*⟩
⟨*a small interactive polyhedron editor*⟩

```
int main(int argc, char* argv[])
{
  /* Enable debugging by introducing prime into the product:
    RP 3 EP 5/59 PM 7 NefTOP 11
    Overlayer 13 PointLoc 17 ConstrTriang 19
    SegSweep 23
  */
  SETDTHREAD(113);
  CGAL::set_pretty_mode ( std::cerr );
```

```
  std::cerr << "using " << CGAL::pointlocationversion << std::endl;
  std::cerr << "using " << CGAL::sweepversion << std::endl;
  ⟨initializing the drawing window⟩
  ⟨initializing the history⟩
  ⟨initializing the two panels⟩
  ⟨load bitmaps and add buttons to panels⟩
  leda_drawing_mode dm;
  Point p_down(0,0);
  main_panel.display();
  int x0,y0,x1,y1;
  W.frame_box(x0,y0,x1,y1);
  W.display_help_text("help/nef-demo");
  Object_handle h;
  for(;;) {
    double x, y;
    int val;
    switch( W.read_event(val, x, y) ) {
    case button_press_event:
      p_down = Point(x,y);
      if (val == MOUSE_BUTTON(1)) {
        std::cerr << "locating " << p_down << std::endl;
        dm = W.set_mode(leda_xor_mode);
        W<<CGAL::GREEN<<p_down; W.set_mode(dm);
        h = N_display.locate(p_down);
        draw(h);
      }
      if (val == MOUSE_BUTTON(2)) {
        std::cerr << "shooting down from " << p_down << std::endl;
        dm = W.set_mode(leda_xor_mode);
        W<<CGAL::GREEN<<p_down; W.set_mode(dm);
        //h = N_display.ray_shoot(p_down,Direction(0,-1),Nef_polyhedron::NAIVE);
        h = N_display.ray_shoot(p_down,Direction(0,-1));
        draw(h);
      }
      if (val == MOUSE_BUTTON(3)) {
        CGAL::show_triangulation = !CGAL::show_triangulation;
        win_redraw_handler(&W);
      }
      break;
    case button_release_event:
      if (val == MOUSE_BUTTON(1))
#ifndef WIN32CONFIG
      { dm = W.set_mode(leda_xor_mode);
        W<<CGAL::GREEN<<p_down; W.set_mode(dm); draw(h); }
#else
      { win_redraw_handler(&W); }
#endif
      if (val == MOUSE_BUTTON(2))
#ifndef WIN32CONFIG
      { dm = W.set_mode(leda_xor_mode);
        W<<CGAL::GREEN<<p_down; W.set_mode(dm); draw(h); }
#else
```

```
      { win_redraw_handler(&W); }
#endif
      break;
    case key_press_event:
      if (val ==  KEY_UP) { // ZOOM IN
        CGAL::frame_default*=2;
        Nef_polyhedron::Extended_kernel::RT::set_R(CGAL::frame_default);
        int r = CGAL::frame_default+10;
        W.init(-r,r,-r);
        win_redraw_handler(&W);
      }
      if (val ==  KEY_DOWN) { // ZOOM OUT
        CGAL::frame_default/=2;
        Nef_polyhedron::Extended_kernel::RT::set_R(CGAL::frame_default);
        int r = CGAL::frame_default+10;
        W.init(-r,r,-r);
        win_redraw_handler(&W);
      }
    default:
      break;
    }
  }
#if !defined(__KCC) && !defined(__BORLANDC__)
  return 0; // never reached
#endif
}

#else // CGAL_USE_LEDA

int main() { return 0; }

#endif // CGAL_USE_LEDA
```

⟨*getting types in global scope*⟩≡

```
typedef  CGAL::Nef_polyhedron_2<EKernel> Nef_polyhedron;
typedef  Nef_polyhedron::Point     Point;
typedef  Nef_polyhedron::Line      Line;
typedef  Nef_polyhedron::Direction Direction;
typedef  Nef_polyhedron::Object_handle Object_handle;
typedef  Nef_polyhedron::Explorer Explorer;
typedef  Nef_polyhedron::Topological_explorer TExplorer;
typedef  Explorer::Vertex_const_handle Vertex_const_handle;
typedef  Explorer::Halfedge_const_handle Halfedge_const_handle;
typedef  Explorer::Face_const_handle Face_const_handle;
```

We create one drawing window W, and two panels controlling the operations *main_panel*, *op_panel*. We store a history of polyhedra: *ML* stores the names displayed, *MH* maps the names to the existing objects. *N_display* stores the current object drawn in W.

⟨*a small interactive polyhedron editor*⟩≡

```
#include <LEDA/panel.h>
#include <LEDA/list.h>
#include <LEDA/d_array.h>
#include <LEDA/file_panel.h>
```

```
#include <LEDA/file.h>
#include <LEDA/stream.h>
static leda_panel      main_panel;
static leda_panel      op_panel;
static panel_item      nef_menu_item;
static panel_item      op_item;
static leda_string     nef1;
static leda_string     nef2;
static menu            create_menu;
static int num;
#ifndef FILTERED_KERNEL
static leda_string  dname = "./homogeneous_data";
#else
static leda_string  dname = "./filtered_homogeneous_data";
#endif
static leda_string  fname = "none";
static leda_string  filter = "*.nef";
CGAL::Window_stream* pW;
Nef_polyhedron* pN;
leda_list<leda_string>* pML;
leda_d_array<leda_string,Nef_polyhedron>* pMH;
static leda_string stripped(leda_string s)
{ int i = s.pos(" = "); return s.head(i); }
```

⟨*initializing the drawing window*⟩≡

```
CGAL::Window_stream W(600,600); pW = &W;
Nef_polyhedron N_display; pN = &N_display;
leda_list<leda_string> ML; pML = &ML;
leda_d_array<leda_string,Nef_polyhedron> MH; pMH = &MH;
```

Redrawing the window is done via *win_redraw_handler*. Our exit handler is *win_del_handler*. Opening a panel is done by *open_panel*.

⟨*a small interactive polyhedron editor*⟩+≡

```
void win_redraw_handler(leda_window*)
{ pW->clear(); (*pW) << (*pN); }
void win_del_handler(leda_window*)
{ leda_panel P("acknowledge");
  P.text_item("");
  P.text_item("\\bf\\blue Do you really want to quit~?");
  P.fbutton("no",0);
  P.button("yes",1);
  if (P.open(main_panel) == 1) exit(0);
}
static int open_panel(leda_panel& p)
{ p.display(main_panel,0,0);
  int res = p.read_mouse();
  p.close();
  return res;
}
```

store_new stores a new polyhedron N with name t at the beginning of our history. *update_history* triggers the visual history update.

⟨*a small interactive polyhedron editor*⟩+≡
```
static void store_new(const Nef_polyhedron& N, leda_string t)
{ leda_string k = leda_string("N%i",++num) ;
  (*pMH)[k] = (*pN) = N;
  pML->push_front(k+" = "+t);
  win_redraw_handler(pW);
}
static void update_history()
{
  nef1=pML->head();
  nef2=pML->head();
  main_panel.add_menu(nef_menu_item,(*pML));
  op_panel.add_menu(op_item,(*pML));
}
```

create is linked to the creation menu (a bitmap button). The enums control the different creation actions which are mapped to some input operation on W followed by a call to some constructor of *Nef_polyhedron_2*.

⟨*a small interactive polyhedron editor*⟩+≡
```
enum { EMPTY=31, FULL, HOPEN, HCLOSED, POPEN, PCLOSED };
enum { FILE_LOAD=111, FILE_SAVE };
void create(int i)
{
  if (pML->back()=="none") pML->pop_back();
  Line l; Point p;
  std::list<Point> Lp;
  leda_point pd;
  leda_list<leda_point> Lpd;
  string_ostream sos; CGAL::set_pretty_mode(sos);
  pW->clear();
  switch (i) {
    case HOPEN:
      pW->message("Insert Half-Space by Line");
      (*pW) >> l; sos << '(' << l << ')' << '\0';
      if ( l.is_degenerate() ) {
        (*pW).acknowledge("Please enter non-degenerate line.");
        win_redraw_handler(pW);
      } else
        store_new(Nef_polyhedron(l,Nef_polyhedron::EXCLUDED),sos.str());
      break;
    case HCLOSED:
      pW->message("Insert Half-Space by Line");
      (*pW) >> l; sos << '[' << l << ']' << '\0';
      if ( l.is_degenerate() ) {
        (*pW).acknowledge("Please enter non-degenerate line.");
        win_redraw_handler(pW);
      } else
```

```
        store_new(Nef_polyhedron(l,Nef_polyhedron::INCLUDED),sos.str());
      break;
    case POPEN:
      pW->message("Insert Polygon by Point Sequence");
      Lpd = pW->read_polygon();
      forall(pd,Lpd) Lp.push_back(Point(pd.xcoord(),pd.ycoord()));
      sos << '[' << Lp.size() << "-gon"<< ']' << '\0';
      store_new(Nef_polyhedron(Lp.begin(),Lp.end(),
                               Nef_polyhedron::EXCLUDED),sos.str());
      break;
    case PCLOSED:
      pW->message("Insert Polygon by Point Sequence");
      Lpd = pW->read_polygon();
      forall(pd,Lpd) Lp.push_back(Point(pd.xcoord(),pd.ycoord()));
      sos << '[' << Lp.size() << "-gon"<< ']' << '\0';
      store_new(Nef_polyhedron(Lp.begin(),Lp.end(),
                               Nef_polyhedron::INCLUDED),sos.str());
      break;
    default:
      std::cout << "created nothing\n";
  }
  sos.freeze(0);
  update_history();
  main_panel.redraw_panel();
}
```

File input and output is done via the operation *file_handler*.

⟨*a small interactive polyhedron editor*⟩+≡

```
static void read_file(leda_string fn)
{
  std::ifstream in(fn);
  Nef_polyhedron N; in >> N;
  fn.replace_all(".nef","");
  store_new(N,fn);update_history();
}

static bool confirm_overwrite(const char* fname)
{
#if defined(_MSC_VER) || defined(__BORLANDC__)
  return true;
#else
  leda_panel P;
  P.buttons_per_line(2);
  P.text_item("");
  P.text_item(leda_string("\\bf\\blue File\\black %s\\blue exists.",fname));
  P.button("overwrite",0);
  P.button("cancel",1);
  return (P.open() == 0);
#endif
}

static void write_file(leda_string fname)
{
```

```
  if (is_file(fname) && !confirm_overwrite(fname)) return;
  // win_ptr->set_status_string(" Writing " + fname);
  leda_string nef = stripped(nef1);
  if (pMH->defined(nef)) {
    std::ofstream out(fname);
    out << (*pMH)[nef];
  } else error_handler(1,"Nef polyhedron "+nef+" not defined.");
}
static void file_handler(int what)
{
  file_panel FP(fname,dname);
  switch (what) {
  case FILE_LOAD: FP.set_load_handler(read_file);
                  break;
  case FILE_SAVE: FP.set_save_handler(write_file);
                  break;
  }
  if (filter != "") FP.set_pattern(filter,filter);
  FP.open();
}
```

We have three more operations. *view* just draws the currently chosen polyhedron of our history. *binop* allows a user to combine the currently chosen polyhedron with one which is chosen in the second panel *op_panel*. The binary operation used is defined by the button which is pressed in *main_panel*. *unop* just triggers the unary operation chosen by the button selection on the currently selected polyhedron of the history.

⟨*a small interactive polyhedron editor*⟩+≡

```
void view(int i)
{
  leda_string nef = stripped(nef1);
  if (pMH->defined(nef)) {
    (*pN) = (*pMH)[nef]; win_redraw_handler(pW);
  } else error_handler(1,"Nef polyhedron "+nef+" not defined.");
}
void binop(int i)
{
  leda_string op1;
  switch (i) {
   case 30: op1="intersection"; break;
   case 31: op1="union";        break;
   case 32: op1="difference";   break;
   case 33: op1="symmdiff"; break;
   default: break;
  }
  op_panel.set_button_label(111,op1);
  open_panel(op_panel);
  op_panel.flush();
  leda_string arg1 = stripped(nef1), arg2 = stripped(nef2);
  Nef_polyhedron N1 = (*pMH)[arg1];
  Nef_polyhedron N2 = (*pMH)[arg2];
```

```
  std::ofstream log("nef-demo.log");
  if ( !log ) CGAL_assertion_msg(0,"no output log nef-demo.log");
  log << 2 << std::endl << N1 << N2 << std::endl;
  log.close();
  switch (i) {
   case 30: *pN = N1*N2; break;
   case 31: *pN = N1+N2; break;
   case 32: *pN = N1-N2; break;
   case 33: *pN = N1^N2; break;
   default: return;
  }
  leda_string descr = op1+"("+arg1+","+arg2+")";
  store_new(*pN,descr);update_history();
  win_redraw_handler(pW);
}
void unop(int i)
{
  leda_string op, arg = stripped(nef1);
  Nef_polyhedron N = (*pMH)[arg];
  std::ofstream log("nef-demo.log");
  if ( !log ) CGAL_assertion_msg(0,"no output log nef-demo.log");
  log << 1 << std::endl << N << std::endl;
  log.close();
  switch (i) {
   case 40: op="interior";   *pN = N.interior(); break;
   case 41: op="complement"; *pN = N.complement(); break;
   case 42: op="closure";    *pN = N.closure(); break;
   case 43: op="boundary";   *pN = N.boundary(); break;
   default: return;
  }
  leda_string descr = op+"("+arg+")";
  store_new(*pN,descr);update_history();
  win_redraw_handler(pW);
}
```

The *draw* operation just marks visually the object of the plane map referenced by *h*.

⟨*a small interactive polyhedron editor*⟩+≡

```
void draw(Object_handle h)
{ CGAL::PM_visualizor<TExplorer,EKernel>
    PMV(*pW,pN->explorer(),pN->EK,
        CGAL::PM_DefColor<TExplorer>(CGAL::RED,CGAL::RED,6,6) );
  leda_drawing_mode prev = pW->set_mode(leda_xor_mode);
  Vertex_const_handle vh; Halfedge_const_handle eh; Face_const_handle fh;
  if ( CGAL::assign(vh,h) ) PMV.draw(vh);
  if ( CGAL::assign(eh,h) ) PMV.draw(eh);
  if ( CGAL::assign(fh,h) ) PMV.draw(fh);
  pW->set_mode(prev);
}
```

We initialize the window to the default of the size of our frame and center it at the origin.

⟨*initializing the drawing window*⟩+≡
```
W.init(-CGAL::frame_default,CGAL::frame_default,-CGAL::frame_default);
W.set_show_coordinates(true);
W.set_grid_mode(5);
W.set_node_width(3);
W.set_redraw(&win_redraw_handler);
W.display(0,0);
```

We initialize the history with several simple nef polyhedra like an empty one, one representing the plane, one left of the y-axis, one above a diagonal and one defined by a simple quadratic polygon.

⟨*initializing the history*⟩≡
```
std::list<Point> Lp;
const int r=70;
Lp.push_back(Point(-r,-r));
Lp.push_back(Point(r,-r));
Lp.push_back(Point(r,r));
Lp.push_back(Point(-r,r));
store_new(Nef_polyhedron(),"empty");
store_new(Nef_polyhedron(Nef_polyhedron::COMPLETE),"plane");
store_new(Nef_polyhedron(Line(Point(0,0),Point(0,1)),
          Nef_polyhedron::INCLUDED),"neg y-plane (closed)");
store_new(Nef_polyhedron(Line(Point(-2,-1),Point(2,1)),
          Nef_polyhedron::EXCLUDED),"above diagonal (open)");
store_new(Nef_polyhedron(Lp.begin(),Lp.end(),
          Nef_polyhedron::INCLUDED),"square (closed)");
if ( argc == 2 ) {
  std::ifstream log( argv[1] );
  if ( !log )
    CGAL_assertion_msg(0,leda_string("no input log ")+argv[1]);
  int n;
  log >> n;
  for (int i=0; i<n; ++i) {
    Nef_polyhedron Ni;
    log >> Ni;
    store_new(Ni,leda_string(argv[1])+leda_string("%d",i+1));
  }
}
```

The head of our history list is displayed in the panels.

⟨*initializing the two panels*⟩≡
```
nef1=ML.head();nef2=ML.head();
nef_menu_item = main_panel.string_item("Polyhedra",nef1,ML);
main_panel.set_window_delete_handler(win_del_handler);
main_panel.buttons_per_line(10);
main_panel.set_item_width(300);
main_panel.set_frame_label("Operations on Nef Polyhedra");
main_panel.set_icon_label("Nef Polyhedra");
op_item = op_panel.string_item("Polyhedra",nef2,ML);
op_panel.fbutton("                  ",111);
```

```
op_panel.set_frame_label("Choose second argument");
op_panel.set_item_width(300);
```

⟨*load bitmaps and add buttons to panels*⟩≡

```
char* p_inter  = main_panel.create_pixrect(intersection_xpm);
char* p_union  = main_panel.create_pixrect(union_xpm);
char* p_diff   = main_panel.create_pixrect(difference_xpm);
char* p_exor   = main_panel.create_pixrect(exor_xpm);
char* p_int    = main_panel.create_pixrect(interior_xpm);
char* p_compl  = main_panel.create_pixrect(complement_xpm);
char* p_clos   = main_panel.create_pixrect(closure_xpm);
char* p_bound  = main_panel.create_pixrect(boundary_xpm);
char* p_eye    = main_panel.create_pixrect(eye_xpm);
char* p_draw   = main_panel.create_pixrect(draw_xpm);
main_panel.button(p_eye,p_eye,"show", 10, view);
main_panel.button(p_draw,p_draw,"new polyhedron", 11, create_menu);
main_panel.button(p_inter,p_inter,"intersection", 30, binop);
main_panel.button(p_union,p_union,"union", 31, binop);
main_panel.button(p_diff,p_diff,"difference", 32, binop);
main_panel.button(p_exor,p_exor,"symmetric difference", 33, binop);
main_panel.button(p_int,p_int,"interior", 40,unop);
main_panel.button(p_compl,p_compl,"complement", 41,unop);
main_panel.button(p_clos,p_clos,"closure", 42,unop);
main_panel.button(p_bound,p_bound,"boundary", 43,unop);
create_menu.button("half-plane (open)", HOPEN, create);
create_menu.button("half-plane (closed)", HCLOSED, create);
create_menu.button("polygon (open)", POPEN, create);
create_menu.button("polygon (closed)", PCLOSED, create);
create_menu.button("load from disk",FILE_LOAD, file_handler);
create_menu.button("save to disk",FILE_SAVE, file_handler);
```

1.8 A Test program

⟨*Nef_polyhedron_2-test.C*⟩≡

```
#include <CGAL/basic.h>
#include <CGAL/test_macros.h>
#include <CGAL/Nef_2/redefine_MSC.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Filtered_extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_2.h>
#ifdef CGAL_USE_LEDA
#include <CGAL/leda_integer.h>
typedef leda_integer Integer;
template <>
struct ring_or_field<leda_integer> {
  typedef ring_with_gcd kind;
  typedef leda_integer RT;
  static RT gcd(const RT& r1, const RT& r2)
  { return ::gcd(r1,r2); }
```

```
};
#else
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz Integer;
template <>
struct ring_or_field<CGAL::Gmpz> {
  typedef ring_with_gcd kind;
  typedef CGAL::Gmpz RT;
  static RT gcd(const RT& r1, const RT& r2)
  { return CGAL::gcd(r1,r2); }
};
#else
typedef long Integer;
#endif
#endif

int main()
{
  SETDTHREAD(41);
  CGAL::set_pretty_mode ( std::cerr );
  std::cerr << "using " << CGAL::pointlocationversion << std::endl;
  std::cerr << "using " << CGAL::sweepversion << std::endl;
  CGAL_TEST_START;
{
  ⟨simple extended⟩
  ⟨nef test suite⟩
}
{
  ⟨filtered extended⟩
  ⟨nef test suite⟩
  Nef_polyhedron::EK.print_statistics();
}
  CGAL_TEST_END;
}
```

⟨*simple extended*⟩≡

```
typedef  CGAL::Extended_homogeneous<Integer> EKernel;
typedef  CGAL::Nef_polyhedron_2<EKernel> Nef_polyhedron;
typedef  Nef_polyhedron::Point     Point;
typedef  Nef_polyhedron::Direction Direction;
typedef  Nef_polyhedron::Line      Line;
```

⟨*filtered extended*⟩≡

```
typedef  CGAL::Filtered_extended_homogeneous<Integer> EKernel;
typedef  CGAL::Nef_polyhedron_2<EKernel> Nef_polyhedron;
typedef  Nef_polyhedron::Point     Point;
typedef  Nef_polyhedron::Direction Direction;
typedef  Nef_polyhedron::Line      Line;
```

⟨*nef test suite*⟩≡

```
typedef  Nef_polyhedron::Object_handle Object_handle;
```

```
typedef  Nef_polyhedron::Explorer Explorer;
typedef  Explorer::Vertex_const_handle Vertex_const_handle;
typedef  Explorer::Halfedge_const_handle Halfedge_const_handle;
typedef  Explorer::Face_const_handle Face_const_handle;
typedef  Explorer::Vertex_const_iterator Vertex_const_iterator;
typedef  Explorer::Halfedge_const_iterator Halfedge_const_iterator;
typedef  Explorer::Face_const_iterator Face_const_iterator;
typedef  Explorer::Ray Ray;
Point p1(0,0), p2(0,1), p3(1,0), p4(-1,-1), p5(0,-1), p6(-1,0), p7(1,1);
Line l1(p2,p1); // neg y-axis
Line l2(p1,p3); // pos x-axis
Nef_polyhedron N1(l1), N2(l2, Nef_polyhedron::EXCLUDED),
               EMPTY(Nef_polyhedron::EMPTY),PLANE(Nef_polyhedron::COMPLETE);
CGAL_TEST((N1*N1) == N1);
CGAL_TEST((N1*!N1) == EMPTY);
CGAL_TEST((N1+!N1) == PLANE);
CGAL_TEST((N1^N2) == ((N1-N2)+(N2-N1)));
CGAL_TEST((!(N1*N2)) == (!N1+!N2));
Nef_polyhedron N3 = N1.intersection(N2);
/* N3 is the first quadrant including the positive y-axis
   but excluding the origin and the positive x-axis */
CGAL_TEST(N3 < N1 && N3 < N2);
CGAL_TEST(N3 <= N1 && N3 <= N2);
CGAL_TEST(N1 > N3 && N2 > N3);
CGAL_TEST(N1 >= N3 && N2 >= N3);
Explorer E = N3.explorer();
Vertex_const_iterator v = E.vertices_begin();
CGAL_TEST( !E.is_standard(v) && E.ray(v) == Ray(p1,p4) );
Halfedge_const_handle e = E.first_out_edge(v);
CGAL_TEST( E.is_frame_edge(e) );
++(++v); // third vertex
CGAL_TEST( E.is_standard(v) && E.point(v) == p1 );
Vertex_const_handle v1,v2;
Halfedge_const_handle e1,e2;
Face_const_handle f1,f2;
Object_handle h1,h2,h3;
h1 = N3.locate(p1);
h2 = N3.locate(p1,Nef_polyhedron::NAIVE);
CGAL_TEST( CGAL::assign(v1,h1) && CGAL::assign(v2,h2) && v1 == v2 );
CGAL_TEST( E.is_standard(v1) && E.point(v1) == p1 );
h1 = N3.locate(p2);
h2 = N3.locate(p2,Nef_polyhedron::NAIVE);
CGAL_TEST( CGAL::assign(e1,h1) && CGAL::assign(e2,h2) );
CGAL_TEST( (e1==e2 || e1==E.twin(e2)) && E.mark(e1) );
h1 = N3.locate(p4);
h2 = N3.locate(p4,Nef_polyhedron::NAIVE);
CGAL_TEST( CGAL::assign(f1,h1) && CGAL::assign(f2,h2) &&
            f1 == f2 && !E.mark(f1) );
// shooting along angular bisector:
h1 = N3.ray_shoot(p4,Direction(1,1));
h2 = N3.ray_shoot(p4,Direction(1,1),Nef_polyhedron::NAIVE);
```

```
CGAL_TEST( CGAL::assign(f1,h1) && CGAL::assign(f2,h2) &&
           f1 == f2 && E.mark(f1) );
// shooting along x-axis:
h1 = N3.ray_shoot(p6,Direction(1,0));
h2 = N3.ray_shoot(p6,Direction(1,0),Nef_polyhedron::NAIVE);
CGAL_TEST( h1 == NULL && h2 == NULL );
// shooting along y-axis:
h1 = N3.ray_shoot(p5,Direction(0,1));
h2 = N3.ray_shoot(p5,Direction(0,1),Nef_polyhedron::NAIVE);
e = e1;
CGAL_TEST( CGAL::assign(e1,h1) && CGAL::assign(e2,h2) &&
           (e1==e2||e1==E.twin(e2)) && E.mark(e1) );
h1 = N3.ray_shoot_to_boundary(p5,Direction(0,1));
h2 = N3.ray_shoot_to_boundary(p5,Direction(0,1),Nef_polyhedron::NAIVE);
CGAL_TEST( N3.contained_in_boundary(h1) && N3.contained_in_boundary(h2) );
CGAL_TEST( CGAL::assign(v1,h1) && CGAL::assign(v2,h2) && v1 == v2 );
h1 = N3.ray_shoot_to_boundary(p7,Direction(0,-1));
h2 = N3.ray_shoot_to_boundary(p7,Direction(0,-1),Nef_polyhedron::NAIVE);
CGAL_TEST( N3.contained_in_boundary(h1) && N3.contained_in_boundary(h2) );
CGAL_TEST( CGAL::assign(e1,h1) && CGAL::assign(e2,h2) &&
           (e1==e2 || e1==E.twin(e2)) );

std::list<Point> L;
L.push_back(p1);
N3 = Nef_polyhedron(L.begin(), L.end(), Nef_polyhedron::INCLUDED);
E = N3.explorer();
h1 = N3.locate(p1);
h2 = N3.locate(p2);
CGAL_TEST( CGAL::assign(v1,h1) && E.point(v1)==p1 && E.mark(v1) );
CGAL_TEST( CGAL::assign(f1,h2) && !E.mark(f1) );

L.push_back(p2);
N3 = Nef_polyhedron(L.begin(), L.end(), Nef_polyhedron::INCLUDED);
E = N3.explorer();
h1 = N3.locate(p1);
h2 = N3.locate(CGAL::midpoint(p1,p2));
h3 = N3.locate(p6);
CGAL_TEST( CGAL::assign(v1,h1) && E.point(v1)==p1 && E.mark(v1) );
CGAL_TEST( CGAL::assign(e1,h2) && E.mark(e1) );
CGAL_TEST( CGAL::assign(f1,h3) && !E.mark(f1) );

L.push_back(p3);
N3 = Nef_polyhedron(L.begin(), L.end(), Nef_polyhedron::INCLUDED);
E = N3.explorer();
h1 = N3.locate(p1);
h2 = N3.locate(CGAL::midpoint(p1,p2));
h3 = N3.locate(p6);
CGAL_TEST( CGAL::assign(v1,h1) && E.point(v1)==p1 && E.mark(v1) );
CGAL_TEST( CGAL::assign(e1,h2) && E.mark(e1) );
CGAL_TEST( CGAL::assign(f1,h3) && E.mark(f1) );
h3 = N3.locate(Point(1,1,3));
CGAL_TEST( CGAL::assign(f1,h3) && !E.mark(f1) );

CGAL_IO_TEST(N1,N2);
```

1.9 A Runtime Test

For an evaluation of Nef polyhedra we do the following. We recursively create a random structure of desired complexity. We start from n half-spaces in an array. We interpret the entries as the leaves of a balanced binary tree. Starting from the leaf level we combine two neighbored nodes in each level into a new polyhedron by a symmetric difference operation. The root of the tree thereby contains a structure of complexity $O(n^2)$ as the symmetric difference does not take away any structure. The result is an arrangement of the lines in the boundary of the half-spaces, where each vertex, edge, and face carries a random bit. We then evaluate the two binary operations *intersection* and *union* on the two such structures. To get an impression of how good the extended approach is we compare four different geometric treatments of the problem. We measure the time for the naive implementation with an explicit usage of a polynomial ring number type based on LEDA's multiprecision integer arithmetic (marked as *naive*). We do the same test on a extended kernel that uses dynamic double filter techniques (based on the CGAL double interval arithmetic *Interval_nt_advanced*). As our predicate expressions are of bounded arithmetic depth, we can program these expressions explicitly in unrolled code blocks. The corresponding expressions are instantiated for the interval data type and for the multiprecision integer number type. The filter stage determines the result of our predicates in many cases by pure double arithmetic as long as the controled error bound guarantees the correctness of the result. We call this the *filtered* approach.

Finally we compare the two instantiations of our Nef polyhedron data type with the simpler scenarios of generic polygons (LEDA *rat_gen_polygon*s). The type has a simpler geometric domain as it only considers regularized polygons. This takes away some complications in the topology of the result of binary operations. generic polygons are programmed around the concept of simple polygonal chains which store the boundary cycles of the faces of the planar subdivision. They also do not consider boundary issues (the boundary is part of the face incident to it). The simpler topology should allow faster binary operations. Thus we cannot expect to beat the data structure with respect to algorithmic processing. Of course Nef polyhedra cover additionally the unbounded nature of half-spaces.

To allow any testing we fix a static boundary large enough to make the topology on the boundary constant (the knowledge about the size is taken from the previously calculated nef polyhedron). Then we convert the geometry to the affine bounded scenario and use generic polygons. This gives us some kind of competitor that competes in a simpler domain and that profits from our knowledge. If we just use the binary operations of generic polygons recursively they loose the competition due to the accumulated mantissae in the multiprecision representation of the polygon vertex embedding. If we normalize the embedding to its minimal representation then the expected runtime hierarchy pops up again. In the following *rgp* refers to *rat_gen_polygon* and *rgpn* refers to *rat_gen_polygon* including normalization after each binary operations. The lines contain times concerning the binary overlay of structures of a certain complexity. The first column determines the size n of a set of half-spaces. Such a set implies a planar arrangement of $n(n-1)/2+2n+4$ vertices, $n(n+1)+2n+4$ edges and $n(n+1)/2+2$ faces (in the non-degenerate case including the objects created by the framing box).

The line marked $\wedge$ presents the complexity and times for the tree-recursive synthesis of the structure. Thus we measure $O(n)$ binary symmetric difference operations on operands of increasing complexity. The three column entries #V, #E, and #F present the actual number of nodes, uedges and faces of the resulting arrangement (deviation from the formulas above due to degeneries are possible). The lines marked with $\cap$ and $\cup$ present the results of one corresponding binary operation where the result has the complexity shown in the columns #V, #E, #F.

Note that the generic polygons are actually slower in the synthesis phase as the normalization takes place for the whole structure and is therefore more expensive than our approach. The -1 entries tell you that we didn't measure those times anymore as the non-normalized approach took just too long. The times are measured in seconds on a SUN Ultra-Enterprise-10000 with an 333 MHz UltraSPARC processors.

#lines	op	#V	#E	#F	naive	filtered	rgpn	rgp
10	$\wedge$	67	121	56	0.64	0.065	0.075	0.125
10	$\cap$	162	217	61	0.74	0.09	0.08	0.93
10	$\cup$	160	213	60	0.75	0.09	0.07	0.92
15	$\wedge$	139	259	122	1.13	0.125	0.14	0.33
15	$\cap$	365	495	142	1.76	0.2	0.18	5.73
15	$\cup$	348	458	134	1.72	0.19	0.16	6.04
20	$\wedge$	230	437	209	2.31	0.26	0.285	2.3
20	$\cap$	622	831	241	2.89	0.36	0.3	19.2
20	$\cup$	577	744	205	2.93	0.36	0.3	19.6
25	$\wedge$	353	678	326	3.19	0.36	0.44	-1
25	$\cap$	965	1281	365	4.9	0.61	0.52	-1
25	$\cup$	981	1314	370	4.78	0.61	0.51	-1
30	$\wedge$	499	964	467	4.01	0.48	0.575	-1
30	$\cap$	1375	1821	514	7	0.92	0.8	-1
30	$\cup$	1410	1894	540	7.55	0.9	0.76	-1
40	$\wedge$	862	1680	820	8.7	1.05	1.25	-1
40	$\cap$	2395	3171	900	12.8	1.67	1.43	-1
40	$\cup$	2509	3370	960	12.9	1.71	1.42	-1
50	$\wedge$	1326	2600	1275	12.1	1.54	1.83	-1
50	$\cap$	3828	5111	1470	21.1	2.88	2.41	-1
50	$\cup$	3809	5073	1455	20.7	2.85	2.35	-1
100	$\wedge$	5149	10195	5048	49.8	6.76	8.29	-1
100	$\cap$	15188	20296	5826	92.6	13.5	13.1	-1
100	$\cup$	15088	20073	5731	92.4	13.6	13.4	-1
150	$\wedge$	11476	22799	11325	146	21.2	24.2	-1
150	$\cap$	34223	45801	13214	217	33.7	39.8	-1
150	$\cup$	33717	44785	12881	217	34.1	40.2	-1
200	$\wedge$	20302	40401	20100	207	33.8	40	-1
200	$\cap$	60043	79905	22909	415	66	101	-1
200	$\cup$	60551	80886	23352	410	67.1	100	-1

⟨*Nef_polyhedron_2-rt.C*⟩≡

```
#include <CGAL/basic.h>
#include <CGAL/Random.h>
#include <CGAL/Homogeneous.h>
#include <vector>
#include <algorithm>
#include <iomanip>
#include <CGAL/leda_integer.h>
#include <CGAL/RPolynomial.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Filtered_extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_2.h>
#include <CGAL/basic_constructions_2.h>
```

```
#include <CGAL/Point_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/copy_n.h>
#include <CGAL/random_selection.h>
#include <CGAL/IO/Window_stream.h>
#include <CGAL/IO/Nef_polyhedron_2_Window_stream.h>
#include <LEDA/misc.h>
#include <LEDA/rat_gen_polygon.h>
#include <LEDA/rat_window.h>
#include <LEDA/param_handler.h>
template <>
struct ring_or_field<leda_integer> {
  typedef ring_with_gcd kind;
  static leda_integer gcd(const leda_integer& i1, const leda_integer& i2)
  { return ::gcd(i1,i2); }
};
typedef  CGAL::Extended_homogeneous<leda_integer>          EKernel;
typedef  CGAL::Nef_polyhedron_2<EKernel>                   Nef_polyhedron;
typedef  Nef_polyhedron::Point       Point;
typedef  Nef_polyhedron::Direction   Direction;
typedef  Nef_polyhedron::Line        Line;
typedef  Nef_polyhedron::Explorer PMExplorer;
typedef  CGAL::Filtered_extended_homogeneous<leda_integer> FEKernel;
typedef  CGAL::Nef_polyhedron_2<FEKernel> Nef_polyhedronF;
typedef  CGAL::Creator_uniform_2<leda_integer,Point> Creator;
enum { NAIVE=0, FILTERED, GENPOLY_NORMALIZED, GENPOLY };
enum { EXOR1=0, EXOR2, INTER, UNION };
static int n, Vn[4],En[4],Fn[4];
static float tt_start, tt_exor, tt_inter, tt_union;
static float t_exor[4], t_inter[4], t_union[4];
static leda_integer R;
static leda_string input_file;
static bool verbose;
⟨tool operations⟩
⟨creating random lines⟩
⟨determining the frame size⟩
⟨combination of lines into nef polyhedron⟩
⟨combination of lines into gen polygon⟩
int main(int argc, char* argv[]) {
  ⟨extract command line parameters⟩
  ⟨verbose introduction⟩
  Nef_polyhedron  N1,N2,N3,N4;
  Nef_polyhedronF NF1,NF2,NF3,NF4;
  std::vector<Line> lines1,lines2;
  ⟨creating random lines or reading them from file⟩
  ⟨write lines to a log file⟩
  combine_and_time(lines1,lines2,N1,N2,N3,N4,"NAIVE:");
  save_times(NAIVE);
  combine_and_time(lines1,lines2,NF1,NF2,NF3,NF4,"FILTERED:");
  save_times(FILTERED);
  save_structure(NF1,EXOR1); save_structure(NF2,EXOR2);
```

```
  save_structure(NF3,INTER); save_structure(NF4,UNION);
  std::ofstream poly1("nef1.log"), poly2("nef2.log");
  poly1 << NF1; poly2 << NF2;
  poly1.close(); poly2.close();
  std::ofstream poly3("nef3.log"), poly4("nef4.log");
  poly3 << NF3; poly4 << NF4;
  poly3.close(); poly4.close();
  R = min_frame_size(NF1);
  R = std::max(min_frame_size(NF2),R);
  R = std::max(min_frame_size(NF3),R);
  R = std::max(min_frame_size(NF4),R);
  leda_rat_gen_polygon G1,G2,G3,G4;
  combine_and_time_leda(lines1,lines2,G1,G2,G3,G4,true,"RGP normalized:");
  save_times(GENPOLY_NORMALIZED);
  if ( n <= 20 ) {
    combine_and_time_leda(lines1,lines2,G1,G2,G3,G4,false,"RGP:");
    save_times(GENPOLY);
  } else {
    tt_exor = -2.0; tt_inter = tt_union = -1.0;
    save_times(GENPOLY);
  }
  ⟨print runtimes⟩
  if (verbose) {
    ⟨report verbose results⟩
    ⟨visualize results⟩
  }
  return 0;
}
```

⟨*creating random lines or reading them from file*⟩≡

```
if ( input_file == "" ) {
  lines1 = std::vector<Line>(n);
  lines2 = std::vector<Line>(n);
  create_random_lines(n,lines1);
  create_random_lines(n,lines2);
} else {
  std::ifstream input(input_file);
  CGAL_assertion_msg(input,"no input log.");
  input >> n;
  lines1 = std::vector<Line>(n);
  lines2 = std::vector<Line>(n);
  for (int j=0; j<n; ++j) input >> lines1[j];
  for (int j=0; j<n; ++j) input >> lines2[j];
}
```

We determine a random line by a random point p and a random direction d. We have to avoid that the direction is trivial.

⟨*creating random lines*⟩≡

```
void create_random_lines(int r, std::vector<Line>& lines)
{ // n random lines that intersect a square of radius r
  CGAL::Random_points_in_square_2<Point,Creator> points( r );
  for (unsigned int i=0; i<lines.size(); ++i) {
    int dx(0),dy(0);
    while ( dx==0 && dy==0 ) {
      dx = CGAL::default_random.get_int(-r,r);
      dy = CGAL::default_random.get_int(-r,r);
    }
    Point p = *points++;
    Direction d(dx,dy);
    lines[i] = Line(p,d);
  }
}
```

We create elemtary nef polyhedra in a vector and use tree constuction bottom-up with the vector entries as the leafs of the tree. We use the symmetric difference operation for the construction of internal nodes of the tree. We measure the time to construct the structure in the root of the tree, which is in general an arrangement of lines where all faces are randomly marked (according to the random selection of half-spaces). The size of the resulting polyhedron is quadratic with respect to the number of lines.

⟨*combination of lines into nef polyhedron*⟩≡

```
template <typename L, typename K>
void
combine(const std::vector<L>& lines, CGAL::Nef_polyhedron_2<K>& N)
{ std::vector< CGAL::Nef_polyhedron_2<K> > V(lines.size());
  int s = lines.size(),n(s);
  for (int i=0; i<s; ++i) {
    V[i] = CGAL::Nef_polyhedron_2<K>(lines[i]);
  }
  tt_start = used_time();
  while (s > 1) {
    for (int i = 0; i<s; i+=2) {
      if ( i+1 == s )
        V[i/2] = V[i];
      else {
        V[i/2] = V[i] ^ V[i+1];
        std::cerr << ".";
      }
    }
    s = s/2 + s%2;
  }
  tt_exor = used_time(tt_start);
  N = V[0];
  std::cerr << " " << n << " lines exor combined in " << tt_exor
            << std::endl;
}
```

We use the *combine* operation to create one complex nef polyhedron from each set of lines. Then we measure the time used for the two binary operations *intersection* and *union*.

⟨*combination of lines into nef polyhedron*⟩+≡
```
template <class Nef>
void combine_and_time(const std::vector<Line>& lines1,
                      const std::vector<Line>& lines2,
                      Nef& N1, Nef& N2, Nef& N3, Nef& N4,
                      const char* title)
{
  std::cerr << "\n" << title << "\n";
  int n = lines1.size(); float t;
  combine(lines1,N1); t = tt_exor;
  combine(lines2,N2); tt_exor += t;
  tt_start = used_time();
  N3 = N1 * N2;
  tt_inter = used_time(tt_start);
  tt_start = used_time();
  N4 = N1 + N2;
  tt_union = used_time(tt_start);
  std::cerr << n << " line arrangement inter " << tt_inter << std::endl;
  std::cerr << n << " line arrangement union " << tt_union << std::endl;
}
```

Concerning general polygons we use the construction of nef polyhedra to extract quadrangles realizing half-spaces. We then use the bottom up tree combination as before.

⟨*combination of lines into gen polygon*⟩≡
```
void combine_leda(const std::vector<Line>& lines,
  leda_rat_gen_polygon& G, bool normalize=true)
{ int s = lines.size();
  std::vector< Nef_polyhedronF >      V(s);
  std::vector< leda_rat_gen_polygon > P(s);
  for (int j=0; j<s; ++j ) {
    V[j] = Nef_polyhedronF(lines[j]);
  }
  for (int i=0; i<s; ++i) {
    typedef Nef_polyhedronF::Explorer Explorer;
    typedef Explorer::Face_const_iterator Face_const_iterator;
    typedef Explorer::Halfedge_around_face_const_circulator
      Halfedge_around_face_const_circulator;
    Explorer E = V[i].explorer();
    leda_list<leda_rat_point> L;
    for (Face_const_iterator f = E.faces_begin(); f != E.faces_end();
         ++f) {
      if ( !E.mark(f) ) continue;
      Halfedge_around_face_const_circulator e(E.halfedge(f)), ee(e);
      CGAL_For_all(e,ee) {
        L.append(cgal_to_leda(e->vertex()->point()));
      }
    }
    P[i] = leda_rat_gen_polygon(L);
    if (normalize) P[i].normalize();
```

```
    }
    tt_start = used_time();
    while (s > 1) {
      for (int i = 0; i<s; i+=2) {
        if ( i+1 == s )
          P[i/2] = P[i];
        else {
          P[i/2] = P[i].sym_diff(P[i+1]);
          if (normalize) P[i/2].normalize();
          std::cerr << ".";
        }
      }
      s = s/2 + s%2;
    }
    G = P[0];
    tt_exor = used_time(tt_start);
    std::cerr << " " << lines.size() << " gen_polygons exor combined in "
              << tt_exor << std::endl;
  }
```

The combination of generic polygons representing half-spaces into larger units follows the same scheme as above. First create the symmetric difference structure of the input objects (elementary polygons representing half-spaces). Then calculate the intersection and union of the two objects.

⟨*combination of lines into gen polygon*⟩+≡

```
  void combine_and_time_leda(
    const std::vector<Line>& lines1, const std::vector<Line>& lines2,
    leda_rat_gen_polygon& G1, leda_rat_gen_polygon& G2,
    leda_rat_gen_polygon& G3, leda_rat_gen_polygon& G4,
    bool normalize, const char* title)
  {
    std::cerr << "\n" << title << "\n";
    int n = lines1.size(); float t;
    combine_leda(lines1,G1,normalize); t = tt_exor;
    combine_leda(lines2,G2,normalize); tt_exor += t;
    tt_start = used_time();
    G3 = G1.intersection(G2);
    tt_inter = used_time(tt_start);
    tt_start = used_time();
    G4 = G1.unite(G2);
    tt_union = used_time(tt_start);
    std::cerr << n << " gen_polygon inter " << tt_inter << std::endl;
    std::cerr << n << " gen_polygon union " << tt_union << std::endl;
  }
```

To determine the frame size is easy when we have a topologically correct nef polyhedron. For standard points the coordinates are a lower bound for the frame radius. For non-standard points the intersection of the underlying lines with the angular bisectors define a lower bound for the fram radius.

⟨*determining the frame size*⟩≡
```
leda_integer min_frame_size(const Nef_polyhedronF& N)
{
  typedef Nef_polyhedronF::Extended_kernel EKernel;
  typedef EKernel::Standard_RT Standard_RT;
  typedef Nef_polyhedronF::Explorer       Explorer;
  typedef Explorer::Topological_explorer  TExplorer;
  typedef Explorer::Vertex_const_iterator Vertex_const_iterator;
  TExplorer D = N.explorer();
  EKernel& E = Nef_polyhedronF::EK;
  Vertex_const_iterator vit = D.vertices_begin(),
                        vend = D.vertices_end();
  leda_integer frame_radius(0);
  for (; vit != vend; ++vit) {
    if ( E.is_standard(D.point(vit)) ) {
      Point p = E.standard_point(D.point(vit));
      Standard_RT m = std::max(p.hx()/p.hw(), p.hy()/p.hw());
      frame_radius = std::max(frame_radius,m+1);
    } else { // non-standard
      Line l = E.standard_line(D.point(vit));
      if ( abs(l.a()) != abs(l.b()) ) {
        Standard_RT m = std::max( abs(l.c()/(l.a()+l.b())),
                                  abs(l.c()/(l.a()-l.b())) );
        frame_radius = std::max(frame_radius,m+1);
      } else {
        frame_radius = std::max(frame_radius,abs(l.c()/l.a()));
      }
    }
  }
  return frame_radius;
}
```

⟨*verbose introduction*⟩≡
```
SETDTHREAD(41);
int nv = n*(n-1)/2 + 2*n + 4;
std::cerr << CGAL::pointlocationversion << std::endl;
std::cerr << CGAL::sweepversion << std::endl;
std::cerr << "creating arrangement of " << nv << " vertices\n";
```

⟨*extract command line parameters*⟩≡
```
leda_param_handler H(argc,argv,".rt",false);
H.add_parameter("number_of_lines:-n:int:10");
H.add_parameter("file_of_lines:-i:string:");
H.add_parameter("verbose:-v:bool:false");
leda_param_handler::init_all();
H.get_parameter("-n",n);
H.get_parameter("-i",input_file);
H.get_parameter("-v",verbose);
```

⟨*print runtimes*⟩≡
```
std::cerr << std::endl;
std::cout << setprecision(3);
```

```
std::cout << n << " & \\EXOR  & "
  << (Vn[EXOR1]+Vn[EXOR2])/2 << " & " << (En[EXOR1]+En[EXOR2])/2 << " & "
  << (Fn[EXOR1]+Fn[EXOR2])/2 << " & "
  << (t_exor[NAIVE]/2) << " & " << (t_exor[FILTERED]/2) << " & "
  << (t_exor[GENPOLY_NORMALIZED]/2) << " & " << (t_exor[GENPOLY]/2)
  << "\\\\\n";
std::cout << n << " & \\INTER & "
  << Vn[INTER] << " & " << En[INTER] << " & "
  << Fn[INTER] << " & "
  << t_inter[NAIVE] << " & " << t_inter[FILTERED] << " & "
  << t_inter[GENPOLY_NORMALIZED] << " & " << t_inter[GENPOLY]
  << "\\\\\n";
std::cout << n << " & \\UNION & "
  << Vn[UNION] << " & " << En[UNION] << " & "
  << Fn[UNION] << " & "
  << t_union[NAIVE] << " & " << t_union[FILTERED] << " & "
  << t_union[GENPOLY_NORMALIZED] << " & " << t_union[GENPOLY]
  << "\\\\ \\hline\n";
```

⟨*report verbose results*⟩≡

```
std::cerr << std::endl << "frame size = " << R << std::endl;
N1.explorer().print_statistics();
N2.explorer().print_statistics();
N3.explorer().print_statistics();
N4.explorer().print_statistics();
Nef_polyhedronF::EK.print_statistics();
ISOTEST(N1,NF1)
ISOTEST(N2,NF2)
ISOTEST(N3,NF3)
ISOTEST(N4,NF4)
```

⟨*write lines to a log file*⟩≡

```
std::ofstream log("nef-rt.log");
CGAL_assertion_msg(log,"no output log nef-rt.log");
log << n << std::endl;
for (int i=0; i<n; ++i) log << lines1[i] << " ";
log << std::endl;
for (int i=0; i<n; ++i) log << lines2[i] << " ";
log << std::endl;
log.close();
```

⟨*visualize results*⟩≡

```
CGAL::Window_stream W(600,600);
W.init(-CGAL::frame_default,CGAL::frame_default,-CGAL::frame_default);
W.set_show_coordinates(true);
W.set_grid_mode(5);
W.set_node_width(3);
W.display(0,0);
W << N3;
W.read_mouse();
W << N4;
W.read_mouse();
```

```
W.clear();
{
  leda_rat_polygon p;
  forall_polygons(p,G3)
    W.draw_filled_polygon(p.to_polygon(),leda_black);
}
W.read_mouse();
```

⟨*tool operations*⟩≡

```
template <typename P>
leda_rat_point cgal_to_leda(const P& p)
{ return leda_rat_point(p.hx().eval_at(R),p.hy().eval_at(R),
                        p.hw()); }

void save_times(int i)
{ t_exor[i]=tt_exor; t_inter[i]=tt_inter; t_union[i]=tt_union; }

void save_structure(const Nef_polyhedronF& N, int i)
{
  Vn[i] = N.explorer().number_of_vertices();
  En[i] = N.explorer().number_of_edges();
  Fn[i] = N.explorer().number_of_faces();
}

#define ISOTEST(n1,n2)\
assert(n1.explorer().number_of_vertices()==\
       n2.explorer().number_of_vertices());\
assert(n1.explorer().number_of_edges()==n2.explorer().number_of_edges());\
assert(n1.explorer().number_of_faces()==n2.explorer().number_of_faces());
```

1.10 Printing Nef polyhedra

⟨*Nef_polyhedron_2-ps.C*⟩≡

```
#include <CGAL/basic.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Filtered_extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_2.h>
#include <CGAL/IO/Nef_polyhedron_2_PS_stream.h>
#include <LEDA/stream.h>

template <>
struct ring_or_field<leda_integer> {
  typedef ring_with_gcd kind;
};

//typedef CGAL::Extended_homogeneous<leda_integer> EKernel;
typedef CGAL::Filtered_extended_homogeneous<leda_integer> EKernel;
typedef  CGAL::Nef_polyhedron_2<EKernel> Nef_polyhedron;

using namespace CGAL;

int main(int argc, char* argv[]) {
  // Create test point set. Prepare a vector for 1000 points.
```

```
  if ( argc == 1 ) {
    std::cout << argv[0] << " input_file\n";
    return 1;
  }
  file_istream f(argv[1]);
  if (f) {
    Nef_polyhedron N;
    f >> N;
    ps_file PS(10,10,"nef.ps");
    PS << N;
  }
  return 0;
}
```

Bibliography

[Bie94] H. Bieri. Boolean and topological operations for Nef polyhedra. In *CSG 94 Set-theoretic Solid Modelling: Techniques and Applications*, pages 35–53. Information Geometers, 1994.

[Bie95] H. Bieri. Nef Polyhedra: A Brief Introduction. *Computing Suppl. Springer-Verlag*, 10:43–60, 1995.

[Bie96] H. Bieri. Two basic operations for Nef polyhedra. In *CSG 96 Set-theoretic Solid Modelling: Techniques and Applications*, pages 337–356. Information Geometers, 1996.

[Bie98] H. Bieri. Representation conversions for Nef Polyhedra. *Computing Suppl. Springer-Verlag*, 13:27–38, 1998.

[BN88] H. Bieri and W. Nef. Elementary set operations with d-dimensional polyhedra. In Hartmut Noltemeier, editor, *Computational Geometry and its Applications*, volume 333 of *LNCS*, pages 97–112. Springer, March 1988.

[dBvKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry : Algorithms and Applications*. Springer, Berlin, 1997.

[DMY93] K. Dobrindt, K. Mehlhorn, and M. Yvinec. A complete and efficient algorithm for the intersection of a general and a convex polyhedron. Technical Report 2023, INRIA, 1993.

[Fer95] V. Ferrucci. Representing Nef Polyhedra. In Kunii Shin, editor, *Proceedings of the 3rd Pacific conference on computer graphics and applications - Pacific Graphics 95*, pages 459–511. World Scientific, 1995.

[Grü67] B. Grünbaum. *Convex Polytopes*, volume 16 of *Pure and Applied Mathematics : A Series of Monographs and Textbooks*. Interscience Publ., London;New York;Sydney, 1967.

[Ket99] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *CGTA: Computational Geometry: Theory and Applications*, 13, 1999.

[Lef71] S. Lefschetz. *Introduction to topology*, volume 11 of *Princeton mathematical series*. Princeton Univ. Press, 8. print. edition, 1971.

[MMS94] J.S.B. Mitchell, D. Mount, and S. Suri. Query-sensitive ray shooting. In *Proceedings of the 10th Annual Symposium on Computational Geometry*, pages 359–368, Stony Brook, NY, USA, June 1994. ACM Press.

[MN99] K. Mehlhorn and S. Näher. *LEDA, A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[Nef78] W. Nef. *Beiträge zur Theorie der Polyeder mit Anwendungen in der Computergraphik*. Herbert Lang & Cie AG, Bern, 1978.

[NS90] W. Nef and P.-M. Schmidt. Computing a sweeping-plane in regular ("general") position: a numerical and a symbolic solution. *Journal of Symbolic Computation*, 10(6):633–646, december 1990.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry : An Introduction*. Springer, 1985.

[RO90] J.R. Rossignac and M.A. O'Connor. SGC: A dimension-independent model for pointsets with internal structures and incomplete boundaries. In M. J. Wozny, J. U. Turner, and K. Preiss, editors, *Geometric Modeling for Product Engineering*, pages 145–180. Elsevier Science Publishers B.V., North Holland, 1990.

[SM00] M. Seel and K. Mehlhorn. Infimaximal Frames: a framework to make lines look like segments. Technical Report MPI-I-2000-1-005, Max-Planck-Institut für Informatik, Saarbrücken, 2000.

2 Segment Intersection

2.1 Introduction

This document describes a generic sweep algorithm of line segments along the lines of the algorithm which is part of the LEDA library. We basically transferred the segment sweep algorithm as described in the LEDA book [MN99] into our generic sweep framework *generic_sweep*. We descibe special adaptations and refer the user to the description in [MN99, chapter 10] for a deeper understanding.

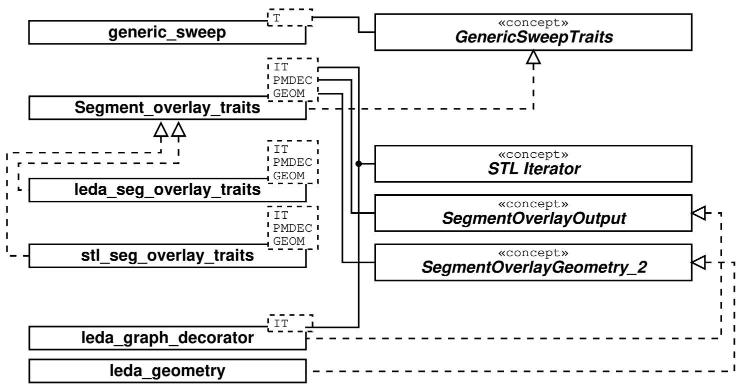


Figure 2.1: The design of the segment overlay module. *Segment_overlay_traits* implements the concept *GenericSweepTraits*. There are actually two instances *leda_seg_overlay_traits* and *stl_seg_overlay_traits* realizing *Segment_overlay_traits* depending on the module configuration. The three template parameters allow an adaptation depending on input, output, and geometry.

To use our generic sweep framework we implement a traits model *Segment_overlay_traits* which is plugged into the class *generic_sweep*<*T*> (the bottom layer). For the concept of the parameter *T* please refer to class *GenericSweepTraits* in the appendix on page 331. For the functionality of class *generic_sweep* see the manual page on page 304. *generic_sweep* $<$ *Segment_overlay_traits* $< ... \gg$ is a generic sweep framework for the calculation of the overlay of segments.

If you browse the original algorithm *SWEEP_SEGMENTS* with respect to code dependencies, you find that it is hard-wired to several LEDA modules. The wires of the original algorithm are the *geometric kernel* which is used, the *input interface* which is a list of segments, and the *output interface* which is a LEDA embedded graph. We decouple the above from concrete data types by introducing concepts for the three units: input, output, geometry.

The *input concept* is easy. We use iterators defining an iterator range of segments (corresponding to the list of segments in the LEDA sweep).

The *output concept* is a plane map data structure like LEDA plane maps (bidirected, embedded graphs). For an introduction refer to [MN99, chapter 8]. But of course there are several other standard data structures in the literature like the *Halfedge Data Structures* (HDS), and *Directed Cyclic Edge Lists* (DCEL). See for example the textbooks [dBvKOS97, PS85] and the CGAL HDS implementation in the manual [CGA]. Sometimes the output has to be enriched by some additional bookkeeping data structures (e.g. maps) to associate additional information to the vertices and edges of the graph.

The *geometric concept* contains the geometry used for the algorithmic decisions of the sweep: geometric types like points and segments and the primitive operations on them. Our correctness considerations are based on affine planar geometry. However this sweep framework works also instantiated with other geometry models.

The genericity is achieved by encapsulating the three concepts into three template parameters of *Segment_overlay_traits*<> which allows a user to transport his geometry, geometric primitives and his input and output structure into the segment sweep framework. We will describe the concept of the geometric types, primitives and the concept of the output graph structure below. We first give an abstract introduction of the output produced.

The ouput of the algorithm is the result of transformations of an output object triggered by method calls of the traits class. There are some methods which can be used to manipulate a graph structure $G = (V,E)$. If the implementation of those graph manipulation methods follows the semantic description below then the output graph obtains a certain structure. Additionally there is also some kind of message passing associated with the output structure. This allows a user to refine necessary additional information from the sweep.

Assume the output contains a graph structure $G = (V,E)$ which can represent an embedded plane map (a bidirected graph where reversal edges are paired and where the nodes are embedded into the plane by associating point coordinates). We state some properties of the output production:

O1. All end points and intersection points of segments are called events and trigger calls to create new nodes $v \in V$ which obtain the knowledge about their embedding via a point p. The creation is done in the lexicographic order on points as specified by the user in the geometric traits class.

O2. The sweep explores the skeleton of the planar subdivision induced by the set of input segments of the iterator range. Thus for each segment s there are method calls which can be used to create edges in G such that the straight line embedding of their union corresponds to s.

O3. Each halfedge $e \in E$ gets to know a list of input segments supporting its straight line embedding.

O4. Each node $v \in V$ gets to know a halfedge below (vertical ray-shooting property) where degeneracies are broken with a left-closed perturbation scheme (all edges include their left source node during the ray shoot). Additionally each node $v \in V$ gets to know the input segments which start at, end at or contain its embedding $point(v)$. Finally each node gets to know explicitly if it is supported by a trivial input segment.

O5. If the edge creation operations follow the semantic description then each node v has an adjacency list such that visiting all adjacent nodes while iterating the adjacency list corresponds to a counterclockwise rotation around v (the straight line drawing of G is counterclockwise *order-preserving*). All adjacency lists additionally have the property of a *forward-prefix*. This means that forward-oriented edges[1] build a prefix in each adjacency list.

[1] an edge e is called forward-oriented when $point(source(e)) <_{lex} point(target(e))$.

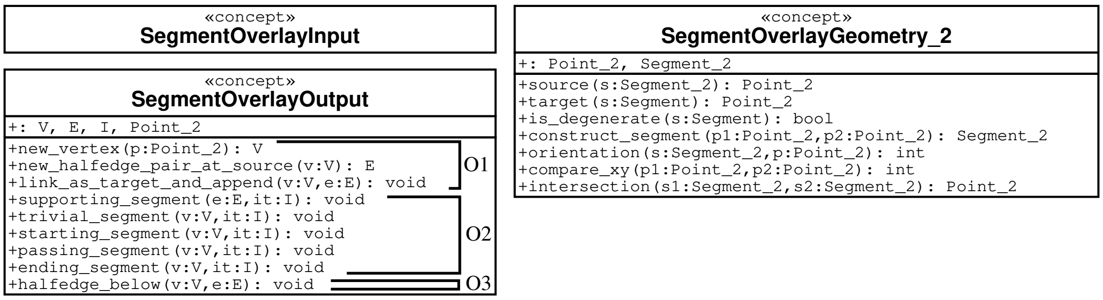


Figure 2.2: The three concepts that allow adaptation of the generic segment sweep. In the output concept the abbreviations are V for *Vertex_handle*, E for *Halfedge_handle*, and I for *Iterator*.

The interfaces of the three concepts are depicted in figure 2.2. For the actual semantics please consult the manual pages for *SegmentOverlayOutput* on page 325 and *SegmentOverlayGeometry_2* on page 325 in the appendix.

2.1.1 Formalizing the sweep — Invariants

We sweep the plane from left to right by a vertical line *SL*. Whenever we encounter an event point we have to take actions to produce the output structure. Both endpoints of each segment and non-degenerate intersection points of any two non-overlapping segments define our *events*. To ensure the correct actions we resort to a list of invariants on which we can rely just before we encounter an event and which we ensure by certain actions for the status thereafter.

The sweep is determined by an interaction between two major data structures, an event queue *XS* and a sweep status structure *YS*. The event queue *XS* controls the stepping of *SL* across the plane. We store a point as the key of each event. The order of the events corresponds to the lexicographical order on all points as defined in the geometry kernel. *YS* stores segments intersecting the sweep line *SL* ordered according to their intersection points from bottom to top. Note that for a segment s in *YS* $\mathit{source}(s) <_{lex} \mathit{target}(s)$ according to our lexicographic order on points.

The above description talks about vertical sweep lines and geometry which is left and right of the sweep line. As soon as there are events with identical x-coordinates and vertical segments we have to be more accurate. Imagine a sweep line which is slanted by an infinitesimal angle counterclockwise thus processing the points on the vertical line bottom-up. A more accurate intuition is created by a sweepline consisting of an infinitesimal small step down, where the vertical ray down is right of the current sweep position and the vertical ray up is left of the current sweep position and the horizontal step marks the current position while going from $-\infty$ to $+\infty$ along the y-axis of the coordinate system. For an extensive treatment please refer to [MN99, chapter 10].

We treat the degenerate cases of several segments ending, intersecting, and starting in one point explicitly in our code. We also handle the possibility that several segments overlap. This implies that just before an event the sweep line may intersect a whole bundle of segments containing the event. Some extend through it, some end there. And just after the event the bundle of the segments extending though the event may be enriched by several segments starting at the event. The segments of both bundles can be ordered according to their points of intersection with the sweep line. In case of overlapping segments their point of intersection with the sweep line is identical. To break the tie

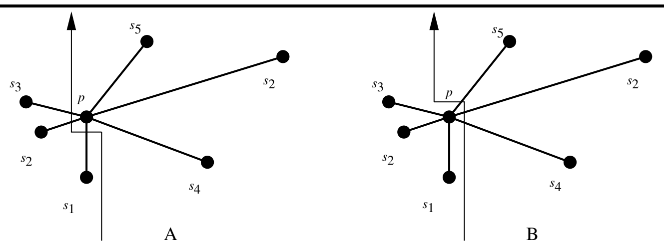


Figure 2.3: Sketching the sweepline intuition in the case of degeneracies. The left figure shows the sweepline before the event at p. The right figure shows the figure just after the event at p.

we take the order on the identity of each input segment[2]. Note that due to the degeneracy events can have multiple character.

Invariant 1: The event queue *XS* is a sorted sequence of items $<p,x>$ called events, where p is the embedding point of the event and x is an associated information link. *Starting* and *ending* events refer to the endpoints of all segments. *Intersection* events are defined by the points of intersection of two non-overlapping segments. At any sweep position *XS* contains all starting and ending events and all intersection events right of *SL* that are the result of an intersection of segments that are neighbors in *YS*. The order in *XS* is the lexicographic order on points *compare_xy*( ) introduced by the geometry concept.

Invariant 2: The sweep status structure *YS* is a sorted sequence of items $<s,y>$ where s is a segment intersected by the sweep line. The order is defined by the points of intersection of the segments and the sweep line from bottom to top.

Consider any bundle of segments ending at or extending through an event. We want to save on geometric calculations. Therefore the information slot of the items in *YS* is used to identify such bundles.

Invariant 3: Let $<s,y>$ and $<s',y'>$ be two successing items in *YS*. The information y is used as a flag how the two segments are geometrically related. If s and s' are overlapping at the current sweep position then y points to its successor item $<s',y'>$ in *YS*. If s and s' intersect right of the sweep line then y points to the corresponding event in *XS*. Otherwise it is the null handle.

Additionally for all items *it* we know an edge in the output graph that is supported by the segment via a map *Edge_of*$[it]$.

Invariant 4: For all intersection events and ending events $<p,x>$ in *XS* the information x is a link to an item $<s,y>$ in *YS* such that the segment s contains p. For ending events the invariant is established as soon as the segment s enters *YS*.

This construction allows us to shortcut from an event into the range of interest within *YS* without using the standard (logarithmic) search operations of *YS*.

Lemma 2.1.1: Starting from any intersection or ending event we can identify the bundle of segments/items in *YS* ending at or extending through the event in time proportional to the size of the bundle.

[2] Internally we handle segments via pointers. Then their identity is just the memory address.

Proof. Consider an event <p,x> that is not only starting event. Then x is a link to an item <s,y> in *YS* such that s contains p due to Invariant 4. Iterating locally up and down starting from x allows us to identify the range of items in *YS* whose segments contain p by the marking links according to Invariant 3. ☐

At last, at each event we ensure partial output correctness that leads to global output correctness after the last event.

Invariant 5: Assume the output model's operations are defined according to our specification. Then the overlay of all segments that are fully left of our sweep line is correctly calculated.

The following hashing idea saves again on geometric calculations and search.

Invariant 6: For each pair of segments $(s1, s2)$ in *YS* and intersecting right of *SL* which have been neighbors in *YS* once (and might have been separated afterwards) there's a hash table short-cut to the corresponding intersection event $it = IEvent(s1, s2)$.

Note that the algorithmic correctness of this framework has two aspects. There are global considerations and local considerations. The global considerations concern issues like why the algorithm terminates and why it does calculate the output we ask for. The local considerations concern the aspects of the event handling. Namely, why our event handling and the intialization phase of our framework ensures the invariants stated above. Taking both issues together we obtain the certainty that our code module actually calculates the overlay correctly.

For the global correctness we will see that all starting and ending events are handled in our code and inflated into the machinery in the initialization phase. One exception that we have to incorporate into our code is the occurance of trivial segments. The final question is if we catch all intersection events? There's a trivial observation why we cannot miss any such event.

Lemma 2.1.2: If we ensure that all our invariants hold at all events then we cannot miss any intersection event.

Proof. Assume we miss an intersection event *ev*. If we miss several take the lexicographically smallest one. Then the event just before *ev* was correctly treated and the two segments that imply *ev* are neighbors in *YS*. But Invariant 1 tells us that *ev* is in *XS* which leads to a contradiction. ☐

Note that global termination is not a big issue in sweep frameworks. As soon as all events have gone we stop the iteration. Note finally that if we locally keep Invariant 5 just after each event then we also know that our result is correctly calculated. We now will link the above insights to the code.

2.1.2 Two generic sweep traits models

We use our generic plane sweep paradigm to execute the sweep. We offer two models to plug into the *generic_sweep* framework. For the concept see *GenericSweepTraits* in the appendix. One based on LEDA [MN99] and including several runtime optimizations, the other one based purely on STL data structures [MS96]. This design was necessary to allow using the sweep module when CGAL is installed without the presence of LEDA. The sweep traits class wraps it all.

⟨*Segment_overlay_traits.h*⟩≡

```
⟨CGAL Header⟩
#ifndef CGAL_SEGMENT_OVERLAY_TRAITS_H
#define CGAL_SEGMENT_OVERLAY_TRAITS_H
```

```
#include <assert.h>
#undef _DEBUG
#define _DEBUG 23
#include <CGAL/Nef_2/debug.h>
//#define INCLUDEBOTH
#if defined(CGAL_USE_LEDA) || defined(INCLUDEBOTH)
⟨leda segment overlay model⟩
#endif // defined(CGAL_USE_LEDA) || defined(INCLUDEBOTH)
#if !defined(CGAL_USE_LEDA) || defined(INCLUDEBOTH)
⟨stl segment overlay model⟩
#endif // !defined(CGAL_USE_LEDA) || defined(INCLUDEBOTH)
namespace CGAL {
#ifdef CGAL_USE_LEDA
#define Segment_overlay_traits leda_seg_overlay_traits
static const char* sweepversion = "LEDA segment overlay sweep";
#else
#define Segment_overlay_traits stl_seg_overlay_traits
static const char* sweepversion = "STL segment overlay sweep";
#endif
} // namespace CGAL
#include <CGAL/generic_sweep.h>
#endif // CGAL_SEGMENT_OVERLAY_TRAITS_H
```

⟨leda segment overlay model⟩≡

```
#include <LEDA/tuple.h>
#include <LEDA/slist.h>
#include <LEDA/list.h>
#include <LEDA/map.h>
#include <LEDA/map2.h>
#include <LEDA/sortseq.h>
#include <LEDA/p_queue.h>
#include <utility>
#include <strstream>
namespace CGAL {
⟨leda debugging routines⟩
⟨leda segment overlay traits class⟩
} // namespace CGAL
```

2.1.3 The LEDA traits model

Our class obtains three template types which have to be models for the corresponding concepts described below.

⟨leda segment overlay traits class⟩≡

```
template <typename IT, typename PMDEC, typename GEOM>
class leda_seg_overlay_traits {
public:
  ⟨leda introducing the types from the traits⟩
  ⟨leda internal segment type⟩
  // types interfacing the generic sweep frame:
```

```
  ITERATOR its, ite;
  OUTPUT&  GO;
  const GEOMETRY& K;
```
  ⟨*leda order types for segments and points*⟩
  ⟨*leda sweep data structures*⟩
  ⟨*leda helping operations*⟩
  ⟨*leda operation for keeping the intersection invariant*⟩
  ⟨*leda initialization of the sweep*⟩
  ⟨*leda iteration control*⟩
  ⟨*leda handling the event*⟩
  ⟨*leda postprocessing of the sweep*⟩
```
}; // leda_seg_overlay_traits
```

The following types are introduced by the traits classes. See the the concept descriptions *SegmentOverlayOutput* for *PMDEC* and *SegmentOverlayGeometry_2* for *GEOM*. The iterator concept required is that of an STL input iterator.

⟨*leda introducing the types from the traits*⟩≡
```
typedef IT                               ITERATOR;
typedef std::pair<IT,IT>                 INPUT;
typedef PMDEC                            OUTPUT;
typedef typename PMDEC::Vertex_handle    Vertex_handle;
typedef typename PMDEC::Halfedge_handle  Halfedge_handle;
typedef GEOM                             GEOMETRY;
typedef typename GEOMETRY::Point_2       Point_2;
typedef typename GEOMETRY::Segment_2     Segment_2;
```

We define an internal segment type *ISegment* based on a pointer to allow two constructions. At first we want to be able to couple the internal segment to the input object. We achieve this by maintaining both as a pair *two_tuple*<*Segment*,*ITERATOR*> in a list. Secondly our internal segment type is just a pointer to such a pair. We can thus not only compare internal segments geometrically by the pair's first component, but also check identity by the pointer address.

⟨*leda internal segment type*⟩≡
```
typedef leda_two_tuple<Segment_2,ITERATOR> seg_pair;
typedef seg_pair*                          ISegment;
typedef leda_list<seg_pair>                IList;
typedef typename IList::iterator           ilist_iterator;
```

The predicates below are solely based on a few geometric kernel predicates. The clever observation is the fact, that the comparison predicate *cmp_segs_at_sweepline* is only called with one segment containing the sweep point. We know that assertion from the specification of LEDA sortseqs. Compared to the LEDA sweep algorithm we add non-geometric sentinel segments to avoid checking of boundary cases.

⟨*leda order types for segments and points*⟩≡
```
class cmp_segs_at_sweepline : public leda_cmp_base<ISegment>
{ const Point_2& p;
  ISegment s_bottom, s_top; // sentinel segments
  const GEOMETRY& K;
```

```
public:
 cmp_segs_at_sweepline(const Point_2& pi,
   ISegment s1, ISegment s2, const GEOMETRY& k) :
   p(pi), s_bottom(s1), s_top(s2), K(k) {}
 int operator()(const ISegment& is1, const ISegment& is2) const
 { // Precondition: p is identical to the left endpoint of s1 or s2.
   if ( is2 == s_top || is1 == s_bottom ) return -1;
   if ( is1 == s_top || is2 == s_bottom ) return 1;
   if ( is1 == is2 ) return 0;
   const Segment_2& s1 = is1->first();
   const Segment_2& s2 = is2->first();
   int s = 0;
   if ( p == K.source(s1) )      s =   K.orientation(s2,p);
   else if ( p == K.source(s2) ) s = - K.orientation(s1,p);
   else ASSERT(0,"compare error in sweep.");
   if ( s || K.is_degenerate(s1) || K.is_degenerate(s2) )
     return s;
   s = K.orientation(s2,K.target(s1));
   if (s==0) return ( is1 - is2 );
   // overlapping segments are not equal
   return s;
 }
};
```

The lexicographic order on points is just transferred from the geometric kernel.

⟨*leda order types for segments and points*⟩+≡

```
struct cmp_pnts_xy : public leda_cmp_base<Point_2>
{ const GEOMETRY& K;
public:
 cmp_pnts_xy(const GEOMETRY& k) : K(k) {}
 int operator()(const Point_2& p1, const Point_2& p2) const
 { return K.compare_xy(p1,p2); }
};
```

We use several LEDA data structures. The x-structure *XS* is an ordered sequence of items based on the key type *Point_2*. For the item concept of LEDA refer to the manual [MNSU99] or the LEDA book. We associate a link into the y-structure for intersection and ending events to save on unnecessary search operations within *YS*. When we reach an event at *p_sweep* we can thus shortcut to find an item in *YS* which contains a segment (as its key) containing *p_sweep*. The y-structure *YS* is a sorted sequence of *ISegments*. The associated *seq_item* link serves two purposes: (1) it bundles segments in *YS* together by pointing to the (lexicographic) next event which they contain. (2) it bundles segments which overlap. See the LEDA book for a more elaborate description.

During the sweep we associate edges from the constructed output graph to the items in YS. We use a hash map *map*<*seq_item*,*Halfedge_handle*> *Edge_of* for this purpose. Finally for each event point there is a possible sequence of input segments starting at the event. To maintain this sequence we use a priority queue *p_queue*<*Point_2*,*ISegment*> *SQ*.

When two segments *s1*,*s2* become neighbors in *YS* we check if they intersect right of the sweep line. If they do we calculate the intersection point *p* and insert a corresponding event into *XS*. Now it can happen that *s1* and *s2* get again separated by a new segment before *p* is reached. We have several possibilities in this case. We could remove the event at *p* again to keep the space bound implied by the Invariant 1. However the size of the output structure anyway comprises the space for keeping the event in *XS*. We can even do better with respect to runtime. Instead of recalculating the geometric intersection information when *s1* and *s2* become neighbors again we can try to recover the previously calculated event from the two dimensional hash $map2\langle ISegment, ISegment, seq\_item\rangle\ IEvent$.

The additional members are used as a central place for data storage. *event* provides a handle on the current event queue item. *p_sweep* is the position of the current event which is also used from the segment comparison object *SLcmp*.

⟨*leda sweep data structures*⟩≡

```
typedef leda_sortseq<Point_2,seq_item>      EventQueue;
typedef leda_sortseq<ISegment,seq_item>     SweepStatus;
typedef leda_p_queue<Point_2,ISegment>      SegQueue;
typedef leda_map<seq_item,Halfedge_handle>  AssocEdgeMap;
typedef leda_slist<ITERATOR>                IsoList;
typedef leda_map<seq_item, IsoList* >       AssocIsoMap;
typedef leda_map2<ISegment,ISegment,seq_item> EventHash;
  seq_item                  event;
  Point_2                   p_sweep;
  cmp_pnts_xy               cmp;
  EventQueue                XS;
  seg_pair                  sl,sh;
  cmp_segs_at_sweepline     SLcmp;
  SweepStatus               YS;
  SegQueue                  SQ;
  EventHash                 IEvent;
  IList                     Internal;
  AssocEdgeMap              Edge_of;
  AssocIsoMap               Isos_of;
  leda_seg_overlay_traits(const INPUT& in, OUTPUT& G,
    const GEOMETRY& k) :
    its(in.first), ite(in.second), GO(G), K(k),
    cmp(K), XS(cmp), SLcmp(p_sweep,&sl,&sh,K), YS(SLcmp), SQ(cmp),
    IEvent(0), Edge_of(0), Isos_of(0) {}
```

We define some code short cuts.

⟨*leda helping operations*⟩≡

```
leda_string dump_structures() const
{
  std::ostrstream out;
  out << "SQ= ";
  pq_item pqit;
  forall_items(pqit,SQ) {
    if (SQ.prio(pqit)==XS.key(XS.succ(XS.min_item())))
    { out << SQ.inf(pqit)->first(); }
    pqit = SQ.next_item(pqit);
  }
  seq_item sit;
```

```
    out << "\nXS=\n";
    forall_items(sit,XS)
      out << "  " << XS.key(sit) << " " << XS.inf(sit)
          <<std::endl;
    out << "YS=\n";
    for( sit = YS.max_item(); sit; sit=YS.pred(sit) )
      out << "  "<<YS.key(sit)->first()<<" "<<YS.inf(sit)<<std::endl;
    out << '\0';
    leda_string res(out.str()); out.freeze(0); return res;
  }
```

⟨*leda helping operations*⟩+≡

```
  Point_2 source(ISegment is) const
  { return K.source(is->first()); }
  Point_2 target(ISegment is) const
  { return K.target(is->first()); }
  ITERATOR original(ISegment s) const
  { return s->second(); }
  int orientation(seq_item sit, const Point_2& p) const
  { return K.orientation(YS.key(sit)->first(),p); }
  bool collinear(seq_item sit1, seq_item sit2) const
  { Point_2 ps = source(YS.key(sit2)), pt = target(YS.key(sit2));
    return ( orientation(sit1,ps)==0 &&
             orientation(sit1,pt)==0 );
  }
```

Most events trigger changes in the segment sequence along the sweep line. We have to reflect such changes in a test for new intersection events right of the sweep line as soon as two segments become neighbors. The following code ensures the Invariants 2, 3, 4 and uses the hash tuning of Invariant 6. *s1* is the successor of *s0* in *YS*, hence, *s0* and *s1* intersect right or above of the event iff *target*(*s1*) is not left of the line supporting *s0*, and *target*(*s0*) is not right of the line supporting *s1*. In this case we intersect the underlying lines.

⟨*leda operation for keeping the intersection invariant*⟩≡

```
  void compute_intersection(seq_item sit0)
  {
    seq_item sit1 = YS.succ(sit0);
    if ( sit0 == YS.min_item() || sit1 == YS.max_item() ) return;
    ISegment s0 = YS.key(sit0);
    ISegment s1 = YS.key(sit1);
    int or0 = K.orientation(s0->first(),target(s1));
    int or1 = K.orientation(s1->first(),target(s0));
    if ( or0 <= 0 && or1 >= 0  ) {
      seq_item it = IEvent(YS.key(sit0),YS.key(sit1));
      if ( it==0 ) {
        Point_2 q = K.intersection(s0->first(),s1->first());
        it = XS.insert(q,sit0);
      }
```

```
    YS.change_inf(sit0, it);
  }
}
```

Event Handling

We start with the knowledge that our invariants from Section 2.1.1 hold. First we create a new vertex v in the ouput structure. Then we work in four phases: (1) We handle the ingoing bundle which ends at v. (2) We communicate all the knowledge about the new vertex (3) We have to deal with the segments starting at *p_sweep*. (4) We clean up to reestablish missing invariants.

⟨*leda handling the event*⟩≡
```
void process_event()
{
  Vertex_handle v = GO.new_vertex(p_sweep);
  seq_item sit = XS.inf(event);
  ⟨leda handling ending and passing segments⟩
  ⟨leda completing additional information of the new vertex⟩
  ⟨leda inserting new segments starting at nodes⟩
  ⟨leda enforcing the invariants for YS⟩
}
```

We first have to locate the bundle going through *p_sweep*. We deviate from the implementation of the LEDA algorithm *SWEEP_SEGMENTS* in one respect. For each segment in *YS* we store a bidirected edge pair extending along the segment. When we reach an event point we connect these edges to the newly created node. Note that this change is necessary if you use halfedge data structures for the output. The original approach used temporarily incomplete edge pairs (only forward directed halfedges) and coupled and embedded them in a postprocessing phase. But space minimally maintained halfedges like those of the CGAL HDS can only exist in pairs.

If there is a non-nil item $sit = XS.inf(event)$ associated with *event*, $key(sit)$ is either an ending or passing segment. We use *sit* as an entry point to compute the bundle of segments ending at or passing through *p_sweep*. In particular, we compute the first (*sit_first*) and the successor (*sit_succ*)) and predecessor (*sit_pred*) items.

⟨*leda handling ending and passing segments*⟩≡
```
seq_item sit_succ(0), sit_pred(0), sit_pred_succ(0), sit_first(0);
if (sit == nil)
  ⟨leda check p_sweep in YS⟩
/* If no segment contains p_sweep then sit_pred and sit_succ are
   correctly set after the above locate operation, if a segment
   contains p_sweep sit_pred and sit_succ are set below when
   determining the bundle.*/
if (sit != nil) { // key(sit) is an ending or passing segment
  ⟨leda determine upper bundle item sit_succ⟩
  ⟨leda hash upper intersection event⟩
  ⟨leda walk ingoing bundle and trigger graph updates⟩
  ⟨leda reverse continuing bundle edges⟩
} // if (sit != nil)
```

As $sit == nil$ we do not know if a segment stored in *YS* does contain *p_sweep*. We have to query *YS* with a trivial segment (*p_sweep*,*p_sweep*) to find out. Two results are possible. Either a segment referenced hereafter by *sit* constains the event point or we determine the two segments above and below *p_sweep* in *sit_pred* and *sit_succ*.

⟨*leda check p_sweep in YS*⟩≡

```
{
  Segment_2 s_sweep = K.construct_segment(p_sweep,p_sweep);
  seg_pair sp(s_sweep,ITERATOR());
  sit_succ = YS.locate( &sp );
  if ( sit_succ != YS.max_item() &&
       orientation(sit_succ,p_sweep) == 0 )
    sit = sit_succ;
  else  {
    sit_pred = YS.pred(sit_succ);
    sit_pred_succ = sit_succ;
  }
}
```

We first walk up as long as the event is contained in the segment referenced via *sit*.

⟨*leda determine upper bundle item sit_succ*⟩≡

```
while ( YS.inf(sit) == event ||
        YS.inf(sit) == YS.succ(sit) ) // overlapping
  sit = YS.succ(sit);
sit_succ = YS.succ(sit);
seq_item sit_last = sit;
```

We hash the upper event according to Invariant 6.

⟨*leda hash upper intersection event*⟩≡

```
seq_item xit = YS.inf(sit_last);
if (xit) {
  ISegment s1 = YS.key(sit_last);
  ISegment s2 = YS.key(sit_succ);
  IEvent(s1,s2) = xit;
}
```

We walk the ingoing bundle down again and trigger the edge closing calls for all items in the bundle (except overlapping segments). Note that after this code chunk we have: (i) the bundle is empty if $succ(sit\_pred) == sit\_first == sit\_succ$, or (ii) the bundle is not empty if $sit\_first\ != sit\_succ$.

The actions on the bundle are easy to specify. We have to glue one edge per segment to the newly created node except when two segments overlap. Note that the walk top-down over the bundle implies the order-preserving embedding of the graph. Note also how we pass the messages about the segments supporting the event via the corresponding methods of the output object.

⟨*leda walk ingoing bundle and trigger graph updates*⟩≡

```
bool overlapping;
do {
  ISegment s = YS.key(sit);
```

```
    seq_item sit_next = YS.pred(sit);
    overlapping = (YS.inf(sit_next) == sit);
    Halfedge_handle e = Edge_of[sit];
    if ( !overlapping ) {
      GO.link_as_target_and_append(v,e);
    }
    GO.supporting_segment(e,original(s));
    if ( target(s) == p_sweep ) { // ending segment
      if ( overlapping ) YS.change_inf(sit_next,YS.inf(sit));
      YS.del_item(sit);
      GO.ending_segment(v,original(s));
    } else {  // passing segment
      if ( YS.inf(sit) != YS.succ(sit) )
        YS.change_inf(sit, seq_item(0));
      GO.passing_segment(v,original(s));
    }
    sit = sit_next;
  }
  while ( YS.inf(sit) == event || overlapping ||
          YS.inf(sit) == YS.succ(sit) );
  sit_pred = sit;
  sit_first = sit_pred_succ = YS.succ(sit_pred); // first item of bundle
```

We have to ensure that segments that continue through the event point have a reversed order within *YS* when *p_sweep* has been passed. This ensures the correct order of *YS* with respect to Invariant 2. Some complication stems from overlapping segments. Their order based on identity may not be changed.

⟨*leda reverse continuing bundle edges*⟩≡

```
  while ( sit != sit_succ ) {
    seq_item sub_first = sit;
    seq_item sub_last  = sub_first;
    while (YS.inf(sub_last) == YS.succ(sub_last))
      sub_last = YS.succ(sub_last);
    if (sub_last != sub_first)
      YS.reverse_items(sub_first, sub_last);
    sit = YS.succ(sub_first);
  }
  // reverse the entire bundle
  if (sit_first != sit_succ)
    YS.reverse_items(YS.succ(sit_pred),YS.pred(sit_succ));
```

For the new node *v* we pass some information to the output structure. We post the halfedge below, we post all trivial input segments supporting the node. We obtain that information from the hash structure *Isos_of* filled during the initialization phase.

⟨*leda completing additional information of the new vertex*⟩≡

```
  assert(sit_pred);
  GO.halfedge_below(v,Edge_of[sit_pred]);
  if ( Isos_of[event] != 0 ) {
```

```
    const IsoList& IL = *(Isos_of[event]);
    slist_item iso_it;
    for (iso_it = IL.first(); iso_it; iso_it=IL.succ(iso_it) )
      GO.trivial_segment(v,IL[iso_it] );
    delete (Isos_of[event]); // clean up the list
  }
```

We insert all segments starting at *p_sweep* into *YS* and create the links within *YS* to mark items with overlapping segments.

⟨*leda inserting new segments starting at nodes*⟩≡

```
  ISegment next_seg;
  pq_item next_it = SQ.find_min();
  while ( next_it &&
          (next_seg = SQ.inf(next_it), p_sweep == source(next_seg)) ) {
    seq_item s_sit = YS.locate_succ(next_seg);
    seq_item p_sit = YS.pred(s_sit);

    if ( YS.max_item() != s_sit &&
         orientation(s_sit, source(next_seg) ) == 0 &&
         orientation(s_sit, target(next_seg) ) == 0 )
      sit = YS.insert_at(s_sit, next_seg, s_sit);
    else
      sit = YS.insert_at(s_sit, next_seg, seq_item(nil));
    assert(YS.succ(sit)==s_sit);

    if ( YS.min_item() != p_sit &&
         orientation(p_sit, source(next_seg) ) == 0 &&
         orientation(p_sit, target(next_seg) ) == 0 )
      YS.change_inf(p_sit, sit);
    assert(YS.succ(p_sit)==sit);

    XS.insert(target(next_seg), sit);
    GO.starting_segment(v,original(next_seg));

    // delete minimum and assign new minimum to next_seg
    SQ.del_min();
    next_it = SQ.find_min();
  }
```

In contrast to the original LEDA segment intersection algorithm *SWEEP_SEGMENTS* we create "semi-open" edges starting at the *event* node and supported by the input segment. The iteration again ensures the correct order-preserving embedding at the currently handled node v.

⟨*leda inserting new segments starting at nodes*⟩+≡

```
  for( seq_item sitl = YS.pred(sit_succ); sitl != sit_pred;
       sitl = YS.pred(sitl) ) {
    if ( YS.inf(sitl) != YS.succ(sitl) ) { // non-overlapping
      Edge_of[sitl] = GO.new_halfedge_pair_at_source(v);
    } else {
      Edge_of[sitl] = Edge_of[ YS.succ(sitl) ];
    }
  }
  sit_first = YS.succ(sit_pred);
```

Depending on the outgoing bundle we determine possible intersections between new neighbors; if *sit_pred* is no longer adjacent to its former successor we change its intersection event to 0. Note that the following chunk finishes Invariant 1 with the help of the method *compute_intersection*( ).

⟨*leda enforcing the invariants for YS*⟩≡
```
assert(sit_pred); assert(sit_pred_succ);
seq_item xit = YS.inf(sit_pred);
if ( xit ) {
  ISegment s1 = YS.key(sit_pred);
  ISegment s2 = YS.key(sit_pred_succ);
  IEvent(s1,s2) = xit;
  YS.change_inf(sit_pred, seq_item(0));
}
compute_intersection(sit_pred);
sit = YS.pred(sit_succ);
if (sit != sit_pred)
  compute_intersection(sit);
```

Initialization

We realize the propositions of the invariants at the beginning in our sweep initialization phase. We insert all segment endpoints into *XS*, insert sentinels into *YS*, and exploit the fact that insert operations into the X-structure leave previously inserted points unchanged to achieve that any pair of endpoints p and q with $p == q$ are identical (if the geometric point type supports this). Degenerate segments are stored in a list associated to their events. The knowledge about their existence is transferred to the corresponding output object as soon as it is constructed.

⟨*leda initialization of the sweep*⟩≡
```
void initialize_structures()
{
  ITERATOR it_s;
  for ( it_s=its; it_s != ite; ++it_s ) {
    Segment_2 s = *it_s;
    seq_item it1 = XS.insert( K.source(s), seq_item(nil));
    seq_item it2 = XS.insert( K.target(s), seq_item(nil));
    if (it1 == it2) {
      if ( Isos_of[it1] == 0 ) Isos_of[it1] = new IsoList;
      Isos_of[it1]->push(it_s);
      continue;  // ignore zero-length segments in SQ/YS
    }
    Point_2 p = XS.key(it1);
    Point_2 q = XS.key(it2);
    Segment_2 s1;
    if ( K.compare_xy(p,q) < 0 )
      s1 = K.construct_segment(p,q);
    else
      s1 = K.construct_segment(q,p);
    Internal.append(seg_pair(s1,it_s));
```

```
      SQ.insert(K.source(s1),&Internal[Internal.last()]);
    }
    // insert a lower and an upper sentinel segment
    YS.insert(&sl,seq_item(nil));
    YS.insert(&sh,seq_item(nil));
  }
```

Note the invariants of the sweep loop. The *event* has to be set before the event action.

⟨*leda iteration control*⟩≡

```
  bool event_exists()
  {
    if (!XS.empty()) {
      // event is set at end of loop and in init
      event = XS.min();
      p_sweep = XS.key(event);
      return true;
    }
    return false;
  }
  void procede_to_next_event()
  { XS.del_item(event); }
```

The structure is finished as soon as all events have been treated. As we always created edges in pairs and respected the adjacency list order we have no completion phase as in [MN99, chapter 10].

⟨*leda postprocessing of the sweep*⟩≡

```
  void complete_structures() {}
  void check_invariants() {}
  void check_final() {}
```

⟨*leda debugging routines*⟩≡

```
  #ifdef _DEBUG
  #define PIS(s) (s->first())
  #endif
```

Runtime Analysis

We shortly give a runtime analysis of the above implementation. Assume $U(S)$ to be the embedded (undirected) graph created by a proper[3] instantiation of our sweep framework. Let n be the number of input segments of the input set S, n_v be the number of nodes of $U(S)$, n_e be the number of (undirected) edges of $U(S)$. In the presence of overlapping segments let $\bar{n}_e := \sum_{e \text{ edge of } U(S)} s_e$ where s_e is the number of segments in S supporting edge e of $U(S)$. Note that during the sweep the whole graph is constructed and explored. Then obviously all graph related operations like creation, and support messaging take time $O(n_v + \bar{n}_e)$.

[3] The output methods have to be constant time operations of the correct semantics.

The sweep initialization takes time $O(n \log n)$. There are n_v events and at each event the sweep status structures are manipulated a constant number of times. Pass again through the event handling routine. All segments are inserted into *YS* and deleted from *YS* once (during the whole sweep) and therefore that cost adds up to $O(n \log |YS|)$. The removal from *SQ* adds up $O(n \log n)$. Finally all events are inserted and deleted once and this takes $O(|XS| \log |XS|)$. When subsequences of *YS* are explored and swapped, this cost can again be dedicated to the exploration of the graph and is therefore subsumed in the $O(n_v + \bar{n}_e)$ from above.

We have to bound the size of *XS* and *YS*. Natural bounds are $|XS| = O(n_v)$ (the number of nodes) and $|YS| = O(n)$ (the number of segments). Therefore accumulating all of the above we obtain $O(n_v + \bar{n}_e + n \log n + n_v \log(n + n_v)) = O(n_v + \bar{n}_e + (n + n_v) \log(n + n_v))$. Note that $\bar{n}_e$ and n_v can be quadratic in n.

Lemma 2.1.3: Assume that n, n_v, $\bar{n}_e$ are defined as above then the runtime of the sweep algorithm is

$$O(n_v + \bar{n}_e + (n + n_v) \log(n + n_v)).$$

2.1.4 The STL traits model

We present an alternative module purely based on STL data structures. As the STL has no hashing support we omit all the optimization used in the LEDA approach. Additionally we cannot use any links from events into the y-structure. As STL maps have no order manipulating operations like *reverse_items* of LEDA *sortseq*s we have to delete and reinsert ranges of segments at events. Therefore no shortcuts can be stored to minimize search operations within *YS*.

⟨*stl segment overlay model*⟩≡
```
#include <list>
#include <map>
#include <string>
#include <strstream>

namespace CGAL {

template <typename IT, typename PMDEC, typename GEOM>
class stl_seg_overlay_traits {
public:
  ⟨stl introducing the types from the traits⟩
  ⟨stl internal segment type⟩

  // types interfacing the generic sweep frame
  ITERATOR its, ite;
  OUTPUT&  GO;
  const GEOMETRY& K;

  ⟨stl order types for segments and points⟩
  ⟨stl sweep data structures⟩
  ⟨stl helping operations⟩
  ⟨stl operation for keeping the intersection invariant⟩
  ⟨stl initialization of the sweep⟩
  ⟨stl iteration control⟩
  ⟨stl handling the event⟩
  ⟨stl postprocessing of the sweep⟩
}; // stl_seg_overlay_traits

} // namespace CGAL
```

The following types are introduced by the traits classes.

⟨*stl introducing the types from the traits*⟩≡
```
typedef IT                               ITERATOR;
typedef std::pair<IT,IT>                 INPUT;
typedef PMDEC                            OUTPUT;
typedef typename PMDEC::Vertex_handle    Vertex_handle;
typedef typename PMDEC::Halfedge_handle  Halfedge_handle;
typedef GEOM                             GEOMETRY;
typedef typename GEOMETRY::Point_2       Point_2;
typedef typename GEOMETRY::Segment_2     Segment_2;
```

Similar to the LEDA implementation we obtain internal segments by pointers to *pair<Segment,ITERATOR>*.

⟨*stl internal segment type*⟩≡
```
typedef std::pair<Segment_2,ITERATOR>     seg_pair;
typedef seg_pair*                         ISegment;
typedef std::list<seg_pair>               IList;
typedef typename IList::const_iterator    ilist_iterator;
```

The STL uses strict order type objects. The predicate is implemented in analogy to *cmp_segs_at_sweepline*.

⟨*stl order types for segments and points*⟩≡
```
class lt_segs_at_sweepline
{ const Point_2& p;
  ISegment s_bottom, s_top; // sentinel segments
  const GEOMETRY& K;
public:
 lt_segs_at_sweepline(const Point_2& pi,
   ISegment s1, ISegment s2, const GEOMETRY& k) :
   p(pi), s_bottom(s1), s_top(s2), K(k) {}
 lt_segs_at_sweepline(const lt_segs_at_sweepline& lt) :
   p(lt.p), s_bottom(lt.s_bottom), s_top(lt.s_top), K(lt.K) {}
 bool operator()(const ISegment& is1, const ISegment& is2) const
 {
   if ( is2 == s_top || is1 == s_bottom ) return true;
   if ( is1 == s_top || is2 == s_bottom ) return false;
   if ( is1 == is2 ) return false;
   // Precondition: p is contained in s1 or s2.
   const Segment_2& s1 = is1->first;
   const Segment_2& s2 = is2->first;
   int s = 0;
   if ( K.orientation(s1,p) == 0 )
     s =   K.orientation(s2,p);
   else if ( K.orientation(s2,p) == 0 )
     s = - K.orientation(s1,p);
   else ASSERT(0,"compare error in sweep.");
   if ( s || K.is_degenerate(s1) || K.is_degenerate(s2) )
     return ( s < 0 );
```

```
    s = K.orientation(s2,K.target(s1));
    if (s==0) return ( is1 - is2 ) < 0;
    // overlapping segments are not equal
    return ( s < 0 );
  }
};
struct lt_pnts_xy
{ const GEOMETRY& K;
public:
 lt_pnts_xy(const GEOMETRY& k) : K(k) {}
 lt_pnts_xy(const lt_pnts_xy& lt) : K(lt.K) {}
 int operator()(const Point_2& p1, const Point_2& p2) const
 { return K.compare_xy(p1,p2) < 0; }
};
```

We need three main STL data structures. The x-structure *XS* is an ordered set based on the key type *Point_2*. The y-structure is a sorted sequence of *ISegments*. During the sweep we associate edges from the constructed output graph to these segments. Finally for each event point there is a possible sequence of input segments starting at the event. To maintain this sequence we use a STL multimap.

⟨*stl sweep data structures*⟩≡

```
typedef std::map<ISegment, Halfedge_handle, lt_segs_at_sweepline>
                                                      SweepStatus;
typedef typename SweepStatus::iterator                ss_iterator;
typedef typename SweepStatus::value_type              ss_pair;
typedef std::list<ITERATOR>                           IsoList;
typedef std::map<Point_2, IsoList*, lt_pnts_xy>       EventQueue;
typedef typename EventQueue::iterator                 event_iterator;
typedef typename EventQueue::value_type               event_pair;
typedef std::multimap<Point_2, ISegment, lt_pnts_xy>  SegQueue;
typedef typename SegQueue::iterator                   seg_iterator;
typedef typename SegQueue::value_type                 ps_pair;
event_iterator    event;
Point_2           p_sweep;
EventQueue        XS;
seg_pair          sl,sh;
SweepStatus       YS;
SegQueue          SQ;
IList             Internal;
stl_seg_overlay_traits(const INPUT& in, OUTPUT& G,
  const GEOMETRY& k) :
  its(in.first), ite(in.second), GO(G), K(k),
  XS(lt_pnts_xy(K)), YS(lt_segs_at_sweepline(p_sweep,&sl,&sh,K)),
  SQ(lt_pnts_xy(K)) {}
```

We have similar helpers *source*(s), *target*(s), *original*(s), *orientation*( ), *collinear*( ) as in the LEDA framework. We don't list them again.

⟨*stl helping operations*⟩≡

```
std::string dump_structures() const
{
  std::ostrstream out;
  out << "EventQueue:\n";
  typename EventQueue::const_iterator sit1;
  for(sit1 = XS.begin(); sit1 != XS.end(); ++sit1)
    out << "  " << sit1->first << std::endl;
  out << "SegQueue:\n";
  typename SegQueue::const_iterator sit2;
  for(sit2 = SQ.begin(); sit2 != SQ.end(); ++sit2)
    out << "  " << sit2->first << " " << sit2->second
        << " " << sit2->first << std::endl;
  out << "SweepStatus:\n";
  typename SweepStatus::const_iterator sit3;
  for( sit3 = YS.begin(); sit3 != YS.end(); ++sit3 )
    out << sit3->first << " " << &*(sit3->second) << std::endl;
  std::string res(out.str()); out.freeze(0); return res;
}
Point_2 source(ISegment is) const
{ return K.source(is->first); }
Point_2 target(ISegment is) const
{ return K.target(is->first); }
ITERATOR original(ISegment s) const
{ return s->second; }
int orientation(ss_iterator sit, const Point_2& p) const
{ return K.orientation(sit->first->first,p); }
bool collinear(ss_iterator sit1, ss_iterator sit2) const
{ Point_2 ps = source(sit2->first), pt = target(sit2->first);
  return ( orientation(sit1,ps)==0 &&
           orientation(sit1,pt)==0 );
}
```

Again there's a *compute_intersection* operations. Only the hash optimization is left out.

⟨*stl operation for keeping the intersection invariant*⟩≡

```
void compute_intersection(ss_iterator sit0)
{
  // Given an item |sit0| in the Y-structure compute the point of
  // intersection with its successor and (if existing) insert it into
  // the event queue and do all necessary updates.
  ss_iterator sit1 = sit0; ++sit1;
  if ( sit0 == YS.begin() || sit1 == --YS.end() ) return;
  const Segment_2& s0 = sit0->first->first;
  const Segment_2& s1 = sit1->first->first;
  int or0 = K.orientation(s0,K.target(s1));
  int or1 = K.orientation(s1,K.target(s0));
  if ( or0 <= 0 && or1 >= 0  ) {
    Point_2 q = K.intersection(s0,s1);
```

```
    XS.insert(event_pair(q,0)); // only done if none existed!!!
  }
}
```

Event handling in this version is similar to the LEDA implementation, only that most algorithmic decisions are now based on geometric predicate evaluation.

⟨*stl handling the event*⟩≡

```
void process_event()
{
  Vertex_handle v = GO.new_vertex(p_sweep);
  ss_iterator sit_succ, sit_pred, sit_first, sit;
  Segment_2 s_sweep = K.construct_segment(p_sweep,p_sweep);
  seg_pair sp(s_sweep,ITERATOR());
  sit_succ = YS.upper_bound(&sp);
  sit = sit_succ; --sit;
  ⟨stl handling ending and passing segments⟩
  ⟨stl completing additional information of the new vertex⟩
  ⟨stl inserting new segments starting at nodes⟩
  ⟨stl enforcing the invariants for YS⟩
}
```

We cannot reverse the bundle passing *p_sweep*, but have to delete all and reinsert them.

⟨*stl handling ending and passing segments*⟩≡

```
  /* |sit| is determined by upper bounding the search for the
     segment (p_sweep,p_sweep) and taking its predecessor.
     if the segment associated to |sit| contains |p_sweep| then
     there's a bundle of segments containing |p_sweep|.
     We compute the successor (|sit_succ)|) and
     predecessor (|sit_pred|) items. */
   if ( sit == YS.begin() || orientation(sit,p_sweep) != 0 ) {
    sit_pred = sit;
    sit = YS.end();
  }
  /* If no segments contain p_sweep then sit_pred and sit_succ are
     correctly set after the above locate operation, if a segment
     contains p_sweep sit_pred and sit_succ are set below when
     determining the bundle.*/
  if ( sit != YS.end() ) { // sit->first is ending or passing segment
    // Determine upper bundle item:
    /* Walk down until |sit_pred|, close edges for all segments
       in the bundle, delete all segments in the bundle, but
       reinsert the continuing ones */
    std::list<ISegment> L_tmp;
    bool overlapping;
    do {
      ISegment s = sit->first;
      ss_iterator sit_next(sit); --sit_next;
```

```
    overlapping = (sit_next != YS.begin()) && collinear(sit,sit_next);
    Halfedge_handle e = sit->second;
    if ( overlapping ) {
    } else {
      GO.link_as_target_and_append(v,e);
      /* in this case we close the output edge |e| associated to
         |sit| by linking |v| as its target and by appending the
         twin edge to |v|'s adjacency list. */
    }
    GO.supporting_segment(e,original(s));
    if ( target(s) == p_sweep ) {
      GO.ending_segment(v,original(s));
    } else { // passing segment, take care of the node here!
      L_tmp.push_back(s);
      GO.passing_segment(v,original(s));
     }
    sit = sit_next;
  }
  while ( sit != YS.begin() && orientation(sit,p_sweep) == 0 );
  sit_pred = sit_first = sit;
  ++sit_first; // first item of the bundle

  /* Interfaceproposition for next chunk:
     - succ(sit_pred) == sit_first == sit_succ
     - bundle not empty: sit_first != sit_succ
  */
  // delete and reinsert the continuing bundle
  YS.erase(sit_first,sit_succ);
  typename std::list<ISegment>::const_iterator lit;
  for ( lit = L_tmp.begin(); lit != L_tmp.end(); ++lit ) {
    YS.insert(sit_pred,ss_pair(*lit,Halfedge_handle()));
  }
} // if (sit != ss_iterator() )
```

Node data association is similar again.

⟨*stl completing additional information of the new vertex*⟩≡

```
assert( sit_pred != YS.end() );
GO.halfedge_below(v,sit_pred->second);
if ( event->second != 0 ) {
  const IsoList& IL = *(event->second);
  typename IsoList::const_iterator iso_it;
  for (iso_it = IL.begin(); iso_it != IL.end(); ++iso_it)
    GO.trivial_segment(v,*iso_it);
  delete (event->second);
}
```

The insertion phase is even simpler.

⟨*stl inserting new segments starting at nodes*⟩≡
```
ISegment next_seg;
seg_iterator next_it = SQ.begin();
while ( next_it != SQ.end() &&
        ( next_seg = next_it->second, p_sweep == source(next_seg)) ) {
  YS.insert(ss_pair(next_seg,Halfedge_handle()));
  GO.starting_segment(v,original(next_seg));
  // delete minimum and assign new minimum to next_seg
  SQ.erase(SQ.begin());
  next_it = SQ.begin();
}
// we insert new edge stubs, non-linked at target
ss_iterator sit_curr = sit_succ, sit_prev = sit_succ;
for( --sit_curr; sit_curr != sit_pred;
     sit_prev = sit_curr, --sit_curr ) {
  if ( sit_curr != YS.begin() && sit_prev != --YS.end() &&
       collinear(sit_curr,sit_prev) ) // overlapping
    sit_curr->second = sit_prev->second;
  else {
    sit_curr->second = GO.new_halfedge_pair_at_source(v);
  }
}
sit_first = sit_prev;
```

For the intersection invariant we always calculate.

⟨*stl enforcing the invariants for YS*⟩≡
```
// compute possible intersections between |sit_pred| and its
// successor and |sit_succ| and its predecessor
compute_intersection(sit_pred);
sit = sit_succ; --sit;
if (sit != sit_pred)
  compute_intersection(sit);
```

We realize the propositions of the invariants at the beginning in our sweep initialization phase. We insert the endpoints of the segments into *XS* and store an internal segment for each non trivial segment. We store a geometric object and the input interator in our *Internal* list. Then we store the internal segment associated to its starting event in our segment queue *SQ*. A trivial segment is associated to the corresponding event in the x-structure. To finish the initialization we have to insert two sentinel segments into the y-structure which avoids boundary checks in any *YS* search. Also *SQ* obtains a trailing sentinel which is never reached during our iteration over *XS*.

⟨*stl initialization of the sweep*⟩≡
```
void initialize_structures()
{
  /* INITIALIZATION
     - insert all vertices into the x-structure
     - insert sentinels into y-structure
     - exploit the fact that insert operations into the x-structure
       leave previously inserted points unchanged to achieve that
```

```
      any pair of endpoints $p$ and $q$ with |p == q| are identical
  */

  ITERATOR it_s;
  for ( it_s=its; it_s != ite; ++it_s ) {
    const Segment_2& s = *it_s;
    event_iterator it1 = (XS.insert(event_pair(K.source(s),0))).first;
    event_iterator it2 = (XS.insert(event_pair(K.target(s),0))).first;
    // note that the STL only inserts if key is not yet in XS
    if (it1 == it2) {
      if ( it1->second == 0 ) it1->second = new IsoList;
      it1->second->push_front(it_s);
      continue;  // ignore zero-length segments regarding YS
    }
    Point_2 p = it1->first;
    Point_2 q = it2->first;
    Segment_2 s1;
    if ( K.compare_xy(p,q) < 0 )
      s1 = K.construct_segment(p,q);
    else
      s1 = K.construct_segment(q,p);
    Internal.push_back(seg_pair(s1,it_s));
    SQ.insert(ps_pair(K.source(s1),&Internal.back()));
  }
  // insert a lower and an upper sentinel segment to avoid special
  // cases when traversing the Y-structure
  YS.insert(ss_pair(&sl,Halfedge_handle()));
  YS.insert(ss_pair(&sh,Halfedge_handle()));
}
```

Note the invariants of the sweep loop. The *event* has to be set before the event action.

⟨*stl iteration control*⟩≡

```
bool event_exists()
{
  if (!XS.empty()) {
    // event is set at end of loop and in init
    event = XS.begin();
    p_sweep = event->first;
    return true;
  }
  return false;
}
void procede_to_next_event()
{ XS.erase(event); }
```

⟨*stl postprocessing of the sweep*⟩≡

```
void complete_structures() {}
void check_invariants() {}
void check_final() {}
```

2.2 Test Cases - Models for Geometry Kernels and Output

The following models fulfill the requirements of the traits class concept used in the above segment overlay framework (LEDA *or* STL). See the appendix for the concept *SegmentOverlayOutput*.

2.2.1 The LEDA segment sweep traits model

As the algorithm originally was hardwired to the LEDA types the adaptation is not too hard. Note however that we have to integrate some information storage into the OUTPUT structure. The OUTPUT is thus a pair: a parameterized graph plus a node map associating edges to nodes.

⟨*leda_overlay_traits.h*⟩≡

```
#ifndef LEDA_OVERLAY_TRAITS_H
#define LEDA_OVERLAY_TRAITS_H

#include <LEDA/rat_kernel.h>
#include <LEDA/graph.h>
#include <LEDA/slist.h>

template <typename T>
ostream& operator<<(ostream& os, const leda_slist<T>& s)
{ return os; }
template <typename T>
istream& operator>>(istream& is, leda_slist<T>& s)
{ return is; }

⟨LEDA OUTPUT model⟩
⟨LEDA GEOMETRY model⟩

#endif // LEDA_OVERLAY_TRAITS_H
```

⟨*LEDA OUTPUT model*⟩≡

```
template <typename ITERATOR>
class leda_graph_decorator {
public:
  typedef leda_node Vertex_handle;
  typedef leda_edge Halfedge_handle;
  typedef leda_rat_point Point_2;

  typedef GRAPH< Point_2, leda_slist<ITERATOR> > Graph;
  typedef leda_node_map<Halfedge_handle>         Below_map;

  Graph&      G;
  Below_map&  M;

leda_graph_decorator(Graph& Gi, Below_map& Mi) : G(Gi), M(Mi) {}

Vertex_handle new_vertex(const Point_2& p)
{ return G.new_node(p); }

void link_as_target_and_append(Vertex_handle v, Halfedge_handle e)
{ Halfedge_handle erev = G.reversal(e);
  G.move_edge(e,G.cyclic_adj_pred(e,G.source(e)),v);
  G.move_edge(erev,v,G.target(erev));
}

Halfedge_handle new_halfedge_pair_at_source(Vertex_handle v)
{ Halfedge_handle e_res,e_rev, e_first = G.first_adj_edge(v);
```

```
    if ( e_first == nil ) {
      e_res = G.new_edge(v,v);
      e_rev = G.new_edge(v,v);
    } else {
      e_res = G.new_edge(e_first,v,LEDA::before);
      e_rev = G.new_edge(e_first,v,LEDA::before);
    }
    G.set_reversal(e_res,e_rev);
    return e_res;
  }

  void supporting_segment(Halfedge_handle e, ITERATOR it)
  { G[e].append(it); }

  void trivial_segment(Vertex_handle v, ITERATOR it)
  { }

  void halfedge_below(Vertex_handle v, Halfedge_handle e)
  { M[v] = e; }

  void starting_segment(Vertex_handle v, ITERATOR it)
  {}

  void passing_segment(Vertex_handle v, ITERATOR it)
  {}

  void ending_segment(Vertex_handle v, ITERATOR it)
  {}

  }; // leda_graph_decorator
```

⟨*LEDA GEOMETRY model*⟩≡

```
  class leda_geometry {
  public:
    typedef leda_rat_point   Point_2;
    typedef leda_rat_segment Segment_2;

  leda_geometry() {}

    Point_2 source(const Segment_2& s) const
    { return s.source(); }

    Point_2 target(const Segment_2& s) const
    { return s.target(); }

    Segment_2 construct_segment(const Point_2& p, const Point_2& q) const
    { return Segment_2(p,q); }

    int orientation(const Segment_2& s, const Point_2& p) const
    { return ::orientation(s,p); }

    bool is_degenerate(const Segment_2& s) const
    { return s.is_trivial(); }

    int compare_xy(const Point_2& p1, const Point_2& p2) const
    { return Point_2::cmp_xy(p1,p2); }

    Point_2 intersection(const Segment_2& s1, const Segment_2& s2) const
    { Point_2 p;
      s1.intersection_of_lines(s2,p);
      return p;
    }
  }; // leda_geometry
```

2.2.2 The visualization model

We visualize the sweep by means of our generic sweep observer, which obtains messages at four events: after initialization, just before a sweep event, just after a sweep event, and after completion. We visualize in a LEDA window. After the intialization we draw all input segments. Before the event we place the sweep line at the event point. After the event we draw all created edges ending at the event. The following class *leda_visualization* can be plugged into the sweep observer class. For the model see *GenericSweepVisualization* in the appendix (Section 11.1.7).

⟨*leda_overlay_visualization.h*⟩≡

```
#ifndef LEDA_OVERLAY_VISUALIZATION_H
#define LEDA_OVERLAY_VISUALIZATION_H
#include <LEDA/rat_window.h>
template <class GENSWEEPTRAITS>
class leda_visualization {
leda_window W;
public:
typedef leda_window VDEVICE;
typedef typename GENSWEEPTRAITS::Halfedge_handle    Halfedge_handle;
typedef typename GENSWEEPTRAITS::Vertex_handle      Vertex_handle;
typedef typename GENSWEEPTRAITS::OUTPUT    Graph;
typedef typename GENSWEEPTRAITS::Segment_2 Segment_2;
typedef typename GENSWEEPTRAITS::Point_2   Point_2;
typedef typename GENSWEEPTRAITS::ISegment  ISegment;
leda_visualization() : W()
{ W.set_show_coordinates(true);
  W.init(-50,50,-50,1);
  W.set_node_width(3);
  W.set_line_width(2);
  W.display();
}
void draw(const Point_2& p, leda_color c)
{ W.draw_filled_node(p.xcoordD(),p.ycoordD(),c); }
void draw(const Segment_2& s, leda_color c)
{ W.draw_segment(s.xcoord1D(),s.ycoord1D(),
                 s.xcoord2D(),s.ycoord2D(),c);}
void post_init_animation(GENSWEEPTRAITS& gpst)
{ typename GENSWEEPTRAITS::ITERATOR it;
  for (it = gpst.its; it != gpst.ite; ++it )
    if ( (*it).is_trivial() )
      draw((*it).source(),leda_blue);
    else
      draw(*it,leda_blue);
  W.read_mouse();
}
void draw_sl(const leda_point& p)
{ leda_drawing_mode mold = W.set_mode(leda_xor_mode);
  W.draw_filled_node(p,leda_red);
  W.draw_vline(p.xcoord(),leda_red);
  W.set_mode(mold);
```

```
}
void pre_event_animation(GENSWEEPTRAITS& gpst)
{ draw_sl(gpst.p_sweep.to_point()); }
void post_event_animation(GENSWEEPTRAITS& gpst)
{ typename GENSWEEPTRAITS::OUTPUT::Graph& G(gpst.GO.G);
  Halfedge_handle e;
  Vertex_handle v = G.last_node();
  forall_out_edges(e, v) {
    if ( source(e)!=target(e) ) {
      draw(leda_rat_segment(G[source(e)],G[target(e)]),leda_black);
    }
  }
  draw(gpst.p_sweep,leda_black);
  W.read_mouse();
  draw_sl(gpst.p_sweep.to_point());
}
void post_completion_animation(GENSWEEPTRAITS& gpst)
{ typename GENSWEEPTRAITS::OUTPUT::Graph& G(gpst.GO.G);
  Halfedge_handle e;
  forall_edges(e,G) {
    draw(leda_rat_segment(G[source(e)],G[target(e)]),leda_black);
  }
  Vertex_handle v;
  forall_nodes(v,G) {
    draw(G[v],leda_black);
  }
}
VDEVICE& device() { return W; }
};
#endif //LEDA_OVERLAY_VISUALIZATION_H
```

2.2.3 Testing the generic LEDA sweep

⟨*leda_segment_overlay-test.c*⟩≡

```
#include <LEDA/rat_window.h>
#include "Segment_overlay_traits.h"
#include "leda_overlay_traits.h"
#include "leda_overlay_visualization.h"
#include <LEDA/stream.h>
#include <LEDA/param_handler.h>
using namespace CGAL;
typedef leda_list<leda_rat_segment>::iterator Seg_iterator;
typedef CGAL::Segment_overlay_traits< Seg_iterator,
          leda_graph_decorator<Seg_iterator>, leda_geometry>
        leda_seg_sweep_traits;
typedef leda_visualization<leda_seg_sweep_traits>   leda_seg_vis;
typedef CGAL::generic_sweep<leda_seg_sweep_traits>  leda_seg_sweep;
typedef CGAL::sweep_observer<leda_seg_sweep,leda_seg_vis> leda_seg_sweep_obs;
int main(int argc, char** argv)
```

```
{
  // SETDTHREAD(23);
  leda_param_handler P(argc,argv,".overlay");
  P.add_parameter("inputfile1:-i:string:e1");
  leda_param_handler::init_all();
  leda_string f1;
  P.get_parameter("-i",f1);
  leda_seg_sweep_obs Obs;
  leda_list<leda_rat_segment> L;
  leda_rat_segment s;
  ifstream if1(f1);
  if (if1) {
    if1 >> L;
    forall (s,L)
     if ( s.is_trivial() ) Obs.device() << s.source().to_point();
     else  Obs.device() << s.to_segment();
  }
  while (Obs.device()>>s) {
    L.append(s);
  }
  ofstream of("input.log");
  forall(s,L) of << s;
  of.close();
  cout << "Starting " << CGAL::sweepversion << endl;
  leda_seg_sweep_traits::OUTPUT::Graph      G;
  leda_seg_sweep_traits::OUTPUT::Below_map  E_below(G);
  leda_seg_sweep_traits::OUTPUT O(G,E_below);
  leda_seg_sweep SI(leda_seg_sweep_traits::INPUT(L.begin(),L.end()),O);
  Obs.attach(SI);
  SI.sweep();
  cout << "Edges and Segments:\n";
  leda_edge e;
  forall_edges(e,G) {
    cout <<"  ("<<G[source(e)].to_point()<<G[target(e)].to_point()<< ") ";
    leda_slist< leda_list<leda_rat_segment>::iterator >& SL(G[e]);
    slist_item it;
    for ( it = SL.first(); it; it=SL.succ(it) )
      cout << *(SL[it]) << " ";
    cout << endl;
  }
  cout << "\nNodes and Edges below\n";
  leda_node v;
  forall_nodes(v,G) {
    cout <<"  "<<G[v]<<" ";
    if ( E_below[v] ) {
      leda_edge e = E_below[v];
      cout << "("<<G[source(e)]<<G[target(e)]<< ")\n";
    } else
      cout << "nil\n";
  }
```

```
  Obs.device().read_mouse();
  return 0;
}
```

We want to compare the runtimes of the two instantiations of our generic segment overlay and additionally the original LEDA 4.1 code *SWEEP_SEGMENTS*. The inputs are randomly distributed segments in a square. The runtimes are in seconds on an Ultra 2 200 MHz.

#segs	LEDA classic	LEDA generic	STL generic
100	0.14	0.19	0.29
200	0.67	0.94	1.53
400	2.59	3.37	5.54
800	12.3	15.68	24.95
1600	55.42	70.9	103.1

⟨*leda_segment_overlay-rt.c*⟩≡

```
#define INCLUDEBOTH // to allow comparison
#include "Segment_overlay_traits.h"
#include "leda_overlay_traits.h"
#include <LEDA/param_handler.h>
#include <LEDA/random_rat_point.h>
#include <LEDA/plane_alg.h>

typedef leda_list<leda_rat_segment>::iterator Seg_iterator;
typedef CGAL::leda_seg_overlay_traits< Seg_iterator,
          leda_graph_decorator<Seg_iterator>,leda_geometry>
        leda_seg_sweep_traits;
typedef CGAL::generic_sweep<leda_seg_sweep_traits> leda_seg_sweep;

typedef CGAL::stl_seg_overlay_traits< Seg_iterator,
          leda_graph_decorator<Seg_iterator>,leda_geometry>
        stl_seg_sweep_traits;
typedef CGAL::generic_sweep<stl_seg_sweep_traits>  stl_seg_sweep;

template <typename T>
bool equal(const leda_list<T>& L1, const leda_list<T>& L2)
{ leda_list<T>::const_iterator it1,it2;
  for (it1 = L1.begin(), it2 = L2.begin();
       it1 != L1.end() && it2 != L2.end();
       ++it1, ++it2 )
    if ( *it1 != *it2 ) return false;
  return (it1 == L1.end()) && (it2 == L2.end());
}

#define OUTPUT(t) \
  if ( !t ) cerr <<" "<<#t<<" = "<<(t)<< endl;

int main(int argc, char** argv)
{
  //SETDTHREAD(23);
  int n = 100;
  if ( argc > 2 ) {
    cout << "usage: " << argv[0] << " #segments\n";
    return 1;
  }
```

```
  if ( argc == 2 ) {
    n = atoi(argv[1]);
  }
  cout << "\\begin{tabular}[t]{llll}\n";
  cout << "\\#segs & LEDA classic & LEDA generic & STL generic \\\\ \\hline\n";
for (int i = 1; i < 6; n*=2,++i ) {
  leda_list<leda_rat_point>   LP;
  leda_list<leda_rat_segment> L;
  leda_rat_segment s;
  random_points_in_square(2*n,100,LP);
  list_item pit1 = LP.first(),pit2 = (pit1 ? LP.succ(pit1) : 0);
  while ( pit1&&pit2 ) {
    L.append(leda_rat_segment(LP[pit1],LP[pit2]));
    pit1 = LP.succ(pit2);
    pit2 = (pit1 ? LP.succ(pit1) : 0);
  }
  ofstream of("input.log");
  of << L;
  of.close();
  leda_seg_sweep_traits::OUTPUT::Graph       G1,G2;
  leda_seg_sweep_traits::OUTPUT::Below_map   E1(G1),E2(G2);
  leda_seg_sweep_traits::OUTPUT O1(G1,E1),O2(G2,E2);
  GRAPH<leda_rat_point,leda_rat_segment> G0;
  float t0 = used_time();
  SWEEP_SEGMENTS(L,G0,true);
  t0 = used_time(t0);
  leda_seg_sweep SSW1(leda_seg_sweep_traits::INPUT(L.begin(),L.end()),O1);
  float t1 = used_time();
  SSW1.sweep();
  t1 = used_time(t1);
  stl_seg_sweep SSW2(leda_seg_sweep_traits::INPUT(L.begin(),L.end()),O2);
  float t2 = used_time();
  SSW2.sweep();
  t2 = used_time(t2);
  cout << n << " & " << t0 << " & " << t1 << " & " << t2 << "\\\\\n";
  leda_list<leda_rat_point> EP1,EP2;
  leda_node v;
  forall_nodes(v,G1) EP1.append(G1[v]);
  forall_nodes(v,G2) EP2.append(G2[v]);
  leda_list<leda_rat_segment> ES1,ES2;
  leda_edge e;
  forall_edges(e,G1)
    ES1.append(leda_rat_segment(G1[source(e)],G1[target(e)]));
  forall_edges(e,G2)
    ES2.append(leda_rat_segment(G2[source(e)],G2[target(e)]));
  OUTPUT(equal(EP1,EP2));
  OUTPUT(equal(ES1,ES2));
} // end of for
  cout << "\\end{tabular}\n";
  return 0;
}
```

2.2.4 A CGAL segment sweep traits model

We define a traits model for the segment overlay sweep which calculates the overlay of CGAL 2-dimensional kernel segments. We have

- input is an iterator range of CGAL segments
- output is a halfedge data structure decorator creating the overlay of the segments in the decorated HDS.

⟨*HDS_decorator.h*⟩≡

```
#ifndef CGAL_HDS_DECORATOR_H
#define CGAL_HDS_DECORATOR_H
#include <CGAL/basic.h>
#include <CGAL/PM_decorator_simple.h>
template <typename HDS, typename I>
class HDS_decorator : public CGAL::PM_decorator_simple<HDS> {
public:
  typedef CGAL::PM_decorator_simple<HDS> Base;
  typedef HDS                            Plane_map;
  typedef typename Base::Point_2         Point_2;
  typedef typename Base::Vertex_handle   Vertex_handle;
  typedef typename Base::Halfedge_handle Halfedge_handle;
  typedef I                              ITERATOR;
  HDS_decorator(HDS& H) : Base(H) {}
  Vertex_handle new_vertex(const Point_2& p) const
  { Vertex_handle v = Base::new_vertex();
    v->point() = p; return v; }
  void link_as_target_and_append(Vertex_handle v, Halfedge_handle e) const
  { Base::link_as_target_and_append(v,e); }
  Halfedge_handle new_halfedge_pair_at_source(Vertex_handle v) const
  { return Base::new_halfedge_pair_at_source(v,Base::BEFORE); }
  void supporting_segment(Halfedge_handle e, ITERATOR it) const {}
  void trivial_segment(Vertex_handle v, ITERATOR it) const {}
  void halfedge_below(Vertex_handle v, Halfedge_handle e) const {}
  void starting_segment(Vertex_handle v, ITERATOR it) const {}
  void passing_segment(Vertex_handle v, ITERATOR it) const {}
  void ending_segment(Vertex_handle v, ITERATOR it) const {}
}; // HDS_decorator
#endif // CGAL_HDS_DECORATOR_H
```

2.2.5 Affine geometry wrapper

⟨*Affine_geometry.h*⟩≡

```
#ifndef CGAL_AFFINE_GEOMETRY_H
#define CGAL_AFFINE_GEOMETRY_H
#include <CGAL/basic.h>
#include <CGAL/Point_2.h>
#include <CGAL/intersections.h>
```

```
#include <CGAL/squared_distance_2.h>
CGAL_BEGIN_NAMESPACE
template <typename R>
struct Affine_geometry : public R {
typedef typename R::Point_2     Point_2;
typedef typename R::Segment_2   Segment_2;
typedef typename R::Direction_2 Direction_2;
typedef typename R::Line_2      Line_2;
typedef typename R::RT          RT;
typedef typename R::FT          FT;
Affine_geometry() : R() {}
Affine_geometry(const Affine_geometry& Gi) : R(GI) {}
Point_2 source(const Segment_2& s) const
{ typename R::Construct_source_point_2 _source =
    construct_source_point_2_object();
  return _source(s); }
Point_2 target(const Segment_2& s) const
{ typename R::Construct_target_point_2 _target =
    construct_target_point_2_object();
  return _target(s); }
int orientation(const Point_2& p1, const Point_2& p2, const Point_2& p3) const
{ typename R::Orientation_2 _orientation =
    orientation_2_object();
  return static_cast<int> ( _orientation(p1,p2,p3) );
}
bool leftturn(const Point_2& p1, const Point_2& p2, const Point_2& p3) const
{ return (orientation(p1,p2,p3) > 0); }
bool is_degenerate(const Segment_2& s) const
{ typename R::Is_degenerate_2 _is_degenerate =
    is_degenerate_2_object();
  return _is_degenerate(s); }
int compare_xy(const Point_2& p1, const Point_2& p2) const
{ typename R::Compare_xy_2 _compare_xy =
    compare_xy_2_object();
  return static_cast<int>( _compare_xy(p1,p2) );
}
int compare_x(const Point_2& p1, const Point_2& p2) const
{ typename R::Compare_x_2 _compare_x =
    compare_x_2_object();
  return static_cast<int>( _compare_x(p1,p2) );
}
int compare_y(const Point_2& p1, const Point_2& p2) const
{ typename R::Compare_y_2 _compare_y =
    compare_y_2_object();
  return static_cast<int>( _compare_y(p1,p2) );
}
bool first_pair_closer_than_second(const Point_2& p1,
  const Point_2& p2, const Point_2& p3, const Point_2& p4) const
```

```
{ return ( squared_distance(p1,p2) < squared_distance(p3,p4) ); }
bool strictly_ordered_along_line(
  const Point_2& p1, const Point_2& p2, const Point_2& p3) const
{ typename R::Are_strictly_ordered_along_line_2 _ordered =
    are_strictly_ordered_along_line_2_object();
  return _ordered(p1,p2,p3);
}
Segment_2 construct_segment(const Point_2& p, const Point_2& q) const
{ typename R::Construct_segment_2 _segment =
    construct_segment_2_object();
  return _segment(p,q); }
int orientation(const Segment_2& s, const Point_2& p) const
{ typename R::Orientation_2 _orientation =
    orientation_2_object();
  return static_cast<int> ( _orientation(source(s),target(s),p) );
}
bool contains(const Segment_2& s, const Point_2& p) const
{ typename R::Has_on_2 _contains = has_on_2_object();
  return _contains(s,p);
}
Point_2 intersection(
  const Segment_2& s1, const Segment_2& s2) const
{ typename R::Intersect_2 _intersect =
    intersect_2_object();
  typename R::Construct_line_2 _line =
    construct_line_2_object();
  Point_2 p;
  CGAL::Object result =
    _intersect(_line(s1),_line(s2));
  if ( !CGAL::assign(p, result) )
  CGAL_assertion_msg(false,"intersection: no intersection.");
  return p;
}
// additional interface for constr triang:
Direction_2 construct_direction(
  const Point_2& p1, const Point_2& p2) const
{ typename R::Construct_direction_of_line_2 _direction =
    construct_direction_of_line_2_object();
  typename R::Construct_line_2 _line =
    construct_line_2_object();
  return _direction(_line(p1,p2)); }
bool strictly_ordered_ccw(const Direction_2& d1,
  const Direction_2& d2, const Direction_2& d3) const
{ return d2.counterclockwise_in_between (d1, d3); }
// additional interface for point location:
}; // Affine_geometry<R>
CGAL_END_NAMESPACE
#endif //CGAL_AFFINE_GEOMETRY_H
```

2.2.6 Testing the CGAL sweep

⟨*cgal_segment_overlay-test.C*⟩≡

```
#include <CGAL/basic.h>
#include <fstream>
#include <list>
#include <algorithm>
#include <CGAL/Cartesian.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/leda_integer.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/function_objects.h>
#include <CGAL/Join_input_iterator.h>
#include <CGAL/copy_n.h>
#include <CGAL/Halfedge_data_structure_default.h>
#include "Segment_overlay_traits.h"
#include "HDS_decorator.h"
#include "Affine_geometry.h"
// GEOMETRY:
typedef CGAL::Homogeneous<leda_integer> Kernel;
//typedef CGAL::Cartesian<double> Kernel;
typedef CGAL::Affine_geometry<Kernel>  Aff_kernel;
typedef Kernel::Point_2   Point_2;
typedef Kernel::Segment_2 Segment_2;
// INPUT:
typedef std::list<Segment_2>::const_iterator SIterator;
// OUTPUT:
typedef CGAL::Halfedge_data_structure_default<Point_2> HDS;
typedef HDS_decorator<HDS,SIterator> HDS_dec;
// SWEEP INSTANTIATION:
typedef CGAL::Segment_overlay_traits<SIterator,HDS_dec,Aff_kernel>
        CGAL_seg_sweep_traits;
typedef CGAL::generic_sweep<CGAL_seg_sweep_traits>  CGAL_seg_sweep;
typedef CGAL::Creator_uniform_2<int,Point_2>        Pnt_creator;
typedef CGAL::Creator_uniform_2<Point_2, Segment_2> Seg_creator;
typedef CGAL::Random_points_in_square_2<Point_2,Pnt_creator> Pnt_iterator;
typedef CGAL::Join_input_iterator_2< Pnt_iterator,Pnt_iterator,Seg_creator>
  Seg_iterator;
using std::cout;
using std::endl;
int main(int argc, char** argv)
{
  SETDTHREAD(23);
  CGAL::set_pretty_mode ( cout );
  std::list<Segment_2> L;
  std::list<Segment_2>::iterator lit;
  Segment_2 s;
  if (argc != 1) {
    std::ifstream if1( argv[1]);
    std::cout << "INPUT:\n";
    if ( if1)
      while (if1 >> s) {
```

```
      L.push_back(s);
      cout << s << ", ";
    }
  cout << endl;
}
Pnt_iterator p1(100),p2(100);
Seg_iterator sit(p1,p2);
for (int i = 0; i < 100; ++i,++sit)
  L.push_back(*sit);
// CGAL::copy_n(sit,100, std::back_inserter(L));
std::ofstream of("input.log");
for (lit = L.begin(); lit!= L.end(); ++lit)
  of << *lit;
of.close();
cout << "Starting " << CGAL::sweepversion << endl;
HDS G;
CGAL_seg_sweep_traits::OUTPUT P(G);
CGAL_seg_sweep SI(CGAL_seg_sweep::INPUT(L.begin(),L.end()),P);
SI.sweep();
cout << "\nVertices\n";
HDS::Halfedge_iterator eit;
HDS::Vertex_iterator   vit;
for( vit=G.vertices_begin(); vit != G.vertices_end();
     ++vit ) {
  cout <<"  "<<vit->point() << endl;
}
cout << "Edges:\n";
for( eit=G.halfedges_begin(); eit != G.halfedges_end();
     ++(++eit) ) {
  cout <<"  "<< eit->opposite()->vertex()->point() << "," <<
         eit->vertex()->point() << endl;
}
return 0;
}
```

⟨*random_overlay-test.c*⟩≡

```
#define CGAL_USE_LEDA
#include "Segment_overlay_traits.h"
#include "leda_overlay_traits.h"
#include "leda_overlay_visualization.h"
#include <LEDA/stream.h>
#include <LEDA/rat_window.h>
#include <LEDA/param_handler.h>
#include <LEDA/random_rat_point.h>
typedef leda_list<leda_rat_segment>::iterator Seg_iterator;
typedef CGAL::Segment_overlay_traits< Seg_iterator,
          leda_graph_decorator<Seg_iterator>, leda_geometry>
        leda_seg_sweep_traits;
typedef leda_visualization<leda_seg_sweep_traits>   leda_seg_vis;
typedef CGAL::generic_sweep<leda_seg_sweep_traits>  leda_seg_sweep;
typedef CGAL::sweep_observer<leda_seg_sweep,leda_seg_vis> leda_seg_sweep_obs;
```

```
int main(int argc, char** argv)
{
  // SETDTHREAD(23);
  leda_param_handler P(argc,argv,".overlay");
  P.add_parameter("inputfile:-i:string:");
  P.add_parameter("segs:-n:int:100");
  leda_param_handler::init_all();
  leda_string f1;
  int n;
  P.get_parameter("-i",f1);
  P.get_parameter("-n",n);
  leda_window W;
  W.init(-500,500,-500);
  W.set_show_coordinates(true);
  W.display();
  leda_list<leda_rat_segment> L;
  leda_list<leda_rat_point> LP;
  leda_rat_segment s;
  leda_rat_point p,po;
  ifstream if1(f1);
  if (if1) {
    if1 >> L;
    forall (s,L)
     if ( s.is_trivial() ) W << s.source().to_point();
     else  W << s.to_segment();
  } else {
    random_points_in_square(n, n, LP);
    bool first = true;
    forall(p,LP) {
      if (first) { po = p; first = false; }
      else {
        L.append(leda_rat_segment(p,po)); first=true;
        W << leda_rat_segment(p,po);
      }
    }
  }
  ofstream of("input.log");
  of << L;
  of.close();
  cout << "Starting " << CGAL::sweepversion << endl;
  leda_seg_sweep_traits::OUTPUT::Graph      G;
  leda_seg_sweep_traits::OUTPUT::Below_map  E_below(G);
  leda_seg_sweep_traits::OUTPUT O(G,E_below);
  leda_seg_sweep SI(leda_seg_sweep_traits::INPUT(L.begin(),L.end()),O);
  //Obs.attach(SI);
  SI.sweep();
  cout << "\nNodes and Edges below\n";
  leda_node v;
  forall_nodes(v,G) {
    cout <<"  "<<G[v]<<" ";
    if ( E_below[v] ) {
      leda_edge e = E_below[v];
```

```
      cout << "("<<G[source(e)]<<G[target(e)]<< ")\n";
    } else
      cout << "nil\n";
  }
  W.read_mouse();
  return 0;
}
```

Bibliography

[CGA] The CGAL home page. www.cgal.org.

[dBvKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry : Algorithms and Applications*. Springer, Berlin, 1997.

[MN99] K. Mehlhorn and S. Näher. *LEDA, A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MNSU99] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User Manual*, November 1999.

[MS96] David R. Musser and Atul Saini. *STL tutorial and reference guide : C++ programming with the standard template library*. Addison-Wesley professional computing ser. Addison-Wesley, Reading, MA, 1996.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry : An Introduction*. Springer, 1985.

3 Plane Map Overlay

3.1 The manual page

Plane Map Overlay (PM_overlayer)

1. Definition

An instance O of data type *PM_overlayer*<*PMD*, *GEO*> is a decorator object offering plane map overlay calculation. Overlay is either calculated from two plane maps or from a set of segments. The result is stored in a plane map P that carries the geometry and the topology of the overlay.

The two template parameters allow to adapt the overlay calculation to different scenarios. The template parameter *PMD* has to be a model conforming to our plane map decorator concept *PMDecorator*. The concept describes the interface how the topological information stored in P can be extracted. The geometry *GEO* has to be a model conforming to the concept *OverlayerGEO2*.

The overlay of a set of segments S is stored in a plane map $P = (V,E,F)$. Vertices are either the endpoints of segments (trivial segments are allowed) or the result of a non-degenerate internal intersection of two segments. Between two vertices there's an edge if there's a segment that supports the straight line embedding of e and if there's no vertex in the relative interior of the embedding of e.

The faces refer to the maximal connected open point sets of the planar subdivision implied by the embedding of the vertices and edges. Faces are bounded by possibly several face cycles[1] including isolated vertices. The overlay process in the method *create* creates the objects, the topology of the result and allows to link the plane map objects to input segments by means of a data accessor. The method starts from zero- and one-dimensional geometric objects in S and produces a plane map P where each point of the plane can be assigned to an object (vertex, edge, or face) of P.

The overlay of two plane maps $P_i = (V_i,E_i,F_i)$ has the additional aspect that we already start from two planar subdivisions. We use the index $i = 0,1$ defining the reference to P_i, unindexed variables refer to the resulting plane map P. The 1-skeleta of the two maps subdivide the edges and faces of the complementary structure into smaller units. This means vertices and edges of P_i can split edges of P_{1-i} and face cycles of P_i subdivide faces of P_{1-i}. The 1-skeleton P' of P is defined by the overlay of the embedding of the 1-skeleta of P_0 and P_1 (Take a trivial segment for each vertex and a segment for each edge and use the overlay definition of a set of segments above). The faces of P refer to the maximal connected open point sets of the planar subdivision implied by the embedding of P'. Each object from the output tuple (V,E,F) has a *supporting* object u_i in each of the two input structures. Imagine the two maps to be transparencies, which we stack. Then each point of the plane is covered by an object from each of the input structures. This support relation from the input structures to the output structure defines an information flow. Each supporting object u_i of u $(i = 0,1)$ carries an attribute $mark(u_i)$. After the subdivision operation this attribute is associated to the output object u by $mark(u,i)$.

[1]For the definition of plane maps and their concepts see the manual page of *PMConstDecorator*.

2. Generalization

3. Types

PM_overlayer<PMD,GEO>::Decorator	the plane map decorator *PMD*.
PM_overlayer<PMD,GEO>::Plane_map	the plane map type decorated by *PMD*.
PM_overlayer<PMD,GEO>::Geometry	the geometry kernel *GEO*.
PM_overlayer<PMD,GEO>::Point	the point type of the geometric kernel, *Precondition*: *Point* equals *Plane_map::Point*.
PM_overlayer<PMD,GEO>::Segment	the segment type of the geometric kernel.
PM_overlayer<PMD,GEO>::Mark	the attribute type of plane map objects.

4. Creation

PM_overlayer<PMD,GEO> O(Plane_map& P, Geometry g = Geometry( ));

O is a decorator object manipulating *P*.

5. Operations

template <*typename Forward_iterator, typename Object_data_accessor*>
void *O*.create(*Forward_iterator start, Forward_iterator end, Object_data_accessor& A*)

produces in *P* the plane map consistent with the overlay of the segments from the iterator range $[start, end)$. The data accessor *A* allows to initialize created vertices and edges with respect to the segments in the iterator range. *A* requires the following methods:

```
void supporting_segment(Halfedge_handle e, Forward_iterator it)
void trivial_segment(Vertex_handle v, Forward_iterator it)
void starting_segment(Vertex_handle v, Forward_iterator it)
void passing_segment(Vertex_handle v, Forward_iterator it)
void ending_segment(Vertex_handle v, Forward_iterator it)
```

where *supporting_segment* is called for each non-trivial segment $*it$ supporting a newly created edge *e*, *trivial_segment* is called for each trivial segment $*it$ supporting a newly created vertex *v*, and the three last operations are called for each non-trivial segment $*it$ starting at/passing through/ending at the embedding of a newly created vertex *v*. *Precondition*: *Forward_iterator* has value type *Segment*.

void *O*.subdivide(*Plane_map P0, Plane_map P1*)

constructs the overlay of the plane maps *P0* and *P1* in *P*, where all objects (vertices, halfedges, faces) of *P* are *enriched* by the marks of the supporting objects of the two input structures: e.g. let *v* be a vertex supported by a node *v0* in *P0* and by a face *f1* in *P1* and *D0*, *D1* be decorators of type *PM_decorator* on *P0*,*P1*. Then $O.mark(v,0) = D0.mark(v0)$ and $O.mark(v,1) = D1.mark(f1)$.

template <*typename Selection*>
void *O*.select(*Selection& predicate*)

sets the marks of all objects according to the selection predicate *predicate*. *Selection* has to be a function object type with a function operator

```
Mark operator()(Mark m0, Mark m1)
```

For each object *u* of *P* enriched by the marks of the supporting objects according to the previous procedure *subdivide*, after this operation $O.mark(u) = predicate\ (\ O.mark(u,0), O.mark(u,1)\ )$. The additional marks are invalidated afterwards.

template <*typename Keep_edge*>
void *O*.simplify(*Keep_edge keep*)

simplifies the structure of P according to the marks of its objects. An edge e separating two faces $f1$ and $f2$ and equal marks $mark(e) == mark(f1) == mark(f2)$ is removed and the faces are unified. An isolated vertex v in a face f with $mark(v) == mark(f)$ is removed. A vertex v with outdegree two, two collinear out-edges $e1$,$e2$ and equal marks $mark(v) == mark(e1) == mark(e2)$ is removed and the edges are unified. The data accessor *keep* requires the function call operator

```
bool operator()(Halfedge_handle e)
```

that allows to avoid the simplification for edge pairs referenced by e.

3.2 Implementation

In this section we present a software module for the overlay of segments and plane maps. We first give a formal introduction to the notions and difficulties concerning overlay and support. We then present the overlay calculation of a set of segments. We show how we use a generic sweep module to produce the 1-skeleton of the output plane map. In a different section we show how to add face objects to the 1-skeleton to complete the output structure. The second operation concerns the overlay of two plane maps. We use the same generic sweep module with slightly more elaborate adaptation to obtain again the 1-skeleton of the overlay. The face production phase will be the same as before. In case of the second overlay operation our sweep adds a transfer of information assigned to the objects of the two input plane maps to corresponding objects in the output structure. This allows us to use the module for binary set operations on plane map structures. Such set operations use a selection phase on the transferred information items. The selection phase is descibed below in an additional section. The last section in this document concerns structural simplification of the output plane map. We will see that there can be substructures in the output plane map that can be simplified without losing any information when the plane map is interpreted as a point set.

3.2.1 Notions and definitions

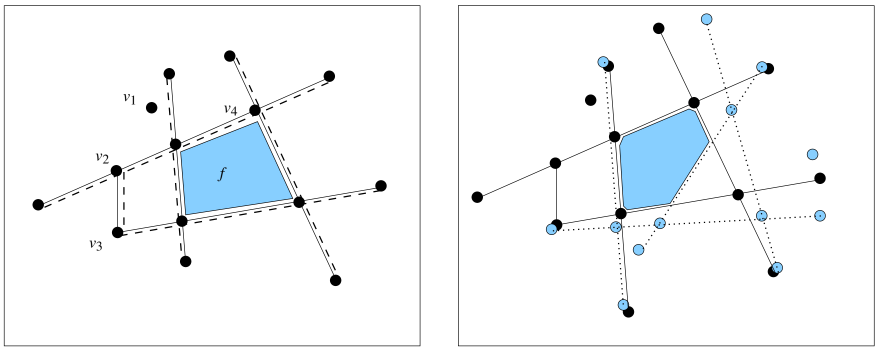


Figure 3.1: The overlay of a set of segments and of two plane maps. The left figure shows a set of dashed segments. v_1 is an isolated vertex, v_2 is an endpoint in the interior of another segment, v_3 is a vertex supported by two endpoints, v_4 is the intersection of the relative interior of two segments. The edges are drawn with solid line segments. One bounded face is greyed. The right figure shows the 1-skeleta of two plane maps. Degenerate situations are identical vertices, vertices in the interior of edges, and overlapping edges.

If we consider our overlay process as a transformation of input objects to output objects then we can define the support relation as follows.

Definition 5 (support): Consider an algorithm T that transforms a set of input objects A to a set of output objects B where each $a \in A$ and $b \in B$ represents a subset of R^2. We say that *a supports b* if b is a subset of a with respect to the represented point sets.

We will anchor this notion in the following.

Overlay of a set of segments For a segment $s = (p,q)$, $p = source(s)$, $q = target(s)$ and p, q are called the endpoints of s. Let us consider s as a disjoint union of its endpoints and its relative interior $\mathsf{relint}\, s$. A set of segments S partitions the plane into cells of different dimensions. For each point $r \in \mathbb{R}^2$ it can happen that

(i) r is equal to some endpoint p of some segment s, or
(ii) r is part of the relative interior of some segment s, or
(iii) r is not part of any segment at all.

Note that (i) and (ii) do not exclude each other. Now consider the geometric structure built by all segments. The *overlay* of all segments is the subdivision of all points in $\mathbb{R}^2$ with respect to the three criteria (i) to (iii) above including their topological neighborhood and the knowledge how parts of the segments in S support the cells of the subdivision.

We store the overlay of S in a plane map $P = (V, E, F)$ in the standard way. For each point r in (i) there is a vertex v in V where the endpoint of the segment supports the vertex. If r is additionally in the relative interior of some other segment according to (ii) then this segment also supports v. For each point in (ii) that is the unique intersection point of the relative interior of two segments (that do not overlap) there is a vertex in V and the relative interior of each of the two segments supports that vertex. Between any two vertices in V there is a uedge e in E if there is a segment s that supports the straight line embedding of e according to (ii) and there is no further vertex in the relative interior of e. The latter can happen for several segments that overlap. Any point of (iii) belongs to one of the maximal connected sets[2] of $\mathbb{R}^2 - S$ that form the faces of P and is thus not supported by any segment at all.

Overlay of two plane maps Let $P_i = (V_i, E_i, F_i), i = 0, 1$ be two plane map structures. The overlay of two plane maps P_0, P_1 is the plane map P representing the subdivision of the plane obtained by interpreting the skeleton objects of P_i according to their embedding as trivial and non-trivial segments, constructing the overlay of these segments and adding the faces. To make this structure really helpful we explore the support relation between object of P_i and P.

In general, each point p in the plane is supported by that object of a plane map whose corresponding point set contains p. The support relation between P_i and P comes in two steps. Each 1-skeleton object of P_i relates to the endpoint or relative interior of a segment that supports a skeleton object in P. Reversely, each object of P (vertex, edge, or face) is supported by a unique supporting object in each of the two structures P_i $(i = 0, 1)$. We show that this relation is well-defined.

Lemma 3.2.1: Any object of P has exactly one supporting object in each of the P_i.

Proof. Obviously, each point of the plane is supported by an object of P_i. Therefore, we only have to argue why no two objects of P_i can support one object of P. For vertices this is trivial. For a uedge e in P there can be only one uedge or one face of P_i that supports e: assume that the embedding of e covers points from more than one object of P_i. Then, e either contains a vertex or crosses an edge in its interior. But then, the corresponding subdivision would have prevented the creation of e in P in the first place.

For a face f of P there can be only one face f' of P_i that supports f: assume otherwise, that f contains points from different objects of P_i. As f is an open connected point set it has to cover points of at least one boundary object from the 1-skeleton of P_i. But this object is part of the 1-skeleton of P and can therefore never be part of f. □

In our implementation we determine the support relation in two phases. Any vertex v in V can be supported by a vertex v_i, a uedge e_i, or a face f_i of P_i. If v is supported by v_i or e_i we obtain this information in a plane sweep process. Assume that v is supported by a face f_i (then v is supported by a vertex v_{1-i}). During the sweep process the determination of a support of f_i is hard, as the face objects

[2] path connected in the strong topological sense.

are not in reach. We determine f_i in a postprocessing phase by a simple iteration over all vertices. Any edge e in E can be supported by a uedge e_i or a face f_i of P_i. A possible support by e_i is handled during the sweep process. In case e is part of a face f_i (again e is then supported by an edge e_{1-i}) we also determine f_i in the postprocessing phase.

The support for a face f in F can be determined as follows. Assume that each directed edge e in E knows the faces f_i supporting points in a small neighborhood on its left side ($i = 0, 1$). Then, f can determine its two supporting faces f_i via any edge in its boundary cycle. We will enrich the edges of E by such support information and use it afterwards to transfer attributes from f_i to f.

3.2.2 The class design

We start with the design of the class object. Our generic overlay class can be adapted via two interface concepts. We interface the underlying plane map via a plane map decorator *PMDEC*, we interface the underlying geometry via a geometry kernel *GEOM*. We inherit from *PMDEC* to obtain its interface methods.

⟨*PM overlayer*⟩≡
```
template <typename PM_decorator_, typename Geometry_>
class PM_overlayer : public PM_decorator_ {
  typedef PM_decorator_ Base;
  typedef PM_overlayer<PM_decorator_,Geometry_>  Self;
  const Geometry_& K; // geometry reference

public:
  typedef PM_decorator_                 Decorator;
  typedef typename Decorator::Plane_map Plane_map;
  typedef Geometry_                     Geometry;
  typedef typename Geometry::Point_2    Point;
  typedef typename Geometry::Segment_2  Segment;
  typedef typename Decorator::Mark      Mark;
  ⟨handles, iterators, and circulators from Decorator⟩
  ⟨info type to link edges and segments⟩
PM_overlayer(Plane_map& P, const Geometry& g = Geometry()) :
  Base(P), K(g) {}
⟨subdivision⟩
⟨selection⟩
⟨simplification⟩
⟨helping operations⟩
}; // PM_overlayer<PM_decorator_,Geometry_>
```

⟨*handles, iterators, and circulators from Decorator*⟩≡
```
#define USING(t) typedef typename Decorator::t t
typedef typename Decorator::Base Const_decorator;
USING(Halfedge_handle);
USING(Vertex_handle);
USING(Face_handle);
USING(Vertex_iterator);
USING(Halfedge_iterator);
```

```
USING(Face_iterator);
USING(Halfedge_const_handle);
USING(Vertex_const_handle);
USING(Face_const_handle);
USING(Halfedge_const_iterator);
USING(Vertex_const_iterator);
USING(Face_const_iterator);
USING(Halfedge_around_vertex_circulator);
USING(Halfedge_around_face_circulator);
USING(Hole_iterator);
USING(Isolated_vertex_iterator);
#undef USING
// C++ is really friendly:
#define USECMARK(t) const Mark& mark(t h) const { return Base::mark(h); }
#define USEMARK(t)  Mark& mark(t h) const { return Base::mark(h); }
USEMARK(Vertex_handle)
USEMARK(Halfedge_handle)
USEMARK(Face_handle)
USECMARK(Vertex_const_handle)
USECMARK(Halfedge_const_handle)
USECMARK(Face_const_handle)
#undef USEMARK
#undef USECMARK
```

3.2.3 Overlay calculation of a list of segments

We want to calculate the plane map P representing the overlay of a set S of segments, some of which may be trivial. This task is basically split in **two phases**:

overlay of segments — the calculation of the 1-skeleton $P' = (V,E)$ of a plane map via the overlay of the segments in S plus the calculation of a map *halfedge_below* : $V \to E$

face creation — the completion of the 1-skeleton P' to a full plane map $P = (V,E,F)$ by creating all faces while using the information of the map *halfedge_below*.

For the overlay process we use the generic segment sweep module as presented in the technical report [MS01]. There we presented a generic class *Segment_overlay_traits* realizing a generic sweep framework. To instantiate it we have to provide three components (input, output, geometry). In this instance the input is an iterator pair, the geometry is forwarded from the current class scope. Only for the output type we have to work a little more. We define a class *PMO_from_segs* that fits the output concept of *Segment_overlay_traits* and at the same time is a model for the *Below_info* concept required for the facet creation in Section 3.2.5. (See Figure 3.2.)

On creation an object of type *PMO_from_segs* references a plane map via a decorator G and obtains a data accessor object D of type *DA*. *PMO_from_segs* as a model of *SegmentOverlayOutput* triggers the correct update operations on the output plane map during the sweep. See the output concept in Figure 3.2. The method part O1 of *SegmentOverlayOutput* takes care of the plane map extension by new vertices and edges. The part O2 allows to obtain information how the creation of the objects is linked to the input interators. In the implementation of *PMO_from_segs* we forward this interface to methods

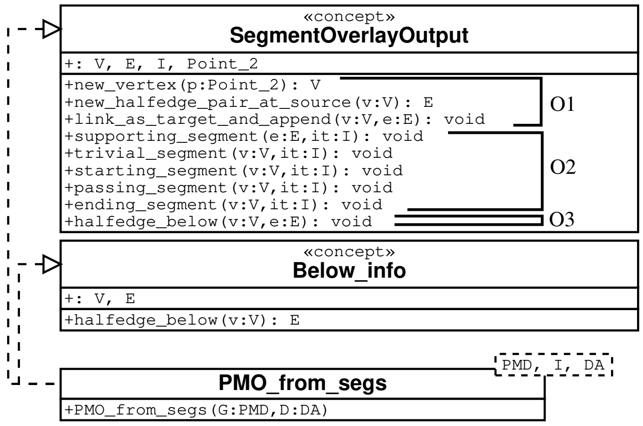


Figure 3.2: *PMO_from_segs* realizes the *Output* concept of the generic sweep module and the *Below_info* concept for the facet creation phase. In the figure *Vertex_handle*, *Halfedge_handle*, and *Iterator* have been replaced by the short symbols V, E, and I.

of the data accessor of type DA. Finally, part O3 can be used to collect the additional information required for the facet creation. An edge e that is immediately below a vertex v is stored in the vertex object in a temporarily assigned data slot and can be retrieved after the sweep. *PMO_from_segs* as a *Below_info* model can thus afterwards deliver the halfedge *halfedge_below*(v) for any vertex v of the plane map.

At this point our readers should take the module

```
generic_sweep< Segment_overlay_traits<PMO_from_segs<... >... >>
```

as a black box producing the 1-skeleton of P with the properties required in Section 3.2.5. The specification of *Segment_overlay_traits* guarantees these properties of P because *PMO_from_segs* fits the requirements of the output concept of *Segment_overlay_traits*.

⟨*PM traits classes for segment overlay*⟩≡

```
template <typename PMD, typename I, typename DA>
struct PMO_from_segs {
  typedef PMD Decorator;
  typedef typename Decorator::Vertex_handle   Vertex_handle;
  typedef typename Decorator::Halfedge_handle Halfedge_handle;
  typedef typename Decorator::Point           Point;
  const Decorator& G;
  DA& D;
  PMO_from_segs(const Decorator& Gi, DA& Di) :
    G(Gi),D(Di) {}
```

⟨*PMO_from_segs segment overlay model interface*⟩
⟨*PMO_from_segs face creation model interface*⟩
⟨*PMO_from_segs additional interface*⟩

```
}; // PMO_from_segs
```

The creation of new objects is forwarded to the *Decorator* object G. The methods of G are members as described in the *PM_decorator* concept. The following methods are called during the sweep at its event points. The first three methods trigger object creation in P. We associate a *Halfedge_handle* to each vertex v via its generic storage slot *GenPtr& info*(v). We use a scheme in analogy to LEDA [MN99, chapter 13], where information (in form of a built-in or class type) is stored directly in the pointer if it has size not larger than the size of a standard word. If it does not fit, the pointer is used to reference a newly allocated information object on the heap. The scheme is bundled in a class called *geninfo*<*T*>. For more information see that manual page in the appendix.

⟨*PMO_from_segs segment overlay model interface*⟩≡
```
Vertex_handle new_vertex(const Point& p)
{ Vertex_handle v = G.new_vertex(p);
  geninfo<Halfedge_handle>::create(G.info(v));
  return v;
}
void link_as_target_and_append(Vertex_handle v, Halfedge_handle e)
{ G.link_as_target_and_append(v,e); }
Halfedge_handle new_halfedge_pair_at_source(Vertex_handle v)
{ Halfedge_handle e =
  G.new_halfedge_pair_at_source(v,Decorator::BEFORE);
  return e;
}
```

The treatment of the new objects is forwarded to the *DA* object *D*. Only the below link is stored via *G*. The following methods allow us to hook methods of *D* into the plane sweep process.

⟨*PMO_from_segs segment overlay model interface*⟩+≡
```
void supporting_segment(Halfedge_handle e, I it) const
{ D.supporting_segment(e,it); }
void trivial_segment(Vertex_handle v, I it) const
{ D.trivial_segment(v,it); }
void starting_segment(Vertex_handle v, I it) const
{ D.starting_segment(v,it); }
void passing_segment(Vertex_handle v, I it) const
{ D.passing_segment(v,it); }
void ending_segment(Vertex_handle v, I it) const
{ D.ending_segment(v,it); }
void halfedge_below(Vertex_handle v, Halfedge_handle e) const
{ geninfo<Halfedge_handle>::access(G.info(v)) = e; }
```

PMO_from_segs fits also the concept *Below_info* for the face creation defined in Section 3.2.5.

⟨*PMO_from_segs face creation model interface*⟩≡
```
Halfedge_handle halfedge_below(Vertex_handle v) const
{ return geninfo<Halfedge_handle>::access(G.info(v)); }
```

We finally add a clean up operation discarding the temporary storage.

⟨*PMO_from_segs additional interface*⟩≡

```
void clear_temporary_vertex_info() const
{ Vertex_handle v;
  for(v = G.vertices_begin(); v!= G.vertices_end(); ++v)
    geninfo<Halfedge_handle>::clear(G.info(v));
}
```

Now, the overlay creation is trivial. Just create an output decorator object *Out* working on the plane map maintained by *PM_overlayer* and plug it into the segment sweep overlay framework *Segment_overlay_traits*. The used geometry is just forwarded from *PM_overlayer*. The *create* method of *PM_overlayer* is parameterized by the iterator type *Forward_iterator* and the data accessor class *Object_data_accessor*.

Note that the *halfedge_below* information collected during the sweep is associated with the vertices of the output map. The corresponding object *Out* triggers the output creation during the sweep and provides the halfedge-below information for the face creation in *create_face_objects*( ). *Out.clear_temporary_vertex_info*( ) just discards the temporarily allocated information slots (internally assigned to the vertices) on the heap.

⟨*subdivision*⟩≡

```
template <typename Forward_iterator, typename Object_data_accessor>
void create(Forward_iterator start, Forward_iterator end,
            Object_data_accessor& A) const
{
  typedef PMO_from_segs<Self,Forward_iterator,Object_data_accessor>
    Output_from_segments;
  typedef Segment_overlay_traits<
    Forward_iterator, Output_from_segments, Geometry> seg_overlay;
  typedef generic_sweep< seg_overlay > seg_overlay_sweep;
  typedef typename seg_overlay::INPUT input_range;
  Output_from_segments Out(*this, A);
  seg_overlay_sweep SOS( input_range(start, end), Out, K);
  SOS.sweep();
  create_face_objects(Out);
  Out.clear_temporary_vertex_info();
}
```

We summarize the calculated overlay properties and anticipate the costs of face creation in Section 3.2.5 and of the plane sweep description. (see Lemma 2.1.3).

Lemma 3.2.2: Assume that $S = set\ [start, end)$ is a set of segments and A is a data accessor with the required methods (of constant cost). Then, $create(start, end, A)$ constructs in $P = (V, E, F)$ the overlay plane map of S. Let n be the number of segments in S, $n_v = |V|$, $n_e = |E|$, and $\bar{n}_e$ the sum of the support multiplicity of each edge over all edges. Then the runtime of the overlay process is dominated by the plane sweep and is therefore $O(n_v + \bar{n}_e + (n + n_v)\log(n + n_v))$.

3.2.4 Overlay calculation of two plane maps

We calculate the overlay P of two plane maps P_0 and P_1. Both input structures are two correctly defined plane maps including incidence, geometric embedding, and markers. In the following we use the index $i = 0,1$ showing a reference to $P_i = (V_i, E_i, F_i)$; non-indexed variables refer to P.

The 1-skeleta of the two maps P_0 and P_1 subdivide the edges and faces of the complementary structures into smaller units. This means vertices and edges of P_i can split edges of P_{1-i} and face cycles of P_i subdivide faces of P_{1-i}. The 1-skeleton $P' = (V,E)$ of P is defined by the overlay of the embedding of the 1-skeleta of P_0 and P_1 (Take a trivial segment for each vertex and a segment for each edge and use the overlay definition of a set of segments above). Additionally, we require that P' has the correct order in each adjacency list such that it is order-preserving regarding the embedding of the vertices.

Finally, the faces of P refer to the maximal connected open point sets of the planar subdivision implied by the embedding of P'. The construction of the faces F from P' is described in Section 3.2.5. Each object u from the output tuple (V,E,F) has a *supporting* object $u_i, i = 0,1$ in each of the two input structures. Imagine the two maps to be transparencies, stacked one on top of the other. Then each point of the plane is covered by an object from each of the input structures. We analyse the support relation from input to output in order to transfer the attributes from u_i to u.

According to our specification each object u_i of P_i carries an attribute[3] $mark(u_i)$ ($mark : (V_i \cup E_i \cup F_i) \to Mark$). We associate this information with the output object u by $mark(u,i)$ (an overloaded function $mark : (V \cup E \cup F) \times \{0,1\} \to Mark$). This two-tuple of information per object can then be processed by some combining operation to a single value $mark(u)$ lateron.

We fix the following **input properties** for our structures P_i. Both plane maps (V_i, E_i, F_i) consist of vertices, edges, and faces whose topology is accessible by our plane map interface and additionally each object u_i carries an attribute $mark(u_i)$. The plane maps have an *order-preserving* embedding and their adjacency lists have a *forward prefix*. Actually we do not use this property of the input plane maps at this point but it is a general invariant of our plane map structures that makes some intermediate actions more efficient. The overlay process consists of **three phases**: The 1-skeleton P' is produced by segment overlay. Afterwards we create the face objects. Finally, we analyse the support relation and transfer the marks of the input objects to the output objects.

overlay of segments — We use our generic segment overlay framework to calculate the overlay of a set of segments S. The set S consists of all segments that are the embedding of edges in E_i and additionally trivial segments representing all isolated vertices in V_i. The output structure $P' = (V,E)$ of the sweep phase is just the 1-skeleton of the output plane map P, but of course including an order-preserving embedding and a forward-prefix in the adjacency lists. The objects of the 1-skeleton carry additional structural information:

I1. Each vertex v in V knows a halfedge $e \in E : e = halfedge\_below(v)$ which is determined by the property that a vertical ray shot from v along the negative y-axis hits e first. Degeneracies are broken with a perturbation scheme: during the ray shooting all edges include their source vertex.

I2. For each object $u \in V \cup E$ there is a mapping to the supporting 1-skeleton objects of the input structures. The support information is incomplete with respect to face support.

[3] we use a general attribute set, though with respect to Nef polyhedra $Mark := \{true, false\}$.

face creation — The next phase after the sweep has to complete the plane map P. We basically have to create the face objects and construct their incidence structure. The face creation is done as presented in Section 3.2.5 and uses only I1.

attribute transfer — The final transfer of marks uses the embedding of the vertex list of P and the additional information I1 and I2 to define $mark(u,i)$ for all objects u in P.

⟨*subdivision*⟩+≡

```
void subdivide(const Plane_map& P0, const Plane_map& P1) const
{
  Const_decorator PI[2];
  PI[0] = Const_decorator(P0); PI[1] = Const_decorator(P1);
  ⟨filling the input segment list⟩
  ⟨sweeping the segments and creating the faces⟩
  ⟨transfering the marks of supporting objects⟩
}
```

Temporary information associated with objects

We have to associate temporary information with the objects of the output plane map. In this section we abstractly use sets in a pseudo code notation to underline the origin of plane map objects. The objects from these sets are realized by the corresponding handle types (and therefore their type does not allow to mark their origin). Undefined objects are detectible via default handles.

At first we interpret the input 1-skeleta geometrically. We collect a set of trivial and non-trivial segments S. For each edge in E_i we add a non-trivial segment to S and for each isolated vertex of V_i we add a trivial segment to S. We store the origin of the objects in S via a function

$$\begin{aligned} From \quad &: \quad S \rightarrow (V_{0,1} \cup E_{0,1}) \times \{0,1\} \\ From(s) \quad &= \quad \begin{cases} (v_i, i) & \text{if } s \text{ is a trivial segment refering to an isolated vertex } v_i \text{ from } P_i, \\ (e_i, i) & \text{if } s \text{ is a non-trivial segment refering to an edge } e_i \text{ from } P_i. \end{cases} \end{aligned}$$

From is implemented as a hash map *From* whose domain are iterators (with value type segment) and whose value is a structure *Seg_info* with members v, e, i storing the above pairs.

⟨*info type to link edges and segments*⟩≡

```
struct Seg_info { // to transport information from input to output
  Halfedge_const_handle e;
  Vertex_const_handle   v;
  int                   i;
  Seg_info() : i(-1) {}
  Seg_info(Halfedge_const_handle e_, int i_)
  { e=e_; i=i_; }
  Seg_info(Vertex_const_handle v_, int i_)
  { v=v_; i=i_; }
  Seg_info(const Seg_info& si)
  { e=si.e; v=si.v; i=si.i; }
  Seg_info& operator=(const Seg_info& si)
```

```
    { e=si.e; v=si.v; i=si.i; return *this; }
    LEDA_MEMORY(Seg_info)
  };
```

⟨*info type to link edges and segments*⟩+≡

```
  typedef std::list&lt;Segment&gt;                   Seg_list;
  typedef typename Seg_list::const_iterator    Seg_iterator;
  typedef std::pair<Seg_iterator,Seg_iterator> Seg_it_pair;
```

In the first phase we fill the segment input list with a non-trivial segment underlying each edge and with a trivial segment for each isolated vertex of the two input structures. Additionally, we store hashed links from the iterators to the edges/vertices to store their origin.

⟨*filling the input segment list*⟩≡

```
  Seg_list Segments; int i;
  CGAL::Unique_hash_map<Seg_iterator,Seg_info> From;
  for (i=0; i<2; ++i) {
    Vertex_const_iterator v;
    for(v = PI[i].vertices_begin(); v != PI[i].vertices_end(); ++v)
      if ( PI[i].is_isolated(v) ) {
        Segments.push_back(segment(PI[i],v));
        From[--Segments.end()] = Seg_info(v,i);
      }
    Halfedge_const_iterator e;
    for(e = PI[i].halfedges_begin(); e != PI[i].halfedges_end(); ++e)
      if ( is_forward_edge(PI[i],e) ) {
        Segments.push_back(segment(PI[i],e));
        From[--Segments.end()] = Seg_info(e,i);
      }
  }
```

During the sweep phase we collect additional information in temporary information containers associated with the objects $u \in V \cup E \cup F$ of P.

assoc_info(u)	creates the temporary object on the heap
discard_info(u)	discards the object and frees the memory

Within these objects we store the following pair of mark attributes (indexed by i):

Mark mark(u,i) for $i = 0,1$ and $u \in V \cup E \cup F$

For each vertex v we collect *skeleton support* information.

V_i *supp_vertex*$(V\,v, int\,i)$	the vertex from V_i supporting v if it exists, else undefined.
E_i *supp_halfedge*$(V\,v, int\,i)$	the edge from E_i supporting e if it exists, else undefined.

And for each edge e we want to know

E_i *supp_halfedge*$(E\ e, int\ i)$	the edge from E_i supporting e if it exists, else undefined.
Mark incident_mark$(E\ e, int\ i)$	the mark of the face from P_i supporting a small neighborhood left of e.

The information is collected during the sweep phase by a corresponding model of the output concept used in our generic sweep framework. We show more details of the information association. We use a trick via generic pointers *GenPtr* (equals *void*∗). Each object u of P has such a slot accessible via *GenPtr*& $info(u)$. We use the pointer to reference an object storing the temporary information until the postprocessing does not need it anymore. We have to ensure that we do not mess around with the memory. The temporary information is collected during the sweep operation. After the selection operation the temporary information is discarded. To avoid superflous indirection for the temporary information we use a scheme in analogy to LEDA [MN99, chapter 13], where information (in form of a built-in or class type) is stored directly in the pointer if it has size not larger than the size of a standard word. If it does not fit, the pointer is used to reference a newly allocated information object on the heap. This scheme leaves the vertex, halfedge, and face objects minimal in the sense that we only have one additional pointer in each of them. The scheme is bundled in a class called *geninfo*<*T*>. For more information see that manual page in the appendix.

We associate the following class to a vertex v. The methods below are added to the interface of *PM_overlayer*. *vertex_info* can store the possible supporting skeleton objects *v_supp*, *e_supp* of the input plane maps, marks m corresponding to the two supporting objects and a halfedge *e_below* of the output structure that is vertically below v.

⟨*helping operations*⟩≡
```
struct vertex_info {
  Mark                  m[2];
  Vertex_const_handle   v_supp[2];
  Halfedge_const_handle e_supp[2];
  Halfedge_handle       e_below;
  vertex_info()
  { v_supp[0]=v_supp[1]=Vertex_const_handle();
    e_supp[0]=e_supp[1]=Halfedge_const_handle(); }
  LEDA_MEMORY(vertex_info)
};

void assoc_info(Vertex_handle v) const
{ geninfo<vertex_info>::create(info(v)); }

void discard_info(Vertex_handle v) const
{ geninfo<vertex_info>::clear(info(v)); }

vertex_info& ginfo(Vertex_handle v) const
{ return geninfo<vertex_info>::access(info(v)); }

Mark& mark(Vertex_handle v, int i) const
{ return ginfo(v).m[i]; }

Vertex_const_handle& supp_vertex(Vertex_handle v, int i) const
{ return ginfo(v).v_supp[i]; }

Halfedge_const_handle& supp_halfedge(Vertex_handle v, int i) const
{ return ginfo(v).e_supp[i]; }

Halfedge_handle& halfedge_below(Vertex_handle v) const
{ return ginfo(v).e_below; }
```

For each halfedge we store the following class. Again we provide an interface in *PM_overlayer*. The *halfedge_info* objects store information common to both halfedge twins (information for the uedge) and information only concerning one halfedge. In the first case we store the information only in the halfedge determined by the smaller memory address which makes access unique. We provide storage *e_supp* for the possible two input edges supporting an output edge, the marks *m* of the two input objects supporting an output edge, and temporary storage *mf* for the marks of the two input faces supporting points in a small neighborhood of the edge. The boolean flag *forw* just caches the geometric property if an edge is forward oriented (the source is lexicographically smaller than the target).

⟨*helping operations*⟩+≡
```
struct halfedge_info {
  Mark                 m[2];
  Mark                 mf[2];
  Halfedge_const_handle e_supp[2];
  bool                 forw;
  halfedge_info()
  { m[0]=m[1]=mf[0]=mf[1]=Mark();
    e_supp[0]=e_supp[1]=Halfedge_const_handle();
    forw=false; }
  LEDA_MEMORY(halfedge_info)
};
void assoc_info(Halfedge_handle e)  const
{ geninfo<halfedge_info>::create(info(e));
  geninfo<halfedge_info>::create(info(twin(e))); }
void discard_info(Halfedge_handle e)  const
{ geninfo<halfedge_info>::clear(info(e));
  geninfo<halfedge_info>::clear(info(twin(e))); }
halfedge_info& ginfo(Halfedge_handle e)  const
{ return geninfo<halfedge_info>::access(info(e)); }
Mark& mark(Halfedge_handle e, int i)  const
// uedge information we store in the smaller one
{ if (&*e < &*(twin(e))) return ginfo(e).m[i];
  else                   return ginfo(twin(e)).m[i]; }
Halfedge_const_handle& supp_halfedge(Halfedge_handle e, int i) const
// uedge information we store in the smaller one
{ if (&*e < &*(twin(e))) return ginfo(e).e_supp[i];
  else                   return ginfo(twin(e)).e_supp[i]; }
Mark& incident_mark(Halfedge_handle e, int i)  const
// biedge information we store in the halfedge
{ return ginfo(e).mf[i]; }
bool& is_forward(Halfedge_handle e) const
// biedge information we store in the halfedge
{ return ginfo(e).forw; }
```

A face just obtains two mark slots *m*.

⟨*helping operations*⟩+≡
```
struct face_info {
  Mark m[2];
  face_info() { m[0]=m[1]=Mark(); }
  LEDA_MEMORY(face_info)
};
void assoc_info(Face_handle f)  const
{ geninfo<face_info>::create(info(f)); }
```

```
void discard_info(Face_handle f)  const
{ geninfo<face_info>::clear(info(f)); }
face_info& ginfo(Face_handle f)  const
{ return geninfo<face_info>::access(info(f)); }
Mark& mark(Face_handle f, int i)  const
{ return ginfo(f).m[i]; }
```

Finally we provide an operation cleaning up the temporary attributes allocated above.

⟨*helping operations*⟩+≡

```
void clear_associated_info_of_all_objects() const
{
  Vertex_iterator vit;
  for (vit = vertices_begin(); vit != vertices_end(); ++vit)
    discard_info(vit);
  Halfedge_iterator hit;
  for (hit = halfedges_begin(); hit != halfedges_end(); ++hit)
    discard_info(hit);
  Face_iterator fit;
  for (fit = faces_begin(); fit != faces_end(); ++fit)
    discard_info(fit);
}
```

The sweep instantiation

We have to provide the three components (input, output, geometry) necessary to instantiate the traits model *Segment_overlay_traits* for our generic plane sweep framework. The input is an iterator pair, the geometry is forwarded from the current class scope. Only for the output type we have to work a little more. We define a class *PMO_from_pm* below which allows us to track the support relationship from input objects (segments handled via iterators) to the output objects (vertices and halfedges) via the call-back methods triggered during the sweep. Please refer to the description of *Segment_overlay_traits*.

The methods of *PMO_from_pm* fit the output concept requirements of *Segment_overlay_traits*. The functionality is such that the skeleton is created and the support information is associated with the newly created objects. *PMO_from_pm* is a class template on the global implementation scope as the usage of local class types (within the scope of *PM_overlayer*) is not allowed by some current C++ compilers.

Note that we forward a reference to our hash map *From* to the output decorator object. Thus we can update the support linkage from output skeleton objects via iterators to input objects on the fly when the sweep frameworks calls the corresponding methods of *PMO_from_pm*.

⟨*PM traits classes for segment overlay*⟩+≡

```
template <typename PMD, typename IT, typename INFO>
struct PMO_from_pm {
  ⟨importing decorators, handles, and point type from PMD⟩
  const Decorator& G;
  const Const_decorator* pGI[2];
```

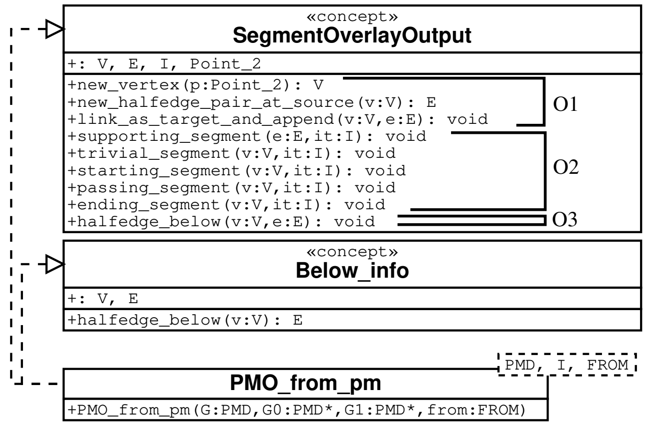


Figure 3.3: *PMO_from_pm* realizes the *Output* concept of the generic sweep module and the *Below_info* concept for the facet creation phase. In the figure *Vertex_handle*, *Halfedge_handle*, and *Iterator* have been replaced by the short symbols V, E, and I.

```
  CGAL::Unique_hash_map<IT,INFO>& M;
  PMO_from_pm(const Decorator& Gi,
              const Const_decorator* pG0,
              const Const_decorator* pG1,
              CGAL::Unique_hash_map<IT,INFO>& Mi) : G(Gi),M(Mi)
 { pGI[0]=pG0; pGI[1]=pG1; }
```

⟨*PMO_from_pm topological updates*⟩
⟨*PMO_from_pm vertical ray shoot knowledge*⟩
⟨*PMO_from_pm support knowledge*⟩
⟨*PMO_from_pm face creation data access*⟩

```
}; // PMO_from_pm
```

⟨*importing decorators, handles, and point type from PMD*⟩≡

```
typedef PMD Decorator;
typedef typename PMD::Const_decorator Const_decorator;
typedef typename Decorator::Vertex_handle Vertex_handle;
typedef typename Decorator::Halfedge_handle Halfedge_handle;
typedef typename Decorator::Vertex_const_handle Vertex_const_handle;
typedef typename Decorator::Halfedge_const_handle Halfedge_const_handle;
typedef typename Decorator::Point Point;
```

New vertices and halfedges are created in the plane map via a call to corresponding creation methods of the decorator. Note that we initialize a temporary storage slot in the objects by a call to *assoc_info*.

⟨*PMO_from_pm topological updates*⟩≡

```
Vertex_handle new_vertex(const Point& p) const
{ Vertex_handle v = G.new_vertex(p);
  G.assoc_info(v);
  return v;
}
void link_as_target_and_append(Vertex_handle v, Halfedge_handle e) const
```

```
{ G.link_as_target_and_append(v,e); }
Halfedge_handle new_halfedge_pair_at_source(Vertex_handle v) const
{ Halfedge_handle e =
  G.new_halfedge_pair_at_source(v,Decorator::BEFORE);
  G.assoc_info(e);
  return e;
}
```

The halfedge vertically below a vertex is stored in a slot of the temporarily associated information container of type *vertex_info*. Access is done via the *PM_overlayer* operation *halfedge_below*( ).

⟨*PMO_from_pm vertical ray shoot knowledge*⟩≡

```
void halfedge_below(Vertex_handle v, Halfedge_handle e) const
{ G.halfedge_below(v) = e; }
```

For a new halfedge *e* we get to know all segments $*it$ that support it. We store the information via the decorator. There can be at most two input segments supporting an edge.

⟨*PMO_from_pm support knowledge*⟩≡

```
void supporting_segment(Halfedge_handle e, IT it) const
{ INFO& si = M[it];
  assert( si.e != Halfedge_const_handle() );
  G.supp_halfedge(e,si.i) = si.e;
  G.is_forward(e) = true;
}
```

For a vertex *v* we get to know support information. There are three basic cases: *v* is supported by an isolated vertex, *v* is supported by a vertex from one input structure which has incident edges (starting or ending) during the sweep, or *v* comes to lie in the relative interior of an input edge. In either case one of the following operations attributes the correct support information.

⟨*PMO_from_pm support knowledge*⟩+≡

```
void trivial_segment(Vertex_handle v, IT it) const
{ INFO& si = M[it];
  assert( si.v != Vertex_const_handle() );
  G.supp_vertex(v,si.i) = si.v;
}
void starting_segment(Vertex_handle v, IT it) const
{ INFO& si = M[it];
  G.supp_vertex(v,si.i) = pGI[si.i]->source(si.e);
}
void ending_segment(Vertex_handle v, IT it) const
{ INFO& si = M[it];
  G.supp_vertex(v,si.i) = pGI[si.i]->target(si.e);
}
void passing_segment(Vertex_handle v, IT it) const
{ INFO& si = M[it];
  G.supp_halfedge(v,si.i) = si.e;
}
```

PMO_from_pm also provides data access in the face creation phase. Therefore this operation that redirects the access to *PM_overlayer* object G.

⟨*PMO_from_pm face creation data access*⟩≡
```
Halfedge_handle halfedge_below(Vertex_handle v) const
{ return G.halfedge_below(v); }
```

Now, creating the overlay is a trivial plugging of types into the generic plane sweep framework, a creation of the sweep object with input, output and geometry references, and a final execution of the sweep. Afterwards, the faces are created.

⟨*sweeping the segments and creating the faces*⟩≡
```
typedef PMO_from_pm<Self,Seg_iterator,Seg_info> Output_from_plane_maps;
typedef Segment_overlay_traits<
  Seg_iterator, Output_from_plane_maps, Geometry> pm_overlay;
typedef generic_sweep< pm_overlay > pm_overlay_sweep;
Output_from_plane_maps Out(*this,&PI[0],&PI[1],From);
pm_overlay_sweep SOS(Seg_it_pair(Segments.begin(),Segments.end()),Out,K);
SOS.sweep();
create_face_objects(Out);
```

Transfering the marks

After the sweep and the face creation the input for this phase is a plane map $P = (V,E,F)$ enriched by additional information attributed to the 1-skeleton objects of P. The output vertices in V are linked to their supporting skeleton input objects (vertices and edges). The output edges in E are linked to their supporting input edges. The support knowledge with respect to input faces is still missing. In the following we analyse this support but do not store it explicitly. Instead we only transfer the marks. There are several properties of the constructed subdivision P which help us to do this.

- the vertices are constructed in the order of the sweep. By iterating them in their construction order we can rely on the fact that we iterate according to the lexicographic order of their embedding.
- the halfedges out of a vertex v are ordered around v counterclockwise (with respect to the embedding of their target). We can therefore use a forward iteration to propagate face information from bottom to top (on forward oriented edges).
- the first face *faces_begin*( ) in the list of all faces is the unbounded face. This holds for P, P_0, and P_1.

⟨*transfering the marks of supporting objects*⟩≡
```
⟨initialize the outer face object⟩
Vertex_iterator v, vend = vertices_end();
for (v = vertices_begin(); v != vend; ++v) {
  ⟨determine mark of face below v⟩
  ⟨complete marks of vertex v⟩
```

⟨*handle all forward oriented edges starting in v*⟩
}
⟨*transfer the marks to face objects*⟩

The transfer of face support marks is based on the following fact.

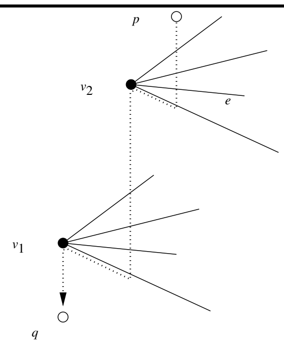


Figure 3.4: The face support iteration unrolled. We examine a position p within the plane map P and try to find the support by a face f_i in the plane map P_i. We have two vertices v_1 and v_2 from V with their forward-oriented edge bundle. The face that supports p with respect to P_i can be determined by following the dotted path until the outer unbounded face of P_i is reached.

Fact 2: Let p be a point of the plane not part of the 1-skeleton of P_i, q be a point within the unbounded face of P_i, and ρ be any curve from p to q not containing any vertex of P_i. Assume ρ intersects an edge e of the 1-skeleton of P_i and let e be the first such edge when following ρ from p to q. Then, p is part of the face incident to e. If ρ does not intersect the 1-skeleton, then p is part of the unbounded face of P_i.

The above fact is a consequence of the connectedness property of the faces of P_i. We now consider point p as part of P. For p we consider a special path ρ as depicted in Figure 3.4. We walk down along a vertical ray (in direction of the negative y-axis). If we cross a bundle of edges incident to a vertex v the path turns just below the lowest edge and follows the lowest edge in parallel until it is just below v. We iterate this construction until it ends in a point q in the unbounded face of P. Each edge e that is crossed by ρ is supported by an edge either from P_i or from P_{1-i}. In the former case the first such edge determines the face f_i supporting p. If there is no such edge then p is supported by the unbounded face of P_i. We want to determine face support for many vertices and edges, thus we do not want to pay such a walk for each query point p. Instead we associate with each edge e face support knowledge in the two slots $mark(e,i)$ and $incident\_mark(e,i)$. The idea is that these slots store the knowledge obtained from a reversal walk from q to p. Whenever our path ρ crosses an edge e in P that is supported by an edge e_i in P_i, then we associate the mark knowledge plus the mark of the supporting faces from P_i (of a small neighborhood left and right of e) with e. If the information is already constructed for all edges below a query point p, we can obtain the support information in constant time.

We now come to the coding. We want to complete the support marks for a vertex v and the edges e of the adjacency list of v that are forward oriented[4]. Consider to follow ρ reversely with a pen starting in q. Then $m\_below[2]$ always stores the marks of the faces of P_i that support the position of the pen.

[4]Backward oriented edges have forward oriented twins.

In the beginning $m\_below[i]$ stores the mark of the face f_i below v. Note that we obtain both marks for $i = 0,1$ either from the outer input faces surrounding the plane maps P_i or from the halfedge below v. If *e_below* exists then it was already treated as a forward oriented edge of a vertex already handled in the vertex iteration.

⟨*initialize the outer face object*⟩≡
```
Face_iterator f = faces_begin(); assoc_info(f);
for (i=0; i<2; ++i) mark(f,i) = PI[i].mark(PI[i].faces_begin());
```

Note that the iteration over all vertices v has the invariant that either v has no halfedge below, or if it has a halfedge *e_below* then *e_below* has all marks correctly assigned ($mark(e\_below, i)$ and $incident\_mark(e\_below, i)$ are set for both $i = 0,1$). Note that each vertex v of P knows the halfedge below it, thus the face support marks can be initialized in constant time.

⟨*determine mark of face below v*⟩≡
```
Halfedge_handle e_below = halfedge_below(v);
Mark m_below[2];
if ( e_below != Halfedge_handle() ) {
  for (int i=0; i<2; ++i) {
    m_below[i] = incident_mark(e_below,i);
  }
} else { // e_below does not exist
  for (int i=0; i<2; ++i)
    m_below[i] = PI[i].mark(PI[i].faces_begin());
}
```

If the vertex v is not supported by a skeleton object of P_i then it is supported by a face. We obtain the mark of the face from *m_below* in this case.

⟨*complete marks of vertex v*⟩≡
```
for (i=0; i<2; ++i)
  if ( supp_halfedge(v,i) != Halfedge_const_handle() ) {
    mark(v,i) = PI[i].mark(supp_halfedge(v,i));
  } else if ( supp_vertex(v,i) != Vertex_const_handle() ) {
    mark(v,i) = PI[i].mark(supp_vertex(v,i));
  } else {
    mark(v,i) = m_below[i];
  }
```

We have to complete the mark information for all edges of P. We do the job for all forward oriented edges in the adjacency list of each vertex v. How does a halfedge e of P obtain mark information with respect to the two input structures P_i? We just have to determine the supporting objects (edge or face) from each of both. It is either supported by two overlapping edges e_0, e_1 or only supported by one edge e_i and one face f_{1-i}. Note that a supporting edge e_i allows access to its mark and to the two faces incident to it and its twin. The supporting edge e_i of e can be obtained via $supp\_halfedge(e,i)$. If e is not supported by an edge in P_i then the mark of the input face can be obtained from $m\_below[i]$. Each supporting input edge e_i of e changes $m\_below[i]$ for the next output edge in the bundle iteration. If e is not supported by an edge in P_i then the supporting face determines the mark of e and the two *incident_mark* entries. The invariant for all edges e in the iteration below is: if e is not supported by an edge e_i of P_i then *m_below*$[i]$ contains the mark of the face supporting e in P_i.

⟨*handle all forward oriented edges starting in v*⟩≡

```
if ( is_isolated(v) ) continue;
Halfedge_around_vertex_circulator
  e(first_out_edge(v)), hend(e);
CGAL_For_all(e,hend) {
  if ( is_forward(e) ) {
    Halfedge_const_handle ei;
    bool supported;
    for (int i=0; i<2; ++i) {
      supported = ( supp_halfedge(e,i) != Halfedge_const_handle() );
      if ( supported ) {
        ei = supp_halfedge(e,i);
        incident_mark(twin(e),i) =
          PI[i].mark(PI[i].face(PI[i].twin(ei)));
        mark(e,i) = PI[i].mark(ei);
        incident_mark(e,i) = m_below[i] =
          PI[i].mark(PI[i].face(ei));
      } else { // no support from input PI[i]
        incident_mark(twin(e),i) = mark(e,i) = incident_mark(e,i) =
          m_below[i];
      }
    }
  } else break;
}
```

The last chunk of this section transfers the support marks to the face object. For all bounded faces f we just transfer the marks from the bounding face cycle to the face. As all edges e carry the *incident_mark*(e,i) attribute this completes the structure.

⟨*transfer the marks to face objects*⟩≡

```
for (f = ++faces_begin(); f != faces_end(); ++f) { // skip first face
  assoc_info(f);
  for (i=0; i<2; ++i) mark(f,i) = incident_mark(halfedge(f),i);
}
```

We can now summarize the calculated overlay properties, we anticipate the costs of face creation as described in the next section and the analysis of the sweep description in Lemma 2.1.3.

Lemma 3.2.3: Assume that P_0 and P_1 are plane maps whose embedding is order-preserving and the adjacency lists have a forward prefix, then *subdivide*(P_0,P_1) constructs in $P=(V,E,F)$ the overlay plane map of P_0 and P_1 and each object $u \in V \cup E \cup F$ carries the mark information *mark*(u,i) from the corresponding supporting object of the input plane map.

Let n_i be the size of P_i and n be the size of P. Then the runtime of the overlay process is dominated by the plane sweep of the skeleton objects of P_0 and P_1 and is therefore $O((n_0+n_1+n)\log(n_0+n_1+n))$.

3.2.5 Creating face objects

Input to this section is the 1-skeleton of a plane map $P' = (V,E)$ whose embedding is *order-preserving* and whose adjacency lists have a *forward-prefix*. The objective of this section is to *create the face objects* that complete P'. The correct output structure $P = (V,E,F)$ of this section is a plane map with the property that there are face objects f in F corresponding to maximal connected point sets which are a result of the partitioning of the plane by the 1-skeleton P'. All faces are defined via their bounding face cycles. Each face object has one halfedge link into the one unique outer face cycle (if existing), a list of halfedges each of which represents interior hole face cycles and a list of isolated vertices that represent trivial face cycles. To assign face cycles to face objects we need to know two properties of the plane map skeleton:

- for each face cycle we need to know if it is an outer face cycle or a hole face cycle.
- for two face cycles *fc1* and *fc2* we need to know if we can connect them by a path in the plane which does not cross any other face cycle.

We adapt an idea from [dBvKOS97]. The path connectivity making disjoint face cycles bounding the same face, can be modeled by a vertical visibility graph of the minimal vertices[5] of each face cycle. We create faces and assign face cycles based on this property and transfer P' to P thereby.

Let C be a set of face cycles of the plane map skeleton. For each face cycle c let $\mathit{MinimalHalfedge}[c]$ be the halfedge e whose target vertex has minimal coordinates (lexicographically). Let $\mathit{FaceCycle}[e]$ be the face cycle containing e. We examine the following implicitly defined graph G. Each face cycle of P' is a node of G. Let us link two face cycles c_1 and c_2 by an undirected edge of G if $\mathit{target}(\mathit{MinimalHalfedge}[c_1])$ has a vertical view down to an edge of c_2 (in P). Note that face cycles consist of halfedges and thus we have to refer to the correct one of the two paired halfedges respecting the embedding when looking at face cycles (our faces are left of the directed halfedges, thus consider the bidirected twins to be separated by an infinitesimal distance, then the visibility is uniquely defined). Note that the embedding of a face cycle c at its minimal halfedge gives us the criterion to separate outer face cycles and hole face cycles. Whenever the underlying line segments of $e = \mathit{MinimalHalfedge}[c]$ and $\mathit{next}(e)$ form a left turn c is an outer face cycle. When they form a right turn the vertex $\mathit{target}(e)$ has a free view down and thus e belongs to a hole.

Note that we do not explicitly model the visibility graph. Instead the recursive behavior of the operation *determine_face*( ) used below imitates a DFS walk on the visibility graph. In the following method we have the vertical visibility coded via a data accessor D providing for all vertices $v \in V$ the knowledge about the halfedge below v. $\mathit{D.halfedge\_below}(v)$ either provides the halfedge of E that is hit first by a vertical ray downwards or an uninitialized halfedge if there is none.

The following template type parameter *Below_info* has to fit the concept *Below_info* of the Figures 3.2 and 3.3.

⟨*helping operations*⟩+≡
```
template <typename Below_info>
void create_face_objects(const Below_info& D) const
{
  CGAL::Unique_hash_map<Halfedge_handle,int> FaceCycle(-1);
  std::vector<Halfedge_handle>  MinimalHalfedge;
  ⟨link halfedges to face cycles and determine minimal halfedges⟩
```

[5] minimal with respect to the lexicographic order of the point coordinates of their embedding

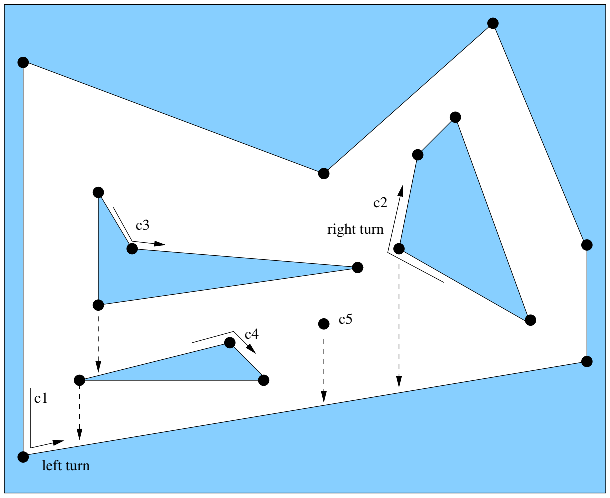


Figure 3.5: Face cycles bounding a face. c1 is the outer face cycle, c2, c3, and c4 are hole cycles, c5 is an isolated vertex. The minimal vertices of each face cycle are the origins of the dashed vertical arrows down.

```
    ⟨create face objects for outer face cycles and create links⟩
    ⟨link holes and isolated vertices to face objects⟩
  }
```

We iterate all halfedges and assign a number for each face cycle. After the iteration for a halfedge e the number of its face cycle is $FaceCycle[e]$ and for a face cycle c we know $MinimalHalfedge[c]$.

⟨*link halfedges to face cycles and determine minimal halfedges*⟩≡

```
  int i=0;
  Halfedge_iterator e, eend = halfedges_end();
  for (e=halfedges_begin(); e != eend; ++e) {
    if ( FaceCycle[e] >= 0 ) continue; // already assigned
    Halfedge_around_face_circulator hfc(e),hend(hfc);
    Halfedge_handle e_min = e;
    CGAL_For_all(hfc,hend) {
      FaceCycle[hfc]=i; // assign face cycle number
      if ( K.compare_xy(point(target(hfc)), point(target(e_min))) < 0 )
        e_min = hfc;
    }
    MinimalHalfedge.push_back(e_min); ++i;
  }
```

We now know the number of face cycles i and we have a minimal halfedge e for each face cycle. We just check the geometric embedding of e and $next(e)$ to characterize the face cycle (outer or hole). Note that the two edges cannot be collinear due to the minimality of e (the lexicographic minimality of the embedding of its target vertex). Outer face cycles obtain face objects right away. Hole cycles whose *halfedge_below* information is undefined are associated with the unique outer face. After this chunk *f_outer* is the first face object *faces_begin*( ) in the list of all face objects, and all outer face cycles have face objects with temporary mark information slots expanded.

⟨*create face objects for outer face cycles and create links*⟩≡

```
Face_handle f_outer = new_face();
for (int j=0; j<i; ++j) {
  Halfedge_handle e = MinimalHalfedge[j];
  Point p1 = point(source(e)),
        p2 = point(target(e)),
        p3 = point(target(next(e)));
  if ( K.leftturn(p1,p2,p3) ) { // leftturn => outer face cycle
    Face_handle f = new_face();
    link_as_outer_face_cycle(f,e);
  }
}
```

Now, the only halfedges not linked are those on hole face cycles. We use a recursive scheme to find the bounding cycle providing the face object and finally iterate over all isolated vertices to link them accordingly to their containing face object. Note that in this final iteration all halfedges already have face links. This ensures termination. The recursive operation *determine_face*$(e,\ldots)$ returns the face containing the hole cycle of e (see the specification in the next section). As a postcondition of this chunk we have all edges and isolated vertices linked to face objects, and all face objects know their bounding face cycles.

⟨*link holes and isolated vertices to face objects*⟩≡

```
for (e = halfedges_begin(); e != eend; ++e) {
  if ( face(e) != Face_handle() ) continue;
  Face_handle f = determine_face(e,MinimalHalfedge,FaceCycle,D);
  link_as_hole(f,e);
}
Vertex_iterator v, v_end = vertices_end();
for (v = vertices_begin(); v != v_end; ++v) {
  if ( !is_isolated(v) ) continue;
  Halfedge_handle e_below = D.halfedge_below(v);
  if ( e_below == Halfedge_handle() )
    link_as_isolated_vertex(f_outer,v);
  else
    link_as_isolated_vertex(face(e_below),v);
}
```

When we call *determine_face*$(e,\ldots)$ we know that the halfedge e is not yet linked to a face object and thus, no halfedge in its face cycle is linked. Thus we jump to the minimal halfedge and look down. If we see nirvana then we have to link the unlimited face *f_outer*. If we see a halfedge we ask for its face. If it does not have one we recurse. Note that the target vertex of the minimal halfedge actually has a view downwards as we examine a hole face cycle. The method *link_as_hole* does the linkage between the face object and all edges of the face cycle. Its cost is linear in the size of the face cycle. Note also that we do the linking bottom up along the recursion stack for all visited hole cycles. Thus, we visit each hole face cycle only once as afterwards each edge of the face cycle is incident to a face.

Look at our example in Figure 3.5. When *determine_face* is called for an edge e of face cycle c3, then the procedure first finds an edge of c4. If c4 was not visited yet by an earlier call, then the method recurses to c4 before it finds the correct face object via the outer face cycle c1.

⟨*helping operations*⟩+≡
```
template <typename Below_info>
Face_handle determine_face(Halfedge_handle e,
  const std::vector<Halfedge_handle>& MinimalHalfedge,
  const CGAL::Unique_hash_map<Halfedge_handle,int>& FaceCycle,
  const Below_info& D) const
{
  Halfedge_handle e_min = MinimalHalfedge[FaceCycle[e]];
  Halfedge_handle e_below = D.halfedge_below(target(e_min));
  if ( e_below == Halfedge_handle() ) // below is nirwana
    return faces_begin();
  Face_handle f = face(e_below);
  if (f != Face_handle()) return f; // has face already
  f = determine_face(e_below, MinimalHalfedge, FaceCycle,D);
  link_as_hole(f,e_below);
  return f;
}
```

The explanations of the recursion condition of *determine_face* should convince you that:

Lemma 3.2.4: Assume that P' is the 1-skeleton of a plane map whose embedding is order-preserving and the adjacency lists have a forward prefix. Let additionally all vertices know the halfedge visible along a vertical ray shot down, then *create_face_objects*( ) completes P as a plane map with runtime *linear* in the size of the 1-skeleton P'.

3.2.6 Selecting marks

For the selection we just iterate over all objects, read the marks refering to the two input structures, apply our selection operation, and store the mark back into the object. At this place, we discard the additional information which was accumulated during the subdivision. The flexibility of the operation is achieved by a template type parameter *Selection*. An object *predicate* of type *Selection* must provide a binary function operator returning a new mark object. The runtime of the selection phase is obviously linear in the size of the plane map P. The method *discard_info* just discards the temporarily allocated information containers associated to the objects.

⟨*selection*⟩≡
```
template <typename Selection>
void select(Selection& predicate) const
{
  Vertex_iterator vit = vertices_begin(),
                  vend = vertices_end();
  for( ; vit != vend; ++vit) {
    mark(vit) = predicate(mark(vit,0),mark(vit,1));
    discard_info(vit);
  }
  Halfedge_iterator hit = halfedges_begin(),
                    hend = halfedges_end();
  for(; hit != hend; ++(++hit)) {
```

```
      mark(hit) = predicate(mark(hit,0),mark(hit,1));
      discard_info(hit);
    }
    Face_iterator fit = faces_begin(),
                  fend = faces_end();
    for(; fit != fend; ++fit) {
      mark(fit) = predicate(mark(fit,0),mark(fit,1));
      discard_info(fit);
    }
  }
```

Note that after this phase the plane map output has again the input properties of the overlay calculation operation from Section 3.2.4.

Lemma 3.2.5: The selection phase has runtime linear in the size of the plane map.

3.2.7 Simplification of attributed plane maps

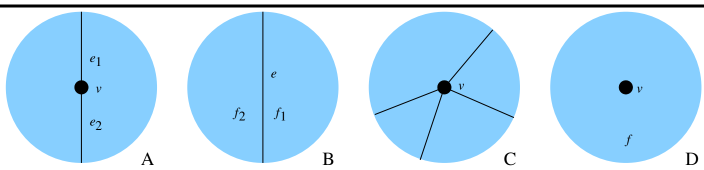


Figure 3.6: The possible configurations for simplification.

In this section we examine the task to simplify a given plane map to reach a minimal representation (minimal number of objects of the plane map structure, while the underlying attributed point set stays the same). There are three situations where one can imagine to simplify the structure (see Figure 3.6):

1. A vertex v which is incident to two edges e_1, e_2 both supported by the same line where all three objects have the same mark can be unified into one edge without changing the stored point set. (Figure 3.6,A)

2. A uedge e which has the same mark as the two faces f_1 and f_2 incident to it does not contribute any structural information and thus can be removed (Figure 3.6,B).

3. A vertex v where all the edges of its adjacency list and also all incident faces have the same mark as the vertex also carries no structural information (Figure 3.6,C,D).

If we first remove edges of the second case then the vertices of case three have no incident edges at all and thus can be easily identified as isolated vertices whose surounding face has the same mark. The first case does only play a role if one of the faces incident to the edge carries a different mark than the edge.

We can thus easily formulate the simplification routine. However, there are some problems with the update operations of the plane map structure. How can we maintain the face objects and incidence links to halfedges and vertices if we are unifying faces by deleting edges? The trivial way does not work within our time bound. We cannot afford to maintain the face objects in a correct status in each step of the simplification, as this would mean to repeatedly iterate total face cycles.

Note that we cannot just discard all faces and recreate them using a similar scheme as the one based on the *halfedge_below* information due to the fact that referenced edges might be deleted in the simplification process. Thereby, face creation as described in Section 3.2.5 is not possible without a new sweep. We take a different approach. We use a unification history stored in a partition data structure instead of the geometrically defined *halfedge_below* information as a criterium for linking face cycles to face objects.

All face cycles (edges and isolated vertices) reference face objects. When we have to unify two different faces due to the deletion of an edge separating them, we store this fact by a union operation in a partition data structure. The face that is finally assigned to all the face cycles of the faces in one block is the one associated with the canonical item of the block (obtained by the find operation).

⟨*simplification*⟩≡

```
template <typename Keep_edge>
void simplify(const Keep_edge& keep) const
{
  typedef typename CGAL::Partition<Face_handle>::item partition_item;
  CGAL::Unique_hash_map<Face_iterator,partition_item> Pitem;
  CGAL::Partition<Face_handle> FP;
  ⟨initialize blocks corresponding to faces⟩
  ⟨simplify via non-separating halfedges⟩
  ⟨recollect face cycles per blocks⟩
  ⟨simplify via vertices⟩
  ⟨remove superflous face objects⟩
}
```

We assign one partition item to each face object and make the item accessible to the face via a hash map. During the assignment of face cycles to face objects we will only use links from skeleton objects like vertices and edges to faces. We therefore can discard all face cycle entries in the faces (the links from face objects to skeleton objects).

⟨*initialize blocks corresponding to faces*⟩≡

```
Face_iterator f, fend = faces_end();
for (f = faces_begin(); f!= fend; ++f) {
   Pitem[f] = FP.make_block(f);
   clear_face_cycle_entries(f);
}
```

Now we take care of the simplification critereon (2.) of page 131. We only iterate halfedge pairs (uedges). When the marks of the incident faces agree with the mark of the uedge, we union the items of the faces if they are different. Special treatment is required for incident vertices if they become isolated when their last icident uedge is deleted.

⟨*simplify via non-separating halfedges*⟩≡

```
Halfedge_iterator e = halfedges_begin(), en,
                  eend = halfedges_end();
for(; en=e, ++(++en), e != eend; e=en) {
```

```
    if ( keep(e) ) continue;
    if ( mark(e) == mark(face(e)) &&
         mark(e) == mark(face(twin(e))) ) {
      if ( !FP.same_block(Pitem[face(e)],
                          Pitem[face(twin(e))]) ) {
        FP.union_blocks( Pitem[face(e)],
                         Pitem[face(twin(e))] );
      }
      if ( is_closed_at_source(e) )       set_face(source(e),face(e));
      if ( is_closed_at_source(twin(e)) ) set_face(target(e),face(e));
      delete_halfedge_pair(e);
    }
  }
```

Now we recollect all face cycles and assign them to the face object f that refers to the partition item obtained by a find operation. In each face cycle we determine the halfedge *e_min* whose target has a minimal embedding (with respect to the lexicographic order on points). If *e_min* and *next*(*e_min*) form a left turn they are part of an outer face cycle, otherwise of a hole face cycle. We associate all edges in the face cycle with f.

⟨*recollect face cycles per blocks*⟩≡

```
  CGAL::Unique_hash_map<Halfedge_handle,bool> linked(false);
  for (e = halfedges_begin(); e != eend; ++e) {
    if ( linked[e] ) continue;
    Halfedge_around_face_circulator hfc(e),hend(hfc);
    Halfedge_handle e_min = e;
    Face_handle f = FP.inf(FP.find(Pitem[face(e)]));
    CGAL_For_all(hfc,hend) {
      set_face(hfc,f);
      if ( K.compare_xy(point(target(hfc)), point(target(e_min))) < 0 )
        e_min = hfc;
      linked[hfc]=true;
    }
    Point p1 = point(source(e_min)),
          p2 = point(target(e_min)),
          p3 = point(target(next(e_min)));
    if ( K.orientation(p1,p2,p3) > 0 ) set_halfedge(f,e_min); // outer
    else set_hole(f,e_min); // store as inner
  }
```

After the previous simplification we still have to take care of the vertex related simplifications (1.) and (3.). In case that a vertex has outdegree two, that the two incident edges are embedded collinearly, and that all three objects have the same mark, we remove the vertex by joining the two uedges into one. In case that a vertex is isolated and its mark agrees with the incident face we remove the vertex. Otherwise, we anchor the vertex in the face by adding it to the isolated vertex list. Note that the face link of each isolated vertex was either already set in the face creation phase, or in the chunk ⟨*simplify via non-separating halfedges*⟩ when the last incident edge was deleted.

⟨*simplify via vertices*⟩≡
```
Vertex_iterator v, vn, vend = vertices_end();
for(v = vertices_begin(); v != vend; v=vn) {
  vn=v; ++vn;
  if ( is_isolated(v) ) {
    if ( mark(v) == mark(face(v)) ) delete_vertex_only(v);
    else set_isolated_vertex(face(v),v);
  } else { // v not isolated
    Halfedge_handle e2 = first_out_edge(v), e1 = previous(e2);
    Point p1 = point(source(e1)), p2 = point(v),
          p3 = point(target(e2));
    if ( has_outdeg_two(v) &&
         mark(v) == mark(e1) && mark(v) == mark(e2) &&
         (K.orientation(p1,p2,p3) == 0) )
      merge_halfedge_pairs_at_target(e1);
  }
}
```

Finally we discard all face objects that have been victims of unification but do not represent the unified face.

⟨*remove superflous face objects*⟩≡
```
Face_iterator fn;
for (f = faces_begin(); f != fend; f=fn) {
  fn=f; ++fn;
  partition_item pit = Pitem[f];
  if ( FP.find(pit) != pit ) delete_face(f);
}
```

The following operations just wrap some basic primitives which make our code more readable.

⟨*helping operations*⟩+≡
```
Segment segment(const Const_decorator& N,
                Halfedge_const_handle e) const
{ return K.construct_segment(
    N.point(N.source(e)),N.point(N.target(e))); }
Segment segment(const Const_decorator& N,
                Vertex_const_handle v) const
{ Point p = N.point(v);
  return K.construct_segment(p,p); }
bool is_forward_edge(const Const_decorator& N,
                     Halfedge_const_iterator hit) const
{ Point p1 = N.point(N.source(hit));
  Point p2 = N.point(N.target(hit));
  return (K.compare_xy(p1,p2) < 0); }
```

The following analysis of the partition data structure is due to Tarjan [Tar83].

Fact 3: A sequence of m union and find operations starting from n singleton blocks can be done in time $O(m\alpha(m,n))$ with a partition data structure that is based on union by rank and path compression. In this time bound α is the very slowly growing inverse of a suitably defined Ackermann function.

We can therefore summarize the runtime of the simplification action.

Lemma 3.2.6: Assume that P is a plane map with the properties cited in the introduction of this section. Then the method *simplify*( ) runs in time $O(n\,\alpha(kn,n))$ where n is the size of P, kn is a bound for the number of face unifications and find operations, and α is the function mentioned above.

Proof. The number of edges and faces of P is linear in n. The number of union operations is bounded by the number of faces, and the number of find operations is bounded by three times[6] the number of edges plus the number of faces. □

Note that after the simplification the plane map output has again the input properties of the overlay calculation operation from Section 3.2.4.

⟨*helping operations*⟩+≡
```
void assert_type_precondition() const
{ typename PM_decorator_::Point p1; Point p2;
  assert_equal_types(p1,p2); }
```

⟨*PM_overlayer.h*⟩≡
```
⟨CGAL Header⟩
#ifndef CGAL_PM_OVERLAYER_H
#define CGAL_PM_OVERLAYER_H

#include <CGAL/basic.h>
#include <CGAL/Unique_hash_map.h>
#include <CGAL/Partition.h>
#include <CGAL/Nef_2/Segment_overlay_traits.h>
#include <CGAL/Nef_2/geninfo.h>
#undef _DEBUG
#define _DEBUG 13
#include <CGAL/Nef_2/debug.h>

#ifndef CGAL_USE_LEDA
#define LEDA_MEMORY(t)
#endif

CGAL_BEGIN_NAMESPACE

⟨PM traits classes for segment overlay⟩
⟨PM overlayer⟩

CGAL_END_NAMESPACE
#endif // CGAL_PM_OVERLAYER_H
```

3.3 Test snippets

3.3.1 A Demo of the overlayer

⟨*PM_overlayer-demo.C*⟩≡
```
#include <CGAL/basic.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Homogeneous.h>
```

[6] look for the *find*( ) and *same_block*( ) operations above. The latter uses two find operations.

```
#include <CGAL/IO/Window_stream.h>
#include <CGAL/Nef_2/HalfedgeDS_default.h>
#include <CGAL/Nef_2/HDS_items.h>
#include <CGAL/Nef_2/PM_decorator.h>
#include <CGAL/Nef_2/PM_io_parser.h>
#include <CGAL/Nef_2/PM_overlayer.h>
#include <CGAL/Nef_2/PM_visualizor.h>
#include <CGAL/test_macros.h>
#include "Affine_geometry.h"

// GEOMETRY:
typedef CGAL::Homogeneous<leda_integer> Hom_kernel;
typedef CGAL::Affine_geometry<Hom_kernel> Aff_kernel;
typedef Aff_kernel::Segment_2 Segment;

// PLANE MAP:
struct HDS_traits {
  typedef Aff_kernel::Point_2 Point;
  typedef bool Mark;
};

typedef  HalfedgeDS_default<HDS_traits,HDS_items> HDS;
typedef  CGAL::PM_decorator< HDS >                PM_dec;
typedef  CGAL::PM_overlayer< PM_dec, Aff_kernel > PM_aff_overlayer;
typedef  PM_dec::Halfedge_handle Halfedge_handle;
typedef  PM_dec::Vertex_handle   Vertex_handle;

// INPUT:
typedef  std::list&lt;Segment&gt;::const_iterator Iterator;

struct Object_DA {
  const PM_dec& D;
  Object_DA(const PM_dec& Di) : D(Di) {}
  void supporting_segment(Halfedge_handle e, Iterator it) const
  { D.mark(e) = true; }
  void trivial_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
  void starting_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
  void passing_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
  void ending_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
};

struct BOP {
  bool operator()(const bool& b1, const bool& b2) const
  { return b1||b2; }
};

int main(int argc, char* argv[])
{
  SETDTHREAD(331);
  CGAL::set_pretty_mode(cerr);
  CGAL::Window_stream W;
  W.init(-50,50,-50,1);
  W.set_show_coordinates(true);
  W.display();
```

```
  HDS H[2], H0;
  char C[2] = { '0','1' };
  for (int i=0; i<2; ++i ) {
    W.message("insert segments to construct a map.");
    PM_aff_overlayer PMOV(H[i],Aff_kernel());
    CGAL::PM_visualizor<PM_aff_overlayer,Aff_kernel> V(W,PMOV);
    std::list&lt;Segment&gt; L;
    Segment s;
    std::string fname;
    if ( argc == 2 ) {
      fname = std::string(argv[1]);
      fname += C[i];
      std::ifstream log(fname.c_str());
      while ( log >> s ) { L.push_back(s); W << s; }
    }
    while ( W >> s ) L.push_back(s);
    fname = std::string(argv[0]);
    fname += ".log";
    fname += C[i];
    std::ofstream log(fname.c_str());
    for (Iterator sit = L.begin(); sit != L.end(); ++sit)
      log << *sit << " ";
    log.close();
    Object_DA ODA(PMOV);
    PMOV.create(L.begin(),L.end(),ODA);
    CGAL::PM_io_parser<PM_aff_overlayer>::dump(PMOV);
    V.draw_skeleton();
    W.read_mouse();
    W.clear();
  }
  PM_aff_overlayer PMOV(H0,Aff_kernel());
  CGAL::PM_visualizor<PM_aff_overlayer,Aff_kernel> V(W,PMOV);
  PMOV.subdivide(H[0],H[1]);
  BOP bop;
  PMOV.select(bop);
  PMOV.simplify();
  V.draw_skeleton();
  CGAL::PM_io_parser<PM_aff_overlayer>::dump(PMOV);
  W.read_mouse();
  return 0;
}
```

3.3.2 A Test of the overlayer

⟨*PM_overlayer-test.C*⟩≡

```
#include <CGAL/basic.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/IO/Window_stream.h>
#include <CGAL/Nef_2/HalfedgeDS_default.h>
#include <CGAL/Nef_2/HDS_items.h>
```

```
#include <CGAL/Nef_2/PM_decorator.h>
#include <CGAL/Nef_2/PM_io_parser.h>
#include <CGAL/Nef_2/PM_overlayer.h>
#include <CGAL/Nef_2/PM_visualizor.h>
#include <CGAL/test_macros.h>
#include "Affine_geometry.h"
// GEOMETRY:
typedef CGAL::Homogeneous<leda_integer> Hom_kernel;
typedef CGAL::Affine_geometry<Hom_kernel> Aff_kernel;
typedef Aff_kernel::Point_2   Point;
typedef Aff_kernel::Segment_2 Segment;
// PLANE MAP:
struct HDS_traits {
  typedef Aff_kernel::Point_2 Point;
  typedef bool Mark;
};
typedef  CGAL::HalfedgeDS_default<HDS_traits,HDS_items> HDS;
typedef  CGAL::PM_decorator< HDS >                PM_dec;
typedef  CGAL::PM_overlayer< PM_dec, Aff_kernel > PM_aff_overlayer;
typedef  PM_dec::Halfedge_handle Halfedge_handle;
typedef  PM_dec::Vertex_handle   Vertex_handle;
// INPUT:
typedef  std::list&lt;Segment&gt;::const_iterator Iterator;
struct Object_DA {
  const PM_dec& D;
  Object_DA(const PM_dec& Di) : D(Di) {}
  void supporting_segment(Halfedge_handle e, Iterator it) const
  { D.mark(e) = true; }
  void trivial_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
  void starting_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
  void passing_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
  void ending_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
};
struct BOP {
  bool operator()(const bool& b1, const bool& b2) const
  { return b1||b2; }
};
int main(int argc, char* argv[])
{
  SETDTHREAD(131); // 13 = PM overlayer debug on
  CGAL::set_pretty_mode(cerr);
  CGAL_TEST_START;
//#define PMOVISUAL
#ifdef PMOVISUAL
  CGAL::Window_stream W;
  W.init(-50,50,-50,1);
  W.set_show_coordinates(true);
```

```
  W.display();
#endif
  HDS H[2], H0, H0C;
  std::list&lt;Segment&gt; L0,L1,L2;
  L0.push_back(Segment(Point(-5,-5),Point(5,-5)));
  L0.push_back(Segment(Point(5,-5),Point(5,5)));
  L0.push_back(Segment(Point(5,5),Point(-5,5)));
  L0.push_back(Segment(Point(-5,5),Point(-5,-5)));
  L0.push_back(Segment(Point(0,0),Point(0,0)));
  PM_aff_overlayer PMOV0(H[0],Aff_kernel());
  Object_DA ODA0(PMOV0);
  PMOV0.create(L0.begin(),L0.end(),ODA0);
  //CGAL::PM_io_parser<PM_aff_overlayer>::dump(PMOV0);
  L1.push_back(Segment(Point(-5,-5),Point(0,-5)));
  L1.push_back(Segment(Point(0,-5),Point(0,0)));
  L1.push_back(Segment(Point(0,0),Point(-5,0)));
  L1.push_back(Segment(Point(-5,0),Point(-5,-5)));
  L1.push_back(Segment(Point(-2,-2),Point(-2,-2)));
  PM_aff_overlayer PMOV1(H[1],Aff_kernel());
  Object_DA ODA1(PMOV1);
  PMOV1.create(L1.begin(),L1.end(),ODA1);
  //CGAL::PM_io_parser<PM_aff_overlayer>::dump(PMOV1);
  PM_aff_overlayer PMOV(H0,Aff_kernel());
  PMOV.subdivide(H[0],H[1]);
  BOP bop;
  PMOV.select(bop);
  PMOV.simplify();
  L2.insert(L2.end(),L0.begin(),L0.end());
  L2.insert(L2.end(),L1.begin(),L1.end());
  PM_aff_overlayer PMOV2(H0C,Aff_kernel());
  Object_DA ODA2(PMOV2);
  PMOV2.create(L2.begin(),L2.end(),ODA2);
  CGAL_TEST(PMOV.number_of_vertices()==8);
  CGAL_TEST(PMOV.number_of_edges()==8);
  CGAL_TEST(PMOV.number_of_faces()==3);
#ifdef PMOVISUAL
  CGAL::PM_visualizor<PM_aff_overlayer,Aff_kernel> V(W,PMOV);
  CGAL::PM_io_parser<PM_aff_overlayer>::dump(PMOV);
  V.draw_skeleton();
  W.read_mouse();
#endif
  CGAL_TEST_END;
  return 0;
}
```

Bibliography

[dBvKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry : Algorithms and Applications*. Springer, Berlin, 1997.

[MN99] K. Mehlhorn and S. Näher. *LEDA, A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MS01] K. Mehlhorn and M. Seel. Implementation of planar nef polyhedra. Technical Report MPI-I-2001-1-003, Max-Planck-Institut für Informatik, Saarbrücken, 2001.

[Tar83] R.E. Tarjan. *Data structures and network algorithms*, volume 44. SIAM, 6th pr. 1991 edition, 1983.

4 Constrained Triangulations

4.1 Introduction

We first introduce the notions or triangulation and constraining segments. Then we present the algorithmic ideas to construct a constrained triangulation by a plane sweep algorithm. In the next section the concrete implementation is shown.

Definition 6 (Triangulation [PS85]): A planar subdivision is a triangulation T if all its bounded regions are triangles. A triangulation of a finite set S of points is a planar graph $T(S)$ with the maximum number of edges, which is equivalent to saying that $T(S)$ is obtained by joining the points of S by non-intersecting straight line segments so that every region internal to the convex hull of S is a triangle.

Definition 7 (Constrained Triangulation): Let S be a finite set of segments which have no pairwise common points except in their endpoints. A constrained triangulation $CT(S)$ is a triangulation of the endpoints of the segments in S such that each non-trivial segment of S corresponds to an edge in $CT(S)$. Note that we allow trivial segments.

We sweep the edges and vertices of of a plane map $G = (V,E)$ to compute the constrained triangulation $CT(G) := CT(S = \mathit{segment}(E) \cup \mathit{point}(V))$[1]. Actually we extend G_{in} until finally $G_{out} \equiv CT(G_{in})$.

What does this mean? We want to obtain a triangulation of the point set $\mathit{point}(V)$ which additionally obeys the constraints given by the segments $\mathit{segment}(E)$ of the graph. In the following description we identify the vertices with their positions and the edges with the segments spanned by the positions of their end points.

If the graph only contains isolated vertices we calculate the standard triangulation given by the standard sweep triangulation algorithm. (see for example a description in the LEDA book [MN99]). This triangulation has the property that any vertical line l subdivides the triangulation into a correctly constructed part left of it and the rest. (Fixing l just remove all triangles which are intersected by it in their interior or which are right of it). This is just the result of the invariant kept during the sweep.

So what is different if we calculate the constrained triangulation? Our sweepline SL contains all edges (segments) that currently intersect it ordered by their intersection points from bottom to top. We use a sorted sequence to store the edges. Let's conceptually identify the line with the data structure. Note that edges might touch in their endpoints as we work on a correctly embedded plane map. Also the vertices do not come to lie in the interior of a non-adjacent edge. We sweep the graph G and extend it by additional edges to create the triangulation. Each input edge encountered during the sweep from

[1] We slopily write $\mathit{point}(V)$ for the set of all embedding points of the vertices in V and $\mathit{segment}(E)$ for the set of segments spanned by the end vertices of the edges in E.

$-\infty$ to ∞ (along the x-axis) enters SL that stores its position in the sweepline. At the top and the bottom of SL we use sentinel segments to avoid the handling of boundary cases. Note that with the above idea all edges in the sweepline are already connected via face cycles (paths) in the extended graph. At the end of the sweep we remove the sentinel edges.

What is our basic invariant? We have already calculated a correct constrained triangulation of all vertices and edges that are fully left of the sweepline including a closure implied by the sweepline. Let's first assume that we don't face degeneracies like equal x-coordinates or vertical segments. We will comment on the removal of this restriction later.

Definition 8 (Restricted Constrained Triangulation): Let SL be the vertical sweepline defined by an event point p in the plane. Let $S[< SL] \subseteq S$ be the finite set of segments already handled completely (all vertices and edges of our input graph fully left of SL). Let $S[SL]$ be the set of segments spanned by edges intersecting SL, but restricted to the closed halfplane left of SL. We define the restricted contrained triangulation of the edges encountered so far to be $CT[\leq SL] := CT(S[< SL] \cup S[SL])$. Accordingly we talk about the plane map $G[\leq SL]$ restricted to the closed halfplane left of SL, consisting of all vertices in the closed halfplane and edges connecting them.

Note that $CT[\leq SL]$ is a valid constrained triangulation that has a part of its hull on SL. We store this triangulation in a special way. All triangulation edges which are finished are already stored in our output graph $G[\leq SL]$. Finished here means that they connect vertices in $G[\leq SL]$ totally left of SL. The missing edges completing $G[\leq SL]$ to $CT(S[\leq SL])$ can be seen as stored implicitly in our structures. We will explain this in a moment.

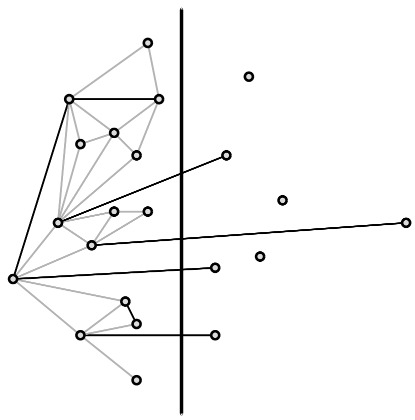


Figure 4.1: The triangles already finished during a sweep. Constraining segments are black. Additional triangulation edges are grey.

When looking at the actions at a sweep event we'll see that we extend $G[\leq SL]$ in a way that it always represents a maximal subgraph of the final triangulation. It holds for example that $G[\leq SL]$ is connected. Each edge e in SL is connected in $G[\leq SL]$ to the edge e_s representing the successor edge in SL. Actually there's a face cycle $C = e_1, \ldots, e_k$ where $e_1 = next(twin(e_s))$ and $e_k = previous(e)$ in our bidirected representation of G which proves connectivity. See figure 4.2. This chain of edges has the additional property that it can be split into two x-monotone parts $C_1 = e_1, \ldots, e_i$ and $C_2 = e_{i+1}, \ldots, e_k$ where the vertex $v_{vis} = source(e_{i+1})$ is a point of maximal x-coordinate visible by any point on SL

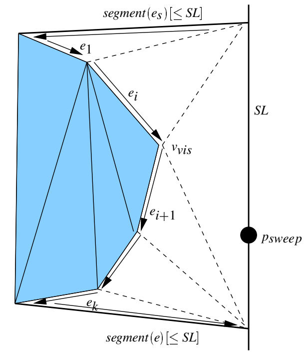


Figure 4.2: The chain between two segments in SL at a point p.

between e and its successor e_s in SL. Moreover the slope of the lines that support the edges in C_1 and C_2 is non-increasing.

We associate the edge $e_{vis} = e_{i+1}$ with source v_{vis} to e to allow fast access into the chain for any pair of neigbored segments in SL. By e_{vis} we have a starting point for a visibility search along the x-monotone chains in which v_{vis} is an extreme vertex.

We come back to our representation of $CT(S[\leq SL])$. $CT(S[\leq SL])$ consists of the explicitly constructed part $G[\leq SL]$ and an implicitly possible triangulation between each pair of edges e, e_s as depicted dashed in figure 4.2. Just connect $target(segment(e)[\leq SL])$ to all vertices of C_2 and symmetrically $target(segment(e_s)[\leq SL])$ to all vertices of C_1 and both via SL.

We will show for the event handling procedure that we keep the explicit and implicit structure consistent. Thus when reaching the end of our scenery where only our global sentinel edges are present (which frame the whole scenery) then the explicit part of G contains the correct constrained triangulation of the input.

Invariants

We recapitulate our implementation invariants:

1. All edges intersecting the vertical line through the current event point are stored in SL ordered according to their intersection points bottom-up. We create hash links from these edges to their item in SL realized by a hash map *SLItem*. This map serves also as a flag of the edges intersected by the sweepline. We set this flag when entering and reset it to the default when leaving SL.

2. For two edges e, e_s that are neighbors in SL we maintain the chain of edges $C = e_1, \ldots, e_k$ stored as a face cycle in the output graph where $source(e_1) == source(e_s)$ and $target(e_k) == source(e)$. When closing $G[\leq SL]$ with our implicit construction, then we obtain a valid constrained triangulation of the objects in the closed halfspace left of SL.

3. Each edge e in SL knows a vertex v_{vis} visible by any point on SL between e and its successor e_s in SL. We realize this knowledge by associating an edge $e_{vis} \in C$ to it with source v_{vis}. By e_{vis} we have a starting point for a visibility search along the chain C in which v_{vis} is an extreme vertex.

All kinds of degeneracies like equal x-coordinates and vertical segments can be integrated into the code if you imagine to twist the whole scenery clockwise by an infinitesimal angle. This implies that vertices are ordered lexicographically and vertical segments belong to a starting bundle of edges. Segments touching in one point are handled explicitly by an iteration over all edges starting in a vertex of the input graph.

4.2 Implementation

We use our generic sweep framework but open the algorithm furtheron by three template parameters of the sweep traits class *Constrained_triang_traits*<>. See the concept *GenericSweepTraits* that has to be implemented in the appendix section 11.1.6. We want to factor out the manipulation of the plane map, the geometric types and predicates used and the action which can be invoked on the additionally created triangulation edges in *CT*. Thus we add a concept for a plane map decorator *PMDEC*, a concept for a geometric kernel *GEOM* and a concept for a new-edge data accessor *NEWEDGE*. Note that the class provides the types and members according to the generic plane sweep traits concept *GenericSweepTraits*.

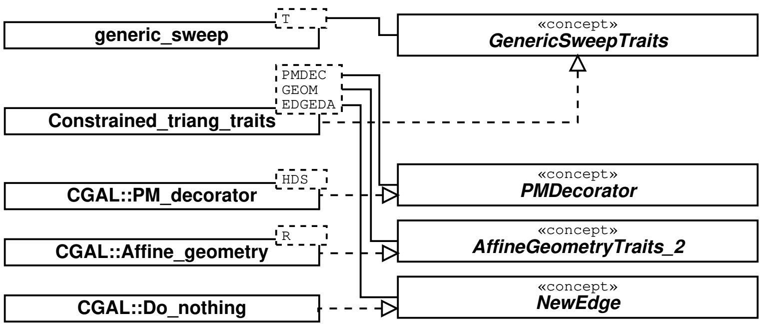


Figure 4.3: The design of the constrained triangulation module. *Constrained_triang_traits*<> implements the concept *GenericSweepTraits*. The three template parameters allow an adaptation of the input/output plane map, the geometry, and the processing of new edges.

⟨*class Constrained_triang_traits*⟩≡

```
template <typename PMDEC, typename GEOM,
          typename NEWEDGE = Do_nothing>
class Constrained_triang_traits : public PMDEC {
public:
  ⟨type definition for the class scope⟩
  ⟨order predicate definition⟩
  ⟨local types for the sweep⟩
  ⟨helping operations⟩
  ⟨ct event handling⟩
  ⟨ct initialization⟩
  ⟨ct cleaning up⟩
  ⟨ct checking⟩
}; // Constrained_triang_traits<PMDEC,GEOM,NEWEDGE>
```

⟨*Constrained_triang_traits.h*⟩≡
```
⟨CGAL Header1⟩
// file          : include/CGAL/Nef_2/Constrained_triang_traits.h
⟨CGAL Header2⟩
#ifndef CGAL_PM_CONSTR_TRIANG_TRAITS_H
#define CGAL_PM_CONSTR_TRIANG_TRAITS_H

#include <CGAL/basic.h>
#include <CGAL/Unique_hash_map.h>
#include <CGAL/generic_sweep.h>
#include <CGAL/Nef_2/PM_checker.h>
#include <string>
#include <map>
#include <set>
#undef _DEBUG
#define _DEBUG 19
#include <CGAL/Nef_2/debug.h>

⟨some predefinitions⟩
⟨class Constrained_triang_traits⟩

#endif // CGAL_PM_CONSTR_TRIANG_TRAITS_H
```

The geometry concept *GEOM* allows us to import the geometric types into *Constrained_triang_traits* and offers the necessary predicates as methods. See *AffineGeometryTraits_2* in the appendix for the concept description. The plane map decorator *PMDEC* exports the handle, iterator and circulator types into the class scope and provides all plane map exploration and manipulation methods. See *PMDecorator* in the appendix for the concept description. We inherit from *PMDEC* to obtain its methods into the class scope.

⟨*type definition for the class scope*⟩≡
```
typedef Constrained_triang_traits<PMDEC,GEOM,NEWEDGE> Self;
typedef PMDEC                                         Base;

// the types interfacing the sweep:
typedef NEWEDGE                  INPUT;
typedef typename PMDEC::Plane_map OUTPUT;
typedef GEOM                     GEOMETRY;

typedef typename GEOM::Point_2     Point;
typedef typename GEOM::Segment_2   Segment;
typedef typename GEOM::Direction_2 Direction;

typedef typename Base::Halfedge_handle   Halfedge_handle;
typedef typename Base::Vertex_handle     Vertex_handle;
typedef typename Base::Face_handle       Face_handle;
typedef typename Base::Halfedge_iterator Halfedge_iterator;
typedef typename Base::Vertex_iterator   Vertex_iterator;
typedef typename Base::Face_iterator     Face_iterator;
typedef typename Base::Halfedge_base     Halfedge_base;
typedef typename Base::Halfedge_around_vertex_circulator
        Halfedge_around_vertex_circulator;
```

We have two sentinel edges which are minimum and maximum in our order by identity, we store them in a reference. Note that we need access to our plane map decorator and to our geometric kernel. The point p determines the sweepline position.

⟨*order predicate definition*⟩≡
```
class lt_edges_in_sweepline : public PMDEC
{  const Point& p;
   const Halfedge_handle& e_bottom;
   const Halfedge_handle& e_top;
   const GEOMETRY& K;
public:
lt_edges_in_sweepline(const Point& pi,
   const Halfedge_handle& e1, const Halfedge_handle& e2,
   const PMDEC& D, const GEOMETRY& k) :
     PMDEC(D), p(pi), e_bottom(e1), e_top(e2), K(k) {}

lt_edges_in_sweepline(const lt_edges_in_sweepline& lt) :
   PMDEC(lt), p(lt.p), e_bottom(lt.e_bottom), e_top(lt.e_top), K(lt.K) {}

Segment seg(const Halfedge_handle& e) const
{ return K.construct_segment(point(source(e)),point(target(e))); }

int orientation(Halfedge_handle e, const Point& p) const
{ return K.orientation(point(source(e)),point(target(e)),p); }
```
⟨*function call member for order of two halfedges*⟩
```
}; // lt_edges_in_sweepline
```

The order predicate on edges is only based on point equality and the orientation predicate on points. We have two sentinel edges which are minimum and maximum in our order by identity. For all geometric cases we have the precondition that the source vertex of one of the edges is equal to the current event point *p_sweep*. The order predicate is only used in search operation at *p_sweep* and for the insertion of new edges starting there, thus our requirement is legal.

⟨*function call member for order of two halfedges*⟩≡
```
bool operator()(const Halfedge_handle& e1, const Halfedge_handle& e2) const
{ // Precondition:
  // [[p]] is identical to the source of either [[e1]] or [[e2]].
  if (e1 == e_bottom || e2 == e_top) return true;
  if (e2 == e_bottom || e1 == e_top) return false;
  if ( e1 == e2 ) return 0;
  int s = 0;
  if ( p == point(source(e1)) )      s =   orientation(e2,p);
  else if ( p == point(source(e2)) ) s = - orientation(e1,p);
  else error_handler(1,"compare error in sweep.");
  if ( s || source(e1) == target(e1) || source(e2) == target(e2) )
    return ( s < 0 );
  s = orientation(e2,point(target(e1)));
  if (s==0) error_handler(1,"parallel edges not allowed.");
  return ( s < 0 );
}
```

The order predicate on vertices maps to the lexicographic order on their embedding.

⟨*order predicate definition*⟩+≡
```
class lt_pnts_xy : public PMDEC
{ const GEOMETRY& K;
public:
 lt_pnts_xy(const PMDEC& D, const GEOMETRY& k) : PMDEC(D), K(k) {}
 lt_pnts_xy(const lt_pnts_xy& lt) : PMDEC(lt), K(lt.K) {}
 int operator()(const Vertex_handle& v1, const Vertex_handle& v2) const
 { return K.compare_xy(point(v1),point(v2)) < 0; }
}; // lt_pnts_xy
```

We use an STL map for the *SL* and a set for the *event_Q*. Note that we use the iterators for local movement in *SL* but also as handles to the items therein which store the halfedge objects. Whenever we talk about items we misuse iterators for that concept. We store always the forward oriented halfedges.

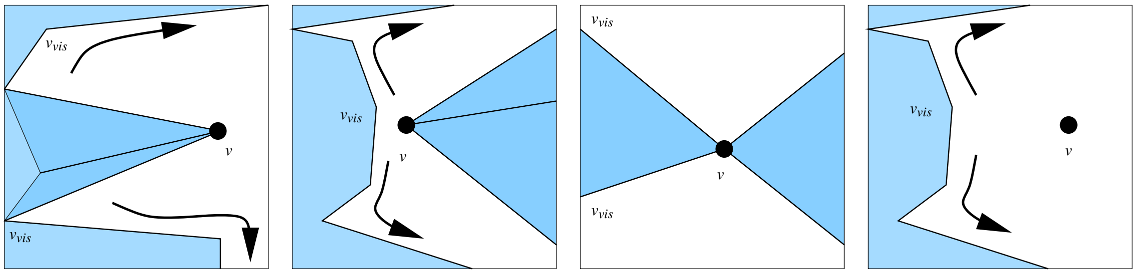
⟨*local types for the sweep*⟩≡
```
typedef std::map<Halfedge_handle, Halfedge_handle, lt_edges_in_sweepline>
        Sweep_status_structure;
typedef typename Sweep_status_structure::iterator   ss_iterator;
typedef typename Sweep_status_structure::value_type ss_pair;
typedef std::set<Vertex_iterator,lt_pnts_xy> Event_Q;
typedef typename Event_Q::const_iterator event_iterator;

const GEOMETRY&         K;
Event_Q                 event_Q;
event_iterator          event_it;
Vertex_handle           event;
Point                   p_sweep;
Sweep_status_structure  SL;
CGAL::Unique_hash_map<Halfedge_handle,ss_iterator> SLItem;
const NEWEDGE&          Treat_new_edge;
Halfedge_handle         e_low,e_high; // framing edges !
Halfedge_handle         e_search;

Constrained_triang_traits(const INPUT& in, OUTPUT& out, const GEOMETRY& k)
  : Base(out), K(k), event_Q(lt_pnts_xy(*this,K)),
    SL(lt_edges_in_sweepline(p_sweep,e_low,e_high,*this,K)),
    SLItem(SL.end()),  Treat_new_edge(in)
{  }
```

4.2.1 Event Handling

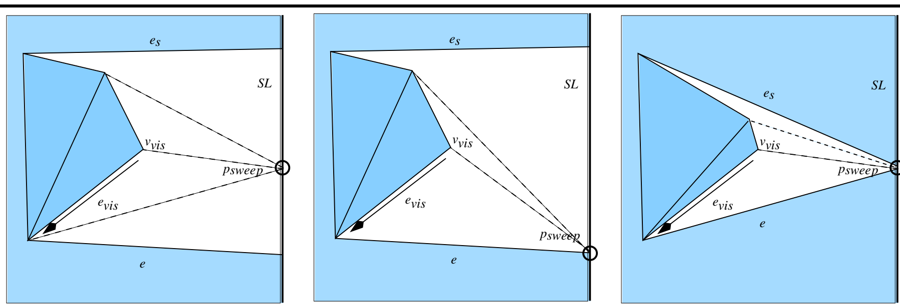
We have to traverse a vertex v of our input graph by the sweepline. Let's first assume this is an *isolated vertex* appearing between two edges in *SL*. Note that between the two edges e, e_s we have a local chain of edges connecting $v_1 = source(e)$ and $v_2 = source(e_s)$ which is similar to the global convex hull chain of the unconstrained triangulation problem. Note that this chain can be even empty if $v_1 = v_2$. In between the segments in the sweepline we know a vertex v_{vis} of maximal coordinates which is visible from any point on *SL* between e and e_s. What happens if we encounter a new event v (a vertex at *p_sweep*)? We locate the edge e below v, and its successor e_s. We obtain the correct v_{vis} and an edge e_{vis} out of v_{vis} which is in the face cycle partly visible from v. Then we determine the visible chain of edges starting in e_{vis} as seen from v. Both ends are determined either by a non-visible edge or by an edge intersecting the sweepline (thus equal to e or e_s). For all the edges in the chain we have to produce new triangles with apex v.



A ending bundle B starting bundle C ending and starting bundle D isolated node

Figure 4.4: The four sweep configurations

Now the general case where the vertex v can be the *center of a star of edges*. Note that all edges are embedded around v in counterclockwise order. This means that some subsequence of the adjacency list is a sequence of starting edges and the corresponding complement is the sequence of ending edges. For the actions we have to consider four configurations as shown in figure 4.4. Note that in case **A** we encounter an ending bundle. We have to triangulate up and down along the limiting edges along the arrows as long as the edges are visible or until they are crossing the sweepline. We also have to remove the ending edges from the sweepline. For the edge in *SL* below the sweep point we have to update the event vertex v as its visible entry vertex into the graph. In case **B** we have only a starting bundle of edges. We first have to link v (at the sweep point) to the visible vertex v_{vis} in the edge chain connecting the segments above and below the sweep point. Then we start the same actions as in case **A** along the chain of visible edges. Finally we insert all starting edges into *SL*. The cases **C** and **D** are combinations of the above. In **C** we first act as in **A** and then as in **B**. The isolated vertex in **D** is handled as in **B** but there are no edges starting.



A *p_sweep* between two segments B *p_sweep* is target of one segment C *p_sweep* is target of two segments

Figure 4.5: The three geometric configurations of v at *p_sweep* with respect to e and e_s and the face cycle in between before the triangular extension.

What are we doing concerning our invariants?

- Between each pair of edges e, e_s in *SL* we extend the chain where the target of e or e_s is the event vertex or e is just below *p_sweep*, by triangles with apex v. Now v becomes the visible vertex in at least one face cycle between two edges in *SL*. See figure 4.5. v can be isolated as in **A**, or connected to G by constraining edges as in **B** or **B**. There's a symmetric case for **B** when $target(e_s) == v$. The latter cases can occur combined if the ending bundle of edges consists of more than one edge.

If you look into the code below, convince yourself that the chains are afterwards again x-monotone and follow our slope criterion. Note also that in case **A** we connect v to $G[\leq SL]$ by new triangles and thus the whole graph is again connected.

Now look at figure 4.6 for the situation after the event. In all three cases our implicit structure $CT[\leq SL]$ is again consistently defined between e and e_s. Note that combinations of the cases **B** and **C** are possible if the outgoing bundle of edges consists of more than one edge.

- We remove the edges ending at v from *SL* and reset the *SLItem* link to the default; we insert some edges starting at v into *SL* and set the *SLItem* link accordingly. Thus invariant 1 holds.
- We have to adjust the visibility information of those edges in *SL* that are below or above v or that have just been inserted into *SL*. Thus invariant 3 is ensured.

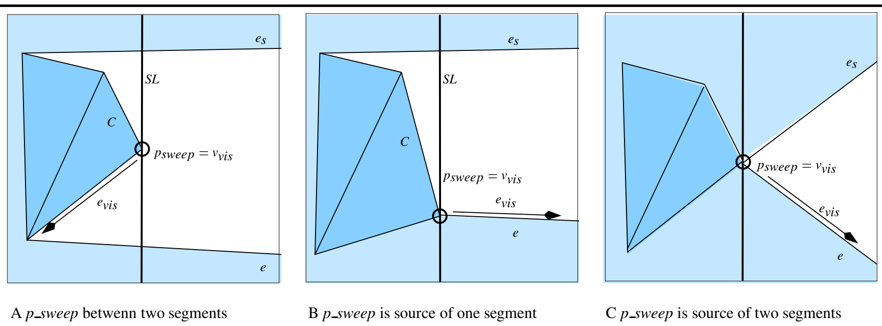


Figure 4.6: The three geometric configurations of v at *p_sweep* with respect to e and e_s and the visible vertex invariant after the insertion of new edges.

Now let's do some coding. For the visibility predicate we use the geometric orientation test provided by the geometric kernel.

⟨*ct event handling*⟩≡

```
bool edge_is_visible_from(Vertex_handle v, Halfedge_handle e)
{
  Point p =  point(v);
  Point p1 = point(source(e));
  Point p2 = point(target(e));
  return ( K.orientation(p1,p2,p)>0 ); // leftturn
}
```

The following operations are used to enrich the output plane map by all the edges completing the triangulation. We only create triangles looking backward from an event. We always start from an edge *e_apex* linking the event vertex to the up to now triangulated plane map. We triangulate away from *e_apex* until we can't see the examined face cycle edge or until the edge crosses the sweepline from left to right. For the visibility test we use the above predicate, for the "edge-crosses-sweepline" test we can use our map *SLItem* as a flag.

⟨*ct event handling*⟩+≡
```
void triangulate_up(Halfedge_handle& e_apex)
{
  Vertex_handle v_apex = source(e_apex);
  while (true) {
    Halfedge_handle e_vis = previous(twin(e_apex));
    bool in_sweep_line = (SLItem[e_vis] != SL.end());
    bool not_visible = !edge_is_visible_from(v_apex,e_vis);
    if ( in_sweep_line || not_visible) {
       return;
    }
    Halfedge_handle e_back = new_bi_edge(e_apex,e_vis);
    if ( !is_forward(e_vis) ) make_first_out_edge(twin(e_back));
    e_apex = e_back;
  }
}
void triangulate_down(Halfedge_handle& e_apex)
{
  Vertex_handle v_apex = source(e_apex);
  while (true) {
    Halfedge_handle e_vis = next(e_apex);
    bool in_sweep_line = (SLItem[e_vis] != SL.end());
    bool not_visible = !edge_is_visible_from(v_apex,e_vis);
    if ( in_sweep_line || not_visible) {
         return;
    }
    Halfedge_handle e_vis_rev = twin(e_vis);
    Halfedge_handle e_forw = new_bi_edge(e_vis_rev,e_apex);
    e_apex = twin(e_forw);
  }
}
```

In this chunk we provide the operation for triangulation of the region left between two edges *e_upper* and *e_lower* with source v in the ending bundle of vertex v. Note that v can see the whole chain of edges calculated so far between $target(e\_upper)$ and $target(e\_lower)$.

⟨*ct event handling*⟩+≡
```
void triangulate_between(Halfedge_handle e_upper, Halfedge_handle e_lower)
{
  // we triangulate the interior of the whole chain between
  // target(e_upper) and target(e_lower)
  assert(source(e_upper)==source(e_lower));
  Halfedge_handle e_end = twin(e_lower);
  while (true) {
    Halfedge_handle e_vis =  next(e_upper);
    Halfedge_handle en_vis = next(e_vis);
```

```
      if (en_vis == e_end) return;
      e_upper = twin(new_bi_edge(twin(e_vis),e_upper));
    }
  }
```

Now the main action. We combine the four cases shown in figure 4.4 in a compact form. To start the event handling correct we expect that the embedding of the target vertices of the edges in the adjacency lists are counterclockwise order-preserving. We expect that the adjacency lists can be split in a first part only consisting of edges e where $point(source(e))$ is smaller than $point(target(e))$ (lexicographic ordering of points) and into a part where the opposite holds. Both parts can be empty. The edges *eb_low* and *eb_high* store the extreme edges of the ending bundle. Both are set during the iteration through the adjacency list of *event* if the ending bundle is non-empty, or to an edge that has to be constructed and that links *event* to the already constructed triangulation $G[\leq SL]$. Both *eb_low* and *eb_high* are manipulated by the $triangulate\_up/down$ operations and are finally part of the chain C of visible edges. Therefore *eb_low* is the visibility edge for the edge below *event* referenced via *sit_pred*.

⟨*ct event handling*⟩+≡

```
  void process_event()
  {
    Halfedge_handle e, ep, eb_low, eb_high, e_end;
    if ( !is_isolated(event) ) {
      e = last_out_edge(event);
      ep = first_out_edge(event);
    }
    ss_iterator sit_pred, sit;
    /* PRECONDITION:
       only ingoing => e is lowest in ingoing bundle
       only outgoing => e is highest in outgoing bundle
       ingoing and outgoing => e is lowest in ingoing bundle */
    eb_high = e_end = ep;
    eb_low = e;
    ⟨determine a handle sit_pred into SL⟩
    ⟨delete ending bundle, insert starting bundle⟩
    triangulate_up(eb_high);
    triangulate_down(eb_low);
    sit_pred->second = eb_low;
  }
```

Given e an edge in the adjacency list of v $SLItem[e]$ is an iterator pointing into SL (and not past the end $==SL.end()$) iff e is an edge in the bundle of edges ending at v (with respect to the sweep). In the latter case $--SLItem[e]$ is an iterator pointing to the edge below *event*. If $SLItem[e]==SL.end()$ then we have no entry point into SL and have to query SL by a call to *upper_bound*. The edge *e_search* is a loop edge with the property $source(e\_search)==target(e\_search)$. We use it for the geometric search in SL. The search is only executed when the *event* is not connected to $G[\leq SL]$.

⟨*determine a handle sit_pred into SL*⟩≡

```
  if ( e != Halfedge_handle() ) {
    point(target(e_search)) = p_sweep; // degenerate loop edge
    sit_pred = SLItem[e];
```

```
    if ( sit_pred != SL.end())  sit = --sit_pred;
    else  sit = sit_pred = --SL.upper_bound(e_search);
  } else { // event is isolated vertex
    point(target(e_search)) = p_sweep; // degenerate loop edge
    sit_pred = --SL.upper_bound(e_search);
  }
```

We iterate the adjacency list *clockwise* starting at e, thus we first encounter the ending bundle (if existing), then the starting bundle (if existing).

⟨*delete ending bundle, insert starting bundle*⟩≡
```
  bool ending_edges(0), starting_edges(0);
  while ( e != Halfedge_handle() ) { // walk adjacency list clockwise
    if ( SLItem[e] != SL.end() )
    ⟨handling ending edges⟩
    else
    ⟨handling starting edges⟩
    if (e == e_end) break;
    e = cyclic_adj_pred(e);
  }
  if (!ending_edges)
  ⟨create link to constrained triangulation⟩
```

When e is directed backwards, then $target(twin(e)) == event$. For each wedge between two ending edges we triangulate the whole face. We also delete all ending edges from *SL* and mark the edges as such.

⟨*handling ending edges*⟩≡
```
  {
    if (ending_edges) triangulate_between(e,cyclic_adj_succ(e));
    ending_edges = true;
    SL.erase(SLItem[e]);
    link_bi_edge_to(e,SL.end());
    // not in SL anymore
  }
```

For starting edges we insert them into *SL* and keep track of the last edge *eb_high* of the ending bundle. For the newly inserted edge their source is the visible vertex.

⟨*handling starting edges*⟩≡
```
  {
    sit = SL.insert(sit,ss_pair(e,e));
    link_bi_edge_to(e,sit);
    if ( !starting_edges ) eb_high = cyclic_adj_succ(e);
    starting_edges = true;
  }
```

The last chunk of this section codes the case where *event* is not connected to $G[\leq SL]$. We first determine the visible vertex and a candidate edge for visibility search along a chain of edges bounding the face which contains *event*. Note that we set *eb_low* and *eb_high* to the newly created edge such that we can triangulate “away” from them afterwards.

⟨*create link to constrained triangulation*⟩≡
```
{
  Halfedge_handle e_vis = sit_pred->second;
  Halfedge_handle e_vis_n = cyclic_adj_succ(e_vis);
  eb_low = eb_high = new_bi_edge(event,e_vis_n);
}
```

Take the time to check that in the combined code the promised invariants are kept.

4.2.2 Initialization

After initialization we want to have all our invariants valid. Remember that G left of the sweepline has to be a valid constrained triangulation, all edges intersecting the sweepline are in *SL* and we have a hashed shortcut to their item. Finally for each area in between two edges which are neighbors in the sweepline we want to know an edge visible from any point of the area. Now what should happen to achieve this. We consider the vertex with minimal coordinates as the initial contrained triangulation. We have to insert all edges in its adjacency list into the sweepline and set the marks and visibility properties. And to avoid special cases we insert two sentinel edges encompassing the whole scenery in a symbolic wedge. Note that the outgoing bundle can be empty in which case we just start with the wedge.

Now our basic invariants hold: the explicit triangulation is just the vertex *event*, all outgoing edges are part of *SL*, and $G[\leq SL]$ is connected.

⟨*ct initialization*⟩≡
```
void link_bi_edge_to(Halfedge_handle e, ss_iterator sit) {
  SLItem[e] = SLItem[twin(e)] = sit;
}

void initialize_structures()
{

  for ( event=vertices_begin(); event != vertices_end(); ++event )
    event_Q.insert(event); // sorted order of vertices

  event_it = event_Q.begin();
  if ( event_Q.empty() ) return;
  event = *event_it;
  p_sweep = point(event);
  ⟨insert all edges starting at event⟩
  ⟨create sentinels for visibility search⟩

  // we move to the second vertex:
  procede_to_next_event();
  event_exists(); // sets p_sweep for check invariants
}
```

We insert all edges in the adjacency list of *event* into *SL*. All are marked to be in *SL* by a call to *link_bi_edge_to*. Thereby we obtain also a hashed shortcut into *SL*. In the wedge between two edges *e* and e_s which are neighbors in *SL* each edge e is also the entry point for a visibility search in the wedge between e and e_s.

⟨*insert all edges starting at event*⟩≡
```
if ( !is_isolated(event) ) {
  Halfedge_around_vertex_circulator
    e(first_out_edge(event)), eend(e);
  CGAL_For_all(e,eend) {
    ss_iterator sit = SL.insert(ss_pair(e,e)).first;
    link_bi_edge_to(e,sit);
  }
}
```

We create two sentinel edges and insert them into *SL*. They are sentinels by identity and not by geometry. The identity is checked in the order predicate of *SL* and thereby all insertions are done between them in the geometric order defined by *lt_edges_in_sweepline*. Additionally as the two sentinel edges *e_low* and *e_high* are marked to be in the sweepline for the whole time of the sweep we never run out of the wedge during our visibility searches. *e_low* and *e_high* start at *event* and extend to a symbolic vertex *v_tmp*. We use one vertex for both edges. Also we insert a loop edge *e_search* at *v_tmp* which we use for lookup within *SL* in our event handling procedure. All edges adjacent to *v_tmp* are removed in the postprocessing step. Note that we don't treat the artificial edges by the new-edge data accessor.

⟨*create sentinels for visibility search*⟩≡
```
Vertex_handle v_tmp = new_vertex(); point(v_tmp) = Point();
e_high = Base::new_halfedge_pair(event,v_tmp);
e_low  = Base::new_halfedge_pair(event,v_tmp);
// this are two symbolic edges just accessed as sentinels
// they carry no geometric information
e_search = Base::new_halfedge_pair(v_tmp,v_tmp);
// this is just a loop used for searches in SL
ss_iterator sit_high = SL.insert(ss_pair(e_high,e_high)).first;
ss_iterator sit_low  = SL.insert(ss_pair(e_low,e_low)).first;
// inserting sentinels into SL
link_bi_edge_to(e_high, sit_high);
link_bi_edge_to(e_low , sit_low);
// we mark them being in the sweepline, which they will never leave
```

Now for the iteration control. We iterate over all vertices, in the order given by the coordinates assigned to the vertices. We set *event* in the initialization and at the end of each event handling phase. We stop when *event_Q* is empty.

⟨*ct event handling*⟩+≡
```
bool event_exists()
{ if ( event_it != event_Q.end() ) {
    // event is set at end of loop and in init
    event = *event_it;
    p_sweep = point(event);
    return true;
  }
```

```
  return false;
}
void procede_to_next_event()
{ ++event_it; }
```

At the end we remove the frame from our structure. This is only the pair of edges spanning the initial wedge.

⟨*ct cleaning up*⟩≡

```
void complete_structures()
{
  if (e_low != Halfedge_handle()) {
    delete_vertex(target(e_search));
  } // removing sentinels and e_search
}
```

During the development we check if the adjacency lists of all vertices have the correct adjacency list embedding. The final check can do the test if the result of our sweep is a correct triangulation. As we don't delete edges or vertices from the output and don't add any vertex[2], all constraining edges and all vertices are in triangulation. If we check the output structure to be a triangulation according to [MNS+99] we have a checking module for our constrained triagulation sweep.

⟨*ct checking*⟩≡

```
void check_ccw_local_embedding() const
{ PM_checker<PMDEC,GEOM> C(*this,K);
  C.check_order_preserving_embedding(event);
}
void check_invariants()
{
#ifdef CGAL_CHECK_EXPENSIVE
  if ( event_it == event_Q.end() ) return;
  check_ccw_local_embedding();
#endif
}
void check_final()
{
#ifdef CGAL_CHECK_EXPENSIVE
  PM_checker<PMDEC,GEOM> C(*this,K); C.check_is_triangulation();
#endif
}
```

The following operations interface the plane map decorator. Note also that the edge data accessor allows to treat the newly created edges of the triangulation.

[2]apart from the temporary one which we delete at the end

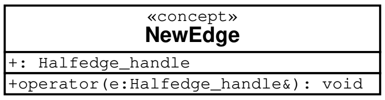


Figure 4.7: The data accessor concept *NewEdge* for the treatment of newly created edges.

⟨*helping operations*⟩≡

```
Halfedge_handle new_bi_edge(Vertex_handle v1, Vertex_handle v2)
{ // appended at v1 and v2 adj list
  Halfedge_handle e = Base::new_halfedge_pair(v1,v2);
  Treat_new_edge(e);
  return e;
}
Halfedge_handle new_bi_edge(Halfedge_handle e_bf, Halfedge_handle e_af)
{ // ccw before e_bf and after e_af
  Halfedge_handle e = Base::new_halfedge_pair(e_bf,e_af,Halfedge_base(),
    Base::BEFORE, Base::AFTER);
  Treat_new_edge(e);
  return e;
}
Halfedge_handle new_bi_edge(Vertex_handle v, Halfedge_handle e_bf)
{ // appended at v's adj list and before e_bf
  Halfedge_handle e = Base::new_halfedge_pair(v,e_bf,Halfedge_base(),
    Base::BEFORE);
  Treat_new_edge(e);
  return e;
}
Segment seg(Halfedge_handle e) const
{ return K.construct_segment(point(source(e)),point(target(e))); }
Direction dir(Halfedge_handle e) const
{ return K.construct_direction(point(source(e)),point(target(e))); }
bool is_forward(Halfedge_handle e) const
{ return K.compare_xy(point(source(e)),point(target(e))) < 0; }
```

4.2.3 Correctness and Running Time

At the end only the sentinels are in *SL*. $G[\leq SL]$ already consists of the constrained triangulation of the input structure. The two parts of the chain C between the source of the sentinels just consist of the upper and lower convex hull chain between the lexicographic smallest and the lexicographic largest vertex. The convexity follows from the slope property. During the sweep we once encountered any vertex and any edge and integrated it into the constrained triangulation according to our invariants. Thus completeness is trivial.

The size of the constrained triangulation of a set of segments is of the same order as the unconstrained triangulation of the segment end points. The sweep procedure takes time for the production of the output which is known to be linear in size. The only additional cost at each event is the insertion of the segments starting at an event point where we use a tree based dictionary to store these.

This costs logarithmic time per segment to insert. We end up with the standard $O(n \log n)$ time bound where $n = |S|$. The space is dominated by the size of the produced output $O(n)$.

⟨*some predefinitions*⟩≡

```
#ifndef LEDA_ERROR_H
static void error_handler(int n, const char* s)
{ std::cerr << s << std::endl;
  exit(n);
}
#endif
struct Do_nothing {
Do_nothing() {}
template <typename ARG>
void operator()(ARG&) const {}
};
```

4.2.4 Visualization via the generic sweep observer

⟨*Constrained_triang_anim.h*⟩≡

```
⟨CGAL Header1⟩
// file          : include/CGAL/Nef_2/Constrained_triang_anim.h
⟨CGAL Header2⟩
#ifndef CGAL_PM_CONSTR_TRIANG_ANIM_H
#define CGAL_PM_CONSTR_TRIANG_ANIM_H
#include <CGAL/Nef_2/PM_visualizor.h>

template <class GT>
class Constrained_triang_anim {
  CGAL::Window_stream _W;
public:
  typedef CGAL::Window_stream   VDEVICE;
  typedef typename GT::GEOMETRY GEOM;
  typedef typename GT::Base     PMDEC;
  typedef typename PMDEC::Point Point;
  Constrained_triang_anim() : _W(400,400)
  { _W.set_show_coordinates(true); _W.init(-120,120,-120,5); _W.display(); }
  VDEVICE& device() { return _W; }
void post_init_animation(GT& gpst)
{
  PM_visualizor<PMDEC,GEOM> V(_W,gpst);
  V.point(V.target(gpst.e_search)) = Point(-120,0);
  // to draw we have to embed the virtual search vertex
  V.draw_skeleton(CGAL::BLUE);
  _W.read_mouse();
}
void pre_event_animation(GT& gpst)
{ }
void post_event_animation(GT& gpst)
{ PM_visualizor<PMDEC,GEOM> V(_W,gpst);
  V.draw_ending_bundle(gpst.event,CGAL::GREEN);
```

```
  _W.read_mouse();
}
void post_completion_animation(GT& gpst)
{ _W.clear();
  PM_visualizor<PMDEC,GEOM> V(_W,gpst);
  V.draw_skeleton(CGAL::BLACK);
  _W.read_mouse(); }
};
#endif // CGAL_PM_CONSTR_TRIANG_ANIM_H
```

4.3 A Test of the plane map triangulation

We produce a simple homogeneous kernel, a plane map and use a segment overlay sweep to create an input structure for the constrained triangulation algorithm.

⟨*Constrained_triang-test.C*⟩≡

```
#include <CGAL/basic.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Point_2.h>
#include "Affine_geometry.h"
#undef CGAL_CFG_NO_TMPL_IN_TMPL_PARAM
#include <CGAL/Nef_2/HalfedgeDS_default.h>
#include <CGAL/Nef_2/HDS_items.h>
#include <CGAL/Nef_2/PM_decorator.h>
#include <CGAL/Nef_2/Constrained_triang_traits.h>
#include <CGAL/Nef_2/Constrained_triang_anim.h>
#include <CGAL/Nef_2/Segment_overlay_traits.h>
#include <CGAL/test_macros.h>
// KERNEL:
typedef CGAL::Homogeneous<leda_integer> Hom_kernel;
typedef CGAL::Affine_geometry<Hom_kernel> Aff_kernel;
typedef Aff_kernel::Segment_2 Segment;
// HALFEDGE DATA STRUCTURE:
struct HDS_traits {
  typedef Aff_kernel::Point_2 Point;
  typedef bool Mark;
};
typedef  CGAL::HalfedgeDS_default<HDS_traits,HDS_items> HDS;
typedef  CGAL::PM_decorator< HDS > PM_dec;
typedef  PM_dec::Halfedge_handle       Halfedge_handle;
typedef  PM_dec::Vertex_handle         Vertex_handle;
typedef  PM_dec::Halfedge_const_handle  Halfedge_const_handle;
// SEGMENT OVERLAY:
template <typename PMDEC, typename I>
class PM_dec_output : public PMDEC {
public:
  typedef PMDEC Base;
```

```
  typedef typename Base::Plane_map       Plane_map;
  typedef typename Base::Point           Point;
  typedef typename Base::Vertex_handle   Vertex_handle;
  typedef typename Base::Halfedge_handle Halfedge_handle;
  typedef I                              ITERATOR;

  PM_dec_output(HDS& H) : Base(H) {}
  PM_dec_output(const PM_dec_output& P) : Base(P) {}

  Vertex_handle new_vertex(const Point& p) const
  { Vertex_handle v = Base::new_vertex();
    v->point() = p; return v; }

  void link_as_target_and_append(Vertex_handle v, Halfedge_handle e) const
  { Base::link_as_target_and_append(v,e); }

  Halfedge_handle new_halfedge_pair_at_source(Vertex_handle v)  const
  { return Base::new_halfedge_pair_at_source(v,Base::BEFORE); }

  void supporting_segment(Halfedge_handle e, ITERATOR it) const {}
  void trivial_segment(Vertex_handle v, ITERATOR it) const {}
  void halfedge_below(Vertex_handle v, Halfedge_handle e) const {}
  void starting_segment(Vertex_handle v, ITERATOR it) const {}
  void passing_segment(Vertex_handle v, ITERATOR it) const {}
  void ending_segment(Vertex_handle v, ITERATOR it) const  {}
}; // PM_dec_output

typedef std::list&lt;Segment&gt;::const_iterator Seg_iterator;
typedef CGAL::Segment_overlay_traits< Seg_iterator,
  PM_dec_output<PM_dec,Seg_iterator>, Aff_kernel> PM_seg_overlay;
typedef CGAL::generic_sweep<PM_seg_overlay> PM_seg_overlay_sweep;

// CONSTRAINED TRIANGULATIONS:
typedef CGAL::Constrained_triang_traits<PM_dec,Aff_kernel> CTT;
typedef CGAL::generic_sweep<CTT> Constrained_triang_sweep;
typedef CGAL::Constrained_triang_anim<CTT> CTA;
typedef CGAL::sweep_observer<Constrained_triang_sweep,CTA> CTS_observer;

// MAIN PROGRAM:

int main(int argc, char* argv[])
{
  // SETDTHREAD(19);
  CGAL::set_pretty_mode ( cerr );
  HDS H;
  Aff_kernel AK;
  CTS_observer Obs;
  Obs.device().message("Insert segments to triangulate.");
  std::list&lt;Segment&gt; L;
  Segment s;
  if ( argc == 2 ) {
    std::ifstream log(argv[1]);
    while ( log >> s ) { L.push_back(s); Obs.device() << s; }
  }
  while ( Obs.device() >> s ) L.push_back(s);
  std::string fname(argv[0]);
  fname += ".log";
  std::ofstream log(fname.c_str());
  for (Seg_iterator sit = L.begin(); sit != L.end(); ++sit)
```

```
    log << *sit << " ";
  log.close();
  PM_seg_overlay::OUTPUT D(H);
  PM_seg_overlay_sweep   OV(PM_seg_overlay::INPUT(L.begin(),L.end()),D,AK);
  CTT::INPUT I;
  Constrained_triang_sweep CT(I,H,AK);
  Obs.attach(CT);
  OV.sweep();
  CT.sweep();
  return 0;
}
```

Bibliography

[MN99] K. Mehlhorn and S. Näher. *LEDA, A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MNS+99] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *CGTA: Computational Geometry: Theory and Applications*, 12, 1999.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry : An Introduction*. Springer, 1985.

5 Plane Map Point Location

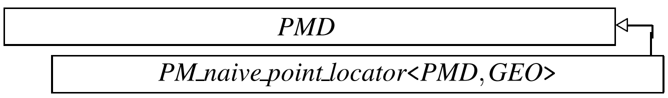
5.1 The Manual Page

5.1.1 Naive point location in plane maps (PM_naive_point_locator)

1. Definition

An instance *PL* of data type *PM_naive_point_locator*<*PMD*,*GEO*> encapsulates naive point location queries within a plane map *P*. The two template parameters are specified via concepts. *PMD* must be a model of the concept *PMDecorator* as described in the appendix. *GEO* must be a model of the concept *AffineGeometryTraits_2* as described in the appendix. For a specification of plane maps see also the concept of *PMConstDecorator*.

2. Generalization



3. Types

PM_naive_point_locator<*PMD*,*GEO*>::*Decorator*

equals *PMD*.

PM_naive_point_locator<*PMD*,*GEO*>::*Plane_map*

the plane map type decorated by *Decorator*.

PM_naive_point_locator<*PMD*,*GEO*>::*Mark*

the attribute of all objects (vertices, edges, faces).

PM_naive_point_locator<*PMD*,*GEO*>::*Geometry*

equals *GEO*.

PM_naive_point_locator<*PMD*,*GEO*>::*Point*

the point type of the geometry kernel.
Requirement: *Geometry*::*Point_2* equals *Plane_map*::*Point*.

PM_naive_point_locator<*PMD*,*GEO*>::*Segment*

the segment type of the geometry kernel.

Local types are handles, iterators and circulators of the following kind: *Vertex_const_handle*, *Vertex_const_iterator*, *Halfedge_const_handle*, *Halfedge_const_iterator*, *Face_const_handle*, *Face_const_iterator*.

PM_naive_point_locator<PMD, GEO>::Object_handle

a generic handle to an object of the underlying plane map. The kind of the object (*vertex*, *halfedge*, *face*) can be determined and the object assigned by the three functions:
bool assign(*Vertex_const_handle*& *h*, *Object_handle o*)
bool assign(*Halfedge_const_handle*& *h*, *Object_handle o*)
bool assign(*Face_const_handle*& *h*, *Object_handle o*)
where each function returns *true* iff the assignment of *o* to *h* was valid.

4. Creation

PM_naive_point_locator<PMD, GEO> PL(*const Plane_map*& *P*, *const Geometry*& *k* = *Geometry*( ));

constructs a point locator working on *P*.

5. Operations

Mark *PL*.mark(*Object_handle h*)

returns the mark associated to the object *h*.

Object_handle *PL*.locate(*Segment s*)

returns a generic handle *h* to an object (vertex, halfedge, face) of the underlying plane map *P* which contains the point p = *s.source*( ) in its relative interior. *s.target*( ) must be a point such that *s* intersects the 1-skeleton of *P*.

template <*typename Object_predicate*>
Object_handle *PL*.ray_shoot(*Segment s*, *Object_predicate M*)

returns an *Object_handle o* which can be converted to a *Vertex_const_handle*, *Halfedge_const_handle*, *Face_const_handle h* as described above. The object predicate *M* has to have function operators
bool operator( ) (*const Vertex_/Halfedge_/Face_const_handle*&).
The object returned is intersected by the segment *s* and has minimal distance to *s.source*( ) and $M(h)$ holds on the converted object. The operation returns the null handle *NULL* if the ray shoot along *s* does not hit any object *h* of *P* with $M(h)$.

6. Implementation

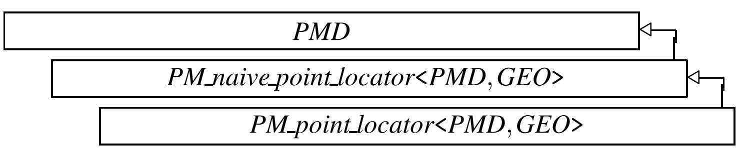
Naive query operations are realized by checking the intersection points of the 1-skeleton of the plane map *P* with the query segments *s*. This method takes time linear in the size *n* of the underlying plane map without any preprocessing.

5.1.2 Point location in plane maps via LMWT (PM_point_locator)

1. Definition

An instance *PL* of data type *PM_point_locator<PMD, GEO>* encapsulates point location queries within a plane map *P*. The two template parameters are specified via concepts. *PMD* must be a model of the concept *PMDecorator* as described in the appendix. *GEO* must be a model of the concept *AffineGeometryTraits_2* as described in the appendix. For a specification of plane maps see also the concept of *PMConstDecorator*.

2. Generalization



3. Types

All local types of *PM_naive_point_locator* are inherited.

4. Creation

PM_point_locator<*PMD*,*GEO*> *PL*(*const Plane_map*& *P*, *const Geometry*& *k* = *Geometry*( ));

constructs a point locator working on *P*.

5. Operations

const Decorator& *PL*.triangulation()

access to the constrained triangulation structure that is superimposed to *P*.

Object_handle *PL*.locate(*Point p*)

returns a generic handle *h* to an object (vertex, halfedge, face) of *P* which contains the point *p* in its relative interior.

template <*typename Object_predicate*>
Object_handle *PL*.ray_shoot(*Segment s*, *Object_predicate M*)

returns an *Object_handle o* which can be converted to a *Vertex_const_handle*, *Halfedge_const_handle*, *Face_const_handle h* as described above. The object predicate *M* has to have function operators
bool operator( ) (*const Vertex_/ Halfedge_/Face_const_handle*&) *const*.
The object returned is intersected by the segment *s* and has minimal distance to *s.source*( ) and $M(h)$ holds on the converted object. The operation returns the null handle *NULL* if the ray shoot along *s* does not hit any object *h* of *P* with $M(h)$.

6. Implementation

The efficiency of this point location module is mostly based on heuristics. Therefore worst case bounds are not very expressive. The query operations take up to linear time for subsequent query operations though they are better in practise. They trigger a one-time initialization which needs worst case $O(n^2)$ time though runtime tests often show subquadratic results. The necessary space for the query structure is subsumed in the storage space $O(n)$ of the input plane map. The query times are configuration dependent. If LEDA is present then point location is done via the slap method based on persistent dictionaries. Then $T_{pl}(n) = O(\log(n))$. If CGAL is not configured to use LEDA then point location is done via a segment walk in the underlying convex subdivision of P. In this case $T_{pl}(n)$ is the number of triangles crossed by a walk from the boundary of the structure to the query point. The time for the ray shooting operation $T_{rs}(n)$ is the time for the point location $T_{pl}(n)$ plus the time for the walk in the triangulation that is superimposed to the plane map. Let's consider the plane map edges as obstacles and the additional triangulation edges as non-obstacle edges. Let's call the sum of the lengths of all edges of the triangulation its weight. If the calculated triangulation approximates[1] the minimum weight triangulation of the obstacle set then the stepping quotient[2] for a random direction of the ray shot is expected to be $O(\sqrt{n})$.

5.2 Implementation

This section describes two point location and ray shooting modules working in plane maps that are decorated according to our plane map decorator model. The first module offers naive point location without any preprocessing of the plane map. The second module implements point location and ray shooting for the case of iterated queries and trades query time for a one time preprocessing phase.

[1]The calculation of general minimum-weight-triangulations is conjectured to be NP-complete and locally-minimum-weight-triangulations that we use are considered good approximations.

[2]The number of non-obstacle edges crossed until an obstacle edge is hit.

In the center of our approach is a constrained triangulation of the obstacle set (the input plane map). For the point location we offer an optimal solution with respect to space and runtime based on a slap subdivision with persistent dictionaries from LEDA. As CGAL demands to minimize cross library dependency we offer a default fall-back solution based on walks in the triangulation. For ray shooting in general obstacle sets the theory recommends to use minimum weight triangulations. However as minimum weight triangulations are difficult to calculate we restrict our effort to optimize the weight of the constrained triangulation only locally. We will present the corresponding theoretical basics in the section where we explain the construction of the triangulation. We start with the naive module and append the triangulation module.

5.2.1 Point location and ray shooting done naively

In this section we implement the class *PM_naive_point_locator*`<>` that wraps naive point location functionality. We first sketch the class design consisting of the local types and the interface. Then we present the two main interface operations for point location and ray shooting.

A point locator *PM_naive_point_locator*`<>` is a generalization of a plane map decorator *PM_decorator_* where we inherit the decorator interface. We obtain the geometry used for point location from *Geometry_*.

⟨*Naive point locator*⟩≡

```
template <typename PM_decorator_, typename Geometry_>
class PM_naive_point_locator : public PM_decorator_ {
protected:
  typedef PM_decorator_ Base;
  typedef PM_naive_point_locator<PM_decorator_,Geometry_> Self;
  const Geometry_& K;
public:
  ⟨PL naive local types⟩
  ⟨PL naive helpers⟩
  ⟨PL naive interface⟩
}; // PM_naive_point_locator<PM_decorator_,Geometry_>
```

We transport types from *PM_decorator_* and *Geometry_* into the local scope. A CGAL *Object* is a general wrapper class for any type. Unwrapping can be done typesave by means of C++ dynamic casts. The *Object_handle* type is just a generalized CGAL *Object* where we add the *NULL* equality test.

⟨*PL naive local types*⟩≡

```
typedef PM_decorator_                  Decorator;
typedef typename Decorator::Plane_map Plane_map;
typedef typename Decorator::Mark      Mark;

typedef Geometry_                       Geometry;
typedef typename Geometry_::Point_2     Point;
typedef typename Geometry_::Segment_2   Segment;
typedef typename Geometry_::Direction_2 Direction;

class Object_handle
  : public CGAL::Object {
  typedef CGAL::Object Base;
```

```
public:
  Object_handle() : Base() {}
  Object_handle(const CGAL::Object& o) : Base(o) {}
  Object_handle(const Object_handle& h) : Base(h) {}
  Object_handle& operator=(const Object_handle& h)
  { Base::operator=(h); return *this; }
  bool operator==(CGAL_NULL_TYPE n) const
  { assert(n == 0); return Base::is_empty(); }
}; // Object_handle
```

⟨*PL naive local types*⟩+≡

```
#define USING(t) typedef typename PM_decorator_::t t
USING(Vertex_handle);
USING(Halfedge_handle);
USING(Face_handle);
USING(Vertex_const_handle);
USING(Halfedge_const_handle);
USING(Face_const_handle);
USING(Vertex_iterator);
USING(Halfedge_iterator);
USING(Face_iterator);
USING(Vertex_const_iterator);
USING(Halfedge_const_iterator);
USING(Face_const_iterator);
USING(Halfedge_around_vertex_circulator);
USING(Halfedge_around_vertex_const_circulator);
USING(Halfedge_around_face_circulator);
USING(Halfedge_around_face_const_circulator);
#undef USING
```

The naive interface provides construction, access to the attributes of type *Mark*, point location, and ray shooting. Note that the ray shooting is parameterized by a template predicate that allows adaptation of the termination criterion of the ray-shot.

⟨*PL naive interface*⟩≡

```
PM_naive_point_locator() : Base() {}

PM_naive_point_locator(const Plane_map& P, const Geometry& k = Geometry()) :
  Base(const_cast<Plane_map&>(P)), K(k) {}

const Mark& mark(Object_handle h) const
{ Vertex_const_handle v;
  Halfedge_const_handle e;
  Face_const_handle f;
  if ( assign(v,h) ) return mark(v);
  if ( assign(e,h) ) return mark(e);
  if ( assign(f,h) ) return mark(f);
  CGAL_assertion_msg(0,
  "PM_point_locator::mark: Object_handle holds no object.");
#if !defined(__BORLANDC__)
  return mark(v); // never reached
```

```
#endif
}
⟨NPL location method⟩
⟨NPL ray shooting method⟩
```

⟨*PL naive interface*⟩+≡

```
// C++ is really friendly:
#define USECMARK(t) const Mark& mark(t h) const { return Base::mark(h); }
#define USEMARK(t)  Mark& mark(t h) const { return Base::mark(h); }
USEMARK(Vertex_handle)
USEMARK(Halfedge_handle)
USEMARK(Face_handle)
USECMARK(Vertex_const_handle)
USECMARK(Halfedge_const_handle)
USECMARK(Face_const_handle)
#undef USEMARK
#undef USECMARK
```

Point location

The following point location scheme runs in linear time without any preprocessing. The segment s is required to intersect the skeleton of our plane map P. We just check if the location $p = source(s)$ is part of the embedding of any vertex or uedge of the 1-skeleton of P. The *make_object*( ) operation just wraps the vertex or halfedge into a CGAL object.

⟨*NPL location method*⟩≡

```
Object_handle locate(const Segment& s) const
{
  if (number_of_vertices() == 0)
    CGAL_assertion_msg(0,"PM_naive_point_locator: plane map is empty.");
  Point p = K.source(s);
  Vertex_const_iterator vit;
  for(vit = vertices_begin(); vit != vertices_end(); ++vit) {
    if ( p == point(vit) ) return make_object(vit);
  }
  Halfedge_const_iterator eit;
  for(eit = halfedges_begin(); eit != halfedges_end(); ++(++eit)) {
    // we only have to check each second halfedge
    if ( K.contains(segment(eit),p) )
      return make_object(eit);
  }
  Vertex_const_handle v_res;
  Halfedge_const_handle e_res;
  ⟨NPL determine closest object intersecting s⟩
  if ( e_res != Halfedge_const_handle() )
    return make_object((Face_const_handle)(face(e_res)));
```

```
    else
      return make_object((Face_const_handle)(face(v_res)));
  }
```

If p is located in the interior of a face f we can identify f only by any object in its boundary. We do this by a thought ray shoot along s through all obstacles of the 1-skeleton. Note that we are sure to hit at least one object of the 1-skeleton of the plane map. Let $q = s.target()$. We check the intersection points of the segment s with all skeleton objects. As soon as we find an object we prune s, store the object that gives possibly access to the face containing p, and iterate.

An edge e that intersects s in its relative interior is trivial to handle. In this case we just clip s by the intersection point and go on with our search to possibly find a closer edge to p.

Isolated vertices are easy to treat too. They can be checked directly as they carry a link to their incident face. Non-isolated vertices don't carry the face information directly. There we have to find an incident edge whose incident face intersects s. Intuitively when s contains a non-isolated vertex v we use a symbolic perturbation scheme. Let's assume that our direction is down along the negative y-axis. We just shift the whole scene infinitesimally left. Then we only hit edges. But of course the closest edge in $A(v)$[3] has to be determined not by proximity but by the direction of edges in $A(v)$. Now we are interested in an edge e of $A(v)$ that has a minimal counterclockwise angle to s. e bounds the wedge containing p. See figure 5.1.

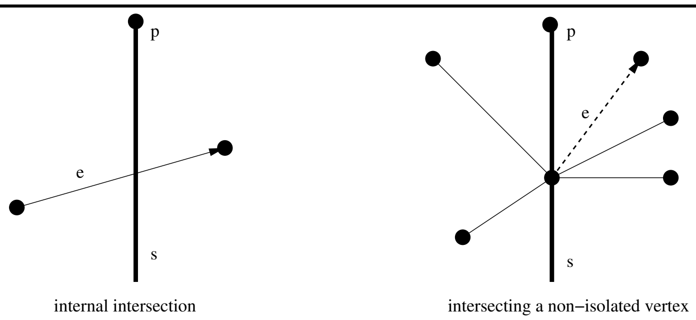


Figure 5.1: Configurations of s and the 1-skeleton of P

Assume d to be a direction anchored in v. The operation $out\_wedge(v, d, collinear)$ returns the halfedge e bounding a wedge in between two neighbored edges in the adjacency list of v that contains d. If d extends along an edge then e is this edge. If d extends into the interior of such a wedge then e is the first edge hit when d is rotated clockwise. As a precondition we ask v not to be isolated.

⟨*PL naive helpers*⟩≡

```
Halfedge_const_handle out_wedge(Vertex_const_handle v,
  const Direction& d, bool& collinear) const
{
  assert(!is_isolated(v));
  collinear=false;
  Point p = point(v);
  Halfedge_const_handle e_res = first_out_edge(v);
  Direction d_res = direction(e_res);
```

[3] the adjacency list of v.

```
    Halfedge_around_vertex_const_circulator el(e_res),ee(el);
    CGAL_For_all(el,ee) {
      if ( K.strictly_ordered_ccw(d_res, direction(el), d) )
        e_res = el; d_res = direction(e_res);
    }
    if ( direction(cyclic_adj_succ(e_res)) == d ) {
      e_res = cyclic_adj_succ(e_res);
      collinear=true;
    }
    return e_res;
  }
```

We use the above operation when the vertex is not isolated.

⟨*NPL determine closest object intersecting s*⟩≡

```
  Segment ss = s; // we shorten the segment iteratively
  Direction dso = K.construct_direction(K.target(s),p), d_res;
  CGAL::Unique_hash_map<Halfedge_const_handle,bool> visited(false);
  for(vit = vertices_begin(); vit != vertices_end(); ++vit) {
    Point p_res, vp = point(vit);
    if ( K.contains(ss,vp) ) {
      ss = K.construct_segment(p,vp); // we shrink the segment
      if ( is_isolated(vit) ) {
        v_res = vit; e_res = Halfedge_const_handle();
      } else { // not isolated
        bool dummy;
        e_res = out_wedge(vit,dso,dummy);
        Halfedge_around_vertex_const_circulator el(e_res),ee(el);
        CGAL_For_all(el,ee)
          visited[el] = visited[twin(el)] = true;
        /* e_res is now the counterclockwise maximal halfedge out
           of v just before s */
        if ( K.orientation(p,vp,point(target(e_res))) < 0 ) // right turn
          e_res = previous(e_res);
        // correction to make e_res visible from p
      }
    }
  }
```

Now we treat the left case of figure 5.1. We check if the current segment *ss* intersects a halfedge and shorten it accordingly. Note that the edge *e_res* is chosen such that *p* is left of it. The *visited* flags allow us to skip all edges that have already been examined incident to vertices on *s* which saves the geometric computations.

⟨*NPL determine closest object intersecting s*⟩+≡

```
  for (eit = halfedges_begin(); eit != halfedges_end(); ++eit) {
    if ( visited[eit] ) continue;
    Point se = point(source(eit)),
          te = point(target(eit));
    int o1 = K.orientation(ss,se);
```

```
    int o2 = K.orientation(ss,te);
    if ( o1 == -o2 && // internal intersection
         K.orientation(se,te,K.source(ss)) !=
         K.orientation(se,te,K.target(ss)) ) {
      Point p_res = K.intersection(s,segment(eit));
      ss = K.construct_segment(p,p_res);
      e_res = (o2 > 0 ? eit : twin(eit));
      // o2>0 => te left of s and se right of s => p left of e
      visited[eit] = visited[twin(eit)] = true;
    }
  }
```

Ray shooting

The ray shooting has to determine a closest object h (vertex, halfedge, face) of our plane map P which fulfills the predicate M. We first locate the source point $p = s.source(\,)$ by our point location method. If we already hit an object that fulfills M then we are done. Otherwise we look for vertices or halfedges along s.

⟨*NPL ray shooting method*⟩≡

```
template <typename Object_predicate>
Object_handle ray_shoot(const Segment& s, const Object_predicate& M) const
{
  assert( !K.is_degenerate(s) );
  Point p = K.source(s);
  Segment ss(s);
  Direction d = K.construct_direction(K.source(s),K.target(s));
  Object_handle h = locate(s);
  Vertex_const_handle v;
  Halfedge_const_handle e;
  Face_const_handle f;
  if ( assign(v,h) && M(v) ||
       assign(e,h) && M(e) ||
       assign(f,h) && M(f) ) return h;
  h = Object_handle();
  ⟨NPL segment s contains vertex⟩
  ⟨NPL segment s intersects halfedge⟩
  return h;
}
```

Look at a vertex v at position pv on segment s. If $M(v)$ we can shorten s and iterate. If not $M(v)$ then a ray shoot along s can still hit an object at v where M is fulfilled in form of an outgoing edge or the face in a wedge between two edges. At each vertex we look only for the part of s between pv and $s.target(\,)$.

⟨*NPL segment s contains vertex*⟩≡

```
for (v = vertices_begin(); v != vertices_end(); ++v) {
  Point pv = point(v);
  if ( !K.contains(ss,pv) ) continue;
```

```
    if ( M(v) ) {
      h = make_object(v);     // store vertex
      ss = K.construct_segment(p,pv); // shorten
      continue;
    }
    // now we know that v is not marked but on s
    bool collinear;
    Halfedge_const_handle e = out_wedge(v,d,collinear);
    if ( collinear ) {
      if ( M(e) ) {
        h = make_object(e);
        ss = K.construct_segment(p,pv);
      }
      continue;
    }
    if ( M(face(e)) ) {
      h = make_object(face(e));
      ss = K.construct_segment(p,pv);
    }
  } // all vertices
```

All halfedges can shorten *ss* when intersecting the segment in its interior and when the halfedge fulfills M.

⟨*NPL segment s intersects halfedge*⟩≡

```
  Halfedge_const_iterator e_res;
  for(e = halfedges_begin(); e != halfedges_end(); ++(++e)) {
    Segment es = segment(e);
    int o1 = K.orientation(ss,K.source(es));
    int o2 = K.orientation(ss,K.target(es));
    if ( o1 == -o2 && o1 != 0 &&
         K.orientation(es, K.source(ss)) ==
         - K.orientation(es, K.target(ss)) ) {
      // internal intersection
      Point p_res = K.intersection(s,es);
      e_res = (o2 > 0 ? e : twin(e));
      // o2 > 0 => te left of s and se right of s => p left of e
      if ( M(e_res) ) {
        h = make_object(e_res);
        ss = K.construct_segment(p,p_res);
      } else if ( M(face(twin(e_res))) ) {
        h = make_object(face(twin(e_res)));
        ss = K.construct_segment(p,p_res);
      }
    }
  }
```

Correctness and Runtimes

We summarize the facts in a lemma.

Lemma 5.2.1: If the segment s intersects the skeleton of P then *point_location*(s) returns a generic handle h to the object (vertex, halfedge, face) whose embedding in the plane contains the source of s. The running time is linear in the size of the plane map P.

If the segment s intersects the skeleton of P then *ray_shoot*(s,M) determines a generic handle h which is *NULL* if no object of P that intersects s fulfills $M(h)$[4] and which can be converted to a corresponding vertex, halfedge, or face otherwise. In the latter case the object fulfills $M(h)$ and additionally has minimal distance to *source*(s). The running time is linear in the size of the plane map.

Proof. For the point location case note that we iterate over all objects of the skeleton of P twice. If $p = \mathit{source}(s)$ is contained in a skeleton object then h is determined already in ⟨*NPL locate*⟩ in one of the two iterations. If p is contained in the plane in between the 1-skeleton then the closest skeleton object along s defines the face. If the closest object is a vertex we determine this vertex in the vertex iteration of ⟨*NPL determine closest object intersecting s*⟩. For non-isolated vertices we also iterate over all edges in the adjacency list of the vertices (in *out_wedge*( )) to determine a halfedge visible from p along s. If the closest object is a halfedge intersecting s in its iterior the halfedge iteration of ⟨*NPL determine closest object intersecting s*⟩ does the job. As all geometric tests take constant time this gives us the linear runtime bound. Note that starting from s we only shorten s. Due to our precondition s indeed is shortened at least once. This ensures that we obtain an object which allows us to obtain the face containing p.

For the ray shooting note that if p already is contained in an object marked correctly then the point location already determines this object. If not then a similar iteration over all objects now only considering the predicate M does the job. If no object is determined then h is the *NULL* handle. The runtime stays linear. □

⟨*PL naive helpers*⟩+≡

```
Segment segment(Halfedge_const_handle e) const
{ return K.construct_segment(point(source(e)), point(target(e))); }

Direction direction(Halfedge_const_handle e) const
{ return K.construct_direction(point(source(e)),point(target(e))); }

template <class Handle>
Object_handle make_object(Handle h) const
{ return CGAL::make_object(h); }
```

⟨*PL naive interface*⟩+≡

5.2.2 Point location and ray shooting based on constrained triangulations

Our second point location structure is based on a convex subdivision of the plane map by means of a constrained triangulation[5]. We create the point location structure starting from a plane map. The edges and isolated vertices of the plane map are our obstacle set. We reference the plane map again via a decorator. We present the class layout, and the implementation of the two main interface operations

[4]Note that the predicate M is defined on vertex, halfedge, and face handles for performance reasons. However we sloppily write $M(h)$ on a generic handle wrapping one of the above.

[5]The constrained triangulation of plane map is a triangulation of the vertices such that all edges are part of the triangulation

locate( ) and *ray_shoot*( ). The construction of the constrained triangulation and the integration of persistent dictionaries as a location structure is shown in the following sections.

The main class is derived from the naive point location class to inherit its primitive methods. The class *PM_point_locator* decorates the input plane map *P*. We store the additional triangulation via an additional decorator *CT* in the local scope. Note that we talk about two plane map structures here. The plane map decorated by the $*this$ object which we also call the input structure and the plane map storing the constrained triangulation of the input structure. We have to maintain links such that each object in *CT* knows the object of $*this$ that supports[6] it. We use an optional point location structure based on persistent dictionaries that is configuration dependent. All properties of that optional module are explained in section 5.2.6.

⟨*Triangulation point locator*⟩≡

```
template <typename PM_decorator_, typename Geometry_>
class PM_point_locator : public
  PM_naive_point_locator<PM_decorator_,Geometry_> {
protected:
  typedef PM_naive_point_locator<PM_decorator_,Geometry_> Base;
  typedef PM_point_locator<PM_decorator_,Geometry_> Self;
  Base CT;
  ⟨TPL persistent module members⟩
public:
  ⟨TPL local types⟩
protected:
  ⟨TPL protected members⟩
public:
  ⟨TPL interface⟩
}; // PM_point_locator<PM_decorator_,Geometry_>
```

⟨*TPL local types*⟩≡

```
#define USING(t) typedef typename Base::t t
USING(Decorator);
USING(Plane_map);
USING(Mark);
USING(Geometry);
USING(Point);
USING(Segment);
USING(Direction);
USING(Object_handle);
USING(Vertex_handle);
USING(Halfedge_handle);
USING(Face_handle);
USING(Vertex_const_handle);
USING(Halfedge_const_handle);
USING(Face_const_handle);
USING(Vertex_iterator);
USING(Halfedge_iterator);
USING(Face_iterator);
```

[6] an object *a* supports *b* if $embedding(a)$ contains $embedding(b)$.

```
USING(Vertex_const_iterator);
USING(Halfedge_const_iterator);
USING(Face_const_iterator);
USING(Halfedge_around_vertex_circulator);
USING(Halfedge_around_vertex_const_circulator);
USING(Halfedge_around_face_circulator);
USING(Halfedge_around_face_const_circulator);
#undef USING
```

Our constrained triangulation *CT* decorates the 1-skeleton of a plane map consisting just of vertices and edges. We save the face objects there. Each vertex *v* of *CT* is linked to an object of type *VF_pair* which stores the input vertex supporting *v* and potentially an incident face of the input plane map if *v* is isolated (before the triangulation process). Each halfedge *e* of *CT* is linked to an object of type *EF_pair* which stores the input halfedge supporting *e* and the face incident to *e* in the input structure.

⟨*TPL local types*⟩+≡

```
typedef std::pair<Vertex_const_handle,Face_const_handle>   VF_pair;
typedef std::pair<Halfedge_const_handle,Face_const_handle> EF_pair;
```

We associate the above pair structures to the skeleton objects via the available *GenPtr* slot. We have one additional pointer in each object that we can use for temporary information. We use a special class *geninfo* for the expansion and the access. See the manual page of *geninfo* in the appendix for its usage.

⟨*TPL protected members*⟩≡

```
Vertex_const_handle input_vertex(Vertex_const_handle v) const
{ return geninfo<VF_pair>::const_access(CT.info(v)).first; }
Halfedge_const_handle input_halfedge(Halfedge_const_handle e) const
{ return geninfo<EF_pair>::const_access(CT.info(e)).first; }
Face_const_handle input_face(Halfedge_const_handle e) const
{ return geninfo<EF_pair>::const_access(CT.info(e)).second; }
```

The public interface provides construction, access to the underlying triangulation, point location, and ray shooting. Note that the ray shooting is parameterized by a template predicate that allows adaptation of the termination criterion of the ray shooting walk.

⟨*TPL interface*⟩≡

```
PM_point_locator() {
  ⟨TPL persistent module default init⟩
}
PM_point_locator(const Plane_map& P, const Geometry& k = Geometry());
~PM_point_locator();
const Decorator& triangulation() const { return CT; }
⟨TPL location method⟩
⟨TPL ray shooting method⟩
Object_handle walk_in_triangulation(const Point& p) const;
```

Construction and Destruction

We use a constrained triangulation of the input plane map to do ray shooting walks. We leave the input structure *P* unchanged. Therefore we clone the structure into one which we extend to a constrained triangulation. The cloning can be enriched by actions on the cloned objects via a data accessor. See the description of the operation *clone_skeleton* in the manual page of *PMDecorator* for the concept of this data accessor. Our data accessor *CT_link_to_original* does the linkage of input objects to cloned objects. Additionally it takes care that the cloned objects know the faces incident to their supporting input objects. For the used data association method see the manual page of *geninfo*.

⟨*TPL local types*⟩+≡
```
struct CT_link_to_original : Decorator { // CT decorator
  const Decorator& Po;
  CT_link_to_original(const Decorator& P, const Decorator& Poi)
    : Decorator(P), Po(Poi) {}
  void operator()(Vertex_handle vn, Vertex_const_handle vo) const
  { Face_const_handle f;
    if ( Po.is_isolated(vo) ) f = Po.face(vo);
    geninfo<VF_pair>::create(info(vn));
    geninfo<VF_pair>::access(info(vn)) = VF_pair(vo,f);
  }
  void operator()(Halfedge_handle hn, Halfedge_const_handle ho) const
  { geninfo<EF_pair>::create(info(hn));
    geninfo<EF_pair>::access(info(hn)) = EF_pair(ho,Po.face(ho));
  }
};
```

Note that we have two decorators: the $*this$ object decorates the plane map *P*, the internal decorator *CT* works on a cloned plane map on the heap. We extend the cloned representation to a constrained triangulation. Afterwards we locally optimize the weight of the constrained triangulation to obtain a better expected performance for the ray shooting walk. See the implementation of *minimize_weight_CT*( ) for more information on this. Depending on the presence of *LEDA* we also create an additional point location structure based on the slap method using LEDA's persistent dictionaries. See ⟨*TPL persistent module pm init*⟩ in section 5.2.6 for further information.

⟨*TPL construction*⟩≡
```
template <typename PMD, typename GEO>
PM_point_locator<PMD,GEO>::
PM_point_locator(const Plane_map& P, const Geometry& k) :
  Base(P,k), CT(*(new Plane_map),k)
{
  CT.clone_skeleton(P,CT_link_to_original(CT,*this));
  triangulate_CT();
  minimize_weight_CT();
  ⟨TPL persistent module pm init⟩
}
```

Destruction mirrors the allocations within *CT* above. We discard the extended information slots in all objects (vertices and halfedges). Then we discard all objects. Finally we free the memory for the *CT* plane map on the heap. Optionally we clean up the memory of the persistent module.

⟨*TPL destruction*⟩≡

```
template <typename PMD, typename GEO>
PM_point_locator<PMD,GEO>::
~PM_point_locator()
{
  Vertex_iterator vit, vend = CT.vertices_end();
  for (vit = CT.vertices_begin(); vit != vend; ++vit) {
    geninfo<VF_pair>::clear(CT.info(vit));
  }
  Halfedge_iterator eit, eend = CT.halfedges_end();
  for (eit = CT.halfedges_begin(); eit != eend; ++eit) {
    geninfo<EF_pair>::clear(CT.info(eit));
  }
  CT.clear();
  delete &(CT.plane_map());
  ⟨TPL persistent module destruction⟩
}
```

Point location

The following chunk interfaces the basic point location layer depending on the presence of LEDA. *LOCATE_IN_TRIANGULATION* is either *walk_in_triangulation* or a call to the persistent point location structure in ⟨*TPL persistent module members*⟩ of section 5.2.6.

⟨*TPL location method*⟩≡

```
Object_handle locate(const Point& p) const
{
  Object_handle h = LOCATE_IN_TRIANGULATION(p);
  Vertex_const_handle v_triang;
  if ( assign(v_triang,h) ) {
    return input_object(v_triang);
  }
  Halfedge_const_handle e_triang;
  if ( assign(e_triang,h) ) {
    Halfedge_const_handle e = input_halfedge(e_triang);
    if ( e == Halfedge_const_handle() ) // inserted during triangulation
      return make_object(input_face(e_triang));
    int orientation_ = K.orientation(segment(e),p);
    if ( orientation_ == 0 ) return make_object(e);
    if ( orientation_ < 0 )  return make_object(face(twin(e)));
    if ( orientation_ > 0 )  return make_object(face(e));
  }
  assert(0); return h; // compiler warning
}
```

The following function *walk_in_triangulation*(q) provides a fall-back solution to point location. It returns an *Object_handle* f which is either a vertex v with $point(v) == q$ or a halfedge e where e contains q in its relative interior, or one of the two triangles incident to e (or $twin(e)$ respectively) contains q. This semantics is equivalent to the function from the generic point location framework programmed by Sven Thiel [Thi99] which allows us to obtain a halfedge or vertex that contains q or that is vertically below q.

We walk from the embedding $p = point(v)$ of the first vertex $v = vertices\_begin(\,)$ of our structure to the point q that we expect to lie in the interior of the triangulation. We examine the skeleton of the triangulation. We can thereby hit vertices and edges, the latter in two ways. We walk along an edge e when $segment(e)$ and $s = pq$ overlap, or we cross e in its relative interior. Our walk is thus an iteration storing a vertex or an edge and a flag how we traverse them. We store the oriented halfedge e either in direction of s or such that q is left of e.

⟨*TPL interface*⟩+≡
```
enum object_kind { VERTEX, EDGE_CROSSING, EDGE_COLLINEAR };
```

⟨*TPL walk in triangulation*⟩≡
```
template <typename PMD, typename GEO>
CGAL_TYPENAME_MSVC_NULL PM_point_locator<PMD,GEO>::Object_handle
PM_point_locator<PMD,GEO>::walk_in_triangulation(const Point& q) const
{
  Vertex_const_handle v = CT.vertices_begin();
  Halfedge_const_handle e;
  Point p = CT.point(v);
  if ( p == q ) return make_object(v);
  Segment s = K.construct_segment(p,q);
  Direction dir = K.construct_direction(p,q);
  object_kind current = VERTEX;
  while (true) switch ( current ) {
    case VERTEX:
      ⟨TPL segment walk via a vertex⟩
      break;
    case EDGE_CROSSING:
      ⟨TPL segment walk via a crossing edge⟩
      break;
    case EDGE_COLLINEAR:
      ⟨TPL segment walk via a collinear edge⟩
      break;
  }
#if !defined(__BORLANDC__)
  return Object_handle(); // never reached warning acceptable
#endif
}
```

We walk along s and hit a vertex v. If we have reached q we are done. Otherwise we walk the adjacency list to find either the wedge between two edges via which we leave the vertex or we find a halfedge collinear to s. The adjacency list walk is encapsulated in *out_wedge*. We obtain a halfedge *e_out* out of v, either collinear or bounding the wedge right of s. If *e_out* is collinear to s it is the next object. Otherwise we take the edge $next(e\_out)$ closing the wedge to a triangle which we certainly hit next on our walk along s.

⟨*TPL segment walk via a vertex*⟩≡

```
{
  if ( CT.point(v) == q )
    return make_object(v); // stop walking at q
  bool collinear;
  Halfedge_const_handle e_out = CT.out_wedge(v,dir,collinear);
  if (collinear) // ray shoot via e_out
  { e = e_out; current = EDGE_COLLINEAR; }
  else  // ray shoot in wedge left of e_out
  { e = CT.twin(CT.next(e_out)); current = EDGE_CROSSING; }
}
```

When we cross an edge e from right to left it could be that q is on e, or contained in the incident triangle right of it. If neither is true we continue through the incident triangle left of e. There we can hit one of two other edges or the vertex opposite to e. Note however that this vertex can lie beyond q on the directed line through p and q.

⟨*TPL segment walk via a crossing edge*⟩≡

```
{
  if ( !(K.orientation(CT.segment(e),q) > 0) ) // q not left of e
    return make_object(e);
  Vertex_const_handle v_cand = CT.target(CT.next(e));
  int orientation_ = K.orientation(p,q,CT.point(v_cand));
  switch( orientation_ ) {
    case 0:  // collinear
      if ( K.strictly_ordered_along_line(p,q,CT.point(v_cand)) )
        return make_object(e);
      v = v_cand; current = VERTEX; break;
    case +1: // leftturn
      e = twin(next(e)); current = EDGE_CROSSING; break;
    case -1:
      e = twin(previous(e)); current = EDGE_CROSSING; break;
  }
}
```

Hitting a collinear edge is easy to treat. If it does not contain q its target is the next object.

⟨*TPL segment walk via a collinear edge*⟩≡

```
{
  if ( K.strictly_ordered_along_line(
         CT.point(CT.source(e)),q,CT.point(CT.target(e))) )
    return make_object(e);
  v = CT.target(e); current = VERTEX;
}
```

Correctness and Runtime

We summarize the execution properties of the above code in a lemma.

Lemma 5.2.2: Point location in a plane map P of size $O(n)$ based on the persistent module determines the object (vertex, halfedge, face) containing p in expected time $O(\log n)$. The fall back solution *walk_in_triangulation*(p) determines the object (vertex, halfedge, face) containing p in time $O(n)$.

Proof. We omit the details of the persistent point location package. The elaborate treatment can be found in [Thi99]. The constrained triangulation CT of the input plane map P is asymptotically not larger than the plane map $O(n)$. We sketch the intuition about the persistent point location module (PPL). The PPL partitions the plane into at most $2n+1$ slaps defined by vertical lines through the n vertices of CT. The stripes between the lines and the lines are considered as slaps. Each slap is subdivided into convex patches by objects of CT. The patches are stored in a (persistent) dictionary via their bounding skeleton objects. A location within a slap can be determined within logarithmic query time. Now a point location can be done in time $O(2\log n)$ first by a binary search for the correct slap and second by a query of the corresponding dictionary. The construction of the PPL can be done in $O(n\log n)$ by the plane sweep framework described in [Thi99] and is used in section 5.2.6. The partially persistent dictionary implementation of S. Thiel stays within the linear space bound of P and CT.

For the *walk_in_triangulation*(q) query operation note that the initialization is well defined. We start with the first vertex of the triangulation. Then each iteration reaches either q or one of the states *VERTEX*, *EDGE_CROSSING*, *EDGE_COLLINEAR*. Convince yourself by revisiting the code chunks that each of the possible three configurations covers all geometric possibilities that the segment walk along s can encounter. □

Ray shooting

The ray shooting along a segment s works as follows. We first locate $p = s.source()$ in our underlying triangulation. We might hit a vertex, an edge or a face. In the triangulation we do a segment walk starting in p along s similar to the one in *walk_in_triangulation*( ). Only that we have the additional termination criterion of the object predicate M. We examine the skeleton of the triangulation. We can thereby hit vertices and edges, the latter in two kinds. We walk along an edge e when *segment*(e) and s overlap, or we cross e in its relative interior. Our walk is an iteration storing a vertex or an edge and a flag how we traverse them. We store the oriented halfedge either in direction of s or such that *s.target*( ) is left of e.

First we have to get to a starting point by point location. Then we iterate until we hit an object of our input structure as determined by the object predicate M.

⟨*TPL ray shooting method*⟩≡

```
template <typename Object_predicate>
Object_handle ray_shoot(const Segment& s, const Object_predicate& M) const
{
  CGAL_assertion( !K.is_degenerate(s) );
  Point p = K.source(s), q = K.target(s);
  Direction d = K.construct_direction(p,q);
  Vertex_const_handle v;
  Halfedge_const_handle e;
  object_kind current;
  Object_handle h = LOCATE_IN_TRIANGULATION(p);
  ⟨TPL ray shoot initialization⟩
  while (true) switch ( current ) {
```

```
    case VERTEX:
      ⟨TPL ray shoot hits a vertex⟩
      break;
    case EDGE_CROSSING:
      ⟨TPL ray shoot hits a crossing edge⟩
      break;
    case EDGE_COLLINEAR:
      ⟨TPL ray shoot hits a collinear edge⟩
      break;
  }
  // assert(0); return h; // compiler warning
}
```

If we hit a vertex we are done.

⟨*TPL ray shoot initialization*⟩≡

```
if ( assign(v,h) ) {
  current = VERTEX;
}
```

If the result is a halfedge e, it can contain p or e (or its twin) can bound the triangle that contains p in its interior.

⟨*TPL ray shoot initialization*⟩+≡

```
if ( assign(e,h) ) {
  int orientation_ = K.orientation( segment(e), p);
  if ( orientation_ == 0 ) { // p on segment
    ⟨edge e contains the starting point p⟩
  } else { // p not on segment, thus in triangle
    if ( orientation_ < 0  ) e = CT.twin(e);
    // now p left of e
    ⟨edge e bounds the triangle containing p⟩
  }
}
```

In this case we have to handle the cases that s overlaps *segment*(e) or that we shoot into a face adjacent to e.

⟨*edge e contains the starting point p*⟩≡

```
if ( d == CT.direction(e) )
{ current = EDGE_COLLINEAR; }
else if ( d == CT.direction(CT.twin(e)) )
{ e = CT.twin(e); current = EDGE_COLLINEAR; }
else { // crossing
  current = EDGE_CROSSING;
  if ( !(K.orientation(CT.segment(e),q)>0) ) // not leftturn
    e = CT.twin(e);
}
```

If the triangle (underlying face) which contains p fulfills M we return it. Otherwise we have to find the object (vertex or edge) hit by the ray shoot from p along s. Note that it might be that s is totally contained in the triangle in which case we return the *NULL* handle.

⟨*edge e bounds the triangle containing p*⟩≡

```
if ( M(input_face(e)) ) // face mark
  return make_object(input_face(e));
Point p1 = CT.point(CT.source(e)),
      p2 = CT.point(CT.target(e)),
      p3 = CT.point(CT.target(next(e)));
int or1 = K.orientation(p,q,p1);
int or2 = K.orientation(p,q,p2);
int or3 = K.orientation(p,q,p3);
if ( or1 == 0 && !K.leftturn(p1,p2,q) )
{ v = CT.source(e); current = VERTEX; }
else if ( or2 == 0 && !K.leftturn(p2,p3,q) )
{ v = CT.target(e); current = VERTEX; }
else if ( or3 == 0 && !K.leftturn(p3,p1,q) )
{ v = CT.target(CT.next(e)); current = VERTEX; }
else if ( or2 > 0 && or1 < 0 && !K.leftturn(p1,p2,q) )
{ e = CT.twin(e); current = EDGE_CROSSING; }
else if ( or3 > 0 && or2 < 0 && !K.leftturn(p2,p3,q) )
{ e = CT.twin(CT.next(e)); current = EDGE_CROSSING; }
else if ( or1 > 0 && or3 < 0 && !K.leftturn(p3,p1,q) )
{ e = CT.twin(CT.previous(e)); current = EDGE_CROSSING; }
else return Object_handle();
```

Now we are on our walk along s. If we hit a vertex v with $M(v)$ we have found the right object. Otherwise it could be that we have reached q. If not we walk the adjacency list to find either the wedge between two edges via which we leave the vertex or we find a halfedge collinear to s. The adjacency list walk is encapsulated in *out_wedge*( ). We obtain a halfedge *e_out* out of v, either collinear or bounding the wedge right of s. Note that if the wedge represents a face f with $M(f)$ we are done. Otherwise we take the edge *next*(*e_out*) closing the wedge to a triangle which we certainly hit next on our walk along s.

⟨*TPL ray shoot hits a vertex*⟩≡

```
{
  Vertex_const_handle v_org = input_vertex(v);
  if ( M(v_org) ) return make_object(v_org);
  if ( CT.point(v) == q ) return Object_handle();
  // stop walking at q, or determine next object on s:
  bool collinear;
  Halfedge_const_handle e_out = CT.out_wedge(v,d,collinear);
  if (collinear) // ray shoot via e_out
  { e = e_out; current = EDGE_COLLINEAR; }
  else { // ray shoot in wedge left of e_out
    if ( M(input_face(e_out)) )
      return make_object(input_face(e_out));
    e = CT.twin(CT.next(e_out)); current = EDGE_CROSSING;
  }
}
```

When we cross an edge e we can find either the edge or its incident face fulfilling M. Note however that edges that were created during the triangulation process are supported by the input face which they split. Thus we just ignore them in this case. If neither the edge nor the face fulfill M we continue through the triangle that e bounds. There we can hit one of two other edges or the vertex opposite to e.

⟨*TPL ray shoot hits a crossing edge*⟩≡

```
{
  if ( K.orientation(CT.segment(e),q) == 0 )
    return Object_handle();
  Halfedge_const_handle e_org = input_halfedge(e);
  if ( e_org != Halfedge_const_handle() ) { // not a CT edge
    if ( M(e_org) ) return make_object(e_org);
    if ( M(face(e_org)) ) return make_object(face(e_org));
  }
  Vertex_const_handle v_cand = CT.target(CT.next(e));
  int orientation_ = K.orientation(p,q,CT.point(v_cand));
  switch( orientation_ ) {
    case 0:
      v = v_cand; current = VERTEX; break;
    case +1:
      e = CT.twin(CT.next(e)); current = EDGE_CROSSING; break;
    case -1:
      e = CT.twin(CT.previous(e)); current = EDGE_CROSSING; break;
  }
}
```

If we hit a collinear edge we check if the edge was inserted during the triangulation process. If it is a triangulation edge we examine the underlying face, otherwise we check the edge itself. The next object is its target vertex.

⟨*TPL ray shoot hits a collinear edge*⟩≡

```
{
  Halfedge_const_handle e_org = input_halfedge(e);
  if ( e_org == Halfedge_const_handle() ) { // a CT edge
    if ( M(input_face(e)) )
      return make_object(input_face(e));
  } else { // e_org is not a CT edge
    if ( M(e_org) )
      return make_object(e_org);
  }
  if ( K.strictly_ordered_along_line(
         CT.point(CT.source(e)),q,CT.point(CT.target(e))) )
    return Object_handle();
  v = CT.target(e); current = VERTEX;
}
```

Correctness and Runtimes

The following lemma summarizes the properties of the ray-shooting operation.

Lemma 5.2.3: If the segment s is non-degenerate, then $ray\_shoot(s, M)$ determines a generic handle h which is $NULL$ if no object of P that intersects s fulfills $M(h)$[7] and which can be converted to a corresponding vertex, halfedge, or face otherwise. In the latter case the object fulfills $M(h)$ and additionally has minimal distance to $source(s)$. The query time is that of the point location plus the time for the walk in CT.

Proof. The starting point is always the point location of $p = source(s)$. If the object u that contains p does not fulfill $M(u)$, then a walk in the triangulation is started in $\langle$*TPL ray shoot*$\rangle$. This iteration is determined by one of the three geometric configurations $VERTEX, EDGE\_CROSSING, EDGE\_COLLINEAR$. We only have to ensure that the iteration terminates correctly. In each case we either look for an object u hit by a ray along s for which $M(u)$ holds, or we check if we have to end our walk as we have reached $q = target(s)$. □

Note that we cannot bound the stepping for the ray shooting walk besides the trivial linear bound (the size of the plane map and the triangulation). However we will cite some theory and heuristic tests below that provide the hope that the walk is much cheaper.

Finally there are two operations for the objects in the constrained triangulation that return the corresponding objects from the input plane map that support them.

$\langle$*TPL protected members*$\rangle + \equiv$
```
Object_handle input_object(Vertex_const_handle v) const
{ return make_object(input_vertex(v)); }
Object_handle input_object(Halfedge_const_handle e) const
{ Halfedge_const_handle e_org = input_halfedge(e);
  if ( e_org != Halfedge_const_handle() )
    return make_object( e_org );
  // now e_org is not existing
  return make_object( input_face(e) );
}
```

5.2.3 The constrained triangulation

We calculate the constrained triangulation of the cloned skeleton by our constrained triangulation sweep as presented in chapter 4. The class *Constrained_triang_traits*<> obtains three traits models *PMDEC*, *GEOM*, and *NEWEDGE* on instantiation. The two former parameters are the local types *PM_decorator* and *Geometry*. The latter is the data accessor type implemented below.

All edges of the cloned skeleton carry links to the edges of P. We mark additional triangulation edges by a default link to *Halfedge_const_handle*( ) that can be checked after the triangulation process to separate input halfedges (obstacles) from triangulation halfedges. We forward input face links from existing edges of the triangulation to newly created edges. Thereby each newly created edge knows the face that it splits into triangles. This information association and transfer is done by means of *CT_new_edge*.

The following function operator *operator*( )(*Halfedge_handle*&) attributes the newly created triangulation edge e in CT by an additional *EF_pair* in a similar way as the cloning process did it for the input edges. As e does not have a supporting halfedge in P it obtains the default handle. e obtains

[7] Note that the predicate M is defined on vertex, halfedge, and face handles for performance reasons. However we sloppily write $M(h)$ on a generic handle wrapping one of the above.

the mark information from a face that supports it with respect to the input structure. The face usually comes from an adjacent edge of *CT*. If there is none, then we obtain is from the source which was isolated in the input structure before *e* was created.

⟨*TPL protected members*⟩+≡
```
struct CT_new_edge : Decorator {
  const Decorator& _DP;
  CT_new_edge(const Decorator& CT, const Decorator& DP) :
    Decorator(CT), _DP(DP) {}
  void operator()(Halfedge_handle& e) const
  { Halfedge_handle e_from = previous(e);
    Face_const_handle f;
    if ( is_closed_at_source(e) ) // source(e) was isolated before
      f = geninfo<VF_pair>::access(info(source(e))).second;
    else
      f = geninfo<EF_pair>::access(info(e_from)).second;
    mark(e) = _DP.mark(f);
    geninfo<EF_pair>::create(info(e));
    geninfo<EF_pair>::create(info(twin(e)));

    geninfo<EF_pair>::access(info(e)).first =
    geninfo<EF_pair>::access(info(twin(e))).first =
      Halfedge_const_handle();

    geninfo<EF_pair>::access(info(e)).second =
    geninfo<EF_pair>::access(info(twin(e))).second = f;

  }
};
```

Now completing *CT* into a full triangulation is a trivial instantiation of the sweep class. We plug the types into *Constrained_triang_traits*, and that type into our generic sweep framework. Then we create the sweep object and trigger the sweep. Note also how we transport references to the plane map structure and the geometric kernel on construction.

⟨*TPL protected members*⟩+≡
```
void triangulate_CT() const
{
  typedef CGAL::Constrained_triang_traits<
    Decorator,Geometry,CT_new_edge> NCTT;
  typedef CGAL::generic_sweep<NCTT> Constrained_triang_sweep;
  CT_new_edge NE(CT,*this);
  Constrained_triang_sweep T(NE,CT.plane_map(),K); T.sweep();
}
```

5.2.4 Minimizing the triangulation weight

For a plane map or a triangulation let its weight be the sum of the length of all edges. After the calculation of the constrained triangulation we minimize locally the weight of the triangulation by a sequence of flipping operations starting from all non-constrained edges. We first motivate the following procedure by some theory and some experiments.

- Ray shooting by line walks in planar triangulations are worst-case expensive. Agarwal et al [AAS95] show that such a walk can have linear costs in the size of the triangulation though the shoot does not hit any obstacle edge. Aronov and Fortune show in [AF97] that average case stepping cost for random lines can be bounded by $O(\sqrt{n})$ for plane maps of size n if we use triangulations of minimal weight. Unfortunately such triangulations are hard to construct in general. For general points sets the problem is conjectured to be NP-hard. In simple polygons there's a $O(n^3)$ solution by dynamic programming which is hardly practical. Thus currently only approximation or even heuristics offer practical solutions.

- Note that the delaunay triangulation of a point set of size n in general can be a bad approximation of the minimal weight triangulation [Kir80] but it has been shown to be a $O(\log n)$ approximation [Lin86] if the points are universally distributed. However we have no such result for the constrained case.

- Heuristic results on the approximation of minimum weight triangulation are presented by Dickerson et al. in [DMM95]. They present results that propose to use local minimal weight triangulations as an approximations of the general unsolved problem.

- The running time of the following flipping algorithm is worst case quadratic though runtime tests seem to show a subquadratic runtime for random inputs[8]. The quadratic bound for the worst case runtime is based on a result of Hurtado et al. [HNU96] where they show that the graph of triangulations of a simple polygon is connected and has the complexity of $O(n^2)$. Note that the termination is guaranteed due to the fact that for each quadrangle of points we can only choose the minimal diagonal once, thus we never cycle.

⟨TPL protected members⟩+≡

```
void minimize_weight_CT() const
{
  if ( number_of_vertices() < 2 ) return;
  std::list<Halfedge_handle> S;
  /* We maintain a stack |S| of edges containing diagonals
     which might have to be flipped. */
  int flip_count = 0;
  ⟨insert all non constrained edges in S⟩
  while ( !S.empty() ) {
    Halfedge_handle e = S.front(); S.pop_front();
    Halfedge_handle r = twin(e);
    Halfedge_const_handle e_org = input_halfedge(e);
    if ( e_org != Halfedge_const_handle() )
      continue;
    ⟨continue if quadrilateral with diagonal e is non-convex⟩
    ⟨flip if there's a lighter edge⟩
  }
}
```

We insert all non-constrained edges into our stack S. An edge is not constrained if it has no link to a corresponding supporting input edge in P.

[8] see the runtime results for the flipping number in [DMM95]

⟨*insert all non constrained edges in S*⟩≡
```
Halfedge_iterator e;
for (e = CT.halfedges_begin(); e != CT.halfedges_end(); ++(++e)) {
  Halfedge_const_handle e_org = input_halfedge(e);
  if ( e_org != Halfedge_const_handle() )
    continue;
  S.push_back(e);
}
```

A non-constrained edge can only be flipped if it is the diagonal of a convex quadrilateral. We can continue if it is not.

⟨*continue if quadrilateral with diagonal e is non-convex*⟩≡
```
Halfedge_handle e1 = next(r);
Halfedge_handle e3 = next(e);
// e1,e3: edges of quadrilateral with diagonal e
Point a = point(source(e1));
Point b = point(target(e1));
Point c = point(source(e3));
Point d = point(target(e3));
if (! (K.orientation(b,d,a) > 0 && // leftturn
       K.orientation(b,d,c) < 0) ) // rightturn
  continue;
```

We check if for our non-constrained edge *e* the complementary diagonal has less weight. If yes we flip *e*.

⟨*flip if there's a lighter edge*⟩≡
```
if ( K.first_pair_closer_than_second(b,d,a,c) ) { // flip
  Halfedge_handle e2 = next(e1);
  Halfedge_handle e4 = next(e3);
  S.push_back(e1);
  S.push_back(e2);
  S.push_back(e3);
  S.push_back(e4);
  flip_diagonal(e);
  flip_count++;
}
```

5.2.5 The file wrapper

⟨*PM_point_locator.h*⟩≡
```
⟨CGAL Header1⟩
// file          : include/CGAL/Nef_2/PM_point_locator.h
⟨CGAL Header2⟩
#ifndef CGAL_PM_POINT_LOCATOR_H
#define CGAL_PM_POINT_LOCATOR_H
#include <CGAL/basic.h>
```

```
#include <CGAL/Unique_hash_map.h>
#include <CGAL/Object.h>
#include <CGAL/Nef_2/Constrained_triang_traits.h>
#undef _DEBUG
#define _DEBUG 17
#include <CGAL/Nef_2/debug.h>
#include <CGAL/Nef_2/geninfo.h>
⟨Conditional persistent point location inclusion⟩
CGAL_BEGIN_NAMESPACE
⟨Naive point locator⟩
⟨Triangulation point locator⟩
⟨TPL construction⟩
⟨TPL destruction⟩
⟨TPL walk in triangulation⟩
CGAL_END_NAMESPACE
#endif // CGAL_PM_POINT_LOCATOR_H
```

5.2.6 Point location with persistent dictionaries

We add a dynamically allocated point location structure of type *PointLocator*<*T*> working on the constrained triangulation of the class *PM_point_locator*. The used persistent dictionaries are part of LEDA starting version 4.2.

⟨*Conditional persistent point location inclusion*⟩≡

```
#ifdef CGAL_USE_LEDA
#include <LEDA/basic.h>
#if __LEDA__ > 410
#define USING_PPL
#include <CGAL/Nef_2/PM_persistent_PL.h>
#endif
#endif
```

The point location framework is added to the point location class only if the necessary modules are present. The traits class *PM_persistent_PL_traits* is defined below. The point location framework *PointLocator* is described in Sven Thiel's masters thesis [Thi99].

⟨*TPL persistent module members*⟩≡

```
#ifdef USING_PPL
typedef PM_persistent_PL_traits<Base>  PMPPLT;
typedef PointLocator<PMPPLT>           PMPP_locator;
PMPP_locator* pPPL;
#define LOCATE_IN_TRIANGULATION pPPL->locate_down
#else
#define LOCATE_IN_TRIANGULATION walk_in_triangulation
#endif
```

The first parameter of *PMPP_locator* is the graph structure worked on, the second is a traits class object. We forward a reference to the geometric kernel K that we already obtained for the $*this$ object of type *PM_point_locator*.

⟨*TPL persistent module pm init*⟩≡
```
#ifdef USING_PPL
pPPL = new PMPP_locator(CT,PMPPLT(K));
#endif
```

⟨*TPL persistent module default init*⟩≡
```
#ifdef USING_PPL
pPPL = 0;
#endif
```

⟨*TPL persistent module destruction*⟩≡
```
#ifdef USING_PPL
delete pPPL; pPPL=0;
#endif
```

⟨*Triangulation point locator*⟩+≡
```
#ifdef USING_PPL
static const char* pointlocationversion = "point location via pers dicts";
#else
static const char* pointlocationversion = "point location via seg walks";
#endif
```

The rest of this section describes a traits model for the persistent point location framework as developed by Sven Thiel [Thi99]. We adapt his framework for point location via persistent dictionaries within the constrained triangulation. Thereby we obtain a better runtime bound than the fall-back method *walk_in_triangulation*( ).

The class *PM_persistent_PL_traits* transports a bunch of types and methods into the point location data structure *PointLocator* of S. Thiel. Note that the class references a plane map and a geometric kernel, that we both forward on construction.

⟨*class PM_persistent_PL_traits*⟩≡
```
template <typename PMPL>
struct PM_persistent_PL_traits
{
  ⟨PP types in local scope⟩
  ⟨PP iterators and conversion⟩
  ⟨PP edge categorization⟩
  ⟨PP x-structure interface⟩
  ⟨PP curve interface⟩
  ⟨PP generic location and postprocessing⟩
  ⟨PP construction⟩
};
```

⟨*PM_persistent_PL.h*⟩≡
```
⟨CGAL Header1⟩
// file          : include/CGAL/Nef_2/PM_persistent_PL.h
⟨CGAL Header2⟩
#ifndef CGAL_PM_PERSISTENT_PL_H
#define CGAL_PM_PM_PERSISTENT_PL_H
#include <CGAL/Nef_2/gen_point_location.h>
⟨class PM_persistent_PL_traits⟩
#endif // CGAL_PM_PM_PERSISTENT_PL_H
```

The type mapping is trivial. For the persistent point location we need the interface types *Graph*, *Node*, *Edge*. We additionally introduce the geometry.

⟨*PP types in local scope*⟩≡
```
typedef PMPL  Graph;
typedef typename PMPL::Vertex_const_handle   Node;
typedef typename PMPL::Halfedge_const_handle Edge;
typedef typename PMPL::Face_const_handle     Face;
typedef typename PMPL::Object_handle         Object_handle;
typedef typename PMPL::Geometry  Geometry;
typedef typename PMPL::Point     Point;
typedef typename PMPL::Segment   Segment;
const Geometry* pK;
```

We store a reference to the geometric kernel in the traits class.

⟨*PP construction*⟩≡
```
PM_persistent_PL_traits() : pK(0) {}
PM_persistent_PL_traits(const Geometry& k) : pK(&k) {}
virtual ~PM_persistent_PL_traits() {}
virtual void sweep_begin(const Graph&) {}
virtual void sweep_moveto(const XCoord&) {}
virtual void sweep_end() {}
virtual void clear() {}
```

⟨*PP iterators and conversion*⟩≡
```
typedef typename PMPL::Vertex_const_iterator NodeIterator;
NodeIterator Nodes_begin(const Graph& G) const { return G.vertices_begin(); }
NodeIterator Nodes_end(const Graph& G) const { return G.vertices_end(); }
Node toNode(const NodeIterator& nit) const { return nit; }
typedef typename PMPL::Halfedge_around_vertex_const_circulator HAVC;
struct IncEdgeIterator {
  HAVC _start, _curr;
  bool met;
  IncEdgeIterator() {}
  IncEdgeIterator(HAVC c) :
    _start(c), _curr(c), met(false) {}
  IncEdgeIterator& operator++()
```

```
  { if (_curr==_start)
      if (!met)  { met=true; ++_curr; }
      else       { _curr=HAVC(); }
    else ++_curr;
    return *this;
  }
  bool operator==(const IncEdgeIterator& it2) const
  { return _curr==it2._curr; }
  bool operator!=(const IncEdgeIterator& it2) const
  { return !(*this==it2); }
};
Edge toEdge(const IncEdgeIterator& eit) const { return eit._curr; }
IncEdgeIterator IncEdges_begin(const Graph& G, const Node& n)
{ return IncEdgeIterator(HAVC(G.first_out_edge(n))); }
IncEdgeIterator IncEdges_end(const Graph& G, const Node& n)
{ return IncEdgeIterator(); }
```

⟨*PP edge categorization*⟩≡

```
enum EdgeCategory
{ StartingNonVertical, StartingVertical, EndingNonVertical, EndingVertical };
Node opposite(const Graph& G, const Edge& e, const Node& u)
{ if ( G.source(e) == u ) return G.target(e);
  else                    return G.source(e); }
EdgeCategory ClassifyEdge(const Graph& G, const Edge& e, const Node& u)
{
  Point p_u = G.point(u);
  Point p_v = G.point(opposite(G,e,u));
  int cmpX = pK->compare_x(p_u, p_v);
  if ( cmpX < 0 ) return StartingNonVertical;
  if ( cmpX > 0 ) return EndingNonVertical;
  int cmpY = pK->compare_y(p_u, p_v);
  assert(cmpY != 0);
  if ( cmpY < 0 ) return StartingVertical;
  return EndingVertical;
}
```

The generic persistent point location framework maintains slaps of the graph skeleton ordered by the x-coordinates of the embedding of the nodes. We make the *point* type representing its x-coordinate and use *compare_x* from the kernel to sort them.

⟨*PP x-structure interface*⟩≡

```
typedef Point XCoord;
const XCoord getXCoord(const Point& p) const
{ return p; }
const XCoord getXCoord(const Graph& G, const Node& n) const
{ return G.point(n); }
class PredLessThanX {
  const Geometry* pK;
public:
  PredLessThanX() : pK(0) {}
```

```
    PredLessThanX(const Geometry* pKi) : pK(pKi) {}
    PredLessThanX(const PredLessThanX& P) : pK(P.pK)
    {  }
    int operator() (const XCoord& x1, const XCoord& x2) const
    { return pK->compare_x(x1,x2) < 0; }
  };
  PredLessThanX getLessThanX() const { return PredLessThanX(pK); }
```

In our case curves are segments. We obtain them from the geometric kernel.

⟨*PP curve interface*⟩≡

```
  // Curve connected functionality:
  typedef Segment  Curve;
  Curve makeCurve(const Point& p) const
  { return pK->construct_segment(p,p); }
  Curve makeCurve(const Graph& G, const Node& n) const
  { return makeCurve(G.point(n)); }
  Curve makeCurve(const Graph& G, const Edge& e) const
  { Point ps = G.point(G.source(e)), pt = G.point(G.target(e));
    Curve res(G.point(G.source(e)),G.point(G.target(e)));
    if ( pK->compare_xy(ps,pt) < 0 ) res = pK->construct_segment(ps,pt);
    else                             res = pK->construct_segment(pt,ps);
    return res;
  }
  struct PredCompareCurves {
   const Geometry* pK;
   PredCompareCurves() : pK(0) {}
   PredCompareCurves(const Geometry* pKi) : pK(pKi) {}
   PredCompareCurves(const PredCompareCurves& P) : pK(P.pK) {}
   int cmppntseg(const Point& p, const Curve& s) const
   {
     if ( pK->compare_x(pK->source(s),pK->target(s)) != 0 ) // !vertical
       return pK->orientation(pK->source(s),pK->target(s), p);
     if ( pK->compare_y(p,pK->source(s)) <= 0 ) return -1;
     if ( pK->compare_y(p,pK->target(s)) >= 0 ) return +1;
     return 0;
   }
   int operator()(const Curve& s1, const Curve& s2) const
   {
     Point a = pK->source(s1);
     Point b = pK->target(s1);
     Point c = pK->source(s2);
     Point d = pK->target(s2);
     if ( a==b )
       if ( c==d ) return pK->compare_y(a,c);
       else        return  cmppntseg(a, s2);
     if ( c==d )   return -cmppntseg(c, s1);
     // now both are non-trivial:
     int cmpX = pK->compare_x(a, c);
     if ( cmpX < 0 )
       return - pK->orientation(a,b,c);
```

```
    if ( cmpX > 0 )
      return   pK->orientation(c,d,a);
    int cmpY = pK->compare_y(a, c);
    if ( cmpY < 0 ) return -1;
    if ( cmpY > 0 )  return +1;
    // cmpX == cmpY == 0 => a == c
    return pK->orientation(c,d,b);
  }
};
PredCompareCurves getCompareCurves() const
{ return PredCompareCurves(pK); }
```

The generic location type *GenericLocation* is defined in the generic point location framework. The framework allows us to introduce a postprocessing phase after the actual location phase. From the location phase we obtain two locations in two slaps, which are neigbored in the x-structure. The second location *L_plus* is non-nil if the first slap has width zero and is defined by a node at an x-coordinate x. Then *L_plus* returns an edge e just below $x+\varepsilon$. This allows us to extract the face that we search from e.

⟨*PP generic location and postprocessing*⟩≡

```
typedef GenericLocation<Node, Edge> Location;
typedef Object_handle QueryResult;
virtual Object_handle
PostProcess(const Location& L, const Location& L_plus,
  const Point& p) const
{ /* we only get an L_plus (non-nil) if L is ABOVE a vertex
     in which case we want to extract the face from the edge
     below (p+epsilon) available via L_plus. */
  if (!L_plus.is_nil()) { assert(L_plus.is_edge());
    return CGAL::make_object(Edge(L_plus));
  } else {
    if ( L.is_edge() ) {
      return CGAL::make_object(Edge(L));
    }
    if ( L.is_node() ) {
      Node v(L); assert( v != Node() );
      return CGAL::make_object(v);
    }
    return Object_handle();
  }
}
```

5.3 A Test Case

⟨*PM_point_locator-test.C*⟩≡

```
#include <CGAL/basic.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Homogeneous.h>
```

```
#include "Affine_geometry.h"
#include <CGAL/Nef_2/HalfedgeDS_default.h>
#include <CGAL/test_macros.h>
#include <CGAL/Nef_2/HDS_items.h>
#include <CGAL/Nef_2/PM_const_decorator.h>
#include <CGAL/Nef_2/PM_decorator.h>
#include <CGAL/Nef_2/PM_io_parser.h>
#include <CGAL/Nef_2/PM_visualizor.h>
#include <CGAL/Nef_2/PM_overlayer.h>
#include <CGAL/Nef_2/PM_point_locator.h>

// KERNEL:
typedef CGAL::Homogeneous<leda_integer>   Hom_kernel;
typedef CGAL::Affine_geometry<Hom_kernel> Aff_kernel;
typedef Aff_kernel::Segment_2 Segment;

// HALFEDGE DATA STRUCTURE:
struct HDS_traits {
  typedef Aff_kernel::Point_2 Point;
  typedef bool Mark;
};
typedef  CGAL::HalfedgeDS_default<HDS_traits,HDS_items> Plane_map;
typedef  CGAL::PM_const_decorator<Plane_map> CDecorator;
typedef  CGAL::PM_decorator<Plane_map>       Decorator;
typedef  CGAL::PM_overlayer<Decorator,Aff_kernel> PM_aff_overlayer;

typedef CDecorator::Halfedge_const_handle  Halfedge_const_handle;
typedef CDecorator::Vertex_const_handle    Vertex_const_handle;
typedef CDecorator::Face_const_handle      Face_const_handle;
typedef Decorator::Halfedge_handle         Halfedge_handle;
typedef Decorator::Vertex_handle           Vertex_handle;

// INPUT:
typedef  std::list&lt;Segment&gt;::const_iterator Iterator;

struct Object_DA {
  const Decorator& D;
  Object_DA(const Decorator& Di) : D(Di) {}
  void supporting_segment(Halfedge_handle e, Iterator it) const
  { D.mark(e) = true; }
  void trivial_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
  void starting_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
  void passing_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
  void ending_segment(Vertex_handle v, Iterator it) const
  { D.mark(v) = true; }
};

// POINT LOCATION:
typedef CGAL::PM_naive_point_locator<Decorator,Aff_kernel> PM_naive_PL;
typedef CGAL::PM_point_locator<Decorator,Aff_kernel>       PM_triang_PL;
typedef PM_naive_PL::Object_handle Object_handle;

void print(Object_handle h)
{
  Vertex_const_handle v;
```

```
  Halfedge_const_handle e;
  Face_const_handle f;
  if (assign(v,h)) cout << " ohandle = vertex " << PV(v) << endl;
  if (assign(e,h)) cout << " ohandle = halfedge " << PE(e) << endl;
  if (assign(f,h)) cout << " ohandle = face " << PE(f->halfedge()) << endl;
}
// struct INSET {  bool operator()(bool b) const { return b; } };
struct INSET {
  const CDecorator& D;
  INSET(const CDecorator& Di) : D(Di) {}
  bool operator()(Vertex_const_handle v) const { return D.mark(v); }
  bool operator()(Halfedge_const_handle e) const { return D.mark(e); }
  bool operator()(Face_const_handle f) const { return D.mark(f); }
};
int main(int argc, char* argv[])
{
  SETDTHREAD(17);
  CGAL::set_pretty_mode(cerr);
  CGAL::Window_stream W;
  W.init(-50,50,-50,1);
  W.display();
  W.message("Insert segments to create a map.");
  Plane_map H;
  PM_aff_overlayer PMOV(H,Aff_kernel());
  CGAL::PM_visualizor<PM_aff_overlayer,Aff_kernel> V(W,PMOV);
  std::list&lt;Segment&gt; L;
  Segment s;
  if ( argc == 2 ) {
    std::ifstream log(argv[1]);
    while ( log >> s ) { L.push_back(s); W << s; }
  }
  while ( W >> s ) L.push_back(s);
  std::string fname(argv[0]);
  fname += ".log";
  std::ofstream log(fname.c_str());
  for (Iterator sit = L.begin(); sit != L.end(); ++sit)
    log << *sit << " ";
  log.close();
  Object_DA ODA(PMOV);
  PMOV.create(L.begin(),L.end(),ODA);
  V.draw_skeleton();
  W.message("Insert one segment for point location");
  PM_naive_PL  PL1(H);
  PM_triang_PL PL2(H);
  Object_handle h;
  W >> s;
  CGAL::PM_io_parser<PM_aff_overlayer>::dump(PMOV);
  h = PL1.locate(s);
  print(h);
  h = PL2.locate(s.source());
  print(h);
  h = PL1.ray_shoot(s,INSET(PL1));
```

```
  h = PL2.ray_shoot(s,INSET(PL2));
  W.read_mouse();
  return 0;
}
```

Bibliography

[AAS95] P. K. Agarwal, B. Aronov, and S. Suri. Stabbing triangulations by lines in 3D. In *Proceedings of the 11th Annual Symposium on Computational Geometry*, pages 267–276. ACM Press, 1995.

[AF97] B. Aronov and S. Fortune. Average-case ray shooting and minimum weight triangulations. In *Proceedings of the 13th International Annual Symposium on Computational Geometry (SCG-97)*, pages 203–211. ACM Press, 1997.

[DMM95] M.T. Dickerson, S.A. McElfresh, and M. Montague. New algorithms and empirical findings on minimum weight triangulation heuristics. In *Proceedings of the 11th Annual Symposium on Computational Geometry*, pages 238–247. ACM Press, 1995.

[HNU96] F. Hurtado, M. Noy, and J. Urrutia. Flipping edges in triangulations. In *Proceedings of the Twelfth Annual Symposium On Computational Geometry (ISG '96)*, pages 214–223. ACM Press, 1996.

[Kir80] D.G. Kirkpatrick. A note on Delaunay and optimal triangulations. *Information Processing Letters*, 10(3):127–128, 1980.

[Lin86] A. Lingas. The greedy and delauney triangulations are not bad in the average case. *Information Processing Letters*, 22, 1986.

[Thi99] S. Thiel. Persistente Suchbäume. Master thesis, Universität des Saarlandes, 1999.

6 Polynomials in one Variable

6.1 The Manual Page

6.1.1 Polynomials in one variable (RPolynomial)

1. Definition

An instance p of the data type *RPolynomial<NT>* represents a polynomial $p = a_0 + a_1 x + \dots a_d x^d$ from the ring $NT[x]$. The data type offers standard ring operations and a sign operation which determines the sign for the limit process $x \to \infty$.

$NT[x]$ becomes a unique factorization domain, if the number type *NT* is either a field type (1) or a unique factorization domain (2). In both cases there's a polynomial division operation defined.

2. Types

RPolynomial<NT>::NT	the component type representing the coefficients.
RPolynomial<NT>::const_iterator	a random access iterator for read-only access to the coefficient vector.

3. Creation

RPolynomial<NT> p;

introduces a variable p of type *RPolynomial<NT>* of undefined value.

RPolynomial<NT> p(*NT a0*);

introduces a variable p of type *RPolynomial<NT>* representing the constant polynomial a_0.

RPolynomial<NT> p(*NT a0*, *NT a1*);

introduces a variable p of type *RPolynomial<NT>* representing the polynomial $a_0 + a_1 x$.

RPolynomial<NT> p(*NT a0*, *NT a1*, *NT a2*);

introduces a variable p of type *RPolynomial<NT>* representing the polynomial $a_0 + a_1 x + a_2 x^2$.

template <*class Forward_iterator*>
RPolynomial<NT> p(*Forward_iterator first*, *Forward_iterator last*);

introduces a variable p of type *RPolynomial<NT>* representing the polynomial whose coefficients are determined by the iterator range, i.e. let $(a_0 = *first, a_1 = *{+}{+}first, \dots a_d = *it)$, where $++it == last$ then p stores the polynomial $a_1 + a_2 x + \dots a_d x^d$.

4. Operations

int	p.degree( )	the degree of the polynomial.

const NT&	$p[$*unsigned int i*$]$	the coefficient a_i of the polynomial.
const_iterator	*p*.begin( )	a random access iterator pointing to a_0.
const_iterator	*p*.end( )	a random access iterator pointing beyond a_d.
NT	*p*.eval_at(*NT R*)	evaluates the polynomial at *R*.
CGAL::*Sign*	*p*.sign( )	returns the sign of the limit process for $x \to \infty$ (the sign of the leading coefficient).
bool	*p*.is_zero( )	returns true iff p is the zero polynomial.
RPolynomial<*NT*>	*p*.abs( )	returns $-p$ if $p.sign(\,) == NEGATIVE$ and p otherwise.
NT	*p*.content( )	returns the content of p (the gcd of its coefficients). *Precondition*: Requires *NT* to provide a *gdc* operation.

Additionally *RPolynomial*<*NT*> offers standard arithmetic ring opertions like $+,-,*,+=,-=,*=$. By means of the sign operation we can also offer comparison predicates as $<,>,\leq,\geq$. Where $p_1 < p_2$ holds iff $sign(p_1 - p_2) < 0$. This data type is fully compliant to the requirements of CGAL number types.

RPolynomial<*NT*> *p1* / *p2*

implements polynomial division of *p1* and *p2*. if $p1 = p2 * p3$ then *p2* is returned. The result is undefined if *p3* does not exist in $NT[x]$. The correct division algorithm is chosen according to a traits class *ring_or_field*<*NT*> provided by the user. If $ring\_or\_field\langle NT\rangle::kind == ring\_with\_gcd$ then the division is done by *pseudo division* based on a *gcd* operation of NT. If $ring\_or\_field\langle NT\rangle::kind == field\_with\_div$ then the division is done by *euclidean division* based on the division operation of the field NT.
Note that $NT = int$ quickly leads to overflow errors when using this operation.

Non member functions

RPolynomial<*NT*> gcd(*RPolynomial*<*NT*> *p1*, *RPolynomial*<*NT*> *p2*)

returns the greatest common divisor of *p1* and *p2*. **Note** that $NT = int$ quickly leads to overflow errors when using this operation. *Precondition*: Requires *NT* to be a unique factorization domain, i.e. to provide a *gdc* operation.

void pseudo_div(*RPolynomial*<*NT*> f, *RPolynomial*<*NT*> g, *RPolynomial*<*NT*>& q, *RPolynomial*<*NT*>& r, *NT*& D)

implements division with remainder on polynomials of the ring $NT[x]$: $D * f = g * q + r$. *Precondition*: *NT* is a unique factorization domain, i.e., there exists a *gcd* operation and an integral division operation on NT.

void euclidean_div(*RPolynomial*<*NT*> f, *RPolynomial*<*NT*> g, *RPolynomial*<*NT*>& q, *RPolynomial*<*NT*>& r)

implements division with remainder on polynomials of the ring $NT[x]$: $f = g * q + r$. *Precondition*: *NT* is a field, i.e., there exists a division operation on NT.

5. Implementation

This data type is implemented as an item type via a smart pointer scheme. The coefficients are stored in a vector of *NT* entries. The simple arithmetic operations $+,-$ take time $O(d * T(NT))$, multiplication is quadratic in maximal degree of the arguments times $T(NT)$, where $T(NT)$ is the time for a corresponding operation on two instances of the ring type.

Range template

template <*class Forward_iterator*>

typename std::*iterator_traits*<*Forward_iterator*>::*value_type*
gcd_of_range(*Forward_iterator its*, *Forward_iterator ite*)

calculates the greates common divisor of the set of numbers $\{*its, *{+}{+}its, \ldots, *it\}$ of type *NT*, where $++it == ite$ and *NT* is the value type of *Forward_iterator*. *Precondition*: there exists a pairwise gcd operation $NT\ gcd(NT, NT)$ and $its\ != ite$.

6.2 Introduction

We present the implementation of a simple polynomial type *RPolynomial* in one variable. The interface is specified in the manual page on page 197. Let *NT* be a either a field number type or an Euclidean ring number type[1]. We use $NT[x]$ to represent the polynomial ring in one variable. For a polynomial $p = \sum_{i=0}^{d} a_i x^i \in NT[x]$ we store its coefficients along its rising exponents $coeff[i] = a_i$ in a vector of size $d+1$. We keep the invariant that $a_d \neq 0$ and do not allow modifying access to the coefficients via the interface. Flexibility in the creation of polynomials is achieved via iterator ranges which can specify a sequence of coefficients of a polynomial. We offer basic arithmetic operations like $+, -, *$, as well as destructive self modifying operations $+=, -=, *=$. When working destructively we need a cloning scheme to cope with the alias effects of one common representation referenced by several handles. For number types that are fields or Euclidean rings we also offer polynomial division. For the field types this can be done directly by so called Euclidean division. For the number types that are Euclidean rings we provide it via so called pseudo division. Based on that operation we also provide a gcd-operation on the ring $NT[x]$.

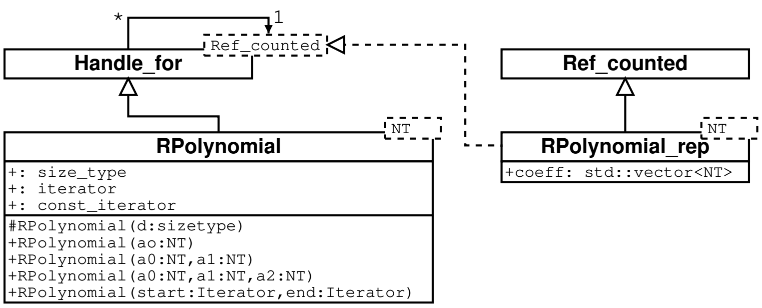


Figure 6.1: Some details of the handle scheme. *RPolynomial_rep*<*NT*> is derived from *Ref_counted* and thereby a model of the template parameter *T* of *Handle_for*<*T*>. It stores the coefficients in an STL vector *coeff*. *RPolynomial*<*NT*> is derived from *Handle_for*< *RPolynomial_rep*<*NT*> >. A *Ref_counted* object carries the reference variable, *Handle_for*<*T*> provides the copy construction and assignment mechanisms.

Implementation

Polynomials are implemented by using a smart-pointer scheme. First we implement the common representation class storing an *NT* vector. The whole smart-pointer scheme is shown in Figure 6.1. For the representation class we keep the invariant that the coefficient vector *coeff* is always reduced such that the highest-order entry is nonzero except when the degree is zero. In this case we allow a

[1] An Euclidean ring type is a ring that additionally offers division with remainder and as such is a unique factorization domain.

zero entry. By doing so the degree of the polynomial is always $coeff.size() - 1$. To keep our invariant we *reduce* the coefficient representation after each construction and arithmetic modification, which basically means we shrink the coefficient vector until the last entry is nonzero or until the polynomial is constant.

⟨*general rep template*⟩≡
```
template <class pNT> class RPolynomial_rep : public Ref_counted
{
  ⟨storage members and types⟩
  ⟨rep interface⟩
};
```

The coefficients are stored in an STL vector. Its resizing operations are usefull when dealing with polynomials of different degree.

⟨*storage members and types*⟩≡
```
typedef pNT NT;
typedef std::vector<NT> Vector;
#else
typedef CGAL::vector_MSC<NT> Vector;
#endif
typedef typename Vector::size_type      size_type;
typedef typename Vector::iterator       iterator;
typedef typename Vector::const_iterator const_iterator;
Vector coeff;
```

The interface allows direct initialization up to degree 3, and via iterator ranges for higher degree.

⟨*rep interface*⟩≡
```
RPolynomial_rep() : coeff() {}
RPolynomial_rep(const NT& n) : coeff(1) { coeff[0]=n; }
RPolynomial_rep(const NT& n, const NT& m) : coeff(2)
  { coeff[0]=n; coeff[1]=m; }
RPolynomial_rep(const NT& a, const NT& b, const NT& c) : coeff(3)
  { coeff[0]=a; coeff[1]=b; coeff[2]=c; }
RPolynomial_rep(size_type s) : coeff(s,NT(0)) {}

template <class Forward_iterator>
RPolynomial_rep(Forward_iterator first, Forward_iterator last SNIHACK)
  : coeff(first,last) {}
#else
template <class Forward_iterator>
RPolynomial_rep(Forward_iterator first, Forward_iterator last SNIHACK)
  : coeff()
{ while (first!=last) coeff.push_back(*first++); }
#endif
```

The *pop_back*( ) operation of the STL vector nicely supports the *reduce*( ) method.

⟨*rep interface*⟩+≡
```
void reduce()
{ while ( coeff.size()>1 && coeff.back()==NT(0) ) coeff.pop_back(); }
```

⟨*rep interface*⟩+≡
```
friend class RPolynomial<pNT>;
friend class RPolynomial<int>;
friend class RPolynomial<double>;
friend std::istream& operator >> CGAL_NULL_TMPL_ARGS
       (std::istream&, RPolynomial<NT>&);
```

Now for the general template class derived from the generic handle type wrapping the smart pointer architecture.

⟨*the polynomial class template*⟩≡
```
template <class pNT> class RPolynomial :
  public Handle_for< RPolynomial_rep<pNT> >
{
  ⟨interface types⟩
  ⟨protected interface⟩
  ⟨construction and destruction⟩
  ⟨public interface⟩
  ⟨friend functions and operations⟩
  ⟨self modifying operations⟩
  ⟨offset multiplication⟩
};
```

The iterator is obtained from the vector data type.

⟨*interface types*⟩≡
```
public:
typedef pNT NT;
typedef Handle_for< RPolynomial_rep<NT> > Base;
typedef RPolynomial_rep<NT> Rep;
typedef typename Rep::Vector    Vector;
typedef typename Rep::size_type size_type;
typedef typename Rep::iterator  iterator;
typedef typename Rep::const_iterator const_iterator;
```

We provide a bunch of operations to implement the arithmetic operations. The *reduce*( ) ensures our invariant that the leading coefficient is nonzero. The static member *R_* is used for the evaluation of a set of polynomials at a constant value.

⟨*protected interface*⟩≡
```
protected:
void reduce() { ptr->reduce(); }
Vector& coeffs() { return ptr->coeff; }
const Vector& coeffs() const { return ptr->coeff; }
```

```
RPolynomial(size_type s) : Base( RPolynomial_rep<NT>(s) ) {}
// creates a polynomial of degree s-1
static NT R_; // for visualization only
```

Each constructor creates a representation object on the heap and initializes it according to the specification. Assignment is handled by the handle base class which just creates an additional link to the representation object. *copy_on_write*( ) allows us to single out a cloned representation object when we want to manipulate the value of a polynomial without alias effects.

⟨*construction and destruction*⟩≡

```
public:
RPolynomial()
  : Base( RPolynomial_rep<NT>() ) {}
RPolynomial(const NT& a0)
  : Base(RPolynomial_rep<NT>(a0)) { reduce(); }
RPolynomial(const NT& a0, const NT& a1)
  : Base(RPolynomial_rep<NT>(a0,a1)) { reduce(); }
RPolynomial(const NT& a0, const NT& a1,const NT& a2)
  : Base(RPolynomial_rep<NT>(a0,a1,a2)) { reduce(); }
template <class Forward_iterator>
RPolynomial(Forward_iterator first, Forward_iterator last)
  : Base(RPolynomial_rep<NT>(first,last)) { reduce(); }
#else
#define RPOL(I)\
RPolynomial(I first, I last) : \
Base(RPolynomial_rep<NT>(first,last SNIINST)) { reduce(); }
RPOL(const NT*)
RPOL(const int*)
RPOL(const double*)
#undef RPOL
#endif // CGAL_SIMPLE_NEF_INTERFACE
```

There are also specialized construtors for the builtin types *int* and *double* that we don't show.

⟨*construction and destruction*⟩+≡

```
RPolynomial(double n) : Base(RPolynomial_rep<NT>(NT(n))) { reduce(); }
RPolynomial(double n1, double n2)
  : Base(RPolynomial_rep<NT>(NT(n1),NT(n2))) { reduce(); }

RPolynomial(int n) : Base(RPolynomial_rep<NT>(NT(n))) { reduce(); }
RPolynomial(int n1, int n2)
  : Base(RPolynomial_rep<NT>(NT(n1),NT(n2))) { reduce(); }
RPolynomial(const RPolynomial<NT>& p) : Base(p) {}
protected: // accessing coefficients internally:
NT& coeff(unsigned int i)
{ CGAL_assertion(!ptr->is_shared() && i<(ptr->coeff.size()));
```

```
    return ptr->coeff[i];
  }
  public:
```

In this chunk we present the method interface of *RPolynomial*<>. The array operators offer read-only access. For manipulation we offer a protected method *coeff*(*int*), that is only for internal use. Evaluation, sign, and content are simple to realize.

⟨*public interface*⟩≡

```
  const_iterator begin() const { return ptr->coeff.begin(); }
  const_iterator end()   const { return ptr->coeff.end(); }
  int degree() const
  { return ptr->coeff.size()-1; }
  const NT& operator[](unsigned int i) const
  { CGAL_assertion( i<(ptr->coeff.size()) );
    return ptr->coeff[i]; }
  const NT& operator[](unsigned int i)
  { CGAL_assertion( i<(ptr->coeff.size()) );
    return ptr->coeff[i]; }
  NT eval_at(const NT& r) const
  { CGAL_assertion( degree()>=0 );
    NT res = ptr->coeff[0], x = r;
    for(int i=1; i<=degree(); ++i)
    { res += ptr->coeff[i]*x; x*=r; }
    return res;
  }
  CGAL::Sign sign() const
  { const NT& leading_coeff = ptr->coeff.back();
    if (leading_coeff < NT(0)) return (CGAL::NEGATIVE);
    if (leading_coeff > NT(0)) return (CGAL::POSITIVE);
    return CGAL::ZERO;
  }
  bool is_zero() const
  { return degree()==0 && ptr->coeff[0]==NT(0); }
  RPolynomial<NT> abs() const
  { if ( sign()==CGAL::NEGATIVE ) return -*this; return *this; }

  NT content() const
  { CGAL_assertion( degree()>=0 );
    return gcd_of_range(ptr->coeff.begin(),ptr->coeff.end());
  }
```

⟨*public interface*⟩+≡

```
  #else // CGAL_SIMPLE_NEF_INTERFACE
  NT content() const
  { CGAL_assertion( degree()>=0 );
    iterator its=ptr->coeff.begin(),ite=ptr->coeff.end();
    NT res = *its++;
```

```
    for(; its!=ite; ++its) res =
      (*its==0 ? res : ring_or_field<NT>::gcd(res, *its));
    if (res==0) res = 1;
    return res;
  }
  #endif
  static void set_R(const NT& R) { R_ = R; }
```

⟨*friend functions and operations*⟩≡

```
  friend   RPolynomial<NT>
    operator - CGAL_NULL_TMPL_ARGS  (const RPolynomial<NT>&);
  friend  RPolynomial<NT>
    operator + CGAL_NULL_TMPL_ARGS (const RPolynomial<NT>&,
                                    const RPolynomial<NT>&);
  friend  RPolynomial<NT>
    operator - CGAL_NULL_TMPL_ARGS (const RPolynomial<NT>&,
                                    const RPolynomial<NT>&);
  friend  RPolynomial<NT>
    operator * CGAL_NULL_TMPL_ARGS (const RPolynomial<NT>&,
                                    const RPolynomial<NT>&);
  friend
  RPolynomial<NT>  operator / CGAL_NULL_TMPL_ARGS
  (const RPolynomial<NT>& p1, const RPolynomial<NT>& p2);
  static RPolynomial<NT> gcd
    (const RPolynomial<NT>& p1, const RPolynomial<NT>& p2);
  static void pseudo_div
    (const RPolynomial<NT>& f, const RPolynomial<NT>& g,
     RPolynomial<NT>& q, RPolynomial<NT>& r, NT& D);
  static void euclidean_div
    (const RPolynomial<NT>& f, const RPolynomial<NT>& g,
     RPolynomial<NT>& q, RPolynomial<NT>& r);
  friend  double to_double
  CGAL_NULL_TMPL_ARGS (const RPolynomial<NT>& p);
```

We allow self manipulating operations of a polynomial. Note that we have to ensure that we avoid alias effects by a clone call when several front end handles point to one backend rep.

⟨*self modifying operations*⟩≡

```
  RPolynomial<NT>& operator += (const RPolynomial<NT>& p1)
  { copy_on_write();
    int d = std::min(degree(),p1.degree()), i;
    for(i=0; i<=d; ++i) coeff(i) += p1[i];
    while (i<=p1.degree()) ptr->coeff.push_back(p1[i++]);
    reduce(); return (*this); }
  RPolynomial<NT>& operator -= (const RPolynomial<NT>& p1)
  { copy_on_write();
    int d = std::min(degree(),p1.degree()), i;
    for(i=0; i<=d; ++i) coeff(i) -= p1[i];
```

```
  while (i<=p1.degree()) ptr->coeff.push_back(-p1[i++]);
  reduce(); return (*this); }
RPolynomial<NT>& operator *= (const RPolynomial<NT>& p1)
{ (*this)=(*this)*p1; return (*this); }
RPolynomial<NT>& operator /= (const RPolynomial<NT>& p1)
{ (*this)=(*this)/p1; return (*this); }
```

We need the same operations with arguments of type *const NT&* to avoid ambiguity errors with the compiler. We do not show their similar definition.

⟨*self modifying operations*⟩+≡

```
RPolynomial<NT>& operator += (const NT& num)
{ copy_on_write();
  coeff(0) += (NT)num; return *this; }
RPolynomial<NT>& operator -= (const NT& num)
{ copy_on_write();
  coeff(0) -= (NT)num; return *this; }
RPolynomial<NT>& operator *= (const NT& num)
{ copy_on_write();
  for(int i=0; i<=degree(); ++i) coeff(i) *= (NT)num;
  reduce(); return *this; }
RPolynomial<NT>& operator /= (const NT& num)
{ copy_on_write(); CGAL_assertion(num!=0);
  for(int i=0; i<=degree(); ++i) coeff(i) /= (NT)num;
  reduce(); return *this; }
```

Arithmetic Ring Operations

Next we come to the implementation of basic arithmetic operations. The negation is trivial. We just iterate over the coefficient array and invert each sign.

⟨*polynomial implementation*⟩≡

```
template <class NT>
RPolynomial<NT> operator - (const RPolynomial<NT>& p)
{
  CGAL_assertion(p.degree()>=0);
  RPolynomial<NT> res(p.coeffs().begin(),p.coeffs().end());
  typename RPolynomial<NT>::iterator it, ite=res.coeffs().end();
  for(it=res.coeffs().begin(); it!=ite; ++it) *it = -*it;
  return res;
}
```

Addition $p1 + p2$ is also easy. Just add all coefficients of the two monomials with the same degree. Note however that the polynomials themselves might have different degree, such that we have to copy all coefficients in the range $min(d_{p1}, d_{p2}) + 1$ up to $max(d_{p1}, d_{p2})$ into the result. Afterwards we have to reduce the coefficient vector. The subtraction routine is symmetric. We only have to deal with the different sign of *p2*.

⟨*polynomial implementation*⟩+≡
```
template <class NT>
RPolynomial<NT> operator + (const RPolynomial<NT>& p1,
                            const RPolynomial<NT>& p2)
{
  typedef typename RPolynomial<NT>::size_type size_type;
  CGAL_assertion(p1.degree()>=0 && p2.degree()>=0);
  bool p1d_smaller_p2d = p1.degree() < p2.degree();
  int min,max,i;
  if (p1d_smaller_p2d) { min = p1.degree(); max = p2.degree(); }
  else                 { max = p1.degree(); min = p2.degree(); }
  RPolynomial<NT>  p( size_type(max + 1));
  for (i = 0; i <= min; ++i ) p.coeff(i) = p1[i]+p2[i];
  if (p1d_smaller_p2d)  for (; i <= max; ++i ) p.coeff(i)=p2[i];
  else /* p1d >= p2d */ for (; i <= max; ++i ) p.coeff(i)=p1[i];
  p.reduce();
  return p;
}
```

Subtraction is symmetric to the addition.

⟨*polynomial implementation*⟩+≡
```
template <class NT>
RPolynomial<NT> operator - (const RPolynomial<NT>& p1,
                            const RPolynomial<NT>& p2)
{
  typedef typename RPolynomial<NT>::size_type size_type;
  CGAL_assertion(p1.degree()>=0 && p2.degree()>=0);
  bool p1d_smaller_p2d = p1.degree() < p2.degree();
  int min,max,i;
  if (p1d_smaller_p2d) { min = p1.degree(); max = p2.degree(); }
  else                 { max = p1.degree(); min = p2.degree(); }
  RPolynomial<NT>  p( size_type(max+1) );
  for (i = 0; i <= min; ++i ) p.coeff(i)=p1[i]-p2[i];
  if (p1d_smaller_p2d)  for (; i <= max; ++i ) p.coeff(i)= -p2[i];
  else /* p1d >= p2d */ for (; i <= max; ++i ) p.coeff(i)=  p1[i];
  p.reduce();
  return p;
}
```

Multiplication is also straightforward. The degree formula tells us that $p1 * p2$ has degree $p1.degree() + p2.degree()$. We just allocate a polynomial p of corresponding size initialized to zero and add the products of all monomials $p1[i] * p2[j]$ for $0 \le i \le d_{p1}$, $0 \le j \le d_{p2}$ slotwise to a_{i+j}.

⟨*polynomial implementation*⟩+≡
```
template <class NT>
RPolynomial<NT> operator * (const RPolynomial<NT>& p1,
                            const RPolynomial<NT>& p2)
```

```
{
  typedef typename RPolynomial<NT>::size_type size_type;
  CGAL_assertion(p1.degree()>=0 && p2.degree()>=0);
  RPolynomial<NT>  p( size_type(p1.degree()+p2.degree()+1) );
  // initialized with zeros
  for (int i=0; i <= p1.degree(); ++i)
    for (int j=0; j <= p2.degree(); ++j)
      p.coeff(i+j) += (p1[i]*p2[j]);
  p.reduce();
  return p;
}
```

Polynomial Division and Reduction

Next we implement polynomial division operations. See also the books of Cohen [Coh93] or Knuth [Knu98] for a thourough treatment. The result of our division operation p_1/p_2 in $NT[x]$ is defined as the polynomial p_3 such that $p_1 = p_2 p_3$. In case there is no such polynomial the result is undefined.

The implementation of *operator/* depends on the number type plugged into the template. To provide the division we implement two division operations *pseudo_div* and *euclidean_div*. The first operation works with ring number types providing a *gcd* operation, so called *unique factorization domains*. The second operation works with polynomials over *field* number types. To separate our number types we introduce a traits class providing tags to choose one or the other code variant. In the header file of *RPolynomial* there are three predefined class types *ring_or_field_dont_know*, *ring_with_gcd*, and *field_with_div*. As a prerequisite a user has just to specialize the class template *ring_or_field*<> to the number type that she wants to plug into *RPolynomial*<>. For the LEDA integer type this can be done as follows

```
template <>
struct ring_or_field<leda_integer> {
  typedef ring_with_gcd kind;
  static leda_integer gcd(const leda_integer& a, const leda_integer& b)
  { return ::gcd(a,b); }
};
```

In case of a Euclidean ring the class *ring_or_field*<*RT*> has to provide the gcd operation of two *RT* operands as a static method [2]. Based on this number type flag *RPolynomial*<*leda_integer*> provides the division operation with the help of *pseudo_div* based on the *gcd* operation of *leda_integer*. For users not providing the *ring_or_field*<> specialization an error message is raised.

⟨*polynomial implementation*⟩+≡

```
template <class NT>
RPolynomial<NT> divop (const RPolynomial<NT>& p1,
                       const RPolynomial<NT>& p2,
                       ring_or_field_dont_know)
{
  CGAL_assertion_msg(0,"\n\
  The division operation on polynomials requires that you\n\
  specify if your number type provides a binary gcd() operation\n\
```

[2]This makes life easier when working with compilers that lack Koenig-lookup.

```
  or is a field type including an operator/().\n\
  You do this by creating a specialized class:\n\
  template <> class ring_or_field<yourNT> with a member type:\n\
  typedef ring_with_gcd kind; OR\n\
  typedef field_with_div kind;\n");
  return RPolynomial<NT>(); // never reached
}
```

The division operator is implemented depending on the number type *NT*. Our number type traits *ring_or_field*<*NT*> provides a tag type to specialize it via three overloaded methods *divop*( ) that are implemented below.

⟨*polynomial implementation*⟩+≡
```
template <class NT> inline
RPolynomial<NT> operator / (const RPolynomial<NT>& p1,
                            const RPolynomial<NT>& p2)
{ return divop(p1,p2,ring_or_field<NT>::kind()); }
```

Field Number Types

We first implement standard polynomial division. Starting from polynomials f and g we determine two polynomials q and r such that $f = q * g + r$ where $d_r \leq d_g$.

⟨*polynomial statics*⟩≡
```
template <class NT>
void RPolynomial<NT>::euclidean_div(
  const RPolynomial<NT>& f, const RPolynomial<NT>& g,
  RPolynomial<NT>& q, RPolynomial<NT>& r)
{
  r = f; r.copy_on_write();
  int rd=r.degree(), gd=g.degree(), qd(0);
  if ( rd < gd ) { q = RPolynomial<NT>(NT(0)); }
  else { qd = rd-gd+1; q = RPolynomial<NT>(size_t(qd)); }
  while ( rd >= gd ) {
    NT S = r[rd] / g[gd];
    qd = rd-gd;
    q.coeff(qd) += S;
    r.minus_offsetmult(g,S,qd);
    rd = r.degree();
  }
  CGAL_postcondition( f==q*g+r );
}
```

We need an operation which allows us to subtract a polynomial s which is the product of a polynomial $p = \sum_{i=0}^{d} a_i x^i$ and a monomial $m = bx^k$. The result is $mp = \sum_{i=0}^{d+k} b\tilde{a}_i x^i$ where $\tilde{a}_i = 0$ for $0 \leq i < k$ and $\tilde{a}_{i+k} = a_i$ for $0 \leq i \leq d$. We implement this by shifting the coefficients of p by k places while multiplying them by b and leave the lower k entries of the resulting polynomial zero.

⟨*offset multiplication*⟩≡
```
void minus_offsetmult(const RPolynomial<NT>& p, const NT& b, int k)
{ CGAL_assertion(!ptr->is_shared());
  RPolynomial<NT> s(size_type(p.degree()+k+1)); // zero entries
  for (int i=k; i <= s.degree(); ++i) s.coeff(i) = b*p[i-k];
  operator-=(s);
}
```

Now we can just specialize *divop* in its third argument:

⟨*polynomial implementation*⟩+≡
```
template <class NT>
RPolynomial<NT> divop (const RPolynomial<NT>& p1,
                       const RPolynomial<NT>& p2,
                       field_with_div)
{ CGAL_assertion(!p2.is_zero());
  if (p1.is_zero()) return 0;
  RPolynomial<NT> q,r;
  RPolynomial<NT>::euclidean_div(p1,p2,q,r);
  CGAL_postcondition( (p2*q+r==p1) );
  return q;
}
```

Unique factorization domains

Polynomial division avoiding using the notion of an inverse that is present in fields, is not so trivial. We first introduce the algebraic notions necessary for some theoretic results. We start with the introduction of divisibility and greatest common divisors. Let $f = \sum_{i=0}^{d_f} a_i x^i$, $g = \sum_{i=0}^{d_g} b_i x^i$ be polynomials of *degree* d_f and d_g. We assume the *leading coefficient* a_{d_f} and b_{d_g} to be nonzero and thereby degree defining.

Definition 9: A *commutative ring* $\mathcal{R}$ with unit 1 and containing no zero divisors is called an *integrity domain*. An integrity domain where every nonzero element is either a *unit* or has a unique representation[3] as a product of primes is called a *unique factorization domain*.

The integers $\mathbb{Z}$ are our default example of a unique factorization domain. One example for a ring that is no unique factorization domain is $\mathbb{Z}/4\mathbb{Z}$ which contains the zero divisor 2.

Definition 10: Let $\mathcal{R}$ be an integrity domain. $\mathcal{K} = Quot(\mathcal{R})$, $\mathcal{K}^* = \mathcal{K} - \{0\}$. Let $a, b \in \mathcal{K}^*$.

$$\begin{array}{ll} (1) & a\,|_{\mathcal{R}}\, b \ \text{"}a \text{ divides } b \text{ in } \mathcal{R}\text{"} \quad :\Leftrightarrow\ \exists c \in R : b = ac \\ (2) & a \sim_{\mathcal{R}} b \ :\Leftrightarrow\ a\,|_{\mathcal{R}}\, b \wedge b\,|_{\mathcal{R}}\, a \end{array}$$

Let $d \in \mathcal{K}^*$. d is called $\gcd_{\mathcal{K}}(a,b)$ $:\Leftrightarrow$

$$\begin{array}{ll} (1) & d\,|_{\mathcal{R}}\, a \wedge d\,|_{\mathcal{R}}\, b \\ (2) & \forall c \in \mathcal{K}^* : c\,|_{\mathcal{R}}\, a \wedge c\,|_{\mathcal{R}}\, b \Longrightarrow c\,|_{\mathcal{R}}\, d \end{array}$$

d is determined uniquely except for multiplication with units.

[3] Uniqueness up to permutations and multiplication with units.

The following lemma assures that we can divide f by g in the integral domain when we accept an expansion of f by a power of the leading coefficient of g.

Lemma 6.2.1: Let $\mathcal{R}$ be a ring and $f,g \in \mathcal{R}[x], g \neq 0$. Let b be the leading coefficient of g. Then there are polynomials $q,r \in \mathcal{R}[x]$, such that

$$b^s f = qg + r$$

where either $r = 0$ or $r \neq 0$ and $d_r < d_g$ and the integer $s = 0$ if $f = 0$ and $s = \max\{0, d_f - d_g + 1\}$ if $f \neq 0$. If b is no zero divisor in $\mathcal{R}$ then q and r are uniquely defined.

Proof. **Existence** — In case $f = 0$ and $f \neq 0, d_f < d_g$ the pair $q = 0, r = f$ is a trivial solution. Let $f \neq 0$, $\Delta(f,g) := d_f - d_g \geq 0$. We show the existence of q,r by induction on $\Delta(f,g)$. Let q,r exist for polynomials $f,g, f \neq 0, \Delta(f,g) < \Delta_0, \Delta_0 \geq 0$. Now let $f \in \mathcal{R}[x], f \neq 0, \Delta(f,g) = \Delta_0$ and a be the leading coefficient of f. We look at $f' := bf - ax^{\Delta_0}g$. Either $f' = 0$ or $f' \neq 0$ but $d_{f'} < d_f, \Delta(f',g) < \Delta_0$. By the induction hypothesis there are $r', q' \in \mathcal{R}[x]$ such that $b^{s'} f = q'g + r'$ where either $r' = 0$ or $r \neq 0$ and $d_{r'} < d_g$ and the non-negative integer $s' \leq d_f - d_g$. Thus $b^{s'+1} f = (b^{s'} a x^{\Delta_0} + q')g + r'$. Because of $s' + 1 \leq d_f - d_g + 1$ the existence of q and r is clear.

Uniqueness — Let b be no zero divisor in $\mathcal{R}$ and assume that there is another representation $b^s f = \tilde{q}g + \tilde{r}$ where either $\tilde{r} = 0$ or $\tilde{r} \neq 0$ and $d_q < d_{\tilde{q}}$. By subtracting one from the other we obtain $(q - \tilde{q})g = \tilde{r} - r$. As the b is no zero divisor in $\mathcal{R}$ the degree formula tells us that $\deg(q - \tilde{q})g = \deg(q - \tilde{q}) + d_g \geq d_g$. Thus $\tilde{r} - r \neq 0$ and $\deg(\tilde{r} - r) \geq d_g$. On the other hand $\tilde{r} - r$ is the difference of two polynomials of degree smaller than d_g and therefore $\deg(\tilde{r} - r) < d_g$. From the contradiction we obtain $q - \tilde{q} = \tilde{r} - r = 0$. □

See any algebra book like [RSV84] for more details. Knuth [Knu98] calls the algorithm based on the above lemma *pseudo-division*. According to this lemma one can determine q and r within the ring without resorting to the quotient field. We follow the construction in the proof in reverse order to reduce f down to r. We use *minus_offsetmult* for the reduction from f to f'.

⟨*polynomial statics*⟩+≡
```
template <class NT>
void RPolynomial<NT>::pseudo_div(
  const RPolynomial<NT>& f, const RPolynomial<NT>& g,
  RPolynomial<NT>& q, RPolynomial<NT>& r, NT& D)
{
  int fd=f.degree(), gd=g.degree();
  if ( fd<gd )
  { q = RPolynomial<NT>(0); r = f; D = 1;
    CGAL_postcondition(RPolynomial<NT>(D)*f==q*g+r); return;
  }
  // now we know fd >= gd and f>=g
  int qd=fd-gd, delta=qd+1, rd=fd;
  q = RPolynomial<NT>( size_t(delta) );
  NT G = g[gd]; // highest order coeff of g
  D = G; while (--delta) D*=G; // D = G^delta
  RPolynomial<NT> res = RPolynomial<NT>(D)*f;
  while (qd >= 0) {
    NT F = res[rd]; // highest order coeff of res
    NT t = F/G;     // ensured to be integer by multiplication of D
    q.coeff(qd) = t;    // store q coeff
```

```
    res.minus_offsetmult(g,t,qd);
    if (res.is_zero()) break;
    rd = res.degree();
    qd = rd - gd;
  }
  r = res;
  CGAL_postcondition(RPolynomial<NT>(D)*f==q*g+r);
}
```

Finally we specialize *divop* for unique factorization domains.

⟨*polynomial implementation*⟩+≡

```
template <class NT>
RPolynomial<NT> divop (const RPolynomial<NT>& p1, const RPolynomial<NT>& p2,
                       ring_with_gcd)
{ CGAL_assertion(!p2.is_zero());
  if (p1.is_zero()) return 0;
  RPolynomial<NT> q,r; NT D;
  RPolynomial<NT>::pseudo_div(p1,p2,q,r,D);
  CGAL_postcondition( (p2*q+r==p1*RPolynomial<NT>(D)) );
  return q/=D;
}
```

For the reduction of polynomials we finally implement the greatest common divisor method as introduced by Euclid. For two elements $a,b \in \mathcal{R}$ Euclids algorithm uses the reduction $\gcd(a,b) \sim_{\mathcal{R}} \gcd(b, a \mod b)$.

Definition 11: For a polynomial $f = \sum_{i=0}^{d} a_i x^i$ we define its *content* as $\mathrm{cont}\,(f) = \gcd(a_0,\dots,a_d)$ and its *primitive part* as $\mathrm{pp}(f) = f/\mathrm{cont}\,(f)$.

Again the content of a polynomial is only unique up to multiplication by units of $\mathcal{R}$. Note that $\mathrm{cont}\,(f)$ is a divisor of all coefficients of f in $\mathcal{R}$ and therefore the division is reducing the representation of f such that $\mathrm{cont}\,(\mathrm{pp}(f)) = 1$. The following lemma tells us something about the composition of the gcd of two polynomials from the contents and the primitive parts. A polynomial whose content is 1 is called *primitive*.

Lemma 6.2.2 (Gauss): The product of two *primitive* polynomials f and g over a unique factorization domain is again *primitive*. Moreover let f and g be two general polynomials over a unique factorization domain $\mathcal{R}$. Then $\mathrm{cont}\,(\gcd(f,g)) \sim_{\mathcal{R}} \gcd(\mathrm{cont}\,(f),\mathrm{cont}\,(g))$ and $\mathrm{pp}(\gcd(f,g)) \sim_{\mathcal{R}} \gcd(\mathrm{pp}(f),\mathrm{pp}(g))$.

Proof. Let $f = \sum_{i=0}^{d_f} a_i x^i, g = \sum_{i=0}^{d_g} b_i x^i$ be primitive polynomials. We show for any prime p of the domain that it does not divide all the coefficients of $f \cdot g$. For both polynomials we chose the smallest indices j and k for which p does divide $(a_i)_i$ and $(b_i)_i$. We then examine the coefficient of x^{j+k} of $f \cdot g$:

$$a_j b_k + a_{j+1} b_{k-1} + \cdots + a_{j+k} b_0 + a_{j-1} b_{k+1} + \cdots + a_0 b_{k+j}$$

As p divides only the first term and none of the following terms, p does not divide the sum.

From the above we can deduce for general polynomials f and g that $\mathrm{pp}(fg) \sim_{\mathcal{R}} \mathrm{pp}(f)\mathrm{pp}(g)$. The product fg can be decomposed as $fg = \mathrm{cont}\,(f)\mathrm{pp}(f)\mathrm{cont}\,(g)\mathrm{pp}(g) = \mathrm{cont}\,(f)\mathrm{cont}\,(g)\mathrm{pp}(g)\mathrm{pp}(f)$ and $fg = \mathrm{cont}\,(fg)\mathrm{pp}(fg)$. Thereby we can deduce that $\mathrm{cont}\,(fg) \sim_{\mathcal{R}} \mathrm{cont}\,(f)\mathrm{cont}\,(g)$.

Now assume that $h \sim_{\mathcal{R}} \gcd(f,g)$ and thus $f = hF$ and $g = hG$ for some polynomials F and G from $\mathcal{R}[x]$. By the previous result we get $\mathrm{cont}\,(f) \sim_{\mathcal{R}} \mathrm{cont}\,(h)\mathrm{cont}\,(F)$ and $\mathrm{cont}\,(g) \sim_{\mathcal{R}} \mathrm{cont}\,(h)\mathrm{cont}\,(G)$ and thereby $\mathrm{cont}\,(\gcd(f,g)) \sim_{\mathcal{R}} \mathrm{cont}\,(h) \sim_{\mathcal{R}} \gcd_{\mathcal{R}}(\mathrm{cont}\,(f), \mathrm{cont}\,(g))$. The latter equality follows from the fact that $\gcd_{\mathcal{R}}(\mathrm{cont}\,(F), \mathrm{cont}\,(G)) \sim_{\mathcal{R}} 1$ due to the properties of h in the decomposition of f and g. A similar argument shows that $\mathrm{pp}(\gcd(f,g)) \sim_{\mathcal{R}} \gcd(\mathrm{pp}(f), \mathrm{pp}(g))$. □

This result simplifies the problem and allows us to keep the size of the coefficients of the polynomials as small as possible. An elaborate treatment of the topic can be found in [Coh93, Knu98].

By the above lemma we obtain the following strategy. First calculate $F = \gcd_{\mathcal{R}}(\mathrm{cont}\,(f), \mathrm{cont}\,(g))$ by the gcd routine on the ring number type *NT*. Reduce both polynomials by their content to their primitive parts $f^\circ = \mathrm{pp}(f)$ and $g^\circ = \mathrm{pp}(g)$.

Then reduce $\gcd(f^\circ, g^\circ) \sim_{\mathcal{R}} \gcd(g^\circ, f^\circ \mod g^\circ)$. However our pseudo-division *pseudo_div* only allows reductions of the form (Df°, g°) to $(g^\circ, Df^\circ \mod g^\circ)$ where $D = b^s$ as described above. This is ok though as $\gcd(Df^\circ, g^\circ) \sim_{\mathcal{R}} \gcd_{\mathcal{R}}(\mathrm{cont}\,(Df^\circ), \mathrm{cont}\,(g^\circ))\gcd(\mathrm{pp}(Df^\circ), \mathrm{pp}(g^\circ)) \sim_{\mathcal{R}} \gcd_{\mathcal{R}}(D,1)\gcd(f^\circ, g^\circ) \sim_{\mathcal{R}} \gcd(f^\circ, g^\circ)$. The final result of the Euclidean reduction delivers $d^\circ = \gcd(f^\circ, g^\circ)$. We obtain the desired result $\gcd(f,g) = F \cdot d^\circ$.

⟨*polynomial statics*⟩+≡
```
template <class NT>
RPolynomial<NT> RPolynomial<NT>::gcd(
  const RPolynomial<NT>& p1, const RPolynomial<NT>& p2)
{
  if ( p1.is_zero() )
    if ( p2.is_zero() ) return RPolynomial<NT>(NT(1));
    else return p2.abs();
  if ( p2.is_zero() )
    return p1.abs();
  RPolynomial<NT> f1 = p1.abs();
  RPolynomial<NT> f2 = p2.abs();
  NT f1c = f1.content(), f2c = f2.content();
  f1 /= f1c; f2 /= f2c;
  NT F = ring_or_field<NT>::gcd(f1c,f2c);
  RPolynomial<NT> q,r; NT M=1,D;
  bool first = true;
  while ( ! f2.is_zero() ) {
    RPolynomial<NT>::pseudo_div(f1,f2,q,r,D);
    if (!first) M*=D;
    r /= r.content();
    f1=f2; f2=r;
    first=false;
  }
  return RPolynomial<NT>(F)*f1.abs();
}
```

The rest of this section just implements all kinds of comparison predicates based on our sign evaluation.

⟨*polynomial implementation*⟩+≡
```
template <class NT>
inline RPolynomial<NT>
gcd(const RPolynomial<NT>& p1, const RPolynomial<NT>& p2)
{ return RPolynomial<NT>::gcd(p1,p2); }

template <class NT>  bool operator ==
  (const RPolynomial<NT>& p1, const RPolynomial<NT>& p2)
  { return ( (p1-p2).sign() == CGAL::ZERO ); }

template <class NT>  bool operator !=
  (const RPolynomial<NT>& p1, const RPolynomial<NT>& p2)
  { return ( (p1-p2).sign() != CGAL::ZERO ); }

template <class NT>  bool operator <
  (const RPolynomial<NT>& p1, const RPolynomial<NT>& p2)
  { return ( (p1-p2).sign() == CGAL::NEGATIVE ); }

template <class NT>  bool operator <=
  (const RPolynomial<NT>& p1, const RPolynomial<NT>& p2)
  { return ( (p1-p2).sign() != CGAL::POSITIVE ); }

template <class NT>  bool operator >
  (const RPolynomial<NT>& p1, const RPolynomial<NT>& p2)
  { return ( (p1-p2).sign() == CGAL::POSITIVE ); }

template <class NT>  bool operator >=
  (const RPolynomial<NT>& p1, const RPolynomial<NT>& p2)
  { return ( (p1-p2).sign() != CGAL::NEGATIVE ); }

#ifndef _MSC_VER
template <class NT>  CGAL::Sign
  sign(const RPolynomial<NT>& p)
  { return p.sign(); }
#endif // collides with global CGAL sign

#ifndef _MSC_VER

  // lefthand side
  template <class NT>    RPolynomial<NT> operator +
  (const NT& num, const RPolynomial<NT>& p2)
  { return (RPolynomial<NT>(num) + p2); }
  template <class NT>    RPolynomial<NT> operator -
  (const NT& num, const RPolynomial<NT>& p2)
  { return (RPolynomial<NT>(num) - p2); }
  template <class NT>    RPolynomial<NT> operator *
  (const NT& num, const RPolynomial<NT>& p2)
  { return (RPolynomial<NT>(num) * p2); }
  template <class NT>    RPolynomial<NT> operator /
  (const NT& num, const RPolynomial<NT>& p2)
  { return (RPolynomial<NT>(num)/p2); }

  // righthand side
  template <class NT>    RPolynomial<NT> operator +
  (const RPolynomial<NT>& p1, const NT& num)
  { return (p1 + RPolynomial<NT>(num)); }
  template <class NT>    RPolynomial<NT> operator -
  (const RPolynomial<NT>& p1, const NT& num)
  { return (p1 - RPolynomial<NT>(num)); }
  template <class NT>    RPolynomial<NT> operator *
```

```
  (const RPolynomial<NT>& p1, const NT& num)
  { return (p1 * RPolynomial<NT>(num)); }
  template <class NT>    RPolynomial<NT> operator /
  (const RPolynomial<NT>& p1, const NT& num)
  { return (p1 / RPolynomial<NT>(num)); }

  // lefthand side
  template <class NT>    bool operator ==
  (const NT& num, const RPolynomial<NT>& p)
  { return ( (RPolynomial<NT>(num)-p).sign() == CGAL::ZERO );}
  template <class NT>    bool operator !=
  (const NT& num, const RPolynomial<NT>& p)
  { return ( (RPolynomial<NT>(num)-p).sign() != CGAL::ZERO );}
  template <class NT>    bool operator <
  (const NT& num, const RPolynomial<NT>& p)
  { return ( (RPolynomial<NT>(num)-p).sign() == CGAL::NEGATIVE );}
  template <class NT>    bool operator <=
  (const NT& num, const RPolynomial<NT>& p)
  { return ( (RPolynomial<NT>(num)-p).sign() != CGAL::POSITIVE );}
  template <class NT>    bool operator >
  (const NT& num, const RPolynomial<NT>& p)
  { return ( (RPolynomial<NT>(num)-p).sign() == CGAL::POSITIVE );}
  template <class NT>    bool operator >=
  (const NT& num, const RPolynomial<NT>& p)
  { return ( (RPolynomial<NT>(num)-p).sign() != CGAL::NEGATIVE );}
  // righthand side
  template <class NT>    bool operator ==
  (const RPolynomial<NT>& p, const NT& num)
  { return ( (p-RPolynomial<NT>(num)).sign() == CGAL::ZERO );}
  template <class NT>    bool operator !=
  (const RPolynomial<NT>& p, const NT& num)
  { return ( (p-RPolynomial<NT>(num)).sign() != CGAL::ZERO );}
  template <class NT>    bool operator <
  (const RPolynomial<NT>& p, const NT& num)
  { return ( (p-RPolynomial<NT>(num)).sign() == CGAL::NEGATIVE );}
  template <class NT>    bool operator <=
  (const RPolynomial<NT>& p, const NT& num)
  { return ( (p-RPolynomial<NT>(num)).sign() != CGAL::POSITIVE );}
  template <class NT>    bool operator >
  (const RPolynomial<NT>& p, const NT& num)
  { return ( (p-RPolynomial<NT>(num)).sign() == CGAL::POSITIVE );}
  template <class NT>    bool operator >=
  (const RPolynomial<NT>& p, const NT& num)
  { return ( (p-RPolynomial<NT>(num)).sign() != CGAL::NEGATIVE );}

#endif // _MSC_VER CGAL_CFG_MATCHING_BUG_2
```

Finally we offer standard I/O operations.

⟨*polynomial implementation*⟩+≡

```
template <class NT>
void print_monomial(std::ostream& os, const NT& n, int i)
{
```

```
  if (i==0) os << n;
  if (i==1) os << n << "R";
  if (i>1)  os << n << "R^" << i;
}
#define RPOLYNOMIAL_EXPLICIT_OUTPUT
// I/O
template <class NT>
std::ostream& operator << (std::ostream& os, const RPolynomial<NT>& p)
{
  int i;
  switch( os.iword(CGAL::IO::mode) )
  {
    case CGAL::IO::ASCII :
      os << p.degree() << ' ';
      for(i=0; i<=p.degree(); ++i)
        os << p[i] << ' ';
      return os;
    case CGAL::IO::BINARY :
      CGAL::write(os, p.degree());
      for(i=0; i<=p.degree(); ++i)
        CGAL::write(os, p[i]);
      return os;
    default:
#ifndef RPOLYNOMIAL_EXPLICIT_OUTPUT
      os << "RPolynomial(" << p.degree() << ", ";
      for(i=0; i<=p.degree(); ++i) {
        os << p[i];
        if (i < p.degree()) os << ", ";
      }
      os << ")";
#else
      print_monomial(os,p[p.degree()],p.degree());
      for(i=p.degree()-1; i>=0; --i) {
        if (p[i]!=NT(0)) { os << " + "; print_monomial(os,p[i],i); }
      }
#endif
      return os;
  }
}
template <class NT>
std::istream& operator >> (std::istream& is, RPolynomial<NT>& p)
{
  int i,d;
  NT c;
  switch( is.iword(CGAL::IO::mode) )
  {
    case CGAL::IO::ASCII :
      is >> d;
      if (d < 0) p = RPolynomial<NT>();
      else {
        typename RPolynomial<NT>::Vector coeffs(d+1);
        for(i=0; i<=d; ++i) is >> coeffs[i];
```

```
      p = RPolynomial<NT>(coeffs.begin(),coeffs.end());
    }
    break;
  case CGAL::IO::BINARY :
    CGAL::read(is, d);
    if (d < 0) p = RPolynomial<NT>();
    else {
      typename RPolynomial<NT>::Vector coeffs(d+1);
      for(i=0; i<=d; ++i)
      { CGAL::read(is,c); coeffs[i]=c; }
      p = RPolynomial<NT>(coeffs.begin(),coeffs.end());
    }
    break;
  default:
    CGAL_assertion_msg(0,"\nStream must be in ascii or binary mode\n");
    break;
  }
  return is;
}
```

We finally give the file wrapper for our definition and implementation part.

⟨*RPolynomial.h*⟩≡

```
⟨CGAL Header⟩
#ifndef CGAL_RPOLYNOMIAL_H
#define CGAL_RPOLYNOMIAL_H

#include <CGAL/basic.h>
#include <CGAL/kernel_assertions.h>
#include <CGAL/Handle_for.h>
#include <CGAL/number_type_basic.h>
#include <CGAL/number_utils.h>
#include <CGAL/IO/io.h>
#undef _DEBUG
#define _DEBUG 3
#include <CGAL/Nef_2/debug.h>

#if defined(_MSC_VER) || defined(__BORLANDC__)
#include <CGAL/Nef_2/vector_MSC.h>
#define CGAL_SIMPLE_NEF_INTERFACE
#define SNIHACK ,char,char
#define SNIINST ,'c','c'
#else
#include <vector>
#define SNIHACK
#define SNIINST
#endif

class ring_or_field_dont_know {};
class ring_with_gcd {};
class field_with_div {};

template <typename NT>
struct ring_or_field {
  typedef ring_or_field_dont_know kind;
```

```
};
template <>
struct ring_or_field<int> {
  typedef ring_with_gcd kind;
  typedef int RT;
  static RT gcd(const RT& a, const RT& b)
  { if (a == 0)
      if (b == 0)  return 1;
      else         return CGAL_NTS abs(b);
    if (b == 0)    return CGAL_NTS abs(a);
    // here both a and b are nonzero
    int u = CGAL_NTS abs(a);
    int v = CGAL_NTS abs(b);
    if (u < v) v = v%u;
    while (v != 0)
    { int tmp = u % v;
      u = v;
      v = tmp;
    }
    return u;
  }
};
template <>
struct ring_or_field<long> {
  typedef ring_with_gcd kind;
  typedef long RT;
  static RT gcd(const RT& a, const RT& b)
  { if (a == 0)
      if (b == 0)  return 1;
      else         return CGAL_NTS abs(b);
    if (b == 0)    return CGAL_NTS abs(a);
    // here both a and b are nonzero
    int u = CGAL_NTS abs(a);
    int v = CGAL_NTS abs(b);
    if (u < v) v = v%u;
    while (v != 0)
    { int tmp = u % v;
      u = v;
      v = tmp;
    }
    return u;
  }
};

template <>
struct ring_or_field<double> {
  typedef field_with_div kind;
  typedef double RT;
  static RT gcd(const RT&, const RT&)
  { return 1.0; }
};

CGAL_BEGIN_NAMESPACE
```

```
template <class NT> class RPolynomial_rep;

template <class NT> class RPolynomial;
```

⟨*gcd of range*⟩
⟨*common prototype statements*⟩
⟨*general rep template*⟩
⟨*the polynomial class template*⟩

```
template <class NT> NT RPolynomial<NT>::R_;
int    RPolynomial<int>::R_;
double RPolynomial<double>::R_;
```

⟨*special operations required by CGAL*⟩
⟨*polynomial implementation*⟩

⟨*polynomial statics*⟩

```
CGAL_END_NAMESPACE

#endif  // CGAL_RPOLYNOMIAL_H
```

Finally we provide a gcd calculation routine for a sequence of numbers. This routine requires the existence of a *gcd* operation as provided by our number type traits *ring_or_field*<*NT*>.

⟨*gcd of range*⟩≡

```
template <class Forward_iterator>
typename std::iterator_traits<Forward_iterator>::value_type
gcd_of_range(Forward_iterator its, Forward_iterator ite)
{ CGAL_assertion(its!=ite);
  typedef typename std::iterator_traits<Forward_iterator>::value_type NT;
  NT res = *its++;
  for(; its!=ite; ++its) res =
    (*its==0 ? res : ring_or_field<NT>::gcd(res, *its));
  if (res==0) res = 1;
  return res;
}
```

Now for some CGAL specific number type reqirements.

⟨*special operations required by CGAL*⟩≡

```
template <class NT>  double to_double
  (const RPolynomial<NT>& p)
  { return (CGAL::to_double(p.eval_at(RPolynomial<NT>::R_))); }

template <class NT>   bool is_valid
  (const RPolynomial<NT>& p)
  { return (CGAL::is_valid(p[0])); }

template <class NT>  bool is_finite
  (const RPolynomial<NT>& p)
  { return CGAL::is_finite(p[0]); }

template <class NT>  CGAL::io_Operator
  io_tag(const RPolynomial<NT>&)
  { return CGAL::io_Operator(); }
```

Now the prototypes which have to be declared due to a friend statement in the rep class.

⟨*common prototype statements*⟩≡

```
template <class NT>   RPolynomial<NT>
  operator - (const RPolynomial<NT>&);
template <class NT>    RPolynomial<NT>
  operator + (const RPolynomial<NT>&, const RPolynomial<NT>&);
template <class NT>    RPolynomial<NT>
  operator - (const RPolynomial<NT>&, const RPolynomial<NT>&);
template <class NT>    RPolynomial<NT>
  operator * (const RPolynomial<NT>&, const RPolynomial<NT>&);
template <class NT> inline RPolynomial<NT>
  operator / (const RPolynomial<NT>&, const RPolynomial<NT>&);

#ifndef _MSC_VER
template<class NT>  CGAL::Sign
  sign(const RPolynomial<NT>& p);
#endif // collides with global CGAL sign

template <class NT>  double
  to_double(const RPolynomial<NT>& p) ;
template <class NT>   bool
  is_valid(const RPolynomial<NT>& p) ;
template <class NT>  bool
  is_finite(const RPolynomial<NT>& p) ;

template<class NT>
  std::ostream& operator << (std::ostream& os, const RPolynomial<NT>& p);
template <class NT>
  std::istream& operator >> (std::istream& is, RPolynomial<NT>& p);

#ifndef _MSC_VER
  // lefthand side
template<class NT> inline RPolynomial<NT> operator +
  (const NT& num, const RPolynomial<NT>& p2);
template<class NT> inline RPolynomial<NT> operator -
  (const NT& num, const RPolynomial<NT>& p2);
template<class NT> inline RPolynomial<NT> operator *
  (const NT& num, const RPolynomial<NT>& p2);
template<class NT> inline RPolynomial<NT> operator /
  (const NT& num, const RPolynomial<NT>& p2);

  // righthand side
template<class NT> inline RPolynomial<NT> operator +
  (const RPolynomial<NT>& p1, const NT& num);
template<class NT> inline RPolynomial<NT> operator -
  (const RPolynomial<NT>& p1, const NT& num);
template<class NT> inline RPolynomial<NT> operator *
  (const RPolynomial<NT>& p1, const NT& num);
template<class NT> inline RPolynomial<NT> operator /
  (const RPolynomial<NT>& p1, const NT& num);

  // lefthand side
template<class NT> inline bool operator ==
  (const NT& num, const RPolynomial<NT>& p);
template<class NT> inline bool operator !=
  (const NT& num, const RPolynomial<NT>& p);
```

```
template<class NT> inline bool operator <
  (const NT& num, const RPolynomial<NT>& p);
template<class NT> inline bool operator <=
  (const NT& num, const RPolynomial<NT>& p);
template<class NT> inline bool operator >
  (const NT& num, const RPolynomial<NT>& p);
template<class NT> inline bool operator >=
  (const NT& num, const RPolynomial<NT>& p);
  // righthand side
template<class NT> inline bool operator ==
  (const RPolynomial<NT>& p, const NT& num);
template<class NT> inline bool operator !=
  (const RPolynomial<NT>& p, const NT& num);
template<class NT> inline bool operator <
  (const RPolynomial<NT>& p, const NT& num);
template<class NT> inline bool operator <=
  (const RPolynomial<NT>& p, const NT& num);
template<class NT> inline bool operator >
  (const RPolynomial<NT>& p, const NT& num);
template<class NT> inline bool operator >=
  (const RPolynomial<NT>& p, const NT& num);
#endif // _MSC_VER
```

6.3 A Test of RPolynomial

We test in different settings concerning instantiation number types and operator argument types. We basically use *leda_integer*, *int*, and *double*.

⟨*RPolynomial-test.C*⟩≡

```
#include <CGAL/basic.h>
#ifndef _MSC_VER
#include <CGAL/RPolynomial.h>
#else
#include <CGAL/RPolynomial_MSC.h>
#endif
#include <CGAL/test_macros.h>
#ifdef CGAL_USE_LEDA
#include <CGAL/leda_integer.h>
typedef leda_integer Integer;
template <>
struct ring_or_field<leda_integer> {
  typedef ring_with_gcd kind;
  typedef leda_integer RT;
  static RT gcd(const RT& r1, const RT& r2)
  { return ::gcd(r1,r2); }
};
#else
#ifdef CGAL_USE_GMP
```

```
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz Integer;
template <>
struct ring_or_field<CGAL::Gmpz> {
  typedef ring_with_gcd kind;
  typedef CGAL::Gmpz RT;
  static RT gcd(const RT& r1, const RT& r2)
  { return CGAL::gcd(r1,r2); }
};
#else
typedef int Integer;
#endif
#endif
using namespace CGAL;
#ifdef _MSC_VER
#define MSCCAST(n) static_cast<RP>(n)
#define RPolynomial RPolynomial_MSC
CGAL_DEFINE_ITERATOR_TRAITS_POINTER_SPEC(Integer)
typedef std::iterator_traits<int*>::iterator_category iiii;
#else
#define MSCCAST(n) n
#endif
#define PRT(t1,t2) std::cout<<"testing instances "<<#t1<<" "<<#t2<<std::endl
int main()
{
  //SETDTHREAD(3); CGAL::set_pretty_mode ( std::cerr );
  CGAL_TEST_START;
  { PRT(Integer,Integer);
    typedef Integer NT; typedef RPolynomial<Integer> RP;
    ⟨test sequence⟩
  }
  { PRT(int,Integer);
    typedef int NT; typedef RPolynomial<Integer> RP;
    ⟨test sequence⟩
  }
  { PRT(double,Integer);
    typedef double NT; typedef RPolynomial<Integer> RP;
    ⟨test sequence⟩
  }
  { PRT(int,int);
    typedef int NT; typedef RPolynomial<int> RP;
    ⟨test sequence⟩
  }
  { PRT(double,int);
    typedef double NT; typedef RPolynomial<int> RP;
    ⟨test sequence⟩
  }
  CGAL_TEST_END;
}
```

⟨*test sequence*⟩≡

```
RP::NT seq[4] = { 0, 1, 2, 0 };
RP p1, p2(NT(1)), p3(NT(1),NT(1)), p4(5,2), p5(-2,5), p6(4,1),
   p7(3,0), p8(seq,seq+4);
RP p10(-1,0,1), p11(-1,1), p12(1,1);
NT r1(2), r2(-2);

CGAL_TEST(p1.degree()==-1);
CGAL_TEST(p2.degree()==0);
CGAL_TEST(p4.degree()==1);
CGAL_TEST(p7.degree()==0);
CGAL_TEST(p8.degree()==2);
CGAL_TEST((-(-p4)) == p4);
CGAL_TEST((-(-p7)) == p7);
CGAL_TEST((p4+p5) == RP(3,7));
CGAL_TEST((p4-p5) == RP(7,-3));
RP::NT prod[3] = { -10, 21, 10 };
CGAL_TEST((p4*p5) == RP(prod,prod+3));
CGAL_TEST((p2*p3) == p3);
MSCCAST(r1)+p3;
p3+MSCCAST(r1);
CGAL_TEST((MSCCAST(r1)+p3) == RP(3,1));
CGAL_TEST((MSCCAST(r1)-p3) == RP(1,-1));
CGAL_TEST((MSCCAST(r1)*p3) == RP(2,2));
CGAL_TEST((p3+MSCCAST(r1)) == RP(3,1));
CGAL_TEST((p3-MSCCAST(r1)) == RP(-1,1));
CGAL_TEST((p3*MSCCAST(r1)) == RP(2,2));
CGAL_TEST(p2 != p3);
CGAL_TEST(p2 < p3);
CGAL_TEST(p2 <= p3);
CGAL_TEST(p5 > p4);
CGAL_TEST(p5 >= p4);

CGAL_TEST(MSCCAST(r1) != p2);
CGAL_TEST(MSCCAST(r2) < p2);
CGAL_TEST(MSCCAST(r2) <= p2);
CGAL_TEST(MSCCAST(r1) > p2);
CGAL_TEST(MSCCAST(r1) >= p2);
CGAL_TEST(p2 != MSCCAST(r1));
CGAL_TEST(p2 > MSCCAST(r2));
CGAL_TEST(p2 >= MSCCAST(r2));
CGAL_TEST(p2 < MSCCAST(r1));
CGAL_TEST(p2 <= MSCCAST(r1));

CGAL_TEST(CGAL_NTS sign(p5)==+1);
CGAL_TEST(CGAL_NTS sign(-p5)==-1);
CGAL_TEST(CGAL_NTS sign(p2)==+1);
CGAL_TEST(CGAL_NTS sign(-p2)==-1);
p3 += p2;
p3 -= p2;
p3 *= p5;
p3 += MSCCAST(r1);
p3 -= MSCCAST(r1);
p3 *= MSCCAST(r2);

RP::NT D;
```

```
  RP q1(17),q2(5),q3,q4;
  RP::pseudo_div(q1,q2,q3,q4,D);
  CGAL_TEST(MSCCAST(D)*q1==q2*q3+q4);
  RP::pseudo_div(-q1,q2,q3,q4,D);
  CGAL_TEST(MSCCAST(D)*-q1==q2*q3+q4);
  RP::pseudo_div(q1,-q2,q3,q4,D);
  CGAL_TEST(MSCCAST(D)*q1==-q2*q3+q4);
  RP::pseudo_div(-q1,-q2,q3,q4,D);
  CGAL_TEST(MSCCAST(D)*-q1==-q2*q3+q4);
  RP qq1(5),qq2(17),qq3,qq4;
  RP::pseudo_div(qq1,qq2,qq3,qq4,D);
  CGAL_TEST(MSCCAST(D)*qq1==qq2*qq3+qq4);
  RP::pseudo_div(-qq1,qq2,qq3,qq4,D);
  CGAL_TEST(MSCCAST(D)*-qq1==qq2*qq3+qq4);
  RP::pseudo_div(qq1,-qq2,qq3,qq4,D);
  CGAL_TEST(MSCCAST(D)*qq1==-qq2*qq3+qq4);
  RP::pseudo_div(-qq1,-qq2,qq3,qq4,D);
  CGAL_TEST(MSCCAST(D)*-qq1==-qq2*qq3+qq4);
  CGAL_TEST(p10/p11 == p12);
  q3 = RP::gcd(q1,q2);
  CGAL_TEST(q3 == MSCCAST(1));
  CGAL_IO_TEST(p4,p1);
  CGAL::to_double(p6);
  CGAL::is_finite(p6);
  CGAL::is_valid(p6);
```

6.4 A Demo of RPolynomial

⟨*RPolynomial-demo.C*⟩≡

```
#include <CGAL/basic.h>
#include <CGAL/Gmpz.h>
#include <CGAL/Quotient.h>
#include <LEDA/string.h>
#include <LEDA/d_array.h>
#include <LEDA/stream.h>
#define RPOLYNOMIAL_EXPLICIT_OUTPUT
#ifndef CARTESIAN
#include <CGAL/leda_integer.h>
#include <CGAL/RPolynomial.h>
typedef leda_integer NT;
template <>
struct ring_or_field<leda_integer> {
  typedef ring_with_gcd kind;
  typedef leda_integer RT;
  static RT gcd(const RT& r1, const RT& r2)
  { return ::gcd(r1,r2); }
};
#else
#include <CGAL/leda_rational.h>
#include <CGAL/RPolynomial.h>
```

```
typedef leda_rational NT;
template <>
struct ring_or_field<leda_rational> {
  typedef field_with_div kind;
  typedef leda_rational FT;
  static FT gcd(const FT&, const FT&)
  { return FT(1); }
};
#endif

typedef CGAL::RPolynomial<NT> Poly;
typedef leda_string lstring;

enum { NEW,PLUS,MINUS,MULT,DIV,MOD,GCD,END,HIST,ERROR };

static leda_d_array<lstring, Poly> M(Poly(1));

void strip(lstring& s)
{ lstring res;
  for (int i=0; i<s.length(); ++i)
    if (s[i]!=' ') res += s[i];
  s = res;
}

bool contains(const lstring& s, const lstring& c, lstring& h, lstring& t)
{ int i = s.pos(c);
  if (i < 0) return false;
  h = s.head(i);
  t = s.tail(s.length()-i-c.length());
  return true;
}

int parse(const lstring& s, Poly& P1, Poly& P2, lstring& l)
{
  if (s =="end") { l="END"; return END; }
  if (s =="history") { return HIST; }
  lstring b,p1,p2,dummy,n;
  NT N;
  int res = ERROR;
  std::vector<NT> V;
  if (contains(s,"=",l,b)) {
    if (contains(b,"+",p1,p2)) res=PLUS;
    if (contains(b,"-",p1,p2)) res=MINUS;
    if (contains(b,"*",p1,p2)) res=MULT;
    if (contains(b,"/",p1,p2)) res=DIV;
    if (contains(b,"%",p1,p2)) res=MOD;
    if (contains(b,"gcd(",dummy,b) &&
        contains(b,")",b,dummy) &&
        contains(b,",",p1,p2)) res=GCD;
    if (contains(b,"(",dummy,b) &&
        contains(b,")",b,dummy)) {
      while (contains(b,",",n,b)) {
        string_istream IS(n.cstring()); IS >> N;
        V.push_back(N);
      }
      string_istream IS(b.cstring()); IS >> N;
      V.push_back(N);
```

```
      P1 = Poly(V.begin(),V.end());
      return NEW;
    }
  }
  P1=M[p1];
  P2=M[p2];
  return res;
}

int main(int argc, char* argv[])
{
  //SETDTHREAD(3);
  CGAL::set_pretty_mode ( cout );
  CGAL::set_pretty_mode ( cerr );
  cout << "insert simple assigments of the following form\n";
  cout << "v1 = (a0,a1,a2)      -> creates a0 + a1 x + a2 x^2\n";
  cout << "v1 = v2 [+-*/] v3    -> triggers arithmetic operation\n";
  cout << "v1 = gcd(v2,v3)      -> triggers gcd operation\n";
  cout << "end                  -> quits program"<< endl;
  cout << "history              -> prints all current vars"<< endl;
  lstring logname = lstring(argv[0])+".log";
  {
    file_istream logfile(logname.cstring());
    lstring line,s; Poly p;
    if (logfile) {
      cout << "initializing history\n";
      while ( (line.read_line(logfile), line!="") ) {
        string_istream line_is(line);
        line_is >> s >> p;
        cout << s << " = " << p << endl;
        M[s]=p;
      }
    }
    logfile.close();
  }

  lstring command,label;
  Poly p1,p2,res,dummy; NT D;
  bool loop=true;
  while (loop) {
    cout ⟨¡ ”⟩>";
    command.read_line(cin);
    strip(command); // strip whitespace
    switch (parse(command,p1,p2,label)) {
      case NEW:   res = p1;    break;
      case PLUS:  res = p1+p2; break;
      case MINUS: res = p1-p2; break;
      case MULT:  res = p1*p2; break;
      case DIV:   res = p1/p2; break;
      case MOD:   Poly::pseudo_div(p1,p2,dummy,res,D); break;
      case GCD:   res = Poly::gcd(p1,p2); break;
      case END:   loop=false; continue;
      case HIST: { lstring s;
        forall_defined(s,M) cout << s << " = " << M[s] << endl;
```

```
        continue;
      }
      default: cout << "wrong syntax\n"; continue;
    }
    cout << "  " << label << " = " << res << endl << endl;
    M[label] = res;
  }
  file_ostream logfile(logname);
  lstring s;
  forall_defined(s,M) logfile << s << " " << M[s] << endl;
  return 0;
}
```

6.5 Perl expansion of specialization

The following perl script expands the RPolynomial implementation above with respect to *int* and *double* specialization.

⟨*specialize*⟩≡

```
#! /opt/gnu/bin/perl
$specialize = 0;
$spec_arg = "";
while (<>) {
  # specializing class templates
  if ( /SPECIALIZE_CLASS\(.*\).*END/ ) {
    print specialize_members($spec_class,$nt_from);
    my @arglist = split(/ /,$class_to);
    foreach my $arg (@arglist) {
      my $copied_class = $spec_class;
      $copied_class =~
      s/template\s*\<class\s+$nt_from\>(\s*)class\s*(\w+)\s*/CGAL_TEMPLATE_NULL$1class $2\<$arg\> /sg;
      $copied_class =~
      s/template\s*\<class\s+p$nt_from\>(\s*)class\s*(\w+)\s*/CGAL_TEMPLATE_NULL$1class $2\<$arg\> /sg;
      $copied_class =~
      s/template\s*\<typename\s+$nt_from\>(\s*)class\s*(\w+)\s*/CGAL_TEMPLATE_NULL$1class $2\<$arg\> /sg;
      $copied_class =~ s/template\s*\<class\s+$nt_from\>/CGAL_TEMPLATE_NULL/sg;
      $copied_class =~ s/template\s*\<typename\s+$nt_from\>/CGAL_TEMPLATE_NULL/sg;
      $copied_class =~ s/([\W]|^)$nt_from([\W]|$)/$1$arg$2/smg;
      $copied_class =~ s/([\W]|^)p$nt_from([\W]|$)/$1$arg$2/smg;
      $copied_class =~ s/typename//sg;
      $copied_class =~ s/typedef $arg $arg/typedef $arg $nt_from/sg;
      $copied_class =~ s/\/\*\{\\M/\/\*\{\\X/sg;
      print specialize_members($copied_class,$arg);
    }
    $nt_from = "";
    $class_to = "";
    $spec_class = "";
  }
  if ( $spec_class ) {
```

```
    $spec_class .= $_; $_="";
  }
  if ( /SPECIALIZE_CLASS\((\w+),([\w\ ]+)\).*START/ ) {
    $nt_from = $1;
    $class_to = $2;
    $spec_class = "// CLASS TEMPLATE $nt_from: \n"
  }
  # specializing function templates
  if ( /SPECIALIZE\_FUNCTION\(.*\).*END/ ) {
    my @arglist = split(/ /,$args_to);
    foreach my $arg (@arglist) {
      my $copied_spec2 = $spec_func;
      my $header2 = "// SPECIALIZING inline to $nt_to:\n";
      $copied_spec2 =~ s/^.*?\n/$header2/;
      $copied_spec2 =~ s/$arg_from/$arg/sg;
      $copied_spec2 =~ s/template\s*\<class $arg\>/inline/sg;
      $copied_spec2 =~ s/template\s*\<typename $arg\>/inline/sg;
      print $copied_spec2;
      my $copied_spec1 = $spec_func;
      my $header1 = "// SPECIALIZING pure $arg params:\n";
      $copied_spec1 =~ s/^.*?\n/$header1/;
      $copied_spec1 =~ s/const\s*$arg_from\s*\&/const $arg\&/sg;
      print $copied_spec1;
    }
    print $spec_func;
    $spec_func = "";
    $args_to = "";
    $arg_from = "";
  }
  if ( $spec_func ) {
    $spec_func .= $_; $_ = "";
  }
  if ( /SPECIALIZE\_FUNCTION\((\w+),([\w\ ]+)\).*START/ ) {
    $arg_from = $1;
    $args_to = $2;
    $spec_func = "
  }
  # specializing implementation
  if ( /SPECIALIZE\_IMPLEMENTATION\(.*\).*END/ ) {
    my @arglist = split(/ /,$args_to);
    foreach my $arg (@arglist) {
      my $copied_spec2 = $spec_impl;
      my $header2 = "// SPECIALIZING to $nt_to:\n";
      $copied_spec2 =~ s/^.*?\n/$header2/;
      $copied_spec2 =~ s/$arg_from/$arg/sg;
      $copied_spec2 =~ s/template\s*\<class $arg\>//sg;
      $copied_spec2 =~ s/template\s*\<typename $arg\>//sg;
      print $copied_spec2;
    }
    print $spec_impl;
    $spec_impl = "";
    $args_to = "";
```

```
    $arg_from = "";
  }
  if ( $spec_impl ) {
    $spec_impl .= $_; $_ = "";
  }
  if ( /SPECIALIZE\_IMPLEMENTATION\((\w+),([\w\ ]+)\).*START/ ) {
    $arg_from = $1;
    $args_to = $2;
    $spec_impl = "
  }
  print;
}
sub specialize_members {
  local($spec,$nt) = @_;
  local($result,$args_to,$spec_mem,$kill);
  $kill=0;
  $spec =~ s/\n\n/\n>>><<</sg;
  @LINES = split(/\n/,$spec);
  foreach my $line (@LINES) {
    # general NT arg specializing for various types
    if ( $line =~ /SPECIALIZE\_MEMBERS\($args_to\).*END/ ) {
      my @arglist = split(/ /,$args_to);
      foreach my $arg (@arglist) {
        if ( $arg ne $nt ) {
          my $copied_spec = $spec_mem;
          my $header = "// SPECIALIZING_MEMBERS FOR const $arg\& \n";
          $copied_spec =~ s/.*\n/$header/;
          $copied_spec =~ s/const\s*$nt\s*\&/const $arg\&/sg;
          $result .= $copied_spec;
        }
      }
      $spec_mem = "";
      $args_to = "";
    }
    if ( $spec_mem ) {
      $spec_mem .= "$line\n";
    }
    if ( $line =~ /SPECIALIZE\_MEMBERS\(([\w\ ]+)\).*START/ ) {
      $args_to = $1;
      $spec_mem = "$line\n";
    }
    if ( $line =~ /KILL\s+$nt\s+START/ ) { $kill=1; }
    if ( $line =~ /KILL\s+$nt\s+END/ ) { $kill=0; $line = ""; }
    if ( $kill == 1 ) { $line = ""; }
    $result .= "$line\n";
  }
  $result =~ s/\n+/\n/sg;
  $result =~ s/\n>>><<</\n\n/sg;
  return $result;
}
```

⟨*specialization-test.C*⟩≡

```
#include <CGAL/basic.h>
#include <LEDA/integer.h>

template <typename T> class A;

template <typename T>
A<T> operator+ (const A<T>&, const A<T>&);
//A<int> operator+ (const int& num, const A<int>& a2);

template <typename T>
class A {
  T a;
  friend
  A<T> operator+ CGAL_NULL_TMPL_ARGS (const A<T>&, const A<T>&);
public:
  A() : a() {}
  A(T i) : a(i) {}
  A(int i) : a(i) {}

  A<T>& operator +=(const T& i)
  { a += (T)i; return *this; }

  static T R_;
};

template <class T> T A<T>::R_ = T(0);
#ifdef INITFORGNU
CGAL_TEMPLATE_NULL int A<int>::R_ = 0;
#endif
// symmetric op+
template <typename T>
A<T> operator+ (const A<T>& a1, const A<T>& a2)
{ return A<T>(a1.a+a2.a); }

// asymmetric op+
template <typename T>
A<T> operator+ (const T& num, const A<T>& a2)
{ return (A<T>(num) + a2); }

typedef leda_integer NT;
int main()
{
  A<NT> ad1(1),ad2(2), ad3;
  ad3 = ad1 + ad2;
  ad3 = 3 + ad1;
  A<NT>::R_ = 3;
  A<int>    ai1(1), ai2(2), ai3;
  ai3 = ai1 + ai2;
  ai3 = 3 + ai1;
  A<int>::R_ = 3;
  return 0;
}
```

Bibliography

[Coh93] H. Cohen. *A Course in Computational Algebraic Number Theory*, volume Graduate Texts in Mathematics 138. Springer Verlag, New York Berlin Heidelberg, 1993.

[Knu98] D.E. Knuth. *The Art of Computer Programming*, volume Volume 2, Seminumerical Algorithms, 3rd edition. Addison-Wesley, 1998.

[RSV84] H.-J. Reiffen, G. Scheja, and U. Vetter. *Algebra*, volume 110. BI Hochschultaschenbücher, Mannheim Wien Zürich, 1984.

7 Simple Extended Kernel

7.1 Introduction

It is convenient to extend the vision of standard rational points by so-called *non-standard* points. We have one non-standard point for each equivalence class of rays. Two rays are called equivalent if one is contained in the other. The geometric properties of non-standard points are derived by giving them a geometric interpretation by means of an infinimaximal R. R is a real variable. The value of R is finite but larger than the value of any concrete real number. Let F be the square box with corners $NW(-R,R)$, $NE(R,R)$, $SE(R,-R)$, and $SW(-R,-R)$. Let p be a non-standard point and let r be a ray defining it. If the frame F contains the source point of r then let $p(R)$ be the intersection of r with the frame F, if F does not contain the source of r then $p(R)$ is undefined. For a standard point let $p(R)$ be equal to p if p is contained in the frame F and let $p(R)$ be undefined otherwise. Clearly, for any standard or non-standard point $p(R)$ is defined for any sufficiently large R.

Let f be any function on standard points, say with k arguments. We call f *extensible* if for any k points $p_1, \ldots, p_k$ the function value $f(p_1(R), \ldots, p_k(R))$ is constant for all sufficiently large R and has the same value as the function evaluated at a fixed large enough value R_0. We also consider geometric constructions. Let g be a construction on standard points constructing a tuple of l points from a tuple of k points. We call g *extensible* if for any k points $p_1, \ldots, p_k$ the construction is closed in our set of extended points: if it constructs a point tuple $q_1, \ldots, q_l$ of extended points for all sufficiently large R with the property that by fixing a large enough R_0 the tuple is the result of the standard construction. As we will see in a moment the predicates *lexicographic order* of points, *orientation*, and *side_of_circle* are extensible. Also the calculation of the point in the *intersection* of line segments defined on two pairs of points is extensible.

For a formal definition of extended points, extended segments and the corresponding predicates see the technical report [SM00].

7.2 Homogeneous Representation

We implement planar extended points by a homogeneous component representation in a polynomial ring type which provides standard ring operations like $+, -, *$. The definition of extended points puts constraints on the kind of polynomials representing the coordinates. We have seen that our extensible predicates are defined via polynomials in the coordinate polynomials and as such are extensible via the limit process on polynomials. Going to infinity the value of a polynomial is determined by the highest-order nonzero coefficient.

Using extensible predicates on an input set of extended points in the execution of an algorithm we can determine a concrete value R_0 which ensures their extendibility for all $R \geq R_0$ (for each evaluation determine one R_i and take the maximum of all). Plugging R_0 into all coordinate polynomials leads back to standard affine geometry and standard predicates. This gives us the possibility to argue also about the correctness of our algorithms. If the algorithm is proven to be correct for standard geometry and it computes a certain output then it will also calculate some extended geometric result when plugging in extended points and when all geometric predicates are extensible.

Note that in this way we can design algorithms that use ray like structures much simpler by enclosing finite structures into the box F and pruning the rays by means of the frame in a ray tip. The calculation with the extended points makes algorithmic decisions trivial if the predicates we use are extensible in the above sense.

In this section we will describe how extended points are stored: they are composed from the 2D CGAL kernel point type *Homogeneous*<...>::*Point_2* and the polynomial ring number type *RPolynomial*<...>. We also describe how the affine world of standard points and rays interacts with the unifying concept extended point. This interaction has two directions: the construction of an extended point from a standard object (point or ray) and the reversal extraction depending on the character of the extended point. Afterwards, we show how simple it is to implement predicates and the intersection construction on top of the genericity of CGAL's standard kernel. We will encounter the problem of simplification of polynomials there. And finally, we give some details about visualization issues of extended objects.

We often use the short term *epoint* to denote extended points. Each epoint is either a standard point, one of the corner ray points or lies in the relative interior of one of the frame segments.

Extended Points

The tip of a ray l can be described in two ways. First in form of its underlying oriented line equation $ax+by+c=0$. But also by its point-vector form $p = p_0 + \lambda d$. The former is the standard representation of lines. The latter is more suitable to explore the character of the corresponding extended point.

Starting from the second representation we have two points p_0 and p_0+d on the line. Now all points on the line can also be described by the determinant equation:

$$\begin{vmatrix} 1 & 1 & 1 \\ x_0 & x_0+d_x & x \\ y_0 & y_0+d_y & y \end{vmatrix} = 0$$

and developing this by the last column leads us to $-d_y x + d_x y + (x_0 d_y - y_0 d_x) = 0$. Thus the direction vector is:

$$d = \begin{pmatrix} d_x \\ d_y \end{pmatrix} = \begin{pmatrix} b \\ -a \end{pmatrix}$$

A point on the line specified by the line equation is:

$$p_0 = \begin{cases} (0, -c/b) & b \neq 0 \\ (-c/a, 0) & a \neq 0 \end{cases}$$

Both a and b cannot be zero. In the following assume *Line_2* to be a model for the CGAL standard geometric kernel.

⟨*line conversion methods*⟩≡
```
static RT dx(const Line_2& l) { return l.b(); }
static RT dy(const Line_2& l) { return -l.a(); }
```

Depending on the slope $m = d_y/d_x(d_x \neq 0)$ of a line l we can define its vertical distance to the origin. If $d_x \neq 0$ ($|m| \neq \infty$) then the ordinate intersection d_o determines that distance $d_o = -c/b$.

⟨*line conversion methods*⟩+≡
```
static FT ordinate_distance(const Line_2& l)
{ return Kernel::make_FT(-l.c(),l.b()); }
```

We introduce enumeration specifiers that describe extended points.

⟨*enumerate extended point character*⟩≡
```
enum Point_type { SWCORNER=1, LEFTFRAME, NWCORNER,
                  BOTTOMFRAME, STANDARD, TOPFRAME,
                  SECORNER, RIGHTFRAME, NECORNER };
```

Now if we look at a non-standard point p with underlying line equation $ax+by+c = 0$ the frame segment which is hit by the ray tip is determined by the slope and in case $|m| = 1$ by the distance d_o defined above. Look for example at a non-standard point hitting the left frame segment. This is generally the case if $d_x < 0$ and $|m| < 1$. The latter is equivalent to the condition $|d_x| > |d_y|$. A special case is $|m| = 1$. Then, we only hit the left segment if either $m = -1 \wedge d_o < 0$ or $m = 1 \wedge d_o > 0$. The latter can be checked by $sign(d_y) == -sign(d_o)$. Note that because $|m| \leq 1$ the line indeed intersects the y-axis. The other cases follow by symmetric reasoning.

⟨*line conversion methods*⟩+≡
```
static Point_type determine_type(const Line_2& l)
{
  RT adx = abs(dx(l)), ady = abs(dy(l));
  int sdx = sign(dx(l)), sdy = sign(dy(l));
  int cmp_dx_dy = compare(adx,ady), s(1);
  if (sdx < 0 && ( cmp_dx_dy > 0 || cmp_dx_dy == 0 &&
      sdy != (s = sign(ordinate_distance(l))))) {
    if (0 == s) return ( sdy < 0 ? SWCORNER : NWCORNER );
    else        return LEFTFRAME;
  } else if (sdx > 0 && ( cmp_dx_dy > 0 || cmp_dx_dy == 0 &&
             sdy != (s = sign(ordinate_distance(l))))) {
    if (0 == s) return ( sdy < 0 ? SECORNER : NECORNER );
    else        return RIGHTFRAME;
  } else if (sdy < 0 && ( cmp_dx_dy < 0 || cmp_dx_dy == 0 &&
             ordinate_distance(l) < FT(0))) {
    return BOTTOMFRAME;
  } else if (sdy > 0 && ( cmp_dx_dy < 0 || cmp_dx_dy == 0 &&
             ordinate_distance(l) > FT(0))) {
    return TOPFRAME;
  }
  CGAL_assertion_msg(false," determine_type: degenerate line.");
  return (Point_type)-1; // never come here
}
```

All the operations above are packaged into the class *Line_to_epoint*<*R*>, where *R* is a model of the CGAL standard 2d geometric kernel. From *R* we derive the types *RT*, *FT*, and *Line_2* as used in the code.

⟨*Line_to_epoint.h*⟩≡
```
  ⟨CGAL L2E Header⟩
  #ifndef CGAL_LINE_TO_EPOINT_H
  #define CGAL_LINE_TO_EPOINT_H

  CGAL_BEGIN_NAMESPACE

  template <class Kernel_>
  struct Line_to_epoint {
    typedef Kernel_ Kernel;
    typedef typename Kernel::RT RT;
    typedef typename Kernel::FT FT;
    typedef typename Kernel::Line_2 Line_2;
    ⟨enumerate extended point character⟩
    ⟨line conversion methods⟩
  };

  CGAL_END_NAMESPACE
  #endif //CGAL_LINE_TO_EPOINT_H
```

Any non-standard point can be expressed as a pair of two polynomials in a variable R — our infimaximal symbolic number. Let's look at our example again. Our point p on the left frame segment supported by the line $ax+by+c=0$ can be described by the tuple $(-R, a/bR - c/b)$. Accordingly, a ray tip on the upper frame segment can be described by $(-b/aR - c/a, R)$. Note that the denominators are nonzero in both cases due to their frame position. Thus we can store epoints in terms of linear polynomials $mR+n$. For standard points the polynomials are just constant with $m=0$. We give the representation of all points in homogeneous representation, such that all coefficients can be represented by a ring type.

$$\begin{array}{ll} \text{STANDARD} & p=(x,y,w) \\ \text{CORNER} & p=(\pm R, \pm R, 1) \\ \text{LEFTFRAME} & p=(-bR, aR-c, b) \\ \text{RIGHTFRAME} & p=(bR, -aR-c, b) \\ \text{BOTTOMFRAME} & p=(bR-c, -aR, a) \\ \text{TOPFRAME} & p=(-bR-c, aR, a) \end{array} \tag{7.1}$$

The general representation can be taken to be $p=(m_xR+n_x, m_yR+n_y, w)$ where $m_{x,y}, n_{x,y}, w$ are objects of a ring number type. We provide the functionality of extended points bundled into an extended geometry kernel. This kernel carries the types, predicates, and constructions that we need in our algorithms. The kernel concept is specified in the manual page *ExtendedKernelTraits_2* of the appendix.

A decorator wraps functionality

We obtain the extended point class by plugging our polynomial arithmetic type into the standard homogeneous point type from the CGAL kernel. We create a traits class *Extended_homogeneous*<*RT*> that carries all types and methods that are used in our algorithmic framework.

To ensure the special character of homogeneous points concerning their coordinates and to offer a comfortable construction of such points we make *Extended_homogeneous*<*RT*> a decorator/factory data type [GHJV95] for the geometric objects. Construction and conversion routines can be accessed as methods of the factory.

⟨*extended homogeneous*⟩≡

```
template <class RT_>
class Extended_homogeneous : public
  CGAL::Homogeneous< CGAL::RPolynomial<RT_> > { public:

  ⟨extended homogeneous kernel interface types⟩
  ⟨extended homogeneous kernel members⟩
};
```

We introduce the standard affine types into our kernel by prefixing them accordingly. The extended types carry the typenames without the prefix. Note that this decorator serves as a traits class to be used in algorithms that are based on our infimaximal frame. It is also the glue between the CGAL standard kernel and the extended geometric objects.

⟨*extended homogeneous kernel interface types*⟩≡

```
typedef CGAL::Homogeneous< CGAL::RPolynomial<RT_> >  Base;
typedef Extended_homogeneous<RT_> Self;

typedef CGAL::Homogeneous<RT_> Standard_kernel;
typedef RT_ Standard_RT;
typedef typename Standard_kernel::FT Standard_FT;
typedef typename Standard_kernel::Point_2     Standard_point_2;
typedef typename Standard_kernel::Segment_2   Standard_segment_2;
typedef typename Standard_kernel::Line_2      Standard_line_2;
typedef typename Standard_kernel::Direction_2 Standard_direction_2;
typedef typename Standard_kernel::Ray_2       Standard_ray_2;
typedef typename Standard_kernel::Aff_transformation_2
  Standard_aff_transformation_2;

typedef typename Base::RT RT;
typedef typename Base::Point_2      Point_2;
typedef typename Base::Segment_2    Segment_2;
typedef typename Base::Direction_2  Direction_2;
typedef typename Base::Line_2       Line_2;
// used only internally
enum Point_type { SWCORNER=1, LEFTFRAME, NWCORNER,
                  BOTTOMFRAME, STANDARD, TOPFRAME,
                  SECORNER, RIGHTFRAME, NECORNER };
```

We now implement the construction deduced above. For a non-standard point on the upper frame segment supported by a line $l \equiv ax + by + c = 0$ the polynomial coefficients are $m = -b/a, n = -c/a$. Accordingly on the left frame segment $m = -a/b, n = -c/b$.

⟨*non-standard point construction*⟩≡

```
Point_2 epoint(const Standard_RT& m1, const Standard_RT& n1,
               const Standard_RT& m2, const Standard_RT& n2,
                              const Standard_RT& n3) const
{ return Point_2(RT(n1,m1),RT(n2,m2),RT(n3)); }

Point_2 construct_point(const Standard_line_2& l, Point_type& t) const
{
  t = (Point_type)Line_to_epoint<Standard_kernel>::determine_type(l);
  Point_2 res;
  switch (t) {
    case SWCORNER:   res = epoint(-1, 0, -1, 0, 1); break;
    case NWCORNER:   res = epoint(-1, 0,  1, 0, 1); break;
    case SECORNER:   res = epoint( 1, 0, -1, 0, 1); break;
    case NECORNER:   res = epoint( 1, 0,  1, 0, 1); break;
    case LEFTFRAME:
      res = epoint(-l.b(), 0,  l.a(), -l.c(), l.b()); break;
    case RIGHTFRAME:
      res = epoint( l.b(), 0, -l.a(), -l.c(), l.b()); break;
    case BOTTOMFRAME:
      res = epoint( l.b(), -l.c(), -l.a(), 0, l.a()); break;
    case TOPFRAME:
      res = epoint(-l.b(), -l.c(),  l.a(), 0, l.a()); break;
    default: CGAL_assertion_msg(0,"EPoint type not correct!");
  }
  return res;
}
```

Type determination

To evaluate the results of an algorithm one also needs an operation that deduces the type from an epoint p. From the polynomial representation we can easily defer this type by checking the homogeneous components $p.hx(\,)$ and $p.hy(\,)$. Of course standard points have zero degree in both x- and y-components. For any non-standard p on the frame we know that the relative interior of the frame box segments is specified by the condition that $|p.hx(\,)| \gtrless |p.hy(\,)|$. The sign of the larger component (larger with respect to its absolute value) determines the box segment. Equality $|p.hx(\,)| = |p.hy(\,)|$ specifies the corners of the box.

```
Point_type type(const Point_2& p)
{
  CGAL_assertion(p.hx().degree()>=0 && p.hy().degree()>=0 );
  CGAL_assertion(p.hw().degree()==0);
  if (p.hx().degree() == 0 && p.hy().degree() == 0)
    return STANDARD;
  // now we are on the square frame box
  RT rx = p.hx(), ry = p.hy();
  int sx = sign(rx), sy = sign(ry);
  if ( sx < 0 ) rx = -rx;
  if ( sy < 0 ) ry = -ry;
  if ( rx > ry )
    if (sx > 0) return RIGHTFRAME; else return LEFTFRAME;
  if ( rx < ry )
```

```
      if (sy > 0) return TOPFRAME; else return BOTTOMFRAME;
    // now (rx == ry)
    if ( sx == sy ) {
      if (sx < 0) return SWCORNER; else return NECORNER;
    } else { CGAL_assertion(sx==-sy);
      if (sx < 0) return NWCORNER; else return SECORNER;
    }
  }
```

Visualization

We finally treat the problem of how to visualize extended objects. Given a set S of extended points let us determine a concrete frame radius R_0 such that all standard points in S are contained inside our frame but also all non-standard points in S can be drawn on the correct frame box segments. Note that the latter is not trivially true for arbitrary small values of R_0.

Consider a line l with slope m. If $|m| \neq 1$ the line l intersects both angular bisectors of our coordinate frame. The intersection point of the larger absolute coordinates determines a lower bound for R_0. If $|m| = 1$ a natural lower bound for R_0 is half the length of the ordinate segment on the y-axis between l and the origin. See Figure 7.1.

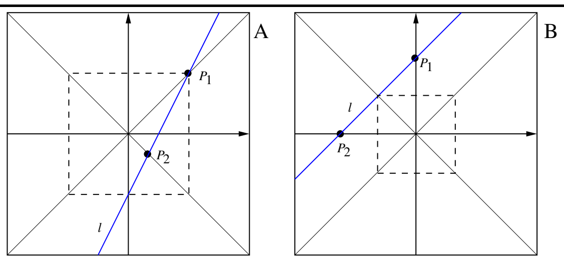


Figure 7.1: The point P_1 determines a lower bound for the frame radius R_0 to display the non-standard points at the tips of line l. In case (A) we take the absolute value of its coordinates, in case (B) we take half of its distance to the origin.

For our polynomial representation (m_xR+n_x, m_yR+n_y, w) we know that for points in the interior of the frame box segments it holds that $|(m_xR+n_x)| \gtrless |(m_yR+n_y)|$. In either case we can set both polynomials equal and resolve for R if $|m_x| \neq |m_y|$. $R = |(n_x - n_y)|/|(m_y - m_x)|$ presumed the line is not parallel to any of the angular bisectors of the coordinate frame. If $|m_x| = |m_y|$ then the constant parts n_x/w or n_y/w determine the abscissa or ordinate distances between the underlying line and the origin (depending on the frame segment that contains the extended point). At least one of n_x/w or n_y/w is actually zero (by definition of our extended points). In this case the minimum frame radius R_0 is half the absolute value of the abscissa or ordinate distance of the line to the origin.

We now code this determination of R_0 for an iterator range of extended points. Note that the common denominator of the homogenous representation is always a constant and positive. Note that we round the integral division operations on the ring type up.

⟨*determining a lower bound for R*⟩≡

```
  template <class Forward_iterator>
  void determine_frame_radius(Forward_iterator start, Forward_iterator end,
                              Standard_RT& R0) const
```

```
{ Standard_RT R, mx, nx, my, ny;
  while ( start != end ) {
    Point_2 p = *start++;
    if ( is_standard(p) ) {
      R = max(abs(p.hx()[0])/p.hw()[0],
              abs(p.hy()[0])/p.hw()[0]);
    } else {
      RT rx = abs(p.hx()), ry = abs(p.hy());
      mx = ( rx.degree()>0 ? rx[1] : 0 ); nx = rx[0];
      my = ( ry.degree()>0 ? ry[1] : 0 ); ny = ry[0];
      if ( mx > my )      R = abs((ny-nx)/(mx-my));
      else if ( mx < my ) R = abs((nx-ny)/(my-mx));
      else /* mx == my */ R = abs(nx-ny)/(2*p.hw()[0]);
    }
    R0 = max(R+1,R0);
  }
}
```

Extended predicates

Remember why the predicates *compare_xy*, *orientation*, *side_of_circle* are extensible. The first is just a cascaded comparison of coordinates (sign of their difference), the latter are sign-of-determinant calculations. The orientation predicate on three points is defined by the homogeneous expression:

$$\text{orientation}(p1,p2,p3) = \text{sign}\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ w_1 & w_2 & w_3 \end{vmatrix}$$

Thus, evaluation of the sign means looking at the sign of the coefficient of *R* if it is nonzero, or at the sign of the constant term if it is zero. The corresponding functionality is programmed into the sign function of our polynomial ring number type *RPolynomial<NT>*. Thus adding the following methods to the extended geometry traits class implements the functionality via the kernel base class.

```
int compare_xy(const Point_2& p1, const Point_2& p2) const
{ typename Base::Compare_xy_2 _compare_xy = compare_xy_2_object();
  return _compare_xy(p1,p2);
}
int orientation(const Point_2& p1, const Point_2& p2, const Point_2& p3)
{ typename Base::Orientation_2 _orientation = orientation_2_object();
  return _orientation(p1,p2,p3);
}
```

Extended constructions

Algorithms in computational geometry can be grouped into three categories: *subset selection*, *computation*, and *decision* [PS85, 1.4]. Algorithms of the first type resort to predicates, algorithms of the second type construct geometric objects. To cover this necessity software libraries like LEDA or CGAL offer a set of so called constructions in their geometric kernels. We have already shown that the intersection construction is extendible to be used with extended segments.

First we want to present three examples how the standard algebraic calculation of intersection points is blown up by common polynomial factors.

The coefficients of a line l through two points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ are

$$a = y_1 - y_2, b = x_2 - x_1, c = x_1 y_2 - x_2 y_1. \tag{7.2}$$

The intersection point is defined by the common point of the two underlying lines $l_i \equiv (a_i x + b_i y + c_i = 0), i = 1, 2$. Their common point is then obtained by solving the linear system which has a solution if the lines are not parallel. We obtain

$$p_i = (b_1 c_2 - b_2 c_1, a_2 c_1 - a_1 c_2, a_1 b_2 - a_2 b_1) \tag{7.3}$$

in homogeneous representation. Apart from the formal argument why these quotients contain common factors and how they can be simplified to a minimal representation we give three examples.

two non-standard points on one frame segment — Look at the case where the frame segment is the upper one. Thus $p_i = (m_i R + n_i, R), i = 1, 2$. According to equation (7.2) we obtain

$$\begin{aligned} a &= R - R = 0 \\ b &= (m_2 - m_1)R + (n_2 - n_1) \\ c &= (m_1 - m_2)R^2 + (n_1 - n_2)R = bR \end{aligned}$$

The common factor is b, the underlying line is $l \equiv by + c = 0 \Leftrightarrow y - R = 0$. We obtain a simple parameterized version of a horizontal line supporting the upper frame segment. The three other cases are symmetric.

two non-standard points spanning a standard line — Look at the case of one point p_1 on the lower frame segment, p_2 on the upper frame segment, both on a line $l \equiv ax + by + c = 0$ where we assume orientation from p_1 to p_2. We get according to Construction (7.1) $p_1 = (b/a\, R - c/a, -R)$, $p_2 = (-b/a\, R - c/a, R)$. According to equation (7.2) we obtain

$$\begin{aligned} a' &= -2R \\ b' &= -2R\, b/a \\ c' &= (b/a - b/a)\, R^2 - 2R\, c/a = -2R\, c/a \end{aligned}$$

The common factor is $-2R$. The underlying line is $l' \equiv a'x + b'y + c' = 0 \Leftrightarrow ax + by + c = 0$ as multiplying by a and dividing by $-2R$ does not change the line.

one non-standard point and one standard point spanning a standard ray — We look again at a line $l \equiv ax + by + c = 0$ supporting p_2 on the upper frame segment and a standard point p_1 on this line. We have $p_1 = (b/a\, y_0 - c/a, -y_0)$, $p_2 = (-b/a\, R - c/a, R)$. According to equation (7.2) we obtain

$$\begin{aligned} a' &= (y_0 - R) \\ b' &= -b/a\, R - c/a + b/a\, y_0 + c/a = b/a\, (y_0 - R) \\ c' &= -b/a\, y_0 R - c/a\, R + b/a\, y_0 R + c/a\, y_0 = c/a\, (y_0 - R) \end{aligned}$$

The common factor is $(y_0 - R)$.

Note that the polynomial factors are very simple. The greatest common divisor operation and the polynomial division scheme of the *RPolynomial* data type can be used to do the simplificication.

```
void simplify(Point_2& p)
{ RT x=p.hx(), y=p.hy(), w=p.hw();
  RT common = x.is_zero() ? y : gcd(x,y);
  common = gcd(common,w);
  p = Point_2(x/common,y/common,w/common);
}
```

Now the intersection uses the kernel operation and simplifies the resulting point afterwards.

```
Point_2 intersection(
  const Segment_2& s1, const Segment_2& s2)
{ typename Base::Intersect_2 _intersect = intersect_2_object();
  typename Base::Construct_line_2 _line = construct_line_2_object();
  Point_2 p;
  CGAL::Object result = _intersect(_line(s1),_line(s2));
  if ( !CGAL::assign(p, result) )
  CGAL_assertion_msg(false,"intersection: no intersection.");
  simplify(p);
  return p;
}
```

We offer construction from standard affine kernel objects.

⟨*extended homogeneous kernel members*⟩≡

```
public:
⟨non-standard point construction⟩
⟨determining a lower bound for R⟩

Point_2 construct_point(const Standard_point_2& p) const
{ return Point_2(p.hx(), p.hy(), p.hw()); }

Point_2 construct_point(const Standard_point_2& p1,
                        const Standard_point_2& p2,
                        Point_type& t) const
{ return construct_point(Standard_line_2(p1,p2),t); }

Point_2 construct_point(const Standard_line_2& l) const
{ Point_type dummy; return construct_point(l,dummy); }

Point_2 construct_point(const Standard_point_2& p1,
                        const Standard_point_2& p2) const
{ return construct_point(Standard_line_2(p1,p2)); }

Point_2 construct_point(const Standard_point_2& p,
                        const Standard_direction_2& d) const
{ return construct_point(Standard_line_2(p,d)); }

Point_2 construct_opposite_point(const Standard_line_2& l) const
{ Point_type dummy; return construct_point(l.opposite(),dummy); }

Point_type type(const Point_2& p) const
{
  CGAL_assertion(p.hx().degree()>=0 && p.hy().degree()>=0 );
  CGAL_assertion(p.hw().degree()==0);
  if (p.hx().degree() == 0 && p.hy().degree() == 0)
    return STANDARD;
  // now we are on the square frame
```

```
  RT rx = p.hx();
  RT ry = p.hy();
  int sx = sign(rx);
  int sy = sign(ry);
  if (sx < 0) rx = -rx;
  if (sy < 0) ry = -ry;
  if (rx>ry) {
    if (sx > 0) return RIGHTFRAME;
    else        return LEFTFRAME;
  }
  if (rx<ry) {
    if (sy > 0) return TOPFRAME;
    else        return BOTTOMFRAME;
  }
  // now (rx == ry)
  if (sx==sy) {
    if (sx < 0) return SWCORNER;
    else        return NECORNER;
  } else { CGAL_assertion(sx==-sy);
    if (sx < 0) return NWCORNER;
    else        return SECORNER;
  }
}

bool is_standard(const Point_2& p) const
{ return (type(p)==STANDARD);  }

Standard_point_2 standard_point(const Point_2& p) const
{ CGAL_assertion(type(p)==STANDARD);
  CGAL_assertion(p.hw() > RT(0));
  return Standard_point_2(p.hx()[0],p.hy()[0],p.hw()[0]);
}

Standard_line_2 standard_line(const Point_2& p) const
{ CGAL_assertion(type(p)!=STANDARD);
  RT hx = p.hx(), hy = p.hy(), hw = p.hw();
  Standard_RT dx,dy;
  if (hx.degree()>0) dx=hx[1]; else dx=0;
  if (hy.degree()>0) dy=hy[1]; else dy=0;
  Standard_point_2 p0(hx[0],hy[0],hw[0]);
  Standard_point_2 p1(hx[0]+dx,hy[0]+dy,hw[0]);
  return Standard_line_2(p0,p1);
}

Standard_ray_2 standard_ray(const Point_2& p) const
{ CGAL_assertion(type(p)!=STANDARD);
  Standard_line_2 l = standard_line(p);
  Standard_direction_2 d = l.direction();
  Standard_point_2 q = l.point(0);
  return Standard_ray_2(q,d);
}

Point_2 NE() const { return construct_point(Standard_line_2(-1, 1,0)); }

Point_2 SE() const { return construct_point(Standard_line_2( 1, 1,0)); }

Point_2 NW() const { return construct_point(Standard_line_2(-1,-1,0)); }

Point_2 SW() const { return construct_point(Standard_line_2( 1,-1,0)); }
```

```
Line_2 upper() const { return construct_line(NW(),NE()); }
Line_2 lower() const { return construct_line(SW(),SE()); }
Line_2 left()  const { return construct_line(SW(),NW()); }
Line_2 right() const { return construct_line(SE(),NE()); }

Point_2 source(const Segment_2& s) const
{ typename Base::Construct_source_point_2 _source =
    construct_source_point_2_object();
  return _source(s); }
Point_2 target(const Segment_2& s) const
{ typename Base::Construct_target_point_2 _target =
    construct_target_point_2_object();
  return _target(s); }
Segment_2 construct_segment(const Point_2& p, const Point_2& q) const
{ typename Base::Construct_segment_2 _segment =
    construct_segment_2_object();
  return _segment(p,q); }
void simplify(Point_2& p) const
{
  RT x=p.hx(), y=p.hy(), w=p.hw();
  RT common = x.is_zero() ? y : RT::gcd(x,y);
  common = RT::gcd(common,w);
  p = Point_2(x/common,y/common,w/common);
}
Line_2 construct_line(const Standard_line_2& l)  const
{ return Line_2(l.a(),l.b(),l.c()); }
Line_2 construct_line(const Point_2& p1, const Point_2& p2) const
{ Line_2 l(p1,p2);
  RT a=l.a(), b=l.b(), c=l.c();
  RT common = a.is_zero() ? b : RT::gcd(a,b);
  common = RT::gcd(common,c);
  l =  Line_2(a/common,b/common,c/common);
  return l;
}
int orientation(const Segment_2& s, const Point_2& p) const
{ typename Base::Orientation_2 _orientation =
    orientation_2_object();
  return static_cast<int> ( _orientation(source(s),target(s),p) );
}
int orientation(const Point_2& p1, const Point_2& p2, const Point_2& p3)
const
{ typename Base::Orientation_2 _orientation =
    orientation_2_object();
  return static_cast<int> ( _orientation(p1,p2,p3) );
}
bool leftturn(const Point_2& p1, const Point_2& p2, const Point_2& p3)
const
{ return orientation(p1,p2,p3) > 0; }
bool is_degenerate(const Segment_2& s) const
```

```
{ typename Base::Is_degenerate_2 _is_degenerate =
    is_degenerate_2_object();
  return _is_degenerate(s); }
int compare_xy(const Point_2& p1, const Point_2& p2) const
{ typename Base::Compare_xy_2 _compare_xy =
    compare_xy_2_object();
  return static_cast<int>( _compare_xy(p1,p2) );
}
int compare_x(const Point_2& p1, const Point_2& p2) const
{ typename Base::Compare_x_2 _compare_x =
    compare_x_2_object();
  return static_cast<int>( _compare_x(p1,p2) );
}
int compare_y(const Point_2& p1, const Point_2& p2) const
{ typename Base::Compare_y_2 _compare_y =
    compare_y_2_object();
  return static_cast<int>( _compare_y(p1,p2) );
}
Point_2 intersection(
  const Segment_2& s1, const Segment_2& s2) const
{ typename Base::Intersect_2 _intersect =
    intersect_2_object();
  typename Base::Construct_line_2 _line =
    construct_line_2_object();
  Point_2 p;
  CGAL::Object result =
    _intersect(_line(s1),_line(s2));
  if ( !CGAL::assign(p, result) )
  CGAL_assertion_msg(false,"intersection: no intersection.");
  simplify(p);
  return p;
}
Direction_2 construct_direction(
  const Point_2& p1, const Point_2& p2) const
{ typename Base::Construct_direction_of_line_2 _direction =
    construct_direction_of_line_2_object();
  return _direction(construct_line(p1,p2)); }
bool strictly_ordered_ccw(const Direction_2& d1,
  const Direction_2& d2, const Direction_2& d3) const
{
  if ( d1 < d2 )  return ( d2 < d3 )||( d3 <= d1 );
  if ( d1 > d2 )  return ( d2 < d3 )&&( d3 <= d1 );
  return false;
}
bool strictly_ordered_along_line(
  const Point_2& p1, const Point_2& p2, const Point_2& p3) const
{ typename Base::Are_strictly_ordered_along_line_2 _ordered =
    are_strictly_ordered_along_line_2_object();
  return _ordered(p1,p2,p3);
}
bool contains(const Segment_2& s, const Point_2& p) const
```

```
{ typename Base::Has_on_2 _contains = has_on_2_object();
  return _contains(s,p);
}
bool first_pair_closer_than_second(
  const Point_2& p1, const Point_2& p2,
  const Point_2& p3, const Point_2& p4) const
{ return ( squared_distance(p1,p2) < squared_distance(p3,p4) ); }
```

We can transform points, but have to be careful about their representation. The method *transform* just applies the standard matrix multiplication of planar affine transformations to our extended points. Afterwards we scale the represenstation back to our square box by the method *scale_first_by_second*.

Note that the correctness of the following piece of code is due to two facts:

- the larger absolute values of the two tranformed components determines the coordinate that can be scaled to the square frame.
- any coordinate transformation $R \leftarrow mR + n$ is a legal transformation of our point representation $(m_x R + n_x, m_y R + n_y)$. One can easily show that both lie on the same line equation.

⟨*extended homogeneous kernel members*⟩+≡

```
void scale_first_by_second(RT& r1, RT& r2, RT& w) const
{ CGAL_assertion(w.degree()==0&&w!=RT(0)&& r2[1]!=Standard_RT(0));
  Standard_RT w_res = w[0]*r2[1];
  int sm2 = sign(r2[1]);
  RT r2_res = RT(Standard_RT(0),sm2 * w_res);
  RT r1_res = RT(r2[1]*r1[0]-r1[1]*r2[0], w[0]*r1[1]*sm2);
  r1 = r1_res; r2 = r2_res; w = w_res;
}
Point_2 transform(const Point_2& p,
                  const Standard_aff_transformation_2& t) const
{
  RT tpx = t.homogeneous(0,0)*p.hx() + t.homogeneous(0,1)*p.hy() +
    t.homogeneous(0,2)*p.hw();
  RT tpy = t.homogeneous(1,0)*p.hx() + t.homogeneous(1,1)*p.hy() +
    t.homogeneous(1,2)*p.hw();
  RT tpw = t.homogeneous(2,2)*p.hw();
  if ( is_standard(p) ) {
    Point_2 res(tpx,tpy,tpw); simplify(res);
    return res;
  }
  RT tpxa = abs(tpx);
  RT tpya = abs(tpy);
  if ( tpxa > tpya ) {
    scale_first_by_second(tpy,tpx,tpw);
  } else { // tpxa <= tpya
    scale_first_by_second(tpx,tpy,tpw);
  }
  Point_2 res(tpx,tpy,tpw); simplify(res);
  return res;
}
const char* output_identifier() const { return "Extended_homogeneous"; }
```

The file wrapper is here.

⟨*Extended_homogeneous.h*⟩≡

```
⟨CGAL EH Header⟩
#ifndef CGAL_EXTENDED_HOMOGENEOUS_H
#define CGAL_EXTENDED_HOMOGENEOUS_H

#include <CGAL/basic.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Point_2.h>
#include <CGAL/Line_2_Line_2_intersection.h>
#include <CGAL/squared_distance_2.h>
#ifndef _MSC_VER
#include <CGAL/RPolynomial.h>
#else
#include <CGAL/RPolynomial_MSC.h>
#define RPolynomial RPolynomial_MSC
#endif
#undef _DEBUG
#define _DEBUG 5
#include <CGAL/Nef_2/debug.h>
#include <CGAL/Nef_2/Line_to_epoint.h>

CGAL_BEGIN_NAMESPACE

template <class T> class Extended_homogeneous;

⟨extended homogeneous⟩

#undef RPolynomial
CGAL_END_NAMESPACE
#endif // CGAL_EXTENDED_HOMOGENEOUS_H
```

We provide a similar kernel based on a *cartesian representation* of points. In this case we use *RPolynomial<NT>* fed with a field type and use standard polynomial division for simplification in the intersection construction. The latter replaces the *gcd* operation of the ring type in the homogeneous case.

7.3 Cartesian Representation

We obtain our epoint class by plugging our polynomial arithmetic type into the standard Cartesian point type from CGAL.

⟨*extended cartesian*⟩≡

```
template <class pFT>
class Extended_cartesian : public
  CGAL::Cartesian< CGAL::RPolynomial<pFT> > { public:
typedef CGAL::Cartesian< CGAL::RPolynomial<pFT> > Base;
typedef Extended_cartesian<pFT> Self;

typedef CGAL::Cartesian<pFT> Standard_kernel;

typedef typename Standard_kernel::RT Standard_RT;

typedef typename Standard_kernel::FT Standard_FT;
```

```
typedef typename Standard_kernel::Point_2     Standard_point_2;
typedef typename Standard_kernel::Segment_2   Standard_segment_2;
typedef typename Standard_kernel::Line_2      Standard_line_2;
typedef typename Standard_kernel::Direction_2 Standard_direction_2;
typedef typename Standard_kernel::Ray_2       Standard_ray_2;
typedef typename Standard_kernel::Aff_transformation_2
  Standard_aff_transformation_2;
typedef typename Base::RT RT;
typedef typename Base::FT FT;
typedef typename Base::Point_2      Point_2;
typedef typename Base::Segment_2    Segment_2;
typedef typename Base::Line_2       Line_2;
typedef typename Base::Direction_2  Direction_2;
enum Point_type { SWCORNER=1, LEFTFRAME, NWCORNER,
                  BOTTOMFRAME, STANDARD, TOPFRAME,
                  SECORNER, RIGHTFRAME, NECORNER };
⟨extended cartesian kernel members⟩
const char* output_identifier() const { return "Extended_cartesian"; }
};
```

We offer construction from a standard affine kernel objects.

⟨*extended cartesian kernel members*⟩≡

```
Point_2 epoint(const Standard_FT& m1, const Standard_FT& n1,
               const Standard_FT& m2, const Standard_FT& n2) const
{ return Point_2(FT(n1,m1),FT(n2,m2)); }
public:
Point_2 construct_point(const Standard_point_2& p) const
{ return Point_2(p.x(), p.y()); }
Point_2 construct_point(const Standard_line_2& l, Point_type& t) const
{
  t = (Point_type)Line_to_epoint<Standard_kernel>::determine_type(l);
  Point_2 res;
  switch (t) {
    case SWCORNER:   res = epoint(-1, 0, -1, 0); break;
    case NWCORNER:   res = epoint(-1, 0,  1, 0); break;
    case SECORNER:   res = epoint( 1, 0, -1, 0); break;
    case NECORNER:   res = epoint( 1, 0,  1, 0); break;
    case LEFTFRAME:
      res = epoint(-1, 0,  l.a()/l.b(), -l.c()/l.b()); break;
    case RIGHTFRAME:
      res = epoint( 1, 0, -l.a()/l.b(), -l.c()/l.b()); break;
    case BOTTOMFRAME:
      res = epoint( l.b()/l.a(), -l.c()/l.a(), -1, 0); break;
    case TOPFRAME:
      res = epoint(-l.b()/l.a(), -l.c()/l.a(),  1, 0); break;
    default: CGAL_assertion_msg(0,"EPoint type not correct!");
```

```
  }
  return res;
}
Point_2 construct_point(const Standard_point_2& p1,
                        const Standard_point_2& p2,
                        Point_type& t) const
{ return construct_point(Standard_line_2(p1,p2),t); }
Point_2 construct_point(const Standard_line_2& l) const
{ Point_type dummy; return construct_point(l,dummy); }
Point_2 construct_point(const Standard_point_2& p1,
                        const Standard_point_2& p2) const
{ return construct_point(Standard_line_2(p1,p2)); }
Point_2 construct_point(const Standard_point_2& p,
                        const Standard_direction_2& d) const
{ return construct_point(Standard_line_2(p,d)); }
Point_2 construct_opposite_point(const Standard_line_2& l) const
{ Point_type dummy; return construct_point(l.opposite(),dummy); }
Point_type type(const Point_2& p) const
{
  CGAL_assertion(p.x().degree()>=0 && p.y().degree()>=0 );
  if ( p.x().degree() == 0 && p.y().degree() == 0)
    return STANDARD;
  // now we are on the square frame
  FT rx = p.x();
  FT ry = p.y();
  int sx = sign(rx);
  int sy = sign(ry);
  if (sx < 0) rx = -rx;
  if (sy < 0) ry = -ry;
  if (rx>ry) {
    if (sx > 0) return RIGHTFRAME;
    else        return LEFTFRAME;
  }
  if (rx<ry) {
    if (sy > 0) return TOPFRAME;
    else        return BOTTOMFRAME;
  }
  // now (rx == ry)
  if (sx==sy) {
    if (sx < 0) return SWCORNER;
    else        return NECORNER;
  } else { CGAL_assertion(sx==-sy);
    if (sx < 0) return NWCORNER;
    else        return SECORNER;
  }
}
bool is_standard(const Point_2& p) const
{ return (type(p)==STANDARD);  }
Standard_point_2 standard_point(const Point_2& p) const
{ CGAL_assertion( type(p)==STANDARD );
  return Standard_point_2(p.x()[0],p.y()[0]);
```

```
}
Standard_line_2 standard_line(const Point_2& p) const
{ CGAL_assertion( type(p)!=STANDARD );
  FT x = p.x(), y = p.y();
  Standard_FT dx = x.degree()>0 ? x[1] : Standard_FT(0);
  Standard_FT dy = y.degree()>0 ? y[1] : Standard_FT(0);
  Standard_point_2 p0(x[0],y[0]);
  Standard_point_2 p1(x[0]+dx,y[0]+dy);
  return Standard_line_2(p0,p1);
}
Standard_ray_2 standard_ray(const Point_2& p) const
{ Standard_line_2 l = standard_line(p);
  Standard_direction_2 d = l.direction();
  Standard_point_2 q = l.point(0);
  return Standard_ray_2(q,d);
}
Point_2 NE() const { return construct_point(Standard_line_2(-1, 1,0)); }
Point_2 SE() const { return construct_point(Standard_line_2( 1, 1,0)); }
Point_2 NW() const { return construct_point(Standard_line_2(-1,-1,0)); }
Point_2 SW() const { return construct_point(Standard_line_2( 1,-1,0)); }
Line_2 upper() const { return construct_line(NW(),NE()); }
Line_2 lower() const { return construct_line(SW(),SE()); }
Line_2 left()  const { return construct_line(SW(),NW()); }
Line_2 right() const { return construct_line(SE(),NE()); }

Point_2 source(const Segment_2& s) const
{ typename Base::Construct_source_point_2 _source =
    construct_source_point_2_object();
  return _source(s); }
Point_2 target(const Segment_2& s) const
{ typename Base::Construct_target_point_2 _target =
    construct_target_point_2_object();
  return _target(s); }
Segment_2 construct_segment(const Point_2& p, const Point_2& q) const
{ typename Base::Construct_segment_2 _segment =
    construct_segment_2_object();
  return _segment(p,q); }
Line_2 construct_line(const Standard_line_2& l)  const
{ return Line_2(l.a(),l.b(),l.c()); }
Line_2 construct_line(const Point_2& p1, const Point_2& p2) const
{ Line_2 l(p1,p2);
  RT a=l.a(), b=l.b(), c=l.c();
  l =  Line_2(a,b,c);
  return l;
}
int orientation(const Segment_2& s, const Point_2& p) const
{ typename Base::Orientation_2 _orientation =
    orientation_2_object();
```

```
  return static_cast<int> ( _orientation(source(s),target(s),p) );
}
int orientation(const Point_2& p1, const Point_2& p2, const Point_2& p3)
const
{ typename Base::Orientation_2 _orientation =
    orientation_2_object();
  return static_cast<int> ( _orientation(p1,p2,p3) );
}
bool leftturn(const Point_2& p1, const Point_2& p2, const Point_2& p3)
const
{ return orientation(p1,p2,p3) > 0; }
bool is_degenerate(const Segment_2& s) const
{ typename Base::Is_degenerate_2 _is_degenerate =
    is_degenerate_2_object();
  return _is_degenerate(s); }
int compare_xy(const Point_2& p1, const Point_2& p2) const
{ typename Base::Compare_xy_2 _compare_xy =
    compare_xy_2_object();
  return static_cast<int>( _compare_xy(p1,p2) );
}
int compare_x(const Point_2& p1, const Point_2& p2) const
{ typename Base::Compare_x_2 _compare_x =
    compare_x_2_object();
  return static_cast<int>( _compare_x(p1,p2) );
}
int compare_y(const Point_2& p1, const Point_2& p2) const
{ typename Base::Compare_y_2 _compare_y =
    compare_y_2_object();
  return static_cast<int>( _compare_y(p1,p2) );
}

Point_2 intersection(
  const Segment_2& s1, const Segment_2& s2) const
{ typename Base::Intersect_2 _intersect =
    intersect_2_object();
  typename Base::Construct_line_2 _line =
    construct_line_2_object();
  Point_2 p;
  CGAL::Object result =
    _intersect(_line(s1),_line(s2));
  if ( !CGAL::assign(p, result) )
  CGAL_assertion_msg(false,"intersection: no intersection.");
  return p;
}
Direction_2 construct_direction(
  const Point_2& p1, const Point_2& p2) const
{ typename Base::Construct_direction_of_line_2 _direction =
    construct_direction_of_line_2_object();
  return _direction(construct_line(p1,p2)); }
bool strictly_ordered_ccw(const Direction_2& d1,
  const Direction_2& d2, const Direction_2& d3) const
```

```
{
  if ( d1 < d2 )  return ( d2 < d3 )||( d3 <= d1 );
  if ( d1 > d2 )  return ( d2 < d3 )&&( d3 <= d1 );
  return false;
}
bool contains(const Segment_2& s, const Point_2& p) const
{ typename Base::Has_on_2 _contains = has_on_2_object();
  return _contains(s,p);
}
bool strictly_ordered_along_line(
  const Point_2& p1, const Point_2& p2, const Point_2& p3) const
{ typename Base::Are_strictly_ordered_along_line_2 _ordered =
    are_strictly_ordered_along_line_2_object();
  return _ordered(p1,p2,p3);
}
bool first_pair_closer_than_second(
  const Point_2& p1, const Point_2& p2,
  const Point_2& p3, const Point_2& p4) const
{ return ( squared_distance(p1,p2) < squared_distance(p3,p4) ); }
template <class Forward_iterator>
void determine_frame_radius(Forward_iterator start, Forward_iterator end,
                            Standard_RT& R0) const
{ Standard_RT R;
  while ( start != end ) {
    Point_2 p = *start++;
    if ( is_standard(p) ) {
      R = max(abs(p.x()[0]), abs(p.y()[0]));
    } else {
      RT rx = abs(p.x()), ry = abs(p.y());
      if ( rx[1] > ry[1] )      R = abs(ry[0]-rx[0])/(rx[1]-ry[1]);
      else if ( rx[1] < ry[1] ) R = abs(rx[0]-ry[0])/(ry[1]-rx[1]);
      else /* rx[1] == ry[1] */ R = abs(rx[0]-ry[0])/2;
    }
    R0 = max(R+1,R0);
  }
}
```

The file wrapper is here.

⟨*Extended_cartesian.h*⟩≡

```
⟨CGAL EC Header⟩
#ifndef CGAL_EXTENDED_CARTESIAN_H
#define CGAL_EXTENDED_CARTESIAN_H
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Line_2_Line_2_intersection.h>
#ifndef _MSC_VER
#include <CGAL/RPolynomial.h>
#else
#include <CGAL/RPolynomial_MSC.h>
#define RPolynomial RPolynomial_MSC
```

```
#endif
#undef _DEBUG
#define _DEBUG 51
#include <CGAL/Nef_2/debug.h>
#include <CGAL/Nef_2/Line_to_epoint.h>
CGAL_BEGIN_NAMESPACE
template <class T> class Extended_cartesian;
⟨extended cartesian⟩
#undef RPolynomial
CGAL_END_NAMESPACE
#endif // CGAL_EXTENDED_CARTESIAN_H
```

7.4 A Test of Extended Points

⟨*EPoint-test.C*⟩≡

```
#define RPOLYNOMIAL_EXPLICIT_OUTPUT
#include <CGAL/Cartesian.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Extended_cartesian.h>
#include <CGAL/Filtered_extended_homogeneous.h>
#include <CGAL/test_macros.h>
#ifdef CGAL_USE_LEDA
#include <CGAL/leda_integer.h>
typedef leda_integer Integer;
template <>
struct ring_or_field<leda_integer> {
  typedef ring_with_gcd kind;
  typedef leda_integer RT;
  static RT gcd(const RT& r1, const RT& r2)
  { return ::gcd(r1,r2); }
};
#include <CGAL/leda_real.h>
typedef leda_real Real;
template <>
struct ring_or_field<leda_real> {
  typedef field_with_div kind;
};
#else
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
#include <CGAL/Quotient.h>
typedef CGAL::Gmpz Integer;
template <>
struct ring_or_field<CGAL::Gmpz> {
  typedef ring_with_gcd kind;
  typedef CGAL::Gmpz RT;
  static RT gcd(const RT& r1, const RT& r2)
  { return CGAL::gcd(r1,r2); }
};
```

```
typedef CGAL::Quotient<Integer> Real;
template <>
struct ring_or_field<Real> {
  typedef field_with_div kind;
};
#else
typedef long   Integer;
typedef double Real;
#endif
#endif
#include <CGAL/intersection_2.h>
using namespace CGAL;
int main()
{
  SETDTHREAD(41);
  CGAL_TEST_START;
{
  typedef CGAL::Extended_homogeneous<Integer> EDec;
  ⟨EDec test body⟩
  ⟨IO test⟩
}
{
  typedef CGAL::Extended_cartesian<Real> EDec;
  ⟨EDec test body⟩
  //IO does not work for LEDA reals
}
{
  typedef CGAL::Filtered_extended_homogeneous<Integer> EDec;
  ⟨EDec test body⟩
  ⟨IO test⟩
  D.print_statistics();
}
  CGAL_TEST_END;
}
```

⟨*EDec test body*⟩≡

```
typedef EDec::Point_2     EP;
typedef EDec::Segment_2   ES;
typedef EDec::Direction_2 ED;
typedef EDec::Standard_kernel::Point_2  Point;
typedef EDec::Standard_kernel::Line_2   Line;
typedef EDec::Standard_RT RT;
EDec D;
CGAL::set_pretty_mode ( std::cerr );
Point ps1(0,0), ps2(1,1), ps3(1,0), ps4(0,1), ps5(1,1,2);
EDec::Point_type t1,t2,t3;
EP eps1 = D.construct_point(ps1);
EP eps2 = D.construct_point(ps2);
EP eps3 = D.construct_point(ps3);
EP eps4 = D.construct_point(ps4);
EP eps5 = D.construct_point(ps5);
```

```
EP epn1 = D.construct_point(ps4,ps1,t1); // vertical down ray
EP epn2 = D.construct_point(ps1,ps4,t2); // vertical up ray
EP epn3 = D.construct_point(ps1,ps3,t3); // horizontal right ray
EP epn4 = D.construct_point(Line(2,3,4));
ES el1 =  D.construct_segment(D.construct_point(Line(ps1,ps4)),
                              D.construct_point(Line(ps4,ps1)));
ES el2 =  D.construct_segment(D.NW(),D.NE());
CGAL_TEST(D.type(D.SW())==EDec::SWCORNER);
CGAL_TEST(D.type(D.NW())==EDec::NWCORNER);
CGAL_TEST(D.type(D.SE())==EDec::SECORNER);
CGAL_TEST(D.type(D.NE())==EDec::NECORNER);
CGAL_TEST(D.type(epn1)==EDec::BOTTOMFRAME);
CGAL_TEST(D.type(epn2)==EDec::TOPFRAME);
CGAL_TEST(D.type(epn3)==EDec::RIGHTFRAME);
CGAL_TEST(D.type(eps1)==EDec::STANDARD);
CGAL_TEST(D.standard_line(epn1) == Line(ps4,ps1));
CGAL_TEST(D.standard_point(eps1) == ps1);
ES es1 = D.construct_segment(epn1,epn2);
ES es2 = D.construct_segment(eps1,eps5);
CGAL_TEST(D.source(es1) == epn1);
CGAL_TEST(D.target(es1) == epn2);
CGAL_TEST(D.orientation(es1,D.construct_point(Point(-1,-2))) > 0 );
CGAL_TEST(D.is_degenerate(D.construct_segment(epn1,epn1)));
CGAL_TEST(D.compare_xy(eps1,eps5)<0);
CGAL_TEST(D.compare_xy(eps1,epn2)<0);
CGAL_TEST(D.compare_xy(epn1,eps1)<0);
CGAL_TEST(D.intersection(es1,es2) == eps1);
CGAL_TEST(D.compare_xy(eps1,eps2)<0);
CGAL_TEST(D.compare_xy(eps4,eps1)>0);
CGAL_TEST(D.compare_xy(eps1,eps1)==0);
CGAL_TEST(D.compare_xy(D.NW(),eps2)<0);
CGAL_TEST(D.compare_xy(eps1,D.NE())<0);
CGAL_TEST(D.compare_xy(D.SW(),D.SE())<0);
CGAL_TEST(D.compare_xy(epn1,eps1)<0);
CGAL_TEST(D.compare_xy(eps1,epn2)<0);
CGAL_TEST(D.orientation(eps1,eps2,eps3)<0);
CGAL_TEST(D.orientation(eps1,eps3,eps2)>0);
CGAL_TEST(D.orientation(eps1,eps2,D.construct_point(Point(2,2)))==0);
CGAL_TEST(D.orientation(eps1,eps2,D.construct_point(ps1,ps2))==0);
CGAL_TEST(D.orientation(eps1,eps2,epn3)<0);
CGAL_TEST(D.orientation(eps1,eps2,epn2)>0);
CGAL_TEST(D.orientation(D.NW(),D.NE(),eps1)<0);
CGAL_TEST(D.orientation(D.NE(),D.NW(),eps1)>0);
CGAL_TEST(D.orientation(D.SW(),D.NE(),eps1)==0);
CGAL_TEST(D.orientation(epn1,epn2,eps1)==0);
CGAL_TEST(D.orientation(epn1,epn2,eps4)==0);
CGAL_TEST(D.orientation(epn1,epn2,eps3)<0);
CGAL_TEST(D.orientation(epn2,epn1,eps3)>0);
CGAL_TEST(D.first_pair_closer_than_second(eps1,eps5,eps1,eps2));
CGAL_TEST(!D.first_pair_closer_than_second(eps1,eps3,eps1,eps4));
CGAL_TEST(D.first_pair_closer_than_second(eps1,eps3,eps2,
```

```
          D.construct_point(Point(2,2,1))));
CGAL_TEST(D.first_pair_closer_than_second(eps1,eps3,eps2,D.NW()));
CGAL_TEST(D.first_pair_closer_than_second(eps1,D.SE(),D.SW(),D.NE()));
CGAL_TEST(!D.first_pair_closer_than_second(eps1,D.SE(),eps5,D.NE()));
CGAL_TEST(D.first_pair_closer_than_second(eps5,D.NE(),eps1,D.SE()));
CGAL_TEST(!D.first_pair_closer_than_second(eps1,D.SE(),eps1,D.NE()));
CGAL_TEST(D.first_pair_closer_than_second(D.SE(),D.NE(),D.NE(),D.SW()));
CGAL_TEST(!D.first_pair_closer_than_second(D.SE(),D.NE(),D.NW(),D.SW()));
CGAL_TEST(D.construct_direction(D.NW(),D.NE())==ED(RT(1),RT(0)));
CGAL_TEST(D.construct_direction(D.NE(),D.NW())==ED(RT(-1),RT(0)));
CGAL_TEST(D.construct_direction(D.SW(),D.NE())==ED(RT(1),RT(1)));
CGAL_TEST(D.construct_direction(D.NW(),D.SE())==ED(RT(1),RT(-1)));
CGAL_TEST(D.construct_direction(D.NW(),D.SW())==ED(RT(0),RT(-1)));
CGAL_TEST(D.construct_direction(D.SW(),D.NW())==ED(RT(0),RT(1)));
CGAL_TEST(D.construct_direction(eps5,D.NE())==ED(RT(1),RT(1)));
CGAL_TEST(D.construct_direction(eps5,D.SW())==ED(RT(-1),RT(-1)));
ES upper = D.construct_segment(D.NW(),D.NE());
ES left  = D.construct_segment(D.SW(),D.NW());
EP ep_res = D.intersection(el1,upper);
CGAL_TEST(ep_res==epn2);
ep_res = D.intersection(left, upper);
CGAL_TEST(ep_res == D.NW());
```

⟨IO test⟩≡

```
CGAL_IO_TEST(eps3,eps1);
CGAL_IO_TEST(epn2,epn1);
```

7.5 A Demo of the EPoint concept

⟨EPoint-demo.C⟩≡

```
#define RPOLYNOMIAL_EXPLICIT_OUTPUT
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/leda_integer.h>
#include <CGAL/test_macros.h>
template <>
struct ring_or_field<leda_integer> {
  typedef ring_with_gcd kind;
};
typedef CGAL::Extended_homogeneous<leda_integer> EDec;
typedef EDec::Standard_point_2 Standard_point;
typedef EDec::Standard_line_2  Standard_line;
typedef EDec::Point_2  Point;
typedef EDec::Standard_aff_transformation_2 Atra;
EDec D;
int main() {
  CGAL::set_pretty_mode ( std::cout );
  CGAL::set_pretty_mode ( std::cerr );
  Standard_point p1(0,0),p2(2,1),p3(1,4);
```

```
  Standard_line l(p1,p2);
  Point ep1 = D.construct_point(l);
  Point ep2 = D.construct_point(p2,p3);
  Point ep3 = D.construct_point(p2);
  Atra T(CGAL::ROTATION,1,0);
  Point ep10 = D.transform(ep1,T);
  Point ep20 = D.transform(ep2,T);
  Point ep30 = D.transform(ep3,T);
}
```

Bibliography

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry : An Introduction*. Springer, 1985.

[SM00] M. Seel and K. Mehlhorn. Infimaximal Frames: a framework to make lines look like segments. Technical Report MPI-I-2000-1-005, Max-Planck-Institut für Informatik, Saarbrücken, 2000.

8 Filtered Extended Kernel

8.1 Introduction

It is convenient to extend the vision of standard rational points by so-called *non-standard* points. We have one non-standard point for each equivalence class of rays. Two rays are called equivalent if one is contained in the other. The geometric properties of non-standard points are derived by giving them a geometric interpretation by means of an infinimaximal R. R is a real variable. The value of R is finite but larger than the value of any concrete real number. Let F be the square box with corners $NW(-R,R)$, $NE(R,R)$, $SE(R,-R)$, and $SW(-R,-R)$. Let p be a non-standard point and let r be a ray defining it. If the frame F contains the source point of r then let $p(R)$ be the intersection of r with the frame F, if F does not contain the source of r then $p(R)$ is undefined. For a standard point let $p(R)$ be equal to p if p is contained in the frame F and let $p(R)$ be undefined otherwise. Clearly, for any standard or non-standard point $p(R)$ is defined for any sufficiently large R.

Let f be any function on standard points, say with k arguments. We call f *extensible* if for any k points $p_1, \ldots, p_k$ the function value $f(p_1(R), \ldots, p_k(R))$ is constant for all sufficiently large R and has the same value as the function evaluated at a fixed large enough value R_0. We also consider geometric constructions. Let g be a construction on standard points constructing a tuple of l points from a tuple of k points. We call g *extensible* if for any k points $p_1, \ldots, p_k$ the construction is closed in our set of extended points: if it constructs a point tuple $q_1, \ldots, q_l$ of extended points for all sufficiently large R with the property that by fixing a large enough R_0 the tuple is the result of the standard construction. As we will see in a moment the predicates *lexicographic order* of points, *orientation*, and *side_of_circle* are extensible. Also the calculation of the point in the *intersection* of line segments defined on two pairs of points is extensible.

For a formal definition of extended points, extended segments and the corresponding predicates see the technical report [SM00].

8.2 Implementation

In this section we descibe a more advanced extended kernel than *Simple_extended_kernel*. This kernel tries to optimize runtime by use of a filter stage. It does not rely on polynomial arithmetic but on unrolled polynomial expressions directly programmed for the evaluation of the predicates and constructions. We explain the techniques used. We implement specialized kernel types that store the arbitrary precision coordinates but also intervals of number type double approximating them. Then we show how our running example, the orientation test, is implemented in its unrolled fashion. We finally present the intersection operation based on a case-dependent implementation. Our description covers all ideas needed to implement the whole extended kernel concept.

Kernel types

In contrast to the first approach based on a polynomial arithmetic type *RPolynomial* plugged into CGAL homogeneous points, we implement an extended kernel based on specialized types. The components are the types listed below plus the predicates and constructions that are required to construct a model of our *ExtendedKernelTraits_2* concept. The types that we implement are

```
SPolynomial<RT>
SQuotient<RT>
Extended_point<RT>
Extended_segment<RT>
Extended_direction<RT>
```

We shortly elaborate on the usage of the above types and their design. Let *RT* be a multiprecision integer number type like LEDA *integer*. *SPolynomial*<*RT*> is a container type storing linear polynomials of the form $mR+n$. *SQuotient*<*RT*> stores a two tuple consisting of an *SPolynomial*<*RT*> and an *RT* object and represents the corresponding quotient.

Our definition tells us that an extended point p in homogeneous representation has the form (m_xR+n_x, m_yR+n_y, w), where R is our frame defining variable and all other identifiers are numbers from RT. $p.hx(\,)$ returns the x-polynomial m_xR+n_x and $p.hy(\,)$ the y-polynomial m_yR+n_y (of type *SPolynomial*<*RT*>). $p.hw(\,)$ returns w. These are the homogeneous x- and y- coordinates of p with common denominator w. For completeness p also provides a Cartesian interface $p.x(\,)$ returning an *SQuotient*<*RT*> of the form $(m_xR+n_x)/w$. In analogy $p.y(\,) = (m_yR+n_y)/w$.

All number entries of p can be accessed as multiprecision numbers as well as as double approximations stored in an interval of type *CGAL::Interval_nt_advanced*. Thus a point stores 5 *RT* entries and 10 double precision entries. The number type interface consists of the operations $p.mx(\,)$, $p.nx(\,)$, $p.my(\,)$, $p.ny(\,)$, $p.hw(\,)$, and $p.mxD(\,)$, $p.nxD(\,)$, $p.myD(\,)$, $p.nyD(\,)$, $p.hwD(\,)$ for the intervals. The operation $p.is\_standard(\,)$ returns true, iff both m_x and m_y are zero.

The points are programmed along the lines of the LEDA and CGAL geometric kernel design. They have I/O stream operators, and they can be drawn in a LEDA window, when our frame parameter R is fixed.

⟨*simple polynomials*⟩≡

```
template <typename RT>
class SPolynomial {
  RT _m,_n;
public:
  SPolynomial() : _m(),_n() {}
  SPolynomial(const RT& m, const RT& n) : _m(m),_n(n) {}
  SPolynomial(const RT& n) : _m(),_n(n) {}
  SPolynomial(const SPolynomial<RT>& p) : _m(p._m),_n(p._n) {}
  SPolynomial<RT>& operator=(const SPolynomial<RT>& p)
  { _m=p._m; _n=p._n; return *this; }

  const RT& m() const { return _m; }
  const RT& n() const { return _n; }
  void negate() { _m=-_m; _n=-_n; }

  SPolynomial<RT> operator*(const RT& c) const
  { return SPolynomial<RT>(c*_m,c*_n); }
  SPolynomial<RT> operator+(const SPolynomial<RT>& p) const
  { return SPolynomial<RT>(_m+p._m,_n+p._n); }
  SPolynomial<RT> operator-(const SPolynomial<RT>& p) const
```

```
  { return SPolynomial<RT>(_m-p._m,_n-p._n); }
  SPolynomial<RT> operator-() const
  { return SPolynomial<RT>(-_m,-_n); }
  void operator /= (const RT& c)
  { _m /= c; _n /= c; }
  const RT& operator[](int i) { return (i%2 ? _n : _m); }
  const RT& operator[](int i) const { return (i%2 ? _n : _m); }
  bool is_zero() const { return (_m==0 && _n==0); }
  int sign() const
  { if ( _m != 0 ) return CGAL_NTS sign(_m);
    return CGAL_NTS sign(_n); }
  // only for visualization:
  static void set_R(const RT& R) { _R = R; }
  RT eval_at(const RT& r) const { return _m*r+_n; }
  RT eval_at_R() const { return _m*_R+_n; }
protected:
  static RT _R;
};
template <class RT> RT SPolynomial<RT>::_R;
template <typename RT>
int sign(const SPolynomial<RT>& p)
{ return p.sign(); }
template <typename RT>
bool operator==(const SPolynomial<RT>& p1, const SPolynomial<RT>& p2)
{ return (p1-p2).is_zero(); }
template <typename RT>
bool operator>(const SPolynomial<RT>& p1, const SPolynomial<RT>& p2)
{ return (p1-p2).sign()>0; }
template <typename RT>
bool operator<(const SPolynomial<RT>& p1, const SPolynomial<RT>& p2)
{ return (p1-p2).sign()<0; }
template <class RT>
inline double to_double(const SPolynomial<RT>& p)
{ return (CGAL::to_double(p.eval_at(SPolynomial<RT>::_R))); }
template <class RT>
std::ostream& operator<<(std::ostream& os, const SPolynomial<RT>& p)
{
  switch( os.iword(CGAL::IO::mode) ) {
    case CGAL::IO::ASCII :
      os << p.m() << " " << p.n(); break;
    case CGAL::IO::BINARY :
      CGAL::write(os,p.m());CGAL::write(os,p.n()); break;
    default:
      if ( p.m() == 0 ) os<<"["<<p.n()<<"]";
      else os<<"["<<p.m()<<" R + "<<p.n()<<"]";
  }
  return os;
}
template <class RT>
std::istream& operator>>(std::istream& is, SPolynomial<RT>& p)
{ RT m,n;
```

```
  switch( is.iword(CGAL::IO::mode) ){
    case CGAL::IO::ASCII :
      is >> m >> n; p = SPolynomial<RT>(m,n); break;
    case CGAL::IO::BINARY :
      CGAL::read(is,m);CGAL::read(is,n);break;
    default:
    CGAL_assertion_msg(0,"\nStream must be in ascii or binary mode\n");
      break;
  }
  return is;
}
template <class RT> /*CGAL_KERNEL_INLINE*/
CGAL::io_Operator io_tag(const SPolynomial<RT>&)
{ return CGAL::io_Operator(); }
```

We need a container quotient type to return cartesian coordinates. We only need it for visualization and interface completion. It has no number type functionality.

⟨*simple polynomials*⟩+≡

```
template <typename RT>
class SQuotient {
  SPolynomial<RT> _p;
  RT              _n;
public:
  SQuotient() : _p(),_n() {}
  SQuotient(const SPolynomial<RT>& p, const RT& n) : _p(p),_n(n) {}
  SQuotient(const SQuotient<RT>& p) : _p(p._p),_n(p._n) {}
  SQuotient<RT>& operator=(const SQuotient<RT>& p)
  { _p=p._p; _n=p._n; return *this; }
  const SPolynomial<RT>& numerator() const { return _p; }
  const RT&              denominator() const { return _n; }
};
template <class RT>
inline double to_double(const SQuotient<RT>& q)
{ return (CGAL::to_double(q.numerator().eval_at_R())/
          CGAL::to_double(q.denominator())); }
```

Points are realized by a smart-pointer scheme. There is a backend object type *Extended_point_rep<RT>* (the representation) and a frontend handle type *Extended_point<RT>*. We only elaborate on the representation type. Details on smart pointers are offered in the LEDA book [MN99].

⟨*extended points*⟩≡

```
template <typename RT> class Extended_point;
template <typename RT> class Extended_point_rep;
template <typename RT>
class Extended_point_rep : public Ref_counted {
  friend class Extended_point<RT>;
  SPolynomial<RT> x_,y_; RT w_;
```

```
  typedef Interval_nt_advanced DT;
  DT mxd,myd,nxd,nyd,wd;
public:
  ⟨construction⟩
};
```

For the filter stage we use interval approximations of type *CGAL*:: *Interval_nt_advanced*. See H. Brönnimann et.al [BBP98] for more information. An object of this type is an interval of two doubles representing any number in the interval. An arithmetic operation *op* of $+$, $-$, $*$, $/$ on two intervals X and Y calculates an interval X op Y such that $\forall x \in X, y \in Y : (x \text{ op } y) \in (X \text{ op } Y)$. This allows us to determine the correct sign of an interval expression as long as the interval does not contain zero. The type uses exceptions to tell user code that a sign determination does not lead to a secure result. That exception can be catched to repair the resulting uncertainty. We will see how this works in our predicates below. The type *Interval_nt_advanced* implements dynamic filtering. Rounding errors are accumulated during the execution of the program. The type requires its user to take the responsibility for the rounding mode of the processor. Whenever an arithmetic interval expression is evaluated the processor should be in its correct rounding mode (switching the processor is an expensive operation, thereby the user's care does pay-off). The switching is done by class declaration statement *Protect_FPU_rounding*<*true*> *P*. The construction of object *P* sets the correct rounding mode, its destruction resets the previous mode which ensures correct execution of code parts that rely on different rounding modes.

The following conversion routine constructs an interval that contains a *double* approximation *cn* from its parameter *n* (a LEDA *integer*). Only two cases can occur: *n* can be approximated accurately by an interval $[cn, cn]$ of zero width if the bit representation of *n* has less than 53 bits. Otherwise we add the smallest representable *double* to make the interval contain *n*. By the addition the interval is expanded by exactly the radius of the machine accuracy. For more information on rounding problems please refer to D. Goldberg [Gol91].

```
DT to_interval(const leda_integer& n)
{ double cn = CGAL::to_double(n);
  leda_integer pn = ( n>0 ? n : -n);
  if ( pn.iszero() || log(pn) < 53 ) return DT(cn);
  else {
    Protect_FPU_rounding<true> P;
    return DT(cn)+CGAL::Interval_base::Smallest;
  }
}
```

On construction of the representation we construct the *double* approximation of the multiprecision entries. Note that we trade space for execution time.

⟨*construction*⟩≡

```
Extended_point_rep(const RT& x, const RT& y, const RT& w) :
  Ref_counted(), x_(x),y_(y),w_(w)
{ CGAL_assertion_msg(w!=0,"denominator is zero.");
  nxd=CGAL::to_interval(x);
  nyd=CGAL::to_interval(y);
  wd=CGAL::to_interval(w);
  mxd=myd=0;
}
Extended_point_rep(const SPolynomial<RT>& x,
```

```
                    const SPolynomial<RT>& y,
                    const RT& w) : Ref_counted(), x_(x),y_(y),w_(w)
{ CGAL_assertion_msg(w!=0,"denominator is zero.");
  mxd=CGAL::to_interval(x.m());
  myd=CGAL::to_interval(y.m());
  nxd=CGAL::to_interval(x.n());
  nyd=CGAL::to_interval(y.n());
  wd=CGAL::to_interval(w);
}
```

We do not show the implementation of the class *Extended_point<RT>*. It mainly serves as an interface of the representation class and inherits the handle maintainance code from the front end class *CGAL::Handle_for*.

⟨*construction*⟩+≡

```
Extended_point_rep() : Ref_counted(), x_(),y_(),w_() {}
~Extended_point_rep() {}
void negate()
{ x_ = -x_; y_ = -y_; w_ = -w_;
  mxd = -mxd; myd = -myd; nxd = -nxd; nyd = -nyd; wd = -wd; }
```

⟨*extended points*⟩+≡

```
template <typename RT_>
class Extended_point : public Handle_for< Extended_point_rep<RT_> > {
  typedef Extended_point_rep<RT_> Rep;
  typedef Handle_for< Rep >       Base;
public:
  typedef typename Rep::DT DT;
  typedef RT_ RT;
  Extended_point() : Base( Rep() ) {}
  Extended_point(const RT& x, const RT& y, const RT& w) :
    Base( Rep(x,y,w) )
  { if (w < 0) ptr->negate(); }
  Extended_point(const SPolynomial<RT>& x,
                 const SPolynomial<RT>& y,
                 const RT& w) : Base( Rep(x,y,w) )
  { if (w < 0) ptr->negate(); }
  Extended_point(const RT& mx, const RT& nx,
                 const RT& my, const RT& ny, const RT& w) :
    Base( Rep(SPolynomial<RT>(mx,nx), SPolynomial<RT>(my,ny), w) )
  { if (w < 0) ptr->negate(); }
  Extended_point(const Extended_point<RT>& p) : Base(p) {}
  ~Extended_point() {}
  Extended_point& operator=(const Extended_point<RT>& p)
  { Base::operator=(p); return *this; }
  const RT& mx() const { return ptr->x_.m(); }
  const RT& nx() const { return ptr->x_.n(); }
  const RT& my() const { return ptr->y_.m(); }
  const RT& ny() const { return ptr->y_.n(); }
  const RT& hw()  const { return ptr->w_; }
```

```
  const DT& mxD() const { return ptr->mxd; }
  const DT& nxD() const { return ptr->nxd; }
  const DT& myD() const { return ptr->myd; }
  const DT& nyD() const { return ptr->nyd; }
  const DT& hwD() const { return ptr->wd; }
  SQuotient<RT> x() const
  { return SQuotient<RT>(ptr->x_, ptr->w_); }
  SQuotient<RT> y() const
  { return SQuotient<RT>(ptr->y_, ptr->w_); }
  const SPolynomial<RT> hx() const { return ptr->x_; }
  const SPolynomial<RT> hy() const { return ptr->y_; }
  bool is_standard() const { return (mx()==0)&&(my()==0); }
  Extended_point<RT> opposite() const
  { return Extended_point<RT>(-mx(),nx(),-my(),ny(),w()); }
⟨point check ops⟩
};
template <class RT>
std::ostream& operator<<(std::ostream& os, const Extended_point<RT>& p)
{ switch( os.iword(CGAL::IO::mode) ) {
    case CGAL::IO::ASCII :
      os << p.hx() << " " << p.hy() << " " << p.hw(); break;
    case CGAL::IO::BINARY :
      CGAL::write(os,p.hx());CGAL::write(os,p.hy());
      CGAL::write(os,p.hw()); break;
    default:
      os << "(" << p.hx() << "," << p.hy() << "," << p.hw() << ")";
      os << "((" << p.nx().to_double()/p.hw().to_double() << ","
         << p.ny().to_double()/p.hw().to_double() << "))";
  }
  return os;
}
template <class RT>
std::istream& operator>>(std::istream& is, Extended_point<RT>& p)
{ SPolynomial<RT> x,y; RT w;
  switch( is.iword(CGAL::IO::mode) ){
    case CGAL::IO::ASCII :
      is >> x >> y >> w; break;
    case CGAL::IO::BINARY :
      CGAL::read(is,x);CGAL::read(is,y);CGAL::read(is,w); break;
    default:
    CGAL_assertion_msg(0,"\nStream must be in ascii or binary mode\n");
      break;
  }
  p = Extended_point<RT>(x,y,w);
  return is;
}
```

Predicates

We show the implementation of the orientation predicate of extended points. All other predicates follow the same strategy. For three homogeneous points in polynomials we derive the formula for the orientation determinant and build up a filter cascade. We implement three template functions that code the unrolled coefficients of the polynomial in R of degree 2. We do not prove the derivation of the following algebraic expressions, we instead explain how we obtained them. We used the math package Maple [CGG+91] to do the algebra. The following script executed in maple produces the code expressions below. The lines with the comments have to be executed for the corresponding indices.

```
> pxi := mxi*R+nxi // i=1,2,3
> pyi := myi*R+nyi // i=1,2,3
> M := array([[px1,py1,w1],[px2,py2,w2],[px3,py3,w3);]]
> orient := collect(det(M),R);
> coeffi := coeff(orient,R,i); // i=0,1,2
> C(coeffi); // i=0,1,2
```

Finally just paste the coefficient code into the template functions. The following operations code the coefficient of the squared, linear, and constant term of the function in R that is the result of the determinant evaluation of M.

⟨*orientation predicate*⟩≡

```
template <typename NT> inline
int orientation_coeff2(const NT& mx1, const NT& /*nx1*/,
                       const NT& my1, const NT& /*ny1*/, const NT& w1,
                       const NT& mx2, const NT& /*nx2*/,
                       const NT& my2, const NT& /*ny2*/, const NT& w2,
                       const NT& mx3, const NT& /*nx3*/,
                       const NT& my3, const NT& /*ny3*/, const NT& w3)
{
  NT coeff2 = mx1*w3*my2-mx1*w2*my3+mx3*w2*my1-
              mx2*w3*my1-mx3*w1*my2+mx2*w1*my3;
  return CGAL_NTS sign(coeff2);
}
template <typename NT> inline
int orientation_coeff1(const NT& mx1, const NT& nx1,
                       const NT& my1, const NT& ny1, const NT& w1,
                       const NT& mx2, const NT& nx2,
                       const NT& my2, const NT& ny2, const NT& w2,
                       const NT& mx3, const NT& nx3,
                       const NT& my3, const NT& ny3, const NT& w3)
{
  NT coeff1 = mx1*w3*ny2-mx1*w2*ny3+nx1*w3*my2-mx2*w3*ny1-
              nx1*w2*my3+mx2*w1*ny3-nx2*w3*my1+mx3*w2*ny1+
              nx2*w1*my3-mx3*w1*ny2+nx3*w2*my1-nx3*w1*my2;
  return CGAL_NTS sign(coeff1);
}
template <typename NT> inline
int orientation_coeff0(const NT& /*mx1*/, const NT& nx1,
                       const NT& /*my1*/, const NT& ny1, const NT& w1,
                       const NT& /*mx2*/, const NT& nx2,
```

```
                        const NT& /*my2*/, const NT& ny2, const NT& w2,
                        const NT& /*mx3*/, const NT& nx3,
                        const NT& /*my3*/, const NT& ny3, const NT& w3)
{
  NT coeff0 = -nx2*w3*ny1+nx1*w3*ny2+nx2*w1*ny3-
              nx1*w2*ny3+nx3*w2*ny1-nx3*w1*ny2;
  return CGAL_NTS sign(coeff0);
}
```

⟨*orientation predicate*⟩+≡

```
DEFCOUNTER(or0)
DEFCOUNTER(or1)
DEFCOUNTER(or2)
```

Now the final orientation predicate consists of three *try-catch* blocks. Each *try* block contains the filtered coefficient evaluation. If the sign evaluation is not defined (the resulting interval contains zero) then the *unsafe_comparison* exception is thrown. The *catch* block evaluates the expression with *RT* arithmetic. Note again the protection of the rounding mode with the *Protect_FPU_rounding*<*true*> class declaration. The macros *INCTOTAL*( ) and *INCEXCEPTION*( ) are used to accumulate the statistics of the filter stages.

⟨*orientation predicate*⟩+≡

```
template <typename RT>
int orientation(const Extended_point<RT>& p1,
                const Extended_point<RT>& p2,
                const Extended_point<RT>& p3)
{
  int res;
  try { INCTOTAL(or2); Protect_FPU_rounding<true> Protection;
    res = orientation_coeff2(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
                             p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                             p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD());
  }
  catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(or2);
    res = orientation_coeff2(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                             p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
                             p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw());
  }
  if ( res != 0 ) return res;
  try { INCTOTAL(or1); Protect_FPU_rounding<true> Protection;
    res = orientation_coeff1(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
                             p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                             p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD());
  }
  catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(or1);
    res = orientation_coeff1(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                             p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
                             p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw());
```

```
    }
    if ( res != 0 ) return res;
    try { INCTOTAL(or0); Protect_FPU_rounding<true> Protection;
      res = orientation_coeff0(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
                               p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                               p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD());
    }
    catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(or0);
      res = orientation_coeff0(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                               p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
                               p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw());
    }
    return res;
  }
```

Corresponding implementations are provided for the predicates *compare_x*( ), *compare_y*( ), *compare_xy*( ), and *compare_pair_dist*( ). The latter realizes the squared distance comparison of two pairs of points. The resulting polynomial has again degree 2 but contains more complicated expressions than the orientation predicate above.

⟨*comparison predicate*⟩≡

```
  template <typename NT>
  inline
  int compare_expr(const NT& n1, const NT& d1,
                   const NT& n2, const NT& d2)
  { return CGAL_NTS sign( n1*d2 - n2*d1 ); }
  DEFCOUNTER(cmpx0)
  DEFCOUNTER(cmpx1)
  template <typename RT>
  int compare_x(const Extended_point<RT>& p1,
                const Extended_point<RT>& p2)
  {
    int res;
    try { INCTOTAL(cmpx1); Protect_FPU_rounding<true> Protection;
      res = compare_expr(p1.mxD(),p1.hwD(),p2.mxD(),p2.hwD());
    }
    catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(cmpx1);
      res = compare_expr(p1.mx(),p1.hw(),p2.mx(),p2.hw());
    }
    if ( res != 0 ) return res;
    try { INCTOTAL(cmpx0); Protect_FPU_rounding<true> Protection;
      res = compare_expr(p1.nxD(),p1.hwD(),p2.nxD(),p2.hwD());
    }
    catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(cmpx0);
      res = compare_expr(p1.nx(),p1.hw(),p2.nx(),p2.hw());
    }
    return res;
  }
  DEFCOUNTER(cmpy0)
  DEFCOUNTER(cmpy1)
  template <typename RT>
```

```
int compare_y(const Extended_point<RT>& p1,
              const Extended_point<RT>& p2)
{
  int res;
  try { INCTOTAL(cmpy1); Protect_FPU_rounding<true> Protection;
    res = compare_expr(p1.myD(),p1.hwD(),p2.myD(),p2.hwD());
  }
  catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(cmpy1);
    res = compare_expr(p1.my(),p1.hw(),p2.my(),p2.hw());
  }
  if ( res != 0 ) return res;
  try { INCTOTAL(cmpy0); Protect_FPU_rounding<true> Protection;
    res = compare_expr(p1.nyD(),p1.hwD(),p2.nyD(),p2.hwD());
  }
  catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(cmpy0);
    res = compare_expr(p1.ny(),p1.hw(),p2.ny(),p2.hw());
  }
  return res;
}

template <typename RT>
inline
int compare_xy(const Extended_point<RT>& p1,
               const Extended_point<RT>& p2)
{ int c1 = compare_x(p1,p2);
  if ( c1 != 0 ) return c1;
  else return compare_y(p1,p2);
}

template <typename RT>
inline
bool strictly_ordered_along_line(const Extended_point<RT>& p1,
                                 const Extended_point<RT>& p2,
                                 const Extended_point<RT>& p3)
{ return ( orientation(p1,p2,p3) == 0 ) &&
         ( compare_xy(p1,p2) * compare_xy(p2,p3) == 1 );
}

template <typename RT>
inline bool operator==(const Extended_point<RT>& p1,
                       const Extended_point<RT>& p2)
{ CHECK(bool(compare_xy(p1,p2) == 0),p1.checkrep()==p2.checkrep())
  return (p1.identical(p2) || compare_xy(p1,p2) == 0); }

template <typename RT>
inline bool operator!=(const Extended_point<RT>& p1,
                       const Extended_point<RT>& p2)
{ return !(p1==p2); }

template <typename NT>
inline
int cmppd_coeff2(const NT& mx1, const NT& /*nx1*/,
                 const NT& my1, const NT& /*ny1*/, const NT& w1,
                 const NT& mx2, const NT& /*nx2*/,
                 const NT& my2, const NT& /*ny2*/, const NT& w2,
                 const NT& mx3, const NT& /*nx3*/,
```

```
                  const NT& my3, const NT& /*ny3*/, const NT& w3,
                  const NT& mx4, const NT& /*nx4*/,
                  const NT& my4, const NT& /*ny4*/, const NT& w4)
{
  NT w1Q(w1*w1), w2Q(w2*w2), w3Q(w3*w3), w4Q(w4*w4);
  NT w1w2Q(w1Q*w2Q), w3w4Q(w3Q*w4Q), two(2);
  NT coeff2 =     w3w4Q * w2Q *mx1*mx1-
                  two* w3w4Q  *w2*mx1*w1*mx2+
                  w3w4Q * w1Q *mx2*mx2+
                  w3w4Q * w2Q *my1*my1-
                  two* w3w4Q  *w2*my1*w1*my2+
                  w3w4Q * w1Q *my2*my2-
                  w1w2Q * w4Q *mx3*mx3+
                  two* w1w2Q  *w4*mx3*w3*mx4-
                  w1w2Q * w3Q *mx4*mx4-
                  w1w2Q * w4Q *my3*my3+
                  two* w1w2Q  *w4*my3*w3*my4-
                  w1w2Q * w3Q *my4*my4;
  return CGAL_NTS sign(coeff2);
}

template <typename NT>
inline
int cmppd_coeff1(const NT& mx1, const NT& nx1,
                  const NT& my1, const NT& ny1, const NT& w1,
                  const NT& mx2, const NT& nx2,
                  const NT& my2, const NT& ny2, const NT& w2,
                  const NT& mx3, const NT& nx3,
                  const NT& my3, const NT& ny3, const NT& w3,
                  const NT& mx4, const NT& nx4,
                  const NT& my4, const NT& ny4, const NT& w4)
{
  NT w1Q(w1*w1), w2Q(w2*w2), w3Q(w3*w3), w4Q(w4*w4);
  NT w1w2Q(w1Q*w2Q), w3w4Q(w3Q*w4Q), two(2);
  NT coeff1 = two * (w3w4Q * w1Q * mx2*nx2-
                      w3w4Q * w2*my1*w1*ny2+
                      w3w4Q * w1Q * my2*ny2+
                      w1w2Q * w4*nx3*w3*mx4-
                      w1w2Q * w4Q *mx3*nx3+
                      w3w4Q * w2Q *mx1*nx1-
                      w3w4Q * w2*mx1*w1*nx2-
                      w3w4Q * w2*nx1*w1*mx2-
                      w3w4Q * w2*ny1*w1*my2-
                      w1w2Q * w4Q *my3*ny3+
                      w1w2Q * w4*my3*w3*ny4+
                      w1w2Q * w4*ny3*w3*my4+
                      w3w4Q * w2Q *my1*ny1-
                      w1w2Q * w3Q *my4*ny4+
                      w1w2Q * w4*mx3*w3*nx4-
                      w1w2Q * w3Q *mx4*nx4);
  return CGAL_NTS sign(coeff1);
}

template <typename NT>
```

```
inline
int cmppd_coeff0(const NT& /*mx1*/, const NT& nx1,
                 const NT& /*my1*/, const NT& ny1, const NT& w1,
                 const NT& /*mx2*/, const NT& nx2,
                 const NT& /*my2*/, const NT& ny2, const NT& w2,
                 const NT& /*mx3*/, const NT& nx3,
                 const NT& /*my3*/, const NT& ny3, const NT& w3,
                 const NT& /*mx4*/, const NT& nx4,
                 const NT& /*my4*/, const NT& ny4, const NT& w4)
{
  NT w1Q(w1*w1), w2Q(w2*w2), w3Q(w3*w3), w4Q(w4*w4);
  NT w1w2Q(w1Q*w2Q), w3w4Q(w3Q*w4Q), two(2);
  NT coeff0 = w3w4Q * (w1Q * ( nx2*nx2 + ny2*ny2 ) +
                       w2Q * ( ny1*ny1 + nx1*nx1 )) -
              w1w2Q * (w4Q * ( nx3*nx3 + ny3*ny3 ) +
                       w3Q * ( nx4*nx4 + ny4*ny4 )) +
              two* (- w3w4Q * (w2*nx1*w1*nx2 + w2*ny1*w1*ny2)
                    + w1w2Q * (w4*ny3*w3*ny4 + w4*nx3*w3*nx4));
  return CGAL_NTS sign(coeff0);
}
DEFCOUNTER(cmppd2)
DEFCOUNTER(cmppd1)
DEFCOUNTER(cmppd0)
// leghth.mws
template <typename RT>
int compare_pair_dist(
  const Extended_point<RT>& p1, const Extended_point<RT>& p2,
  const Extended_point<RT>& p3, const Extended_point<RT>& p4)
{
  int res;
  try { INCTOTAL(cmppd2); Protect_FPU_rounding<true> Protection;
    res = cmppd_coeff2(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
                       p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                       p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD(),
                       p4.mxD(),p4.nxD(),p4.myD(),p4.nyD(),p4.hwD());
  }
  catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(cmppd2);
    res = cmppd_coeff2(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                       p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
                       p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw(),
                       p4.mx(),p4.nx(),p4.my(),p4.ny(),p4.hw());
  }
  if ( res != 0 ) return res;
  try { INCTOTAL(cmppd1); Protect_FPU_rounding<true> Protection;
    res = cmppd_coeff1(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
                       p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                       p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD(),
                       p4.mxD(),p4.nxD(),p4.myD(),p4.nyD(),p4.hwD());
  }
  catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(cmppd1);
    res = cmppd_coeff1(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                       p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
```

```
                     p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw(),
                     p4.mx(),p4.nx(),p4.my(),p4.ny(),p4.hw());
  }
  if ( res != 0 ) return res;
  try { INCTOTAL(cmppd0); Protect_FPU_rounding<true> Protection;
    res = cmppd_coeff0(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
                     p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                     p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD(),
                     p4.mxD(),p4.nxD(),p4.myD(),p4.nyD(),p4.hwD());
  }
  catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(cmppd0);
    res = cmppd_coeff0(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                     p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
                     p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw(),
                     p4.mx(),p4.nx(),p4.my(),p4.ny(),p4.hw());
  }
  return res;
}
```

Extended segments and intersection

We provide extended segments and some primitives. We mostly concentrate on the intersection predicate of lines supported by non-trivial extended segments. In this implementation we want to avoid the calculation of the polynomial *gcd*. Therefore, we extract the possible intersection configurations in advance and avoid higher degree polynomials resulting from the general algebraic term.

⟨*extended segments*⟩≡

```
template <typename RT>
class Extended_segment {
  Extended_point<RT> _p1,_p2;
public:
  Extended_segment() : _p1(),_p2() {}
  Extended_segment(const Extended_point<RT>& p1,
                   const Extended_point<RT>& p2) :
    _p1(p1), _p2(p2) {}
  Extended_segment(const Extended_segment<RT>& s) :
    _p1(s._p1), _p2(s._p2) {}
  Extended_segment<RT>& operator=(const Extended_segment<RT>& s)
  { _p1 = s._p1; _p2 = s._p2; return *this; }
  const Extended_point<RT>& source() const { return _p1; }
  const Extended_point<RT>& target() const { return _p2; }
  void line_equation(RT& a, RT& b, SPolynomial<RT>& c) const;
};
```

⟨*extended segments*⟩+≡
```
template <class RT>
std::ostream& operator<<(std::ostream& os, const Extended_segment<RT>& s)
{ os << s.source() << s.target(); return os; }
template <class RT>
std::istream& operator>>(std::istream& is, Extended_segment<RT>& s)
{ Extended_point<RT> p1,p2;
  is >> p1 >> p2; s=Extended_segment<RT>(p1,p2); return is; }
```

We extract the line equation $ax+by+c=0$ directly from non-standard points if the extended segment is just a segment. Note that for all lines crossing the interior of our box a,b,c are just constants from our integer ring type RT. However the frame segments also support lines of the from $x \pm R = 0$ and $y \pm R = 0$. Therefore, we allow a linear polynomial for the coefficient c.

⟨*extended segment primitives*⟩≡
```
template <typename RT>
void Extended_segment<RT>::
line_equation(RT& a, RT& b, SPolynomial<RT>& c) const
{
  bool sstandard = _p1.is_standard();
  bool tstandard = _p2.is_standard();
  if (sstandard && tstandard) {
    ⟨standard segment⟩
  }
  Extended_point<RT> p;
  bool correct_orientation=true;
  if (!sstandard && !tstandard) {
    ⟨two points on the frame box⟩
  }
  else if (sstandard && !tstandard)
  { p = _p2; }
  else if (!sstandard && tstandard)
  { p = _p1; correct_orientation=false; }
  ⟨one point on the frame box⟩
}
```

If two points span a standard segment then the three coefficients of the line through the points can be derived from the following determinant equation (just resolve for the variables in the last row).

$$\begin{vmatrix} x1 & y1 & w1 \\ x2 & y2 & w2 \\ a & b & c \end{vmatrix} = 0$$

.

⟨*standard segment*⟩≡
```
a = _p1.ny()*_p2.hw() - _p2.ny()*_p1.hw();
b = _p1.hw()*_p2.nx() - _p2.hw()*_p1.nx();
c = SPolynomial<RT>(_p1.nx()*_p2.ny() - _p2.nx()*_p1.ny());
return;
```

Two points on the box produce two basic configurations. Either the points are both on one frame segment. Or they are part of one affine line crossing the box. If they lie on the same frame segment then their corresponding coordinate polynomials are equal in homogeneous representation and equal to $\pm R$. Note that we keep the algebraic calculation within the polynomials of degree less than 2. When both points lie on the same frame segment then the minimal representation of the corresponding coordinate polynomial is $\pm 1R + 0$. This leads to line equations of the form $0x + 1y + \mp R = 0$ when the y-coordinates are equal and to $1x + 0y + \mp R = 0$ when the x-coordinates are equal. In case that the points span a standard affine line we forward the treatment to the mixed case by setting p to one of the points.

⟨*two points on the frame box*⟩≡
```
bool x_equal = (_p1.hx()*_p2.hw() - _p2.hx()*_p1.hw()).is_zero();
bool y_equal = (_p1.hy()*_p2.hw() - _p2.hy()*_p1.hw()).is_zero();
if (x_equal && CGAL_NTS abs(_p1.mx())==_p1.hw() && _p1.nx()==0 )
{ int dy = (_p2.hy()-_p1.hy()).sign();
  a=-dy; b=0; c = SPolynomial<RT>(dy*_p1.hx().sign(),0); return; }
if (y_equal && CGAL_NTS abs(_p1.my())==_p1.hw() && _p1.ny()==0 )
{ int dx = (_p2.hx()-_p1.hx()).sign();
  a=0; b=dx; c = SPolynomial<RT>(-dx*_p1.hy().sign(),0); return; }
p = _p2; // evaluation according to mixed case
```

Finally we have the point p at the tip of a line (somewhere on the frame box). Obviously we can extract the affine equation of the line from the polynomial representation. We just set the parameter R to 0 and 1, obtain two points, and calculate the equation. Note that by the special structure of our non-standard points the 2×2 determinants reduce nicely. $p.hw(\,)$ is a common factor of all coefficients a, b, c of the line:

$$a = y_1 w_1 - y_2 w_1, b = x_2 w_1 - x_1 w_2, c = x_1 y_2 - x_2 y_1$$

where

$$x_1 = p.nx(\,), y_1 = p.ny(\,), w_1 = p.hw(\,), x_2 = p.mx(\,) + p.nx(\,), y_2 = p.my(\,) + p.ny(\,), w_2 = p.hw(\,)$$

For a and b it is obvious that $p.hw(\,)$ can be canceled out, and the difference simplified. c can be reduced to $p.nx(\,) * p.my(\,) - p.ny(\,) * p.mx(\,)$ due to the special choice of our points. Remember that for non-standard points either their x- or y-coordinate is equal to $\pm R$. In the homogeneous representation this means that either $p.mx(\,) = \pm p.hw(\,)$ and $p.nx(\,) = 0$, or $p.my(\,) = \pm p.hw(\,)$ and $p.ny(\,) = 0$. In either case one can safely divide by $p.hw(\,)$.

⟨*one point on the frame box*⟩≡
```
RT x1 = p.nx(), y1 = p.ny();                // R==0
RT x2 = p.mx()+p.nx(), y2 = p.my()+p.ny(); // R==1
RT w = p.hw();
RT ci;
if ( correct_orientation ) {
  a = -p.my(); // (y1*w-w*y2)/w
  b =  p.mx(); // (x2*w-w*x1)/w
  ci = (p.nx()*p.my()-p.ny()*p.mx())/w; // (x1*y2-x2*y1)/w;
} else {
  a =  p.my(); // (y2*w-w*y1)
```

```
  b = -p.mx(); // (x1*w-w*x2)
  ci = (p.ny()*p.mx()-p.nx()*p.my())/w; // (x2*y1-x1*y2)/w;
}
c = SPolynomial<RT>(ci);
```

We finally provide the intersection construction of two lines supported by two segments. We use the linear system that defines the common point of two lines.

$$\begin{pmatrix} a1 & b1 \\ a2 & b2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -c_1 \\ -c_2 \end{pmatrix}$$

Note that the line equations are either standard affine lines or lines supporting box segments. The expressions for x and y are polynomials in R up to degree 1, for w it is just a constant.

⟨*extended segment primitives*⟩+≡

```
template <typename RT>
Extended_point<RT> intersection(
  const Extended_segment<RT>& s1, const Extended_segment<RT>& s2)
{
  RT a1,b1,a2,b2;
  SPolynomial<RT> c1,c2;
  s1.line_equation(a1,b1,c1);
  s2.line_equation(a2,b2,c2);
  SPolynomial<RT> x = c2*b1 - c1*b2;
  SPolynomial<RT> y = c1*a2 - c2*a1;
  RT w = a1*b2 - a2*b1; CGAL_assertion(w!=0);
  ⟨reduce point representation by gcd operation⟩
  return Extended_point<RT>(x,y,w);
}
```

We introduce an option that allows us to reduce the homogeneous representation of points by dividing both polynomials by the gcd of their content[1] and their common denominator. This leads to a representation of minimal bitlength. We found that this reduction pays off a lot. Remember that we use the kernel in binary operations of Nef polyhedra and their recursive usage of intermediate structures accumulates long point representations without this reduction.

⟨*reduce point representation by gcd operation*⟩≡

```
#ifdef REDUCE_INTERSECTION_POINTS
RT xgcd,ygcd;
if ( x.m() == RT(0) )  xgcd = ( x.n() == 0 ? RT(1) : x.n() );
else /* != 0 */    xgcd = ( x.n() == 0 ? x.m() : gcd(x.m(),x.n()) );
if ( y.m() == RT(0) )  ygcd = ( y.n() == 0 ? RT(1) : y.n() );
else /* != 0 */    ygcd = ( y.n() == 0 ? y.m() : gcd(y.m(),y.n()) );
RT d = gcd(w,gcd(xgcd,ygcd));
x /= d;
y /= d;
w /= d;
#endif // REDUCE_INTERSECTION_POINTS
```

[1] Remember: the content of a polynomial is the gcd of all nonzero coefficients.

We introduce the two primitives *orientation* and *is_degenerate* for user comfort. The *contains* predicate checks if a point is part of a relatively closed segment.

⟨*extended segment primitives*⟩+≡
```
template <typename RT>
inline
int orientation(const Extended_segment<RT>& s, const Extended_point<RT>& p)
{ return orientation(s.source(),s.target(),p); }

template <typename RT>
inline
bool is_degenerate(const Extended_segment<RT>& s)
{ return s.source()==s.target(); }

template <typename RT>
inline
bool contains(const Extended_segment<RT>& s,
              const Extended_point<RT>& p)
{ int p_rel_source = compare_xy(p,s.source());
  int p_rel_target = compare_xy(p,s.target());
  return ( orientation(s,p) == 0 ) &&
         ( p_rel_source >= 0 && p_rel_target <= 0 ||
           p_rel_source <= 0 && p_rel_target >= 0 );
}
```

Direction predicates

⟨*extended directions*⟩≡
```
template <typename RT>
class Extended_direction {
  Extended_point<RT> _p1,_p2;
public:
  Extended_direction() : _p1(),_p2() {}
  Extended_direction(const Extended_direction<RT>& d) :
    _p1(d._p1),_p2(d._p2) {}
  Extended_direction<RT>& operator=(const Extended_direction<RT>& d)
  { _p1 = d._p1; _p2 = d._p2; return *this; }

  Extended_direction(const Extended_point<RT>& p1,
                     const Extended_point<RT>& p2) :
    _p1(p1),_p2(p2) {}

  Extended_direction(const RT& x, const RT& y) :
    _p1(0,0,1),_p2(x,y,1) {}

  const Extended_point<RT>& p1() const { return _p1; }
  const Extended_point<RT>& p2() const { return _p2; }
  int dx_sign() const
  { return (_p2.hx()*_p1.hw()-_p1.hx()*_p2.hw()).sign(); }
  int dy_sign() const
  { return (_p2.hy()*_p1.hw()-_p1.hy()*_p2.hw()).sign(); }
};

template <class RT>
std::ostream& operator<<(std::ostream& os, const Extended_direction<RT>& d)
```

```
{ os << d.p1() << "," << d.p2();
  return os; }
template <class RT>
std::istream& operator>>(std::istream& is, Extended_direction<RT>& d)
{ Extended_point<RT> x,y;
  is >> x >> y; d = Extended_direction<RT>(x,y);
  return is; }
```

⟨*direction predicates*⟩≡

```
template <typename NT>
inline
int coeff2_dor(const NT& mx1, const NT& /*nx1*/,
               const NT& my1, const NT& /*ny1*/, const NT& w1,
               const NT& mx2, const NT& /*nx2*/,
               const NT& my2, const NT& /*ny2*/, const NT& w2,
               const NT& mx3, const NT& /*nx3*/,
               const NT& my3, const NT& /*ny3*/, const NT& w3,
               const NT& mx4, const NT& /*nx4*/,
               const NT& my4, const NT& /*ny4*/, const NT& w4)
{
  NT coeff2 = w1*mx2*w3*my4-w1*mx2*w4*my3-w2*mx1*w3*my4+w2*mx1*w4*my3-
              w1*my2*w3*mx4+w1*my2*w4*mx3+w2*my1*w3*mx4-w2*my1*w4*mx3;
  return CGAL_NTS sign(coeff2);
}

template <typename NT>
inline
int coeff1_dor(const NT& mx1, const NT& nx1,
               const NT& my1, const NT& ny1, const NT& w1,
               const NT& mx2, const NT& nx2,
               const NT& my2, const NT& ny2, const NT& w2,
               const NT& mx3, const NT& nx3,
               const NT& my3, const NT& ny3, const NT& w3,
               const NT& mx4, const NT& nx4,
               const NT& my4, const NT& ny4, const NT& w4)
{
  NT coeff1 = -w1*my2*w3*nx4+w1*mx2*w3*ny4+w1*my2*w4*nx3-w1*mx2*w4*ny3+
               w1*nx2*w3*my4-w1*nx2*w4*my3+w2*my1*w3*nx4-w2*mx1*w3*ny4-
               w2*my1*w4*nx3+w2*mx1*w4*ny3-w2*nx1*w3*my4+w2*nx1*w4*my3-
               w1*ny2*w3*mx4+w1*ny2*w4*mx3+w2*ny1*w3*mx4-w2*ny1*w4*mx3;
  return CGAL_NTS sign(coeff1);
}

template <typename NT>
inline
int coeff0_dor(const NT& /*mx1*/, const NT& nx1,
               const NT& /*my1*/, const NT& ny1, const NT& w1,
               const NT& /*mx2*/, const NT& nx2,
               const NT& /*my2*/, const NT& ny2, const NT& w2,
               const NT& /*mx3*/, const NT& nx3,
               const NT& /*my3*/, const NT& ny3, const NT& w3,
               const NT& /*mx4*/, const NT& nx4,
               const NT& /*my4*/, const NT& ny4, const NT& w4)
{
```

```
  NT coeff0 = w1*nx2*w3*ny4-w1*nx2*w4*ny3-w2*nx1*w3*ny4+w2*nx1*w4*ny3-
              w1*ny2*w3*nx4+w1*ny2*w4*nx3+w2*ny1*w3*nx4-w2*ny1*w4*nx3;
  return CGAL_NTS sign(coeff0);
}
DEFCOUNTER(ord2)
DEFCOUNTER(ord1)
DEFCOUNTER(ord0)

template <typename RT>
inline
int orientation(const Extended_direction<RT>& d1,
                const Extended_direction<RT>& d2)
{
  Extended_point<RT> p1(d1.p1()), p2(d1.p2()),
                     p3(d2.p1()), p4(d2.p2());
  int res;
  try { INCTOTAL(ord2); Protect_FPU_rounding<true> Protection;
    res = coeff2_dor(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
                     p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                     p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD(),
                     p4.mxD(),p4.nxD(),p4.myD(),p4.nyD(),p4.hwD());
  } catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(ord2);
    res = coeff2_dor(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                     p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
                     p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw(),
                     p4.mx(),p4.nx(),p4.my(),p4.ny(),p4.hw());
  }
  if ( res != 0 ) return res;
  try { INCTOTAL(ord1); Protect_FPU_rounding<true> Protection;
    res = coeff1_dor(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
                     p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                     p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD(),
                     p4.mxD(),p4.nxD(),p4.myD(),p4.nyD(),p4.hwD());
  } catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(ord1);
    res = coeff1_dor(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                     p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
                     p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw(),
                     p4.mx(),p4.nx(),p4.my(),p4.ny(),p4.hw());
  }
  if ( res != 0 ) return res;
  try { INCTOTAL(ord0); Protect_FPU_rounding<true> Protection;
    res = coeff0_dor(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
                     p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                     p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD(),
                     p4.mxD(),p4.nxD(),p4.myD(),p4.nyD(),p4.hwD());
  } catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(ord0);
    res = coeff0_dor(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                     p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
                     p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw(),
                     p4.mx(),p4.nx(),p4.my(),p4.ny(),p4.hw());
  }
  return res;
}
```

In affine geometry two directions are equal iff

```
d1.dx()*d2.dy() == d1.dy()*d2.dx() &&
sign(d1.dx()) == sign(d2.dx()) &&
sign(d1.dy()) == sign(d2.dy())
```

We thus translate the first into a filtered determinant expression and check the second and third line directly.

⟨*direction predicates*⟩+≡

```
template <typename RT>
inline
bool operator==(const Extended_direction<RT>& d1,
                const Extended_direction<RT>& d2)
{
  return orientation(d1,d2) == 0 &&
         d1.dx_sign() == d2.dx_sign() &&
         d1.dy_sign() == d2.dy_sign();
}

template <typename RT>
inline
bool operator!=(const Extended_direction<RT>& d1,
                const Extended_direction<RT>& d2)
{ return !(d1==d2); }

template <typename RT>
bool strictly_ordered_ccw(const Extended_direction<RT>& d1,
                          const Extended_direction<RT>& d2,
                          const Extended_direction<RT>& d3)
{
  if (d1 == d3) return (d1 != d2);
  int or12 = orientation(d1,d2);
  int or13 = orientation(d1,d3);
  int or32 = orientation(d3,d2);
  if ( or13 >= 0 ) // not rightturn
    return ( or12 > 0 && or32 < 0 );
  else // ( or13 < 0 ) rightturn
    return ( or12 > 0 || or32 < 0 );
}

template <typename RT>
inline
bool operator<(const Extended_direction<RT>& d1,
               const Extended_direction<RT>& d2)
{ Extended_direction<RT> d0(1,0);
  bool d0d1eq = (d1 == d0);
  bool d0d2eq = (d2 == d0);
  return (d0d1eq && !d0d2eq) ||
         strictly_ordered_ccw(d0,d1,d2) && !d0d2eq;
}
```

The kernel wrapper

All operations are either mapped to the above primitives or similar to the ones in the default homogeneous kernel. We do not present the layout of the kernel class *Filtered_extended_homogeneous*<*RT*>. It is similar to *Extended_homogeneous*<*RT*>. We only present the usefulness of *Extended_homogeneous*<*RT*> with respect to checking of our advanced kernel. We used the naive approach to back-up the results of our filtered code. As the expressions in this code base are much more complicated, errors were hard to determine. We enriched the filtered kernel by checking statements that we switched on when runtime examples seemed to produce problems. (Of course we had to log our random test inputs to allow checking in case of errors). When switched on by using the compile flag *KERNEL_CHECK* our checking macro is defined to be

```
#define CHECK(c1,c2) CGAL_assertion((c1) == (c2));
```

Then the orientation member of *Filtered_extended_homogeneous*<*RT*> is just defined as:

```
int orientation(const Point_2& p1, const Point_2& p2, const Point_2& p3)
const
{ CHECK(K.orientation(p1.checkrep(),p2.checkrep(),p3.checkrep()),
        CGAL::orientation(p1,p2,p3))
  return CGAL::orientation(p1,p2,p3); }
```

where *Point_2* is the extended point type in the local scope of *Filtered_extended_homogeneous*<*RT*>. *p.checkrep*( ) returns an extended point of type *Extended_homogeneous*<*RT*>::*Point_2* based on the *RPolynomial*<*RT*> number type. And *K* is a kernel object of type *Extended_homogeneous*<*RT*> in the local scope.

At last the filtered kernel has a member method *print_statistics*( ) that outputs the total number of failed filter stages in its base version. If the kernel is used with the compile flag *KERNEL_ANALYSIS* then each filtered code section is evaluated separately. All sections give the number of failed stages and the number of total evaluations thereby the efficiency of the filter can be evaluated with respect to the input used. This ends the description of the filtered extended kernel model.

⟨*extended kernel*⟩≡

```
template <typename RT_>
class Filtered_extended_homogeneous {
typedef Filtered_extended_homogeneous<RT_> Self;

public:
typedef CGAL::Homogeneous<RT_> Standard_kernel;
typedef typename Standard_kernel::RT           Standard_RT;
typedef typename Standard_kernel::FT           Standard_FT;
typedef typename Standard_kernel::Point_2      Standard_point_2;
typedef typename Standard_kernel::Segment_2    Standard_segment_2;
typedef typename Standard_kernel::Line_2       Standard_line_2;
typedef typename Standard_kernel::Direction_2  Standard_direction_2;
typedef typename Standard_kernel::Ray_2        Standard_ray_2;
typedef typename Standard_kernel::Aff_transformation_2
  Standard_aff_transformation_2;

typedef SPolynomial<RT_>       RT;
typedef SQuotient<RT_>         FT;
typedef Extended_point<RT_>     Point_2;
typedef Extended_segment<RT_>   Segment_2;
typedef Extended_direction<RT_> Direction_2;
```

⟨*kernel check ops*⟩
```
enum Point_type { SWCORNER=1, LEFTFRAME, NWCORNER,
                  BOTTOMFRAME, STANDARD, TOPFRAME,
                  SECORNER, RIGHTFRAME, NECORNER };
```
⟨*extended filtered homogeneous kernel members*⟩
```
};
```

⟨*extended filtered homogeneous kernel members*⟩≡
```
Standard_RT dx(const Standard_line_2& l) const { return l.b(); }
Standard_RT dy(const Standard_line_2& l) const { return -l.a(); }
Standard_FT abscissa_distance(const Standard_line_2& l) const
{ return Standard_kernel::make_FT(-l.c(),l.b()); }

Point_type determine_type(const Standard_line_2& l) const
{
  //
  Standard_RT adx = CGAL_NTS abs(dx(l)), ady = CGAL_NTS abs(dy(l));
  int sdx = CGAL_NTS sign(dx(l)), sdy = CGAL_NTS sign(dy(l));
  int cmp_dx_dy = CGAL_NTS compare(adx,ady), s(1);
  //
  if (sdx < 0 && ( cmp_dx_dy > 0 || cmp_dx_dy == 0 &&
      sdy != (s=CGAL_NTS sign(abscissa_distance(l))))) {
    if (0 == s) return ( sdy < 0 ? SWCORNER : NWCORNER );
    else        return LEFTFRAME;
  } else if (sdx > 0 && ( cmp_dx_dy > 0 || cmp_dx_dy == 0 &&
             sdy != (s=CGAL_NTS sign(abscissa_distance(l))))) {
    if (0 == s) return ( sdy < 0 ? SECORNER : NECORNER );
    else        return RIGHTFRAME;
  } else if (sdy < 0 && ( cmp_dx_dy < 0 || cmp_dx_dy == 0 &&
             abscissa_distance(l) < Standard_FT(0))) {
    return BOTTOMFRAME;
  } else if (sdy > 0 && ( cmp_dx_dy < 0 || cmp_dx_dy == 0 &&
             abscissa_distance(l) > Standard_FT(0))) {
    return TOPFRAME;
  }
  CGAL_assertion_msg(false," determine_type: degenerate line.");
  return (Point_type)-1; // never come here
}

Point_2 epoint(const Standard_RT& m1, const Standard_RT& n1,
               const Standard_RT& m2, const Standard_RT& n2,
                                      const Standard_RT& n3) const
{ return Point_2(m1,n1,m2,n2,n3); }

public:

Point_2 construct_point(const Standard_point_2& p) const
{ return Point_2(p.hx(), p.hy(), p.hw()); }

Point_2 construct_point(const Standard_line_2& l, Point_type& t) const
{
  t = determine_type(l);
  //
  Point_2 res;
  switch (t) {
```

```
    case SWCORNER:   res = epoint(-1, 0, -1, 0, 1); break;
    case NWCORNER:   res = epoint(-1, 0,  1, 0, 1); break;
    case SECORNER:   res = epoint( 1, 0, -1, 0, 1); break;
    case NECORNER:   res = epoint( 1, 0,  1, 0, 1); break;
    case LEFTFRAME:  res = epoint(-l.b(), 0,  l.a(), -l.c(), l.b());
                     break;
    case RIGHTFRAME: res = epoint( l.b(), 0, -l.a(), -l.c(), l.b());
                     break;
    case BOTTOMFRAME: res = epoint( l.b(), -l.c(), -l.a(), 0, l.a());
                     break;
    case TOPFRAME: res = epoint(-l.b(), -l.c(),  l.a(), 0, l.a());
                     break;
    default: CGAL_assertion_msg(0,"EPoint type not correct!");
  }
  return res;
}
Point_2 construct_point(const Standard_point_2& p1,
                        const Standard_point_2& p2,
                        Point_type& t) const
{ return construct_point(Standard_line_2(p1,p2),t); }
Point_2 construct_point(const Standard_line_2& l) const
{ Point_type dummy; return construct_point(l,dummy); }
Point_2 construct_point(const Standard_point_2& p1,
                        const Standard_point_2& p2) const
{ return construct_point(Standard_line_2(p1,p2)); }
Point_2 construct_point(const Standard_point_2& p,
                        const Standard_direction_2& d) const
{ return construct_point(Standard_line_2(p,d)); }
Point_2 construct_opposite_point(const Standard_line_2& l) const
{ Point_type dummy; return construct_point(l.opposite(),dummy); }
Point_type type(const Point_2& p) const
{
  if (p.is_standard()) return STANDARD;
  // now we are on the square frame
  RT rx = p.hx();
  RT ry = p.hy();
  int sx = CGAL_NTS sign(rx);
  int sy = CGAL_NTS sign(ry);
  if (sx < 0) rx = -rx;
  if (sy < 0) ry = -ry;
  if (rx>ry) {
    if (sx > 0) return RIGHTFRAME;
    else        return LEFTFRAME;
  }
  if (rx<ry) {
    if (sy > 0) return TOPFRAME;
    else        return BOTTOMFRAME;
  }
  // now (rx == ry)
  if (sx==sy) {
    if (sx < 0) return SWCORNER;
    else        return NECORNER;
```

```
  } else { CGAL_assertion(sx==-sy);
    if (sx < 0) return NWCORNER;
    else        return SECORNER;
  }
}

bool is_standard(const Point_2& p) const
{ return p.is_standard();  }

Standard_point_2 standard_point(const Point_2& p) const
{ CGAL_assertion(is_standard(p));
  return Standard_point_2(p.nx(),p.ny(),p.hw());
}

Standard_line_2 standard_line(const Point_2& p) const
{ CGAL_assertion(!p.is_standard());
  Standard_point_2 p0(p.nx(),p.ny(),p.hw());
  Standard_point_2 p1(p.mx()+p.nx(),p.my()+p.ny(),p.hw());
  return Standard_line_2(p0,p1);
}

Standard_ray_2 standard_ray(const Point_2& p) const
{ CGAL_assertion(!p.is_standard());
  Standard_line_2 l = standard_line(p);
  Standard_direction_2 d = l.direction();
  Standard_point_2 q = l.point(0);
  return Standard_ray_2(q,d);
}

Point_2 NE() const { return construct_point(Standard_line_2(-1, 1,0)); }
Point_2 SE() const { return construct_point(Standard_line_2( 1, 1,0)); }
Point_2 NW() const { return construct_point(Standard_line_2(-1,-1,0)); }
Point_2 SW() const { return construct_point(Standard_line_2( 1,-1,0)); }

int orientation(const Point_2& p1, const Point_2& p2, const Point_2& p3)
const
{ CHECK(K.orientation(p1.checkrep(),p2.checkrep(),p3.checkrep()),
        CGAL::orientation(p1,p2,p3))
  return CGAL::orientation(p1,p2,p3); }

bool leftturn(const Point_2& p1, const Point_2& p2, const Point_2& p3)
const
{ return orientation(p1,p2,p3) > 0; }

bool first_pair_closer_than_second(
  const Point_2& p1, const Point_2& p2,
  const Point_2& p3, const Point_2& p4) const
{ CHECK(K.first_pair_closer_than_second(p1.checkrep(),p2.checkrep(),
                                        p3.checkrep(),p4.checkrep()),
        CGAL::compare_pair_dist(p1,p2,p3,p4)<0)
  return CGAL::compare_pair_dist(p1,p2,p3,p4)<0; }

int compare_xy(const Point_2& p1, const Point_2& p2) const
{ CHECK(K.compare_xy(p1.checkrep(),p2.checkrep()),
        CGAL::compare_xy(p1,p2))
  return CGAL::compare_xy(p1,p2); }

int compare_x(const Point_2& p1, const Point_2& p2) const
{ CHECK(K.compare_x(p1.checkrep(),p2.checkrep()),
        CGAL::compare_x(p1,p2))
```

```
    return CGAL::compare_x(p1,p2); }
  int compare_y(const Point_2& p1, const Point_2& p2) const
  { CHECK(K.compare_y(p1.checkrep(),p2.checkrep()),
          CGAL::compare_y(p1,p2))
    return CGAL::compare_y(p1,p2); }
  bool strictly_ordered_along_line(
    const Point_2& p1, const Point_2& p2, const Point_2& p3) const
  { CHECK(K.strictly_ordered_along_line(
            p1.checkrep(),p2.checkrep(),p3.checkrep()),
          CGAL::strictly_ordered_along_line(p1,p2,p3))
    return CGAL::strictly_ordered_along_line(p1,p2,p3); }

  Segment_2 construct_segment(const Point_2& p, const Point_2& q) const
  { return Segment_2(p,q); }
  Point_2 source(const Segment_2& s) const
  { return s.source(); }
  Point_2 target(const Segment_2& s) const
  { return s.target(); }
  bool is_degenerate(const Segment_2& s) const
  { return s.source()==s.target(); }
  int orientation(const Segment_2& s, const Point_2& p) const
  { return orientation(s.source(),s.target(),p); }
  Point_2 intersection(const Segment_2& s1, const Segment_2& s2) const
  { CHECK(CGAL::intersection(s1,s2).checkrep(),
          K.intersection(convert(s1),convert(s2)))
    return CGAL::intersection(s1,s2); }
  bool contains(const Segment_2& s, const Point_2& p) const
  { return CGAL::contains(s,p); }
  Direction_2 construct_direction(
    const Point_2& p1, const Point_2& p2) const
  { return Direction_2(p1,p2); }
  bool strictly_ordered_ccw(const Direction_2& d1,
    const Direction_2& d2, const Direction_2& d3) const
  { CHECK(K.strictly_ordered_ccw(convert(d1),convert(d2),convert(d3)),
          CGAL::strictly_ordered_ccw(d1,d2,d3));
    return CGAL::strictly_ordered_ccw(d1,d2,d3); }
  void print_statistics() const
  {
    std::cout << "Statistics of filtered kernel:\n";
    std::cout << "total failed double filter stages = ";
    std::cout << CGAL::Interval_nt_advanced::number_of_failures << std::endl;
    PRINT_CHECK_ENABLED;
    PRINT_STATISTICS(or2);
    PRINT_STATISTICS(or1);
    PRINT_STATISTICS(or0);
    PRINT_STATISTICS(cmpx1);
    PRINT_STATISTICS(cmpx0);
    PRINT_STATISTICS(cmpy1);
    PRINT_STATISTICS(cmpy0);
    PRINT_STATISTICS(cmppd2);
    PRINT_STATISTICS(cmppd1);
```

```
  PRINT_STATISTICS(cmppd0);
  PRINT_STATISTICS(ord2);
  PRINT_STATISTICS(ord1);
  PRINT_STATISTICS(ord0);
}
template <class Forward_iterator>
void determine_frame_radius(Forward_iterator start, Forward_iterator end,
                            Standard_RT& R0) const
{ Standard_RT R;
  while ( start != end ) {
    Point_2 p = *start;
    if ( is_standard(p) ) {
      R = CGAL_NTS max(CGAL_NTS abs(p.mx())/p.hw(),
                       CGAL_NTS abs(p.my())/p.hw());
    } else {
      RT rx = CGAL_NTS abs(p.hx()), ry = CGAL_NTS abs(p.hy());
      if ( rx[1] > ry[1] )      R = CGAL_NTS abs(ry[0]-rx[0])/(rx[1]-ry[1]);
      else if ( rx[1] < ry[1] ) R = CGAL_NTS abs(rx[0]-ry[0])/(ry[1]-rx[1]);
      else /* rx[1] == ry[1] */ R = CGAL_NTS abs(rx[0]-ry[0])/(2*p.hw());
    }
    R0 = CGAL_NTS max(R+1,R0); ++start;
  }
}
const char* output_identifier() const
{ return "Filtered_extended_homogeneous"; }
```

⟨*Filtered_extended_homogeneous.h*⟩≡
⟨*CGAL EH Header*⟩

```
#ifndef CGAL_FILTERED_EXTENDED_HOMOGENEOUS_H
#define CGAL_FILTERED_EXTENDED_HOMOGENEOUS_H
#include <CGAL/basic.h>
#include <CGAL/Handle_for.h>
#include <CGAL/Interval_arithmetic.h>
#include <CGAL/Homogeneous.h>
#undef _DEBUG
#define _DEBUG 59
#include <CGAL/Nef_2/debug.h>
#define REDUCE_INTERSECTION_POINTS
//#define KERNEL_ANALYSIS
//#define KERNEL_CHECK
#ifdef  KERNEL_CHECK
#include <CGAL/Extended_homogeneous.h>
#define CHECK(c1,c2) CGAL_assertion((c1) == (c2));
#define PRINT_CHECK_ENABLED std::cout << "kernel check enabled!\n"
#else
#define CHECK(c1,c2)
#define PRINT_CHECK_ENABLED std::cout << "no kernel check!\n"
#endif
#ifdef KERNEL_ANALYSIS
#define DEFCOUNTER(c) \
  static int c##_total=0; static int c##_exception=0;
```

```
#define INCTOTAL(c) c##_total++
#define INCEXCEPTION(c) c##_exception++
#define PRINT_STATISTICS(c) \
std::cout << #c##" " << c##_exception << "/" << c##_total << std::endl
#else
#define DEFCOUNTER(c)
#define INCTOTAL(c)
#define INCEXCEPTION(c)
#define PRINT_STATISTICS(c)
#endif
CGAL_BEGIN_NAMESPACE
⟨simple polynomials⟩
⟨extended points⟩
⟨orientation predicate⟩
⟨comparison predicate⟩
⟨extended segments⟩
⟨extended segment primitives⟩
⟨extended directions⟩
⟨direction predicates⟩
⟨extended kernel⟩
CGAL_END_NAMESPACE
#undef CHECK
#undef KERNEL_CHECK
#undef REDUCE_INTERSECTION_POINTS
#undef KERNEL_ANALYSIS
#undef COUNTER
#undef INCTOTAL
#undef INCEXCEPTION
#undef PRINT_STATISTICS
#undef PRINT_CHECK_ENABLED
#endif // CGAL_FILTERED_EXTENDED_HOMOGENEOUS_H
```

Within our extended point type we add conversion to the polynomial based version.

⟨*point check ops*⟩≡

```
#ifdef KERNEL_CHECK
typedef CGAL::Extended_homogeneous<RT_> CheckKernel;
typedef typename CheckKernel::Point_2   CheckPoint;
typedef typename CheckKernel::RT        CheckRT;
CheckRT convert(const CGAL::SPolynomial<RT_>& p) const
{ return CheckRT(p.n(),p.m()); }
CheckRT convert(const RT_& t) const
{ return CheckRT(t); }
CheckPoint checkrep() const
{ return CheckPoint(convert(hx()),convert(hy()),convert(w())); }
#endif // KERNEL_CHECK
```

⟨*kernel check ops*⟩≡

```
#ifdef KERNEL_CHECK
typedef Extended_homogeneous<RT_>         CheckKernel;
```

```
typedef typename CheckKernel::Point_2     CheckPoint;
typedef typename CheckKernel::Direction_2 CheckDirection;
typedef typename CheckKernel::Segment_2   CheckSegment;
CheckKernel K;

CheckSegment convert(const Segment_2& s) const
{ return CheckSegment(s.source().checkrep(),
                      s.target().checkrep()); }
CheckDirection convert(const Direction_2& d) const
{ return K.construct_direction(d.p2().checkrep(),d.p1().checkrep()); }

#endif // KERNEL_CHECK
```

Bibliography

[BBP98] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1998.

[CGG+91] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt. *Maple V : language reference manual*. Springer, / 1st ed. edition, 1992/1991.

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991.

[MN99] K. Mehlhorn and S. Näher. *LEDA, A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[SM00] M. Seel and K. Mehlhorn. Infimaximal Frames: a framework to make lines look like segments. Technical Report MPI-I-2000-1-005, Max-Planck-Institut für Informatik, Saarbrücken, 2000.

9 Plane Map Implementation

We present some implementation details of our plane map decorator that provides an abstract interface to a plane map. The abstract interface of our plane map data type is sufficiently specified in the manual page. We sketch how we implement this interface by the CGAL halfedge data structure. We use the new HDS design as described in the design paper [Ket99]. The paper contains also a survey of classical plane map implementations and a motivation for the HDS design. The generic HDS collection allows to choose different flavors of HDS structures. A user can specify if she uses explicit vertex or face objects and how the iteration facilities are implemented. For the topological Nef layer we choose the default implementation including vertex and face objects, however the offered design is limited to one single face cycle bounding a face. Our definition of plane maps requires to have multiple face cycles and also trivial face cycles in form of isolated vertices. We do not want to bore the reader with the technical details of the implementation but we describe the extension process from the functionality of the default HDS design to our plane map data type. Fortunately, the CGAL HDS allows a user to extend the functionality by extending the objects (vertices, halfedges, faces). The types are transported into the container type HDS in a so-called items class; in our case called *HDS_items*. The possibility of this extension is one advantage of the generic design.

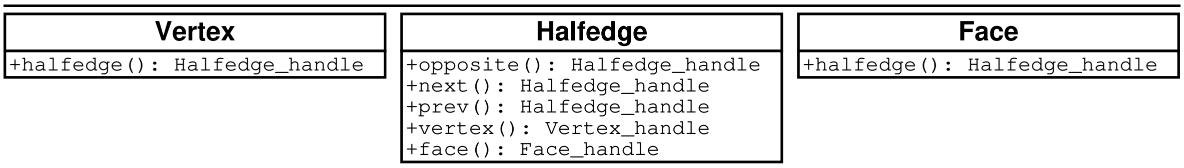


Figure 9.1: The default HDS design

Figure 9.1 presents the default layout of the three objects. The interface methods map to member variables. A vertex v stores an incident edge e such that $v.\mathit{halfedge}(\,) = e$ and $e.\mathit{vertex}(\,) = v$. Dually symmetrical a face f stores an edge e in its bounding face cycle: $f.\mathit{halfedge}(\,) = e$ and $e.\mathit{face}(\,) = f$. The additional links of an edge e create the topological structure of the graph. $e.\mathit{opposite}(\,)$ is used to make the graph bidirected and $e.\mathit{next}(\,)$ and $e.\mathit{prev}(\,)$ are used for the circular ordering of edges in the face cycle of a face.

The extended structure in Figure 9.2 adds the possibility to assign multiple face cycles as a boundary to a face f to the above structure. We give generic container access by means of two iterator ranges. The range $[\,f.\mathit{holes\_begin}(\,), f.\mathit{holes\_end}(\,)\,)$ stores halfedges that can be used as entry points into disjoint face cycles. Accordingly, the range $[\,f.\mathit{isolated\_vertices\_begin}(\,), f.\mathit{isolated\_vertices\_end}(\,)\,)$ maintains a set of vertices in the interior of f. Such vertices v also store their containing face by $v.\mathit{face}(\,) = f$. Note that our implementation requires that insertion or deletion operations in the

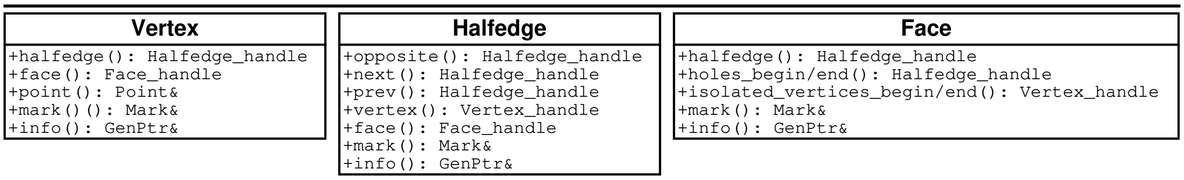


Figure 9.2: The extended HDS design

two sets are constant time operations. We do not show the details. Note that all three objects are attributed twice in addition to the topological linkage. All objects store a mark and a generic slot of type $GenPtr = void*$ for further data association. This two extensions are mandatory for the Nef structure. The *mark*( ) slots map to set inclusion flags. The *info*( ) slots allow us to associate temporary information required for the binary overlay of two structures. Finally, vertices carry a point representing their embedding into the plane. The full interfaces of the extended objects are presented in the manual pages on page 300, 300, and 301.

Decorator classes

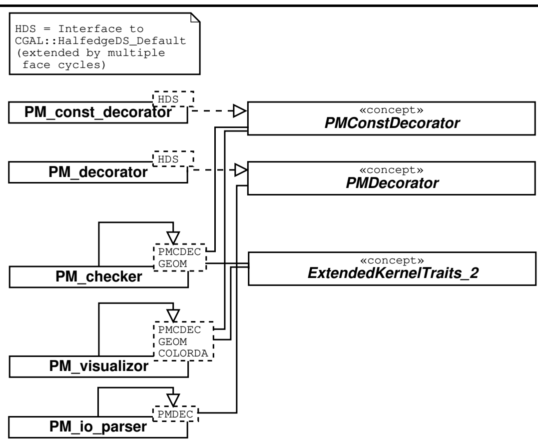


Figure 9.3: The decorator family defining the interface to the HDS data structure

We interface the HDS via decorator classes that encapsulate a certain functionality. The classes are depicted in Figure 9.3. We implemented the two main concepts *PMConstDecorator* and *PMDecorator* on top of the CGAL HDS. The first gives read-only access to the HDS the second provides manipulation operations. The concepts carry their interface into the additional modules *PM_checker*, *PM_io_parser*, *PM_visualizor*. Whenever geometric kernel operations are needed in the module as for example in the checker, we add a template parameter carrying geometric kernel methods. The *COLORDA* template parameter in the visualizor module *PM_visualizor* allows the adaptation of the drawing of plane maps. We elaborate on some details of the modules but do not show the whole implementation.

PM_const_decorator - the read-only interface

PM_const_decorator<> realizes non-mutable access to plane maps. It provides interface operations on the objects as presented in our concepts section on page 332. The sole link to geometry is the embedding via a point type. All circular structures are realized via circulators (the variation of iterators as introduced in CGAL). The only method that carries more involved coding is the integrity check operation. That operation checks the sanity of the link structure coding the incidence relations of vertices, edges, and faces and additionally checks the topological planarity of the structure by checking that the genus of the plane map is zero. The integrity check of the topological decorator does the following:

- all vertices are partitioned into two sets by the *is_isolated*( ) predicate. All isolated vertices v have face links where v is in the isolated vertices list of $v \to face()$. All non-isolated vertices are bound to adjacency lists by their halfedge link.
- for all vertices v we check that $source(A(v)) == v$
- for all edges e we check that $twin(twin(e)) == e$
- we check that the Euler formula is correctly fulfilled. Let n_v be the number of vertices, n_e be number of edges (= number of halfedges divided by 2), n_f be the number of faces, n_{fc} be the number of face cycles, and n_{cc} be the number of connected components of the map. Then at first we have $n_f = n_{fc} - n_{cc} + 1$ and we check that $n_v - n_e + n_f = 1 + n_{cc}$. Note that we have to cope with isolated vertices. They are counted in our connected component number n_{cc} and in n_v.

See Chapter 8 of the LEDA book for an elaborate treatment of this check.

PM_checker - checking geometric properties

Our checker mainly realizes the integrity checks of the basic properties of the plane map like that of an order-preserving embedding or the forward-prefix property of the adjacency lists. We also added a checker method that examines if a plane map represents a triangulation of its vertices. The implemented methods are

```
void check_order_preserving_embedding(Vertex_const_handle v) const;
void check_forward_prefix_condition(Vertex_const_handle v) const;
void check_order_preserving_embedding() const;
void check_is_triangulation() const;
```

The methods check the basic properties that we require from a plane map. The task to check if our plane map actually is a triangulation of its vertices follows the ideas as presented in [MNS$^+$99] and the LEDA book.

PM_decorator - manipulating the plane map

PM_decorator<> gives mutable access to a plane map. Apart from standard operations the interface also provides operations that are very specially designed for the updates needed in our sweep framework or in the simplification phase of our binary operations. Some operations allow changing the incidence of plane map objects only partially e.g. create an edge that is only linked to a vertex at its source. With these operations one has to be careful not to spoil the plane map structure. The advantage is that we do not need superflous allocations of objects that are only needed temporarily.

The implementation of most of the operations is straight-forward. The only operation that should be mentioned is the clone operation for plane maps. As the generic HDS container does not know the layout of the objects that it maintains, a copy construction is hard to realize for the general case. In the rare case where we actually need to copy a plane map, we use the methods:

```
void clone(const HDS& H);
template <typename LINKDA>
void clone_skeleton(const HDS& H, const LINKDA& L)
```

Both methods basically work in two stages. Let H' be the target copy of H. First each object o in H is cloned into an object o' in H' whose links still point to objects in H. We store the correspondance of o to o' in a map $M(o) = o'$. Then in the second stage we iterate all objects o' in H' and replace the links to the objects in H by the corresponding objects in H' via the map. The result is an isomorphic structure. Note that due to the fact that the *prev-next* links of the halfedges also code the embedding, this isomorphy is also topological and not only combinatorial [Die97]. Of course the geometric embedding of the vertices and the attributed marks are just transferred. The second cloning operation just extracts an topological isomorphic 1-skeleton from a full-fledged plane map. In that operation we also provide access to the newly created objects by an additional data accessor *L*. The *LINKDA* concept requires the methods:

```
struct LINKDA {
  void operator()(Vertex_handle vn, Vertex_const_handle vo) const;
  void operator()(Halfedge_handle hn, Halfedge_const_handle ho) const;
}
```

where *vn*, *hn* are the cloned objects in H' and *vo*, *ho* are the original objects of H. *L* can now be used to get a hand on the cloning process on the object level. The method is used to obtain an isomorphic graph structure that can be used for further subdivision (e.g. point location in constrained triangulations). We leave out the details, as its design is mainly determined by the design of the CGAL HDS.

PM_io_parser - stream input and output

The input and output is mainly triggered by a decorator which has the control over the I/O format and does some basic parsing when reading input. The class template *PM_io_parser<PMDEC>* has two constructors and two corresponding actions on the streams obtained on construction:

```
PM_io_parser(std::istream& is, Plane_map& H);
void read();
PM_io_parser(std::ostream& os, const Plane_map& H);
void print() const;
```

The template parameter refers to the concept *PMDecorator*. A decorator object decorating H is used to construct the plane map H when reading from the input stream, or to explore the structure when printing to the output stream. We omit the implementation details.

We only present the I/O format that is similar to that used in LEDA for general graphs. There is a header and then three sections storing the objects vertices, halfedges, faces:

```
Plane_map_2
vertices n1
halfedges n2
faces n3
0    { isolated incident_object, mark, point }
...
n1-1 { isolated incident_object, mark, point }
0    { opposite, prev, next, vertex, face, mark }
...
```

```
n2-1 { opposite, prev, next, vertex, face, mark }
0    { halfedge, fclist, ivlist, mark }
...
n3-1 { halfedge, fclist, ivlist, mark }
```

there are n_1 lines for vertices, n_2 lines for halfedges, and n_3 lines for faces. All objects are indexed by non-negative integers. Vertex lines contain a boolean marker *isolated* followed by the index of an incident object (a face if *isolated* is true, otherwise the first halfedge of the adjacency list), the attribute and the embedding. Halfedge lines store the link structure (again by indices representing the objects): the *opposite* (also called twin or reversal) halfedge, the *prev*ious and *next* halfedge of its face cycle, the incident *vertex* and the incident *face*, and the attributed *mark*. The face lines have no fixed length as the number of face cycles and isolated vertices is not bounded. Both lists *fclist* (for face cycles) and *ivlist* (for isolated vertices) are white space separated lists of numbers. Their elements are the indices of one halfedge from the corresponding face cycle or the indices of the isolated vertices in the interior of the faces respectively. The *halfedge* is the index of a halfedge of the outer face cycle of the face and the *mark* is again the attribute of the face. Note that as our index range starts at 0, we code undefined references by -1.

I/O and cloning processes bear a strong similarity. In both processes one creates isomorphic representations of pointer structures. In case of output the representation of typed pointers (handles) is a unique numbering of all objects that can be translated back during an input process.

PM_visualizor - drawing plane maps in a window

We offer a decorator drawing a plane map into a CGAL window stream, which is basically a LEDA window offering stream operations for all affine kernel objects of the CGAL geometry kernels. The class template *PM_visualizor*<*PMCDEC*, *GEOM*, *COLORDA*> requires models of the three template parameters for instantiation. The first two can be instantiated by *PM_const_decorator*<> and any geometry kernel being a model of the concept *AffineGeometryTraits_2*. The third parameter assigns colors and sizes to the objects of the plane map depending on their attributes by the following class concept:

```
struct COLORDA {
  CGAL::Color color(Vertex_const_handle, const Mark& m) const;
  CGAL::Color color(Halfedge_const_handle, const Mark& m) const;
  CGAL::Color color(Face_const_handle, const Mark& m) const;
  int width(Vertex_const_handle, const Mark& m) const;
  int width(Halfedge_const_handle, const Mark& m) const;
};
```

On construction the visualizor obtains a window stream W, a decorator D, a geometry kernel K, and a color data accessor C. The plane map referenced by D is drawn in the window W with the properties as specified by C.

```
PM_visualizor(CGAL::Window_stream& W, const PMCDEC& D,
              const GEOM& K, const COLORDA& C);
```

The class offers drawing by object or drawing of the full structure by the methods:

```
void draw(Vertex_const_handle v) const
void draw(Halfedge_const_handle e) const
void draw(Face_const_handle f) const
void draw_map() const
```

We do not show their implementation here. For the drawing of the faces we use the techniques that are used by LEDA windows to draw polygons.

9.1 Extending the HDS

We extend the basic HDS design by multiple face cycles and isolated vertices.

9.1.1 Extended vertices

To summarize the vertex design compared to the default HDS:

topology and combinatorics a vertex has extended incidence operations. Depending on the predicate *is_isolated*( ) a vertex has either a link to an incident face via [*set_*]*face*( ) or it has a link to an incident halfedge via [*set_*]*halfedge*( ).

geometry an object of type *Point* for its embedding.

attribute a mark of type *Mark* and an information slot stored in a *GenPtr*. The latter can be used by the *geninfo* class.

Internal implementation invariants:

- *is_isolated*( ) ⇔ *ivit*( ) != *nil_* an isolated vertex is stored in some faces isolated vertices list
- *is_isolated*( ) ⇔ *halfedge*( ) == *Halfedge_handle*( ) an isolated vertex has no incident halfedge.

⟨*epm vertices*⟩≡

```
template <typename Refs, typename Traits>
class Vertex_wrapper { public:
  typedef typename Traits::Point Point;
  class Vertex {
  public:
    typedef Refs     HalfedgeDS;
    typedef Vertex   Base;
    typedef CGAL::Tag_true Supports_vertex_halfedge;
    typedef CGAL::Tag_true Supports_vertex_point;
    typedef typename Refs::Vertex_handle         Vertex_handle;
    typedef typename Refs::Vertex_const_handle   Vertex_const_handle;
    typedef typename Refs::Halfedge_handle       Halfedge_handle;
    typedef typename Refs::Halfedge_const_handle Halfedge_const_handle;
    typedef typename Refs::Face_handle           Face_handle;
    typedef typename Refs::Face_const_handle     Face_const_handle;
    typedef typename Refs::Halfedge              Halfedge;
    typedef typename Refs::Face                  Face;
    typedef void*                                GenPtr;

    typedef typename Traits::Point Point;

    typedef typename Traits::Mark  Mark;

    typedef typename std::list<Vertex_handle>::iterator iv_iterator;
  private:
    Halfedge_handle _h;
```

```
    Face_handle     _f;
    Point           _p;
    iv_iterator     _ivit;
    Mark            _m;
    GenPtr          _i;
  public:
    Vertex() :
      _h(),_f(),_ivit(nil_),_m(),_i((GenPtr)0xABCD) {}
    Vertex(const Point& p) :
      _h(),_f(),_p(p),_ivit(nil_),_m(),_i((GenPtr)0xABCD) {}
    bool is_isolated() const
    { return _h == Halfedge_handle(); }
    Halfedge_handle halfedge() { return _h; }
    Halfedge_const_handle halfedge() const { return _h; }
    void set_halfedge(Halfedge_handle h) { _h=h; }
    Face_handle face() { return _f; }
    Face_const_handle face() const { return _f; }
    void set_face(Face_handle f) { _f=f; }
    Point& point() { return _p; }
    const Point& point() const { return _p; }
    Mark& mark() { return _m; }
    const Mark& mark() const { return _m; }
    GenPtr& info() { return _i; }
    const GenPtr& info() const { return _i; }
    iv_iterator ivit() const { return _ivit; }
    void set_ivit(iv_iterator it) { _ivit = it; }
    static iv_iterator nil_;
    /* stl iterators have default construction but are only equal
    comparable when copy constructed, what a mess in the specification */
    LEDA_MEMORY(Vertex)
  };
};
```

9.1.2 Extended faces

To summarize the vertex design compared to the default concept:

topology and combinatorics a face has extended incidence operations. A face stores multiple bounding face cycles in a list *FC* of halfedges serving as entry points into the face cycles. Additionally a face can store isolated vertices contained in its interior in a list *IV*.

geometry no explicit geometric information. The embedding of its bounding structure is obtained via the multiple non-trivial face cycles stored in the *FC* list and the multiple contained isolated vertices stored in the *IV* list.

attribute a mark of type *Mark* and a flexible usable information slot stored via a *GenPtr*. The latter can be used by the *geninfo* class.

Internal implementation invariants:

- the halfedge *FC.front*( ) refers to the outer face cycle of the face, all others are hole cycles. (this does not hold for our outer face which has no outer face cycle)
- all halfedges e in *FC* keep iterator links to their item in *FC* via $e \rightarrow [set\_]fcit(\,)$.
- all vertices v in *IV* keep iterator links to their item in *IV* via $v \rightarrow [set\_]ivit(\,)$.

⟨*epm faces*⟩≡

```
template <typename Refs, typename Traits>
class Face_wrapper { public:
  class Face {
  public:
    typedef CGAL::Tag_true                       Supports_face_halfedge;
    typedef Refs  HalfedgeDS;
    typedef Face  Base;
    typedef typename Refs::Vertex_handle         Vertex_handle;
    typedef typename Refs::Vertex_const_handle   Vertex_const_handle;
    typedef typename Refs::Halfedge_handle       Halfedge_handle;
    typedef typename Refs::Halfedge_const_handle Halfedge_const_handle;
    typedef typename Refs::Face_handle           Face_handle;
    typedef typename Refs::Face_const_handle     Face_const_handle;
    typedef typename Refs::Vertex                Vertex;
    typedef typename Refs::Halfedge              Halfedge;
    typedef void*                                GenPtr;
    typedef typename Traits::Mark  Mark;
  class Hole_iterator
      : public std::list<Halfedge_handle>::iterator
  { typedef typename std::list<Halfedge_handle>::iterator Ibase;
    public:
      Hole_iterator() : Ibase() {}
      Hole_iterator(const Ibase& b) : Ibase(b) {}
      Hole_iterator(const Hole_iterator& i) : Ibase(i) {}
      operator Halfedge_handle() const { return Ibase::operator*(); }
      Halfedge& operator*() { return *(Ibase::operator*()); }
      Halfedge_handle operator->() { return Ibase::operator*(); }
  };
  class Hole_const_iterator :
    public std::list<Halfedge_handle>::const_iterator
  { typedef typename std::list<Halfedge_handle>::const_iterator Ibase;
    public:
      Hole_const_iterator() : Ibase() {}
      Hole_const_iterator(const Ibase& b) : Ibase(b) {}
      Hole_const_iterator(const Hole_const_iterator& i) : Ibase(i) {}
      operator Halfedge_const_handle() const { return Ibase::operator*(); }
      const Halfedge& operator*() { return *(Ibase::operator*()); }
      Halfedge_const_handle operator->() { return Ibase::operator*(); }
  };
  class Isolated_vertex_iterator
      : public std::list<Vertex_handle>::iterator
```

```
{ typedef typename std::list<Vertex_handle>::iterator Ibase;
  public:
    Isolated_vertex_iterator() : Ibase() {}
    Isolated_vertex_iterator(const Ibase& b) : Ibase(b) {}
    Isolated_vertex_iterator(const Isolated_vertex_iterator& i)
      : Ibase(i) {}
    operator Vertex_handle() const { return Ibase::operator*(); }
    Vertex& operator*() { return *(Ibase::operator*()); }
    Vertex_handle operator->() { return Ibase::operator*(); }
};
class Isolated_vertex_const_iterator
  : public std::list<Vertex_handle>::const_iterator
{ typedef typename std::list<Vertex_handle>::const_iterator Ibase;
  public:
    Isolated_vertex_const_iterator() : Ibase() {}
    Isolated_vertex_const_iterator(const Ibase& b) : Ibase(b) {}
    Isolated_vertex_const_iterator(
      const Isolated_vertex_const_iterator& i) : Ibase(i) {}
    operator Vertex_const_handle() const { return Ibase::operator*(); }
    const Vertex& operator*() { return *(Ibase::operator*()); }
    Vertex_const_handle operator->() { return Ibase::operator*(); }
};

private:
  Halfedge_handle            _e;
  std::list<Halfedge_handle> FC;
  std::list<Vertex_handle>   IV;
  Mark                       _m;
  GenPtr                     _i;
public:
  Face() : _e(),_m(),_i((GenPtr)0xABCD) {}
  ~Face() { FC.clear(); IV.clear(); }

  void store_fc(Halfedge_handle h)
  { FC.push_back(h); h->set_fcit(--FC.end());
    CGAL_assertion(h->is_hole_entry()); }
  void remove_fc(Halfedge_handle h)
  { CGAL_assertion(h->is_hole_entry());
    FC.erase(h->fcit()); h->set_fcit(Halfedge::nil_); }
  void store_iv(Vertex_handle v)
  { IV.push_back(v); v->set_ivit(--IV.end()); }
  void remove_iv(Vertex_handle v)
  { CGAL_assertion(v->is_isolated());
    IV.erase(v->ivit()); v->set_ivit(Vertex::nil_); }

  Hole_iterator  fc_begin() { return FC.begin(); }
  Hole_iterator  fc_end()   { return FC.end(); }
  Isolated_vertex_iterator  iv_begin() { return IV.begin(); }
  Isolated_vertex_iterator  iv_end()   { return IV.end(); }
  void clear_all_entries()
  { Hole_iterator hit;
    for (hit = fc_begin(); hit!=fc_end(); ++hit)
```

```
      hit->set_fcit(Halfedge::nil_);
    Isolated_vertex_iterator vit;
    for (vit = iv_begin(); vit!=iv_end(); ++vit)
      vit->set_ivit(Vertex::nil_);
    FC.clear(); IV.clear(); }

  Hole_const_iterator fc_begin() const { return FC.begin(); }
  Hole_const_iterator fc_end() const   { return FC.end(); }
  Isolated_vertex_const_iterator iv_begin() const { return IV.begin(); }
  Isolated_vertex_const_iterator iv_end() const   { return IV.end(); }

  void set_halfedge(Halfedge_handle h)   { _e = h; }
  Halfedge_handle       halfedge()       { return _e; }
  Halfedge_const_handle halfedge() const { return _e; }

  Mark& mark() { return _m; }
  const Mark& mark() const { return _m; }

  GenPtr&       info()       { return _i; }
  const GenPtr& info() const { return _i; }

  LEDA_MEMORY(Face)
 };
};
```

9.1.3 Extended Halfedges

Halfedge objects extend the default concepts:

topology and combinatorics as in the default concept [*set_*]*next*, [*set_*]*prev*, [*set_*]*vertex*,[*set_*]*face*

geometry indirect via embedding of vertices.

attribute a mark of type *Mark* and a flexible usable information slot stored via a *GenPtr*. The latter can be used by the *geninfo* class.

Internal implementation invariants:

- *is_hole_entry*( ) $\Leftrightarrow$ *fcit*( ) != *nil_* the iterator link marks its role as an entry point into the face cycle of the incident face *face*( ).

⟨*halfedge base*⟩≡

```
template <typename Refs >
struct Halfedge__base {
  typedef typename Refs::Halfedge_handle Halfedge_handle;
  typedef typename Refs::Halfedge_const_handle Halfedge_const_handle;
protected:
  Halfedge_handle opp;
public:
  Halfedge_handle       opposite()                        { return opp; }
  Halfedge_const_handle opposite() const                  { return opp; }
  void                  set_opposite(Halfedge_handle h)   { opp = h; }
};
```

⟨*epm halfedges*⟩≡

```
template <typename Refs, typename Traits>
class Halfedge_wrapper { public:
  struct Halfedge {
  public:
    typedef Refs                HalfedgeDS;
    typedef Halfedge            Base;
    typedef Halfedge            Base_base;
    typedef CGAL::Tag_true      Supports_halfedge_prev;
    typedef CGAL::Tag_true      Supports_halfedge_vertex;
    typedef CGAL::Tag_true      Supports_halfedge_face;
    typedef typename Refs::Vertex_handle         Vertex_handle;
    typedef typename Refs::Vertex_const_handle   Vertex_const_handle;
    typedef typename Refs::Halfedge_handle       Halfedge_handle;
    typedef typename Refs::Halfedge_const_handle Halfedge_const_handle;
    typedef typename Refs::Face_handle           Face_handle;
    typedef typename Refs::Face_const_handle     Face_const_handle;
    typedef typename Refs::Vertex                Vertex;
    typedef typename Refs::Face                  Face;
    typedef void*                                GenPtr;
    typedef typename std::list<Halfedge_handle>::iterator fc_iterator;
    typedef typename Traits::Mark  Mark;
  protected:
    Halfedge_handle  opp, prv, nxt;
    Vertex_handle    _v;
    Face_handle      _f;
    fc_iterator      _fcit;
    Mark             _m;
    GenPtr           _i;
  public:
    Halfedge() :
      opp(),prv(),nxt(),_v(),_f(),_fcit(nil_),_m(),_i((GenPtr)0xABCD) {}
    Halfedge_handle       opposite()                          { return opp; }
    Halfedge_const_handle opposite() const                    { return opp; }
    void                  set_opposite(Halfedge_handle h)     { opp = h; }
    Halfedge_handle       prev()                              { return prv; }
    Halfedge_const_handle prev() const                        { return prv; }
    void                  set_prev(Halfedge_handle h)         { prv = h; }
    Halfedge_handle       next()                              { return nxt; }
    Halfedge_const_handle next() const                        { return nxt; }
    void                  set_next(Halfedge_handle h)         { nxt = h; }
    Vertex_handle         vertex()                       { return _v; }
    Vertex_const_handle   vertex() const                 { return _v; }
    void                  set_vertex(Vertex_handle v)    { _v = v; }
    Face_handle           face()                         { return _f; }
    Face_const_handle     face() const                   { return _f; }
    void                  set_face(Face_handle f)        { _f = f; }
    bool is_border() const { return _f == Face_handle(); }
```

```
    Mark& mark() { return _m; }
    const Mark& mark() const { return _m; }
    GenPtr&       info()       { return _i; }
    const GenPtr& info() const { return _i; }
    fc_iterator fcit() const      { return _fcit; }
    void set_fcit(fc_iterator it) { _fcit=it; }
    bool is_hole_entry() const
    { return _fcit != nil_; }
    static fc_iterator nil_;
    /* stl iterators have default construction but are only equal comparable
       when copy constructed, what a mess in the specification */
    LEDA_MEMORY(Halfedge)
  };
};
```

The file wrapper:

⟨*HDS_items.h*⟩≡
⟨*CGAL Header*⟩

```
#ifndef CGAL_HDS_ITEMS_H
#define CGAL_HDS_ITEMS_H
#include <CGAL/basic.h>
#include <list>
```

⟨*halfedge base*⟩

```
#ifndef CGAL_USE_LEDA
#define LEDA_MEMORY(t)
#endif
struct HDS_items {
```

⟨*epm vertices*⟩
⟨*epm halfedges*⟩
⟨*epm faces*⟩

```
}; // HDS_items
template <typename R,class T>
typename HDS_items::Vertex_wrapper<R,T>::Vertex::iv_iterator
HDS_items::Vertex_wrapper<R,T>::Vertex::nil_;
template <typename R,class T>
typename HDS_items::Halfedge_wrapper<R,T>::Halfedge::fc_iterator
HDS_items::Halfedge_wrapper<R,T>::Halfedge::nil_;
#endif // CGAL_HDS_ITEMS_H
```

9.2 Object Concepts

The HDS vertex base class (Vertex)

1. Types

Vertex::Point	geometric embedding
Vertex::Mark	information

2. Creation

Vertex v; constructs an uninitialized vertex concerning embedding and mark. All links are initialized by their default value.

Vertex v(const Point& p);

constructs a vertex with embedding *p* and mark *m*. All links are initialized by their default value.

3. Operations

bool	*v*.is_isolated()	returns true iff *v* is isolated, else false.
Halfedge_handle	*v*.halfedge()	returns an incident halfedge. *Precondition*: !*is_isolated*( ).
void	*v*.set_halfedge(*Halfedge_handle h*)	makes *h* the entry point into the adjacency cycle of *v*.
Face_handle	*v*.face()	returns the incident face if *is_isolated*( ).
void	*v*.set_face(*Face_handle f*)	makes *f* the incident face of *v*.
Point&	*v*.point()	returns the embedding point of *v*.
Mark&	*v*.mark()	returns the mark of *v*.
GenPtr&	*v*.info()	returns a generic information slot of *v*.

The HDS halfedge base class (Halfedge)

1. Types

Halfedge::Mark information

2. Creation

Halfedge e; constructs an uninitialized halfedge concerning embedding and mark. All links are initialized by their default value.

3. Operations

Halfedge_handle	*e*.opposite()	returns the twin of *e*.
void	*e*.set_opposite(*Halfedge_handle h*)	makes *h* the twin of *e*.
Halfedge_handle	*e*.prev()	returns the previous edge of the face cycle of *e*.
void	*e*.set_prev(*Halfedge_handle h*)	makes *h* the previous edge in the face cycle of *e*.
Halfedge_handle	*e*.next()	returns the next edge of the face cycle of *e*.

void	*e*.set_next(*Halfedge_handle h*)	makes *h* the next edge in the face cycle of *e*.
Vertex_handle	*e*.vertex( )	returns the vertex incident to the halfedge *e*.
void	*e*.set_vertex(*Vertex_handle v*)	makes *v* the vertex incident to *e*.
Face_handle	*e*.face( )	returns the face incident to the halfedge *e*.
void	*e*.set_face(*Face_handle f*)	makes *f* the face incident to *e*.
Mark&	*e*.mark( )	returns the mark of *e*.
GenPtr&	*e*.info( )	returns a generic information slot of *e*.
bool	*e*.is_hole_entry( )	returns true iff *e* is entry point into a hole face cycle of *e.face*( ).

The HDS face base class (Face)

1. Types

Face::*Mark* mark information

Face::*Hole_iterator* iterator for face cycles. Fits the concept *Halfedge_handle*.

Face::*Isolated_vertex_iterator*

iterator for isolated vertices. Fits the concept *Vertex_handle*.

Hole_const_iterator and *Isolated_vertex_const_iterator* are the non mutable versions.

2. Creation

Face f; constructs an uninitialized face with undefined mark, empty face cycle list, and empty isolated vertices list.

3. Operations

void *f*.store_fc(*Halfedge_handle h*)

stores halfedge *h* as an entry into a face cycle of *f*. Postcondition: $h \rightarrow$ *is_hole_entry*( ).

void *f*.remove_fc(*Halfedge_handle h*)

removes halfedge *h* as an entry into a face cycle of *f*. *Precondition*: $h \rightarrow$ *is_hole_entry*( ) and *h* is stored in the face cycle list of *f*. Postcondition: $!h \rightarrow$ *is_hole_entry*( ).

void *f*.store_iv(*Vertex_handle v*)

stores vertex *v* as an isolated vertex of *f*.

void *f*.remove_iv(*Vertex_handle v*)

removes vertex *v* as an isolated vertex of *f*. *Precondition*: $v \rightarrow$ *is_isolated*( ) and *v* is stored in the isolated vertices list of *f*. Postcondition: $!v \rightarrow$ *is_isolated*( ).

Hole_iterator *f*.fc_begin()

Hole_iterator *f*.fc_end()

the iterator range [*fc_begin*( ),*fc_end*( )) spans the set of interior face cycles.

Isolated_vertex_iterator f.iv_begin()

Isolated_vertex_iterator f.iv_end()

the iterator range $[iv\_begin(\ ), iv\_end(\ ))$ spans the set of isolated vertices.

There are the same iterator ranges defined for the const iterators *Hole_const_iterator*, *Isolated_vertex_const_iterator*.

void f.set_halfedge(*Halfedge_handle h*)

makes h the entry edge into the outer face cycle.

Halfedge_handle f.halfedge() returns a halfedge in the outer face cycle.

Mark& f.mark() returns the mark of f.

GenPtr& f.info() returns a generic information slot of f.

Bibliography

[Die97] R. Diestel. *Graph theory*, volume 173 of *Graduate texts in mathematics*. Springer, 1997.

[Ket99] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *CGTA: Computational Geometry: Theory and Applications*, 13, 1999.

[MNS+99] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *CGTA: Computational Geometry: Theory and Applications*, 12, 1999.

10 Generic Sweep Framework

10.1 Introduction

Why is a generic framework for line sweep algorithms useful?

generic programming and efficiency – at first we offer the possibility to create a plane sweep algorithm with a clearly structured concept. This enables a user to experiment with different realizations of data structures while at the same time keeping the outer interface of the framework. The flexibility is realized by a traits class which allows straight forward adaptation of used components. All operations used in the sweep are members of the traits class and they work on common data structures stored in the traits class.

simplification of implementation – the documentation reduces to a description of used data structures and a description of sweep operations. Additionally the user can concentrate on the real tasks of the sweep clearly separated into subtasks.

program verification – correctness checking of used data structures can easily be added.

animation support – animation support can easily be added by using an observer which has certain visualization routines attached to event hooks that are triggered during the sweep.

10.2 The manual pages of the generic sweep

10.2.1 A Generic Plane Sweep Framework (generic_sweep)

1. Definition

The data type *generic_sweep*<*T*> provides a general framework for algorithms following the plane sweep paradigm. The plane sweep paradigm can be described as follows. A vertical line sweeps the plane from left to right and constructs the desired output incrementally left behind the sweep line. The sweep line maintains knowledge of the scenery of geometric objects and stops at points where changes of this knowledge relevant for the output occur. These points are called events.

A general plane sweep framework structures the execution of the sweep into phases and provides a storage place for all data structures necessary to execute the sweep. An object *GS* of type *generic_sweep*<*T*> maintains an object of type T which generally can be used to store necessary structures. The content is totally dependent of the sweep specification and thereby varies within the application domain of the framework.

The traits class T has to provide a set of types which define the input/output interface of the sweep: the input type *INPUT*, the output type *OUTPUT*, and a geometry kernel type *GEOMETRY*.

The natural phases which determine a sweep are

```
// INITIALIZATION
initialize_structures();
check_invariants();

// SWEEP LOOP
while ( event_exists() ) {
  process_event();
  check_invariants();
  procede_to_next_event();
}

// COMPLETION
complete_structures();
check_final();
```

Initialization – initializing the data structures, ensuring preconditions, checking invariants

Sweep Loop – iterating over all events, while handling the event stops, ensuring invariants and the soundness of all data structures and maybe triggering some animation tasks.

Completion – cleaning up some data structures and completing the output.

The above subtasks are specified in the traits concept *GenericSweepTraits*.

2. Types

generic_sweep<*T*>::*TRAITS*	the traits class
generic_sweep<*T*>::*INPUT*	the input interface.
generic_sweep<*T*>::*OUTPUT*	the output container.
generic_sweep<*T*>::*GEOMETRY*	the geometry kernel.

3. Creation

generic_sweep<*T*> *PS*(*INPUT input*, *OUTPUT*& *output*, *GEOMETRY geometry* = *GEOMETRY*( ));

creates a plane sweep object for a sweep on objects determined by *input* and delivers the result of the sweep in *output*. The traits class *T* specifies the models of all types and the implementations of all methods used by *generic_sweep*<*T*>. At this point, it suffices to say that *INPUT* represents the input data type and *OUTPUT* represents the result data type. The *geometry* is an object providing object bound, geometry traits access.

generic_sweep<*T*> *PS*(*OUTPUT*& *output*, *GEOMETRY geometry* = *GEOMETRY*( ));

a simpler call of the above where *output* carries also the input.

4. Operations

void	*PS*.sweep( )	execute the plane sweep.

5. Example

A typical sweep based on *generic_sweep*<*T*> looks like the following little program:

```
typedef std::list<POINT>::const_iterator iterator;
typedef std::pair<iterator,iterator> iterator_pair;
std::list<POINT>  P; // fill input
```

```
GRAPH<POINT,LINE> G; // the output
generic_sweep<triang_sweep_traits>
  triangulation(iterator_pair(P.begin(),P.end()),G);
triangulation.sweep();
```

6. Events

To enable animation of the sweep there are event hooks inserted which allow an observer to attach certain visualization actions to them. There are four such hooks:

PS.post_init_hook(*TRAITS*&)	triggered just after initialization.
PS.pre_event_hook(*TRAITS*&)	triggered just before the sweep event.
PS.post_event_hook(*TRAITS*&)	triggered just after the sweep event.
PS.post_completion_hook(*TRAITS*&)	triggered just after the completion phase.

All of these are triggered during the sweep with the instance of the *TRAITS* class that is stored inside the plane sweep object. Thus any animation operation attached to a hook can work on that class object which maintains the sweep status.

Observing the plane sweep class (sweep_observer)

1. Definition

The data type *sweep_observer*<*GPS*, *VT*> provides an observer approach to the visualization of a sweep algorithm realized by $GPS = generic\_sweep<T>$ by means of an event mechanism. It allows to connect the events of an instance of *GPS* to the visualization operations provided by the traits class *VT*.

2. Creation

sweep_observer<*GPS*, *VT*> *Obs*;

creates an object of type *VT* which can support a visualization device to visualize a sweep object of type *GPS*.

sweep_observer<*GPS*, *VT*> *Obs*(*GPS*& *gps*);

creates an object of type *VT* which can support a visualization device to visualize a sweep object of type *GPS* and makes it an observer of *gps*.

3. Operations

void *Obs*.attach(*GPS*& *gps*)

makes *Obs* an observer of *gps*.

4. Example

A typical sweep observation based on *sweep_observer*<*GPS*, *VT*> looks like the following little program:

```
typedef generic_sweep<triang_sweep_traits> triang_sweep;
triang_sweep Ts(...);
sweep_observer< triang_sweep,
                cgal_window_stream_ts_visualization > Obs(Ts);
Ts.sweep();
```

This would visualize the sweep actions in the observer window by means of the visualization functions provided in *cgal_window_stream_ts_visualization*

10.3 The Implementation

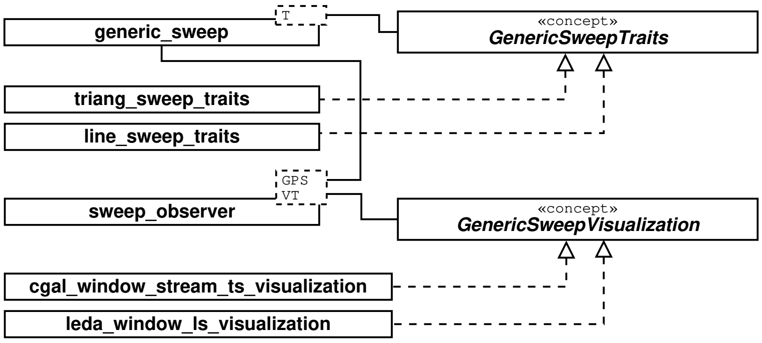


Figure 10.1: The classes and concepts of the generic plane sweep module.

The framework is based on a class template that can be adapted by means of the traits class T. The traits class is specified via the concept *GenericSweepTraits*.

⟨*generic_sweep.h*⟩≡
    ⟨*CGAL Header1*⟩
```
// file          : include/CGAL/generic_sweep.h
```
    ⟨*CGAL Header2*⟩

⟨*generic_sweep.h*⟩+≡
```
#ifndef CGAL_GENERIC_SWEEP_H
#define CGAL_GENERIC_SWEEP_H

#include <CGAL/sweep_observer.h>

namespace CGAL {

template <typename T>
class generic_sweep {

typedef generic_sweep<T>  Self;

T traits;

public :

typedef T TRAITS;

typedef typename TRAITS::INPUT  INPUT;

typedef typename TRAITS::OUTPUT OUTPUT;

typedef typename TRAITS::GEOMETRY GEOMETRY;
```
⟨*animation events*⟩
```
generic_sweep(const INPUT& input, OUTPUT& output,
  const GEOMETRY& geometry = GEOMETRY()) :
  traits(input,output,geometry) {}

generic_sweep(OUTPUT& output, const GEOMETRY& geometry = GEOMETRY()) :
```

```
  traits(output,geometry) {}

void sweep()
{
  traits.initialize_structures();
  traits.check_invariants();
      post_init_hook(traits);
  while ( traits.event_exists() ) {
        pre_event_hook(traits);
    traits.process_event();
        post_event_hook(traits);
    traits.check_invariants();
    traits.procede_to_next_event();
  }
  traits.complete_structures();
  traits.check_final();
      post_completion_hook(traits);
}

}; // generic_sweep<T>

} // namespace CGAL

#endif // CGAL_GENERIC_SWEEP_H
```

10.3.1 Animation by Events

In the public scope of *generic_sweep* there are four event hooks accessible. These event hooks correspond to the states of a standard sweep algorithm.

⟨*animation events*⟩≡

```
Event_hook<TRAITS&>   post_init_hook;
Event_hook<TRAITS&>   pre_event_hook;
Event_hook<TRAITS&>   post_event_hook;
Event_hook<TRAITS&>   post_completion_hook;
```

The rest of this section implements the modules necessary for an observer class of the sweep class. We implement the event hooks that maintain lists of clients representing observers. A client implements the interface to an observer and its event method. Such a client can be attached to an event such that it gets informed when the event occurs. The event hook just knows an anonymous interface for each of its clients. Therefore the message passed from the event hook is decoubled from the observing module. At the end we provide an observer class that can be used to visualize any sweep algorithm based on the generic plane sweep framework.

⟨*sweep_observer.h*⟩≡
⟨*CGAL Header1*⟩

```
// file          : include/CGAL/sweep_observer.h
```

⟨*CGAL Header2*⟩

```
#ifndef CGAL_SWEEP_OBSERVER_H
#define CGAL_SWEEP_OBSERVER_H
```

```
#include <list>
namespace CGAL {
⟨virtual client base class⟩
⟨event hook class template⟩
⟨client class template⟩
⟨attach operation⟩
⟨sweep observer class template⟩
} // namespace CGAL
#endif // CGAL_SWEEP_OBSERVER_H
```

All clients are derived from a virtual base class while overloading a *call* method.

⟨*virtual client base class*⟩≡

```
// TR is traits
template <typename TR>
class client_base
{
public:
  virtual void call(TR) const = 0;
};
```

An event hook stores a list of clients that have to be notified when the event occurs. The template typename *TR* is the type of information that will be transmitted when the event occurs.

⟨*event hook class template*⟩≡

```
template <typename TR>
class Event_hook
{
  typedef client_base<TR>* p_client_base;
  typedef std::list< p_client_base > clientlist;
protected:
  clientlist clients;
public:
  Event_hook() : clients() {}
  ~Event_hook() {
    while ( !clients.empty() ) {
      delete (*clients.begin());
      clients.pop_front();
    }
  }
  void operator()(TR t) const
  { if ( clients.empty() ) return;
    for ( clientlist::const_iterator it=clients.begin();
          it != clients.end(); ++it )
      (*it)->call(t);
  }
  void attach(p_client_base psb)
  { clients.push_back(psb); }
};
```

The client template is derived from the client base. It is instantiated with an observer type *OBS* and a traits type *TR*. It stores the method reference *void* $(OBS::* p\_fnc)(TR)$ and an observer reference *obs_ref*. When its vitual call method is invoked the call is forwarded to the observers method. An object of type *TR* carries the event information.

⟨*client class template*⟩≡
```
// TR is traits, OBS is observer
template <class OBS, class TR>
class client : public client_base<TR>
{
protected:
  OBS& obs_ref;
  void (OBS::* p_fnc)(TR);
  // pointer to member of Observer which works on an object of type TR
public:
  client( OBS& obs, void (OBS::* p_fnc_init)(TR)  ) :
    obs_ref(obs), p_fnc(p_fnc_init)  {}
  void call(TR t) const
  { (obs_ref.*p_fnc)(t); }
};
```

The function template *attach* just attaches a new client object based on an observer-method pair to an event hook.

⟨*attach operation*⟩≡
```
template <class OBS, class TR>
inline void attach(Event_hook<TR>& h,
       OBS& obs, void (OBS::* p_fct)(TR t))
{
  client<OBS,TR>* ps = new client<OBS,TR>(obs,p_fct);
  h.attach( (client_base<TR>*) ps);
}
```

The following sweep observer implements a unified visualization module of a generic plane sweep instance of type *GPS*. The *VT* traits class allows to connect the observer to a visualization device (like e.g. LEDA windows).

⟨*sweep observer class template*⟩≡
```
template <class GPS, class VT>
class sweep_observer  {
  VT vt;
  typedef typename GPS::TRAITS GPSTRAITS;
  typedef typename VT::VDEVICE VDEVICE;
  void post_init_animation(GPSTRAITS& gpst)
  { vt.post_init_animation(gpst); }
  void pre_event_animation(GPSTRAITS& gpst)
  { vt.pre_event_animation(gpst); }
  void post_event_animation(GPSTRAITS& gpst)
  { vt.post_event_animation(gpst); }
```

```
  void post_completion_animation(GPSTRAITS& gpst)
  { vt.post_completion_animation(gpst); }
public :
  sweep_observer() : vt() {}
  sweep_observer(GPS& gps);
   void attach(GPS& gps);
   VDEVICE& device()
   { return vt.device(); }
};

template <class GPS, class VT>
sweep_observer<GPS,VT>::
sweep_observer(GPS& gps) : vt() { attach(gps); }
template <class GPS, class VT>
void
sweep_observer<GPS,VT>::attach(GPS& gps)
{
  CGAL::attach(gps.post_init_hook, *this,
        &sweep_observer::post_init_animation);
  CGAL::attach(gps.pre_event_hook, *this,
        &sweep_observer::pre_event_animation);
  CGAL::attach(gps.post_event_hook, *this,
        &sweep_observer::post_event_animation);
  CGAL::attach(gps.post_completion_hook, *this,
        &sweep_observer::post_completion_animation);
}
```

⟨*sweep observer class template*⟩+≡

```
#ifdef THIS_IS_JUST_A_CONCEPT_DEFINITION
class GenericSweepVisualization {
typedef some_visualization_device VDEVICE;
GenericSweepVisualization();
void post_init_animation(GPS::TRAITS& gpst)
void pre_event_animation(GPS::TRAITS& gpst)
void post_event_animation(GPS::TRAITS& gpst)
void post_completion_animation(GPS::TRAITS& gpst)
VDEVICE& device()
};
#endif // THIS_IS_JUST_A_CONCEPT_DEFINITION
```

10.4 Simple Applications

10.4.1 An Empty Sweep Traits Class Model

⟨*empty_sweep.h*⟩≡

```
#ifndef EMPTY_SWEEP_H
#define EMPTY_SWEEP_H

class empty_definition {};
#define USERTYPE empty_definition

/* fill in the following typedef statements HERE by your types */

class empty_sweep_traits {
public:
  typedef USERTYPE INPUT;
  typedef USERTYPE OUTPUT;
  typedef USERTYPE GEOMETRY;

  /* here a list of local data structures like
     a priority queue of type EVENT_STRUCTURE or
     a sorted sequence for maintaining the sweepline
     of type SWEEP_STATUS_STRUCTURE */

empty_sweep_traits(const INPUT& in, OUTPUT& out,
  const GEOMETRY& geom) {}

empty_sweep_traits(OUTPUT& out, const GEOMETRY& geom) {}

void initialize_structures() {}
bool event_exists()          {}
void procede_to_next_event() {}
void process_event()         {}
void complete_structures()   {}
void check_invariants()      {}
void check_final()           {}

};

#endif // EMPTY_SWEEP_H
```

A Debug Observer

Another application of the observer pattern concerning the line segment intersection sweep:

```
typedef PMO_from_pm<Self,Seg_iterator,Seg_info> PMO;
typedef PM_segment_overlay_traits<Seg_iterator,PMO,GEOM> pm_overlay;
typedef generic_sweep< pm_overlay > pm_overlay_sweep;
PMO Out(*this,&PI[0],&PI[1],From);
pm_overlay_sweep SOS(Seg_it_pair(Segments.begin(),Segments.end()),Out,K);
Debug_observer<pm_overlay_sweep> Obs(SOS);
SOS.sweep();
```

⟨*Sweep_debug_observer.h*⟩=

```
#ifndef CGAL_SWEEP_DEBUG_OBSERVER_H
#define CGAL_SWEEP_DEBUG_OBSERVER_H

#include <CGAL/Hash_map.h>

template <class GPS>
```

```
class Sweep_debug_observer  {
public :
Sweep_debug_observer() { i=0; n=0; }
Sweep_debug_observer(GPS& gps) { attach(gps); i=0;n=0; }
void attach(GPS& gps)
{
  ::attach(gps.post_init_hook, *this,
        &Debug_observer::post_init_animation);
  ::attach(gps.pre_event_hook, *this,
        &Debug_observer::pre_event_animation);
  ::attach(gps.post_event_hook, *this,
        &Debug_observer::post_event_animation);
  ::attach(gps.post_completion_hook, *this,
        &Debug_observer::post_completion_animation);
}
protected:
typedef typename GPS::TRAITS GPSTRAITS;
typedef typename GPSTRAITS::Vertex_handle Vertex_handle;
typedef typename GPSTRAITS::SegQueue SegQueue;
typedef typename SegQueue::item pq_item;
CGAL::Hash_map<Vertex_handle,int> M;
int i,n;
void post_init_animation(GPSTRAITS& gpst)
{
  std::cerr << "SQ= \n";
  pq_item pqit;
  forall_items(pqit,gpst.SQ) {
    //if ( gpst.SQ.prio(pqit)== gpst.XS.key(gpst.XS.min_item()))
    std::cerr << gpst.SQ.prio(pqit) << gpst.SQ.inf(pqit) << std::endl;
  }
  std::cerr << std::endl;
}
void post_completion_animation(GPSTRAITS& gpst) {}
void pre_event_animation(GPSTRAITS& gpst) {
  std::cout << "VERTEX " << i << std::endl;
  if (i == n) { std::cin >> n; }
}
void post_event_animation(GPSTRAITS& gpst)
{ Vertex_handle v = --gpst.GO.G.vertices_end();
  M[v] = i;
  i++;
}
}; // Sweep_debug_observer
#endif // CGAL_SWEEP_DEBUG_OBSERVER_H
```

10.4.2 A triangulation model for CGAL

Our first example is the sweep algorithm to triangulate a set of points. The *INPUT* is a list of points that have to be triangulated. We define the input interface to be a pair of iterators along the lines of the STL. The *OUTPUT* is a plane map that represents the triangulation of the points. We store the constructed triangulation in a halfedge data structure. The *GEOMETRY* kernel is the CGAL 2d Cartesian kernel.

The only event to handle is the access of a new point which has to be added as a vertex of the triangulation left of the sweep line. The sweep status is implicitly stored in the output structure. We use the lexicographical maximal node as a reference into the convex hull of the up to the current event point *p_sweep* calculated structure. From this visible node we walk the convex hull to find all edges visible from *p_sweep* and update the hull and the triangulation accordingly.

For the event structure we use a plain list of points, as there is no new event creation necessary during the sweep.

⟨*triang_sweep.h*⟩≡

```
#ifndef TRIANG_SWEEP_H
#define TRIANG_SWEEP_H

#include <CGAL/basic.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/predicate_classes_2.h>
#include <CGAL/predicates_on_points_2.h>
#include <CGAL/predicate_objects_on_points_2.h>
#include <CGAL/Halfedge_data_structure_default.h>
#include <CGAL/IO/Window_stream.h>
#include "PM_decorator_simple.h"

#undef _DEBUG
#define _DEBUG 5
#include <LOCAL/debug.h>
#include <assert.h>
#include <list>
#include <vector>

class triang_sweep_traits {
public:
  typedef double                        COORD;
  typedef CGAL::Cartesian<COORD>        GEOMETRY;
  typedef GEOMETRY::Point_2             POINT;
  typedef GEOMETRY::Direction_2         DIRECTION;
  typedef GEOMETRY::Segment_2           SEGMENT;

  typedef std::list<POINT> PLIST;
  typedef PLIST::const_iterator         INPUT_ITERATOR;
  typedef std::pair<INPUT_ITERATOR,INPUT_ITERATOR>
                                        INPUT;
  typedef CGAL::Halfedge_data_structure_default<POINT>
                                        OUTPUT;
  typedef CGAL::PM_decorator_simple<OUTPUT>
                                        PMDecorator;

  typedef OUTPUT::Halfedge_iterator Halfedge_iterator;
  typedef PMDecorator::Vertex    Vertex;
  typedef PMDecorator::Halfedge  Halfedge;
  typedef Vertex*                v_handle;
  typedef Halfedge*              e_handle;

  const GEOMETRY&          geom;
```

```
  INPUT_ITERATOR          its,ite;
  OUTPUT&                 triangulation;
  PLIST                   sorted_points;
  PLIST::const_iterator   event;
  POINT                   p_sweep;
  v_handle                v_last;
  PMDecorator             D;
  triang_sweep_traits(const INPUT& in, OUTPUT& out, const GEOMETRY& g) :
     geom(g), its(in.first), ite(in.second), triangulation(out),
     D(triangulation) {}
  ⟨ts initialization⟩
  ⟨ts event handling⟩
  ⟨ts cleaning up⟩
  ⟨ts checking⟩
};
#include <CGAL/generic_sweep.h>
typedef CGAL::generic_sweep<triang_sweep_traits> triang_sweep;
⟨cgal window stream ts visualization⟩
#endif // TRIANG_SWEEP_H
```

The working procedures of the sweep

The initialization of the event structure is trivial. We copy the input list into *sorted_points*, sort it, and remove all coordinate identical points up to one. The post conditions are:

- *v_last* stores the maximal node of *triangulation*
- either *triangulation* contains only one vertex when all points in *sorted_points* are equal or there's an edge adjacent to *v_last* allowing a visibility search along the convex hull chain.
- *event* points to the next item in *sorted_points*

⟨*ts initialization*⟩≡

```
void initialize_structures()
{
  if ( its==ite ) return;
  // Initialize sorted_points by sorted input_points,
  // remove doubles from sorted sequence
  GEOMETRY::Less_xy_2 _less_xy =
    geom.less_xy_2_object();
  sorted_points.insert(sorted_points.end(),its,ite);
  sorted_points.sort(_less_xy);
  sorted_points.unique();
  event = sorted_points.begin();
  // the output graph triangulation is initially empty
  if ( sorted_points.size() == 1 ) {
    v_last = D.new_vertex(*event++);
  } else { // event != sorted_points.end()
    v_handle v1 = D.new_vertex(*event++);
```

```
    v_last =      D.new_vertex(*event++);
    D.new_halfedge_pair(v1,v_last);
  }
}
bool event_exists()
{
  if ( event != sorted_points.end() ) {
    p_sweep = *event;
    return true;
  }
  return false;
}
void procede_to_next_event()
{ ++event; }
```

Before handling the next event at *p_sweep* all data structures are correct in the sense that the output graph *G* contains all vertices and edges that are left of *p_sweep* which build a bidirected triangulation of all points.

The method *process_event* realizes our event handling. It executes the changes that are neccessary to correct all data structures and keep all invariants when crossing *p_sweep* by the sweep line.

1. use *v_last* as an entry point into the triangulation. Note that all adjacency lists are kept ordered such that the embedding of the plane map is counter clockwise s are kept counter clockwise order preserving. Then the last edge of the adjacency list of *v_last* lies on the convex hull of all up to now handled points.
2. find the range of vertices and edges on the eastern convex hull which is visible from *p_sweep*.
3. connect a new vertex at *p_sweep* to all the vertices in the range.
4. update *v_last*.

⟨*ts event handling*⟩≡

```
SEGMENT seg(e_handle e) const
{ GEOMETRY::Construct_segment_2 _segment =
    geom.construct_segment_2_object();
  return _segment(D.source(e)->point(),
                  D.target(e)->point()); }
inline bool edge_is_visible(e_handle e, const POINT& p) const
{
  POINT p1 = D.source(e)->point();
  POINT p2 = D.target(e)->point();
  GEOMETRY::Leftturn_2 _leftturn =
    geom.leftturn_2_object();
  return _leftturn(p1,p2,p);
}
/* returns the first visible edge in the clockwise chain of convex hull
   edges of G viewed from p. precondition: start edge is visible. */
void move_ccw_to_visibility_chain_start(e_handle& e, const POINT& p)
  const
```

```
{ assert(edge_is_visible(e,p));
  e_handle e_start = e;
  do {
    e = e->prev();
  } while ( e != e_start && edge_is_visible(e,p) );
  e = e->next();
}

void process_event()
{
  // only called for event and p_sweep set
  // v_last contains the last vertex of triangulation;
  e_handle e = D.last_out_edge(v_last);
  v_handle v_new = D.new_vertex(p_sweep);

  if ( edge_is_visible(e,p_sweep) )
    move_ccw_to_visibility_chain_start(e,p_sweep);
  else if ( edge_is_visible(e->prev(),p_sweep) ) {
    e = e->prev();
    move_ccw_to_visibility_chain_start(e,p_sweep);
  }
  else {
    /* special case when both edges are not visible means
       p_sweep is collinear with e and e->prev() */
    D.new_halfedge_pair(v_new,v_last);
    v_last = v_new;
    return;
  }

  // now we have the visibility chain starting edge in e
  e_handle ee = e;
  do {
    D.new_halfedge_pair(e,v_new);
    if ( !edge_is_visible(e,p_sweep) ) break;
      /* we insert all new edges to the sources of e after the previous
         convex hull edge; we embed the adjacency lists counterclockwise,
         thus we append all edges of v_new counterclockwise */
    e = e->next();
  } while (e != ee);
  v_last = v_new;
}
```

The plane map is a full triangulation after each event. So there's nothing to be done here.

⟨*ts cleaning up*⟩≡

```
void complete_structures() {}
```

⟨*ts checking*⟩≡

```
void check_invariants() {}
void check_final() {}
```

Animating the triangulation sweep

For the animation we provide animation operations for the event hooks defined by the sweep class which are drawn via a CGAL window stream. We provide a traits class knowing the triangulation traits class and its slots and attached to the animation of the sweep via the sweep observer defined above.

⟨*cgal window stream ts visualization*⟩≡

```
class cgal_window_stream_ts_visualization {
  CGAL::Window_stream W;
public:
typedef CGAL::Window_stream VDEVICE;
cgal_window_stream_ts_visualization() : W()
{
  W.set_node_width(2);
  W.set_grid_mode(1);
  W.set_show_coordinates(true);
  W.display();
}
VDEVICE& device() { return W; }
void post_init_animation(triang_sweep_traits& t)
{ W << CGAL::BLACK << t.triangulation.vertices_begin()->point()
    << t.v_last->point() << t.seg(&*(t.triangulation.halfedges_begin()));
}
void pre_event_animation(triang_sweep_traits& t)
{
  W << CGAL::RED << t.p_sweep;
  // only called for event and p_sweep set
  // v_last contains the last vertex of triangulation;
  triang_sweep_traits::e_handle
    e(t.D.last_out_edge(t.v_last));
  if ( e == 0 ) return;
  if ( t.edge_is_visible(e,t.p_sweep) )
    t.move_ccw_to_visibility_chain_start(e,t.p_sweep);
  else if ( t.edge_is_visible(e->prev(),t.p_sweep) ) {
    e = e->prev();
    t.move_ccw_to_visibility_chain_start(e,t.p_sweep);
  } else {
    W.read_mouse();return;
  }
  triang_sweep_traits::e_handle eend(e);
  do {
    if ( !t.edge_is_visible(e,t.p_sweep) ) break;
    W << CGAL::BLUE << t.seg(e);
    e = e->next();
  } while (e != eend);
  W.read_mouse();
}
void post_event_animation(triang_sweep_traits& t)
{ W << CGAL::BLACK << t.p_sweep;
  triang_sweep_traits::e_handle
```

```
    e(t.D.last_out_edge(t.v_last)),eend(e);
  if ( e == 0 ) return;
  do {
    W << CGAL::BLACK << t.seg(e)
      << t.seg(e->next());
    e = t.D.cap(e);
  } while (e != eend);
}
void post_completion_animation(triang_sweep_traits& t)
{
  triang_sweep_traits::Halfedge_iterator
    eit = t.triangulation.halfedges_begin();
  while ( eit != t.triangulation.halfedges_end() )
    W << t.seg(&*eit++);
  W.read_mouse();
}
};
```

A Test of the Triangulation Sweep

The whole program then looks as follows.

⟨*triang-test.C*⟩≡

```
#include <CGAL/basic.h>
#include "triang_sweep.h"
#include <fstream.h>
#include <strstream.h>
#include <LEDA/gw_observer.h>
typedef triang_sweep::TRAITS TTRAITS;
typedef TTRAITS::POINT  POINT;
typedef TTRAITS::OUTPUT HDS;
int main(int argc, char* argv[])
{
  CGAL::set_pretty_mode(cerr);
  CGAL::sweep_observer<triang_sweep,
    cgal_window_stream_ts_visualization> SO;
  HDS G;
  std::list<POINT> L;
  POINT p;
  if (argc==2) {
    ifstream from(argv[1]);
    while (from >> p) L.push_back(p);
  }
  ostrstream fname;
  fname << argv[0] << ".log" << '\0';
  ofstream to(fname.str()); CGAL_assertion(to);
  for(std::list<POINT>::const_iterator lit=L.begin();
      lit!=L.end(); ++lit)
    to << *lit << endl;
```

```
    while (SO.device() >> p) {
      L.push_back(p);
    }
    to.close();
    triang_sweep TS(triang_sweep::INPUT(L.begin(),L.end()),G);
    SO.attach(TS);
    TS.sweep();
    return 0;
  }
```

10.4.3 A line intersection model for LEDA

We list the intersection points of a set of lines. The idea is easy. We sweep the plane with a vertical line from $x = -\infty$ to $x = \infty$ and report all intersection points of the lines in the order encountered during the sweep. The y-structure is a sorted sequence with the lines as its key type. The x-structure is a sorted sequence of a subset of the intersection points — namely those which refer to two lines which are neighbors in the y-structure. As an optimization each intersection point knows the item of the lower line (that with a greater slope) within the y-structure. Note that for simplification we just consider non-degenerate positions, thus we demand that each intersection event is a point which lies on only two lines.

The initialization is just the insertion of all lines according to their slope into the y-structure and the insertion of the intersection events of each neighbor pair.

What are we doing at an event? We report the intersection, then swap the position of the two lines causing it within the y-structure and ensure our invariant that the intersection event of the new neighbors within the y-structure are in the x-structure.

⟨*line_sweep.h*⟩≡

```
  #ifndef LINE_SWEEP_H
  #define LINE_SWEEP_H
  #define LEDA_STL_ITERATORS
  #undef _DEBUG
  #define _DEBUG 2
  #include <LOCAL/debug.h>
  #include <LEDA/list.h>
  #include <LEDA/line.h>
  #include <LEDA/sortseq.h>
  #include <assert.h>
  static int cmp_lines(const leda_line& l1,
                       const leda_line& l2)
  { return -cmp_slopes(l1,l2); }
  class line_sweep_traits {
  public:
    typedef leda_list<leda_line>  INPUT;
    typedef leda_list<leda_point> OUTPUT;
    typedef void*                 GEOMETRY;
    const INPUT&                        line_list;
    OUTPUT&                             intersection_points;
    leda_sortseq<leda_line,void*>       ordered_lines;
```

```
  leda_sortseq<leda_point,seq_item>   event_queue;
  seq_item                            event;
  leda_point                          p_sweep;
  line_sweep_traits(const INPUT& in, OUTPUT& out, const GEOMETRY&) :
     line_list(in), intersection_points(out),
     ordered_lines(cmp_lines) {}
  ⟨ls procedures⟩
};
#include <CGAL/generic_sweep.h>
typedef CGAL::generic_sweep<line_sweep_traits> line_sweep;
⟨leda_window_ls_visualization⟩
#endif // LINE_SWEEP_H
```

The working procedures of the sweep

⟨ls procedures⟩≡

```
void compute_intersection( seq_item yit1 )
{ if ( !yit1 ) return;
  seq_item yit2 = ordered_lines.succ(yit1);
  if ( !yit2 ) return;
  leda_line l1 = ordered_lines.key(yit1);
  leda_line l2 = ordered_lines.key(yit2);
  leda_point p;
  if ( cmp_slopes(l1,l2) > 0 ) {
    l1.intersection(l2,p);
    event_queue.insert(p,yit1);
  }
}
void initialize_structures()
{
  leda_line l;
  forall(l,line_list)
  { ordered_lines.insert(l,0); }
  seq_item lit;
  forall_items(lit,ordered_lines)
  { compute_intersection(lit); }
  event = event_queue.min();
}
bool event_exists()
{
  if ( event ) {
    p_sweep = event_queue.key(event);
    return true;
  }
  return false;
}
void procede_to_next_event()
{ event = event_queue.succ(event); }
void process_event()
```

```
{
  // only called for event and p_sweep set
  intersection_points.push_back(p_sweep);
  seq_item yit1 = event_queue.inf(event),
           yit2 = ordered_lines.succ(yit1);
  assert(yit2);
  ordered_lines.reverse_items(yit1,yit2);
  compute_intersection(yit1);
  compute_intersection( ordered_lines.pred(yit2) );
}
void complete_structures() {}
void check_invariants() {}
void check_final() {}
```

Animating the Sweep

For the animation we provide animation operations for the event hooks defined by the sweep class which are drawn in a LEDA window. We provide a traits class knowing the triangulation traits class and its slots and attached to the animation of the sweep via the sweep observer defined above.

⟨*leda_window_ls_visualization*⟩≡

```
#include <LEDA/window.h>
class leda_window_ls_visualization {
  leda_window W;
public:
typedef leda_window VDEVICE;
leda_window_ls_visualization() : W()
{ W.set_mode(leda_xor_mode);
  W.set_grid_mode(1);
  W.set_node_width(4);
  W.set_show_coordinates(true);
  W.display();
}
VDEVICE& device() { return W; }
void post_init_animation(line_sweep_traits& t)
{
  W.clear();
  seq_item it;
  forall_items(it,t.ordered_lines)
    W.draw_line(t.ordered_lines.key(it),leda_black);
}
void pre_event_animation(line_sweep_traits& t)
{
  leda_line l1,l2;
  seq_item i1 = t.event_queue.inf(t.event);
  seq_item i2 = t.ordered_lines.succ(i1);
  W.draw_line(t.ordered_lines.key(i1),leda_yellow);
  W.draw_line(t.ordered_lines.key(i2),leda_yellow);
```

```
    W.draw_filled_node(t.p_sweep,leda_red);
    W.read_mouse();
  }
  void post_event_animation(line_sweep_traits& t)
  {
    leda_line l1,l2;
    seq_item i1 = t.event_queue.inf(t.event);
    seq_item i2 = t.ordered_lines.pred(i1);
    W.draw_filled_node(t.p_sweep,leda_red);
    W.draw_line(t.ordered_lines.key(i1),leda_yellow);
    W.draw_line(t.ordered_lines.key(i2),leda_yellow);
  }
  void post_completion_animation(line_sweep_traits& t)
  {
    leda_point p;
    forall(p,t.intersection_points) W.draw_filled_node(p,leda_blue);
    W.read_mouse();
  }
  };
```

A Test of the Line Sweep

The whole program then looks as follows.

⟨*line-test.C*⟩≡

```
  #include "line_sweep.h"
  #include <fstream.h>
  #include <strstream.h>
  int main(int argc, char* argv[])
  {
    CGAL::sweep_observer<line_sweep,leda_window_ls_visualization> SO;
    leda_list<leda_line> L;
    leda_list<leda_point> S;
    leda_point p,q;
    while ( true ) {
      if (SO.device().read_mouse(p) == MOUSE_BUTTON(3)) break;
      if (SO.device().read_mouse_seg(p,q) == MOUSE_BUTTON(3)) break;
      SO.device() << leda_line(p,q);
      L.push_back(leda_line(p,q));
    }
    line_sweep LS(L,S);
    SO.attach(LS);
    LS.sweep();
    return 0;
  }
```

⟨*empty_sweep.c*⟩≡

```
  #include <CGAL/generic_sweep.h>
  #include "empty_sweep.h"
```

```
int main(int argc, char* argv[])
{
  empty_definition E;
  CGAL::generic_sweep< empty_sweep_traits > G(E,E,E);
  G.sweep();
  return 0;
}
```

11 Appendix - Specification Pages

11.1 Concepts

11.1.1 Geometry for segment overlay (SegmentOverlayGeometry_2)

1. Definition

SegmentOverlayGeometry_2 is a kernel concept providing affine geometry for the overlay of segment. The concept specifies geometric types, predicates, and constructions.

2. Types

Local types are *Point_2*, *Segment_2*, the ring type *RT*, and the field type *FT*. See the CGAL 2d kernel for a description of *RT* and *FT*.

3. Creation

The kernel must be default and copy constructible. Let *K* be an object of type *SegmentOverlayGeometry_2*.

4. Operations

Point_2	*K*.source(*Segment_2 s*)	returns the source point of *s*.
Point_2	*K*.target(*Segment_2 s*)	returns the target point of *s*.
bool	*K*.is_degenerate(*Segment_2 s*)	return true iff *s* is degenerate.
int	*K*.compare_xy(*Point_2 p1*, *Point_2 p2*)	returns the lexicographic order of *p1* and *p2*.
Segment_2	*K*.construct_segment(*Point_2 p*, *Point_2 q*)	constructs a segment *pq*.
int	*K*.orientation(*Segment_2 s*, *Point_2 p*)	returns the orientation of *p* with respect to the line through *s*.
Point_2	*K*.intersection(*Segment_2 s1*, *Segment_2 s2*)	returns the point of intersection of the lines supported by *s1* and *s2*. The algorithm asserts that this intersection point exists.

11.1.2 Output traits for segment overlay (SegmentOverlayOutput)

1. Definition

This is the plane map decorator concept for the *PMDEC* template parameter of *PM_seg_overlay_traits*.

2. Types

SegmentOverlayOutput::*Vertex_handle*

the vertex handle.

SegmentOverlayOutput::*Halfedge_handle*

the halfedge handle.

SegmentOverlayOutput::*Point_2*

embedding type. *Precondition*: *Point_2* equals *GEOM*::*Point_2*.

3. Creation

Let *G* be an object of type *SegmentOverlayOutput*.

4. Operations

Vertex_handle *G*.new_vertex(*Point_2 p*) — creates a new vertex in the output structure and embeds it via the point *p*.

void *G*.link_as_target_and_append(*Vertex_handle v*, *Halfedge_handle e*)

makes *v* the target of *e* and appends the twin of *e* (its reversal edge) to *v*'s adjacency list.

Halfedge_handle *G*.new_halfedge_pair_at_source(*Vertex_handle v*)

returns a newly created edge inserted before the first edge of the adjacency list of *v*. It also creates a reversal edge whose target is *v*.

Additional sweep information

The iterator type *ITERATOR* has to be the same type as the first type parameter of *Segment_overlay_traits*.

void *G*.supporting_segment(*Halfedge_handle e*, *ITERATOR it*)

the non-trivial segment $*it$ supports the edge *e*.

void *G*.trivial_segment(*Vertex_handle v*, *ITERATOR it*)

the trivial segment $*it$ supports vertex *v*.

void *G*.halfedge_below(*Vertex_handle v*, *Halfedge_handle e*)

associates the edge *e* as the edge below *v*.

void *G*.starting_segment(*Vertex_handle v*, *ITERATOR it*)

the segment $*it$ starts in *v*.

void *G*.passing_segment(*Vertex_handle v*, *ITERATOR it*)

the segment $*it$ passes *v* (contains it in its relative interior) .

void *G*.ending_segment(*Vertex_handle v*, *ITERATOR it*)

the segment $*it$ ends in *v*.

11.1.3 Geometry for plane map overlay (OverlayerGeometry_2)

1. Definition

OverlayerGeometry_2 is a kernel concept providing affine geometry for the overlay of plane maps. The concept specifies geometric types, predicates, and constructions. This concept generalizes the concept *SegmentOverlayerGeometry_2*

2. Types

Local types are *Point_2*, *Segment_2*, the ring type *RT*, and the field type *FT*. See the CGAL 2d kernel for a description of *RT* and *FT*.

3. Creation

The kernel must be default and copy constructible.

4. Operations

Point_2	*K*.source(*Segment_2 s*)	returns the source point of *s*.
Point_2	*K*.target(*Segment_2 s*)	returns the target point of *s*.
bool	*K*.is_degenerate(*Segment_2 s*)	return true iff *s* is degenerate.
int	*K*.compare_xy(*Point_2 p1*, *Point_2 p2*)	returns the lexicographic order of *p1* and *p2*.
Segment_2	*K*.construct_segment(*Point_2 p*, *Point_2 q*)	constructs a segment *pq*.
int	*K*.orientation(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*)	returns the orientation of *p3* with respect to the line through *p1p2*.
int	*K*.orientation(*Segment_2 s*, *Point_2 p*)	returns the orientation of *p* with respect to the line through *s*.
bool	*K*.leftturn(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*)	return true iff the *p3* is left of the line through *p1p2*.
Point_2	*K*.intersection(*Segment_2 s1*, *Segment_2 s2*)	returns the point of intersection of the lines supported by *s1* and *s2*. The algorithm asserts that this intersection point exists.

11.1.4 An affine kernel traits (AffineGeometryTraits_2)

1. Definition

AffineGeometryTraits_2 is a kernel concept providing affine geometry. The concept specifies geometric types, predicates, and constructions.

2. Types

Local types are *Point_2*, *Segment_2*, *Direction_2*, *Line_2*, the ring type *RT*, and the field type *FT*. See the CGAL 2d kernel for a description of *RT* and *FT*.

3. Creation

The kernel must be default and copy constructible.

4. Operations

Point_2	*K*.source(*Segment_2 s*)	returns the source point of *s*.
Point_2	*K*.target(*Segment_2 s*)	returns the target point of *s*.
int	*K*.orientation(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*)	returns the orientation of *p3* with respect to the line through *p1p2*.
bool	*K*.leftturn(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*)	return true iff the *p3* is left of the line through *p1p2*.
bool	*K*.is_degenerate(*Segment_2 s*)	return true iff *s* is degenerate.

int *K*.compare_xy(*Point_2 p1*, *Point_2 p2*)

returns the lexicographic order of *p1* and *p2*.

int *K*.compare_x(*Point_2 p1*, *Point_2 p2*)

returns the order on the *x*-coordinates of *p1* and *p2*.

int *K*.compare_y(*Point_2 p1*, *Point_2 p2*)

returns the order on the *y*-coordinates of *p1* and *p2*.

bool *K*.first_pair_closer_than_second(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*, *Point_2 p4*)

returns *true* iff the distance *p1p2* is smaller than the distance *p3p4*.

bool *K*.strictly_ordered_along_line(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*)

returns *true* iff *p2* is in the relative interior of the segment *p1p3*.

Segment_2 *K*.construct_segment(*Point_2 p*, *Point_2 q*)

constructs a segment *pq*.

int *K*.orientation(*Segment_2 s*, *Point_2 p*)

returns the orientation of *p* with respect to the line through *s*.

bool *K*.contains(*Segment_2 s*, *Point_2 p*)

returns true iff *s* contains *p*.

Point_2 *K*.intersection(*Segment_2 s1*, *Segment_2 s2*)

returns the point of intersection of the lines supported by *s1* and *s2*. The algorithm asserts that this intersection point exists.

Direction_2 *K*.construct_direction(*Point_2 p1*, *Point_2 p2*)

returns the direction of the vector *p2* - *p1*.

bool *K*.strictly_ordered_ccw(*Direction_2 d1*, *Direction_2 d2*, *Direction_2 d3*)

returns *true* iff *d2* is in the interior of the counterclockwise angular sector between *d1* and *d3*.

11.1.5 Extended Kernel Traits (ExtendedKernelTraits_2)

1. Definition

ExtendedKernelTraits_2 is a kernel concept providing extended geometry[1]. Let *K* be an instance of the data type *ExtendedKernelTraits_2*. The central notion of extended geomtry are extended points. An extended point represents either a standard affine point of the Cartesian plane or a non-standard point representing the equivalence class of rays where two rays are equivalent if one is contained in the other.

Let R be an infinimaximal number[2], F be the square box with corners $NW(-R,R)$, $NE(R,R)$, $SE(R,-R)$, and $SW(-R,-R)$. Let p be a non-standard point and let r be a ray defining it. If the frame F contains the source point of r then let $p(R)$ be the intersection of r with the frame F, if F does not contain the source of r then $p(R)$ is undefined. For a standard point let $p(R)$ be equal to p if p is contained in the frame F and let $p(R)$ be undefined otherwise. Clearly, for any standard or non-standard point p, $p(R)$ is defined for any sufficiently large R. Let f be any function on standard points, say with k arguments. We call f *extensible* if for any k points $p_1, \ldots, p_k$ the function value $f(p_1(R), \ldots, p_k(R))$ is constant for all sufficiently large R. We define this value as $f(p_1, \ldots, p_k)$. Predicates like lexicographic order of points, orientation, and incircle tests are extensible.

An extended segment is defined by two extended points such that it is either an affine segment, an affine ray, an affine line, or a segment that is part of the square box. Extended directions extend the affine notion of direction to extended objects.

[1] It is called extended geometry for simplicity, though it is not a real geometry in the classical sense.

[2] A finite but very large number.

This extended geometry concept serves two purposes. It offers functionality for changing between standard affine and extended geometry. At the same time it provides extensible geometric primitives on the extended geometric objects.

2. Types

Affine kernel types

ExtendedKernelTraits_2::Standard_kernel	the standard affine kernel.
ExtendedKernelTraits_2::Standard_RT	the standard ring type.
ExtendedKernelTraits_2::Standard_point_2	standard points.
ExtendedKernelTraits_2::Standard_segment_2	standard segments.
ExtendedKernelTraits_2::Standard_line_2	standard oriented lines.
ExtendedKernelTraits_2::Standard_direction_2	standard directions.
ExtendedKernelTraits_2::Standard_ray_2	standard rays.
ExtendedKernelTraits_2::Standard_aff_transformation_2	standard affine transformations.

Extended kernel types

ExtendedKernelTraits_2::RT	the ring type of our extended kernel.
ExtendedKernelTraits_2::Point_2	extended points.
ExtendedKernelTraits_2::Segment_2	extended segments.
ExtendedKernelTraits_2::Direction_2	extended directions.

ExtendedKernelTraits_2:: Point_type { SWCORNER, LEFTFRAME, NWCORNER, BOTTOMFRAME, STANDARD, TOPFRAME, SECORNER, RIGHTFRAME, NECORNER }

a type descriptor for extended points.

3. Operations

Interfacing the affine kernel types

Point_2 *K*.construct_point(*Standard_point_2 p*)

creates an extended point and initializes it to the standard point p.

Point_2 *K*.construct_point(*Standard_line_2 l*)

creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line l.

Point_2 *K*.construct_point(*Standard_point_2 p1*, *Standard_point_2 p2*)

creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line $l(p1,p2)$.

Point_2 *K*.construct_point(*Standard_point_2 p*, *Standard_direction_2 d*)

creates an extended point and initializes it to the equivalence class of all the rays underlying the ray starting in p in direction d.

Point_2 *K*.construct_opposite_point(*Standard_line_2 l*)

creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line opposite to l.

Point_type	*K*.type(*Point_2 p*)	determines the type of p and returns it.
bool	*K*.is_standard(*Point_2 p*)	returns *true* iff p is a standard point.

Standard_point_2 *K*.standard_point(*Point_2 p*)
returns the standard point represented by p. *Precondition*: $K.is\_standard(p)$.

Standard_line_2 *K*.standard_line(*Point_2 p*) returns the oriented line representing the bundle of rays defining p. *Precondition*: $!K.is\_standard(p)$.

Standard_ray_2 *K*.standard_ray(*Point_2 p*) a ray defining p. *Precondition*: $!K.is\_standard(p)$.

Point_2 *K*.NE() returns the point on the northeast frame corner.

Point_2 *K*.SE() returns the point on the southeast frame corner.

Point_2 *K*.NW() returns the point on the northwest frame corner.

Point_2 *K*.SW() returns the point on the southwest frame corner.

Geometric kernel calls

Point_2 *K*.source(*Segment_2 s*) returns the source point of *s*.

Point_2 *K*.target(*Segment_2 s*) returns the target point of *s*.

Segment_2 *K*.construct_segment(*Point_2 p*, *Point_2 q*)
constructs a segment *pq*.

int *K*.orientation(*Segment_2 s*, *Point_2 p*)
returns the orientation of *p* with respect to the line through *s*.

int *K*.orientation(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*)
returns the orientation of *p3* with respect to the line through *p1p2*.

bool *K*.leftturn(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*)
return true iff the *p3* is left of the line through *p1p2*.

bool *K*.is_degenerate(*Segment_2 s*)
return true iff *s* is degenerate.

int *K*.compare_xy(*Point_2 p1*, *Point_2 p2*)
returns the lexicographic order of *p1* and *p2*.

int *K*.compare_x(*Point_2 p1*, *Point_2 p2*)
returns the order on the *x*-coordinates of *p1* and *p2*.

int *K*.compare_y(*Point_2 p1*, *Point_2 p2*)
returns the order on the *y*-coordinates of *p1* and *p2*.

Point_2 *K*.intersection(*Segment_2 s1*, *Segment_2 s2*)
returns the point of intersection of the lines supported by *s1* and *s2*. *Precondition*: the intersection point exists.

Direction_2 *K*.construct_direction(*Point_2 p1*, *Point_2 p2*)
returns the direction of the vector *p2* - *p1*.

bool *K*.strictly_ordered_ccw(*Direction_2 d1*, *Direction_2 d2*, *Direction_2 d3*)
returns *true* iff *d2* is in the interior of the counterclockwise angular sector between *d1* and *d3*.

bool *K*.strictly_ordered_along_line(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*)
returns *true* iff *p2* is in the relative interior of the segment *p1p3*.

bool *K*.contains(*Segment_2 s*, *Point_2 p*)
returns true iff *s* contains *p*.

bool	*K*.first_pair_closer_than_second(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*, *Point_2 p4*)	
		returns true iff $\|p1 - p2\| < \|p3 - p4\|$.
char∗	*K*.output_identifier( )	returns a unique identifier for kernel object input/output.

11.1.6 Traits concept for the generic sweep (GenericSweepTraits)

1. Definition

GenericSweepTraits is the concept for the template parameter *T* of the generic plane sweep class *generic_sweep*<*T*>. It defines the interface that has to be implemented to adapt the generic sweep framework to a concrete instance.

2. Types

GenericSweepTraits::*INPUT* the input interface.

GenericSweepTraits::*OUTPUT* the output container.

GenericSweepTraits::*GEOMETRY*

the geometry used.

3. Creation

GenericSweepTraits *T*(*INPUT in*, *OUTPUT*& *out*, *GEOMETRY geom*);

creates an object *T* of type *GenericSweepTraits*, and allows thereby to transport the input/output data into the traits class.

GenericSweepTraits *T*(*OUTPUT*& *out*, *GEOMETRY geom*);

creates an object *T* of type *GenericSweepTraits*, and allows thereby to transport the input/output data into the traits class.

4. Operations

void	*T*.initialize_structures( )	codes initialization of structures before the sweep loop.
bool	*T*.event_exists( )	codes loop control at the beginning of the sweep loop body.
void	*T*.procede_to_next_event( )	codes loop progress at the end of the sweep loop body.
void	*T*.process_event( )	codes the actual event handling (the loop body).
void	*T*.complete_structures( )	codes the completion phase after the sweep loop.
void	*T*.check_invariants( )	allows checking sweep loop invariants.
void	*T*.check_final( )	allows checking of final invariants (after completion).

11.1.7 Visualization of the generic sweep (GenericSweepVisualization)

1. Definition

GenericSweepVisualization is the concept for the second template parameter *VT* of the sweep observer *sweep_observer*<*GPS*, *VT*> defined above. It provides the interface to adapt the sweep observation process to a visualization device.

2. Types

GenericSweepVisualization::*VDEVICE*

the visualization device

3. Creation

GenericSweepVisualization C;

can be used to initialize and display the visualization device.

4. Operations

void	*C*.post_init_animation(*GPS*::*TRAITS*& *gpst*)	
		animation actions after the initialization of the sweep.
void	*C*.pre_event_animation(*GPS*::*TRAITS*& *gpst*)	
		animation actions before each event handling.
void	*C*.post_event_animation(*GPS*::*TRAITS*& *gpst*)	
		animation actions after each event handling.
void	*C*.post_completion_animation(*GPS*::*TRAITS*& *gpst*)	
		animation actions after the completion phase of the sweep.
VDEVICE&	*C*.device()	access operation to the visualization device.

Note that the entry point for visualization of the sweep is the access to an object *gpst* of type *GPS*::*TRAITS*. This is the sweep traits class triggering the sweep within the generic sweep framework and storing all status information of the sweep. Thereby it contains also all information necessary for visualization. *C* obtains access to this object at defined event points and can thereby analyze the status of *gpst* and trigger corresponding visualization actions via its visualization methods.

11.1.8 Topological plane map exploration (PMConstDecorator)

1. Definition

An instance *D* of the data type *PMConstDecorator* is a decorator for interfacing the topological structure of a plane map *P* (read-only).

A plane map *P* consists of a triple (V,E,F) of vertices, edges, and faces. We collectively call them objects. An edge *e* is a pair of vertices (v,w) with incidence operations $v = source(e)$, $w = target(e)$. The list of all edges with source *v* is called the adjacency list $A(v)$.

Edges are paired into twins. For each edge $e = (v,w)$ there's an edge $twin(e) = (w,v)$ and $twin(twin(e)) == e$[3].

An edge $e = (v,w)$ knows two adjacent edges $en = next(e)$ and $ep = previous(e)$ where $source(en) = w$, $previous(en) = e$ and $target(ep) = v$ and $next(ep) = e$. By this symmetric $previous - next$ relationship all edges are partitioned into face cycles. Two edges *e* and e' are in the same face cycle if $e = next^*(e')$. All edges *e* in the same face cycle have the same incident face $f = face(e)$. The cyclic order on the adjacency list of a vertex $v = source(e)$ is given by $cyclic\_adj\_succ(e) = twin(previous(e))$ and $cyclic\_adj\_pred(e) = next(twin(e))$.

A vertex *v* is embedded via coordinates $point(v)$. By the embedding of its source and target an edge corresponds to a segment. *P* has the property that the embedding is always *order-preserving*. This means a ray fixed in $point(v)$ of a vertex *v* and swept around counterclockwise meets the embeddings of $target(e)$ $(e \in A(v))$ in the cyclic order defined by the list order of *A*.

The embedded face cycles partition the plane into maximal connected subsets of points. Each such set corresponds to a face. A face is bounded by its incident face cycles. For all the edges in the non-trivial face cycles it holds that the face is left of the edges. There can also be trivial face cycles in form of isolated vertices in the interior of a face. Each such vertex *v* knows its surrounding face $f = face(v)$.

We call the embedded map (V,E) also the 1-skeleton of *P*.

[3] The existence of the edge pairs makes *P* a bidirected graph, the *twin* links make *P* a map.

Plane maps are attributed. To each object $u \in V \cup E \cup F$ we attribute a value $mark(u)$ of type *Mark*. *Mark* fits the concepts assignable, default-constructible, and equal-comparable.

2. Types

PMConstDecorator::*Plane_map* — The underlying plane map type

PMConstDecorator::*Point* — The point type of vertices.

PMConstDecorator::*Mark* — All objects (vertices, edges, faces) are attributed by a *Mark* object.

PMConstDecorator::*Size_type* — The size type.

Local types are handles, iterators and circulators of the following kind: *Vertex_const_handle*, *Vertex_const_iterator*, *Halfedge_const_handle*, *Halfedge_const_iterator*, *Face_const_handle*, *Face_const_iterator*. Additionally the following circulators are defined.

PMConstDecorator::*Halfedge_around_vertex_const_circulator*

circulating the outgoing halfedges in $A(v)$.

PMConstDecorator::*Halfedge_around_face_const_circulator*

circulating the halfedges in the face cycle of a face f.

PMConstDecorator::*Hole_const_iterator* — iterating all holes of a face f. The type is convertible to *Halfedge_const_handle*.

PMConstDecorator::*Isolated_vertex_const_iterator*

iterating all isolated vertices of a face f. The type generalizes *Vertex_const_handle*.

3. Creation

PMConstDecorator *D*(*const Plane_map*& *P*);

constructs a plane map decorator exploring *P*.

4. Operations

Vertex_const_handle *D*.source(*Halfedge_const_handle e*)

returns the source of *e*.

Vertex_const_handle *D*.target(*Halfedge_const_handle e*)

returns the target of *e*.

Halfedge_const_handle *D*.twin(*Halfedge_const_handle e*)

returns the twin of *e*.

bool *D*.is_isolated(*Vertex_const_handle v*)

returns *true* iff $A(v) = \emptyset$.

Halfedge_const_handle *D*.first_out_edge(*Vertex_const_handle v*)

returns one halfedge with source *v*. It's the starting point for the circular iteration over the halfedges with source *v*. *Precondition*: $!is\_isolated(v)$.

Halfedge_const_handle *D*.last_out_edge(*Vertex_const_handle v*)

returns the halfedge with source *v* that is the last in the circular iteration before encountering $first\_out\_edge(v)$ again. *Precondition*: $!is\_isolated(v)$.

Halfedge_const_handle *D*.cyclic_adj_succ(*Halfedge_const_handle e*)

returns the edge after *e* in the cyclic ordered adjacency list of $source(e)$.

Halfedge_const_handle *D*.cyclic_adj_pred(*Halfedge_const_handle e*)

returns the edge before e in the cyclic ordered adjacency list of $source(e)$.

Halfedge_const_handle *D*.next(*Halfedge_const_handle e*)

returns the next edge in the face cycle containing e.

Halfedge_const_handle *D*.previous(*Halfedge_const_handle e*)

returns the previous edge in the face cycle containing e.

Face_const_handle *D*.face(*Halfedge_const_handle e*)

returns the face incident to e.

Face_const_handle *D*.face(*Vertex_const_handle v*)

returns the face incident to v. *Precondition*: $is\_isolated(v)$.

Halfedge_const_handle *D*.halfedge(*Face_const_handle f*)

returns a halfedge in the bounding face cycle of f (*Halfedge_const_handle*( ) if there is no bounding face cycle).

Iteration

Halfedge_around_vertex_const_circulator *D*.out_edges(*Vertex_const_handle v*)

returns a circulator for the cyclic adjacency list of v.

Halfedge_around_face_const_circulator *D*.face_cycle(*Face_const_handle f*)

returns a circulator for the outer face cycle of f.

Hole_const_iterator *D*.holes_begin(*Face_const_handle f*)

returns an iterator for all holes in the interior of f. A *Hole_iterator* can be assigned to a *Halfedge_around_face_const_circulator*.

Hole_const_iterator *D*.holes_end(*Face_const_handle f*)

returns the past-the-end iterator of f.

Isolated_vertex_const_iterator *D*.isolated_vertices_begin(*Face_const_handle f*)

returns an iterator for all isolated vertices in the interior of f.

Isolated_vertex_const_iterator *D*.isolated_vertices_end(*Face_const_handle f*)

returns the past the end iterator of f.

Associated Information

The type *Mark* is the general attribute of an object. The type *GenPtr* is equal to type $void*$.

const Point& *D*.point(*Vertex_const_handle v*)

returns the embedding of v.

const Mark& *D*.mark(*Vertex_const_handle v*)

returns the mark of v.

const Mark& *D*.mark(*Halfedge_const_handle e*)

returns the mark of e.

const Mark& *D*.mark(*Face_const_handle f*)

returns the mark of f.

const GenPtr&	*D*.info(*Vertex_const_handle v*)	returns a generic information slot.
const GenPtr&	*D*.info(*Halfedge_const_handle e*)	returns a generic information slot.
const GenPtr&	*D*.info(*Face_const_handle f*)	returns a generic information slot.

Statistics and Integrity

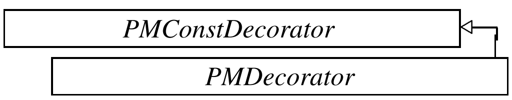
Size_type	*D*.number_of_vertices()	returns the number of vertices.
Size_type	*D*.number_of_halfedges()	returns the number of halfedges.
Size_type	*D*.number_of_edges()	returns the number of halfedge pairs.
Size_type	*D*.number_of_faces()	returns the number of faces.
Size_type	*D*.number_of_face_cycles()	returns the number of face cycles.
Size_type	*D*.number_of_connected_components()	calculates the number of connected components of *P*.
void	*D*.print_statistics(*std::ostream& os = std::cout*)	print the statistics of *P*: the number of vertices, edges, and faces.
void	*D*.check_integrity_and_topological_planarity(*bool faces = true*)	checks the link structure and the genus of *P*.

11.1.9 Plane map manipulation (PMDecorator)

1. Definition

An instance *D* of the data type *PMDecorator* is a decorator to examine and modify a plane map. *D* inherits from *PMConstDecorator* but provides additional manipulation operations.

2. Generalization



3. Types

Local types are handles, iterators and circulators of the following kind: *Vertex_handle*, *Vertex_iterator*, *Halfedge_handle*, *Halfedge_iterator*, *Face_handle*, *Face_iterator*. Additionally the following circulators are defined. The *circulators* can be constructed from the corresponding halfedge handles or iterators.

PMDecorator::Halfedge_around_vertex_circulator

circulating the outgoing halfedges in $A(v)$.

PMDecorator::Halfedge_around_face_circulator

circulating the halfedges in the face cycle of a face f.

PMDecorator::Hole_iterator

iterating all holes of a face f. The type is convertible to *Halfedge_handle*.

PMDecorator::Isolated_vertex_iterator

iterating all isolated vertices of a face f. The type generalizes *Vertex_handle*.

PMDecorator:: { BEFORE, AFTER }

insertion order labels.

4. Creation

PMDecorator D(*Plane_map*& *p*);

constructs a decorator working on *P*.

5. Operations

Plane_map&	*D*.plane_map()	returns the plane map decorated.
void	*D*.clear()	reinitializes *P* to the empty map.
Vertex_handle	*D*.source(*Halfedge_handle e*)	returns the source of *e*.
Vertex_handle	*D*.target(*Halfedge_handle e*)	returns the target of *e*.
Halfedge_handle	*D*.twin(*Halfedge_handle e*)	returns the twin of *e*.
bool	*D*.is_isolated(*Vertex_handle v*)	returns *true* iff *v* is linked to the interior of a face. This is equivalent to the condition that $A(v) = \emptyset$.
bool	*D*.is_closed_at_source(*Halfedge_handle e*)	returns *true* when $prev(e) == twin(e)$.
Halfedge_handle	*D*.first_out_edge(*Vertex_handle v*)	returns a halfedge with source *v*. It's the starting point for the circular iteration over the halfedges with source *v*. *Precondition*: $!is\_isolated(v)$.
Halfedge_handle	*D*.last_out_edge(*Vertex_handle v*)	returns a the halfedge with source *v* that is the last in the circular iteration before encountering $first\_out\_edge(v)$ again. *Precondition*: $!is\_isolated(v)$.
Halfedge_handle	*D*.cyclic_adj_succ(*Halfedge_handle e*)	returns the edge after *e* in the cyclic ordered adjacency list of $source(e)$.
Halfedge_handle	*D*.cyclic_adj_pred(*Halfedge_handle e*)	returns the edge before *e* in the cyclic ordered adjacency list of $source(e)$.
Halfedge_handle	*D*.next(*Halfedge_handle e*)	returns the next edge in the face cycle containing *e*.
Halfedge_handle	*D*.previous(*Halfedge_handle e*)	returns the previous edge in the face cycle containing *e*.
Face_handle	*D*.face(*Halfedge_handle e*)	returns the face incident to *e*.
Face_handle	*D*.face(*Vertex_handle v*)	returns the face incident to *v*. *Precondition*: $is\_isolated(v)$.

Halfedge_handle *D*.halfedge(*Face_handle f*)

returns a halfedge in the bounding face cycle of f (*Halfedge_handle*( ) if there is no bounding face cycle).

Iteration

Halfedge_around_vertex_circulator *D*.out_edges(*Vertex_handle v*)

returns a circulator for the cyclic adjacency list of v.

Halfedge_around_face_circulator *D*.face_cycle(*Face_handle f*)

returns a circulator for the outer face cycle of f.

Hole_iterator *D*.holes_begin(*Face_handle f*)

returns an iterator for all holes in the interior of f. A *Hole_iterator* can be assigned to a *Halfedge_around_face_circulator*.

Hole_iterator *D*.holes_end(*Face_handle f*)

returns the past-the-end iterator of f.

Isolated_vertex_iterator *D*.isolated_vertices_begin(*Face_handle f*)

returns an iterator for all isolated vertices in the interior of f.

Isolated_vertex_iterator *D*.isolated_vertices_end(*Face_handle f*)

returns the past the end iterator of f.

Update Operations

Vertex_handle *D*.new_vertex(*Vertex_base vb = Vertex_base*( ))

creates a new vertex.

Face_handle *D*.new_face(*Face_base fb = Face_base*( ))

creates a new face.

void *D*.link_as_outer_face_cycle(*Face_handle f*, *Halfedge_handle e*)

makes e the entry point of the outer face cycle of f and makes f the face of all halfedges in the face cycle of e.

void *D*.link_as_hole(*Face_handle f*, *Halfedge_handle e*)

makes e the entry point of a hole face cycle of f and makes f the face of all halfedges in the face cycle of e.

void *D*.link_as_isolated_vertex(*Face_handle f*, *Vertex_handle v*)

makes v an isolated vertex within f.

void *D*.clear_face_cycle_entries(*Face_handle f*)

removes all isolated vertices and halfedges that are entrie points into face cycles from the lists of f.

Halfedge_handle *D*.new_halfedge_pair(*Vertex_handle v1*, *Vertex_handle v2*, *Halfedge_base hb = Halfedge_base*( ))

creates a new pair of edges $(e1,e2)$ representing $(v1,v2)$ by appending the *ei* to $A(vi)$ $(i = 1,2)$.

Halfedge_handle *D*.new_halfedge_pair(*Halfedge_handle e1*, *Halfedge_handle e2*, *Halfedge_base hb = Halfedge_base*( ), *int pos1 = AFTER*, *int pos2 = AFTER*)

creates a new pair of edges $(h1,h2)$ representing $(source(e1),source(e2))$ by inserting the *hi* before or after *ei* into the cyclic adjacency list of $source(ei)$ depending on *posi* $(i = 1,2)$ from *PMDecorator*::*BEFORE*, *PMDecorator*::*AFTER*.

Halfedge_handle *D*.new_halfedge_pair(*Halfedge_handle e*, *Vertex_handle v*, *Halfedge_base hb* = *Halfedge_base*( ), *int pos* = *AFTER*)

creates a new pair of edges $(e1, e2)$ representing $(source(e), v)$ by inserting *e1* before or after *e* into the cyclic adjacency list of $source(e)$ depending on *pos* from *PMDecorator*:: *BEFORE*, *PMDecorator*::*AFTER* and appending *e2* to $A(v)$.

Halfedge_handle *D*.new_halfedge_pair(*Vertex_handle v*, *Halfedge_handle e*, *Halfedge_base hb* = *Halfedge_base*( ), *int pos* = *AFTER*)

symmetric to the previous one.

void *D*.delete_halfedge_pair(*Halfedge_handle e*)

deletes *e* and its twin and updates the adjacency at its source and its target.

void *D*.delete_vertex(*Vertex_handle v*)

deletes *v* and all outgoing edges $A(v)$ as well as their twins. Updates the adjacency at the targets of the edges in $A(v)$.

void *D*.delete_face(*Face_handle f*)

deletes the face *f* without consideration of topological linkage.

bool *D*.has_outdeg_two(*Vertex_handle v*)

return true when *v* has outdegree two.

void *D*.merge_halfedge_pairs_at_target(*Halfedge_handle e*)

merges the halfedge pairs at $v = target(e)$. *e* and $twin(e)$ are preserved, $next(e)$, $twin(next(e))$ and *v* are deleted in the merger. *Precondition*: *v* has outdegree two. The adjacency at $source(e)$ and $target(next(e))$ is kept consistent.

Incomplete topological update primitives

Halfedge_handle *D*.new_halfedge_pair_at_source(*Vertex_handle v*, *int pos* = *AFTER*, *Halfedge_base hb* = *Halfedge_base*( ))

creates a new pair of edges $(e1, e2)$ representing $(v, ())$ by inserting *e1* at the beginning (BEFORE) or end (AFTER) of adjacency list of *v*.

void *D*.delete_halfedge_pair_at_source(*Halfedge_handle e*)

deletes *e* and its twin and updates the adjacency at its source.

void *D*.link_as_target_and_append(*Vertex_handle v*, *Halfedge_handle e*)

makes *v* the target of *e* and appends $twin(e)$ to $A(v)$.

Halfedge_handle *D*.new_halfedge_pair_without_vertices()

inserts an open edge pair, and inits all link slots to their default handles.

void *D*.delete_vertex_only(*Vertex_handle v*)

deletes *v* without consideration of adjacency.

void *D*.delete_halfedge_pair_only(*Halfedge_handle e*)

deletes *e* and its twin without consideration of adjacency.

void *D*.link_as_target_of(*Halfedge_handle e*, *Vertex_handle v*)

makes $target(e) = v$ and sets *e* as the first in-edge if *v* was isolated before.

void	*D*.link_as_source_of(*Halfedge_handle e*, *Vertex_handle v*)	makes $source(e) = v$ and sets e as the first out-edge if v was isolated before.
void	*D*.make_first_out_edge(*Halfedge_handle e*)	makes e the first outgoing halfedge in the cyclic adjacency list of $source(e)$.
void	*D*.set_adjacency_at_source_between(*Halfedge_handle e*, *Halfedge_handle en*)	makes e and en neigbors in the cyclic ordered adjacency list around $v = source(e)$. *Precondition*: $source(e) == source(en)$.
void	*D*.set_adjacency_at_source_between(*Halfedge_handle e1*, *Halfedge_handle e_between*, *Halfedge_handle e2*)	inserts *e_between* into the adjacency list around $source(e1)$ between $e1$ and $e2$ and makes $source(e1)$ the source of *e_between*. *Precondition*: $source(e1) == source(e2)$.
void	*D*.close_tip_at_target(*Halfedge_handle e*, *Vertex_handle v*)	sets v as target of e and closes the tip by setting the corresponding pointers such that $prev(twin(e)) == e$ and $next(e) == twin(e)$.
void	*D*.close_tip_at_source(*Halfedge_handle e*, *Vertex_handle v*)	sets v as source of e and closes the tip by setting the corresponding pointers such that $prev(e) == twin(e)$ and $next(twin(e)) == e$.
void	*D*.remove_from_adj_list_at_source(*Halfedge_handle e*)	removes a halfedge pair $(e, twin(e)$ from the adjacency list of $source(e)$. Afterwards $next(prev(e)) == next(twin(e))$ and $first\_out\_edge(source(e))$ is valid if $degree(source(v)) > 1$ before the operation.
void	*D*.unlink_as_hole(*Halfedge_handle e*)	removes e's existence as an face cycle entry point of $face(e)$. Does not update the face links of the corresponding face cycle halfedges.
void	*D*.unlink_as_isolated_vertex(*Vertex_handle v*)	removes v's existence as an isolated vertex in $face(v)$. Does not update v's face link.
void	*D*.link_as_prev_next_pair(*Halfedge_handle e1*, *Halfedge_handle e2*)	makes $e1$ and $e2$ adjacent in the face cycle $\ldots - e1 - e2 - \ldots$. Afterwards $e1 = previous(e2)$ and $e2 = next(e1)$.
void	*D*.set_face(*Halfedge_handle e*, *Face_handle f*)	makes f the face of e.
void	*D*.set_face(*Vertex_handle v*, *Face_handle f*)	makes f the face of v.
void	*D*.set_halfedge(*Face_handle f*, *Halfedge_handle e*)	makes e entry edge in the outer face cycle of f.
void	*D*.set_hole(*Face_handle f*, *Halfedge_handle e*)	makes e entry edge in a hole face cycle of f.
void	*D*.set_isolated_vertex(*Face_handle f*, *Vertex_handle v*)	makes v an isolated vertex of f.

Cloning

void *D*.clone(*Plane_map H*)

clones H into P. Afterwards P is a copy of H.
Precondition: *H.check_integrity_and_topological_planarity*( ) and P is empty.

template <*typename LINKDA*>
void *D*.clone_skeleton(*Plane_map H*, *LINKDA L*)

clones the skeleton of H into P. Afterwards P is a copy of H. The link data accessor allows to transfer information from the old to the new objects. It needs the function call operators:
void operator( )(*Vertex_handle vn*, *Vertex_const_handle vo*) *const*
void operator( )(*Halfedge_handle hn*, *Halfedge_const_handle ho*) *const*
where vn, hn are the cloned objects and vo, ho are the original objects.
Precondition: *H.check_integrity_and_topological_planarity*( ) and P is empty.

Associated Information

Point& *D*.point(*Vertex_handle v*)

returns the embedding of v.

Mark& *D*.mark(*Vertex_handle v*)

returns the mark of v.

Mark& *D*.mark(*Halfedge_handle e*)

returns the mark of e.

Mark& *D*.mark(*Face_handle f*)

returns the mark of f.

GenPtr& *D*.info(*Vertex_handle v*)

returns a generic information slot.

GenPtr& *D*.info(*Halfedge_handle e*)

returns a generic information slot.

GenPtr& *D*.info(*Face_handle f*)

returns a generic information slot.

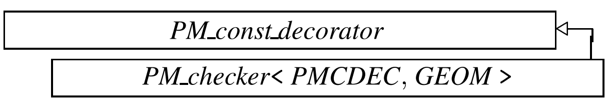
11.2 Manpages

11.2.1 Plane map checking (PM_checker)

1. Definition

An instance D of the data type *PM_checker*< *PMCDEC*, *GEOM* > is a decorator to check the structure of a plane map. It is generic with respect to two template concepts. *PMCDEC* has to be a decorator model of our *PM_const_decorator* concept. *GEOM* has to be a model of our geometry kernel concept.

2. Generalization



3. Types

PM_checker< *PMCDEC*, *GEOM* >::*PM_const_decorator*

equals *PMCDEC*.

PM_checker< PMCDEC, GEOM >::Plane_map

equals *PMCDEC::Plane_map*, the underlying plane map type.

PM_checker< PMCDEC, GEOM >::Geometry

equals *GEOM*. Add link to GEOM concept.
Precondition: *Geometry::Point_2* equals *Plane_map::Point*.

Iterators, handles, and circulators are inherited from *PM_const_decorator*.

4. Creation

PM_checker< PMCDEC, GEOM > D(Plane_map& P, const Geometry& k = Geometry( ));

constructs a plane map checker working on P with geometric predicates used from k.

5. Operations

void *D*.check_order_preserving_embedding(*Vertex_const_handle v*)

checks if the embedding of the targets of the edges in the adjacency list $A(v)$ is counter-clockwise order-preserving with respect to the order of the edges in $A(v)$.

void *D*.check_order_preserving_embedding( )

checks if the embedding of all vertices of P is counter-clockwise order-preserving with respect to the adjacency list ordering of all vertices.

void *D*.check_forward_prefix_condition(*Vertex_const_handle v*)

checks the forward-prefix property of the adjacency list of v.

Halfedge_const_iterator *D*.check_boundary_is_clockwise_weakly_polygon( )

checks if the outer face cycle of P is a clockwise weakly polygon and returns a halfedge on the boundary. *Precondition*: P is a connected graph.

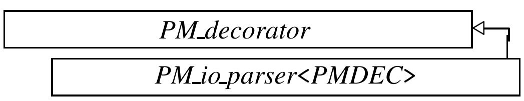
void *D*.check_is_triangulation( ) checks if P is a triangulation.

11.2.2 IO of plane maps (PM_io_parser)

1. Definition

An instance *IO* of the data type *PM_io_parser<PMDEC>* is a decorator to provide input and output of a plane map. *PM_io_parser* is generic with respect to the *PMDEC* parameter. *PMDEC* has to be a decorator model of our *PM_decorator* concept.

2. Generalization



3. Creation

PM_io_parser<PMDEC> IO(std::istream& is, Plane_map& H);

creates an instance *IO* of type *PM_io_parser<PMDEC>* to input H from *is*.

PM_io_parser<PMDEC> IO(std::ostream& os, const Plane_map& H);

creates an instance *IO* of type *PM_io_parser<PMDEC>* to output H to *os*.

4. Operations

void *IO*.print() prints H to *os*.

void *IO*.read() reads H from *is*.

void *PM_io_parser*<*PMDEC*>::dump(*const PMDEC*& D, *std*::*ostream*& *os* = *std*::*cerr*)

prints the plane map decorated by D to *os*.

11.2.3 Drawing plane maps (PM_visualizor)

1. Definition

An instance V of the data type *PM_visualizor*< *PMCDEC*, *GEOM*, *COLORDA* > is a decorator to draw the structure of a plane map into a CGAL window stream. It is generic with respect to two template concepts. *PMCDEC* has to be a decorator model of our *PM_const_decorator* concept. *GEOM* has to be a model of our geometry kernel concept. The data accessor *COLORDA* has to have two members determining the visualization parameters of the objects of P:
CGAL::*Color color*(*Vertex*/*Halfedge*/*Face_const_handle h*) *const*
int width(*Vertex*/*Halfedge_const_handle h*) *const*.

2. Generalization

PM_const_decorator

PM_visualizor< *PMCDEC*, *GEOM*, *COLORDA* >

3. Types

PM_visualizor< *PMCDEC*, *GEOM*, *COLORDA* >::*PM_const_decorator*

The plane map decorator.

PM_visualizor< *PMCDEC*, *GEOM*, *COLORDA* >::*Geometry*

The used geometry.

PM_visualizor< *PMCDEC*, *GEOM*, *COLORDA* >::*Color_objects*

The color data accessor.

4. Creation

PM_visualizor< *PMCDEC*, *GEOM*, *COLORDA* >
V(*CGAL*::*Window_stream*& W, *const PM_const_decorator*& D,
const Geometry& K = *Geometry*( ), *const Color_objects*& C = *Color_objects*( ));

creates an instance V of type *PM_visualizor*< *PMCDEC*, *GEOM*, *COLORDA* > to visualize the vertices, edges, and faces of D in window W. The coloring of the objects is determined by data accessor C.

5. Operations

void V.draw(*Vertex_const_handle v*)

draws v according to the color and width specified by *C.color*(v) and *C.width*(v).

void V.draw(*Halfedge_const_handle e*)

draws e according to the color and width specified by *C.color*(e) and *C.width*(e).

void V.draw(*Face_const_handle f*)

draws f with color *C.color*(f).

void V.draw_map()

draw the whole plane map.

11.2.4 Information association via GenPtr (geninfo)

1. Definition

geninfo<*T*> encapsulates information association via generic pointers of type *GenPtr* $(= void*)$. An object t of type T is stored directly in a variable p of type *GenPtr* if $sizeof(T)$ is not larger than $sizeof(GenPtr)$ (also called word size). Otherwise t is allocated on the heap and referenced via p. This class encapsulates the technicalities, however the user always has to obey the order of its usage: *create-access*/*const_access-clear*. On misuse memory problems occur.

2. Operations

void *geninfo*<*T*>::create(*GenPtr*& *p*)

create a slot for an object of type T referenced via p.

T& *geninfo*<*T*>::access(*GenPtr*& *p*)

access an object of type T via p. *Precondition*: p was initialized via *create* and was not cleared via *clear*.

const T& *geninfo*<*T*>::const_access(*const GenPtr*& *p*)

read-only access of an object of type T via p. *Precondition*: p was initialized via *create* and was not cleared via *clear*.

void *geninfo*<*T*>::clear(*GenPtr*& *p*)

clear the memory used for the object of type T via p. *Precondition*: p was initialized via *create*.

3. Example

In the first example we store a pair of boolean values which normally fit into one word. Thus there will no heap allocation take place.

```
struct A { bool a,b };
GenPtr a;
geninfo<A>::create(a);
A& a_access = geninfo<A>::access(a);
geninfo<A>::clear(a);
```

The second example uses the heap scheme as two longs do not fit into one word.

```
struct B { long a,b };
GenPtr b;
geninfo<B>::create(b);
B& b_access = geninfo<B>::access(b);
geninfo<B>::clear(b);
```

Note that usage of the scheme takes away with the actual check for the type size. Even more important this size might depend on the platform which is used to compile the code and thus the scheme enables platform independent programming.

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2001-4-004	S.W. Choi, H. Seidel	Linear One-sided Stability of MAT for Weakly Injective Domain
MPI-I-2001-4-003	K. Daubert, W. Heidrich, J. Kautz, J. Dischler, H. Seidel	Efficient Light Transport Using Precomputed Visibility
MPI-I-2001-4-002	H.P.A. Lensch, J. Kautz, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing, Transmission, and Interactive Display of High Quality 3D Models on the Web
MPI-I-2001-4-001	H.P.A. Lensch, J. Kautz, M. Goesele, W. Heidrich, H. Seidel	Image-Based Reconstruction of Spatially Varying Materials
MPI-I-2001-2-003	S. Vorobyov	Experiments with Iterative Improvement Algorithms on Completely Unimodel Hypercubes
MPI-I-2001-2-002	P. Maier	A Set-Theoretic Framework for Assume-Guarantee Reasoning
MPI-I-2001-2-001	U. Waldmann	Superposition and Chaining for Totally Ordered Divisible Abelian Groups
MPI-I-2001-1-003	M. Seel	Implementation of Plenar Nef Polyhedra
MPI-I-2001-1-002	U. Meyer	Directed Single-Source Shortest-Paths in Linear Average-Case Time
MPI-I-2001-1-001	P. Krysta	Approximating Minimum Size 1,2-Connected Networks
MPI-I-2000-4-003	S.W. Choi, H. Seidel	Hyperbolic Hausdorff Distance for Medial Axis Transform
MPI-I-2000-4-002	L.P. Kobbelt, S. Bischoff, K. Kähler, R. Schneider, M. Botsch, C. Rössl, J. Vorsatz	Geometric Modeling Based on Polygonal Meshes
MPI-I-2000-4-001	J. Kautz, W. Heidrich, K. Daubert	Bump Map Shadows for OpenGL Rendering
MPI-I-2000-2-001	F. Eisenbrand	Short Vectors of Planar Lattices Via Continued Fractions
MPI-I-2000-1-005	M. Seel, K. Mehlhorn	Infimaximal Frames A Technique for Making Lines Look Like Segments
MPI-I-2000-1-004	K. Mehlhorn, S. Schirra	Generalized and improved constructive separation bound for real algebraic expressions
MPI-I-2000-1-003	P. Fatourou	Low-Contention Depth-First Scheduling of Parallel Computations with Synchronization Variables
MPI-I-2000-1-002	R. Beier, J. Sibeyn	A Powerful Heuristic for Telephone Gossiping
MPI-I-2000-1-001	E. Althaus, O. Kohlbacher, H. Lenhof, P. Müller	A branch and cut algorithm for the optimal solution of the side-chain placement problem
MPI-I-1999-4-001	J. Haber, H. Seidel	A Framework for Evaluating the Quality of Lossy Image Compression
MPI-I-1999-3-005	T.A. Henzinger, J. Raskin, P. Schobbens	Axioms for Real-Time Logics
MPI-I-1999-3-004	J. Raskin, P. Schobbens	Proving a conjecture of Andreka on temporal logic
MPI-I-1999-3-003	T.A. Henzinger, J. Raskin, P. Schobbens	Fully Decidable Logics, Automata and Classical Theories for Defining Regular Real-Time Languages

MPI-I-1999-3-002	J. Raskin, P. Schobbens	The Logic of Event Clocks
MPI-I-1999-3-001	S. Vorobyov	New Lower Bounds for the Expressiveness and the Higher-Order Matching Problem in the Simply Typed Lambda Calculus
MPI-I-1999-2-008	A. Bockmayr, F. Eisenbrand	Cutting Planes and the Elementary Closure in Fixed Dimension
MPI-I-1999-2-007	G. Delzanno, J. Raskin	Symbolic Representation of Upward-closed Sets
MPI-I-1999-2-006	A. Nonnengart	A Deductive Model Checking Approach for Hybrid Systems
MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-007	C. Burnikel, K. Mehlhorn, M. Seel	A simple way to recognize a correct Voronoi diagram of line segments
MPI-I-1999-1-006	M. Nissen	Integration of Graph Iterators into LEDA
MPI-I-1999-1-005	J.F. Sibeyn	Ultimate Parallel List Ranking ?
MPI-I-1999-1-004	M. Nissen, K. Weihe	How generic language extensions enable “open-world” desing in Java
MPI-I-1999-1-003	P. Sanders, S. Egner, J. Korst	Fast Concurrent Access to Parallel Disks
MPI-I-1999-1-002	N.P. Boghossian, O. Kohlbacher, H.-. Lenhof	BALL: Biochemical Algorithms Library
MPI-I-1999-1-001	A. Crauser, P. Ferragina	A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron
MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid *E*-Unification
MPI-I-98-2-004	S. Vorobyov	Satisfiability of Functional+Record Subtype Constraints is NP-Hard
MPI-I-98-2-003	R.A. Schmidt	E-Unification for Subsystems of S4
MPI-I-98-2-002	F. Jacquemard, C. Meyer, C. Weidenbach	Unification in Extensions of Shallow Equational Theories
MPI-I-98-1-031	G.W. Klau, P. Mutzel	Optimal Compaction of Orthogonal Grid Drawings