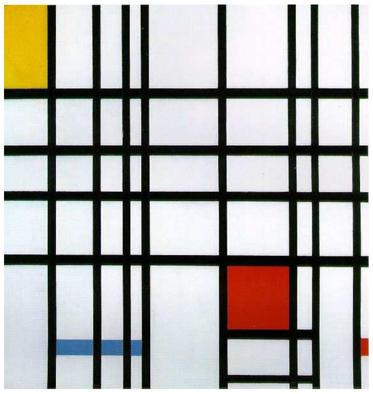
Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik I

Jannis Warnat [1]

Computing a single face in an arrangement of line segments with CGAL

August 22, 2009



Diploma thesis

Supervisors: Prof. Dr. Rolf Klein, Dr. Elmar Langetepe

[1] *Email address:* jannis.warnat@gmx.de

Erklärung

Mit der Abgabe der Diplomarbeit versichere ich gemäß §19 Absatz 7 der DPO vom 15. August 1998, dass ich die Arbeit selbstständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Bonn, den 22. August 2009 Jannis Warnat

Acknowledgment

I would like to thank my parents Bernd Warnat and Elisabeth Warnat-Ostermaier for supporting my academic studies.

Furthermore, I would like to thank Prof. Dr. Rolf Klein and Dr. Elmar Langetepe for introducing me to the field of computational geometry in general and to the subject of this thesis in particular.

Last but not least I owe thanks to to all contributors to CGAL, especially Efi Fogel, Ron Wein, Baruch Zukerman and Tali Zvi along with all current and past members of the Applied Computational Geometry Lab at Tel Aviv university for providing their arrangement package under an open source license.

Cover: Piet Mondrian, *Composition with Yellow, Blue and Red* , 1937-42, oil on canvas, Tate Gallery. London.
Source: http://en.wikipedia.org/wiki/Piet_Mondrian

Contents

Chapter 1

Introduction

Planar arrangements are fundamental structures in the field of computational geometry. Apart from their natural appearance and usefulness in many geometric problems, they are interesting entities in their own right. Some artists (and computer scientists) even seem to find arrangements visually appealing. The most obvious example is Piet Mondrian, as shown on the title page being engaged with what a computational geometrician would consider to be an arrangement of line segments. Another artist worth mentioning in this context is Joan Miró whose paintings frequently resemble arrangements of planar curves.

In this thesis we will be concerned mainly with a special subject-matter explored in the book *Davenport-Schinzel sequences and Their Geometric Applications* by Micha Sharir and Pankaj K. Agarwal [1]. The authors show that a single face in an arrangement of Jordan arcs is not too complex and is not too expensive to compute with an algorithm including - in the words of Sharir and Agarwal - a 'sophisticated sweep-line technique'. We will give a detailed description of the problem and the algorithm solving it as well as present an implementation for the restricted case of bounded line segments with the help of the well-known CGAL library.

CGAL (pronounced 'seagull') is the acronym for the 'Computational Geometry Algorithms Library'.

> The goal of the CGAL Open Source Project is to provide easy access to efficient and reliable geometric algorithms in the form of a C++ library [5].

CGAL was started in 1996 as a collaborative effort of seven universities and research institutes in Europe and Israel. Today CGAL is developed and maintained by a community of developers guided by the CGAL Editorial Board. It is available under an open source license and under a commercial license provided by the spin-off company GeometryFactory [19].

The CGAL community is very active and aims for a release of new CGAL versions every six months. During the work on this thesis, the latest official version switched from 3.3.1 to 3.4. The first beta version of CGAL 3.5 is available since July 31 2009.

A picture says more than a thousand words, especially in the field of computational geometry. The figures in this work, provided to help the reader to visualize the proofs and algorithms, were drawn using the wonderful Ipe by Otfried Cheong [12]. CGAL offers very helpful plug-ins for Ipe (*ipelets*) to assist in visualizing CGAL data structures [6].

Excerpts of the implementation are displayed using a pseudo-code that strongly resembles the actual C++-code so the reader is assumed to be familiar with the basics of the language. Even the theoretical proofs are demonstrated in a way that stays close to the reality of the data structures in CGAL and C++. This is an exercise in translating the theoretical insights about the geometric structure of the problem into the reality of contemporary computer programming.

The programming for this work was done using the well-established integrated development environment Eclipse [15] under Windows XP and cygwin. The reader is encouraged to take a look at the source code since a lot of effort was put into enhancing its readability.

1.1 Motivation

Let R be a polygonal robot with m edges moving on the unbounded plane amid disjoint polygonal obstacles P_i with a total of n edges. Let r be a reference point inside the robot's polygon. We define the following notions ([21], [28]):

Definition 1

- A *placement* is a position of the robot in the plane described by the position of its reference point.

- The *configuration space* AP is the space of *all possible* placements of the robot. In our case the configuration space is $\mathbb{R} \times \mathbb{R}$; the robot has two degrees of freedom. (We consider pure translation and don't allow the robot to rotate in any way.)

- The *free space* FP is the space of all *free placements* in which the robot doesn't intersect the boundary or the interior of any obstacle:

$$FP = \left\{ x \in AP | R(x) \cap \bigcup P_i = \emptyset \right\}$$

  $R(x)$ denotes the robot's polygon at placement x.

The *motion planning problem* in this situation is the following:

Given an initial placement $a \in FP$ and a desired final placement $b \in FP$, is there a collision-free motion of R from a to b?

Although the complexity of FP is in $\Omega(n^2m^2)$ in the worst case, we will show that this problem can be solved in close to linear time by computing a single face in an arrangement of line segments.

1.2 Robustness

An integral element of our algorithm will be a sweep over line segments to find some intersections between them. As in the implementation of any *geometric algorithm* we will have to address *robustness* issues. Following Yap [36], we consider an algorithm to be geometric if

- it consists of a numerical and a combinatorial component,
- the combinatorial component is concerned with discrete relations between geometric objects
- and these discrete relations (embodied in geometric predicates) are characterized by the algorithm via numerical computations.

Robustness issues arise when erroneous numerical computations lead to wrong characterizations of discrete relations between geometric objects.

The flow of a geometric algorithm typically depends on geometric predicates and comparisons. A branch for example may depend on the relative position of a point in comparison to a an oriented line segment; whether it is on, left or right from the segment's supporting line. In numerical terms, this is done by evaluating the sign of a determinant. In the idealized *Real RAM* model ([27]) used in the theoretical analysis of topics in computational geometry, we assume the availability of exact ('infinite-precision') real arithmetic. (For the analysis of time and space requirements we even assume any operation on geometric objects of constant size to take only constant time and require only constant space.) Real-world computers work on just a subset of the rational numbers, usually following the well-known IEEE 754 standard for floating-point arithmetic. Even rational calculations can only be computed with fixed precision. In case of the evaluation of a geometric predicate, we might get an erroneous computation because of round-off errors or overflow. Our program could crash, enter an infinite loop or just return a catastrophically wrong result. In C++ we will usually end up with a segmentation fault. Even small numerical deviations can lead to the wrong evaluation of a predicate and destroy the combinatorial consistency of our algorithmic flow. Some nicely elaborated examples for algorithmic failures due to the shortcomings of fixed precision arithmetic can be found in [25].

In the case of our sweep over line segments, computed intersection points become future sweep events. The event structure is ordered lexicographically by the event points. If two intersection points are very close to each other and their position is not calculated precisely, their order in the event structure may be wrong and the satisfaction of a sweep invariant we have proven in theory can now longer be guaranteed. On the other hand, we have to maintain the order of points and segments along the sweep line by evaluating associated geometric predicates. A wrong answer to the question 'what is the closest segment above the point p along the status line' due to arithmetic imprecision will ruin our computation.

One proposal to deal with this issue is proposed and explored by Fortune [17]. We accept that we are stuck with finite precision arithmetic in the real-world and adapt the theory to this limitation. Fortune defined the following two notions:

- A geometric algorithm working with imprecise arithmetic is called *robust* if it always computes an output that is the (exactly) correct output for some perturbation of the input.

- A robust geometric algorithm is called *stable* if the above mentioned perturbation is small, i.e. constantly bounded.

This approach requires the careful analysis of error bounds and imprecise predicates specifically designed for a certain situation. Fortune gives examples for this difficult task for a few basic predicates and algorithms [17]. An important advantage of this approach is that *degeneracies* disappear naturally. See [27] for examples of degeneracies in configurations of line segments.

Another proposal to reconcile the theory of computational geometry with real-world computers is the *Exact Geometric Computation (EGC)* paradigm [35]. We try to make the real world resemble the theory as closely as possible. The input to a geometric computation can reasonably be assumed to be exact and discrete. We let decisions and branches in our algorithm depend on the evaluation of geometric predicates only and make sure to get all this decisions right by representing our geometric objects exactly. In this work we will always assume an input consisting of rational coordinates and not leave the realm of rational arithmetic. So the required geometric objects can be naively represented exactly by storing their rational coordinates using an arbitrary-precision 'BigNumber' package, like the GNU Multiple Precision Library GMP [20]. We won't always employ the provided exact rational arithmetic but we will employ bignum arithmetic sufficiently accurate to always evaluate geometric predicates correctly. Hence we don't risk errors in the computational path and therefore the combinatorial structure of our output.

The size of exact representations of geometric objects can grow quickly and the assumption of constant-time operations on these objects is challenged. To speed things up, we may consider to accept deviations in parts of the output that are not decisive for its combinatorial structure, for example the exact geometric location of a Voronoi vertex.

By applying the EGC on our line segment sweep, robustness issues will disappear. The most obvious downside is decreased efficiency. Another disadvantage is that we will run into degeneracies. These have to be addressed algorithmically. Since degeneracies are not the focus of this work we will only address a few of them. If we wanted to apply our implementation on real-world inputs we would have to put additional effort into this issue.

Chapter 2

CGAL

The CGAL library follows many design ideas of the *generic programming* paradigm as it is realized in the C++ Standard Template Library (STL) and also adopts STL's notions of *containers*, *iterators* and *functors*. A lot of great resources for information about the STL and generic programming can be found on the Internet, for example provided by Silicon Graphics [32]. CGAL also requires and is inspired by the well-known boost C++ libraries [2]. A short but recommendable survey on generic programming techniques used in the boost libraries that is also relevant to CGAL can be found on [3]. Users and the people involved in the CGAL project can interact via an active and helpful mailing list [7].

CGAL provides a large number of data structures and algorithms from the realm of computational geometry, such as convex hull algorithms, triangulations, arrangements of curves, Voronoi diagrams, polygons, polyhedra and mesh generation to name just a few.

> All these data structures and algorithms operate on geometric objects like points and segments, and perform geometric tests on them. These objects and predicates are regrouped in CGAL Kernels [5].

2.1 Kernels

A CGAL kernel encapsulates fundamental geometric types like points, lines, segments, rays, planes, circles etc. and operations on these. Both types and operations are implemented as C++ classes. The classes representing geometric types go by obvious names such as `Point_2`, `Segment_2` or `Ray_2` with the number after the underscore denoting the dimension of the object. In this work we will only deal with two-dimensional types.

Objects of a class representing a geometric operation appear as *function objects* or *functors*. We distinguish between two types of operations:

- A *predicate* returns either a value of type `bool` or `enum`. For example:
  - A functor of type `Is_vertical_2` checks if a segment is vertical and returns `true` or `false`.
  - A functor of type `Compare_xy_2` compares two points lexicographically and returns a value of the enumeration type `Comparison_result`; either `SMALLER`, `EQUAL` or `LARGER`.
- A *construction* involves the computation of new numerical values and returns something more involved than just a value of type `bool` or `enum`. For example:
  - A functor of type `Intersect_2` computes the intersection of two segments and returns a value of type `Object` which can represent no intersection, a point or a segment in case of an overlap.
  - A functor of type `Compute_squared_distance_2` takes two points and returns the squared distance between them.

The functor classes define the function call operator `operator()` as a member function. If we wanted to compare two points lexicographically we could do the following:

```
Point_2 p,q;
Compare_xy_2 compare_xy_2;
Comparison_result res = compare_xy_2(p,q);
```

A detailed description of CGAL's kernel concept can be found in [22].

The kernel design nicely parallels the familiar approach of theoretical computational geometry, where geometric algorithms are usually described as being applied to constant-sized geometric objects while abstracting from their numerical representation. We assume that there are operations defined on these objects like comparing their relative position or computing intersections. For the correctness of an algorithm presented in a theoretical work the numerical basis of these operations is not important.

CGAL allows us to follow the same approach when implementing an algorithm. We just take care of the combinatorial correctness of our implementation by operating on the geometric objects only via the operations provided by the kernel. At best we don't have to concern ourselves with their numerical basis at all. A kernel itself is parametrized with a number type that is used to store coordinates and to calculate results. As said, in this work we will always start with an input consisting of rational coordinates and we will always stay within rational arithmetic. If we implement an algorithm combinatorially correct and instantiate the chosen kernel with an exact arbitrary-precision big number type, we should always get the correct result.

The use of so-called *floating point filters* can make the evaluation of predicates considerably faster. Remember that we usually have to compute the sign of a determinant. When filtering, we initially employ a fast fixed precision floating point number type while maintaining error-bounds from which we can deduce if the sign of the determinant is known for sure. Only if it is not we have to resort to expensive exact arithmetic. Since the latter should be relatively rare, this approach can make an implementation considerably faster. CGAL offers a predefined `Exact_predicates_exact_constructions_kernel` which provides easy access to the current filtering techniques implemented in CGAL to the user. We will later compare the performance of this filtered kernel to the performance of a non-filtered kernel.

There is still much work in progress on filtering and lazy evaluation schemes in CGAL. See [33] and [4] for a brief introduction and further reference.

To speed things up further and still obtain a high degree of robustness, a kernel can be tailored to evaluate predicates exactly but to use inexact constructions, like in the case of the Voronoi vertex mentioned above. For this purpose, there is a predefined `Exact_predicates_inexact_constructions_kernel`. For some algorithms we even have the choice to instantiate the kernel with a fast fixed-precision number type like `double`. This might work fine for inputs containing few or no degeneracies or we might choose to risk a crashing program now and then if it guarantees us quick results for most inputs. Both approaches are not supported by CGAL's sweep line framework we are about to use, when the construction of intersections is involved.

2.2 Traits

One key aspect that CGAL borrows from the STL is the notion of *concepts* and *models* [10]. A concept defines a set of requirements a concrete type has to satisfy to *conform to the concept* or in other words to be a *model of the concept*. The algorithms provided by CGAL are implemented as class or function templates that are parametrized with a geometric *traits* concept. This concept aggregates the specific geometric objects and operations the specific algorithm requires. The implementation of the algorithm will work with every traits class that is a model of the traits concept. Concepts are not part of the current C++ standard and do not appear in the code (other than as a template parameter) but they are comprehensively documented in the CGAL User and Reference Manual [11]. (Concepts were supposed to be featured in the upcoming new C++ standard C++0x but were removed from the draft standard in July 2009 [13].) We will look into a concrete example of this design later in our description of the CGAL arrangement package.

A traits class template takes a kernel as a template parameter in order to have access to the types and operations defined in the kernel. Most algorithms only need a subset of a kernel's functionality and the traits class picks out what is required; just points, segments and an orientation test for example.

On the other hand we can easily extend or modify the types and operations defined in the kernel. We may want to store non-geometric data with a point or build a more complex predicate from the basic predicates of the kernel to make our code more readable. These selections and extensions are encapsulated in the traits class, while the algorithm works on the more abstract types and operations provided to him via the traits class. This makes the implementation of an algorithm very flexible, since we can instantiate a function template implementing this algorithm with different traits classes while instantiating the traits classes with different kernels. The implementation will still work without us having to adapt to details in the representation of the geometric objects and operations.

Chapter 3

Arrangements

Let's revisit the motion planning problem from the introduction. Our robot has m edges and moves between obstacles with a total of n edges. We can define pairs (e, v) of one robot edge e and one obstacle corner v. If we move the corner v along the edge e, the reference point draws a line segment 'in the sand'. By doing this for all possible pairs of robot edges and obstacle corners we get mn line segments. Analogously, we get mn line segments if we pair each robot corner with each obstacle edge, giving us $2mn$ line segments altogether [28]. We can define the *arrangement* of these line segments.

Definition 2 Let $S = \{s_1, ..., s_l\}$ be a set of l bounded line segments in the plane. The *arrangement* $A(S)$ is the decomposition of the plane in zero-dimensional *vertices*, one-dimensional *edges* and two-dimensional *faces* [1].

- The vertices are the endpoints and intersection points of the line segments.
- The edges are (maximal) connected portions of the s_i that do not contain a vertex.
- The faces are the connected components of

$$\mathbb{R}^2 \setminus \bigcup_{i=1}^{l} s_i.$$

For every face f in our particular arrangement, the following holds:

- either $f \cap FP = \emptyset$
- or $f \cap FP = f$

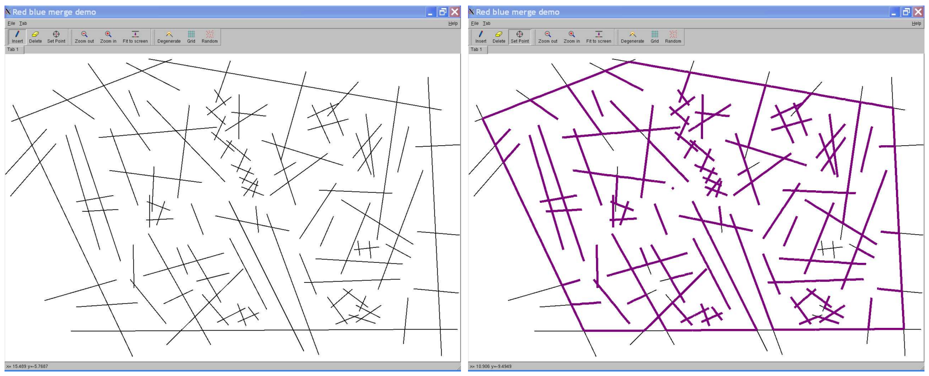


Figure 3.1: A set of line segments and a bounded face in their arrangement.

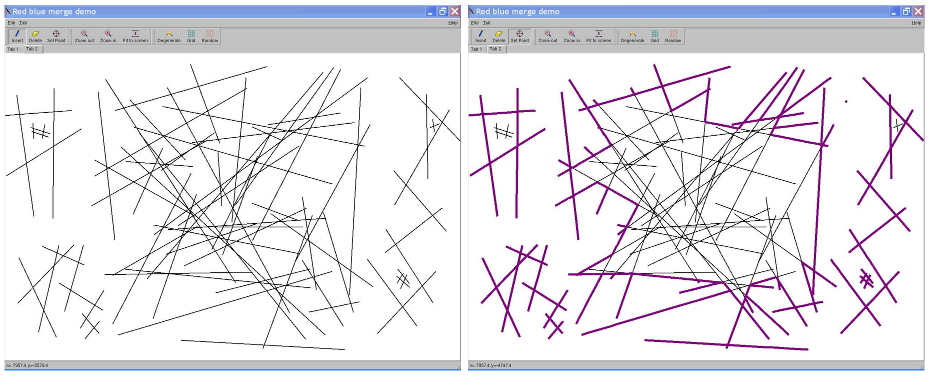


Figure 3.2: A set of line segments and the unbounded face in their arrangement.

The free space could be composed of up to $\Omega(m^2n^2)$ faces; think of a 'grid-like' arrangement. Fortunately we don't have to compute the entire free space. We note that our motion planning problem can be reduced to computing the single face f that contains our initial placement a. Then we can solve the motion planning problem for any desired final placement b by just checking if b is contained in f. If so, a collision-free motion is possible.

3.1 The CGAL Arrangement package

Later in this work we will see that the complexity of a single face is close to linear and that we can compute a single face without computing the full arrangement [1]. To store this face and intermediate results we use the CGAL arrangement package [34]. A single face can be a rather intricate structure. See figure 3.1 for a bounded face and figure 3.2 for an unbounded face.

An arrangement of line segments can be embedded into the plane as

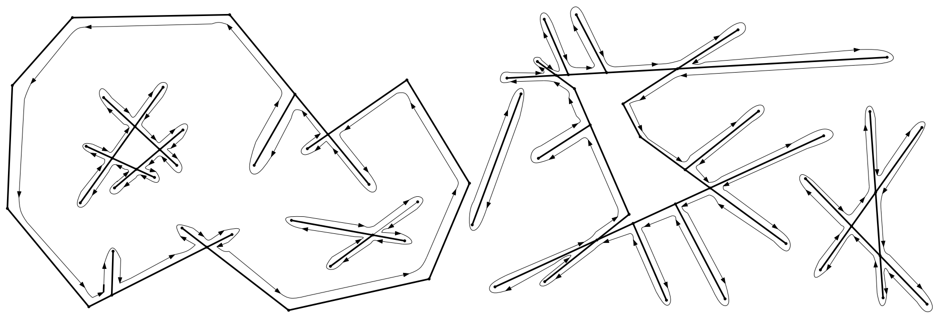


Figure 3.3: The ccbs of a bounded and an unbounded face.

an undirected planar graph. We split the segments into maximal portions that are pairwise non-intersecting. The endpoints of these non-intersecting segments become the vertices of the planar graph, the segments themselves become the edges. The planar graph is internally represented as a *doubly connected edge list (dcel)*. A CGAL arrangement can also contain isolated points as vertices.

Every edge of the underlying undirected planar graph is represented by two directed *halfedges* in the dcel. One halfedge is directed from the vertex representing the lexicographically smaller point to the vertex representing the lexicographically larger point. Its corresponding *twin* halfedge is directed in the opposite direction. Every halfedge has a pointer to the face on its left and to the next and previous halfedges according to its direction. Thus the halfedges are linked in circular lists that form chains winding around the boundary of a face from both sides. These closed circular lists of halfedges are called *connected components of the boundary (ccb)*.

Every face can have any number of *holes*, which are represented in the containing face as an inner ccb ordered in a clockwise manner. Aside from the unbounded face, every face has an outer ccb ordered in a counterclockwise manner. The ccbs are ordered in a way that the associated face always lies to the left of the halfedges of its ccbs.

If we imagine halfedges to 'hover' a tiny bit above or below their inducing segments, connect them at their endpoints and soften the corners, we can regard the resulting curves as closed Jordan curves. We will take this view for the theoretical proofs later. See figure 3.3 for the ccbs of a bounded and an unbounded face.

3.1.1 The `Arrangement_2` class template

The `Arrangement_2<Traits,Dcel>` class template must be instantiated with the following two classes [33]:

- A `Traits` class aggregating the geometric requirements of the arrangement data structures and algorithms. The class must be a model of one of the concepts

  - `ArrangementBasicTraits_2`,
  - `ArrangementXMonotoneTraits_2` or
  - `ArrangementTraits_2`.

  It provides the essential types together with geometric predicates and constructions on these types. For line segments we use the CGAL class `Arr_segment_traits_2<Kernel>` which is a model of the concept `ArrangementXMonotoneTraits_2` and must be instantiated with a kernel.

- A `Dcel` class that must conform to the concept `ArrangementDcel`. It manages the combinatorial structure of the arrangement in terms of vertices, (half)edges and faces. We will use the CGAL class template `Arr_extended_dcel<Traits,VData,HData,FData>`, which provides the opportunity to associate additional non-geometric data with the elements of the arrangement.

To get the idea, let us look at excerpts of the part of the class template `Arr_segment_traits_2<Kernel>` that ensures that the class is a model of the concept `ArrangementXMonotoneTraits_2`. An arrangement can contain only two types of geometric objects:

```
typedef typename Kernel::Point_2 Point_2;
typedef Arr_segment_2<Kernel> X_monotone_curve_2;
```

The type `Point_2` is provided directly be the kernel. Instead of just taking the kernel type `Segment_2` as `X_monotone_curve_2`, the class defines its own extended segment type. The extension is defined as a class template `Arr_segment_2<Kernel>` instantiated with the kernel. This type is more efficient if the segments are being split often, which is common in constructing arrangements. When a kernel type segment is split at an intersection point, the numerical representation of its endpoints might become larger. In case of a cascade of split operations the computing of further intersections becomes more and more time consuming [16]. The type `Arr_segment_2` is tailored to additionally store the coefficients of the segment's supporting line. Most computations and predicate evaluations can be performed on the supporting line which representation does not grow at a split operation.

Every geometric predicate or construction is implemented as a functor class.

```
class Compare_x_2{...};
class Compare_xy_2{...};
class Is_vertical_2{...};
class Compare_y_at_x_2{...};
class Compare_y_at_x_left_2{...};
class Compare_y_at_x_right_2{...};
class Split_2{...};
class Intersect_2{...};
```

Public member functions of the traits class return function objects, which provide these geometric operations to the user like this:

```
Compare_xy_2 compare_xy_2_object () const
{
  return Compare_xy_2();
}
```

In each of the functor classes the function call operator `operator()` is defined as a public member function. The following are the most important for our purpose:

- The `class Compare_x_2` defines

  ```
  Comparison_result operator()(Point_2 p1, Point_2 p2)
  ```

  which compares the x-coordinates of two points. The return value is of type `Comparison_result`, an enumeration consisting of `SMALLER`, `EQUAL` and `LARGER`.

- The `class Compare_xy_2` defines

  ```
  Comparison_result operator()(Point_2 p1, Point_2 p2)
  ```

  which compares two points lexicographically.

- The `class Is_vertical_2` defines

  ```
  bool operator()(X_monotone_curve_2 c)
  ```

  which checks whether `c` is vertical.

- The `class Compare_y_at_x_2` defines

  ```
  Comparison_result operator()(Point_2 p,
     X_monotone_curve_2 c)
  ```

  which checks whether `p` is above (`LARGER`), on (`EQUAL`) or below (`SMALLER`) an x-monotone curve `c`.

- The `class Compare_y_at_x_right_2` defines

  ```
  Comparison_result operator()(Point_2 p,
     X_monotone_curve_2 c1,X_monotone_curve_2 c2)
  ```

  which checks whether `c1` is above (`LARGER`), on (`EQUAL`) or below (`SMALLER`) `c2` to the right of `p` while `p` must be an intersection point of `c1` and `c2`.

- `class Compare_y_at_x_left_2` defines

```
Comparison_result operator()(Point_2 p,
   X_monotone_curve_2 c1, X_monotone_curve_2 c2)
```

which checks whether `c1` is above (`LARGER`), on (`EQUAL`) or below (`SMALLER`) `c2` to the left of `p` while `p` must be an intersection point of `c1` and `c2`.

- `class Split_2` defines

  ```
  void operator()(X_monotone_curve_2 c, Point_2 p,
     X_monotone_curve_2 sc1, X_monotone_curve_2 sc2)
  ```

  which splits `c` at `p` in two curves `sc1` and `sc2`.

- `class Intersect_2` defines (simplified)

  ```
  Object operator()(X_monotone_curve_2 c1,
     X_monotone_curve_2 c2)
  ```

  which computes the intersection between `c1` and `c2`. The output `Object` might represent a point, a x-monotone curve in case of an overlap or nothing.

We will use the terms larger, smaller, above, below, higher, lower in terms of these predicates. We say a point is left (or right) from another point if it is lexicographically smaller (or larger). So the lower endpoint of a vertical segment is left from the segment's upper endpoint.

3.1.2 Accessing and Traversing an arrangement

Whenever we want to access a single element of an arrangement (a vertex, a halfedge or a face), we acquire a *handle* to it. A handle is similar to a pointer in that it provides the dereference operator `operator*` and the member access operator `operator->` [14]. The following handles are available:

- On a `Vertex_handle vh` pointing to a vertex we can call (among others) the following functions:
  - `vh->point()` gives us the underlying geometric `Point_2` object.
  - `vh->degree()` gives us the number of incident edges.
  - `vh->face()` in case of an isolated vertex gives us a `Face_handle` to the face the vertex is contained in.

- On a `Halfedge_handle he` pointing to a halfedge we can call the following functions:
  - `he->curve()` gives us the underlying geometric `X_monotone_curve_2` object.

  - `he->source()` gives us a `Vertex_handle` to the source vertex.
  - `he->target()` gives us a `Vertex_handle` to the target vertex.

- On a `Face_handle fh` pointing to a face we can call the following functions:

  - `fh->is_unbounded()` checks whether `fh` points to the unbounded face.
  - `fh->outer_ccb()` gives us a `Ccb_halfedge_circulator` (see below) to traverse the outer ccb if `fh` represents a bounded face.

  Calling `arr.unbounded_face()` on an `Arrangement_2` object `arr` gives us a handle to the one unbounded face that is part of any arrangement of bounded curves.

CGAL uses the concept of *containers* and *iterators* in the spirit of the STL to cope with sequences of objects. For circular containers we use *circulators*, an extension of the iterator concept for CGAL [10]. The difference to iterators is, that there is no past-the-end position. Incrementing a circulator pointing to the 'last' element of a circular list brings us right to the 'first' element. To traverse an arrangement, the following iterators and circulators are provided:

- A `Vertex_iterator` is used to go over all arrangement vertices. It is convertible to a `Vertex_handle`.

- A `Halfedge_iterator` is used to go over all arrangement halfedges. It is convertible to a `Halfedge_handle`.

- An `Edge_iterator` is used to go over all arrangement edges by skipping every other halfedge. It is convertible to a `Halfedge_handle`.

- A `Face_iterator` is used to go over all arrangement faces. It is convertible to a `Face_handle`.

- A `Halfedge_around_vertex_circulator` is used to traverse all halfedges incident to a specific vertex. It is convertible to a `Halfedge_handle`.

- A `Ccb_halfedge_circulator` is used to traverse all halfedges along a specific ccb. The order depends on whether we are traversing an inner ccb (clockwise) or an outer ccb (counter-clockwise). It is convertible to a `Halfedge_handle`.

- A `Inner_ccb_iterator` is used to go over all holes in a specific face. It is convertible to a `Ccb_halfedge_circulator` and hence to a `Halfedge_handle`.

- A `Isolated_vertex_iterator` is used to go over all isolated vertices contained in a specific face. It is convertible to `Vertex_handle`.

As an example we define a free function to traverse the halfedges of a ccb and count them.

```
int num_of_ccb_halfedges(Ccb_halfedge_circulator start)
{
  int num = 0;
  Ccb_halfedge_circulator current = start;

  do
  {
    num++;
    current++;
  }while(current != start);

  return num;
}
```

We are interested in the complexity of a face. Let's say we have a `Face_handle fh` to any face in an arrangement. The combinatorial complexity of this face is the number of halfedges on all its ccbs. We can obtain this number by traversing the outer ccb (if it exists) and all of the face's holes:

```
int face_complexity (Face_handle fh)
{
  int complexity = 0;

  // Traverse the outer ccb if it exists
  if ( !(fh->is_unbounded()) )
  {
    int num = num_of_ccb_halfedges( fh->outer_ccb() );
    complexity += num;
  }

  // Traverse the boundary of each of the holes.
  Inner_ccb_iterator ici = fh->holes_begin();
  for (; ici != fh->holes_end(); ++ici)
  {
    int num = num_of_ccb_halfedges(ici);
    complexity += num;
  }

  return complexity;
}
```

We will employ this function later.

3.1.3 Insertion functions

We can solve our single face problem in quadratic worst-case time using the functionality of the CGAL arrangement package. Wherever we speak of a *vector* from now on we mean a data structure like the `std::vector`, i.e. a set with random access to its elements. A call to `vec.size()` on a vector `vec` gives us the number of elements in the vector.

Let `xcurves` be a `std::vector<X_monotone_curve_2>` containing our input segments and `point_x` the `Point_2` contained in the single face we are looking for. We create an empty arrangement, insert the point and the segments as efficiently as possible and obtain a handle to the face containing `point_x`:

```
Arrangement_2 arr;
Face_handle uf = arr.unbounded_face();
Vertex_handle vh_x = arr.insert_in_face_interior(point_x,
    uf);
insert(arr, xcurves.begin(), xcurves.end());
Face_handle fh_x = vh_x->face();
```

The specialized insertion function `insert_in_face_interior` inserts `point_x` as an isolated vertex in the interior of face `uf`. It is implemented as a member function of `Arrangement_2`, does not require any geometric computation and is therefore very efficient. Functions that require non-trivial geometric algorithms are implemented as free functions in the arrangement package. The overloaded, aggregated insertion function `insert` in our case sweeps over the input curves, computes their intersection points and constructs the full arrangement data structure. This process takes time

$$O\Big(\big(\texttt{xcurves.size()} + k\big)\log\big(\texttt{xcurves.size()}\ \big)\Big),$$

where k is the number of intersection points, which can be quadratic in the number of segments. This aggregated construction is usually much faster than an incremental construction. See [33] for a comparison. Obtaining the `Face_handle` takes constant time.

The call to our function `face_complexity(fh_x)` will give us the complexity of the face containing `point_x`. We will prove an upper bound for the complexity in the next section.

Aside from `insert_in_face_interior` to insert points, there is another specialized insertion function that is important to us. The function

```
Halfedge_handle arr.insert_at_vertices(X_monotone_curve_2
    c, Vertex_handle v1, Vertex_handle v2)
```

inserts `c` as an edge between the vertices `v1` and `v2`. We must be sure that `c` is interior disjoint from all existing arrangement edges and vertices and that `v1` and `v2` correspond to the endpoints of `c`.

These two insertion functions work exclusively on the combinatorial structure without doing any time-consuming geometric operations. This

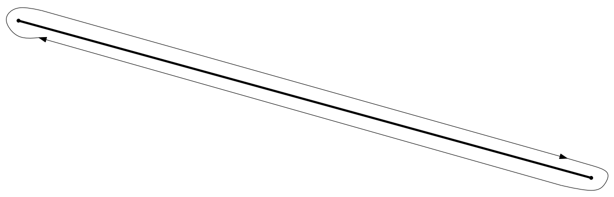


Figure 3.4: A 'singleton' arrangement

makes them very efficient and therefore we will only use these two functions to insert points or segments into an arrangement in our implementation of the algorithm to compute a single face.

3.2 Complexity of a single face

Although in this work we concentrate on arrangements of bounded line segments, the following theoretical insights into the combinatorial complexity of a single face will hold for more general bounded x-monotone arcs of Jordan curves. For the sake of exposition we make the following non-degeneracy assumptions:

- No two endpoints or intersection points of the input curves share the same x-value.
- Not more than two curves intersect in any point.
- Every intersection point is transversal, i.e. there are no weak intersections and no overlapping curves.

We introduce the notion *oriented origin* that is useful for our analysis and could be implemented as a function `oriented_origin()` for halfedges.

Imagine we created a vector `singletons` of 'singleton' arrangements by inserting every input curve `xcurves[i]` in its own empty arrangement `singletons[i]`. Every singleton arrangement would consist of exactly two halfedges. Consider figure 3.4. Note that the underlying `X_monotone_curve_2` object does not need to be a segment. The full arrangement `arr` is the overlay of all these singleton arrangements.

Let `he` be a handle to a halfedge in the complete arrangement `arr` of all curves in `xcurves`. Calling `he->curve()` gives us the underlying, non-oriented `X_monotone_curve_2` object which is a portion of exactly one of the curves in `xcurves`, let's say of `xcurves[i]`. By `he->oriented_origin()` we refer to the `Halfedge_handle` pointing to the halfedge in the singleton arrangement `singletons[i]` oriented in the same direction as `he`. So `he->curve()` is a piece of `he->oriented_origin()->curve()`. Note that

```
he->twin()->oriented_origin() == he->oriented_origin()->
    twin().
```

Theorem 3 Let `xcurves` be a vector of `X_monotone_curve_2` objects so that each pair of them intersects in at most s points. Insert these curves into an empty `Arrangement_2 arr`. Let `f` be a `Face_handle` to any single face in `arr`. The result of `face_complexity(f)` is in

$$O\Big(\lambda_{s+2}\big(\texttt{xcurves.size()}\big)\Big)$$

where $\lambda_s(n)$ is the maximal length of a Davenport-Schinzel sequence of the order s from an alphabet of n letters [1].

Since every `X_monotone_curve_2` cannot contribute to more than one ccb of any specific face, it suffices to show that if n x-monotone curves contribute to this one ccb then the number of halfedges along it is in

$$O\Big(\lambda_{s+2}\Big(n\Big)\Big).$$

Without loss of generality we can consider the outer ccb of a bounded face `f`, since we could latch any hole into the outer ccb to merge the total complexity into one ccb. In the worst case, all input x-monotone curves contribute to the outer ccb.

Let `ccb_halfedges` be a vector of the halfedges along `f->outer_ccb()` in circular order. We define another vector `ccb_origins` with

```
ccb_origins[i] = ccb_halfedges[i]->oriented_origin().
```

Since n curves contribute to the ccb, there are $2n$ oriented original halfedges. This is the maximum number of unique elements in `ccb_origins`, which can be regarded as the letters of an alphabet for a Davenport-Schinzel sequence. For each unique element of `ccb_origins[i]` we could acquire an *induced edges vector* of the halfedges it induces along our ccb. We could sort this vector so that its elements are arranged in the linear order they appear along their oriented origin.

Lemma 4 *(Consistency lemma)* Any two halfedges `he1` and `he2` in `ccb_halfedges` with

```
he1->oriented_origin() == he2->oriented_origin()
```

appear in a circular order in `ccb_halfedges` that is consistent with their linear order along their oriented origin.

Proof. Let `he1` and `he2` be any pair of halfedges with the same oriented origin appearing consecutively along the ccb in the sense that no further halfedge with the same oriented origin appears between them. We will show that the portion of the oriented origin between `he1` and `he2` cannot induce a halfedge breaking the aforementioned consistency anywhere along the ccb.

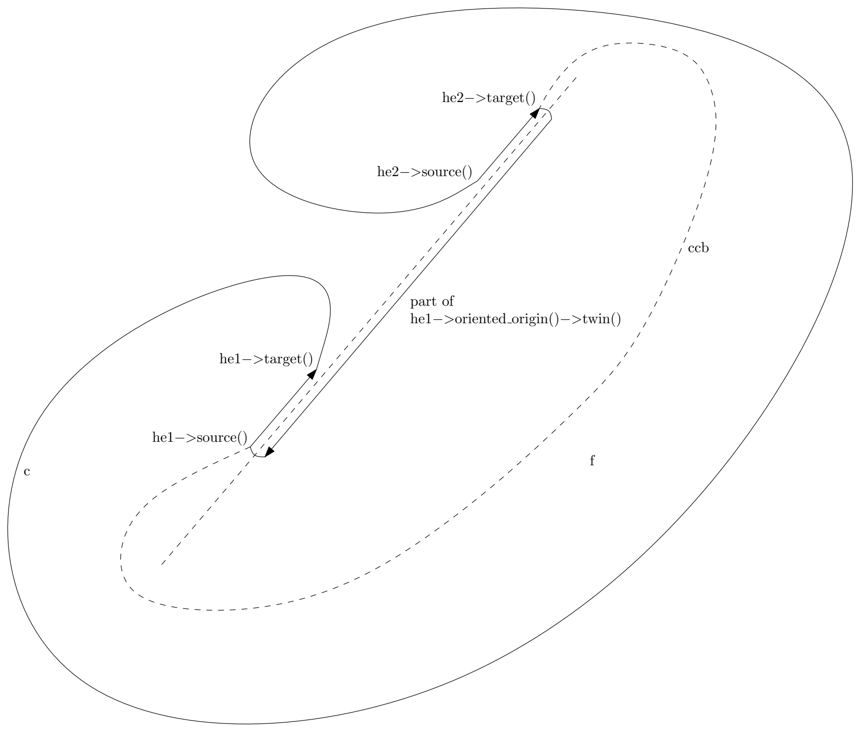


Figure 3.5: Consistency

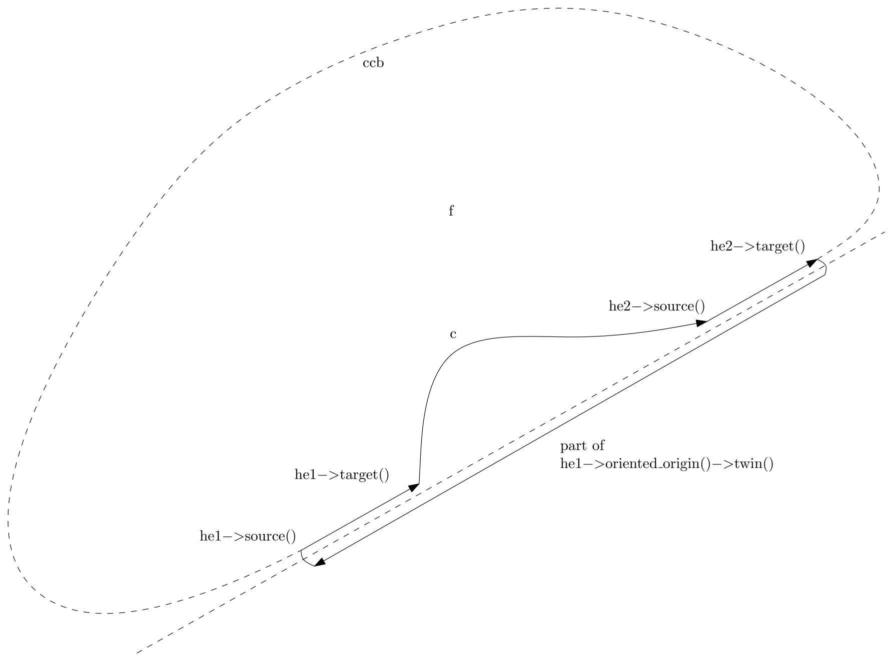


Figure 3.6: Consistency

The face `f` always lies to the left of the halfedges `he1` and `he2`. Consider figures 3.5 and 3.6.

Connect the vertex `he1->target()` with the vertex `he2->source()` by the path in a circular traversal of the ccb. Connect `he1->source()` with `he2->target()` in the opposite direction along the halfedge `he1->oriented_origin()->twin()`. We get a closed Jordan curve `c` with the face `f` fully contained either in its exterior or in its interior. Every point `p` on `he1->oriented_origin()` lying between the vertices `he1->target()` and `he2->source()` is contained in the side of `c` not containing `f`. So `p` cannot lie on a halfedge in `ccb_halfedges`. □

From now on we want to regard `ccb_halfedges` as a linear rather than a circular sequence. There is a one-to-one mapping of every element `ccb_halfedges[i]` to exactly one element in exactly one of the induced edges vectors associated with the original halfedges.

Visualize each element of an induced edges vector placed below the `ccb_halfedges` vector. Some of the induced edges vectors might lose their linear order because of the circular character of `ccb_halfedges`, i.e. they could overshoot the end of `ccb_halfedges`. We want to get rid of that effect. The original halfedge corresponding to such an overshooting induced edges vector must be split in a way that we get two induced edges vectors that keep their linear order when their elements are placed below `ccb_halfedges`. In the worst case we have to split all $2n$ original halfedges and we end up with $4n$ original halfedges. We update our data structures accordingly and get a new adapted `ccb_origins` vector, which can now be composed of up to $4n$ unique original halfedges. The sequence `ccb_origins` can be considered as a word from an alphabet of $4n$ letters.

Lemma 5 The elements of the vector `ccb_origins` represent a Davenport-Schinzel sequence of the order $(s + 2)$ from an alphabet of $4n$ letters.

Proof. No two consecutive halfedges in `ccb_halfedges` can be induced by the same original halfedge, so there are no two adjacent equal elements in `ccb_origins`. We prove the fact that there are no subsequences with more than $s + 2$ alternations by contradiction. Suppose there is a subsequence of original halfedges `he1` and `he2`

$$\left(\ldots\ \texttt{he1}\underbrace{\ldots}_{1}\texttt{he2}\underbrace{\ldots}_{2}\texttt{he1}\underbrace{\ldots}_{3}\texttt{he2}\ \ldots\ \ldots\ \ldots\ \texttt{he2}\underbrace{\ldots}_{s+1}\texttt{he1}\underbrace{\ldots}_{s+2}\texttt{he2}\underbrace{\ldots}_{s+3}\texttt{he1}\ \ldots\right)$$

of length $s + 4$ with $s + 3$ alternations.

We put any four consecutive elements of this alternation in the sequence together into overlapping quadruples. A quadruple starts at every element except for the last three elements. We get $s + 4 - 3 = s + 1$ quadruples.

Without loss of generality we consider a quadruple starting with `he1`. During the now linear traversal of the ccb we pass two halfedges induced by

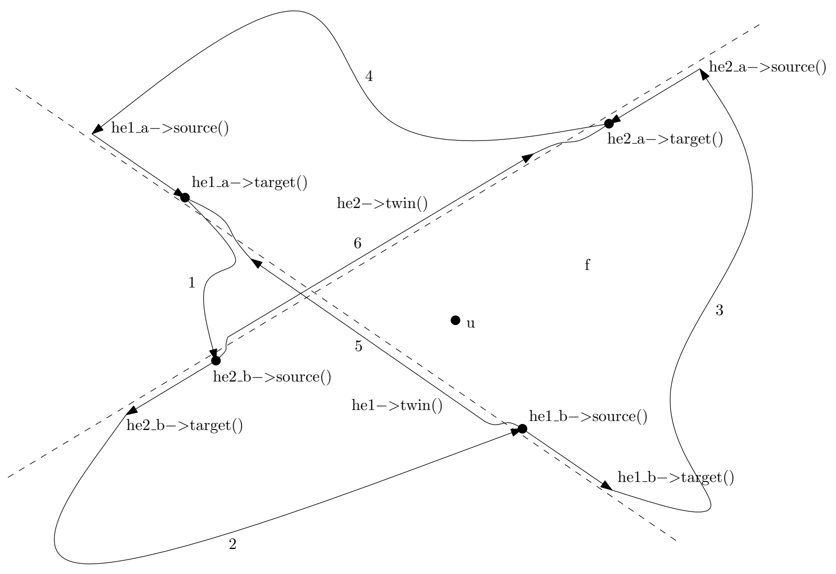


Figure 3.7: A quadruple

`he1` (`he1_a` and `he1_b`) and two halfedges induced by `he2` (`he2_a` and `he2_b`). Consider figure 3.7. We connect their vertices in the following manner:

1. Connect `he1_a->target()` with `he2_b->source()` by the path in a counterclockwise traversal of the ccb.
2. Connect `he2_b->target()` with `he1_b->source()` by the path in a counterclockwise traversal of the ccb.
3. Connect `he1_b->target()` with `he2_a->source()` by the path in a counterclockwise traversal of the ccb.
4. Connect `he2_a->target()` with `he1_a->source()` by the path in a counterclockwise traversal of the ccb.
5. Connect `he1_b->source()` with `he1_a->target()` along the original halfedge `he1->twin()`.
6. Connect `he2_b->source()` with `he2_a->target()` along the original halfedge `he2->twin()`.

The first four arcs are pairwise non-intersecting. It seems obvious from figure 3.7 that there is an intersection between `he1` and `he2` associated with every such quadruple. We will give a formal proof by contradiction.

Let's assume all above mentioned arcs not to intersect each other. We insert an additional vertex `u` into the face `f`. We can add non-intersecting edges to connect `u` with the sources of the four induced edges `he1_a`, `he1_b`, `he2_a` and `he2_b`. By doing that, we would have embedded the graph K_5 in the plane $\lightning$.

In fact every such quadruple induces an intersection between `he1` and `he2`. Since we have $s+1$ quadruples we end up with $s+1$ intersections which would give us a contradiction to our premise of a maximum of s intersection points between any pair of curves. But of course the quadruples overlap. Does every quadruple really produce a distinct intersection point? This follows from the consistency lemma when we compare one quadruple to the next:

$$(\texttt{he1}, \texttt{he2}, \texttt{he1}, \texttt{he2})$$
$$(\texttt{he2}, \texttt{he1}, \texttt{he2}, \texttt{he1})$$

Here the sequence has 'moved on' along `he1` since the intersection point associated with the first quadruple. So either the 5. or the 6. arc associated with the second quadruple is disjoint from the according arc in the first quadruple. □

Lemma 5 finishes the proof of Theorem 3 since it implies

$$\texttt{ccb\_origins.size()} \in O\Big(\lambda_{s+2}\big(4n\big)\Big) = O\Big(\lambda_{s+2}\big(n\big)\Big)$$

and `ccb_halfedges` traversed and counted by the function `face_complexity` is of the same size as `ccb_origins`.

We have just proven an upper bound for the complexity of any single face in an arrangement of bounded x-monotone Jordan arcs. In the following we aim for a deterministic divide and conquer algorithm that runs in time

$$O\Big(\lambda_{s+2}\big(n\big) \cdot \log^2\big(n\big)\Big)$$

to compute such a single face. The upcoming result will be vital to keep the complexity of the merge procedure under control.

3.3 Combination Lemma

Let `isolated_points` be a vector of `Point_2` objects which are contained in both an arrangement `red_arr` and an arrangement `blue_arr` as isolated vertices. No more isolated vertices are contained in either `red_arr` or `blue_arr`. Any of these isolated vertices is contained in exactly one face in `red_arr` and in exactly one face in `blue_arr`, potentially in the unbounded faces.

Let the arrangement `purple_arr` be the overlay of `red_arr` and `blue_arr`, also containing all `isolated_points`. Let `red_faces`, `blue_faces` and `purple_faces` be vectors of `Face_handle`s to all faces containing isolated points. Each of those faces is represented only once, even if it contains multiple isolated points. We define

$$\texttt{red\_complexity} = \sum_{\texttt{i=0}}^{\texttt{red\_faces.size()-1}} \texttt{face\_complexity(red\_faces[i])}$$

and `blue_complexity` and `purple_complexity` accordingly.

Lemma 6 *(Combination lemma)* The sum `purple_complexity` is in

$$O\big(\texttt{red\_complexity} + \texttt{blue\_complexity} + \texttt{isolated\_points.size()}\big).$$

3.3.1 A single point

We start slowly by considering the case of just one isolated point, represented in `red_arr` by a `Vertex_handle red_isolated_vertex`, in `blue_arr` by a `Vertex_handle blue_isolated_vertex` and in `purple_arr` by a `Vertex_handle purple_isolated_vertex`. We define:

```
Face_handle red_face = red_isolated_vertex->face();
Face_handle blue_face = blue_isolated_vertex->face();
Face_handle purple_face = purple_isolated_vertex->face();
```

A face can consist of multiple ccbs; one or no outer ccb and any number of holes. We can imagine having stored these ccbs in vectors `red_ccbs`, `blue_ccbs` and `purple_ccbs`. We claim the following (claim I):

$$\begin{aligned}&\texttt{face\_complexity(purple\_face)} <= \\ &\quad (s+2)\cdot\Big(\texttt{face\_complexity(blue\_face)} + \\ &\quad \texttt{face\_complexity(red\_face)} + \\ &\quad 2\cdot\texttt{red\_ccbs.size()} + \\ &\quad 2\cdot\texttt{blue\_ccbs.size()} - \\ &\quad 4\cdot\texttt{purple\_ccbs.size()}\Big)\end{aligned}$$

Without loss of generality we assume `purple_face` to be bounded and take a closer look at `purple_face->outer_ccb()`. Comparable to the situation in the proof of Theorem 3 we assume we can call `purple_he->oriented_origin()` on a `Halfedge_handle purple_he` along the outer ccb of `purple_face` to get a handle to the red or blue halfedge in `red_arr` or `blue_arr` of which `purple_he` is a portion of. Note that many purple halfedges can have the same oriented origin. We obtain a vector `ccb_halfedges` of the halfedges along the purple outer ccb and a vector `ccb_origins` of the same size consisting of red and blue original halfedges.

Let purple_he1 and purple_he2 be two elements of ccb_halfedges with the same oriented origin, say red_he, such that purple_he1 precedes purple_he2 in linear order along red_he and no element between purple_he1 and purple_he2 along the circular ccb has red_he as its oriented origin. Note that red_he is in its entirety and without interruptions on a red ccb of red_face. With the consistency lemma we follow that this red ccb cannot provide a portion of the purple outer ccb between purple_he1 and purple_he2. So either the oriented origins of the halfedges between (in a circular sense) purple_he1 and purple_he2 are part of a red ccb not including red_he or they are blue halfedges.

With the halfedge handles in ccb_origins we can store their color. We obtain two vectors red_origins and blue_origins by copying just the red or blue halfedge handles respectively in two empty vectors. We traverse both of these vectors to delete equal consecutive (in a circular sense) elements.

Any number of red_ccbs can contribute to the purple outer ccb we are looking at. We copy the contributing red ccbs in their own vector contributing_red_ccbs ordered circularly along the purple ccb and count all halfedges along these contributing red ccbs. The resulting number we call num_of_contributing_red_halfedges. We prove the following claim (claim II):

$$\begin{aligned}&\texttt{red\_origins.size()} <= \\ &\quad \texttt{num\_of\_contributing\_red\_halfedges} + \\ &\quad 2 \cdot \texttt{contributing\_red\_ccbs.size()} - 2\end{aligned}$$

Let's look at a halfedge he = red_origins[i] on a red ccb ccb1 that appears more than once, say it appears again at red_origins[j] = he. We know that every halfedge between (in a circular sense) position i and position j must belong to a ccb other than ccb1. We charge this repeated appearance of he on the first one of these other ccbs. Just like red_origins the vector contributing_red_ccbs has no consecutive equal elements.

By definition all ccbs in contributing_red_ccbs do not intersect each other. We prove by contradiction that contributing_red_ccbs is a circular Davenport-Schinzel-sequence with $s = 0$ and therefore a maximum complexity of

$$\begin{aligned}&\lambda_2\big(\texttt{contributing\_red\_ccbs.size()}\big) = \\ &\quad 2 \cdot (\texttt{contributing\_red\_ccbs.size()}) - 2.\end{aligned}$$

Assume there are two red ccbs that alternate more than two times forming a quadruple:

$$\left(\ldots\ \texttt{ccb1}\underbrace{\ldots}_{1}\texttt{ccb2}\underbrace{\ldots}_{2}\texttt{ccb1}\underbrace{\ldots}_{3}\texttt{ccb2}\ \ldots\right)$$

Very similar to the situation in the proof of Lemma 5 we follow that such a quadruple must always be associated with an intersection while we assumed $s = 0$ ↯.

With the `contributing_red_ccbs` being a Davenport-Schinzel sequence, it is guaranteed that our above charging scheme charges no red ccb more than once. Let's say we charged a second appearance of a red halfedge `he` along a ccb `ccb1` on another ccb `ccb2` between the two appearances of `he`. After the second appearance of `he` no halfedge from `ccb2` can come up, because this would add a third alternation between `ccb1` and `ccb2` in `contributing_red_ccbs`. And since `ccb2` doesn't appear again it won't be charged again.

Now we can prove claim II. Every repeated appearance of a red original halfedge in `red_origins` is now charged exactly once on an element of a Davenport-Schinzel sequence of the length

$$2 \cdot \texttt{contributing\_red\_ccbs.size()} - 2.$$

This proves the claim. An analogous claim can be proven about `blue_origins.size()`.

Remember that we deleted circular adjacent equal elements from the vectors `red_origins` and `blue_origins`. Before this procedure, the vectors consisted of subsequences of consecutive adjacent equal elements. We call such a maximal subsequence a *run*. In the final vector every run is represented by just its first element. We mark these first halfedges of each run. Let us transfer this to the vector `ccb_halfedges`. It includes

$$\begin{aligned}&\texttt{red\_origins.size()} + \texttt{blue\_origins.size()} <= \\ &\quad \texttt{num\_of\_contributing\_red\_halfedges} + \\ &\quad \texttt{num\_of\_contributing\_blue\_halfedges} + \\ &\quad 2 \cdot \texttt{contributing\_red\_ccbs.size()} + \\ &\quad 2 \cdot \texttt{contributing\_blue\_ccbs.size()} - 4\end{aligned}$$

marked elements now. What is going on in `ccb_halfedges` between two consecutive marked elements? The in-between might look like this:

$$\ldots a \quad \underbrace{b}_{\text{marked}} \quad \underbrace{ababa}_{\text{in-between}} \quad \underbrace{c}_{\text{marked}} \quad \ldots$$

There are only two halfedges a and b of opposite color involved in the in-between because when a third run (of c) is about to get involved it will start with a marked element. We know that the number of alternations of a and b in ccb_halfedges can be $s+2$. Only s of them can occur in the in-between so the in-between has a maximum length of $s+1$. These elements have not yet been accounted for. Between any two marked elements we deleted a maximum of $s+1$ elements in red_origins and blue_origins.

The complexity of the original vector ccb_origins for one purple ccb can be up to

$$\text{(number of marked elements)} + (s+1)\cdot\text{(number of marked elements)}$$

which equates to

$$\begin{aligned}(s+2)\cdot\Big(&\texttt{num\_of\_contributing\_red\_halfedges}+\\ &\texttt{num\_of\_contributing\_blue\_halfedges}+\\ &2\cdot\texttt{contributing\_red\_ccbs.size()}+\\ &2\cdot\texttt{contributing\_blue\_ccbs.size()}-4\Big)\end{aligned}$$

We have to sum over all purple ccbs to get our face_complexity(purple_face) and finally prove claim I:

$$\begin{aligned}&\texttt{face\_complexity(purple\_face)} \le (s+2)\cdot\\ &\Bigg(\underbrace{\sum_{\texttt{purple\_ccbs}}\texttt{num\_of\_contributing\_red\_halfedges}+}_{<=\texttt{face\_complexity(red\_face)}}\\ &\underbrace{\sum_{\texttt{purple\_ccbs}}\texttt{num\_of\_contributing\_blue\_halfedges}+}_{<=\texttt{face\_complexity(blue\_face)}}\\ &2\cdot\underbrace{\sum_{\texttt{purple\_ccbs}}\texttt{contributing\_red\_ccbs.size()}+}_{<=\texttt{red\_ccbs.size()}}\\ &2\cdot\underbrace{\sum_{\texttt{purple\_ccbs}}\texttt{contributing\_blue\_ccbs.size()}-}_{<=\texttt{blue\_ccbs.size()}}\\ &\underbrace{\sum_{\texttt{purple\_ccbs}}4}_{=4\cdot\texttt{purple\_ccbs.size()}}\Bigg)\end{aligned}$$

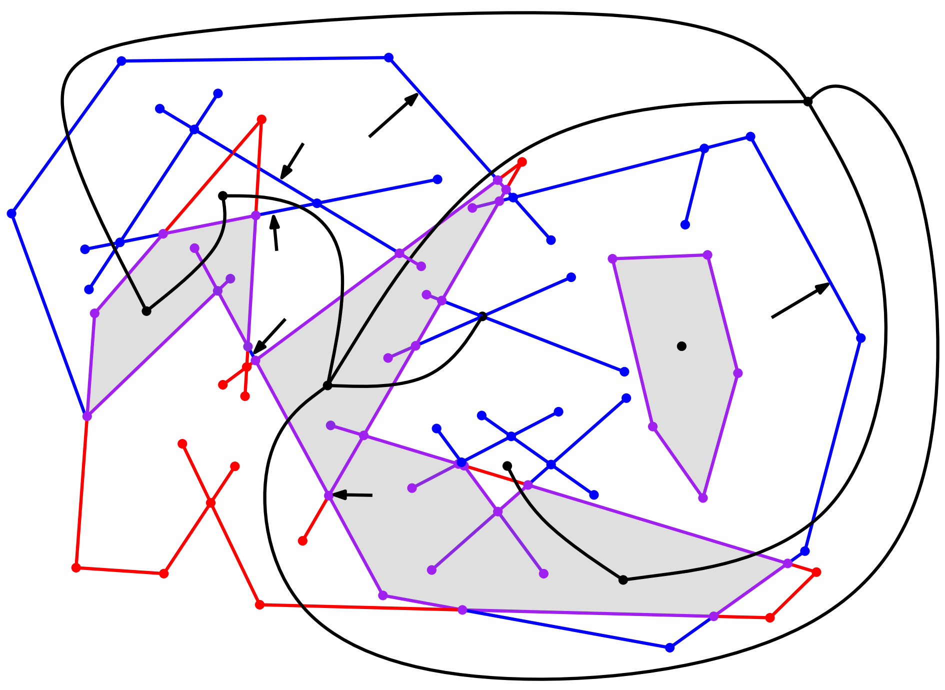


Figure 3.8: The graph. The arrows denote the points where we ripped the ccbs.

To informally summarize, we have shown that the complexity of a single purple face in the overlay of a single red and a single blue face (which can consist of many purple faces) is not too high.

3.3.2 Multiple points

We want to extend our analysis to the case of multiple isolated points in multiple purple faces since our algorithm will have to compute multiple purple faces in the merge procedure. We will explain the reasons for this later.

Let's take one face `blue_face = blue_faces[i]`, i.e. it contains isolated points. Each point will also be contained in some `purple_face`. Let's say that all in all the points in `blue_face` end up in k purple faces. Let `blue_ccbs` be the ccbs of `blue_face`.

Now we choose one isolated vertex to represent each `purple_face` and throw out the others, so we keep k points. We cut the `blue_ccbs` (by ripping the circular lists into linear lists) into portions so that each linear list induces halfedges in only one `purple_face`. Store those linear lists in `blue_ccb_portions`. How many elements can `blue_ccb_portions` have? To find out, we define planar graph G on this configuration. Consider figure 3.8.

- Every purple_face contains exactly one isolated vertex. Those become vertices of G.
- Insert one isolated vertex in blue_arr->unbounded_face() and one in each hole of blue_face. Those too become vertices of G. Note that there are blue_ccbs.size() vertices like this all lying outside of blue_face.
- Over each linear chain of halfedges in blue_ccb_portions we draw an edge in G connecting one of the isolated vertices outside of blue_face with one of the k isolated vertices representing the purple faces.

We get a bipartite planar graph G with

$$k + \texttt{blue\_ccbs.size()}$$

vertices and

$$O\Big(k + \texttt{blue\_ccbs.size()}\Big)$$

edges. The graph has exactly one edge for each element of blue_ccb_portions, so the latter bounds blue_ccb_portions.size().

For each of the k purple faces we define a modified blue_red_face in such a way that it contains the respective purple_face and is contained in blue_face.

- If purple_face is induced completely by red ccbs lying inside blue_face without intersecting an element of blue_ccb_portions, we set set blue_red_face = purple_face and stop.
- Otherwise we start traversing an element of blue_ccb_portions that has an intersection point z with the border of purple_face. From z we follow the blue ccb portion to its last intersection with purple_face while having blue_face to our left.
- At this last intersection, we follow the border of purple_face (which is now induced by a red ccb) with purple_face to our left until an intersection with another element of blue_ccb_portions. Again we follow it until its last intersection with purple_face and so on. We continue this until we reach our starting point.
- If there are elements of blue_ccb_portions left that have an intersection with purple_face but are not yet part of our blue_red_face we start another traversal in the same way as described above.
- Finally, we add red ccbs to blue_red_face that bound purple_face but are not intersected by blue ccbs.

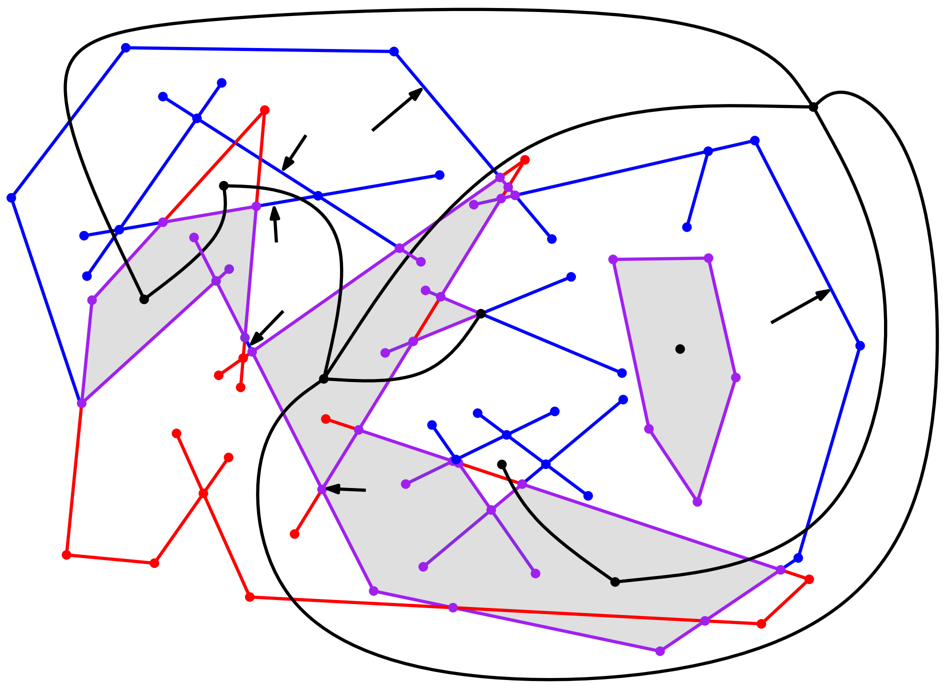


Figure 3.9: The 'blue-red' regions.

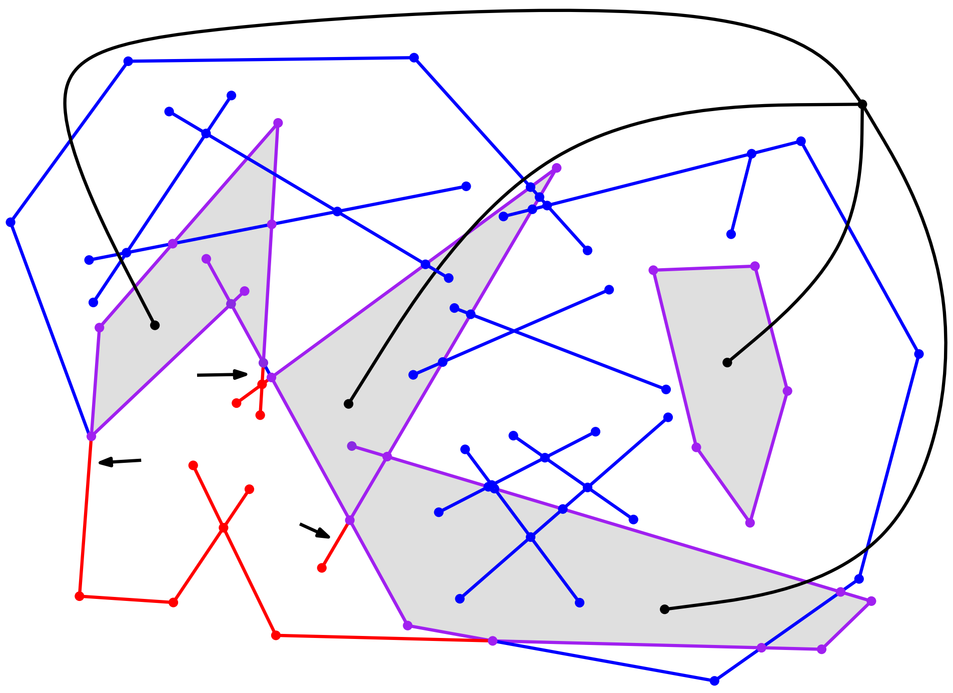


Figure 3.10: The 'red-blue' regions.

In a fully symmetric manner we can define 'red-blue' faces. Consider figure 3.9 for the 'blue-red' faces in our example and figure 3.10 for the 'red-blue' faces.

Note that each `purple_face` is now associated with exactly one point, exactly one `blue_red_face` and exactly one `red_blue_face` such that

$$\texttt{blue\_red\_face} \cap \texttt{red\_blue\_face} = \texttt{purple\_face}$$

and that no halfedge on a ccb of `blue_red_face` or `red_blue_face` can appear on a ccb of a purple face other than the associated `purple_face`.

We make an important adaption to our function `face_complexity()` specifically to count only blue halfedges when called for a `blue_red_face` and to count only red halfedges when called for a `red_blue_face`.

Every halfedge on a ccb of a `purple_face` is a (portion of a) blue halfedge in the associated `blue_red_face` or a (portion of a) red halfedge in the associated `red_blue_face`. This is a situation very similar to that in the proof of claim I in subsection 3.3.1. Remember that we have shown that for a single purple face the following holds:

$$\begin{aligned}
&\texttt{face\_complexity(purple\_face)} <= \\
&\quad (s+2)\cdot\Big(\texttt{face\_complexity(red\_face)} + \\
&\quad \texttt{face\_complexity(blue\_face)} + \\
&\quad 2\cdot\texttt{red\_ccbs.size()} + \\
&\quad 2\cdot\texttt{blue\_ccbs.size()} - \\
&\quad 4\cdot\texttt{purple\_ccbs.size()}\Big)
\end{aligned}$$

In the current situation this translates to the following:

$$\begin{aligned}
&\texttt{face\_complexity(purple\_face)} \in \\
&\quad O\Big(\texttt{face\_complexity(red\_blue\_face)} + \\
&\quad \texttt{face\_complexity(blue\_red\_face)} + \\
&\quad \texttt{red\_blue\_ccbs.size()} + \\
&\quad \texttt{blue\_red\_ccbs.size()}\Big)
\end{aligned}$$

By summing up over all purple faces (in other words: over all remaining points) with

$$n = \texttt{purple\_faces.size()-1}$$

we get

$$\begin{aligned}
&\sum_{\mathtt{i=0}}^{\mathtt{n}} \mathtt{face\_complexity(purple\_faces[i])} \in \\
&\quad O\Big(\sum_{\mathtt{i=0}}^{\mathtt{n}} \mathtt{face\_complexity(blue\_red\_faces[i])} + \\
&\quad \sum_{\mathtt{i=0}}^{\mathtt{n}} \mathtt{face\_complexity(red\_blue\_faces[i])} + \\
&\quad \sum_{\mathtt{i=0}}^{\mathtt{n}} \mathtt{blue\_red\_ccbs[i].size()} + \\
&\quad \sum_{\mathtt{i=0}}^{\mathtt{n}} \mathtt{red\_blue\_ccbs[i].size()}\Big).
\end{aligned}$$

How big can

$$\sum_{\mathtt{i=0}}^{\mathtt{n}} \mathtt{face\_complexity(blue\_red\_faces[i])}$$

become when it only counts blue halfedges as defined above? Let's look at one fixed `blue_face` that induces a number of blue-red faces as described above. These blue-red faces contained in `blue_face` can have

$$O\Big(\mathtt{face\_complexity(blue\_face)} + k + \mathtt{blue\_ccbs.size()}\Big)$$

blue halfedges. Every halfedge in `blue_face` can be touched multiple times but the number of these repeated appearances is bounded by `blue_ccb_portions.size()`, i.e. the number of edges in the bipartite graph. We sum that up over all `blue_faces`:

$$\begin{aligned}
&\sum_{\mathtt{i=0}}^{\mathtt{n}} \mathtt{face\_complexity(blue\_red\_faces[i])} \in \\
&\quad O\Big(\sum_{\mathtt{blue\_faces}} \Big(\mathtt{face\_complexity(blue\_face)} + k + \mathtt{blue\_ccbs.size()}\Big)\Big) = \\
&\quad O\Big(\sum_{\mathtt{blue\_faces}} \Big(\mathtt{face\_complexity(blue\_face)}\ \Big) + \mathtt{isolated\_points.size()}\Big)
\end{aligned}$$

The same rationale applies to the number of red halfedges in all red-blue faces. To bound the term

$$\sum_{\mathtt{i=0}}^{\mathtt{n}} \mathtt{blue\_red\_ccbs[i].size()}$$

we notice that each summand is again bounded by the number of edges in the bipartite graph, so we have

$$\sum_{\texttt{i=0}}^{\texttt{n}} \texttt{blue\_red\_ccbs[i].size()} \in O\Bigg(\sum_{\texttt{blue\_faces}} \Big(\texttt{face\_complexity(blue\_face)}\Big) + \texttt{isolated\_points.size()}\Bigg)$$

and the analogous holds for the red-blue faces. Combining all this completes our final result:

$$\texttt{purple\_complexity} \in O\big(\texttt{red\_complexity} + \texttt{blue\_complexity} + \texttt{isolated\_points.size()}\big).$$

Even if our algorithm has to compute multiple purple faces in the merge procedure, their combined complexity is under control if we can bound the number of isolated points contained in these purple faces.

Chapter 4

Algorithm and implementation

We start out with n line segments in the plane and a point `point_x` not lying on any of these segments. We want to compute the single face in the arrangement of these line segments that contains `point_x`. As we mentioned, the algorithm follows the divide and conquer paradigm [1]. The complicated part is the merge procedure that computes the 'purple' face containing `point_x` in the overlay of one 'red' face and one 'blue' face each containing `point_x`.

The merge will be conducted via two sweeps, one from left to right and one from right to left. Both their outputs will have to be combined. The cost of these two sweeps will be determined by the number of event points times the worst-case cost for the handling of one event point. It is too expensive to compute the full overlay of the red and the blue face. We know the complexities of the red face and the blue face are close to linear, but there could be a quadratic number of red blue intersections. So we have to ensure that not too many of these can become event points during the sweep.

To make sure we acquire all parts of the purple face containing `point_x` in just two sweeps, we might have to compute more than this one purple face, namely all purple faces containing a red or blue segment's original endpoint. The combination lemma guarantees that we can compute these superfluous faces without increasing the asymptotical complexity of the merge step.

My implementation takes an iterator range over a container of bounded line segments as an input. These have to comply with the following non-degeneracy conditions:

- Not more than two segments intersect in any point.
- Every intersection is transversal, i.e. there are no weak intersections and no overlapping segments.

During the sweep it is not allowed for `point_x` or a segment's original endpoint to lie immediately below a red blue intersection. These degeneracies (and some more involving vertical segments) are not taken care of and will cause the program to crash with a segmentation fault. Note that these are rare when entering input via the mouse or the input buttons in the demo program. In fact, during this whole work these degeneracies never appeared by chance, only by design. The only degeneracies taken care of are the ones produced by the 'random grid' button in the demo program, i.e. a certain configuration involving vertical segments.

We are going to classify arrangement vertices according to the following enumeration:

```
enum Vertex_type
{
  //none of below
  DEFAULT = 0,

  //vertex represents point_x
  POINT_X = 1,

  //vertex represents an original segment's endpoint
  INTERNAL_ENDPOINT = 2,

  //vertex represents an original segment's endpoint
  //in the arrangement of the other color
  EXTERNAL_ENDPOINT = 4
};
```

At one point of our procedure we have to insert 'blue' endpoints into the 'red' arrangement and vice versa. These additional vertices will be of type `EXTERNAL_ENDPOINT`. We will use the functionality of the extended dcel to equip each vertex with its type.

Only red blue intersections that end up as vertices of a purple face become event points. So the complexity of all computed purple faces bounds the number of purple events, that can arise in addition to red and blue events. At the end of the two sweeps we will have computed all purple faces containing `point_x` and/or containing red or blue endpoints.

The combination lemma guarantees that the combined complexity of these purple faces is in

$$O(\texttt{red\_complexity} + \texttt{blue\_complexity}).$$

In the case of line segments the number of events in the final merge will be in

$$O\Big(\lambda_3(n)\Big).$$

with

$$n = \texttt{xcurves.size()}.$$

As we will see, the handling of any event will cost no more than

$$O\Big(\log\big(\lambda_3(n)\big)\Big) = O\Big(\log(n)\Big),$$

so the two sweeps will run in

$$O\Big(\lambda_3(n)\log(n)\Big).$$

'Around' the sweeps we have to do some pre- and postcomputation; vertically decompose the red and blue arrangements, combine the results of the two sweeps and get rid of the superfluous purple faces. These things combined will take the same asymptotic time as the two sweeps.

By putting all this together and doing the straightforward analysis of the divide and conquer scheme (adding up the costs of every level in the recursion tree) this amounts to the following upper bound for the runtime of the whole computation:

$$O\Big(\lambda_3(n)\log^2(n)\Big).$$

In the case of line segments this equals

$$O\Big(\alpha(n)n\log^2(n)\Big)$$

where $\alpha(n)$ denotes the extremely slow-growing inverse Ackermann function which can be regarded as constant for any conceivable real-world input [1] .

4.1 Divide

The input to the free function starting the divide and conquer process is an iterator range over a container `xcurves` of `X_monotone_curve_2` objects representing bounded line segments (a `std::list<X_monotone_curve_2>` for example), a `Point_2 point_x` and a `Point_2 pivot`. The single purple face containing `point_x` will be stored in a returned `Arrangement_2` object `purple`. Since we will simulate a sweep from right to left by conducting a sweep from left to right over the rotated arrangement, we have to provide a point `pivot` which for example could be the lower left corner of the bounding box around all input segments.

By calling

```
purple = red_blue_divide_and_conquer(xcurves.begin(),
    xcurves.end(), point_x, pivot);
```

we start the following recursive function on the input:

```
Arrangement_2* red_blue_divide_and_conquer(
    Iterator begin, Iterator end,
    Point_2 point_x, Point_2 pivot)
{
  Arrangement_2* purple;

  //no input curves, return empty arrangement
  if (begin == end)
  {
    purple = new Arrangement_2();
    return purple;
  }

  Iterator iter = begin;
  ++iter;

  //one curve
  if (iter == end)
  {
    purple = singleton_arrangement(*begin, point_x);
  }
  else //more than one curve
  {
    //find the position to divide the set of curves
    Iterator div_it;
    ...

    //continue recursively
    Arrangement_2* red = red_blue_divide_and_conquer
       (begin, div_it, point_x, pivot);

    Arrangement_2* blue = red_blue_divide_and_conquer
       (div_it, end, point_x, pivot);

    //merge red and blue
    purple = red_blue_merge(red, blue, pivot);
  }
  return purple;
}
```

We recursively halve `xcurves` until we get to single line segments. In the leaves of the recursion tree we include the single segment and the isolated point `point_x` into an empty arrangement. The face `singleton->unbounded_face()` is the face that contains the vertex representing `point_x`. The method `singleton_arrangement` looks like this:

```
Arrangement_2* singleton_arrangement(
    X_monotone_curve_2 curve,
    Point_2 point_x)
{
  Arrangement_2* singleton = new Arrangement_2();
  Face_handle uf = singleton->unbounded_face();

  //insert point_x
  Vertex_handle vh_x = singleton->insert_in_face_interior
     (point_x, uf);

  vh_x->set_data(POINT_X);

  //insert the left endpoint of the curve
  Vertex_handle vh_left = singleton->
     insert_in_face_interior(curve.left(), uf);

  vh_left->set_data(INTERNAL_ENDPOINT);

  //insert the right endpoint of the curve
  Vertex_handle vh_right = singleton->
     insert_in_face_interior(curve.right(), uf);

  vh_right->set_data(INTERNAL_ENDPOINT);

  //insert the curve
  singleton->insert_at_vertices(curve,vh_left,vh_right);

  return singleton;
}
```

These functions are found in the file `Red_blue_divide_and_conquer.h`.

4.2 Conquer

4.2.1 Red blue merge

Before starting a red blue merge procedure, both the red and the blue arrangements are each defined by a single face containing an isolated vertex representing `point_x` and possible holes. Both faces can be unbounded. We compute the purple face that contains `point_x` in the overlay of the arrangements `red` and `blue`. This is done via a call to the function `red_blue_merge`:

```
red_blue_merge(Arrangement_2* red, Arrangement_2* blue,
   Point_2 pivot)
{
  //compute the vertical decompositions
  //of both arrangements
  decompose_vertically(red,blue);

  //sweep from left to right
  Arrangement_2* purple = prepare_and_sweep(red,blue);
  //rotate both arrangements
  Arrangement_2* red_rotated = rotate(red,pivot);
  Arrangement_2* blue_rotated = rotate(blue,pivot);

  //equivalent to sweep from right to left
  Arrangement_2* purple_rotated = prepare_and_sweep
     (red_rotated,blue_rotated);

  //undo the rotation...
  Arrangement_2* purple_rotated_back = rotate
     (purple_rotated,pivot);

  //... and fuse the results of the two sweeps
  Arrangement_2* purple_fused = fuse
     (purple, purple_rotated_back);

  //extract the single face containing POINT_X
  Arrangement_2* purple_single_face = single_face
     (purple_fused);

  return purple_single_face;
}
```

Let us take a quick look at the free auxiliary functions implemented and used for this step. All functions are found in the file `Red_blue_divide_and_conquer.h`.

- `Arrangement_2* decompose_vertically`
  `(Arrangement_2* red, Arrangement_2* blue)`

  This function vertically decomposes the red and the blue arrangement. We will explain the exact workings of this later.

- `Arrangement_2* prepare_and_sweep`
  `(Arrangement_2* red, Arrangement_2* blue)`

  Prepares and commissions a sweep from left to right. The result is returned in a purple arrangement.

- `Arrangement_2* rotate`
  `(Arrangement_2* arr, Point_2 pivot)`

  This function returns a rotated arrangement. The rotation is done with three objects of type `CGAL::Aff_transformation_2<Kernel>` instantiated with our specific kernel [4]. These objects represent the following transformation matrices:

$$\begin{pmatrix}1 & 0 & \texttt{-pivot.x()}\\ 0 & 1 & \texttt{-pivot.y()}\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}-1 & 0 & 0\\ 0 & -1 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}1 & 0 & \texttt{pivot.x()}\\ 0 & 1 & \texttt{pivot.y()}\\ 0 & 0 & 1\end{pmatrix}$$

We translate each point so that the pivot coincides with the the origin, rotate around the origin and undo the translation. The rotated points are included into a new arrangement. Afterwards, we include the appropriate edges between the rotated points. Since we only use the most efficient insertion functions, the process takes time

$$O\Big(\texttt{arr->number\_of\_edges()}\Big).$$

-
  ```
  Arrangement_2* fuse
  (Arrangement_2* first, Arrangement_2* second)
  ```

  This function fuses two overlapping arrangements. That is, there may be points and segments that are contained in both arrangements but there are no additional strong intersections between segments of the two arrangements and segments overlap only if they are equal. The first arrangement is copied into an empty arrangement during which the points and segments already included are memorized. While we add the second arrangement, we have to look up whether a point or segment is already included in the fused arrangement. So the function takes time

  $$O\Big(\big(n+m\big)\log\big(n+m\big)\Big)$$

  with

  $$n = \texttt{first->number\_of\_edges()}$$

  and

  $$m = \texttt{second->number\_of\_edges()}\ .$$

-
  ```
  Arrangement_2* single_face(Arrangement_2* arr)
  ```

  This function extracts the single face containing the vertex representing `point_x` from a source arrangement `arr`. First `point_x` is inserted into a new empty target arrangement. Then the ccbs of the containing face are copied into the target arrangement and the target arrangement is returned.

  This function is necessary, because the two sweeps and the fuse operation can leave us with superfluous faces. It was a bit tricky to do this only using the efficient insertion functions (handling antennas for example) but by storing additional information with the vertices and halfedges in the dcel the function runs in time

  $$O\Big(\texttt{face\_complexity}\big(\texttt{fh\_x}\big)\Big).$$

4.2.2 Vertical decomposition

We come to the explanation of the function `decompose_vertically`.

Before we can start with the sweep process, we need to obtain a special kind of vertical decomposition of the red and blue input arrangements. For a start, we insert the red vertices of type `INTERNAL_ENDPOINT` into the blue arrangement and vice versa via the function `insert_external_endpoints`. We consider these additional vertices as being of type `EXTERNAL_ENDPOINT`. Note that because of our non-degeneracy conditions these vertices will be isolated vertices in the arrangement of the other color.

We cannot straightforwardly use the efficient function `insert_in_face_interior` since we do not know the face the new vertices will be contained in. Just using the general free function

```
Vertex_handle insert_point(Arrangement_2& arr, Point_2 p)
```

would be too slow since this function makes no use of our knowledge that the points to insert are definitely isolated.

Instead we do the following. We extend our function `single_face` with the capability to insert additional isolated points besides `point_x` in the interior of the unbounded face of the target arrangement before extracting and copying the single face from the source arrangement. So if we provide a vector with the points underlying blue vertices of type `INTERNAL_ENDPOINT` to a call of `single_face` for the red arrangement, we obtain the desired extended red arrangement. This copying procedure is purely combinatorial and therefore pretty fast.

To obtain the desired decomposition for each extended arrangement, we shoot rays up and down from each vertex of type `POINT_X`, `INTERNAL_ENDPOINT` and `EXTERNAL_ENDPOINT` until they hit a feature of the arrangement. If this feature is an arrangement edge, we split the edge at the intersection between the ray and the edge by inserting a new vertex. Consider figure 4.1. Note that there is a vertex above and below (typically of type `DEFAULT`) any of the relevant non-`DEFAULT` vertices.

The actual ray-shooting is done by the function `decompose` provided with CGAL's arrangement package [34], which sweeps over the red and blue arrangements. We adapt its output for our purposes and split the edges in the function `split_edges` which involves geometric intersection computations and split operations. Aside from `decompose` the mentioned functions can be found in `Red_blue_divide_and_conquer.h`.

4.2.3 Preparing the sweep

The sweep line framework of CGAL doesn't work on the arrangement directly but sweeps over geometric points and curves of the traits class. This is an obstacle for us because we stored important information about the vertex types in the dcel, which will be lost if we extract the geometric objects

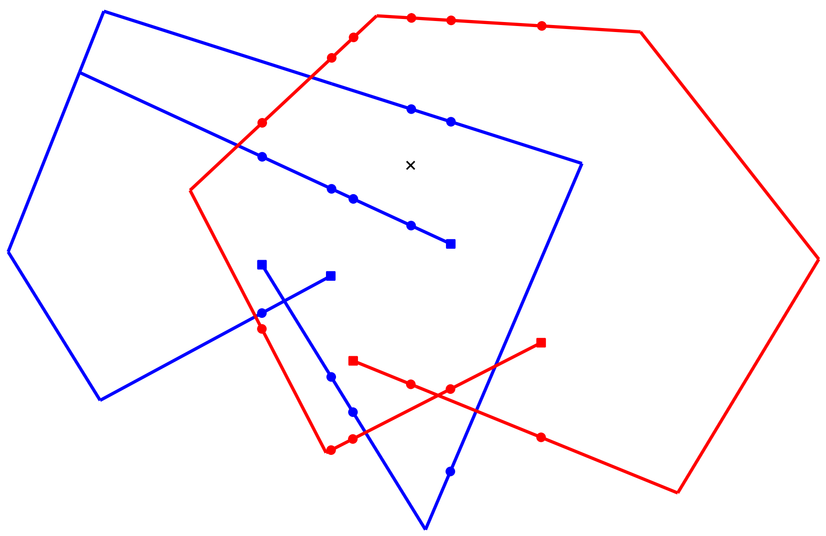


Figure 4.1: The vertical decomposition. The cross marks `point_x`. Segment endpoints are shown as boxes, the split vertices as discs.

from the arrangement. This problem is resolved by defining an extended (or meta) traits class which enables us to store information directly with extended `Point_2` and `X_monotone_curve_2` objects.

A similar functionality to the one we need can already be found in CGAL in the undocumented class template `Arr_overlay_traits_2` which is used while overlaying arrangements (see free function `overlay()` in [34]). This could be used as a basis for a customized class template `My_Arr_overlay_traits_2`.

Most importantly, with every geometric point we store three `Vertex_handle` objects which point to the vertices that represent the underlying point in the red, blue or purple arrangement respectively. Any of these might be equivalent to null pointers.

The sweep line will sweep over these extended meta-points and meta-curves. In summary, the function `prepare_and_sweep` extracts the geometric objects from the arrangements, initializes the extended geometric objects and starts the sweep. The preparation procedure runs in time

$$O\Big(\big(r+b\big)\log\big(r+b\big)\Big)$$

with

$$r = \texttt{red\_complexity}$$

and

$$b = \texttt{blue\_complexity}.$$

4.2.4 The CGAL sweep line framework

The CGAL Arrangement package contains a flexible implementation of the well-known sweep line paradigm. The framework is not documented in detail; see [33] for a brief overview.

The

```
Sweep_line_2<Traits,Event,Subcurve,Visitor>
```

class template implements a generic sweep-line algorithm. It maintains the sweep status structure and the event queue during the sweep. Both data structures are efficiently implemented as binary search trees using the class template `CGAL::Multiset` which is an extension of STL's `multiset` [24]. We will use the terms sweep line and status line interchangeably.

The following classes must be provided to instantiate the `Sweep_line_2` class template:

- The `Traits` class provides the geometric types `Point_2` and `X_monotone_curve_2`. `Point_2` objects will be associated with an `Event`. Portions of `X_monotone_curve_2` objects will be wrapped in a `Subcurve` wrapper object to become elements of the sweep status structure.

- The `Event` class provides functionality to store information associated with an event, most notably the event point. The `Event` objects will be stored in the event queue, lexicographically ordered by their respective event points.

- The `Subcurve` class is a wrapper class for `X_monotone_curve_2` objects. It stores additional information to be used during the sweep process. The sweep status structure essentially consists of a multiset of `Subcurve` objects, ordered lexicographically by their intersection point with the sweep line.

- The `Visitor` is a realization of the *visitor design pattern*; see [18]. It is supposed to handle the 'non-generic' part of a particular sweep line algorithm, including the output.

The `Sweep_line_2` class is able to maintain one event queue and one sweep status structure while handling all degenerate cases in an arrangement of curves, namely isolated points, vertical segments, more that two curves intersecting in one point, weak intersections and even overlapping curves.

At an event point, the sweep status structure is updated; starting curves are inserted, ending curves are deleted. At an intersection point, the respective curves are split.

At each event the implementation provides two sorted lists for the visitor, a list of left `Subcurve` objects and a list of right `Subcurve` objects incident to the event point. Before, during and after the handling of an event, the

`Sweep_line_2` class sends notifications to the visitor, who can then work on the neatly maintained data to produce an output. CGAL's arrangement package provides a number of visitors that employ the generic sweep line implementation. Just a few examples:

- `Arr_vert_decomp_sl_visitor` is used to compute the vertical decomposition of an arrangement. This is how the above-mentioned function `decompose` is implemented.
- `Arr_overlay_sl_visitor` is used to compute the full overlay of two arrangements.
- `Sweep_line_points_visitor` is used to report all intersection points among a set of input curves.

The implementation of many new sweep line algorithms can be reduced to the implementation of an appropriate visitor class.

Yet the visitor approach didn't seem suitable for our purpose. We need multiple independent sweep status structures while the framework is designed to maintain a single sweep status structure. Because of this limitation the visitor notification scheme was not applied for the implementation of the red blue merge. Instead, the code of the classes in the sweep line framework was directly adapted to make it handle red and blue curves independently.

The following files were studied, adapted and utilized. All files are published under the open source Q Public license 1.0 [29] and carry the following copyright notice:

```
// Copyright (c) 2005  Tel-Aviv University (Israel).
// All rights reserved.
```

In brackets after each file name we refer the authors mentioned in the source code.

- `Basic_sweep_line_2.h` (Baruch Zukerman, Tali Zvi)

  Declaration of the class template `Basic_sweep_line_2` which implements the sweep line algorithm for x-monotone curves that are pairwise interior disjoint.

- `Basic_sweep_line_2_impl.h` (Baruch Zukerman, Efi Fogel, Tali Zvi)

  Function definitions.

- `Sweep_line_2.h` (Baruch Zukerman, Tali Zvi)

  Declaration of the class template `Sweep_line_2` which extends the functionality to handle intersecting x-monotone curves as well as degenerate cases, in particular vertical segments, weak intersections, multiple intersections in one point and overlapping curves.

- `Sweep_line_2_impl.h` (Baruch Zukerman, Efi Fogel, Tali Zvi)

  Function Definitions.

- `Sweep_line_2_debug.h` (Baruch Zukerman)

  Function definitions for debug purposes.

- `Sweep_line_event.h` (Tali Zvi, Baruch Zukerman, Ron Wein, Efi Fogel)

  Definition of the class template `Sweep_line_event`.

- `Sweep_line_functors.h` (Tali Zvi, Baruch Zukerman, Ron Wein, Efi Fogel)

  This file defines two class templates:

  - The class template `Compare_events` provides a functor to compare events lexicographically by their event points in order to maintain the event queue.
  - The class template `Curve_comparer` provides functors to compare points and subcurves according to their order along the status line.

- `Sweep_line_subcurve.h` (Tali Zvi, Baruch Zukerman, Ron Wein)

  Definitions of the class template `Sweep_line_subcurve` which is a wrapper class for an object of type `X_monotone_curve_2`. It stores additional information about the curve needed for the sweep line procedure.

- `Arr_overlay_traits_2.h` (Baruch Zukerman, Ron Wein, Efi Fogel)

  Definitions of the class template `Arr_overlay_traits_2` which defines an extended (or meta) traits class which enables us to store information directly with extended `Point_2` and `X_monotone_curve_2` objects.

The first step was to copy these files into a new working directory and rename all files and classes by putting a `My_` in front of every class or file name. The next step was coarse but sure to succeed: the status line and every function dealing with maintaining the status line was duplicated. Instead of one object of type `m_statusLine` representing the sweep status structure we now have two objects `m_red_statusLine` and `m_blue_statusLine`. A large number of functions in the sweep line framework had to be duplicated like this; functions like `handle_left_curves()` for example become `handle_red_left_curves()` and `handle_blue_left_curves()`.

4.2.5 Important adaptions of the existing framework

- Duplication of functions as mentioned above to allow the framework to maintain a red and a blue sweep status structure.

- Addition of a purple sweep status structure, i.e. an ordered set of *scouts*. More on scouts later.

- The functor class `My_Curve_comparer` must be able to compare intersecting (non-interior disjoint) curves along the status line at a certain event point. This was not yet implemented (or not anymore) since it apparently is not needed for any other sweep line algorithm in the arrangement package.

  We want to postpone the computation of the intersection point as long as possible, since it is the most time-consuming operation. The functor takes two pointers to subcurves `c1` and `c2` and works like this:

  1. Check the pointers for equality.
  2. If the subcurves share a common left endpoint employ the `Compare_y_at_x_right_2` functor from the traits class.
  3. If the subcurves share a common right endpoint employ the `Compare_y_at_x_left_2` functor from the traits class.
  4. Check the relative position of `c1`'s endpoints to `c2` using the functor `Compare_y_at_x_2` from the traits class. If both are smaller, then `c1` is smaller than `c2`. If both are larger, then `c1` is larger than `c2`.
  5. Repeat the latter step with switched roles.
  6. Compute the intersection between `c1` and `c2`.
  7. If the current event point is left from the intersection point employ the `Compare_y_at_x_left_2` functor.
  8. If the current event point is right from the intersection point employ the `Compare_y_at_x_right_2` functor.

- Based on `My_Curve_comparer`, a functor class `Scout_comparer` was implemented to compare the relative position of scouts along the status line.

- The sweep line maintains a purple arrangement to insert the output. All red and blue event points are inserted before the sweep. During the sweep we have to insert red blue intersections as vertices and segments that are passed by the sweep line while being followed by scouts as edges. We only use the most efficient combinatorial insertion functions.

- At a red blue intersection the blue left curve and the red left curve were not available after the 'generic' handling of an event, so in this case the inner workings of the generic event handling had to be adapted.

- The subcurves store pointers to related scouts.

The better part of my additions to the sweep line framework is found in the files `My_Sweep_line_handler.h` and `Scout.h` which will be explained in detail below.

4.2.6 Scouts

We want to start with an informal description of the sweep line process. At first, we initialize the event queue with all red and all blue points. During the sweep we will start 'purple regions' on the status line that yield parts of the purple faces we look for. Every purple region is enclosed by exactly one pair of *scouts* that moves along with the sweep line on the boundaries of the region. The upper boundary is 'manned with' or 'followed by' exactly one *upper scout*, the lower boundary by exactly one *lower scout*. A scout manning a red curve is a red scout, a scout manning a blue curve is a blue scout. A scout may or may not 'guard' or 'observe' a curve of the manned curve's opposite color.

Those parts of a curve that are passed by the sweep line while being followed by a scout will be inserted into the purple arrangement. At every point underlying a vertex of type `POINT_X` or `INTERNAL_ENDPOINT` we will end and / or start regions:

- At `point_x` we may end one region and start one region.
- At a left endpoint we may end one region and start two regions.
- At a right endpoint we may end up to two regions and start one region.

Ending a region corresponds to deleting two scouts, starting a region corresponds to spawning two scouts. The lowest and the highest purple regions on the status line can be unbounded. These cases we handle with 'dummy' scouts that man dummy fictitious curves above and below.

In addition to the red and blue sweep status structure we maintain a purple sweep status structure, which is implemented as a `Multiset` of scouts ordered by the scout's followed curve's position along the status line.

The job of the scouts is to look out for changes in the boundary of the region they are manning. An upper red scout for example has to observe the closest blue curve above him, i.e. it has to test his boundary curve for intersection with the blue curve above. If it finds an intersection, the intersection point must be inserted into the event queue. If the sweep line reaches that point, the scout must jump on the blue curve because the latter will now be the upper boundary of the purple region. After the switch, the scout has to observe the red curve now it was manning just before. This is of course not necessary in case of line segments, since the segments will not intersect again. The algorithm is designed to work for more general x-monotone Jordan arcs though.

Another event the scouts have to look out for is the meeting of a pair of scouts enclosing a purple region. At this point the purple region will end without a new region being started. If the upper and lower scout are of the same color, their meeting point will already be contained in the event queue. If not, we duly have to compute a possible red blue intersection between the upper boundary curve the upper scout is manning and the lower boundary curve the lower scout is manning.

Scouts can either be *active* or *idled*. An idled scout is not observing any curve because it is protected from curves breaching its boundary by a vertically neighbored scout of the same color.

To maintain the algorithmic flow and to keep the runtime under control, the following invariant must hold after every event:

Every curve is observed by at most one lower scout and by at most one upper scout which are manning curves of the observed curve's opposite color.

How do we make sure of this when starting or ending a region?

- When ending a region, both scouts are deleted. If the upper scout was a red scout guarding a blue curve above it, it was the only upper scout guarding this curve due to the invariant. Therefore we have to check the highest red upper scout below to see if it was idled by the deleted scout. If it was, it has to guard the blue curve from now on. Analogously, we inform the lowest lower scout above of the same color about the deletion of the lower scout of the ended region.

- When starting a region, two scouts are spawned. A new red upper scout has to search the blue sweep status structure for the lowest blue curve above. This blue curve may already be guarded from below and it is not allowed to be guarded by two upper scouts. Then there are two cases to consider:

  - The blue curve is guarded by a red scout above the newly spawned scout. Then the newly spawned scout can forget about the blue curve for the time being and become idled. It does not need to test for intersection, because it is protected by the red upper scout above.
  - The blue curve is guarded by a red scout below the newly spawned scout. Then the newly spawned scout must take over the guarding duties from the red upper scout below. The red upper scout below becomes idled and forgets about the blue curve it was guarding.

  Analogous actions have to be taken for a newly spawned lower scout.

Similar reassignments have to be done at a red blue intersection.

Points of type `DEFAULT` are always monochromatic and not endpoints of an original segment. We may have to end a region if a scout pair enclosing a purple region meets at a monochromatic point. Remaining scouts on the curves left from the event point just jump over to the appropriate curve of the same color right from the event point. Scouts guarding a left curve just guard the appropriate right curve from now on.

We will take a more detailed look at the possible events in the upcoming description of our implementation.

4.2.7 The `Scout` class

The class `Scout` is defined in the file `Scout.h`. With a `Scout` object we basically store the following data:

- A pointer to the one `Subcurve` the scout is following called the *boundary curve*. This might be fictitious if the scout is a dummy.
- A pointer to the one `Subcurve` the scout is guarding called the *guarded curve*. This might be null if the scout is a dummy or idled.
- A pointer to its partner scout in enclosing a region. Every scout has exactly one partner, possibly a dummy scout.
- A boolean value whether the scout is active or idled.
- A `Scout_set_iterator` to its position in the scout set for efficient insertion and erasure.

Any scout is of one of the following types:

```
enum Scout_type
{
  UPPER = 0,
  LOWER = 1,
  UPPER_DUMMY = 2,
  LOWER_DUMMY = 4
};
```

The purple sweep status structure consists of a `Multiset` of pointers to scouts that is iterated via a `Scout_set_iterator`. The scouts are ordered along the sweep line according to their boundary curves. We defined the functor `Scout_comparer` based on `My_Curve_comparer` to provide the ability to compare scouts according to the order just mentioned. This functor is used to instantiate the `Multiset` class template for the purple sweep status structure.

Besides member functions to access and modify a scout's data, the following member functions are important:

- `check_individual_for_intersection()`

  checks for an intersection between the scout's boundary curve and its guarded curve. If a new intersection is found it will be inserted as an event into the event queue.

- `check_partners_for_intersection()`

  checks for an intersection between the scout's boundary curve and its partner's boundary curve. If a new intersection is found it will be inserted as an event into the event queue.

The following functions address the issues of scout notification, idling and activation. These always include search operations on the scout set and possibly intersection tests because of the activation of formerly idled scouts. The search operations could be implemented in logarithmic worst-case time by binary searching the scout set but this would involve time-consuming geometric operations, namely comparing curves by their order along the status line. The search is usually much faster if we accept linear worst-case time. For example, we may have to find the highest red upper scout v below a red upper scout u. We enter the scout set at u and decrement our `Scout_set_iterator` until we reach the desired scout v. This procedure is purely combinatorial and v is usually very close.

- `end_lower_inform_lower_above()`

  is called in case a lower scout is about to be deleted. The lowest lower scout above of the same color may have to be activated to take on the duty of guarding the curve the ended lower scout guarded before. This may lead to an intersection test.

- `end_upper_inform_upper_below()`

  is called in case an upper scout is about to be deleted. The highest upper scout below of the same color may have to be activated to take on the duty of guarding the curve the ended upper scout guarded before. This may lead to an intersection test.

- `start_lower_inform_lower_above()`

  is called in case a lower scout has been started. If the lowest lower scout above of the same color is guarding the same curve the new scout wants to guard, the former can be idled.

- `start_lower_inform_lower_below()`

  is called in case a lower scout has been started. If the highest lower scout below of the same color is guarding the same curve the new scout wants to guard, the new scout can be idled. Otherwise the new scout must be activated and perform an intersection test.

- `start_upper_inform_upper_below()`

  is called in case an upper scout has been started. If the highest upper scout below of the same color is guarding the same curve the new scout wants to guard, the former can be idled.

- `start_upper_inform_upper_above()`

  is called in case an upper scout has been started. If the lowest upper scout above of the same color is guarding the same curve the new scout wants to guard, the new scout can be idled. Otherwise the new scout must be activated and perform an intersection test.

The type `My_Sweep_line_subcurve` is extended to store four (possibly null) pointers to scouts:

- The lower scout manning the subcurve.
- The upper scout manning the subcurve.
- The upper scout guarding the subcurve from below.
- The lower scout guarding the subcurve from above.

4.2.8 The sweep loop

As mentioned before, the sweep line framework is not documented but it is commented and offers a modularized and readable code. The call to the public member function `sweep(...)` on an object of type `My_Sweep_line_2` starts the sweep line procedure. We provide the red and blue x-monotone curves as well as all red and blue points via the function parameters. To begin with, the status lines and the event queue are initialized. The actual sweep loop happens inside the protected function `_sweep()`:

```
void _sweep()
{
  Event_queue_iterator eventIter = m_queue->begin();

  // Looping over the events in the queue.
  while (eventIter != m_queue->end())
  {
    //before the _handle_..._curves functions we may
    //have to take care of some special cases
    prepare_red_blue_intersection();
    prepare_red_blue_vertical();

    //'generic' event handling
    //done by the sweep line framework
    _handle_red_left_curves();
    _handle_blue_left_curves();
    _handle_red_right_curves();
    _handle_blue_right_curves();

    proceed_according_to_vertex_type();

    m_queue->erase(eventIter);
    eventIter = m_queue->begin();
  }
}
```

The functions contributed and explained below are implemented as member functions of the class template `My_Basic_sweep_line_2` and are found in the file `My_Sweep_line_handler.h`.

Ideally, we would let the sweep line framework do its generic event handling via the `handle_..._curves` functions. Then we would simply work on the data structures these functions provide. Unfortunately, some special cases need to be handled before. These special cases are being taken care of by `prepare_red_blue_intersection()` and `prepare_vertical()`.

After its part of the event handling, the sweep line framework leaves us in the situation shown in figure 4.2. We have access to four lists containing the incident red curves to the left and to the right of the event point as well as the blue left and right curves. By calling `proceed_according_to_vertex_type ()` we start our part of the workload. Four types of event points can come up and need to be handled in different ways:

- Red-blue intersections. They are not yet associated with an arrangement vertex. Call `handle_red_blue_intersection()`.

- Endpoints of original segments. They are associated with a red (respectively blue) vertex of type `INTERNAL_ENDPOINT` and a blue (respectively red) vertex of type `EXTERNAL_ENDPOINT`. Call `handle_left_endpoint()` or `handle_right_endpoint()`.

- The `point_x`. It is associated with a red and a blue vertex of type `POINT_X`. Call `handle_point_x()`.

- Other points. They are associated with either a red or a blue vertex of type `DEFAULT`. Call `handle_default()`.

By appropriately updating two curves `current_red_below` and `current_blue_below` we can evade many expensive geometric status line search operations.

4.2.9 `handle_point_x()`

Consider figure 4.3. In in this situation we may have to end a region and we definitely have to start a region. Remember that the curves above and below

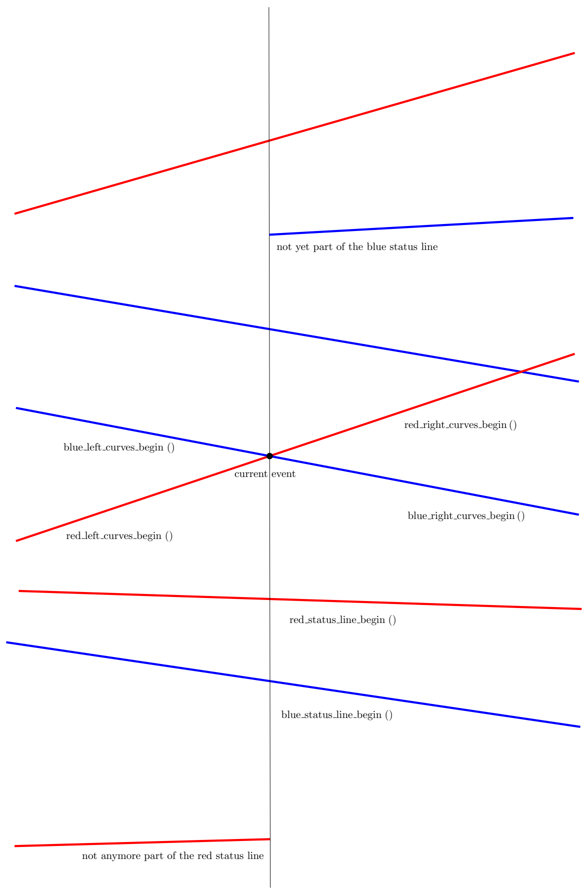

not yet part of the blue status line
red_right_curves_begin ()
blue_left_curves_begin ()
current event
blue_right_curves_begin ()
red_left_curves_begin ()
red_status_line_begin ()
blue_status_line_begin ()
not anymore part of the red status line

Figure 4.2: The setup provided by the sweep line framework after its part of the event handling.

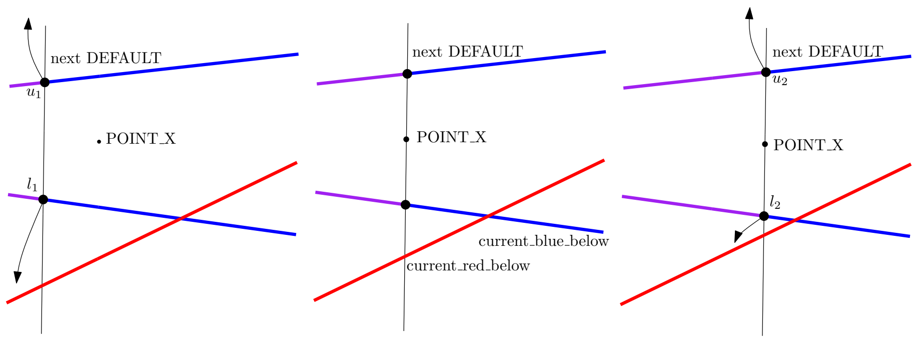


Figure 4.3: Handling of `POINT_X`.

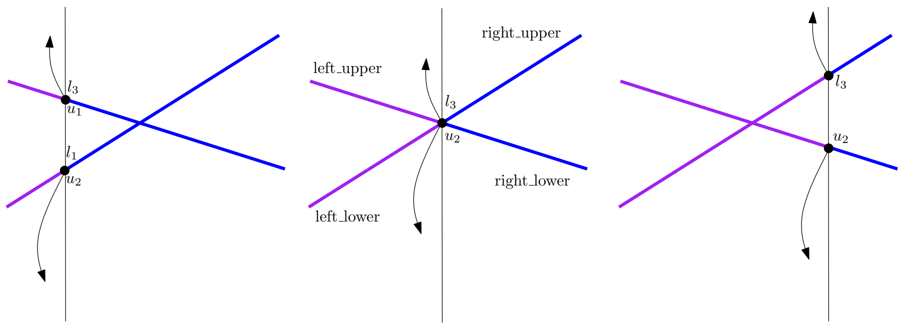


Figure 4.4: Handling of a `DEFAULT` vertex.

`POINT_X` have been split during the vertical decomposition. We check if there is a vertex vertically above and store this information with the sweep line. If there is a `DEFAULT` vertex above, we can delegate the operations of ending and starting regions to it. By that we reduce the number of geometric search operations on the status line. Only in case there is no curve above we start the new region here; then the newly started region will be unbounded and the upper scout will be a dummy scout. Either way, the region enclosed by l_1 and u_1 is being ended. The current curve below, which will be the lower boundary of the new region is being remembered from the previous `DEFAULT` event. The upper boundary is available as the right curve of the next `DEFAULT` event after `POINT_X`. A new region enclosed by two scouts l_2 and u_2 is being started. No costly search operations on the red or blue status line are necessary.

4.2.10 `handle_default()`

Consider figure 4.4. There are up to two left curves (of the same color), a lower and an upper left curve. If there are two left curves, a region (enclosed by l_1 and u_1) between them may have to be ended. In the figure l_1 was idling l_3 and u_1 was idling u_2, so these scouts have to be activated.

The lower left curve may be manned by an upper scout u_2 and may be guarded from below by an upper scout. Both assignments have to be transferred to right curves by calling `change_assignment_of_upper_guard(left_lower)` (which leads to one intersection test) and `change_boundary_of_upper_scout(left_lower)` (which leads to two intersection tests, on for the scout individually and one with its partner). Analogously, `change_assignment_of_lower_guard(left_upper)` and `change_boundary_of_lower_scout(left_upper)` have to be called for the left upper curve. Altogether up to six intersection test will be carried out.

We check whether the start of a region was delegated to the current `DEFAULT` vertex by a former event of type `POINT_X` or `INTERNAL_ENDPOINT`. If so, we start a new region.

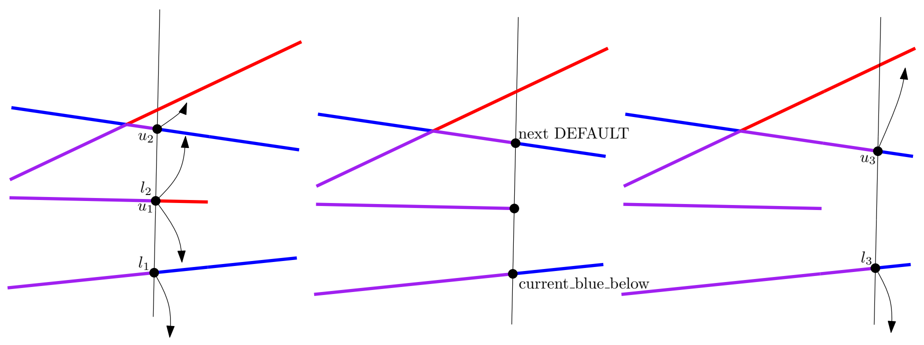


Figure 4.5: Handling of a right endpoint.

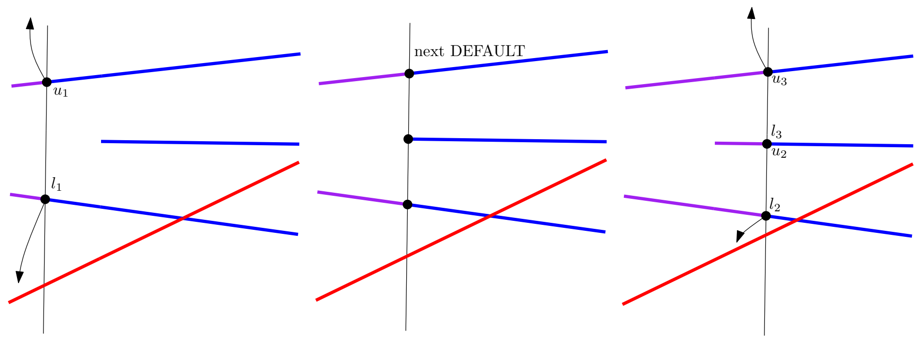


Figure 4.6: Handling of a left endpoint.

4.2.11 `handle_right_endpoint()`

Consider figure 4.5. We have one left curve which may be manned by two scouts u_1 and l_2 and may guarded from below and above. The guards (not shown in the figure) have to be informed that the curve ends here by calling `change_assignment_of_lower_guard(left_curve)` and `change_assignment_of_upper_guard(left_curve)`. In this functions we may have to do two search operations on the status lines to find the two curves that have to be guarded from now on instead. This takes logarithmic time and also leads to two intersection tests due to the new assignments. Two regions may have to be ended; a region below the left curve enclosed by l_1 and u_1 and a region above the left curve enclosed by l_2 and u_2. After this we can handle the right endpoint exactly like we would handle `point_x`, so we call `handle_point_x`.

4.2.12 `handle_left_endpoint()`

Consider figure 4.6. We have one right curve which is part of a segment starting here. The curve may have to be guarded from above and / or below, so we have to inform possible guards by calling

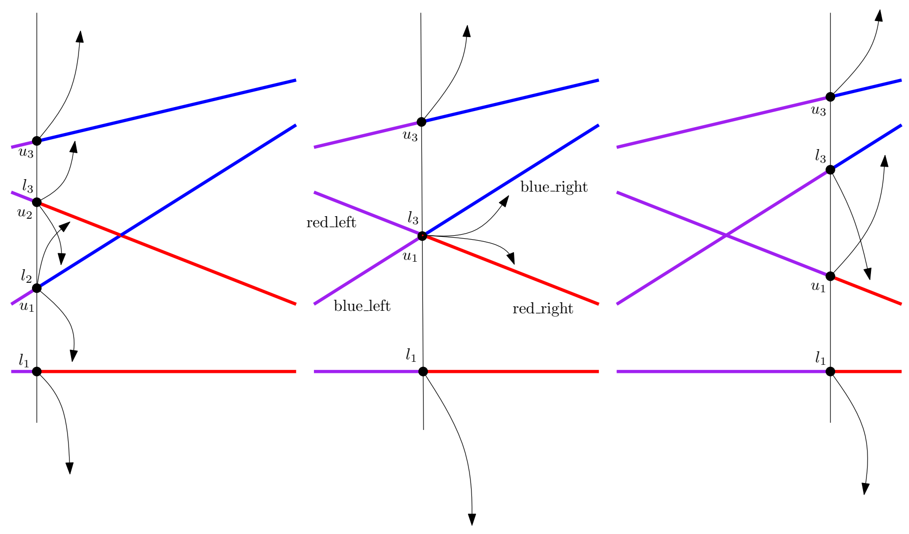


Figure 4.7: Handling of a red blue intersection.

`inform_guard_above` and `inform_guard_below`. For example, if our right curve is blue (like shown in the figure) and we find a red lower scout above guarding a red curve below our right curve, this scout must guard our right curve from now on. We need to do two search operations on the status lines to find the guards to inform which takes logarithmic time and also leads to two intersection tests.

Two regions must be started. The region below the right curve can be started right now by calling `start_region` and will be enclosed by l_2 and u_2. The ending of the region between l_1 and u_1 and the start of the region above the right curve between l_3 and u_3 can be delegated to the `DEFAULT` vertex above unless there is no vertex above, in which case the upper scout will be a dummy and we start the region at the current endpoint event.

4.2.13 `handle_red_blue_intersection()`

Consider figure 4.7. There is exactly one red left curve, one blue left curve, one red right curve and one blue right curve. We must end the region between the left curves enclosed by l_2 and u_2. Up to two remaining scouts (u_1 and l_3 in the figure) change their color by swapping the role of the boundary curve and the guarded curve. First these two are being removed from the scout set in amortized constant time and we call `scout_swap()` for both. We handle each one as if a scout has ended and a new scout is spawned, which can lead to three intersection tests per scout, i.e. six in all. After changing the boundary, the scouts must be reinserted into the scout set in logarithmic time.

Part of the procedure described above is done in

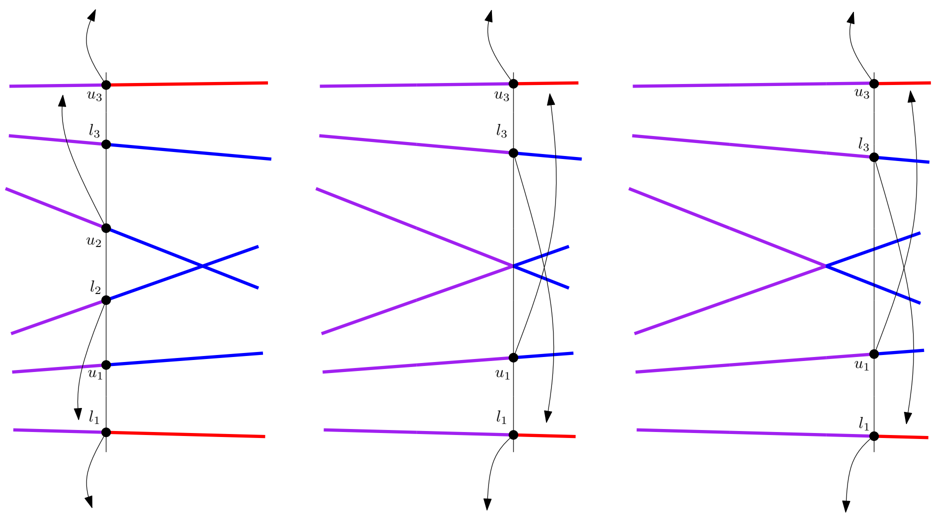


Figure 4.8: Ending a region.

`prepare_red_blue_intersection()` for technical reasons before the sweep line framework does its generic handling of the event.

4.2.14 `end_region()`

Consider figure 4.8. The region between l_2 and u_2 ends because the partners meet at a `DEFAULT` vertex. We make sure that the boundaries these two are manning are being inserted into the purple arrangement. We call the member function `end_upper_inform_upper_below()` on the upper scout and `end_lower_inform_lower_above()` on the lower scout. This leads to the activation of u_1 which was idled by u_2 and the activation of l_3 which was idled by l_2. Two intersection tests may be performed for the newly activated scouts. The erasure of the two scouts from the scout set takes amortized constant time.

4.2.15 `start_region()`

Consider figure 4.9. A region is started at the `DEFAULT` vertex vertically above `POINT_X`. We start two new partnered scouts, a lower l and an upper u one by calling `start_lower_scout()` and `start_upper_scout`. This takes logarithmic time as we will see. We need to do four search operations on the scout set to inform affected scouts about the newly started ones. Again, we search the scout set combinatorially without geometric computations but in linear worst-case time. In the case shown in the figure this will lead to u_{below} and l_{above} being idled. Additionally, up to three intersection tests may be necessary, one for each new scout individually and one between the partners.

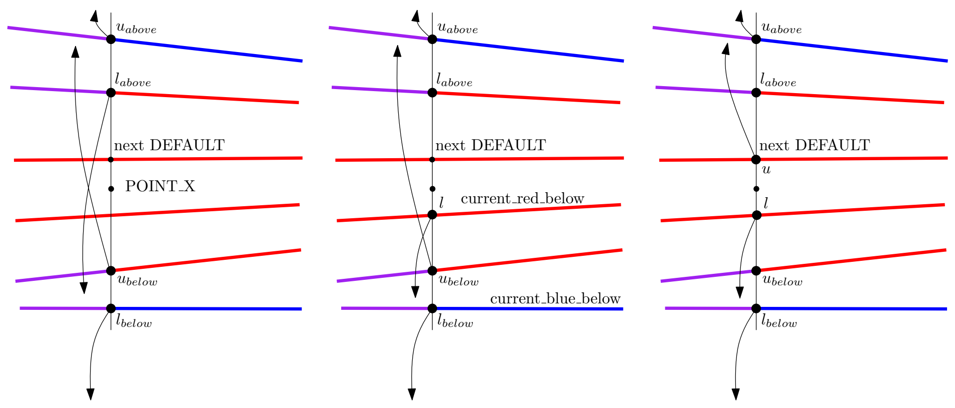


Figure 4.9: Starting a region.

4.2.16 `start_lower_scout()`

Consider figure 4.9 again. Inserting the new scout into the scout set takes logarithmic time. Since we combinatorially maintained the curves `current_red_below` and `current_blue_below`, we don't need to do any geometric search operations on the status lines at this point. In the figure, l follows `current_red_below` and guards `current_blue_below` by testing for an intersection.

4.2.17 `start_upper_scout()`

Consider figure 4.9 again. Inserting the new scout into the scout set takes amortized constant time since we can insert it right above the already inserted lower partner scout. The upper boundary is available as the right curve at the `DEFAULT` vertex, so we just have to do one geometric binary search operation on the appropriate status line to find the curve to guard.

4.3 Handling degeneracies

The implementation can handle grids like those created by the demo program. Therefore we have to adapt the `My_Curve_comparer` functor to be able to deal with vertical segments correctly. During the event handling we have to be careful not to update `current_red_below` or `current_blue_below` with vertical segments. In case of a lexicographically left endpoint of a vertical segment we start only one region instead of two. The most difficult aspect to deal with are red blue intersections involving a vertical and a horizontal segment. They are found as weak intersections by the sweep line framework because of the preceding vertical decomposition. Some minor adaptions to the generic framework and the function `prepare_red_blue_vertical` were designed to deal with this types of complications.

Chapter 5

Practical experience

5.1 Debugging

To debug an implementation of a geometric algorithm it is essential to produce graphical output which is usually much easier to interpret than numerical output. With the `Arrangement_on_surface_2` package comes an extensive demo program with a graphical user interface written with Qt 3.3.4. This could be adapted to produce graphical output for the presented implementation.

In the early stages the possibility to visualize the output was explored. Circles on the segments represented the scouts while arrows pointed to the segment guarded by this particular scout etc. Via a button the user could click through the algorithm sweep by sweep, event by event and even function by function. While this was helpful to get started it proved hard to maintain. It was error-prone when the code changed and made the code much less readable.

By far the most helpful debugging technique was the use of the macro `CGAL_assertion`. An assertion checks if a statement is true and aborts the execution while outputting the responsible code line. So at the beginning and after a function one could check for assumed preconditions and if the function really did what it was supposed to do.

It was very helpful to check our sweep invariants after each event. For that purpose the debug function `_validate_invariants()` was written that is called right after the handling of one event is over. It does the following checks on the scout:

- The size of the scout set is even.
- Dummy scouts are always active.
- Idled scouts do not guard a curve.
- Lower and upper scouts alternate in the scout set.

- If a non-dummy scout is active, it guards the nearest curve of the opposite color above or below.
- If a non-dummy scout is idled, it is protected by an active scout of the same type and the same color.

Additionally we can check the curves in the red and blue status lines:

- Guarded curves are guarded by an active scout of the opposite color.

If the program aborts during `_validate_invariants()` it is pretty certain that the error is in the handling of the previous event. The function is an addition to `My_Sweep_line_2_debug.h` which already provided functions to produce formatted text output during the sweep line procedure.

5.2 Testing

The implementation is compared to the best solution to compute a single face in an arrangement of line segments available to CGAL users at the moment, described in subsection 3.1.3. The running times were obtained on a 2 GHz Athlon 3200+ machine with 2 GB of RAM under ubuntu Linux 8.04.3. The code was compiled using the GNU C++ compiler version 4.2.4 in release mode with the optimization flag `-O3`.

Two non-degenerate data test sets were produced via the demo program (see section 5.3); on the one hand segments from random input points in a square and on the other hand a grid with guaranteed quadratic (and therefore worst-case) complexity. The data sets come in files

```
/random/random200.txt
...
/random/random3400.txt
```

and

```
/grid/grid200.txt
...
/grid/grid2400.txt
```

with the source code.

In the tested situations the unbounded face is the most complex and therefore the most expensive to compute. The bounded face was 'randomly' handpicked at around the middle of the respective arrangement. Its complexity is very low and hence accommodates our implementation. A bounded face in the grid consists of only four segment portions. See figures 5.1, 5.2 and 5.3.

CGAL's arrangement package has undergone involved overhauls to optimize the efficiency by minimizing the calls to traits class functors, i.e. geometric predicates and constructions [33]. The aggregated full construction

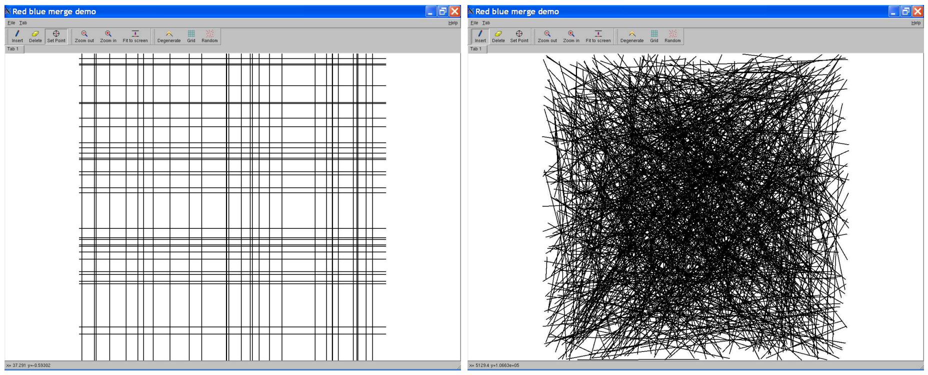

Red blue merge demo
File Edit Help
Insert Delete Set Point Zoom out Zoom in Fit to screen Degenerate Grid Random
Tab 1

Figure 5.1: A grid-like set of line segments and random line segments in a square.

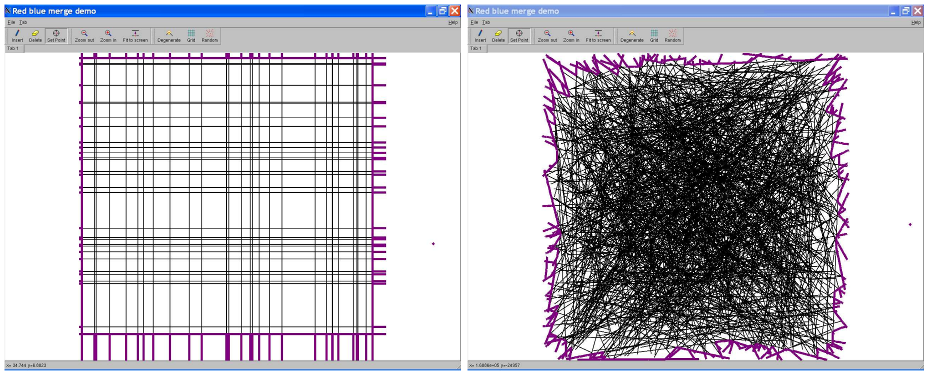

Red blue merge demo
File Edit Help
Insert Delete Set Point Zoom out Zoom in Fit to screen Degenerate Grid Random
Tab 1

Figure 5.2: The unbounded face is the most complex one.

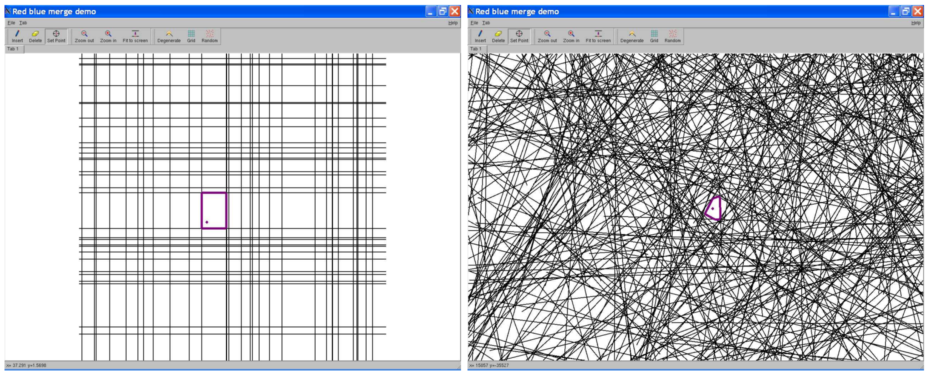

Red blue merge demo
File Edit Help
Insert Delete Set Point Zoom out Zoom in Fit to screen Degenerate Grid Random
Tab 1

Figure 5.3: A random bounded face of low complexity.

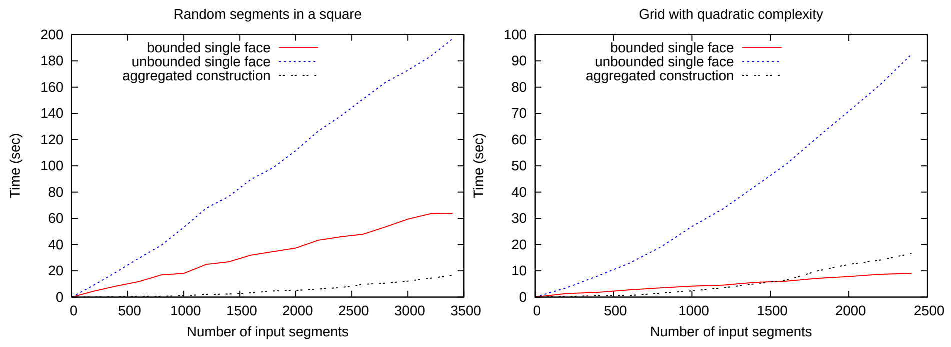


Figure 5.4: Runtime test results

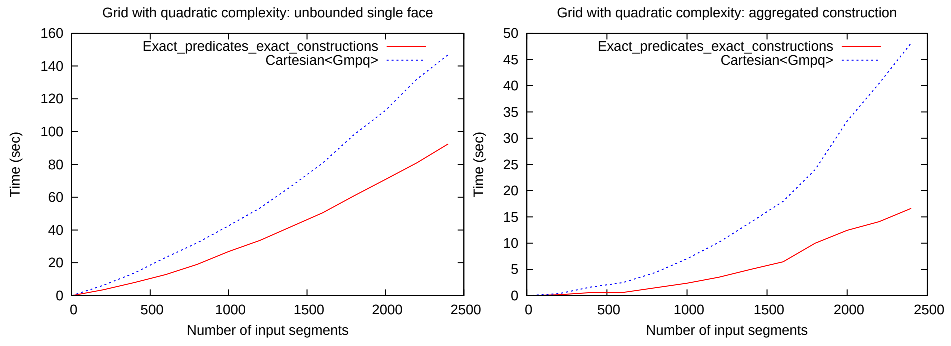


Figure 5.5: Comparing kernels

is impressively fast and is far superior to the presented implementation of the single face algorithm at conceivable input sizes, see figure 5.4. The tests are conducted using the `Exact_predicates_exact_constructions_kernel`.

There is only one situation in which we beat the aggregated construction of the full arrangement at manageable input sizes: when we compute a single bounded face in a grid. We of course expect to overtake the quadratic function of the full arrangement construction time at some point.

The unfiltered `Cartesian<Gmpq>` kernel is compared to the filtered `Exact_predicates_exact_constructions_kernel` (which also employs the number type `Gmpq`) in two situations; see figure 5.5. The computation of the unbounded single face in a grid is sped up by a factor of about 1.5 while the aggregated construction achieves a factor of about 4. This is due to 'major efforts' reported in [33] to reduce redundant calls to predicates that diminish the impact of the filtering techniques, i.e. when the result of a predicate is `EQUAL`.

Hope for our implementation comes from looking at the memory consumption, see figure 5.2. The memory was measured using the CGAL class `Memory_sizer` [26].

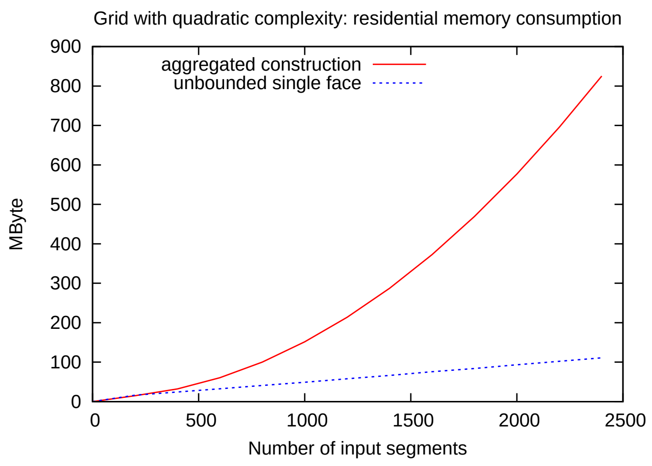


Figure 5.6: The residential memory consumption in the case of a grid.

The curves shows the expected almost linear $O\big(\alpha(n)n\big)$ space requirements for computing the unbounded face and the $O\big(n^2\big)$ space requirements for the full construction. This supports the statement that our implementation could already be useful for very large input sizes.

5.3 The Demo Program

CGAL comes with a package `Qt_widget` which provides an interface between CGAL and the GUI toolkit Qt. See [31] for details on `Qt_widget` and [30] for details on Qt 3.3.

The demo program is a stripped down and adapted version of the demo included in CGAL for the `Arrangement_on_surface_2` package and consists of the following files. All files are published under the Q Public license 1.0 [29] and carry the following copyright notice:

```
// Copyright (c) 2005  Tel-Aviv University (Israel).
// All rights reserved.
```

In brackets after each file name we refer the authors mentioned in the source code of the original files.

- `cgal_types.h` (Baruch Zukerman)

  Basic declarations and instantiations of the used number type, the kernel, the traits class, the arrangement class and more.

- `My_Window.h` (Baruch Zukerman)

  Declarations for our main window.

- `My_Window.cpp` (Baruch Zukerman)

`My_Window` constructor and `main()`.

- `My_Window_files.cpp` (Baruch Zukerman)

  Member function definitions for handling files.

- `My_Window_operations.cpp` (Baruch Zukerman)

  Rest of the member function definitions including added functions to produce random segments.

- `base_tab.h` (Baruch Zukerman, Efi Fogel)

  Declaration of a basic tab handling everything non-geometric.

- `base_tab.cpp` (Baruch Zukerman, Efi Fogel)

  Member function definitions.

- `demo_tab.h` (Baruch Zukerman, Efi Fogel)

  Declaration of an extended tab class template inheriting a base tab. Must be instantiated with a traits class and declares functions that are not independent of the arrangement traits.

- `demo_tab.cpp` (Baruch Zukerman, Efi Fogel)

  Member function definitions.

- `qt_layer.h` (Baruch Zukerman)

  Declaration of the attached layer.

- `qt_layer.cpp` (Baruch Zukerman)

  Member function definitions.

5.3.1 Installation

Download and install CGAL 3.4 from [8]. My implementation is not tested with newer versions yet. You need to patch a bug in the file

```
/CGAL-3.4/include/CGAL/Arr_vertical_decomposition_2.h
```

before installing. Remove the `++i` from line 92 and put it at the end of the `if`-clause inside the `for`-loop (line 99).

Follow the installation manual which is included within the tarball and can also be found online [9]. The installation employs `cmake` and may or may not work flawlessly. The installation manual will also give hints about how to obtain the mandatory third-party libraries boost, GMP and Qt3. People on the CGAL mailing list are very helpful [7].

After compiling and installing the CGAL library, copy the files provided with this work in a folder

```
/CGAL-3.4/demo/Red_blue_merge_demo/
```

and type

```
cmake .
make
```

within the folder. The name of the executable is `redblue`.

I'd be happy to help in case of any installation troubles. Write me under `jannis.warnat@gmx.de`.

5.3.2 Usage

In addition to the obvious insertion and deletion functionality there are three buttons:

- 'Degenerate': Check the current set of line segments for some degeneracies that will cause the single face computation to crash with a segmentation fault. This is done by an aggregated construction of the full arrangement. To be non-degenerate, every vertex representing an original segment endpoint must have degree one and every vertex representing an intersection of two original segments must have degree four. The elapsed time is written to `stdout`. Degenerate vertices are highlighted in green.

- 'Grid': Produces a random grid after asking for the number of vertical segments. The full arrangement of a grid-like set of line segments has quadratic complexity.

- 'Random': Produces a set of random line segments in a square. The endpoints of these line segments are randomly distributed inside a square which size depends on the chosen number of segments.

The latter buttons employ random geometric object generators offered by CGAL, see [23]. The elapsed time for a single face computation is written to `stdout`, broken down into the time spent on the vertical decomposition, the sweeps, the rotations and the fusion of the outputs of the two sweeps. The time not covered by this four items is negligible.

See the 'How to' in the help menu for a short manual.

Chapter 6

Conclusion and future work

To my knowledge, this work presents the first implementation of the deterministic algorithm for computing a single face in an arrangement of line segments. With the help of CGAL and its realization of the EGC paradigm, it is a stable implementation for non-degenerate inputs.

The actual runtimes leave a lot to be desired. Although much effort was put into this issue already, there are lots of possibilities to reduce the number of calls to traits class functions which dominate the runtime. Input from experienced implementers of geometric algorithms could help.

Further obvious future work would be to make the implementation handle all degenerate inputs correctly. This would be an interesting, yet tedious task although CGAL's sweep line framework already supports degenerate cases and should be of great help.

The arrangement package already provides traits classes for lines, polylines, circular arcs, conic arcs, arcs of rational functions and Bézier curves. In theory, the algorithm naturally extends to work on more general types of Jordan arcs. For a more generic implementation it would be even more important to further reduce the number of calls to geometric traits methods, since they will dominate the runtime stronger than in the test runs for line segments.

Since CGAL version 3.4 the arrangement package carries the name `Arrangement_on_surface_2` reflecting the future (not yet fully implemented) capability to represent arrangements of curves embedded on two dimensional surfaces in space.

A link to this thesis and the source code will be posted on the CGAL mailing list in the hope for some feedback. Maybe this work can be the basis for an interesting contribution to CGAL's arrangement package sometime in the future. It would be nice to extend the sweep line framework to allow for an elegant inclusion of unusual sweep line algorithms like the one in this work, including a right to left sweep without the need to rotate the arrangement. Possibly this could be achieved by inverting the employed functors. The framework could also be extended to support multiple sweep status structures.

Another way forward could be to drop the idea of multiple independent sweep status structures and embrace the visitor approach on the sweep line framework. Some search operations on the status lines would have to be done combinatorially leading to linear worst-case time for these operations. For most inputs this could prove to be superior to the logarithmic time search involving expensive geometric operations. This idea was already followed in the implementation of search operations on the scout set.

Since this was my first programming project of this scale, the implementation probably does not follow all conventions and best practices a more experienced C++ programmer would naturally comply with, although the high-quality code of CGAL gave an excellent example to follow.

Bibliography

[1] P. K. Agarwal and M. Sharir. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge Univerity Press, 1995.

[2] boost C++ libraries. http://www.boost.org.

[3] Generic programming techniques used in the boost libraries. http://www.boost.org/community/generic_programming.html.

[4] H. Brönnimann, A. Fabri, G.-J. Giezeman, S. Hert, M. Hoffmann, L. Kettner, S. Schirra, and S. Pion. 2D and 3D Geometry Kernel. In *CGAL User and Reference Manual.* 3.4 edition, 2008.

[5] Cgal, Computational Geometry Algorithms Library. http://www.cgal.org.

[6] CGAL ipelets. http://cgal-ipelets.gforge.inria.fr.

[7] CGAL mailing lists. http://www.cgal.org/mailing_list.html.

[8] INRIAGForge CGAL download page. http://gforge.inria.fr/frs/?group_id=52.

[9] CGAL 3.4 installation manual. http://www.cgal.org/Manual/3.4/doc_html/installation_manual/Chapter_installation_manual.html.

[10] CGAL Editorial Board. *CGAL Developers' Manual*, 3.3.1 edition, 2007.

[11] CGAL Editorial Board. *CGAL User and Reference Manual*, 3.4 edition, 2008.

[12] O. Cheong. The ipe extensible drawing editor. http://tclab.kaist.ac.kr/ipe.

[13] Concepts in C++0x. http://en.wikipedia.org/wiki/Concepts_(C++).

[14] O. Devillers, L. Kettner, M. Seel, and M. Yvinec. Handles and Circulators. In *CGAL User and Reference Manual.* 3.4 edition, 2008.

[15] Eclipse IDE. http://www.eclipse.org.

[16] E. Fogel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Algorithms - ESA 2004: 12th Annual European Symposium*, pages 664–676, Bergen, Norway, 2004. Springer-Verlag.

[17] S. Fortune. Stable maintenance of point set triangulations in two dimensions. In *SFCS '89: Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 494–499, Washington, DC, USA, 1989. IEEE Computer Society.

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

[19] GeometryFactory. http://www.geometryfactory.com.

[20] The GNU Multi-Precision Bignum Library. http://gmplib.org.

[21] L. J. Guibas, M. Sharir, and S. Sifrony. On the general motion-planning problem with two degrees of freedom. *Discrete Comput. Geom.*, 4(5):491–521, 1989.

[22] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. *Computational Geometry: Theory and Applications*, 38(1-2):16–36, September 2007.

[23] S. Hert, M. Hoffmann, L. Kettner, and S. Schönherr. Geometric Object Generators. In *CGAL User and Reference Manual*. 3.4 edition, 2008.

[24] M. Hoffmann, L. Kettner, S. Pion, and R. Wein. STL Extensions for CGAL. In *CGAL User and Reference Manual*. 3.4 edition, 2008.

[25] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computation. *Computational Geometry*, 40(1):61–78, 2007. Also: Proc. 12th ESA 2004, LNCS No.3221, pp.702-713.

[26] L. Kettner, S. Pion, , and M. Seel. Profiling Tools Timers, Hash Map, Union-find, Modifiers. In *CGAL User and Reference Manual*. 3.4 edition, 2008.

[27] R. Klein. *Algorithmische Geometrie*. Springer, 2005.

[28] R. Klein, E. Langetepe, and T. Kamphans. Bewegungsplanung für Roboter, 2008. Manuscript.

[29] The Q Public License version 1.0. http://doc.trolltech.com/3.0/license.html.

[30] Trolltech Qt 3.3 reference. http://doc.trolltech.com/3.3/index.html.

[31] L. Rineau and R. Ursu. Qt Widget. In *CGAL User and Reference Manual*. 3.3 edition, 2007.

[32] Silicon Graphics STL reference. http://www.sgi.com/tech/stl.

[33] R. Wein, E. Fogel, B. Zukerman, and D. Halperin. Advanced programming techniques applied to CGAL's arrangement package. *Comput. Geom. Theory Appl.*, 38(1-2):37–63, 2007.

[34] R. Wein, E. Fogel, B. Zukerman, and D. Halperin. 2D Arrangements. In *CGAL User and Reference Manual*. 3.4 edition, 2008.

[35] C. K. Yap. Towards exact geometric computation. *Comp. Geoem. Theory and Appl.*, 7:3–23, 1997. Invited talk at 5th Canadian Conf. on Computational Geometry (CCCG'93).

[36] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapmen & Hall/CRC, Boca Raton, FL, 2nd edition, 2004. Revised and expanded from 1997 version.