Computational Geometry on the Grid: Traversal and Plane-Sweep Algorithms for Spatial Applications

Markus Schneider, Ralf H. Güting   Thomas de Ridder

Praktische Informatik IV   Praktische Informatik III
FernUniversität Hagen
D-58084 Hagen, Germany
[Markus.Schneider, gueting, Thomas.deRidder]@fernuni-hagen.de

Abstract. To guarantee numerical robustness, topological correctness, and simultaneously efficiency of geometrical algorithms can be currently regarded as one of the most urgent challenges of computational geometry. These properties represent fundamental requirements for spatial predicates and operations in non-standard application areas like spatial data base systems, GIS, VLSI design, and CAD. Due to the discrepancy between the infinite-precision assumption of computational geometry and the finite-precision reality of computer systems, these properties cannot be warranted by geometric algorithms currently available. The employment of these algorithms in practice frequently amounts to unacceptable numerical rounding errors and topological inconsistencies and degeneracies. This paper presents geometric algorithms that satisfy the properties required above and that rest on a discrete geometric basis in form of a uniform integer grid. The paper also emphasizes that, from an application point of view, the design goals pursued by algorithms of computational geometry differ to some degree from those of the application areas mentioned above.

Keywords. Applied/finite-resolution/finite-precision computational geometry, discrete geometric basis, grid, numerical robustness, topological correctness, parallel traversal, plane-sweep

1 Introduction

Numerical robustness, topological correctness, reliability, accuracy, and simultaneously efficiency of geometrical algorithms are fundamental requirements for spatial predicates and operations in spatial databases, geographical information systems, VLSI design, CAD, and other related, non-standard application areas. Although *theoretical* computational geometry has provided a large number of useful and efficient geometric algorithms, we encounter the problem that it is based on Euclidean geometry and on continuous, infinite-precision arithmetic (real numbers) and that it ignores the reality of a discrete, finite-precision arithmetic (floating point numbers) available in computers. Thus, theoretically correct geometric algorithms are not necessarily practically valid!

Frequently, precision, robustness, and correctness of geometric primitives (like segment intersection, point-in-polygon test) within geometric algorithms are simply taken for granted. But geometric algorithms are quite sensitive regarding this procedure, and in the end, the task is mostly left to the programmer to close the gap between theory and practice. This leads inevitably not only to numerical rounding errors but also to topological inconsistencies and degeneracies [DS90, For85, GY86, Hof89], since topological information is commonly inferred from coordinate-based geometric data. Hence, an *applied* computational geometry is needed which takes into account the finite representations available in computer systems.

In this paper we design numerically robust, topologically consistent, and at the same time efficient geometric algorithms whose argument and result objects are defined on an integer grid. They rest on the traversal and plane-sweep paradigm. We will also see that, from an application point of view, the design goals pursued by algorithms of computational geometry differ to some degree from those of the application areas mentioned above.

The knowledge about the gap between theory and practice of computational geometry has raised growing interest in solving issues of numerical robustness and topological correctness of geometric algorithms. Two approaches can be mainly distinguished: perturbation-free and perturbation approaches. Perturbation relates to slightly changing input data or computed values in a suitable way when they are assigned to variables. Perturbation-free approaches (e.g. [KM83, MK84, OTU87, SI88]) aim at performing exact geometric computations with such sufficiently high precision that correct and robust numerical results must be obtained. Provided that the input data are exact, the task is to determine how many digits of precision are required by numerical computations so that the algorithm produces correct results and takes into account desired accuracy. Perturbation approaches (e.g. [DS90, EM88, GM95, GSS89, GY86, HHK88, Mil89, NME90, Sch94]) allow to slightly change input data or computed results. Because in many applications the input data are approximate from the beginning, such slight alterations seem to be tolerable.

This paper is based on an interesting subclass of perturbation approaches that attempt to transform geometric data from the continuous domain to the discrete domain in the form of a uniform (integer) grid and to perform computational geometry in this discrete domain. That is, points, end points of line segments, vertices of polygons, etc. have integer instead of floating point coordinates and lie on grid points. This has led to a subfield called *finite-resolution* [GY86] or *finite-precision* [Yao92] *computational geometry* dealing with geometry performed on a discrete domain. On a grid, problems can frequently be solved more efficiently and simpler than in Euclidean space [Ove88a]. On the other hand, problems like how to handle the intersection point of two integer-based line segments are more complicated [GM95, GY86].

Finite-precision computational geometry has so far only been studied by a few researchers (overviews can be found in [KK81, Ove88a, Ove88b]). More efficient solutions in comparison with their Euclidean counterparts have been found for the nearest neighbor searching problem [KM85], range searching on a grid [Ove88b, Ove88c], the point location problem [Mül85], the computation of rectangle intersections and maximal elements by divide-and-conquer [KO88b], computing the convex hull of a set of points, reporting all intersections of a set of arbitrarily oriented line segments, and the calculation of rectangle intersections and maximal elements by using the plane-sweep technique [KO88a, Ove88b].

A main problem is the treatment of the intersection point of two integer line segments which usually does not fall on a grid point with integer coordinates but has rational coordinates and thus violates the closure property of the grid domain. Greene and Yao's *redrawing* concept [GY86] and Guibas and Marimont's *snap rounding* method [GM85] have turned out to be acceptable and robust solutions for this problem. Both approaches allow slight perturbations of the original line segments by rounding the end points and intersection points of line segments to representable grid points, but in a way so that topological consistency is globally guaranteed. That is, both methods cannot and do not prevent imprecision from the original data, but they provide consciously controlled perturbation of the original data and the maintenance of their global topology.

Section 2 summarizes the assumptions and the design goals that underlie this paper and that have both especially to be seen in a database context. In this context geometric data structures have to be viewed as representations of spatial objects furnished with a complex semantics and stored in databases. Geometric algorithms are then realizations of operations on these spatial objects. Section 3 introduces data structures for appropriately defined point, line, and area features based on an integer grid. Section 4 and Section 5 introduce some grid-based geometric algorithms which are based on the parallel traversal and the plane-sweep paradigm, respectively. Section 6 draws some conclusions.

2 Assumptions and Design Goals

This section summarizes assumptions and design goals that are important for an understanding of the objectives of this paper. The assumptions reflect the concepts and results of the author's work in the past and motivate the design goals of the geometric algorithms presented here from an application-oriented point of view.

2.1 Reorganizing the Underlying Space: Discrete Geometric Bases

As a basis for geometric modeling and computing, very often Euclidean space is used or implicitly assumed which represents points in the plane by pairs of infinitely precise, real numbers. Unfortunately, number systems in computers are finite and offer only limited approximations like floating numbers. Ignoring this fact almost certainly leads to unacceptable errors in spatial query processing since spatial objects, predicates, and operations are usually realized by geometric data structures, predicates, and algorithms, respectively, which are just based on these finite representations and thus have shortcomings with respect to numerical robustness and topological consistency.

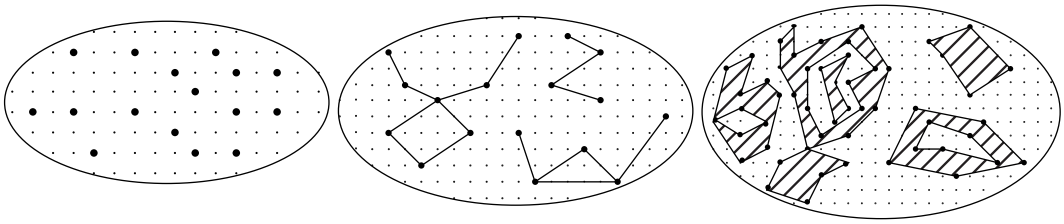
A proposal to overcome these problems is to exchange the underlying domain upon which geometric objects are defined and to introduce a *discrete geometric basis* for these objects both for modeling and for implementation. Let $P_N = N \times N$ with $N = \{0, ..., n-1\} \subseteq \mathbb{N}$ be a finite discrete space (a uniform integer grid). Points and end points of line segments are presupposed to have coordinates in N. We assume that the intersection point of any two line segments, which usually does not lie on the grid, is perturbed to the nearest grid (i.e., representable) point by applying Greene & Yao's *redrawing* concept [GY86] or Guibas & Marimont's *snap rounding* [GM95] idea, for instance. These concepts ensure numerical robustness and topological consistency of a collection of line segments.

Transforming an application's set of points and possibly intersecting line segments in this way, we obtain a discrete geometric basis called *realm* [GS93, Sch97]. It represents the full underlying geometry of a particular application space and consists of a finite set of points and *non-intersecting* line segments. Hence, a realm can be regarded as another term for *domain* and is somewhat similar to an enumeration type in programming languages. Benefits of the realm concept are, in particular, that it shields spatial algebras and geometric algorithms built on top of realms from problems of numerical robustness and topological correctness.

Graph-theoretically, a realm is a *spatially embedded planar graph* over a finite resolution grid. Therefore, we can more precisely characterize a realm (over N) as a set of points and line segments with the following properties: (1) Each point and each end point of a line segment of a realm is a grid point. (2) Each end point of a realm segment is also a point of the realm. (3) No realm point lies within a realm segment (which means on it without being an end point). (4) No two realm segments have common points except at their end points. This definition is based on a collection of *robust geometric primitives* offering basic predicates on grid-based points and segments like whether two line segments intersect within their interiors, whether a point lies on a line segment, or whether two line segments meet in one of their end points. A formal definition of realms and robust geometric primitives is given in [GS93, Sch97].

2.2 Spatial Data Types

Spatial data types [Sch97] like *points*, *lines*, and *regions* are special abstract data types that are employed to model geometry and to suitably represent geometric data in database systems. They provide a fundamental abstraction for modeling the geometric structure of objects in



a *points* object a *lines* object a *regions* object

Figure 1

space as well as their relationships, properties, and operations. We speak of *spatial objects* as values of spatial data types. In the context here, we focus on two-dimensional spatial objects that are defined on top of realms and in terms of points and line segments present in the realm. That is, their construction and update are indirectly performed by selecting some realm objects and by propagating changes to the dependent spatial objects, respectively. Thus, all spatial objects considered here are *realm-based*. A proposal of such a collection of realm-based spatial data types has been provided by the *ROSE* (*RObust Spatial Extension*) *algebra* [GS95, Sch97].

Within this framework, a *points* object is a finite set of grid points, a *lines* object is a set of pairwise disjoint *blocks*, each block being a maximal set of connected line segments, and a *regions* object is a set of pairwise edge-disjoint *faces* (edge-disjoint means that two faces may have a common vertex but no common segment), each face being a simple polygon possibly containing a set of disjoint simple polygons called *holes* (see the example in Figure 1). Moreover, a comprehensive set of *spatial operations* (the ROSE algebra contains approximately 70 different spatial operations) is specified on these objects. This set, for instance, includes binary spatial predicates expressing topological relationships like *intersects*, *inside*, *adjacent*, and binary spatial operators returning spatial objects like *intersection*, *common_border*, and *contour*. All data types and operations maintain closure properties. They are, in particular, closed under the operations *union*, *intersection*, and *difference* with regard to the same realm. That is, the result of such an operation is a realm-based object as well and corresponds to the definitions of the spatial data types given above. A formal definition of realm-based spatial data types and operations is given in [GS95, Sch97].

2.3 Design Goals

The main objective of this paper is to demonstrate that the implementation of many operations of a spatial algebra can be simplified if this algebra has a discrete geometric basis. Simplification in this context relates on the one hand to the reduction of the algorithmic complexity of known Euclidean-based solutions for the operations and on the other hand to the reduction of run time, either by a whole time complexity class or at least by constant factors. Taking the implementation of the ROSE algebra as an example, we provide efficient data structures for the realm-based spatial data types and numerically robust and efficient *realm-based geometric algorithms* for the operations, all defined over a uniform and discrete integer grid.

From an application point of view, the design goals pursued by algorithms of current computational geometry differ to some degree from those of the application areas mentioned above. These application-oriented design aspects are motivated as follows:

- The representations for spatial data types by means of geometric data structures are designed for use in a database context. This implies, in particular, that these representations are no dynamic pointer data structures in main memory, but are all embedded into compact storage areas which can be efficiently transferred between a main memory buffer and disk.

- All spatial objects processed and produced by the operations are realm-based, i.e., they are defined over a discrete basis and in particular no two segments intersect within their interiors and no point lies within a segment (closure property of spatial data types and operations). As implementations of realm-based spatial operations, geometric algorithms operate on representations of realm-based spatial objects with a complex semantics and *not* on *arbitrary* sets of points or line segments. This means that the geometric problems considered here are red/blue problems. Geometric algorithms for spatial operations that produce realm-based spatial objects yield representations that conform to the corresponding realm-based spatial data type (closure property of geometric data structures and algorithms).
- The fact that all spatial objects processed by operations are realm-based can be exploited for designing efficient geometric algorithms. For example, some operations can now be realized through a simple parallel traversal for which otherwise more complex and expensive plane sweep algorithms would be needed. Operations that are to be realized by plane-sweep algorithms are now much simpler and more efficient, since they need not discover new intersections and treat special cases. Sweep stations can only be isolated points or end points of segments which are all known in advance. Hence, a static sweep event structure for managing the sweep stations is sufficient. We will also see that an initial sorting phase is unnecessary for plane-sweep algorithms.
- Different algorithms processing the same kind of spatial objects usually prefer different internal object representations for achieving highest efficiency. But in contrast to traditional work on geometric algorithms, the focus here is *not* on finding the *most* efficient algorithm for one single problem (operation) together with a corresponding sophisticated data structure, but rather on considering a spatial algebra with a large number of operations as a whole and on reconciling the various requirements posed by different algorithms within a single, universal data structure for each type. Otherwise, we would be forced to design expensive conversion functions for different representations of the same type.

In the sequel, data structures and algorithms are described at a high abstraction level. The algorithms can be grouped into *parallel traversal*, *plane-sweep*, and *graph algorithms* [GRS95, Sch97]. The latter group is not described in this paper. For the first two paradigms, we show a few “prototype” operators and their algorithms and sketch which other operators can be realized similarly. Many algorithms require only linear time, the remaining ones $O(n \log n)$ time where n is a bound on the size of the operand objects.

3 Data Structures

This section introduces the data structures representing objects of the three data types *points*, *lines*, and *regions*. Rather than describing these data structures directly in terms of arrays, records, etc., we introduce a high-level description which offers suitable access and construction operations to be used in the algorithms. Basically, we define an abstract data type for each of the three data structures. It is then left to a later step to design and implement each data structure concretely.

Type *points* is defined as the set of all ordered sequences $< p_1, ..., p_n >$ of n *N-points* (denoting the elements of P_N) together with a logical pointer indicating a position within the sequence.

$$
\begin{aligned}
points = \{ (pos, < p_1, ..., p_n >) \mid \; & \\
& (1)\; pos \geq 0; n \geq 0 \\
& (2)\; \forall\, 1 \leq i \leq n : p_i \in P_N \\
& (3)\; \forall\, 1 \leq i < n : p_i < p_{i+1} \}
\end{aligned}
$$

The "<"-relation in the definition above denotes the usual (x, y)-lexicographic order on P_N which is defined as

$$\forall\, p_i, p_j \in P_N, i \neq j : \quad p_i < p_j \;\Leftrightarrow\; x_i < x_j \vee (x_i = x_j \wedge y_i < y_j).$$

A number of functions is defined on *points* objects. These functions (and later corresponding functions on types *lines* and *regions*) serve as construction, scan, retrieval, and manipulation functions in algorithms for ROSE algebra operations. Their syntax is given by the following signature:

new	:		$\rightarrow$	*points*
select_first	:	*points*	$\rightarrow$	*points*
select_next	:	*points*	$\rightarrow$	*points*
end_of_pt	:	*points*	$\rightarrow$	*bool*
get_pt	:	*points*	$\rightarrow$	P_N
insert	:	*points* $\times P_N$	$\rightarrow$	*points*

Their semantics description is given by a set-theoretic specification. Let $P_p = < p_1, ..., p_n >$, $P = (i, P_p) \in$ *points*, and $p \in P_N$. Moreover, let symbol $\Diamond$ denote the empty sequence.

$$
\begin{aligned}
&new() &&= (0, \Diamond) \\
&select\_first(P) &&= \begin{cases} (1, P_p) & \text{if } n \geq 1 \\ (0, \Diamond) & \text{otherwise} \end{cases} \\
&select\_next(P) &&= \begin{cases} (i+1, P_p) & \text{if } 1 \leq i < n \\ (0, P_p) & \text{otherwise} \end{cases} \\
&end\_of\_pt(P) &&= (i = 0) \\
&get\_pt(P) &&= \begin{cases} p_i & \text{if } 1 \leq i \leq n \\ undefined & \text{otherwise} \end{cases} \\
&insert(P, p) &&= \begin{cases} (j, P_p) & \text{if } \exists\, j \in \{1, ..., n\} : p = p_j \\ (1, < p >) & \text{if } n = 0 \\ (1, < p, p_1, ..., p_n >) & \text{if } p < p_1 \\ (n+1, < p_1, ..., p_n, p >) & \text{if } p > p_n \\ (j+1, < p_1, ..., p_j, p, p_{j+1}, ..., p_n >) & \text{if } \exists\, j \in \{1, ..., n\text{-}1\} : p_j < p < p_{j+1} \end{cases}
\end{aligned}
$$

The representation of the more complex *lines* and *regions* objects is based on *ordered sequences of halfsegments*. Let $S_N = \{(p, q) \mid p \in P_N, q \in P_N\}$ denote the set of *N-segments*. The equality of two N-segments $s_i = (p_i, q_i)$ and $s_j = (p_j, q_j)$ is defined as

$$s_i = s_j \;\Leftrightarrow\; (p_i = p_j \wedge q_i = q_j) \vee (p_i = q_j \wedge p_j = q_i)$$

Without loss of generality we normalize S_N by the assumption that

$$\forall\, s \in S_N : \quad s = (p, q) \;\Rightarrow\; p < q$$

This enables us to speak of a *left* and a *right end point* of an N-segment. Let further $H_N = \{(s, d) \mid s \in S_N, d \in \{left, right\}\}$ be the set of *halfsegments*. A halfsegment $h = (s, d)$ consists of an N-segment s and a flag d emphasizing one of the N-segment's end points which is called the *dominating point* of h. If d = *left* then the left (smaller) end point of s is the dominating point of h, and h is called *left halfsegment*. Otherwise, the right end point of s is the dominating point of h, and h is called *right halfsegment*. Hence, each N-segment s is mapped to two halfsegments (s, *left*) and (s, *right*). Let *dp* be the function which yields the dominating point of a halfsegment.

For two distinct halfsegments h_1 and h_2 with a common end point p, let α be the enclosed angle such that $0° < \alpha \leq 180°$ (an overlapping of h_1 and h_2 is excluded by the realm properties). Let a predicate *rot* be defined as follows: $rot(h_1, h_2)$ is true iff h_1 can be rotated around p through α to overlap h_2 in counterclockwise direction. We can now define a complete order on halfsegments which is basically the (x, y)-lexicographic order by dominating points. For two halfsegments $h_1 = (s_1, d_1)$ and $h_2 = (s_2, d_2)$ we obtain:

$$\begin{aligned} h_1 < h_2 \quad \Leftrightarrow \quad & dp(h_1) < dp(h_2) \vee \\ & (dp(h_1) = dp(h_2) \wedge ((d_1 = right \wedge d_2 = left) \vee (d_1 = d_2 \wedge rot(h_1, h_2)))) \end{aligned}$$

We can now continue with the definition of type *regions* (the definition of type *lines* is almost the same, see below). Type *regions* is defined as the set of ordered sequences $< h_1, ..., h_n >$ of n halfsegments where each halfsegment h_i has an attached *set of attributes* a_i whose elements are values of some new type *attr*. Attribute sets are used in algorithms to attach auxiliary information to N-segments.

$$\begin{aligned} regions = \{(pos, < h_1, ..., h_n >, < a_1, ..., a_n >) \mid \; & (1)\;\; pos \geq 0, n \geq 0 \\ & (2)\;\; \forall\, i \in \{1, ..., n\} : h_i \in H_N, a_i \subseteq attr \\ & (3)\;\; \forall\, i \in \{1, ..., n\text{-}1\} : h_i < h_{i+1}\} \end{aligned}$$

Also on *regions* objects a number of functions is defined. Their syntax is given by the following signature:

$$\begin{array}{lcll} new & : & & \to\; regions \\ select\_first & : & regions & \to\; regions \\ select\_next & : & regions & \to\; regions \\ end\_of\_hs & : & regions & \to\; bool \\ get\_hs & : & regions & \to\; H_N \\ get\_attr & : & regions & \to\; attr \\ update\_attr & : & regions \times attr & \to\; regions \\ insert & : & regions \times H_N & \to\; regions \end{array}$$

For their semantics description let $R_h = < h_1, ..., h_n >$, $R_a = < a_1, ..., a_n >$, $R = (i, R_h, R_a) \in$ *regions*, and $h \in H_N$.

$$\begin{aligned} new() &= (0, \Diamond, \Diamond) \\ select\_first(R) &= \begin{cases} (1, R_h, R_a) & \text{if } n \geq 1 \\ (0, \Diamond, \Diamond) & \text{otherwise} \end{cases} \\ select\_next(R) &= \begin{cases} (i + 1, R_h, R_a) & \text{if } 1 \leq i < n \\ (0, R_h, R_a) & \text{otherwise} \end{cases} \\ end\_of\_hs(R) &= (i = 0) \\ get\_hs(R) &= \begin{cases} h_i & \text{if } 1 \leq i \leq n \\ undefined & \text{otherwise} \end{cases} \end{aligned}$$

$$get\_attr(R) = \begin{cases} a_i & \text{if } 1 \le i \le n \\ undefined & \text{otherwise} \end{cases}$$

$$update\_attr(R, a) = \begin{cases} (i, R_h, < a_1, ..., a_{i-1}, a, a_{i+1}, ..., a_n >) & \text{if } 1 \le i \le n \\ undefined & \text{otherwise} \end{cases}$$

$$insert(R, h) = \begin{cases} (j, R_h, R_a) & \text{if } \exists\, j \in \{1, ..., n\} : h = h_j \\ (1, < h >, < \emptyset >) & \text{if } R_h = \lozenge \\ (1, < h, h_1, ..., h_n >, < \emptyset, a_1, ..., a_n >) & \text{if } h < h_1 \\ (n+1, \; < h_1, ..., h_n, h >, & \\ \qquad < a_1, ..., a_n, \emptyset >) & \text{if } h > h_n \\ (j+1, \; < h_1, ..., h_j, h, h_{j+1}, ..., h_n >, & \\ \qquad < a_1, ..., a_j, \emptyset, a_{j+1}, ..., a_n >) & \text{if } \exists\, j \in \{1, ..., n\text{-}1\} : h_j < h < h_{j+1} \end{cases}$$

Note that these functions manipulate halfsegment sequences. This does not guarantee that such a sequence really represents a well-defined and correct *regions* object. The algorithms using this structure are responsible for constructing only sequences that indeed represent *regions* objects. Type *lines* (not presented here in detail) is identical to type *regions* except for all the parts related to attributes which are not needed.

Concrete implementations for each of the three data types could represent a sequence of n points or halfsegments in a linked list or sequentially in an array. The latter representation would also be compatible with the “compact storage area” requirement needed for efficient database loading/storing. In this case, all functions except for *insert* need $O(1)$ time; *insert* requires $O(n)$ time for arbitrary positions and $O(1)$ time for appending an element at the end of the sequence. Such a representation would in fact be quite good for all “parallel traversal” algorithms, because result objects are always constructed in the lexicographical point or halfsegment order and can therefore be built in linear time.[1]

4 Traversal Algorithms

A number of operators defined on the types *points*, *lines*, and *regions* can be realized by a *simple* or *parallel traversal* (i.e., a scan) [GRS95, Sch97] through the point or halfsegment sequences of one or two objects. In this section we will explain some example algorithms which are analysed with respect to their worst case time and space requirements. To simplify the description of algorithms, for each possible combination of two spatial data types our high-level notation is extended by two further functions which enable a parallel traversal through two ordered sequences of elements (halfsegments, points).

As an example, we consider the two functions for two *regions* objects given by their halfsegment sequences. Function *rr_select_first*(R_1, R_2, *object*, *status*) selects the first halfsegment of each of the two *regions* objects R_1 and R_2 (compare to the function *select_first* defined on type *regions*) and positions a logical pointer on both of them. The parameter *object* with possible values {*none*, *first*, *second*, *both*} indicates which of the two object representations contains the smaller halfsegment. If the value of *object* is *none*, no halfsegment is selected, since R_1 and R_2 are empty. If the value is *first* (*second*), the smaller halfsegment belongs to R_1 (R_2). If it is *both*, the first halfsegments of R_1 and R_2 are identical. The parameter *status* with possible values {*end_of_none*, *end_of_first*, *end_of_second*, *end_of_both*} describes the state of both halfsegment sequences. If the value of *status* is *end_of_none*, both objects still have half-

[1] For database-specific reasons that are out of the scope of this paper, the actual representation (see Section 6) uses for all three data structures an AVL-tree embedded into an array. The elements (points or halfsegments) are additionally linked in sequence order. With this representation, all functions except *insert* need $O(1)$ time and *insert* $O(\log n)$ time.

segments. If it is *end_of_first* (*end_of_second*), R_1 (R_2) is empty. If it is *end_of_both*, both object representations are empty.

Function *rr_select_next*(R_1, R_2, *object*, *status*) searches for the next smaller halfsegment of R_1 and R_2; parameters have the same meaning as for *rr_select_first*. Obviously, this is realized by *select_next* functions of the two objects.

Both functions together allow one to scan in linear time two object representations like one ordered sequence. Analogous functions can be defined for two *lines* objects (*ll_select_first*, *ll_select_next*) and a *lines* and a *regions* object (*lr_select_first*, *lr_select_next*). For the comparison of halfsegments with points, the dominating points of the halfsegments are used so that *points* and *lines* objects (*pl_select_first*, *pl_select_next*) and correspondingly *points* and *regions* objects (*pr_select_first*, *pr_select_next*) can be treated in a similar way.

To distinguish the functions on data types (written in italic) from the ROSE algebra operations on data types, the latter are written in bold face. In the sequel, we discuss example algorithms for three operations. The second and third operation are at the same time examples of operations whose algorithms only need O(n) time (n denotes the total size of the operands) in contrast to their Euclidean counterparts which require O(n log n) time. Their signatures are:

pr_on_border_of	:	*points* × *regions*	→	*bool*
ll_intersects	:	*lines* × *lines*	→	*bool*
ll_disjoint	:	*lines* × *lines*	→	*bool*

Operator **pr_on_border_of** determines whether all points of a *points* object lie on the faces' boundaries of a *regions* object. Hence the algorithm checks whether for each point p of a *points* object P (denoted by $p \in P(P) = \{p_1, ..., p_n\}$) a halfsegment h of a *regions* object R (denoted by $h \in H(R) = \{h_1, ..., h_n\}$) exists whose dominating point is equal to p.

algorithm pr_on_border_of
input: A *points* object P and a *regions* object R
output: *true*, if $\forall\, p \in P(P)\ \exists\, h \in H(R) : p = dp(h)$
    false, otherwise
begin
  pr_select_first(*P*, *R*, *object*, *status*);
  while (*object* ≠ *first*) **and** (*status* = *end_of_none*) **do**
    pr_select_next(*P*, *R*, *object*, *status*);
  end-while;
  return (*object* ≠ *first*) **and** (*status* ≠ *end_of_second*)
end pr_on_border_of.

The while-loop of the algorithm is executed as long as no point is found which is in P but not a dominating point of a halfsegment of R and as long as none of the object sequences is exceeded. For the predicate to be *true*, termination of the while-loop must not have occurred because a point was found which is not on the boundary of R (*object* ≠ *first*). This implies that termination is due to reaching the end of one or both sequences, and the predicate is *true* if this was not the *regions* sequence alone (*status* ≠ *end_of_second*).

Operator **ll_intersects** examines whether two *lines* objects L_1 and L_2 intersect. According to its definition in the ROSE algebra it yields *true* if both objects have no common (half)segments but at least one common point which is not a *meeting point* but an intersection point. Point p is a *meeting point* if the angularly sorted list of halfsegments of L_1 and L_2 with the same dominating point p can be subdivided into two sublists so that one list contains only halfsegments of L_1 and the other list only halfsegments of L_2. The idea is now to walk around p, scanning the segments, and to count the number of "object changes" in this ordered list of halfsegments of L_1 and L_2. Point p is a meeting point if this number is less than or equal to two; otherwise an intersection point has been found.

```
algorithm ll_intersects
input: Two lines objects L1 and L2
output: true,  if no common segment exists, but a common point which is not a meeting
               point
        false, otherwise
begin
  ll_select_first(L1, L2, object, status);
  if object = first then act_dp := dp(get_hs(L1))
  else if object = second then act_dp := dp(get_hs(L2))
  end-if;
  old_obj := object; found := false; count := 0;
  while (status = end_of_none) and (object ≠ both) do
    ll_select_next(L1, L2, object, status);
    if (status ≠ end_of_both) and (object ≠ both) and not found then
      if object = first then
        new_dp := dp(get_hs(L1))
      else if object = second then
        new_dp := dp(get_hs(L2))
      end-if;
      if new_dp ≠ act_dp then (* new point *)
        act_dp := new_dp;
        count := 0;
        old_obj := object;
      else if object ≠ old_obj then (* object switch *)
        count := count + 1;
        old_obj := object;
        found := found or (count > 2);
      end-if;
    end-if;
  end-while;
  return found and (object ≠ both);
end ll_intersects.
```

The while-loop of the algorithm terminates if either the end of one of the objects has been reached or a common halfsegment has been found. In the latter case the result value is *false* (*object* ≠ *both*), in the first case the decision is based on whether at least one intersection point has been found or not (*found*).

Operator **ll_disjoint** examines whether two *lines* objects L_1 and L_2 are disjoint. According to its definition in the ROSE algebra it is not sufficient only to test for common halfsegments because the operator yields also *false* if there are segments of both objects which have common points. Due to the realm properties such a common point can only be an end point of two segments of L_1 and L_2. Hence, there must be two halfsegments $h_1 \in H(L_1)$ and $h_2 \in H(L_2)$ with the same dominating point. Since in the halfsegment order all halfsegments with the same dominating point lie one behind the other, a parallel object traversal can check whether two consecutive halfsegments from different objects have the same dominating point. In such a case, L_1 and L_2 are not disjoint.

algorithm ll_disjoint
input: Two *lines* objects L_1 and L_2
output:*true*, if $\forall\ h_1 \in H(L_1)\ \forall\ h_2 \in H(L_2) : dp(h_1) \neq dp(h_2)$
false, otherwise
begin
  ll_select_first(L_1, L_2, *object*, *status*);
  if *object* = *first* **then** *act_dp* := *dp*(*get_hs*(L_1))
  else if *object* = *second* **then** *act_dp* := *dp*(*get_hs*(L_2))
  end-if;
  old_obj := *object*;
  found := *false*;
  while (*object* ≠ *both*) **and** (*status* = *end_of_none*) **and not** *found* **do**
    ll_select_next(L_1, L_2, *object*, *status*);
    if (*status* ≠ *end_of_both*) **and** (*object* ≠ *both*) **then**
      if *object* = *first* **then**
        new_dp := *dp*(*get_hs*(L_1))
      else
        new_dp := *dp*(*get_hs*(L_2))
      end-if;
      if *object* ≠ *old_obj* **then**
        found := (*new_dp* = *act_dp*);
      end-if;
      act_dp := *new_dp*;
      old_obj := *object*;
    end-if;
  end-while;
  return (*object* ≠ *both*) **and not** *found*;
end ll_disjoint.

The while-loop is executed as long as no common halfsegment has been found (*object* ≠ *both*), the end of both objects has not been reached (*status* = *end_of_none*) and no common point has been discovered (**not** *found*). Hence, the operator returns *true* if the end of at least one object has been reached and neither a common halfsegment nor a common point has been found.

Further 29 operators of the ROSE algebra can be realized either by simple or parallel traversal. These operators include tests whether two *points* objects are equal, unequal, or disjoint (**pp_equal**, **pp_unequal**, **pp_disjoint**), whether two *lines* or two *regions* objects are equal, unequal, or have common segments (**ll_equal**, **ll_unequal**, **ll_border_in_common**, **rr_equal**, **rr_unequal**, **rr_border_in_common**), whether two *lines* objects meet in common end points (**ll_meets**), whether a *points* objects lies completely on a *lines* object (**pl_on_border_of**), and whether a *lines* and a *regions* object have segments in common (**lr_border_in_common**, **rl_border_in_common**). Several operators yield spatial objects as results. The operators **pp_intersection**, **pp_plus**, **pp_minus**, **ll_intersection**, **ll_plus**, and **ll_minus** yield the intersection, union, and difference of two *points* objects and two *lines* objects, respectively. Some operators compute the common segments of two *lines* objects, two *regions* objects, or a *line* and a *region* object, resp. (**ll_common_border**, **rr_common_border**, **lr_common_border**, **rl_common_border**). The operators **l_vertices** and **r_vertices** return the end points of the boundary segments of a *lines* object and a *regions* object, respectively. The remaining operations yield numerical values. They calculate the number of single points of a *points* object (**p_no_of_components**), the length of a *lines* object (**l_length**), and the area and perimeter of a *regions* object (**r_area**, **r_perimeter**).

For all predicates and for operations returning numbers the worst case time complexity is $O(n)$, where n is the total number of points or halfsegments in the one or two operands. For operations returning new spatial objects the time bound is $O(n + k)$ where[2] k is the number of points or halfsegments in the result object. $O(n)$ time is needed for scanning the operands and $O(k)$ for constructing the result. Since $k = O(n)$, this is always bounded by $O(n)$. It is left to the reader to determine those operators based on the traversal paradigm whose algorithms need $O(n\ [+k])$ time in a realm-based but $O(n \log n\ [+k])$ time in Euclidean space.

5 Plane-Sweep Algorithms

A number of operators defined on the types *points*, *lines*, and *regions* are realized by a *plane-sweep* [GRS95, Sch97]. In this section, too, we will explain some example algorithms which are analysed with respect to their worst case time and space requirements. In the special case of realm-based (grid-based) computational geometry where no two segments intersect within their interiors, the event point schedule is static (because new event points cannot exist) and given by the ordered sequence of points or halfsegments of the operand objects. No further explicit event point structure is needed. Also, no initial sorting is necessary since the plane-sweep order of points and segments is the base representation of objects anyway.

If a left (right) halfsegment of a *regions* object is reached during a plane-sweep, its segment component is stored into (removed from) the segment sequence of the sweep line status sorted by the order relation *above*. A segment s lies *above* a segment t if the intersection of their x-intervals is not empty and if for each x of the intersection interval the y-coordinate of s is greater than the one of t (except possibly for a common end point where the y-coordinates are equal). Points and halfsegments of *lines* objects are used to query the sweep line status. A point p lies *above* a non-vertical segment t if the x-coordinate of p lies within the x-interval of t and the y-coordinate of p is greater than the y-coordinate of t at $p.x$. If t is vertical, the x-coordinates of p and t must be identical and the y-coordinate of p must be greater than all y-coordinates of t.

We define an auxiliary type *status* which reflects the sweep line status at each moment of a plane-sweep algorithm. The sweep line status is described by an ordered sequence of segments with respect to the order relation *above*. A logical pointer indicates the position within the sequence, and each segment has an attached set of attributes of some sort *attr*. Several functions are defined on type *status*. Function *new_sweep* creates a new sweep line status structure S. The functions *add_left* and *del_right* insert and remove a segment, respectively, from S. Functions *pred_of_s* and *pred_of_p* yield the segment lying directly below a given segment and point, respectively, in S. Function *current_exists* checks whether S contains at least one segment. Function *pred_exists* tests whether the segment whose position is currently indicated by the logical pointer has a predecessor. Function *get_attr* (*get_pred_attr*) retrieves the attribute of (the predecessor of) the segment currently indicated by the logical pointer. Function *set_attr* assigns an attribute to the segment currently indicated by the logical pointer. For the sweep line status an efficient internal dynamic structure like the AVL tree can be employed which realizes each of the operations *add_left*, *del_right*, *pred_of_s*, and *pred_of_p* in worst case time $O(\log n)$ and the other operations in constant time.

---

2 For an AVL-tree embedding in an array this time bound is $O(n + k \log k)$.

For all algorithms we assume that all those halfsegments of a *regions* object *R* have an associated attribute *InsideAbove* where the area of *R* lies above or left of its segments. This *segment classification* can be computed by a plane-sweep algorithm (not shown here) which views all segments intersecting the current sweep line from bottom to top. It is obvious that the lowest segment obtains the attribute *InsideAbove*, the following does not, the third again obtains it, etc. Whether the attribute *InsideAbove* is associated with a segment depends on the assignment of the attribute to the immediately preceding segment in the sweep line status.

We distinguish two classes of plane-sweep algorithms. The first class of plane-sweep algorithms considers the relationships between a *points* or *lines* object and a *regions* object. The algorithmic scheme is to insert only the segments of the *regions* object into the sweep line status and to use the elements of the *points* and *lines* object, respectively, as query elements. As an example, we show the algorithm for **rl_intersection** which has the signature

rl_intersection : $\underline{regions} \times \underline{lines} \quad \rightarrow \quad \underline{lines}$

The algorithm for **rl_intersection** produces a new *lines* object which contains all segments lying within *R*. It is crucial for the correctness of this algorithm that we can be sure that a complete (half)segment lies within *R*, if its dominating point lies within an area of *R*. This is because the boundary of *R* cannot intersect the interior of the segment due to the realm properties. This algorithm requires $O((l + m) \log m + k)$ where[3] k is the size of the result object and l and m denote the size of the *lines* and *regions* operand, respectively.

```
algorithm rl_intersection
input: A lines object L and a regions object R
output: A new lines object L_new containing all halfsegments of L whose segment
        components lie in R
begin
  L_new := new();
  S := new_sweep();
  lr_select_first(L, R, object, status);
  while status = end_of_none do
    if (object = both) or (object = second) then
      h := get_hs(R); (* Let h = (s, d). *)
      attr := get_attr(R);
      if d = left then
        S := add_left(S, s);
        if InsideAbove ∈ attr then
          S := set_attr(S, {InsideAbove});
        end-if
      else
        S := del_right(S, s);
      end-if
    end-if;
    if object = both then
      h := get_hs(L);
      L_new := insert(L_new, h);
    else if object = first then
      h := get_hs(L); (* Let h = (s, d). *)
```

[3] For an AVL-tree embedding in an array this time bound is $O((l + m) \log m + k \log k)$.

```
        S := pred_of_s(S, s);
        if current_exists(S) and (InsideAbove ∈ get_attr(S)) then
            L_new := insert(L_new, h);
        end-if;
    end-if;
    lr_select_next(L, R, object, status);
  end-while;
  return L_new;
end rl_intersection.
```

Further six operators belonging to this class check whether a *points* object lies inside a *regions* object (**pr_inside**) and whether a *lines* object intersects, has single common end points with, or lies inside a *regions* object (**lr_intersects/rl_intersects**, **lr_meets/rl_meets**, **lr_inside**). For all these operations of this class, the time complexity is $O((l + m) \log m)$ if m is the size of the *regions* operand and l the size of the other operand. Of course, for $n = l + m$, $O(n \log n)$ is a simpler upper bound for all operations.

The second class of plane-sweep algorithms considers the relationships between two *regions* objects. Note that here the immediate application of the technique introduced above is impeded by the fact that *regions* objects may have holes. Hence, for the algorithms of this class we introduce the concepts of *overlap numbers* and *segment classification*. A point of the realm grid obtains the *overlap number k* if it is covered by (or part of) *k regions* objects. For example, for two intersecting simple polygons the area outside of both polygons gets overlap number 0, the intersecting areas overlap number 2, and the other areas overlap number 1. Since a segment of a *regions* object separates space into two parts, an inner and an exterior one, each segment is associated with a pair (m/n) of overlap numbers, a *lower* (or right) one m and an *upper* (or left) one n. The lower (upper) overlap number indicates the number of overlapping *regions* objects below (above) the segment. In this way, we obtain a *segment classification* of a fixed set of *regions* objects and speak of (m/n)-segments.

For two *regions* objects (we only consider binary operators here) $m, n \leq 2$ holds; of the nine possible combinations only seven describe valid segment classes. This is because a (0/0)-segment contradicts the definition of a *regions* object, since then at least one of both *regions* objects would have two holes or an outer cycle and a hole with a common border. For similar reasons, (2/2)-segments cannot exist, since then at least one of the two *regions* objects would have a segment which is common to two outer cycles of the object. Hence, possible (m/n)-segments are (0/1)-, (0/2)-, (1/0)-, (1/1)-, (1/2)-, (2/0)-, and (2/1)-segments.

As an example for a plane-sweep algorithm of the second class we present the algorithm for **rr_inside** which has the signature

rr_inside : $regions \times regions \rightarrow bool$

and which tests whether a *regions* object R_1 is completely contained in a *regions* object R_2. This means that all segments of R_1 must lie within the area of R_2 but no segment (and hence no hole) of R_2 may lie within R_1. If we consider the objects R_1 and R_2 as halfsegment sequences together with the segment classes, the predicate **rr_inside** is true if (i) all halfsegments that are *only* element of R_1 have segment class (1/2) or (2/1), since only these segments lie within R_2, (ii) all halfsegments that are *only* element of R_2 have segment class (0/1) or (1/0), since these definitely do not lie within R_1, and (iii) all *common* halfsegments have segment class (0/2) or (2/0), since the areas of both objects lie on the same side of the halfsegment. In the case of a (1/1)-segment the areas would lie side by side so that R_1 could not be contained by R_2. In the algorithm, whenever a segment is inserted into the sweep line status, first the pair (m_p/n_p) of overlap numbers of the predecessor is determined (it is set to (*/0) if no predecessor exists). Then the overlap numbers (m_s/n_s) for this segment are computed. Obviously $m_s = n_p$ must hold; n_s is also initialized to n_p and then corrected.

```
algorithm rr_inside
input: Two regions objects R1 and R2
output: true, if R1 lies within R2
        false, otherwise
begin
  S := new_sweep();
  inside := true;
  rr_select_first(R1, R2, object, status);
  while (status ≠ end_of_first) and inside do
    if (object = first) or (object = both)
      then h := get_hs(R1); (* Let h = (s, d). *)
      else h := get_hs(R2); (* Let h = (s, d). *)
    end-if;
    if d = right then
      S := del_right(S, s);
    else
      S := add_left(S, s);
      if not pred_exists(S)
        then (mp/np) := (*/0)
        else {(mp/np)} := get_pred_attr(S)
      end-if;
      ms := np;
      ns := np;
      if ((object = first) or (object = both)) and
        (InsideAbove ∈ get_attr(R1))
        then ns := ns + 1
        else ns := ns – 1
      end-if;
      if ((object = second) or (object = both)) and
        (InsideAbove ∈ get_attr(R2))
        then ns := ns + 1
        else ns := ns – 1
      end-if;
      S := set_attr(S, {(ms/ns)});
      if object = first then inside := ((ms/ns) ∈ {(1/2), (2/1)})
      else if object = second then inside := ((ms/ns) ∈ {(0/1), (1/0)})
      else inside := ((ms/ns) ∈ {(0/2), (2/0)})
      end-if;
    end-if;
    rr_select_next(R1, R2, object, status);
  end-while;
  return inside;
end rr_inside.
```

If R_1 has l and R_2 m halfsegments, the while-loop is executed at most $n = l + m$ times, since each time a new halfsegment is visited. The most expensive operations within the loop are the insertion and the removal of a segment into and from the sweep line status. Since at most n elements can be contained in the sweep line status, the worst case time complexity of the algorithm is $O(n \log n)$ which is also valid for all other operations of this class.

Further operators belonging to this second class check whether two *regions* objects are disjoint with respect to area (**rr_area_disjoint**), additionally have no segments together (**rr_edge_disjoint**), and also have no common end points together (**rr_disjoint**), whether one *regions* object R_1 is inside another *regions* object R_2 and has no common segments with R_2 (**rr_edge_inside**) and also no common end points with R_2 (**rr_vertex_inside**), whether two *regions* objects intersect (**rr_intersects**), are adjacent (**rr_adjacent**) or have common end points (**rr_meets**), and whether a *regions* object is contained in another one (**rr_encloses**). Moreover, operators computing *regions* objects as the result of intersection, union, and difference of two *regions* objects belong to this class (**rr_intersection**, **rr_plus**, **rr_minus**).

6 Conclusions

This paper has demonstrated that it is feasible to design numerically robust, topologically correct, and efficient geometric algorithms on a finite discrete domain, i.e., on a uniform grid. The use of high-level primitives has made it possible to describe a large number of algorithms in compact, precise notation. Due to a large amount of operators the emphasis has not been to design a special solution for each single geometric problem but to treat a spatial algebra as a whole and in a uniform way. This view implies the development of universal data structures and the employment of general algorithmic paradigms (e.g., simple and parallel traversal, plane-sweep) that are well appropriate for a large number of geometric problems. As an important result, a comparison with the algorithms of classical computational geometry reveals that realm-based geometric algorithms are much simpler and more efficient than their Euclidean counterparts.

The realm-based (grid-based) data structures and algorithms have been implemented within a collection of software modules called the *ROSE system* [Rid95] which was part of a diploma thesis written by Thomas de Ridder. The source code written in Modula-2 is available to the interested reader by the author of this paper.

References

[DS90] D. Dobkin & D. Silver. Applied Computational Geometry: Towards Robust Solutions of Basic Problems. *Journal of Computer and System Sciences*, 40, 70-87, 1990.

[EM88] H. Edelsbrunner & E.P. Mücke. Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms. *ACM Symp. on Computational Geometry*, 118-133, 1988.

[For85] A.R. Forrest. Computational Geometry in Practice. *Fundamental Algorithms for Computer Graphics*, Springer Verlag, 707-723, 1985.

[GM95] L.J. Guibas & D.H. Marimont. Rounding Arrangements Dynamically. *11th Annual Symp. on Computational Geometry*, 190-199, 1995.

[GRS95] R.H. Güting, T. de Ridder & M. Schneider. Implementation of the ROSE Algebra: Efficient Algorithms for Realm-Based Spatial Data Types. *4th Int. Symp. on Advances in Spatial Databases* (*SSD '95*), LNCS 951, 216-239, 1995.

[GS93] R.H. Güting & M. Schneider. Realms: A Foundation for Spatial Data Types in Database Systems. *3rd Int. Symp. on Advances in Spatial Databases*, Springer Verlag, LNCS 692, 14-35, 1993.

[GS95] R.H. Güting & M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4, 100-143, 1995.

[GSS89] L. Guibas, D. Salesin & J. Stolfi. Epsilon-Geometry: Building Robust Algorithms from Imprecise Computations. *SIAM Conf. on Geometric Design*, 208-217, 1989.

[GY86] D. Greene & F. Yao. Finite-Resolution Computational Geometry. *27th IEEE Symp. on Foundations of Computer Science*, 143-152, 1986.

[HHK88] C.M. Hoffmann, J.E. Hopcroft & M.S. Karasick. Towards Implementing Robust Geometric Computations. *ACM Symp. on Computational Geometry*, 106-117, 1988.

[Hof89] C.M. Hoffmann. The Problems of Accuracy and Robustness in Geometric Computation. *Computer*, 22:3, 31-42, 1989.

[KK81] J.M. Keil & D.G. Kirkpatrick. Computational Geometry on Integer Grids. *19th Annual Allerton Conf. on Communication, Control, and Computing*, 41-50, 1981.

[KM83] P. Kornerup & D.W. Matula. Finite Precision Rational Arithmetic: An Arithmetic Unit. *IEEE Transactions on Computers*, C-32, 378-388, 1983.

[KM85] R.G. Karlsson & J.I. Munro. Proximity on a Grid. *2nd Symp. on Theoretical Aspects of Computer Science*, Springer-Verlag, LNCS 182, 187-196, 1985.

[KO88a] R.G. Karlsson & M.H. Overmars. Scanline Algorithms on a Grid. *BIT*, 28, 227-241, 1988.

[KO88b] R.G. Karlsson & M.H. Overmars. Normalized Divide-and-Conquer: A Scaling Technique for Solving Multi-Dimensional Problems. *Information Processing Letters*, 26, 307-312, 1988.

[Mil89] V. Milenkovic. Double Precision Geometry: A General Technique for Calculating Line and Segment Intersections Using Rounded Arithmetic. *30th Annual Symp. on Foundations of Computer Science*, 500-505, 1989.

[MK84] S.P. Mudur & P.A. Koparkar. Interval Methods for Processing Geometric Objects. *IEEE Computer Graphics & Applications*, 4:2, 7-17, 1984.

[Mül85] H. Müller. Rastered Point Location. *Proc. Workshop on Graphtheoretic Concepts in Computer Science*, Trauner Verlag, 281-293, 1985.

[NME90] G. Nagy, M. Mukherjee & D.W. Embley. Making Do with Finite Numerical Precision in Spatial Data Structures. *4th Int. Symp. on Spatial Data Handling*, 55-65, 1990.

[OTU87] T. Ottmann, G. Thiemt & C. Ullrich. Numerical Stability of Geometric Algorithms. *3rd ACM Symp. on Computational Geometry*, 119-125, 1987.

[Ove88a] M.H. Overmars. New Algorithms for Computer Graphics. *Advances in Computer Graphics*, Eurographics Seminars, Springer-Verlag, 3-19, 1988.

[Ove88b] M.H. Overmars. Computational Geometry on a Grid: an Overview. *Theoretical Foundations for Computer Graphics and CAD*, Springer-Verlag, 167-184, 1988.

[Ove88c] M.H. Overmars. Efficient Data Structures for Range Searching on a Grid. *Journal of Algorithms*, vol. 9, 254-275, 1988.

[Rid95] T. de Ridder. *The ROSE System*. Modula-2 Program System (Source Code). FernUniversität Hagen, Praktische Informatik IV, Software Report 1, 1995. Available as a LaTeX file for printing and/or as a compressed collection of ASCII files.

[Sch94] P. Schorn. Degeneracy in Geometric Computation and the Perturbation Approach. *The Computer Journal*, 37:1, 1994.

[Sch97] M. Schneider. Spatial Data Types for Database Systems - Finite Resolution Geometry for Geographic Information Systems. LNCS 1288, Springer-Verlag, 1997.

[SI88] K. Sugihara & M. Iri: Geometric Algorithms in Finite-Precision Arithmetic. Research Memorandum RMI 88-10, Department of Mathematical Engineering and Information Physics, University of Tokyo, 1988.

[Yao92] F.F. Yao. Computational Geometry. Algorithms and Complexity, *Handbook of Theoretical Computer Science*, vol. A, Elsevier Science Publishers B.V., 343-389, 1992.